



SAPIENZA
UNIVERSITÀ DI ROMA

Algorithms

Volume 5 — Recursion

Recursion: classical recursive algorithms, recurrence relations, the Master Theorem, recursive algorithms (array sum, merge sort, binary search, maximum sub-array) and the comparison between recursive and iterative versions.

Fabio Furini

fabiofurini.github.io

Department of Computer, Control and Management Engineering — Sapienza University of Rome

Online version and downloadable material:

fabiofurini.github.io/algorithms/
github.com/fabiofurini/algorithms

Academic year 2026–2027

Contents

14	Recursion	3
14.1	Recursive algorithms	4
14.2	Tree of recursive calls and call stack	6
14.3	Correctness of recursive algorithms	8
14.4	Factorial	10
14.5	Fibonacci numbers	14
14.6	Greatest common divisor	19
14.7	Fast exponentiation	23
14.8	Towers of Hanoi	27
14.9	Summary	32
15	Recurrences	33
	Introduction	33
15.1	Recurrences	33
15.2	The expansion method	40
15.3	The substitution method	43
15.4	The recursion-tree method	48
15.5	Towards the Master Theorem	54
15.6	Summary: which method when	55
16	The Master Theorem	56
	Introduction	56
16.1	Recurrences in standard form	56
16.2	The Master Theorem	57
16.3	Applying the Master Theorem	60
16.4	When the Master Theorem does not apply	64
16.5	Proof of the Master Theorem	66
16.6	Summary	76
17	Recursive algorithms	77
	Introduction	77
17.1	The sum of the elements of an array	78
17.2	Merge Sort	88



17.3 Recursive Binary Search	103
17.4 The maximum subarray problem	114
17.5 Summary	130
18 Recursive versus iterative algorithms	131
Introduction	131
18.1 Recursion and iteration	132
18.2 Comparison at a glance	134
18.3 Divide and conquer is what gives the gain, not recursion itself	136
18.4 When subproblems overlap	141
18.5 Space	144
18.6 C++ implementation	149
18.7 Experiment	151
18.8 Guidelines	154

Recursion

In the previous chapters every algorithm was **iterative**: a sequence of instructions and loops, whose number of steps we counted line by line (chapter «Running time») and whose correctness we proved with loop invariants (chapter «Correctness»). In this chapter we study a different way of designing algorithms: an algorithm may **call itself** on smaller instances of the same problem.

The plan of the chapter is the following.

- In Section 14.1 we define recursive algorithms, base cases and recursive steps, and we see why a recursive algorithm terminates.
- In Section 14.2 we describe how a recursive algorithm is executed: the **tree of recursive calls** (which determines the number of steps) and the **call stack** (which determines the memory).
- In Section 14.3 we show that the correctness of a recursive algorithm is proved by **strong induction**.
- In Sections 14.4–14.8 we apply these tools to five classical recursive algorithms: the factorial, the Fibonacci numbers, the greatest common divisor, fast exponentiation and the Towers of Hanoi. For each of them we prove the correctness, we count the number of steps by unrolling a recurrence, and we give a C++ implementation executed step by step.

14.1 Recursive algorithms

- Many problems have a natural **self-similar** structure: the solution of an instance can be easily obtained from the solution of one or more **smaller instances of the same problem**.
- For example, for every natural number $n \geq 1$,

$$n! = n \cdot (n - 1)!$$

that is, the factorial of n is obtained from the factorial of the smaller number $n - 1$.

- Algorithms that exploit this structure are called recursive algorithms.

Definition 14.1: recursive algorithm

A **recursive algorithm** is an algorithm that solves an instance of a problem by **calling itself** on one or more **smaller instances** of the same problem, and then combining the results of these calls.

- Every recursive algorithm is made of two parts:
 1. **Base case(s)**: one or more instances that are small enough to be solved directly, without any recursive call (for example, $0! = 1$).
 2. **Recursive step**: on every other instance, the algorithm calls itself on smaller instances and combines the returned results (for example, $n! = n \cdot (n - 1)!$).
- Each execution of the algorithm on a given input is called a **call** (or invocation). A call that is made from inside another call is a **recursive call**.
- To measure how “small” an instance is, we associate to each input a **size**, which is a natural number (for example the value n of the input number, or the number n of entries of an array $a[1..n]$).

14.1.1 Termination

- A recursive algorithm terminates only if the chain of recursive calls eventually reaches a base case.
- If the size of the instance strictly decreases at each recursive call, we obtain a strictly decreasing sequence of natural numbers

$$n_0 > n_1 > n_2 > \dots$$

- There is no infinite strictly decreasing sequence of natural numbers (by the well-ordering property of $\mathbb{N}$): after at most n_0 calls the size cannot decrease anymore. Therefore, if every instance which is not a base case is reduced to strictly smaller instances, a base case is necessarily reached after finitely many calls.

Key idea of recursion.

- Solve the **base cases** directly.
- Reduce every other instance to one or more **strictly smaller** instances of the same problem, solve them by recursive calls, and combine their results.
- Since the size is a natural number that **strictly decreases** at every recursive call, a base case is always reached and the algorithm **terminates**.

Example 14.1: What can go wrong

- **Missing base case.** Consider the function `WrongFactorial( $n$ )` that returns $n \cdot \text{WrongFactorial}(n - 1)$, without any base case. The call `WrongFactorial(2)` calls `WrongFactorial(1)`, which calls `WrongFactorial(0)`, which calls `WrongFactorial(-1)`, and so on forever: the recursion never stops.
 - **Size not decreasing.** A function that, on input n , calls itself on input n (or $n + 1$) never reaches a base case $n = 0$ when $n \geq 1$.
 - In practice, an infinite recursion is stopped by the computer when the memory reserved to the calls is exhausted (a **stack overflow** error).
- To count the number of steps of a recursive algorithm we use the same conventions as in the chapter «Running time»: each execution of a line of pseudo-code is **one step**. A line that contains a recursive call counts as one step **plus** all the steps executed by that recursive call.
 - As a consequence, the number of steps $S(n)$ of a recursive algorithm is naturally expressed in terms of the number of steps on smaller instances, i.e., by a **recurrence**, such as

$$S(n) = S(n - 1) + 2.$$

In this chapter we solve the recurrences of the step counts by **simple unrolling** (also called **expansion**): we replace repeatedly $S(\cdot)$ by its definition until we reach a base case.

14.2 Tree of recursive calls and call stack

14.2.1 Tree of recursive calls

Definition 14.2: tree of recursive calls

The **tree of recursive calls** of a recursive algorithm on a given input is the rooted tree in which:

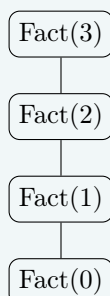
- every node is a call of the algorithm;
- the root is the initial call;
- the children of a node are the recursive calls made directly by that call, from left to right in the order in which they are executed.

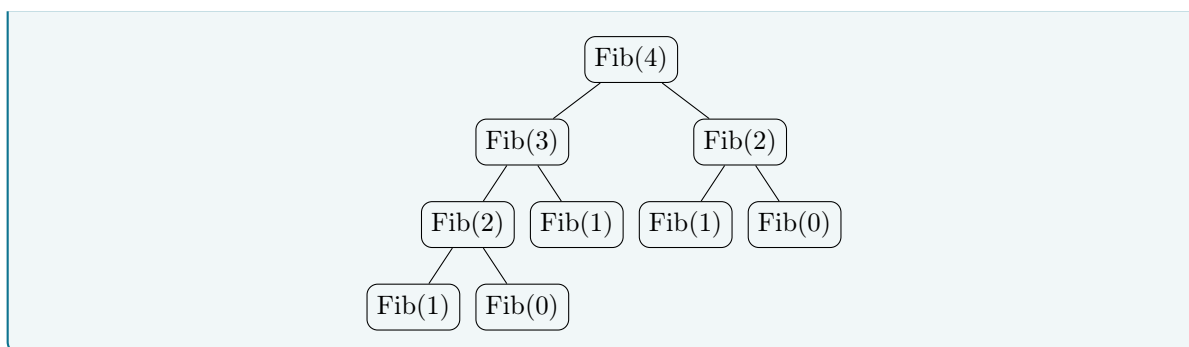
The leaves of the tree are the calls that do not make recursive calls (typically the base cases).

- The total number of calls executed by the algorithm is the **number of nodes** of the tree.
- If every call executes a constant number of steps (excluding the steps of its recursive calls), then the number of steps of the algorithm is proportional to the number of nodes of the tree.
- The calls are executed following a **depth-first** visit of the tree: a call starts, executes its first recursive call (and the whole subtree below it) completely, then its second recursive call, and so on; it returns its result only when all its recursive calls have returned.

Example 14.2: Two shapes of trees of recursive calls

- If each call makes **one** recursive call (as for $n! = n \cdot (n-1)!$), the tree is a **path** (left).
- If each call makes **two** recursive calls (as for the Fibonacci numbers, defined by $F(n) = F(n-1) + F(n-2)$ with $F(0) = 0$ and $F(1) = 1$), the tree **branches** and the number of nodes can grow very quickly (right).





14.2.2 Call stack

- When a call starts, the computer reserves a piece of memory, called a **stack frame** (or activation record), that contains the parameters of the call, its local variables, and the point of the program where the execution must continue when the call returns.
- The frames are stored in a memory area organised as a **stack** (the **call stack**):
 - when a call starts, its frame is **pushed** on top of the stack;
 - when a call returns, its frame is **popped** from the top of the stack, and the execution continues in the call whose frame is now on top.
- At any moment, the frames in the stack are exactly the calls on the path from the root of the tree of recursive calls to the call currently executed.

Definition 14.3: depth of the recursion

The **depth of the recursion** of a recursive algorithm on a given input is the maximum number of calls that are simultaneously active, i.e., the maximum number of nodes on a path from the root to a leaf of the tree of recursive calls.

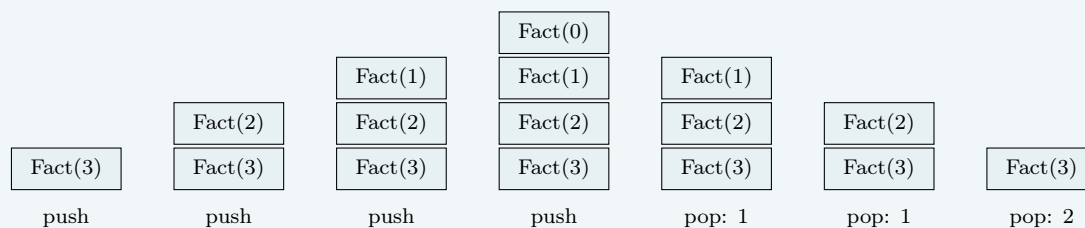
The maximum number of stack frames simultaneously in memory is equal to the **depth of the recursion**. If every frame uses a constant amount of memory, the memory used by the call stack is

$$\Theta(\text{depth of the recursion}).$$

- Note the difference: the **number of steps** depends on the **number of nodes** of the tree, while the **memory** used by the call stack depends only on its **depth**.
- For the tree of **Fib(4)** above, there are 9 calls, but at most 4 of them are simultaneously in the stack (for example **Fib(4)**, **Fib(3)**, **Fib(2)**, **Fib(1)**).

Example 14.3: Evolution of the call stack for Fact(3)

The figure shows the call stack (top of the stack is up) after each push and after each pop, during the computation of $3!$ with the recursion $n! = n \cdot (n-1)!$, $0! = 1$.



- Four frames are pushed: **Fact(3)**, **Fact(2)**, **Fact(1)**, **Fact(0)**.
- **Fact(0)** is a base case: it returns 1 and its frame is popped. Then **Fact(1)** returns $1 \cdot 1 = 1$, **Fact(2)** returns $2 \cdot 1 = 2$ and, finally, **Fact(3)** returns $3 \cdot 2 = 6$ and the stack becomes empty.
- The depth of the recursion is 4.

- The size of the call stack is limited (typically a few megabytes). A very deep recursion (for example, millions of nested calls) causes a **stack overflow**, even if the algorithm is correct and terminates.

14.3 Correctness of recursive algorithms

- The natural tool to prove the correctness of a recursive algorithm is **mathematical induction** on the size of the input.
- Since a recursive call can be made on any smaller instance (not only on the instance of size $n-1$), we use **strong induction**.

Strong induction. Let $P(n)$ be a statement depending on $n \in \mathbb{N}$, and let $n_0 \in \mathbb{N}$. If

1. **base case:** $P(n_0)$ is true, and
2. **inductive step:** for every $n > n_0$, if $P(k)$ is true for every k with $n_0 \leq k < n$ (**inductive hypothesis**), then $P(n)$ is true,

then $P(n)$ is true for every natural number $n \geq n_0$.

- It is sometimes convenient to prove more than one base case (for example $n = 0$ and $n = 1$), and then the inductive step only for n larger than all of them.
- The structure of a recursive algorithm mirrors the structure of an induction proof:
 - the **base cases** of the algorithm correspond to the **base cases** of the induction;
 - in the **recursive step**, we may assume (inductive hypothesis) that the recursive calls, which are made on smaller inputs, return correct results, and we only need to show that the results are **combined** correctly.

Observation 14.4: correctness of recursive algorithms

Consider a recursive algorithm in which every input has a size in $\mathbb{N}$. Assume that:

1. on every input which is a base case, the algorithm returns the correct output;
2. on every other input of size n , the algorithm makes a finite number of recursive calls, all on inputs of size strictly smaller than n , and returns the correct output whenever all these recursive calls return the correct output.

Then the algorithm terminates and returns the correct output on every input.

Proof. Let $P(n)$ be the statement: “on every input of size n the algorithm terminates and returns the correct output”. We prove $P(n)$ for every $n \in \mathbb{N}$ by strong induction.

Let $n \in \mathbb{N}$ and assume that $P(k)$ is true for every $k < n$ (for $n = 0$ this assumption is empty). Consider an input of size n .

- If the input is a base case, then the algorithm terminates and returns the correct output by hypothesis 1.
- Otherwise, by hypothesis 2, the algorithm makes a finite number of recursive calls on inputs of size $k < n$. By the inductive hypothesis, each of these calls terminates and returns the correct output. Hence the algorithm terminates and, by hypothesis 2, it returns the correct output.

Therefore $P(n)$ is true, and by strong induction $P(n)$ is true for every $n \in \mathbb{N}$. $\square$

14.3.1 Link with loop invariants

- In the chapter «Correctness» we proved the correctness of iterative algorithms by means of **loop invariants**, which also have the structure of an induction proof.
- The three properties of a loop invariant correspond to the parts of the proof by induction of a recursive algorithm:

Loop invariant (iterative algorithm)	Induction (recursive algorithm)
Initialization: true before the first iteration	Base case(s) of the recursion
Maintenance: preserved by each iteration	Inductive step: correct combination of the results
Termination: the loop stops	The size strictly decreases towards a base case

- The main difference is the direction: an iterative algorithm builds the solution “from the bottom” (from the initialization towards the final iteration), while a recursive algorithm starts “from the top” (the original instance) and reduces it to smaller instances until a base case is reached.

14.4 Factorial

Given a natural number n , we consider the problem of computing the **factorial** of n :

$$n! = 1 \cdot 2 \cdot \dots \cdot n \quad \text{with } 0! = 1.$$

- Since $n! = n \cdot (n-1)!$ for every $n \geq 1$, the factorial has the following recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

- This is the pseudo-code of the recursive algorithm **Factorial**:

Algorithm: Factorial

Input : A natural number n

Output: The factorial $n!$

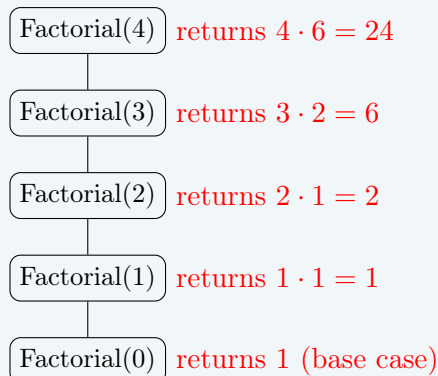
```

1 function Factorial( $n$ )
2   if  $n = 0$  then
3     return 1                                // base case
4   end
5   return  $n \cdot \text{Factorial}(n-1)$            // recursive step
6 end
```

- Explanation of the algorithm:
 - The base case is $n = 0$, for which the algorithm returns $0! = 1$.
 - For $n \geq 1$, the algorithm calls itself on the smaller input $n - 1$ and multiplies the result by n .
 - The size of the input is the value n , which decreases by one at each recursive call. Hence the base case is reached after n recursive calls.

Example 14.4: Tree of recursive calls of Factorial(4)

The tree is a path with 5 nodes. The value returned by each call is written in red next to the call.



The calls are made from the top to the bottom; the results are returned from the bottom to the top. The depth of the recursion is 5.

14.4.1 Correctness

Observation 14.5: correctness of Factorial

For every natural number n , the call **Factorial**(n) terminates and returns $n!$.

Proof. We proceed by induction on n .

- **Base case** ($n = 0$): the condition of line 2 is true and the algorithm returns $1 = 0!$ at line 3.
- **Inductive step** ($n \geq 1$): assume that **Factorial**($n - 1$) terminates and returns $(n - 1)!$ (inductive hypothesis). Since $n \neq 0$, the algorithm executes line 5 and returns

$$n \cdot \text{Factorial}(n - 1) = n \cdot (n - 1)! = n!.$$

Therefore **Factorial**(n) returns $n!$ for every $n \in \mathbb{N}$. □

14.4.2 Number of steps

- Let $S(n)$ denote the number of steps executed by **Factorial** on the input n (there is only one input for each value of n , so this is also the worst case).
- If $n = 0$, lines 2 and 3 are executed once each:

$$S(0) = 2.$$

- If $n \geq 1$, lines 2 and 5 are executed once each, and line 5 also executes all the steps of the call **Factorial**($n - 1$):

$$S(n) = S(n - 1) + 2.$$

- We solve the recurrence by unrolling it:

$$\begin{aligned}
 S(n) &= S(n-1) + 2 \\
 &= (S(n-2) + 2) + 2 = S(n-2) + 2 \cdot 2 \\
 &= (S(n-3) + 2) + 2 \cdot 2 = S(n-3) + 3 \cdot 2 \\
 &\vdots \\
 &= S(n-k) + k \cdot 2 \quad \forall k \in \{1, 2, \dots, n\}.
 \end{aligned}$$

Choosing $k = n$ (base case), we obtain

$$S(n) = S(0) + 2n = 2n + 2.$$

- Therefore,

$$S(n) = 2n + 2 = \Theta(n),$$

since

$$\lim_{n \rightarrow \infty} \frac{2n+2}{n} = \lim_{n \rightarrow \infty} \left(2 + \frac{2}{n}\right) = 2 \in \mathbb{R}_{>0}.$$

- The depth of the recursion is $n + 1$ (the calls with inputs $n, n-1, \dots, 0$ are all simultaneously in the stack), so the call stack uses $\Theta(n)$ memory. The iterative algorithm that computes $n!$ with a **while** loop uses only a constant amount of memory.
- We now express the number of steps as a function of the input size $|I|$. As in the chapter «Running time», we count each multiplication as one step. The input is the natural number n , which requires $\lfloor \log_2 n \rfloor + 1$ bits (for $n \geq 1$), hence

$$|I| = \Theta(\log_2 n).$$

This means that there exist constants $c_1, c_2 > 0$ such that, for all sufficiently large n ,

$$c_1 \log_2 n \leq |I| \leq c_2 \log_2 n \quad \Longleftrightarrow \quad 2^{\frac{1}{c_2}|I|} \leq n \leq 2^{\frac{1}{c_1}|I|}.$$

Since $S(n) = \Theta(n)$, there exist constants $a_1, a_2 > 0$ such that $a_1 n \leq S(n) \leq a_2 n$ for all sufficiently large n , and therefore

$$a_1 2^{\frac{1}{c_2}|I|} \leq S(n) \leq a_2 2^{\frac{1}{c_1}|I|},$$

that is, $T(|I|) = 2^{\Theta(|I|)}$.

The number of steps of the **Factorial** algorithm is $S(n) = 2n + 2 = \Theta(n)$, and its depth of recursion is $n + 1$. As a function of the input size $|I| = \Theta(\log_2 n)$, the worst-case running time is

$$T(|I|) = 2^{\Theta(|I|)},$$

i.e., it is linear in the numerical value n , but exponential in the input size $|I|$.

14.4.3 C++ implementation

- This is a C++ implementation of the Factorial algorithm. We use the type `unsigned long long` (64 bits): the result is correct only for $n \leq 20$, since $21! > 2^{64} - 1$.

```

1 // Returns n! for n >= 0 (correct for 0 <= n <= 20 with 64-bit integers)
2 unsigned long long Factorial(int n)
3 {
4     if (n == 0)
5     {
6         return 1;                // base case
7     }
8     return n * Factorial(n - 1); // recursive step
9 }

```

Example 14.5: execution of the C++ implementation of Factorial for $n = 4$

Calls (the stack grows)

1. Factorial(4): $4 \neq 0 \Rightarrow$ it must compute $4 \cdot \text{Factorial}(3)$ and waits. stack: [4]
2. Factorial(3): $3 \neq 0 \Rightarrow$ it must compute $3 \cdot \text{Factorial}(2)$ and waits. stack: [4, 3]
3. Factorial(2): $2 \neq 0 \Rightarrow$ it must compute $2 \cdot \text{Factorial}(1)$ and waits. stack: [4, 3, 2]
4. Factorial(1): $1 \neq 0 \Rightarrow$ it must compute $1 \cdot \text{Factorial}(0)$ and waits. stack: [4, 3, 2, 1]
5. Factorial(0): $n = 0 \Rightarrow$ base case. stack: [4, 3, 2, 1, 0]

Returns (the stack shrinks)

1. Factorial(0) returns 1. stack: [4, 3, 2, 1]
2. Factorial(1) returns $1 \cdot 1 = 1$. stack: [4, 3, 2]
3. Factorial(2) returns $2 \cdot 1 = 2$. stack: [4, 3]
4. Factorial(3) returns $3 \cdot 2 = 6$. stack: [4]
5. Factorial(4) returns $4 \cdot 6 = 24$. stack: []

The algorithm terminates with result $24 = 4!$, after 5 calls.

14.5 Fibonacci numbers

Given a natural number n , we consider the problem of computing the n -th **Fibonacci number** $F(n)$, defined as

$$F(n) = \begin{cases} n & \text{if } n \in \{0, 1\} \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

- The first Fibonacci numbers are:

n	0	1	2	3	4	5	6	7	8	9	10
$F(n)$	0	1	1	2	3	5	8	13	21	34	55

- The definition is itself recursive, and it directly gives the following (naive) recursive algorithm `Fib`, with **two** base cases and **two** recursive calls:

Algorithm: Fib

Input : A natural number n

Output: The Fibonacci number $F(n)$

```

1 function Fib( $n$ )
2   if  $n \leq 1$  then
3     return  $n$                                 // base cases
4   end
5   return Fib( $n-1$ ) + Fib( $n-2$ )                // recursive step
6 end
```

Observation 14.6: correctness of Fib

For every natural number n , the call `Fib( $n$ )` terminates and returns $F(n)$.

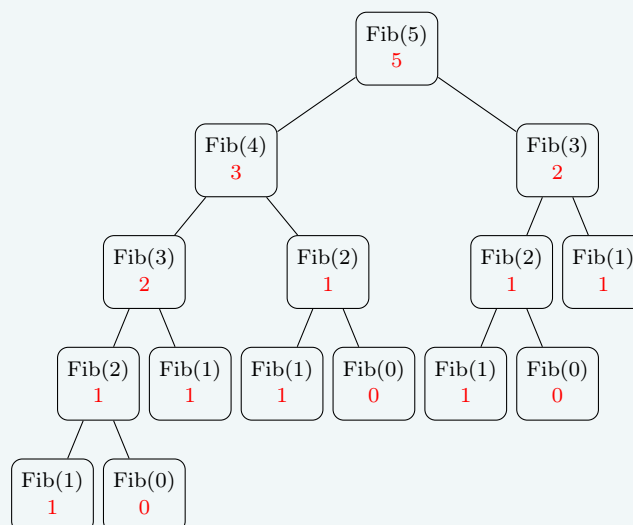
Proof. We proceed by strong induction on n .

- Base cases** ($n \in \{0, 1\}$): the algorithm returns $n = F(n)$ at line 3.
- Inductive step** ($n \geq 2$): by the inductive hypothesis the two calls on the smaller inputs $n-1$ and $n-2$ return $F(n-1)$ and $F(n-2)$, and line 5 returns their sum $F(n)$.

Here two base cases are needed, because the inductive step for n uses both $n-1$ and $n-2$. □

Example 14.6: Tree of recursive calls of Fib(5)

Each node is a call; the value returned by the call is written in red below it.



- The tree has 15 nodes: the algorithm makes 15 calls to compute $F(5) = 5$.
- The same subproblems are solved many times: **Fib(3)** is computed 2 times, **Fib(2)** 3 times, **Fib(1)** 5 times and **Fib(0)** 3 times.
- The depth of the recursion is 5 (for example, the path **Fib(5)**, **Fib(4)**, **Fib(3)**, **Fib(2)**, **Fib(1)**).

14.5.1 Number of calls and number of steps

- Let $C(n)$ denote the number of calls (nodes of the tree) executed by **Fib**(n). Each call with $n \geq 2$ generates one call on $n - 1$ and one call on $n - 2$, hence

$$C(n) = \begin{cases} 1 & \text{if } n \in \{0, 1\} \\ C(n-1) + C(n-2) + 1 & \text{if } n \geq 2 \end{cases}$$

- The first values are $C(0) = 1$, $C(1) = 1$, $C(2) = 3$, $C(3) = 5$, $C(4) = 9$, $C(5) = 15$, $C(6) = 25$. They are related to the Fibonacci numbers themselves:

Observation 14.7: number of calls of Fib

For every natural number n ,

$$C(n) = 2F(n+1) - 1.$$

Proof. We proceed by strong induction on n .

- **Base cases:** $C(0) = 1 = 2 \cdot 1 - 1 = 2F(1) - 1$ and $C(1) = 1 = 2 \cdot 1 - 1 = 2F(2) - 1$.

- **Inductive step** ($n \geq 2$): assume $C(k) = 2F(k+1) - 1$ for every $k < n$. Then

$$\begin{aligned} C(n) &= C(n-1) + C(n-2) + 1 \\ &= (2F(n) - 1) + (2F(n-1) - 1) + 1 \\ &= 2(F(n) + F(n-1)) - 1 = 2F(n+1) - 1. \end{aligned}$$

□

- For example, $C(5) = 2F(6) - 1 = 2 \cdot 8 - 1 = 15$, as in Example 14.6.
- Each call executes exactly 2 lines (lines 2–3 for a base case, lines 2 and 5 otherwise), hence the number of steps $S(n)$ satisfies

$$S(n) = S(n-1) + S(n-2) + 2 \quad (n \geq 2), \quad S(0) = S(1) = 2,$$

and therefore $S(n) = 2C(n) = 4F(n+1) - 2$.

- The number of calls grows **exponentially** in n :

Observation 14.8: exponential number of calls of Fib

For every natural number $n \geq 2$,

$$2^{n/2} \leq C(n) \leq 2^n.$$

Proof. Lower bound. We proceed by strong induction on $n \geq 2$.

- **Base cases:** $C(2) = 3 \geq 2 = 2^{2/2}$ and $C(3) = 5 \geq 2\sqrt{2} = 2^{3/2}$.
- **Inductive step** ($n \geq 4$): assume the claim for every k with $2 \leq k < n$; in particular for $k = n-2 \geq 2$. Since $C(n-1) \geq C(n-2)$ (the sequence C is non-decreasing),

$$C(n) = C(n-1) + C(n-2) + 1 > 2C(n-2) \geq 2 \cdot 2^{(n-2)/2} = 2^{n/2}.$$

Upper bound. Also by strong induction: $C(2) = 3 \leq 4$, $C(3) = 5 \leq 8$ and, for $n \geq 4$,

$$C(n) = C(n-1) + C(n-2) + 1 \leq 2^{n-1} + 2^{n-2} + 2^{n-2} = 2^n,$$

where we used $1 \leq 2^{n-2}$. □

- Note that $C(1) = 1 < 2^{1/2}$: this is why the lower bound is stated for $n \geq 2$.
- Therefore $S(n) = 2C(n) = 2^{\Theta(n)}$: the naive recursive algorithm is **exponential** in n . For example, the computation of $F(30) = 832040$ requires $C(30) = 2F(31) - 1 = 2 \cdot 1346269 - 1 = 2692537$ calls.
- (It can be shown that $F(n)$ grows as $\varphi^n / \sqrt{5}$, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio; hence $C(n) = \Theta(\varphi^n)$.)

14.5.2 An iterative algorithm

- The reason of the exponential number of calls is that the naive algorithm **solves the same subproblems again and again** (see Example 14.6). An iterative algorithm computes each Fibonacci number only once, from $F(0)$ and $F(1)$ upwards, keeping only the last two values:

Algorithm: IterativeFib

Input : A natural number n

Output: The Fibonacci number $F(n)$

```

1 function IterativeFib( $n$ )
2   if  $n \leq 1$  then
3     return  $n$ 
4   end
5    $prev := 0$ 
6    $curr := 1$ 
7    $i := 2$ 
8   while  $i \leq n$  do
9      $next := prev + curr$ 
10     $prev \leftarrow curr$ 
11     $curr \leftarrow next$ 
12     $i \leftarrow i + 1$ 
13  end
14  return  $curr$ 
15 end

```

- For $n \geq 2$: lines 2, 5, 6, 7, 14 are executed once, line 8 is executed n times and lines 9–12 are executed $n - 1$ times. Hence

$$S(n) = 5 + n + 4(n - 1) = 5n + 1 = \Theta(n).$$

- (Loop invariant: before each test of line 8, $prev = F(i - 2)$ and $curr = F(i - 1)$; at the end $i = n + 1$ and $curr = F(n)$.)

n	$F(n)$	steps of Fib: $4F(n + 1) - 2$	steps of IterativeFib: $5n + 1$
5	5	30	26
10	55	354	51
20	6765	43782	101
30	832040	5385074	151

The naive recursive algorithm **Fib** executes $S(n) = 4F(n + 1) - 2 = 2^{\Theta(n)}$ steps (exponential in n), while **IterativeFib** executes $S(n) = 5n + 1 = \Theta(n)$ steps (linear in n). Since $|I| = \Theta(\log_2 n)$, as functions of the input size the naive algorithm is even doubly exponential, while the iterative one is exponential (as **Factorial**).

- The recursion itself is not the problem: the problem is the **repetition of identical subproblems**. Storing the solution of each subproblem the first time it is solved, and reusing it afterwards, is the key idea of the technique called **Dynamic Programming**.

14.5.3 C++ implementation

- This is a C++ implementation of the naive recursive algorithm and of the iterative algorithm:

```

1 // Naive recursive algorithm: exponential number of calls
2 long long Fib(int n)
3 {
4     if (n <= 1)
5     {
6         return n;                // base cases F(0)=0, F(1)=1
7     }
8     return Fib(n - 1) + Fib(n - 2); // two recursive calls
9 }
10
11 // Iterative algorithm: n-1 iterations
12 long long IterativeFib(int n)
13 {
14     if (n <= 1)
15     {
16         return n;
17     }
18     long long prev = 0; // F(i-2)
19     long long curr = 1; // F(i-1)
20     for (int i = 2; i <= n; i++)
21     {
22         long long next = prev + curr;
23         prev = curr;
24         curr = next;
25     }
26     return curr;
27 }

```

Example 14.7: execution of the C++ implementations for $n = 4$

This is the step-by-step execution of `Fib(4)`. The calls are executed in the depth-first order of the tree of Example 14.2 (indentation = depth in the stack).

call	action
Fib(4)	calls Fib(3)
Fib(3)	calls Fib(2)
Fib(2)	calls Fib(1)
Fib(1)	base case, returns 1
Fib(2)	calls Fib(0)
Fib(0)	base case, returns 0
Fib(2)	returns $1 + 0 = 1$
Fib(3)	calls Fib(1)
Fib(1)	base case, returns 1
Fib(3)	returns $1 + 1 = 2$
Fib(4)	calls Fib(2)
Fib(2)	calls Fib(1)
Fib(1)	base case, returns 1
Fib(2)	calls Fib(0)
Fib(0)	base case, returns 0
Fib(2)	returns $1 + 0 = 1$
Fib(4)	returns $2 + 1 = 3$

The result is $F(4) = 3$, after $C(4) = 2F(5) - 1 = 9$ calls. Instead, `IterativeFib(4)` executes 3 iterations:

iteration	i	$next = prev + curr$	$prev$	$curr$
initialization	2	–	0	1
1	$2 \rightarrow 3$	$0 + 1 = 1$	1	1
2	$3 \rightarrow 4$	$1 + 1 = 2$	1	2
3	$4 \rightarrow 5$	$1 + 2 = 3$	2	3

The loop ends with $i = 5 > 4$ and the function returns $curr = 3$.

14.6 Greatest common divisor

Given two natural numbers n, m with $n \geq 1$ and $m \geq 0$, we consider the problem of computing their greatest common divisor $\gcd(n, m)$ (with $\gcd(n, 0) = n$).

- In the chapter «Numerical algorithms» we proved that, for all natural numbers $n \geq 1$ and $m \geq 1$,

$$\gcd(n, m) = \gcd(m, n \bmod m),$$

and we obtained the iterative algorithm `EuclideanAlgorithm`: it replaces the pair (n, m) by the pair $(m, n \bmod m)$ again and again, until $m = 0$.

- The same property, together with $\gcd(n, 0) = n$, is a recursive definition:

$$\gcd(n, m) = \begin{cases} n & \text{if } m = 0 \\ \gcd(m, n \bmod m) & \text{if } m \geq 1 \end{cases}$$

- This gives the recursive version of the Euclidean algorithm:

Algorithm: RecursiveGCD

Input : Two natural numbers $n \geq 1$ and $m \geq 0$

Output: $\text{gcd}(n, m)$

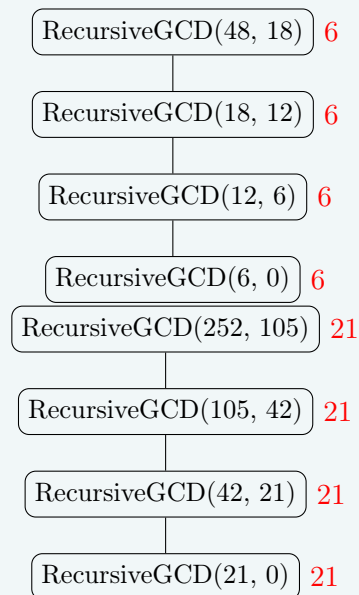
```

1 function RecursiveGCD( $n, m$ )
2   if  $m = 0$  then
3     return  $n$                                 // base case
4   end
5   return RecursiveGCD( $m, n \bmod m$ )           // recursive step
6 end
  
```

- The size of the input is the second argument m . Since $0 \leq n \bmod m < m$, the second argument strictly decreases at each recursive call, hence the base case $m = 0$ is always reached.

Example 14.8: Trees of recursive calls of RecursiveGCD

Each call makes at most one recursive call, so the trees are paths.



- $48 \bmod 18 = 12$, $18 \bmod 12 = 6$, $12 \bmod 6 = 0$: hence $\text{gcd}(48, 18) = 6$.
- $252 \bmod 105 = 42$, $105 \bmod 42 = 21$, $42 \bmod 21 = 0$: hence $\text{gcd}(252, 105) = 21$.
- The value found by the base case is returned unchanged by all the calls.

14.6.1 Correctness

Observation 14.9: correctness of RecursiveGCD

For every pair of natural numbers $n \geq 1$ and $m \geq 0$, the call `RecursiveGCD( $n, m$ )` terminates and returns $\text{gcd}(n, m)$.

Proof. We proceed by strong induction on m .

- **Base case** ($m = 0$): the algorithm returns $n = \text{gcd}(n, 0)$ at line 3.
- **Inductive step** ($m \geq 1$): assume the claim for every second argument $k < m$ and every first argument ≥ 1 . The algorithm calls `RecursiveGCD( $m, n \bmod m$ )`, where $m \geq 1$ and $n \bmod m < m$. By the inductive hypothesis, this call returns $\text{gcd}(m, n \bmod m)$, which is equal to $\text{gcd}(n, m)$ by the property $\text{gcd}(n, m) = \text{gcd}(m, n \bmod m)$ recalled at the beginning of this section.

□

14.6.2 Tail recursion

Definition 14.10: tail recursion

A recursive call is a **tail call** if it is the last operation executed by the calling function, and its result is returned unchanged. A recursive algorithm whose recursive calls are all tail calls is **tail recursive**.

- `RecursiveGCD` is tail recursive: line 5 returns directly the result of the recursive call, without any further operation.
- `Factorial` is **not** tail recursive: after the recursive call `Factorial( $n - 1$ )` returns, the result must still be multiplied by n , so the frame of the caller must be kept in the stack.

A tail-recursive algorithm can always be transformed into a **while** loop: the tail call is replaced by the update of the parameters, and the loop continues until the base case is reached. Applying this transformation to `RecursiveGCD`, with the update $(n, m) \leftarrow (m, n \bmod m)$, we obtain exactly the iterative `EuclideanAlgorithm`, which uses a constant amount of memory. Many compilers apply this transformation automatically (**tail-call optimization**).

14.6.3 Number of steps

- Each call executes exactly 2 lines (lines 2–3, or lines 2 and 5), so the number of steps is twice the number of calls. We bound the number of calls.
- Let $m_0 = m$ and let $m_0, m_1, m_2, \dots, m_K = 0$ be the second arguments of the successive calls. The call $k \geq 1$ is `RecursiveGCD`(m_{k-1}, m_k) and therefore

$$m_{k+1} = m_{k-1} \bmod m_k, \quad m_{k+1} < m_k.$$

Observation 14.11: the second argument halves every two calls

For every k such that m_{k+2} is defined,

$$m_{k+2} < \frac{m_k}{2}.$$

Proof. We have $m_{k+2} = m_k \bmod m_{k+1}$ with $m_{k+1} < m_k$. We distinguish two cases.

- If $m_{k+1} \leq m_k/2$, then $m_{k+2} < m_{k+1} \leq m_k/2$.
- If $m_{k+1} > m_k/2$, then $m_k < 2m_{k+1}$, so the quotient of m_k divided by m_{k+1} is 1 and $m_{k+2} = m_k - m_{k+1} < m_k - m_k/2 = m_k/2$.

□

- Let $m \geq 1$ and $j = \lfloor \log_2 m \rfloor + 1$, so that $2^j > m$. Applying the Observation j times, if m_{2j} is defined then

$$m_{2j} < \frac{m}{2^j} < 1 \implies m_{2j} = 0.$$

Hence $K \leq 2j$ and the number of calls ($K + 1$) is at most $2\lfloor \log_2 m \rfloor + 3$. Therefore

$$S(n, m) \leq 2(2\lfloor \log_2 m \rfloor + 3) = O(\log_2 m).$$

- The input consists of the two numbers n and m , so $|I| = \Omega(\log_2 m)$ (for $m \geq 2$) and, as for `ChangeToBase2` in the chapter «Running time», $S(n, m) = O(|I|)$.

The `RecursiveGCD` algorithm makes at most $2\lfloor \log_2 m \rfloor + 3$ calls, and its worst-case running time is

$$T(|I|) = O(|I|),$$

i.e., it is at most linear in the input size. (It can be shown that the worst case occurs for two consecutive Fibonacci numbers: for example $\text{gcd}(89, 55)$ requires 10 calls, against the bound $2\lfloor \log_2 55 \rfloor + 3 = 13$.)

14.6.4 C++ implementation

```

1 // Returns gcd(n, m), with n >= 1 and m >= 0
2 int RecursiveGCD(int n, int m)
3 {
4     if (m == 0)
5     {
6         return n;                // base case: gcd(n,0) = n
7     }
8     return RecursiveGCD(m, n % m); // tail call
9 }

```

Example 14.9: execution of the C++ implementation of RecursiveGCD

This is the step-by-step execution of the C++ implementation for $n = 18$ and $m = 48$ (note that $n < m$: the first call simply swaps the two numbers, since $18 \bmod 48 = 18$).

1. RecursiveGCD(18, 48): $m \neq 0$, $18\%48 = 18 \Rightarrow$ calls RecursiveGCD(48, 18)
2. RecursiveGCD(48, 18): $m \neq 0$, $48\%18 = 12 \Rightarrow$ calls RecursiveGCD(18, 12)
3. RecursiveGCD(18, 12): $m \neq 0$, $18\%12 = 6 \Rightarrow$ calls RecursiveGCD(12, 6)
4. RecursiveGCD(12, 6): $m \neq 0$, $12\%6 = 0 \Rightarrow$ calls RecursiveGCD(6, 0)
5. RecursiveGCD(6, 0): $m = 0 \Rightarrow$ base case, returns 6

The value 6 is returned unchanged by all the calls: $\text{gcd}(18, 48) = 6$, after 5 calls.

14.7 Fast exponentiation

Given a number x and a natural number n , we consider the problem of computing the power x^n (with $x^0 = 1$).

- The naive algorithm multiplies n times by x :

$$p := 1, \quad p \leftarrow p \cdot x \quad (n \text{ times}),$$

which requires n multiplications and $\Theta(n)$ steps.

- A much faster algorithm is obtained by **squaring**. Since $n = 2\lfloor n/2 \rfloor$ if n is even, and $n = 2\lfloor n/2 \rfloor + 1$ if n is odd,

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x^{\lfloor n/2 \rfloor})^2 & \text{if } n \geq 1 \text{ is even} \\ x \cdot (x^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is odd} \end{cases}$$

(for n odd, $\lfloor n/2 \rfloor = (n-1)/2$). The size of the exponent is halved at each recursive call.

Algorithm: FastPower**Input** : A number x and a natural number n **Output**: The power x^n

```

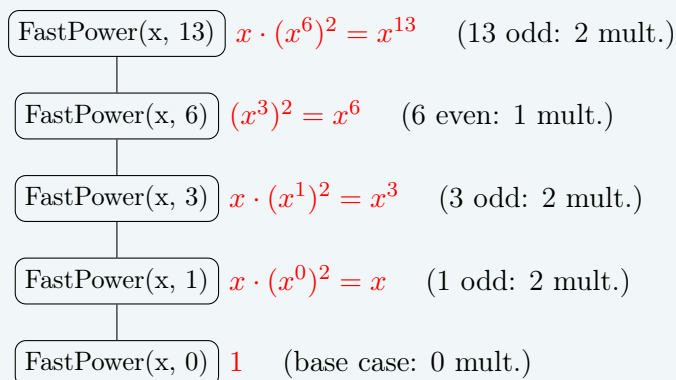
1 function FastPower( $x, n$ )
2   if  $n = 0$  then
3     return 1                                // base case
4   end
5    $y := \text{FastPower}(x, \lfloor n/2 \rfloor)$           // one recursive call
6   if  $n \bmod 2 = 0$  then
7     return  $y \cdot y$                           //  $n$  even
8   else
9     return  $x \cdot y \cdot y$                     //  $n$  odd
10  end
11 end

```

- Note that the result of the recursive call is stored in the variable y and used twice: writing $\text{FastPower}(x, \lfloor n/2 \rfloor) \cdot \text{FastPower}(x, \lfloor n/2 \rfloor)$ would make **two** identical recursive calls and would destroy the efficiency of the algorithm (this variant is analysed in the exercises of the chapter).

Example 14.10: Tree of recursive calls of FastPower($x, 13$)

The exponents of the calls are 13, 6, 3, 1, 0 ($\lfloor 13/2 \rfloor = 6$, $\lfloor 6/2 \rfloor = 3$, $\lfloor 3/2 \rfloor = 1$, $\lfloor 1/2 \rfloor = 0$). On the right of each call: the value returned and the number of multiplications executed by that call.



- Total number of multiplications: $2 + 1 + 2 + 2 + 0 = 7$, against the 13 multiplications of the naive algorithm.
- Note that $13 = (1101)_2$: the odd/even exponents met going up from $\text{FastPower}(x, 1)$ to $\text{FastPower}(x, 13)$ (odd, odd, even, odd) are exactly the binary digits 1, 1, 0, 1 of 13 read from the most significant one. In general, the number of multiplications is the number of binary digits of n plus the number of digits equal to 1: here $4 + 3 = 7$.

14.7.1 Correctness

Observation 14.12: correctness of FastPower

For every number x and every natural number n , the call **FastPower**(x, n) terminates and returns x^n .

Proof. We proceed by strong induction on n .

- **Base case** ($n = 0$): the algorithm returns $1 = x^0$ at line 3.
- **Inductive step** ($n \geq 1$): assume that **FastPower**(x, k) returns x^k for every $k < n$. Since $n \geq 1$, we have $\lfloor n/2 \rfloor < n$, so by the inductive hypothesis line 5 assigns $y = x^{\lfloor n/2 \rfloor}$.
 - If n is even, then $n = 2\lfloor n/2 \rfloor$ and line 7 returns $y \cdot y = x^{2\lfloor n/2 \rfloor} = x^n$.
 - If n is odd, then $n = 2\lfloor n/2 \rfloor + 1$ and line 9 returns $x \cdot y \cdot y = x^{2\lfloor n/2 \rfloor + 1} = x^n$.

Note that strong induction is needed: the inductive hypothesis is used for $k = \lfloor n/2 \rfloor$, not for $k = n - 1$. □

14.7.2 Number of steps

- Let $S(n)$ denote the number of steps of **FastPower** on the exponent n . If $n = 0$, lines 2–3 are executed: $S(0) = 2$. If $n \geq 1$, lines 2, 5, 6 and one of the lines 7, 9 are executed once each, plus the steps of the call on $\lfloor n/2 \rfloor$:

$$S(n) = S(\lfloor n/2 \rfloor) + 4 \quad (n \geq 1).$$

- We solve the recurrence by unrolling it. Since $\lfloor \lfloor n/2^k \rfloor / 2 \rfloor = \lfloor n/2^{k+1} \rfloor$,

$$\begin{aligned} S(n) &= S(\lfloor n/2 \rfloor) + 4 \\ &= S(\lfloor n/4 \rfloor) + 2 \cdot 4 \\ &\vdots \\ &= S(\lfloor n/2^k \rfloor) + k \cdot 4, \end{aligned}$$

as long as $\lfloor n/2^{k-1} \rfloor \geq 1$. Now $\lfloor n/2^k \rfloor \geq 1$ if and only if $2^k \leq n$, i.e., $k \leq \lfloor \log_2 n \rfloor$. Hence, for $n \geq 1$, the exponents $n, \lfloor n/2 \rfloor, \dots, \lfloor n/2^{\lfloor \log_2 n \rfloor} \rfloor = 1$ are all ≥ 1 , and the next one is 0. Choosing $k = \lfloor \log_2 n \rfloor + 1$:

$$S(n) = S(0) + 4(\lfloor \log_2 n \rfloor + 1) = 4\lfloor \log_2 n \rfloor + 6.$$

- For example, $S(13) = 4 \cdot 3 + 6 = 18$. As shown for **ChangeToBase2** in the chapter «Running time»,

$$\lim_{n \rightarrow \infty} \frac{4\lfloor \log_2 n \rfloor + 6}{\log_2 n} = 4 \in \mathbb{R}_{>0}, \quad \text{hence} \quad S(n) = \Theta(\log_2 n).$$

- For $n \geq 1$, the number of calls, and therefore the depth of the recursion, is $\lfloor \log_2 n \rfloor + 2$ (the number of binary digits of n , plus the call with exponent 0).

- The number of multiplications is at most $2(\lfloor \log_2 n \rfloor + 1)$, against n for the naive algorithm. For example, for $n = 1000 = (1111101000)_2$: $10 + 6 = 16$ multiplications instead of 1000.
- Assume that x and all the intermediate results use a constant number of bits (so each multiplication is one step). The input size is $|I| = \Theta(\log_2 n)$, hence, as for **ChangeToBase2** in the chapter «Running time», $S(n) = \Theta(\log_2 n)$ gives $T(|I|) = \Theta(|I|)$.

The number of steps of the **FastPower** algorithm is $S(n) = 4\lfloor \log_2 n \rfloor + 6 = \Theta(\log_2 n)$, and its worst-case running time is

$$T(|I|) = \Theta(|I|),$$

i.e., it is linear in the input size. The naive algorithm executes $\Theta(n)$ steps, i.e., it is exponential in the input size.

14.7.3 C++ implementation

```

1 // Returns x^n (n >= 0), by repeated squaring
2 long long FastPower(long long x, int n)
3 {
4     if (n == 0)
5     {
6         return 1;                // base case: x^0 = 1
7     }
8     long long y = FastPower(x, n / 2); // integer division = floor(n/2)
9     if (n % 2 == 0)
10    {
11        return y * y;             // n even
12    }
13    else
14    {
15        return x * y * y;         // n odd
16    }
17 }
```

Example 14.11: execution of the C++ implementation of FastPower

This is the step-by-step execution of the C++ implementation for $x = 3$ and $n = 13$.

Calls

1. **FastPower**(3, 13): $13 \neq 0 \Rightarrow$ calls **FastPower**(3, 6) (13/2 = 6)
2. **FastPower**(3, 6): $6 \neq 0 \Rightarrow$ calls **FastPower**(3, 3) (6/2 = 3)
3. **FastPower**(3, 3): $3 \neq 0 \Rightarrow$ calls **FastPower**(3, 1) (3/2 = 1)
4. **FastPower**(3, 1): $1 \neq 0 \Rightarrow$ calls **FastPower**(3, 0) (1/2 = 0)
5. **FastPower**(3, 0): base case, returns 1

Returns

1. **FastPower**(3, 1): $y = 1$, 1 is odd $\Rightarrow$ returns $3 \cdot 1 \cdot 1 = 3$

2. `FastPower(3, 3)`: $y = 3$, 3 is odd $\Rightarrow$ returns $3 \cdot 3 \cdot 3 = 27$

3. `FastPower(3, 6)`: $y = 27$, 6 is even $\Rightarrow$ returns $27 \cdot 27 = 729$

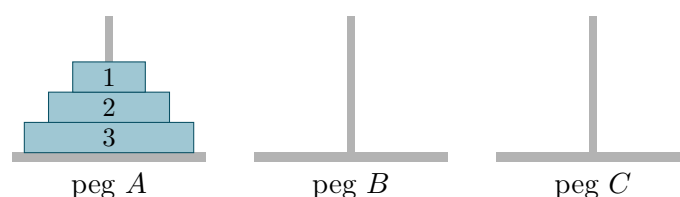
4. `FastPower(3, 13)`: $y = 729$, 13 is odd $\Rightarrow$ returns $3 \cdot 729 \cdot 729 = 1594323$

The algorithm terminates with result $1594323 = 3^{13}$, after 5 calls and 7 multiplications.

14.8 Towers of Hanoi

There are three pegs A, B, C and $n \geq 1$ disks of different sizes, numbered $1, 2, \dots, n$ from the smallest to the largest. Initially all the disks are stacked on peg A , in decreasing order of size from the bottom to the top. We consider the problem of moving all the disks to peg C , with the following rules:

- only one disk is moved at a time, taking the top disk of a peg and putting it on top of another peg;
- a disk can never be placed on top of a smaller disk.



- **Recursive idea.** To move n disks from peg A to peg C using peg B as auxiliary peg:
 1. move the $n - 1$ smallest disks from A to B (using C as auxiliary peg), recursively;
 2. move the largest disk n from A to C ;
 3. move the $n - 1$ smallest disks from B to C (using A as auxiliary peg), recursively.
- During steps 1 and 3, disk n is at the bottom of a peg and it is larger than all the moved disks, so it never violates the rules: the $n - 1$ disks can be moved as if disk n were not there.

Algorithm: Hanoi

Input : The number of disks $n \geq 1$, the source peg *from*, the destination peg *to*, the auxiliary peg *aux*

Output: The sequence of moves that transfers disks $1, 2, \dots, n$ from *from* to *to*

```

1 function Hanoi(n, from, to, aux)
2   if n = 1 then
3     | move disk 1 from from to to                                // base case
4   else
5     | Hanoi(n - 1, from, aux, to)
6     | move disk n from from to to
7     | Hanoi(n - 1, aux, to, from)
8   end
9 end

```

- The initial call is `Hanoi(n, A, C, B)`.

Observation 14.13: correctness of Hanoi

For every natural number $n \geq 1$, if disks $1, 2, \dots, n$ are the top n disks of peg *from* and all the disks on pegs *to* and *aux* are larger than n , then `Hanoi(n, from, to, aux)` moves disks $1, 2, \dots, n$ on top of peg *to*, respecting the rules.

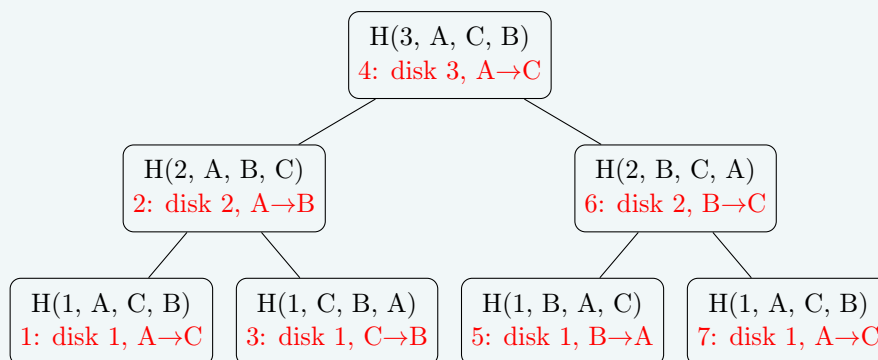
Proof. We proceed by induction on n .

- **Base case** ($n = 1$): the single move of disk 1 is valid, since disk 1 is the smallest disk.
- **Inductive step** ($n \geq 2$): by the inductive hypothesis, the call of line 5 correctly moves the tower of the $n - 1$ smallest disks from *from* to *aux* (disk n and the disks below it are larger, so they do not interfere). Then disk n is on top of peg *from*, and every disk on peg *to* is larger than n : line 6 moves disk n correctly. Finally, by the inductive hypothesis, the call of line 7 moves the $n - 1$ smallest disks from *aux* on top of disk n .

□

Example 14.12: Tree of recursive calls of Hanoi(3, A, C, B)

Each node shows the call (with arguments n , *from*, *to*, *aux*) and, in red, the move executed by the call itself (line 3 or line 6), numbered in the order of execution.



The moves are executed in the order of an **in-order** visit of the tree (left subtree, node, right subtree):

move	disk	from	to
1	1	A	C
2	2	A	B
3	1	C	B
4	3	A	C
5	1	B	A
6	2	B	C
7	1	A	C

The tree has 7 nodes and every node executes exactly one move: 7 moves in total.

14.8.1 Number of moves

- Let $M(n)$ be the number of moves executed by **Hanoi** with n disks. From the pseudo-code,

$$M(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2M(n-1) + 1 & \text{if } n \geq 2 \end{cases}$$

- Unrolling: $M(n) = 2M(n-1) + 1 = 4M(n-2) + 2 + 1 = 8M(n-3) + 4 + 2 + 1 = \dots = 2^{n-1}M(1) + 2^{n-2} + \dots + 2 + 1 = 2^n - 1$. We prove it formally:

Observation 14.14: number of moves of Hanoi

For every natural number $n \geq 1$,

$$M(n) = 2^n - 1.$$

Proof. We proceed by induction on n .

- Base case** ($n = 1$): $M(1) = 1 = 2^1 - 1$.
- Inductive step** ($n \geq 2$): assume $M(n-1) = 2^{n-1} - 1$. Then

$$M(n) = 2M(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1.$$

□

- The number of steps satisfies $S(1) = 2$ (lines 2–3) and $S(n) = 2S(n-1) + 4$ for $n \geq 2$ (lines 2, 5, 6, 7, plus the two recursive calls). Unrolling as above,

$$S(n) = 2^{n-1}S(1) + 4(2^{n-2} + \cdots + 2 + 1) = 2^n + 4(2^{n-1} - 1) = 3 \cdot 2^n - 4 = \Theta(2^n),$$

since $\lim_{n \rightarrow \infty} (3 \cdot 2^n - 4)/2^n = 3 \in \mathbb{R}_{>0}$.

- The depth of the recursion is only n : the memory used by the call stack is $\Theta(n)$.

14.8.2 Optimality

- The exponential number of moves is not a weakness of the algorithm: **no** algorithm can solve the problem with fewer moves.

Observation 14.15: optimality of Hanoi

Let $M^*(n)$ be the minimum number of moves needed to transfer a tower of $n \geq 1$ disks from one peg to another one. Then

$$M^*(n) = 2^n - 1.$$

Sketch of proof. Since the algorithm **Hanoi** uses $2^n - 1$ moves, $M^*(n) \leq 2^n - 1$. We show $M^*(n) \geq 2^n - 1$ by induction on n . For $n = 1$, at least one move is needed.

Let $n \geq 2$. The largest disk n must be moved **at least once**. Consider its first move, from the source peg to some other peg P : at that moment disk n is the only disk on the source peg and peg P is empty, so the $n - 1$ smaller disks are all stacked on the third peg. Moving them there from the source peg required at least $M^*(n - 1)$ moves.

Consider now the last move of disk n , onto the destination peg. After this move, the $n - 1$ smaller disks are all stacked on a peg different from the destination, and they must still be transferred on top of disk n : this requires at least $M^*(n - 1)$ further moves.

These two groups of moves are distinct (one happens before the first move of disk n , the other after its last move), hence

$$M^*(n) \geq M^*(n - 1) + 1 + M^*(n - 1) = 2M^*(n - 1) + 1 \geq 2(2^{n-1} - 1) + 1 = 2^n - 1.$$

□

The **Hanoi** algorithm executes $M(n) = 2^n - 1$ moves and $S(n) = 3 \cdot 2^n - 4 = \Theta(2^n)$ steps, with depth of recursion n . The number of moves is **optimal**: every algorithm must output $2^n - 1$ moves, so the running time of any algorithm for this problem is exponential in n (and therefore doubly exponential in the input size $|I| = \Theta(\log_2 n)$).

- With $n = 64$ disks (the original legend) and one move per second, $2^{64} - 1$ seconds are more than $5.8 \cdot 10^{11}$ years.

14.8.3 C++ implementation

```

1 #include <iostream>
2 using namespace std;
3
4 // Prints the moves that transfer disks 1..n from peg "from" to peg "to"
5 void Hanoi(int n, char from, char to, char aux)
6 {
7     if (n == 1)
8     {
9         cout << "move disk 1 from " << from << " to " << to << endl;
10        return;
11    }
12    Hanoi(n - 1, from, aux, to);
13    cout << "move disk " << n << " from " << from << " to " << to << endl;
14    Hanoi(n - 1, aux, to, from);
15 }
16
17 int main()
18 {
19     Hanoi(3, 'A', 'C', 'B');
20     return 0;
21 }

```

Example 14.13: execution of the C++ implementation of Hanoi for $n = 3$

This is the step-by-step execution for $n = 3$ (indentation = depth in the stack; the printed lines are the moves of Example 14.12).

call	action
Hanoi(3,A,C,B)	calls Hanoi(2,A,B,C)
Hanoi(2,A,B,C)	calls Hanoi(1,A,C,B)
Hanoi(1,A,C,B)	prints move disk 1 from A to C , returns
Hanoi(2,A,B,C)	prints move disk 2 from A to B , calls Hanoi(1,C,B,A)
Hanoi(1,C,B,A)	prints move disk 1 from C to B , returns
Hanoi(2,A,B,C)	returns
Hanoi(3,A,C,B)	prints move disk 3 from A to C , calls Hanoi(2,B,C,A)
Hanoi(2,B,C,A)	calls Hanoi(1,B,A,C)
Hanoi(1,B,A,C)	prints move disk 1 from B to A , returns
Hanoi(2,B,C,A)	prints move disk 2 from B to C , calls Hanoi(1,A,C,B)
Hanoi(1,A,C,B)	prints move disk 1 from A to C , returns
Hanoi(2,B,C,A)	returns
Hanoi(3,A,C,B)	returns

All the 7 moves are printed and all the disks are on peg C .

14.9 Summary

algorithm	recurrence for $S(n)$	$S(n)$	calls	depth
Factorial	$S(n) = S(n-1) + 2$	$2n + 2 = \Theta(n)$	$n + 1$	$n + 1$
Fib (naive)	$S(n) = S(n-1) + S(n-2) + 2$	$4F(n+1) - 2 = 2^{\Theta(n)}$	$2F(n+1) - 1$	$\max(n, 1)$
RecursiveGCD	$m_{k+2} < m_k/2$ (Obs. 14.11)	$O(\log_2 m)$	$\leq 2\lfloor \log_2 m \rfloor + 3$	= calls
FastPower	$S(n) = S(\lfloor n/2 \rfloor) + 4$	$4\lfloor \log_2 n \rfloor + 6 = \Theta(\log_2 n)$	$\lfloor \log_2 n \rfloor + 2$	= calls
Hanoi	$S(n) = 2S(n-1) + 4$	$3 \cdot 2^n - 4 = \Theta(2^n)$	$2^n - 1$	n

- A **recursive algorithm** solves the **base cases** directly and reduces every other instance to **strictly smaller** instances of the same problem; this guarantees termination.
- The execution is described by the **tree of recursive calls**: the number of steps depends on the **number of nodes**, the memory used by the **call stack** on the **depth** of the tree.
- Correctness is proved by **(strong) induction** on the size of the input: base cases $\leftrightarrow$ base of the induction, recursive step $\leftrightarrow$ inductive step (as initialization and maintenance for loop invariants).
- The number of steps is described by a **recurrence** for $S(n)$: decreasing by one with constant work gives $\Theta(n)$ (Factorial); halving with constant work gives $\Theta(\log_2 n)$ (FastPower); two calls on $n-1$ give $\Theta(2^n)$ (Hanoi).
- Recursion can be very inefficient when the **same subproblems** are solved many times (naive Fib): this motivates **Dynamic Programming**.
- A **tail-recursive** algorithm (RecursiveGCD) can be directly rewritten as a loop.

Recurrences

Introduction

In the chapter *Recursion* we counted the steps of a recursive algorithm line by line, and we noticed that the number of steps executed by a call depends on the number of steps executed by its recursive calls, which are calls of the *same* algorithm on a *smaller* input. The quantity we want to compute is therefore defined in terms of itself: this is a **recurrence**.

In this chapter we first define recurrences and show where they come from (Section 15.1). Then we study three methods to solve them, that is, to obtain their closed form or at least their order of growth:

- the **expansion method** (Section 15.2), which unrolls the recurrence until the boundary condition is reached;
- the **substitution method** (Section 15.3), which proves a guessed bound by induction;
- the **recursion-tree method** (Section 15.4), which visualises the costs of all the calls and sums them level by level.

Finally (Section 15.5) we draw the recursion tree of a generic divide-and-conquer recurrence: the three behaviours that we observe there are the starting point of the chapter *The Master Theorem*.

15.1 Recurrences

Definition 15.1: recurrence

A **recurrence** is an equation (or an inequality) that describes a function $T : \mathbb{N} \rightarrow \mathbb{R}$ in terms of its values on smaller inputs.

- Each recurrence has **boundary conditions**, which give the value of the function when the input size is small, that is, when n reaches (or is below) a specific value n_0 . Precisely, the boundary conditions are

$$T(n) = g(n) \quad \text{if } n \leq n_0,$$

where $g(n)$ is the value of the function for the small values of n . Typically $n_0 = 1$ and the boundary condition is $T(1) = \Theta(1)$.

- **Solving a recurrence** means determining a closed form of $T(n)$ or, at least, its **order of growth**, that is, a function $h(n)$ such that $T(n) = \Theta(h(n))$ (or $T(n) = O(h(n))$, or $T(n) = \Omega(h(n))$).

Example 15.1: A first recurrence

Consider the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n > 1 \end{cases}$$

The boundary condition is $T(1) = 1$ (here $n_0 = 1$); for $n > 1$ every value is obtained from the previous one:

- $T(1) = 1$ (boundary condition);
- $T(2) = T(1) + 2 = 1 + 2 = 3$;
- $T(3) = T(2) + 3 = 3 + 3 = 6$;
- $T(4) = T(3) + 4 = 6 + 4 = 10$.

The recurrence defines $T(n)$ for every $n \in \mathbb{N}$, $n \geq 1$, but computing $T(n)$ in this way requires $n - 1$ additions. In Example 15.4 we solve it and obtain the closed form $T(n) = \frac{n(n+1)}{2}$.

15.1.1 Recurrences and recursive algorithms

Recurrences are the natural tool to determine the asymptotic complexity of **recursive algorithms**.

In this chapter recurrences are mathematical objects, and we use a generic function $T(n)$. When we analyse a recursive algorithm, the function described by the recurrence is the number of steps $S(n)$ executed in the worst case on an input of size parameter n (see the chapter *Recursion*): the number of steps of a call is the number of steps executed by the call itself plus the number of steps executed by its recursive calls.

- Consider the following recursive algorithm that computes the sum of the entries of an array.

Algorithm: RecursiveSum

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The sum $a[1] + a[2] + \dots + a[n]$

```

1 function RecursiveSum(a, n)
2   if n = 1 then
3     return a[1]
4   end
5   s := RecursiveSum(a, n - 1)
6   return s + a[n]
7 end

```

Example 15.2: The recurrence of RecursiveSum

Let $S(n)$ be the number of steps executed by `RecursiveSum` on an array of size n . We count the executions of each line.

- If $n = 1$, lines 2 and 3 are executed once: $S(1) = 2$.
- If $n > 1$, lines 2, 5 and 6 are executed once (3 steps), and in addition the recursive call at line 5 executes $S(n - 1)$ steps.

Therefore

$$S(n) = \begin{cases} 2 & \text{if } n = 1 \\ S(n - 1) + 3 & \text{if } n > 1 \end{cases}$$

For instance, $S(2) = 2 + 3 = 5$, $S(3) = 5 + 3 = 8$ and $S(4) = 8 + 3 = 11$. In Example 15.3 we show that $S(n) = 3n - 1 = \Theta(n)$.

- The same algorithm, written in C++, is the following function of the code of the course. In C++ the array is indexed from 0, so its entries are $a[0], \dots, a[n-1]$, and the last entry is $a[n-1]$.

```

1 // sum of a[0..n-1] by decrease by one: depth of the recursion Theta(n)
2 long long sumRecDecrease(int *a, int n)
3 {
4     if (n == 0)
5     {
6         return 0;
7     }
8
9     return sumRecDecrease(a, n - 1) + a[n - 1];
10 }
```

- The C++ function stops at the **empty** array ($n = 0$, sum 0) instead of at $n = 1$: it also accepts $n = 0$, and it executes one more call than the pseudo-code. The recurrence has the same form $S(n) = S(n - 1) + \text{constant}$, and the order of growth does not change. On the array 3, 1, 4, 1, 5, 9, 2, 6 the function returns 31.

15.1.2 Different forms of recurrences

The shape of the recurrence mirrors the way the algorithm splits its input. The most common shapes are the following.

- **Divide-and-conquer.** A recursive algorithm divides the problem into a subproblems, each of size n/b , solves them recursively and combines their solutions. This gives rise to a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

For example, the algorithm **MergeSort** (studied in the chapter *Recursive algorithms*) gives rise to

$$T(n) = 2T(n/2) + \Theta(n).$$

- **Unequal split.** A recursive algorithm might divide the problem into subproblems of different sizes. For example, if the input is divided into a $1/3$ and a $2/3$ split, we obtain the recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + f(n),$$

where $f(n)$ is the time required to divide the problem and to combine the solutions of the subproblems.

- **Decrease-and-conquer.** Subproblems are not necessarily a constant fraction of the original problem. For example, a recursive algorithm might create just one subproblem containing one element less than the original problem (as **RecursiveSum**). This gives rise to the recurrence

$$T(n) = T(n-1) + f(n).$$

- **Several subproblems of size $n-1$.** For example, the algorithm that solves the *Tower of Hanoi* puzzle with n disks moves $n-1$ disks, then one disk, then again $n-1$ disks, giving rise to the recurrence

$$T(n) = 2T(n-1) + 1,$$

whose solution is exponential in n (see Example 15.5).

Definition 15.2: recurrence in standard form

A recurrence is in **standard form** if it can be written as

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{15.1}$$

where

- n is the size of the input (typically $n \in \mathbb{N}$);
- $a \geq 1$ is the number of subproblems to solve;
- $\frac{n}{b}$ is the size of each subproblem ($b > 1$);
- $f(n)$ is the time required to divide the problem and to combine the solutions of the subproblems.

- Including the boundary conditions, a recurrence in standard form is

$$T(n) = \begin{cases} g(1) & \text{if } n = 1 \\ a T\left(\frac{n}{b}\right) + f(n) & \text{if } n > 1 \end{cases}$$

where $g(1)$ is the time required to solve a subproblem of size 1, typically $g(1) = \Theta(1)$.

- Occasionally, we shall see recurrences that are not equalities but inequalities, such as

$$T(n) \leq a T\left(\frac{n}{b}\right) + f(n).$$

Because such a recurrence states only an **upper bound** on $T(n)$, its solution is given in O -notation rather than in Θ -notation.

- Similarly, if the inequality is reversed,

$$T(n) \geq a T\left(\frac{n}{b}\right) + f(n),$$

the recurrence gives only a **lower bound** on $T(n)$, and its solution is given in Ω -notation.

15.1.3 Methods for solving recurrences

- In this chapter we study three methods for solving recurrences:
 1. The **expansion method** (also called iteration or unrolling): we repeatedly substitute the recurrence into itself until we reach the boundary condition, and we compute the resulting summation.
 2. The **substitution method**: we guess a bound and then use mathematical induction to prove that the guess is correct.
 3. The **recursion-tree method**: we convert the recurrence into a tree whose nodes represent the costs incurred at the various levels of the recursion, and we bound the resulting summations.
- In the chapter *The Master Theorem* we study a fourth method, the **Master Theorem**, which provides directly the solution of recurrences in standard form (15.1).

Before applying the methods, we fix two technical points: which details of a recurrence can be neglected, and which summations appear most often.

15.1.4 Technicalities in recurrences

- We often neglect certain technical details when we state and solve recurrences.
- **Floors and ceilings.** Consider the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

When n is odd, the two subproblems cannot both have size $n/2$ (which is not an integer): typically, they have sizes $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. The precise recurrence is therefore

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1 \end{cases}$$

The two recurrences have the same asymptotic solution. Since we are interested in the order of growth, floors and ceilings can usually be omitted.

- **Exact powers.** For the same reason, when solving a recurrence in standard form (15.1) we often assume that n is an **exact power of b** , that is, $n = b^k$ for some $k \in \mathbb{N}$. In this way all the subproblem sizes $n/b, n/b^2, \dots, 1$ are integers. In the example above, we assume $n = 2^k$.
- **Boundary conditions.** Boundary conditions are another class of details that we typically ignore. Since the running time of an algorithm on an input of constant size is a constant, the recurrences that arise from the analysis of algorithms generally satisfy

$$T(n) = \Theta(1) \quad \text{for sufficiently small } n.$$

Consequently, for convenience, we shall generally omit the boundary conditions and assume that $T(n)$ is constant for small n . Changing the value of $T(1)$ typically changes the exact solution, but not its order of growth.

Omitting floors, ceilings and boundary conditions, and assuming that n is an exact power of b , simplifies the computations. When a rigorous proof is required (for example in the substitution method), these details must be handled explicitly.

15.1.5 Useful summations

All three methods end with a summation: the costs of the single calls must be added up. The following formulas are used throughout the chapter.

Observation 15.3: useful summations and logarithms

- The *arithmetic series*: for every $n \geq 1$

$$\sum_{i=1}^n i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}. \quad (15.2)$$

- The *geometric series*: for every real number $x \neq 1$ and every $k \in \mathbb{N}$

$$\sum_{i=0}^k x^i = 1 + x + x^2 + \cdots + x^k = \frac{x^{k+1} - 1}{x - 1}. \quad (15.3)$$

In particular, for $x = 2$,

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1. \quad (15.4)$$

- The *infinite decreasing geometric series*: for every real number x with $|x| < 1$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}. \quad (15.5)$$

- Two properties of logarithms: for every $a, b, n > 0$ (with $b \neq 1$)

$$a^{\log_b n} = n^{\log_b a}, \quad \log_b n = \frac{\log_2 n}{\log_2 b}. \quad (15.6)$$

Proof. Arithmetic series. Writing the sum twice, once in increasing and once in decreasing order, and adding term by term, we obtain n pairs, each with sum $n + 1$: twice the sum is $n(n + 1)$.

Geometric series. Multiplying the sum by $x - 1$, all the intermediate terms cancel:

$$(x - 1) \sum_{i=0}^k x^i = \sum_{i=1}^{k+1} x^i - \sum_{i=0}^k x^i = x^{k+1} - 1,$$

and we divide by $x - 1 \neq 0$. For $x = 2$ and $k - 1$ in place of k we obtain $2^k - 1$.

Infinite series. If $|x| < 1$, then $x^{k+1} \rightarrow 0$ for $k \rightarrow \infty$, and the finite sum tends to $\frac{0-1}{x-1} = \frac{1}{1-x}$.

Logarithms. $a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}$. Moreover $n = b^{\log_b n}$ gives $\log_2 n = \log_b n \cdot \log_2 b$. $\square$

- Since all the terms are positive when $0 < x < 1$, every finite sum $\sum_{i=0}^k x^i$ is smaller than $\frac{1}{1-x}$: the infinite series gives an **upper bound** on a finite geometric sum that does not

depend on k .

- The second property of the logarithms shows that logarithms in different bases differ by a constant factor; hence, inside Θ , O and Ω the base of the logarithm is irrelevant.

15.2 The expansion method

The most direct way to solve a recurrence is to apply it again and again: each application replaces a value of T with a value on a smaller argument, and after enough applications only the boundary condition remains.

In the **expansion method** (also called **iteration** or **unrolling**) we:

1. substitute the recurrence into itself a few times, until we recognise the general pattern after k substitutions;
2. choose k such that the argument reaches the boundary condition;
3. compute (or bound) the resulting summation.

- The method gives an exact closed form when the summation can be computed. The pattern recognised in step 1 can be proved rigorously by induction on k .

We start with the recurrences of the decrease-and-conquer type, $T(n-1) + f(n)$, and then move to the divide-and-conquer type, $aT(n/2) + f(n)$.

Example 15.3: $T(n) = T(n-1) + c$

Let $c, d > 0$ be constants and

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ T(n-1) + c & \text{if } n > 1 \end{cases}$$

- *Step 1 (expand).*

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= (T(n-2) + c) + c = T(n-2) + 2c \\ &= (T(n-3) + c) + 2c = T(n-3) + 3c \\ &\vdots \\ &= T(n-k) + kc. \end{aligned}$$

- *Step 2 (boundary condition).* The boundary condition is reached when $n-k=1$, that is, $k=n-1$.
- *Step 3 (sum).*

$$T(n) = T(1) + (n-1)c = cn + (d-c) = \Theta(n).$$

For **RecursiveSum** (Example 15.2) we have $c=3$ and $d=2$, hence

$$S(n) = 3(n-1) + 2 = 3n - 1 = \Theta(n),$$

which gives $S(2)=5$, $S(3)=8$, $S(4)=11$, as computed directly from the recurrence.

Example 15.4: $T(n) = T(n-1) + n$

Let

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n > 1 \end{cases}$$

- *Step 1 (expand).*

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(n-k) + (n-k+1) + \cdots + (n-1) + n. \end{aligned}$$

- *Step 2 (boundary condition).* $n-k=1$, that is, $k=n-1$.
- *Step 3 (sum).* Using the arithmetic series (15.2),

$$T(n) = T(1) + 2 + 3 + \cdots + n = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2).$$

Indeed,

$$\lim_{n \rightarrow \infty} \frac{n(n+1)/2}{n^2} = \lim_{n \rightarrow \infty} \frac{1}{2} \left(1 + \frac{1}{n} \right) = \frac{1}{2} \in \mathbb{R}_{>0}.$$

The closed form gives $T(1) = 1$, $T(2) = 3$, $T(3) = 6$, $T(4) = 10$, the values of Example 15.1.

Example 15.5: $T(n) = 2T(n-1) + 1$: the Tower of Hanoi

Let

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

- *Step 1 (expand).*

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 = 2^2 T(n-2) + 2 + 1 \\ &= 2^2(2T(n-3) + 1) + 2 + 1 = 2^3 T(n-3) + 2^2 + 2 + 1 \\ &\vdots \\ &= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i. \end{aligned}$$

- *Step 2 (boundary condition).* $n-k=1$, that is, $k=n-1$.
- *Step 3 (sum).* Using (15.4),

$$T(n) = 2^{n-1} T(1) + (2^{n-1} - 1) = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 = \Theta(2^n).$$

This is the number of moves needed to solve the Tower of Hanoi puzzle with n disks: the running time is **exponential** in n . The function `hanoi` of the code of the course counts its moves; running it we obtain

n	1	2	3	4	5	6	7	8	10
moves	1	3	7	15	31	63	127	255	1023

that is, exactly $2^n - 1$ moves.

The next two recurrences halve the size of the problem at each call. To avoid floors we assume that n is an exact power of 2.

Example 15.6: $T(n) = T(n/2) + c$

Let $c, d > 0$ be constants, let $n = 2^k$ be an exact power of 2, and

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

- *Step 1 (expand).*

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + 2c = T\left(\frac{n}{2^3}\right) + 3c = \dots = T\left(\frac{n}{2^j}\right) + jc.$$

- *Step 2 (boundary condition).* $\frac{n}{2^j} = 1$, that is, $j = \log_2 n$.
- *Step 3 (sum).*

$$T(n) = T(1) + c \log_2 n = c \log_2 n + d = \Theta(\log_2 n).$$

This recurrence describes, for instance, the recursive version of `BinarySearch`: at each call the size of the sub-array is halved.

Example 15.7: $T(n) = 2T(n/2) + n$

Let $n = 2^k$ be an exact power of 2 and

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{cases}$$

- *Step 1 (expand).*

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n = 2^2 T\left(\frac{n}{2^2}\right) + 2n \\ &= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n = 2^3 T\left(\frac{n}{2^3}\right) + 3n \end{aligned}$$

$$\vdots$$

$$= 2^j T\left(\frac{n}{2^j}\right) + j n.$$

- *Step 2 (boundary condition).* $j = \log_2 n$, so that $2^j = n$.
- *Step 3 (sum).*

$$T(n) = n T(1) + n \log_2 n = n \log_2 n + n = \Theta(n \log_2 n).$$

Check: from the recurrence $T(2) = 2 + 2 = 4$, $T(4) = 8 + 4 = 12$, $T(8) = 24 + 8 = 32$, and from the closed form $T(8) = 8 \cdot 3 + 8 = 32$.

The expansion method works well when the recurrence has a **single recursive term**. With several recursive terms of different sizes (for example $T(n/3) + T(2n/3)$) the expansion quickly becomes unmanageable: in these cases the recursion-tree method (Section 15.4) is more convenient.

15.3 The substitution method

The expansion method needs exact subproblem sizes: with floors and ceilings the pattern is no longer clean. When we already have a good idea of the solution, a more robust approach is to *prove* it by induction.

The **substitution method** comprises two steps:

1. **guess** the form of the solution (for example $T(n) \leq c n \log_2 n$, with c an unknown constant);
2. use **mathematical induction** to find the constants and show that the solution works.

- The name comes from the fact that, in the inductive step, we *substitute* the guessed bound for the values of the function on the smaller inputs.
- The method is very powerful, but it requires a good guess. The guess can come from experience (similar recurrences), from the expansion method or from a recursion tree.
- We can use the substitution method to prove upper bounds (O) and lower bounds (Ω) separately. Both together give Θ .
- Since the asymptotic notation requires the bound only for $n \geq n_0$, we are free to choose n_0 : this is useful when the bound does not hold for the smallest values of n .

15.3.1 An example with floors: upper bound

- Consider the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor n/2 \rfloor) + n & \text{if } n > 1 \end{cases} \quad (15.7)$$

- From Example 15.7 (where n is a power of 2) we guess $T(n) = O(n \log_2 n)$. We want to prove that there exist constants $c > 0$ and n_0 such that

$$T(n) \leq cn \log_2 n \quad \text{for all } n \geq n_0.$$

Inductive step. Assume that the bound holds for all the values m with $n_0 \leq m < n$, in particular for $m = \lfloor n/2 \rfloor$ (this requires $\lfloor n/2 \rfloor \geq n_0$):

$$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor.$$

Substituting into the recurrence:

$$\begin{aligned} T(n) &\leq 2c \lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + n \\ &\leq 2c \frac{n}{2} \log_2 \frac{n}{2} + n && (\text{since } \lfloor n/2 \rfloor \leq n/2) \\ &= cn \log_2 n - cn \log_2 2 + n \\ &= cn \log_2 n - cn + n \\ &\leq cn \log_2 n, \end{aligned}$$

where the last step holds as long as $c \geq 1$.

Base cases. The induction needs the bound to hold for the boundary conditions. However, for $n = 1$:

$$T(1) = 1 > c \cdot 1 \cdot \log_2 1 = 0,$$

for every c . So the bound *does not hold* for $n = 1$. We take advantage of the freedom of choosing n_0 : we prove the bound for $n \geq n_0 = 2$.

- For $n \geq 4$ we have $\lfloor n/2 \rfloor \geq 2$, so the inductive step only uses values of T on arguments ≥ 2 .
- For $n = 2$ and $n = 3$ we have $\lfloor n/2 \rfloor = 1$, so the inductive step would use $T(1)$, which does not satisfy the bound. Therefore $n = 2$ and $n = 3$ must be treated as **base cases**.

From the recurrence, $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$. We need

$$4 \leq c \cdot 2 \log_2 2 = 2c \quad \text{and} \quad 5 \leq c \cdot 3 \log_2 3.$$

Both hold for $c \geq 2$ (since $3 \log_2 3 \approx 4.75$, we have $2 \cdot 3 \log_2 3 \approx 9.51 \geq 5$).

Observation 15.4: upper bound for $2T(\lfloor n/2 \rfloor) + n$

The function $T(n)$ defined in (15.7) satisfies $T(n) \leq 2n \log_2 n$ for all $n \geq 2$. Hence $T(n) = O(n \log_2 n)$.

Proof. Let $c = 2$. We proceed by strong induction on $n \geq 2$.

Base cases $n = 2, 3$: $T(2) = 4 \leq 2 \cdot 2 \cdot \log_2 2 = 4$ and $T(3) = 5 \leq 2 \cdot 3 \log_2 3 \approx 9.51$.

Inductive step $n \geq 4$: we have $2 \leq \lfloor n/2 \rfloor < n$, so by the inductive hypothesis $T(\lfloor n/2 \rfloor) \leq 2\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor$. Then

$$T(n) \leq 4\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + n \leq 2n \log_2 \frac{n}{2} + n = 2n \log_2 n - 2n + n \leq 2n \log_2 n. \quad \square$$

15.3.2 An example with floors: lower bound

- We now prove $T(n) = \Omega(n \log_2 n)$ for the same recurrence (15.7). We look for a constant $d > 0$ such that $T(n) \geq dn \log_2 n$ for all $n \geq 1$.
- For the lower bound, the floor works against us: we need lower bounds on $\lfloor n/2 \rfloor$. For every $n \geq 2$:

$$2\lfloor n/2 \rfloor \geq n - 1 \quad \text{and} \quad \lfloor n/2 \rfloor \geq \frac{n}{3}.$$

The first holds because $\lfloor n/2 \rfloor \geq (n-1)/2$. The second holds for $n = 2$ ($1 \geq 2/3$) and, for $n \geq 3$, because $(n-1)/2 \geq n/3$ is equivalent to $n \geq 3$.

Inductive step ($n \geq 2$). Assume $T(\lfloor n/2 \rfloor) \geq d\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor$. Since $\lfloor n/2 \rfloor \geq 1$, the factor $\log_2 \lfloor n/2 \rfloor$ is non-negative, and

$$\begin{aligned} T(n) &\geq 2d\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + n \\ &\geq d(n-1) \log_2 \frac{n}{3} + n \\ &= dn \log_2 n - dn \log_2 3 - d \log_2 n + d \log_2 3 + n \\ &\geq dn \log_2 n + (n - dn \log_2 3 - d \log_2 n). \end{aligned}$$

Since $\log_2 n \leq n$, the term in parentheses is at least $n(1 - d \log_2 3 - d)$, which is non-negative if

$$d \leq \frac{1}{1 + \log_2 3} \approx 0.387.$$

We choose $d = \frac{1}{4}$.

Base case ($n = 1$): $T(1) = 1 \geq \frac{1}{4} \cdot 1 \cdot \log_2 1 = 0$. ✓

Observation 15.5: lower bound for $2T(\lfloor n/2 \rfloor) + n$

The function $T(n)$ defined in (15.7) satisfies $T(n) \geq \frac{1}{4}n \log_2 n$ for all $n \geq 1$. Hence $T(n) = \Omega(n \log_2 n)$.

Proof. By strong induction on $n \geq 1$, with $d = \frac{1}{4}$. For $n = 1$, $T(1) = 1 \geq 0$. For $n \geq 2$ we have $1 \leq \lfloor n/2 \rfloor < n$, so the inductive hypothesis holds for $\lfloor n/2 \rfloor$, and the computation of the inductive step above gives

$$T(n) \geq \frac{1}{4} n \log_2 n + n \left(1 - \frac{1}{4} \log_2 3 - \frac{1}{4}\right) \geq \frac{1}{4} n \log_2 n,$$

since $1 - \frac{1}{4}(1 + \log_2 3) \approx 0.35 > 0$. □

From Observations 15.4 and 15.5, the solution of (15.7) is

$$T(n) = \Theta(n \log_2 n).$$

The floor does not change the order of growth found for n power of 2. Numerically, for every n with $2 \leq n < 200\,000$ the ratio $\frac{T(n)}{n \log_2 n}$ lies between 0.83 and 2.

15.3.3 When the guess must be strengthened

- Sometimes the guess is correct, but the induction does not work because the inductive hypothesis is *not strong enough*. Consider

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\lfloor n/2 \rfloor) + 1 & \text{if } n > 1 \end{cases} \quad (15.8)$$

- For n power of 2 one easily finds $T(n) = 2n - 1$ (for instance $T(2) = 3$, $T(4) = 7$, $T(8) = 15$), so we guess $T(n) = O(n)$, that is, $T(n) \leq cn$.
- **Naive attempt.** Substituting the guess,

$$T(n) \leq 2c \lfloor n/2 \rfloor + 1 \leq cn + 1,$$

which does **not** imply $T(n) \leq cn$ for any choice of c . The induction fails, by a lower-order term (the constant 1).

- **Strengthened guess.** It is tempting to try a larger guess, such as cn^2 , but this is not needed. Instead, we *subtract* a lower-order term: we guess

$$T(n) \leq cn - d,$$

with $d \geq 0$ a constant. Then

$$T(n) \leq 2(c \lfloor n/2 \rfloor - d) + 1 \leq cn - 2d + 1 \leq cn - d,$$

where the last step holds as long as $d \geq 1$.

- **Base case.** $T(1) = 1 \leq c - d$: with $d = 1$ we need $c \geq 2$.

Observation 15.6: strengthened guess for $2T(\lfloor n/2 \rfloor) + 1$

The function $T(n)$ defined in (15.8) satisfies $n \leq T(n) \leq 2n - 1$ for all $n \geq 1$. Hence $T(n) = \Theta(n)$.

Proof. By strong induction on n . For $n = 1$: $1 \leq T(1) = 1 \leq 2 \cdot 1 - 1$. For $n \geq 2$, we have $1 \leq \lfloor n/2 \rfloor < n$ and, by the inductive hypothesis, $\lfloor n/2 \rfloor \leq T(\lfloor n/2 \rfloor) \leq 2\lfloor n/2 \rfloor - 1$.
Upper bound: $T(n) \leq 2(2\lfloor n/2 \rfloor - 1) + 1 = 4\lfloor n/2 \rfloor - 1 \leq 2n - 1$.
Lower bound: $T(n) \geq 2\lfloor n/2 \rfloor + 1 \geq (n - 1) + 1 = n$. □

When the induction fails only because of a **lower-order term**, do not increase the guess: **subtract a lower-order term** from it (for example $cn - d$ instead of cn , or $cn^2 - dn$ instead of cn^2). The stronger hypothesis makes the induction work.

- **Avoid pitfalls.** In the naive attempt it would be **wrong** to conclude “ $T(n) \leq cn + 1 = O(n)$, done”. The induction must prove *exactly* the form of the inductive hypothesis, $T(n) \leq cn$, with the *same* constant c ; otherwise the constant would grow at each step of the recursion.

15.3.4 Changing variables

Sometimes an algebraic manipulation, such as a change of variables, transforms an unknown recurrence into a known one, whose solution can then be reused.

Example 15.8: $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log_2 n$

Consider

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log_2 n,$$

with boundary condition $T(2) = 1$.

- *Step 1 (rename).* Ignoring the rounding (that is, assuming that $n = 2^{2^k}$, so that all the square roots are exact powers of 2), we rename $m = \log_2 n$, that is, $n = 2^m$ and $\sqrt{n} = 2^{m/2}$:

$$T(2^m) = 2T(2^{m/2}) + m.$$

- *Step 2 (new function).* We define $R(m) = T(2^m)$ and obtain

$$R(m) = 2R\left(\frac{m}{2}\right) + m,$$

which is the recurrence of Example 15.7. Since $T(2) = 1$, we have $R(1) = 1$, and therefore

$$R(m) = m \log_2 m + m.$$

- *Step 3 (back to T).* With $m = \log_2 n$:

$$T(n) = R(\log_2 n) = \log_2 n \cdot \log_2 \log_2 n + \log_2 n = \Theta(\log_2 n \cdot \log_2 \log_2 n).$$

Check: $T(4) = 2T(2) + 2 = 4$ and $R(2) = 2 \cdot 1 + 2 = 4$; $T(16) = 2T(4) + 4 = 12$ and $R(4) = 4 \cdot 2 + 4 = 12$.

15.4 The recursion-tree method

The substitution method proves a guess, but does not produce it. The recursion tree is the tool that produces good guesses, and in many cases it directly gives the solution.

- In a **recursion tree**, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. The children of a node are the subproblems generated by it.
- It can be used to solve recurrences, that is, to determine their asymptotic complexity, or to produce a good guess to be verified with the substitution method.
- **Main idea:** we sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all the levels of the recursion.

In the **recursion-tree method** we determine:

1. the size of the subproblems and the number of nodes at each depth i ;
2. the cost of each level (number of nodes $\times$ cost of each node);
3. the depth of the tree (when the subproblem size reaches the boundary condition) and the cost of the leaves;
4. the sum of the costs of all levels.

We apply the method to four recurrences, which show the three possible behaviours of the level costs (decreasing, constant, increasing) and the case of an unbalanced tree.

15.4.1 Decreasing level costs: $T(n) = 3T(n/4) + cn^2$

- Consider the recurrence ($c > 0$ a constant)

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2.$$

For convenience, we assume that n is an exact power of 4, so that all subproblem sizes are integers.

- The cn^2 term at the root represents the cost at the top level of the recursion (dividing and combining).
- The three subtrees of the root represent the costs incurred by the subproblems of size $\frac{n}{4}$; at the next depth the size becomes $\frac{n}{16}$, and so on.
- The leaves represent the points where the recursion “bottoms out”, that is, where the *base case* of the recursion is reached.

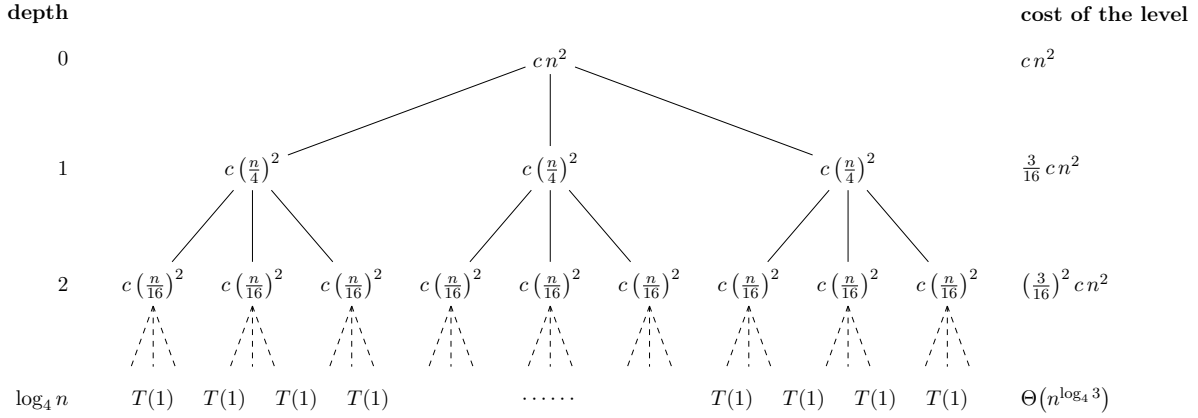


Figure 15.1: Recursion tree of $T(n) = 3T(n/4) + cn^2$: every node has three children, the subproblem size is divided by 4 at each level, and the leaves are at depth $\log_4 n$.

Example 15.9: Solving $T(n) = 3T(n/4) + cn^2$ with the recursion tree

- *Step 1 (sizes and nodes).* The subproblem size of each node at depth i is $\frac{n}{4^i}$. Each level has 3 times as many nodes as the previous level, so the number of nodes at depth i is 3^i .
- *Step 2 (cost of a level).* Each node at depth i ($i = 0, 1, 2, \dots, \log_4 n - 1$) has a cost of $c\left(\frac{n}{4^i}\right)^2$. Multiplying, the overall cost of all the nodes at depth i is

$$3^i c \left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2.$$

- *Step 3 (depth and leaves).* The subproblem size reaches 1 when $\frac{n}{4^i} = 1$, that is, when $i = \log_4 n$. Accordingly, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \dots, \log_4 n$). The bottom level has, using (15.6),

$$3^{\log_4 n} = n^{\log_4 3}$$

nodes (the leaves of the tree), each contributing a constant cost $T(1)$. The total cost of the bottom level is

$$n^{\log_4 3} T(1) = \Theta(n^{\log_4 3}).$$

- *Step 4 (sum).* Summing up the costs of all levels we obtain

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}). \end{aligned}$$

Using the geometric series (15.3) we obtain the exact value

$$T(n) = \frac{16}{13} \left(1 - \left(\frac{3}{16}\right)^{\log_4 n}\right) cn^2 + \Theta(n^{\log_4 3}).$$

Upper bound. It is simpler to bound the finite sum by the infinite decreasing geometric series (15.5):

$$\begin{aligned} T(n) &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i c n^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1 - \frac{3}{16}} c n^2 + \Theta\left(n^{\log_4 3}\right) \\ &= \frac{16}{13} c n^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2), \end{aligned}$$

since $\log_4 3 \approx 0.79 < 2$, so that $n^{\log_4 3} = O(n^2)$.

Lower bound. All the terms of the sum are non-negative, and the root alone costs $c n^2$. Hence

$$T(n) \geq c n^2 = \Omega(n^2).$$

Numerical check ($c = 1$, $T(1) = 1$): the recurrence gives $T(4) = 19$, $T(16) = 313$, $T(64) = 5035$, the values of the exact formula above, and $\frac{T(n)}{n^2} = 1.1875$, 1.2227 , 1.2292 , which approach $\frac{16}{13} \approx 1.2308$.

The solution of $T(n) = 3T(n/4) + c n^2$ is $T(n) = \Theta(n^2)$. The costs of the levels decrease geometrically: the total cost is dominated by the cost of the **root**.

15.4.2 Constant level costs: $T(n) = 2T(n/2) + c n$

- Consider the recurrence ($c > 0$ a constant), which describes algorithms such as MergeSort:

$$T(n) = 2T\left(\frac{n}{2}\right) + c n,$$

where n is an exact power of 2.

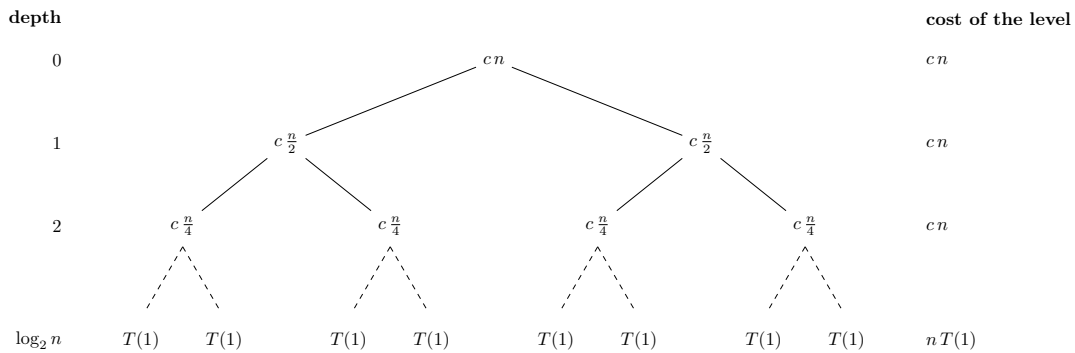


Figure 15.2: Recursion tree of $T(n) = 2T(n/2) + cn$: every level has total cost cn .

Example 15.10: Solving $T(n) = 2T(n/2) + cn$ with the recursion tree

- Steps 1 and 2.* At depth i there are 2^i nodes, each corresponding to a subproblem of size $\frac{n}{2^i}$ and with cost $c \frac{n}{2^i}$. The cost of each level is therefore

$$2^i c \frac{n}{2^i} = cn.$$

- *Step 3.* The subproblem size reaches 1 at depth $\log_2 n$. The bottom level has $2^{\log_2 n} = n$ leaves, with total cost $nT(1) = \Theta(n)$.
- *Step 4.* Summing the costs of all levels:

$$T(n) = \sum_{i=0}^{\log_2 n - 1} cn + nT(1) = cn \log_2 n + nT(1) = \Theta(n \log_2 n).$$

For $c = 1$ and $T(1) = 1$ we obtain again $T(n) = n \log_2 n + n$ (Example 15.7).

The solution of $T(n) = 2T(n/2) + cn$ is $T(n) = \Theta(n \log_2 n)$. All the levels have the **same cost** cn , and there are $\Theta(\log_2 n)$ levels.

15.4.3 An unbalanced tree: $T(n) = T(n/3) + T(2n/3) + cn$

- Consider the recurrence ($c, d > 0$ constants), defined for every real number $n > 0$:

$$T(n) = \begin{cases} d & \text{if } 0 < n < 1 \\ T(n/3) + T(2n/3) + cn & \text{if } n \geq 1 \end{cases}$$

(Working with real sizes allows us to avoid floors and ceilings.)

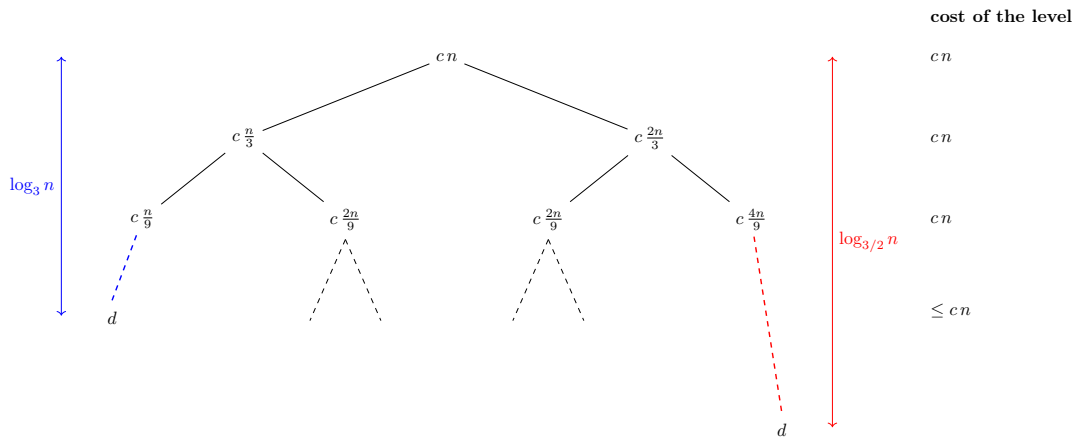


Figure 15.3: Recursion tree of $T(n) = T(n/3) + T(2n/3) + cn$: the shortest root-to-leaf path (blue) has length about $\log_3 n$, the longest (red) about $\log_{3/2} n$.

- Each internal node of size s has two children of sizes $\frac{s}{3}$ and $\frac{2s}{3}$, whose sum is s . Hence the sizes of the nodes at any given depth sum to at most n (exactly n if the level is complete).
- The **shortest path** from the root to a leaf is the leftmost one: $n \rightarrow \frac{n}{3} \rightarrow \frac{n}{9} \rightarrow \dots$. A node at depth i on this path has size $\frac{n}{3^i}$, which is ≥ 1 as long as $i \leq \log_3 n$.
- The **longest path** is the rightmost one: $n \rightarrow \frac{2n}{3} \rightarrow \frac{4n}{9} \rightarrow \dots$. A node at depth i on this path has size $(\frac{2}{3})^i n$, which is ≥ 1 as long as $i \leq \log_{3/2} n$.

- Unlike the previous examples, the tree is **not complete**: the leaves are at different depths, between $\log_3 n$ and $\log_{3/2} n$ (approximately).

Example 15.11: Bounds for the unbalanced tree

Upper bound.

- *Internal nodes.* Every node at depth i has size at most $(\frac{2}{3})^i n$; an internal node has size ≥ 1 , so internal nodes appear only at depths $i \leq \log_{3/2} n$. At each depth, the internal nodes have total size at most n , so their total cost is at most cn . Hence the cost of the internal nodes is at most

$$cn \left(\log_{3/2} n + 1 \right).$$

- *Leaves.* A leaf has size < 1 , but its parent has size ≥ 1 , so a leaf has size $\geq \frac{1}{3}$. The sizes of all the leaves sum to exactly n (each internal node splits its size among its children). Hence there are at most $3n$ leaves, with total cost at most $3dn = O(n)$.
- Therefore

$$T(n) \leq cn \left(\log_{3/2} n + 1 \right) + 3dn = O(n \log_2 n),$$

since $\log_{3/2} n = \frac{\log_2 n}{\log_2(3/2)}$ differs from $\log_2 n$ by a constant factor (15.6).

Lower bound.

- Every node at depth i has size at least $\frac{n}{3^i}$. Hence, for every depth $i \leq \log_3 n$, all the nodes at depth i have size ≥ 1 : they are internal nodes, the level is complete (2^i nodes) and their sizes sum to exactly n . The cost of each of these levels is exactly cn .
- There are $\lfloor \log_3 n \rfloor + 1 \geq \log_3 n$ such levels (depths $0, 1, \dots, \lfloor \log_3 n \rfloor$), so

$$T(n) \geq cn \log_3 n = \Omega(n \log_2 n).$$

Numerical check ($c = d = 1$): computing $T(n)$ exactly from the recurrence, $T(100) \approx 929.2$ and $T(1000) \approx 13\,002.3$; both values lie between the lower bound $n \log_3 n$ (419.2 and 6\,287.7) and the upper bound $n(\log_{3/2} n + 1) + 3n$ (1\,535.8 and 21\,036.6).

The solution of $T(n) = T(n/3) + T(2n/3) + cn$ is $T(n) = \Theta(n \log_2 n)$. Each level costs at most cn ; the upper bound uses the **longest path** ($\log_{3/2} n$) and the lower bound uses the **shortest path** ($\log_3 n$). Both are $\Theta(\log_2 n)$.

- **Remark.** If the tree were a complete binary tree of height $\log_{3/2} n$, it would have $2^{\log_{3/2} n} = n^{\log_{3/2} 2} \approx n^{1.71}$ leaves, which is $\omega(n \log_2 n)$. The argument above shows that the tree is far from complete: the actual number of leaves is $O(n)$. When the leaves are not easy to count, a safe approach is to use the recursion tree only to *guess* the solution and then to prove it with the substitution method.

15.4.4 Increasing level costs: $T(n) = 4T(n/2) + n$

Example 15.12: Solving $T(n) = 4T(n/2) + n$ with the recursion tree

Consider the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 4T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{cases}$$

where n is an exact power of 2.

- *Steps 1 and 2.* At depth i there are 4^i nodes, each of size $\frac{n}{2^i}$ and cost $\frac{n}{2^i}$. The cost of the level is

$$4^i \frac{n}{2^i} = 2^i n.$$

- *Step 3.* The size reaches 1 at depth $\log_2 n$, where there are $4^{\log_2 n} = n^2$ leaves:

depth	number of nodes	cost of each node	cost of the level
0	1	n	n
1	4	$n/2$	$2n$
2	16	$n/4$	$4n$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	4^i	$n/2^i$	$2^i n$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_2 n$	$4^{\log_2 n} = n^2$	$T(1) = 1$	n^2

- *Step 4.* The costs *increase* geometrically going down the tree. Using (15.4),

$$T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i n + n^2 = n \left(2^{\log_2 n} - 1 \right) + n^2 = n(n - 1) + n^2 = 2n^2 - n.$$

Hence $T(n) = 2n^2 - n = \Theta(n^2)$: indeed $\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0}$. Check: $T(2) = 4 + 2 = 6$, $T(4) = 24 + 4 = 28$, $T(8) = 112 + 8 = 120 = 2 \cdot 64 - 8$.

The solution of $T(n) = 4T(n/2) + n$ is $T(n) = \Theta(n^2)$. The costs of the levels increase geometrically: the total cost is dominated by the **leaves**, whose number is $4^{\log_2 n} = n^{\log_2 4} = n^2$.

15.5 Towards the Master Theorem

The three balanced trees of the previous section are instances of the same picture. We now draw the recursion tree of a general recurrence in standard form (15.1)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{if } n > 1 \end{cases}$$

with $a \geq 1$, $b > 1$ constants and n an exact power of b .

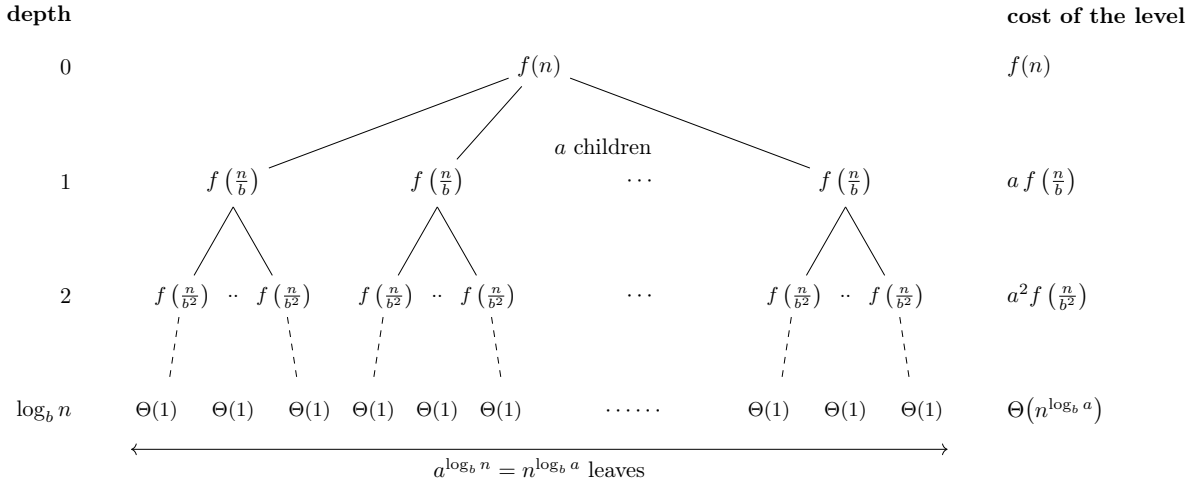


Figure 15.4: Recursion tree of a recurrence in standard form $T(n) = aT(n/b) + f(n)$, for n exact power of b .

- At depth j there are a^j nodes, each corresponding to a subproblem of size $\frac{n}{b^j}$ with cost $f\left(\frac{n}{b^j}\right)$. The cost of the level is

$$a^j f\left(\frac{n}{b^j}\right).$$

- The subproblem size reaches 1 at depth $\log_b n$. The number of leaves is, by (15.6),

$$a^{\log_b n} = n^{\log_b a},$$

and their total cost is $\Theta(n^{\log_b a})$.

- Summing the costs of all levels:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right). \quad (15.9)$$

- The solution depends on how the level costs $a^j f(n/b^j)$ evolve with j , that is, on the comparison between $f(n)$ (the cost of the root) and $n^{\log_b a}$ (the cost of the leaves). There are three possible behaviours, all of which we have already met:

1. **The leaves dominate:** the level costs increase geometrically going down the tree, and $T(n) = \Theta(n^{\log_b a})$.

Example: $T(n) = 4T(n/2) + n$, with $n^{\log_2 4} = n^2$ leaves, $T(n) = \Theta(n^2)$ (Example 15.12).

2. **All levels have (about) the same cost:** the cost of each of the $\log_b n + 1$ levels is $\Theta(n^{\log_b a})$, and $T(n) = \Theta(n^{\log_b a} \log_2 n)$.
Example: $T(n) = 2T(n/2) + cn$, with $n^{\log_2 2} = n$, $T(n) = \Theta(n \log_2 n)$ (Example 15.10).
3. **The root dominates:** the level costs decrease geometrically going down the tree, and $T(n) = \Theta(f(n))$.
Example: $T(n) = 3T(n/4) + cn^2$, with $f(n) = cn^2$, $T(n) = \Theta(n^2)$ (Example 15.9).

The **Master Theorem**, studied in the chapter *The Master Theorem*, makes these three behaviours precise. It compares $f(n)$ with $n^{\log_b a}$ and gives the solution of a recurrence in standard form directly, without drawing the recursion tree.

15.6 Summary: which method when

- **Expansion method:** recurrences with a *single* recursive term, for example $T(n - 1) + f(n)$, $2T(n - 1) + 1$, $T(n/2) + c$, $2T(n/2) + n$. It gives the exact closed form when the resulting summation can be computed (arithmetic or geometric series).
- **Substitution method:** when we have a guess and want a *rigorous* proof, in particular with floors and ceilings and explicit boundary conditions. Prove O and Ω separately; handle the base cases by choosing n_0 ; if the induction fails by a lower-order term, subtract a lower-order term from the guess. It can be combined with a change of variables.
- **Recursion-tree method:** to visualise the costs and to obtain a guess; ideal when the recursive terms have *different sizes* (unbalanced trees, as $T(n/3) + T(2n/3) + cn$). Compute the cost per level, the depth, the number of leaves, and sum using the geometric series.
- **Master Theorem** (chapter *The Master Theorem*): recurrences in *standard form* $aT(n/b) + f(n)$; it gives the answer directly by comparing $f(n)$ with $n^{\log_b a}$. It does not apply to $T(n - 1) + f(n)$ or to unbalanced splits.

The Master Theorem

Introduction

In the chapter *Recurrences* we solved recurrences by expansion, by substitution and with recursion trees. Each method requires some computation, and for divide-and-conquer recurrences the computation is always the same: draw the tree, sum the level costs, compare the root with the leaves. At the end of that chapter we observed three possible behaviours: the leaves dominate, all the levels cost the same, or the root dominates.

The **Master Theorem** turns these three behaviours into a ready-to-use result. In this chapter we

- recall the recurrences in standard form and the algorithms that produce them (Section 16.1);
- state the theorem and explain its three cases (Section 16.2);
- apply it with a fixed five-step scheme to many examples (Section 16.3), and show the situations in which it cannot be applied (Section 16.4);
- prove it (Section 16.5), first for exact powers of b and then for every natural number.

16.1 Recurrences in standard form

- In the chapter *Recurrences* we have seen that the number of steps executed by a recursive algorithm is naturally described by a **recurrence**, i.e., an equation that expresses the value of a function on an input of size n in terms of its values on smaller inputs.
- A large family of *divide-and-conquer* algorithms produces recurrences of a very specific shape, called the **standard form**:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- n is the size of the input;
- a is the **number of subproblems** generated by each recursive call ($a \geq 1$, constant);

- $\frac{n}{b}$ is the **size of each subproblem** ($b > 1$, constant); when n/b is not a natural number it is understood as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$;
- $f(n)$ is the **cost of dividing** the problem into the a subproblems **and of combining** their solutions (everything that is done outside the recursive calls). We always assume that $f(n)$ is **asymptotically positive**, i.e., $f(n) > 0$ for all sufficiently large n .
- For sufficiently small n (for example $n = 1$) the recursion stops and $T(n) = \Theta(1)$: solving a problem of constant size requires a constant number of steps. For this reason the base case is usually not even written.

In this chapter a recurrence is a **mathematical object**: we use a generic function $T(n)$. When we analyse a recursive algorithm, the function described by the recurrence is the number of steps $S(n)$ executed in the worst case on an input of size parameter n ; the worst-case running time $T(|I|)$ is then obtained from $S(n)$ exactly as in the chapter *Running time*.

- Examples of recurrences in standard form (they will all be solved in Section 16.3):

Algorithm	Recurrence	a	b	$f(n)$
Binary search (recursive)	$T(n) = T(n/2) + 1$	1	2	1
Array sum (recursive)	$T(n) = 2T(n/2) + 1$	2	2	1
Merge sort	$T(n) = 2T(n/2) + n$	2	2	n
Karatsuba multiplication	$T(n) = 3T(n/2) + n$	3	2	n
Strassen matrix multiplication	$T(n) = 7T(n/2) + n^2$	7	2	n^2

- The **Master Theorem** gives the order of growth of $T(n)$ for most recurrences in standard form, directly from a , b and $f(n)$, without any computation of the recursion.

16.2 The Master Theorem

Theorem 16.1: Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be defined on the natural numbers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where n/b is interpreted as either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, and $T(n) = \Theta(1)$ for sufficiently small n . Then $T(n)$ has the following asymptotic bounds.

Case 1: if $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then

$$T(n) = \Theta(n^{\log_b a});$$

Case 2: if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n);$$

Case 3: if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if

$$a f\left(\frac{n}{b}\right) \leq c f(n)$$

for some constant $c < 1$ and all sufficiently large n (**regularity condition**), then

$$T(n) = \Theta(f(n)).$$

- This is the classical statement of the theorem (as in the textbook of Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*), with the **extended case 2**: for $k = 0$ we obtain the basic statement “if $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$ ”.
- The notations O , Ω and Θ in the statement are those defined in chapter 9 «Asymptotic notation».
- The proof is given in Section 16.5.

16.2.1 Rule of thumb

- In each of the three cases we compare the function $f(n)$ with the function

$$n^{\log_b a},$$

which (as we will see in Lemma 16.3) is the number of leaves of the recursion tree. As a rule of thumb, check:

$$n^{\log_b a} \begin{cases} \text{polynomially larger than } f(n) & \longrightarrow \text{case 1: } T(n) = \Theta(n^{\log_b a}) \\ \text{of the same order as } f(n) & \longrightarrow \text{case 2: } T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \\ \text{(up to } \log^k n) & \\ \text{polynomially smaller than } f(n) & \longrightarrow \text{case 3: } T(n) = \Theta(f(n)) \end{cases}$$

and determine the order of growth of $T(n)$ accordingly (if possible).

- **Intuitively, the larger of the two functions determines the solution of the recurrence.**
- **Polynomially** smaller (larger) means smaller (larger) by a factor n^ϵ for some constant $\epsilon > 0$. For example:
 - n is polynomially smaller than n^2 (factor n^1), and n^2 is polynomially smaller than $n^{2.1}$ (factor $n^{0.1}$);
 - n is smaller than $n \log n$, but **not** polynomially smaller: for every $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^{1+\epsilon}} = \lim_{n \rightarrow \infty} \frac{\log n}{n^\epsilon} = 0^+,$$

hence $n \log n$ grows more slowly than $n \cdot n^\epsilon$, whatever $\epsilon > 0$ we choose.

16.2.2 Intuition

- Unfolding the recurrence (see the recursion tree in Figure 16.1) the total cost is

$$T(n) = \underbrace{\Theta(n^{\log_b a})}_{\text{cost of the leaves}} + \underbrace{\sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)}_{\text{cost of the internal levels}}$$

where level j of the tree contains a^j subproblems of size n/b^j .

- Consider the simple case $f(n) = n^d$. The cost of level j is

$$a^j \left(\frac{n}{b^j}\right)^d = n^d \left(\frac{a}{b^d}\right)^j,$$

so the costs of the levels form a **geometric progression with ratio** a/b^d . Since $a/b^d > 1 \iff d < \log_b a$:

- **case 1** ($d < \log_b a$): the level costs **increase** geometrically from the root to the leaves; the total is dominated by the leaves, i.e., by $n^{\log_b a}$;
- **case 2** ($d = \log_b a$): all the $\log_b n$ levels have **the same cost** $n^d = n^{\log_b a}$; the total is $n^{\log_b a}$ times the number of levels, i.e., $n^{\log_b a} \log n$;
- **case 3** ($d > \log_b a$): the level costs **decrease** geometrically; the total is dominated by the root, i.e., by $f(n)$.

16.2.3 The gaps between the cases

- The three cases **do not cover all the possible functions** $f(n)$.
 - There is a gap between case 1 and case 2 when $f(n)$ is smaller than $n^{\log_b a}$ but **not polynomially smaller** (and not of the form $n^{\log_b a} \log^k n$ with $k \geq 0$), e.g., $f(n) = n^{\log_b a} / \log n$.
 - There is a gap between case 2 and case 3 when $f(n)$ is larger than $n^{\log_b a}$ but **not polynomially larger** (and not of the form $n^{\log_b a} \log^k n$), e.g., $f(n) = n^{\log_b a} 2^{\sqrt{\log_2 n}}$.
 - Case 3 does not apply also when $f(n)$ is polynomially larger than $n^{\log_b a}$ but the **regularity condition** fails.
- In these situations the Master Theorem **cannot be used**, and the recurrence must be solved with other techniques (recursion tree, iteration, substitution method), see Section 16.4.

16.2.4 A remark on the base of the logarithms

- Since $\log_c n = \frac{\log_2 n}{\log_2 c}$ for every base $c > 1$, we have

$$\log_c^k n = \frac{1}{\log_2^k c} \log_2^k n = \Theta(\log_2^k n).$$

Thus, **inside Θ , O and Ω the base of the logarithm is irrelevant**: in case 2 we simply write $\log^k n$.

- On the contrary, the base b in the **exponent** $n^{\log_b a}$ is essential: for instance $n^{\log_2 4} = n^2$ whereas $n^{\log_4 4} = n$, and $n^2 \neq \Theta(n)$.

- The Master Theorem compares the cost of the root, $f(n)$, with the number of leaves of the recursion tree, $n^{\log_b a}$: the larger of the two determines $T(n)$; when they are of the same order (up to $\log^k n$) one more factor $\log n$ appears.
- “Larger” and “smaller” must be **polynomial** (by a factor n^ϵ); in case 3 the **regularity condition** must also hold.
- When $f(n)$ falls in a gap between the cases, the theorem gives no answer.

16.3 Applying the Master Theorem

To solve a recurrence with the Master Theorem we always follow this scheme:

1. **identify** a , b and $f(n)$ (and check that $a \geq 1$, $b > 1$ are constants and $f(n)$ is asymptotically positive);
2. **compute** $n^{\log_b a}$;
3. **compare** $n^{\log_b a}$ with $f(n)$ and guess the case;
4. **verify** the hypothesis of the case: find $\epsilon > 0$ (case 1 or 3), find $k \geq 0$ (case 2), and check the regularity condition (case 3);
5. **conclude**.

We start with three recurrences, one for each case, and then solve the recurrences of the algorithms listed in Section 16.1.

Example 16.1: $T(n) = 16T(n/4) + n$

1. $a = 16$, $b = 4$, $f(n) = n$.
2. $n^{\log_b a} = n^{\log_4 16} = n^2$.
3. n^2 is polynomially larger than $f(n) = n \rightarrow$ **case 1**.
4. $f(n) = n = O(n^{2-\epsilon})$ with $\epsilon = 1$.

$$5. T(n) = \Theta(n^{\log_4 16}) = \Theta(n^2).$$

Example 16.2: $T(n) = 8T(n/2) + n^3$

1. $a = 8, b = 2, f(n) = n^3$.
2. $n^{\log_b a} = n^{\log_2 8} = n^3$.
3. n^3 and $f(n) = n^3$ have the same order $\rightarrow$ **case 2**.
4. $f(n) = n^3 = \Theta(n^3 \log^k n)$ with $k = 0$.
5. $T(n) = \Theta(n^3 \log^{k+1} n) = \Theta(n^3 \log n)$.

Example 16.3: $T(n) = 8T(n/3) + n^2$

1. $a = 8, b = 3, f(n) = n^2$.
2. $n^{\log_b a} = n^{\log_3 8} \approx n^{1.893}$.
3. $n^{1.893}$ is polynomially smaller than $f(n) = n^2 \rightarrow$ **case 3**.
4. $f(n) = n^2 = \Omega(n^{\log_3 8 + \epsilon})$ for every $0 < \epsilon \leq 2 - \log_3 8 \approx 0.107$.
Regularity condition: we look for a constant $c < 1$ such that, for all large n ,

$$8f\left(\frac{n}{3}\right) \leq cf(n) \iff 8\left(\frac{n}{3}\right)^2 \leq cn^2 \iff \frac{8}{9}n^2 \leq cn^2,$$

which holds for every c with $\frac{8}{9} \leq c < 1$, e.g., $c = \frac{8}{9}$.

$$5. T(n) = \Theta(n^2).$$

Example 16.4: Binary search: $T(n) = T(n/2) + 1$

1. $a = 1, b = 2, f(n) = 1$.
2. $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$.
3. 1 and $f(n) = 1$ have the same order $\rightarrow$ **case 2**.
4. $f(n) = 1 = \Theta(n^0 \log^k n)$ with $k = 0$.
5. $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.

Example 16.5: Merge sort: $T(n) = 2T(n/2) + n$

1. $a = 2, b = 2, f(n) = n$.
2. $n^{\log_b a} = n^{\log_2 2} = n$.

3. n and $f(n) = n$ have the same order $\rightarrow$ **case 2**.
4. $f(n) = n = \Theta(n \log^k n)$ with $k = 0$.
5. $T(n) = \Theta(n \log n)$.

Example 16.6: Array sum: $T(n) = 2T(n/2) + 1$

1. $a = 2, b = 2, f(n) = 1$.
2. $n^{\log_b a} = n^{\log_2 2} = n$.
3. n is polynomially larger than $f(n) = 1 \rightarrow$ **case 1**.
4. $f(n) = 1 = O(n^{1-\epsilon})$ with $\epsilon = 1$ (indeed $n^{1-1} = n^0 = 1$).
5. $T(n) = \Theta(n)$.

The C++ function `sumRecHalving` of the code of the course is an algorithm of this kind: it sums the sub-array `a[l..r]` (indices from 0) by splitting it into two halves.

```

1 long long sumRecHalving(int *a, int l, int r)
2 {
3     if (l == r)
4     {
5         return a[l];
6     }
7
8     int mid = l + (r - l) / 2;
9
10    return sumRecHalving(a, l, mid) + sumRecHalving(a, mid + 1, r);
11 }
```

Each call does a constant amount of work outside the recursive calls, and the total number of calls is $\Theta(n)$: counting them with an instrumented copy of the function, on $n = 8, 1000$ and $2^{20} = 1\,048\,576$ entries we obtain 15, 1999 and 2097151 calls, that is, exactly $2n - 1$.

Example 16.7: Karatsuba: $T(n) = 3T(n/2) + n$

1. $a = 3, b = 2, f(n) = n$.
2. $n^{\log_b a} = n^{\log_2 3} \approx n^{1.585}$.
3. $n^{1.585}$ is polynomially larger than $f(n) = n \rightarrow$ **case 1**.
4. $f(n) = n = O(n^{\log_2 3 - \epsilon})$ with $\epsilon = \log_2 3 - 1 \approx 0.585 > 0$.
5. $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$: Karatsuba's algorithm is asymptotically faster than the schoolbook $\Theta(n^2)$ multiplication.

Example 16.8: Strassen: $T(n) = 7T(n/2) + n^2$

1. $a = 7, b = 2, f(n) = n^2$.
2. $n^{\log_b a} = n^{\log_2 7} \approx n^{2.807}$.
3. $n^{2.807}$ is polynomially larger than $f(n) = n^2 \rightarrow$ **case 1**.
4. $f(n) = n^2 = O(n^{\log_2 7 - \epsilon})$ with $\epsilon = \log_2 7 - 2 \approx 0.807 > 0$.
5. $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$, better than the $\Theta(n^3)$ of the standard matrix multiplication (whose divide-and-conquer recurrence is $T(n) = 8T(n/2) + n^2$).

Example 16.9: $T(n) = 2T(n/2) + n \log n$

1. $a = 2, b = 2, f(n) = n \log n$.
2. $n^{\log_b a} = n^{\log_2 2} = n$.
3. $f(n) = n \log n$ is larger than n , but only by a factor $\log n$, not polynomially $\rightarrow$ not case 3; it has the form $n^{\log_b a} \log^k n \rightarrow$ **case 2**.
4. $f(n) = n \log n = \Theta(n \log^k n)$ with $k = 1$.
5. $T(n) = \Theta(n \log^2 n)$.

Example 16.10: $T(n) = 3T(n/4) + n \log n$

1. $a = 3, b = 4, f(n) = n \log n$ (logarithm in base 2).
2. $n^{\log_b a} = n^{\log_4 3} \approx n^{0.792}$.
3. $n^{0.792}$ is polynomially smaller than $f(n) = n \log n \rightarrow$ **case 3**.
4. Since $n \log n \geq n$ for $n \geq 2$, we have $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ with $\epsilon = 1 - \log_4 3 \approx 0.208$.
Regularity condition: for every $n \geq 4$,

$$3f\left(\frac{n}{4}\right) = 3 \cdot \frac{n}{4} \log \frac{n}{4} \leq \frac{3}{4} n \log n = c f(n) \quad \text{with } c = \frac{3}{4} < 1,$$

because $\log(n/4) \leq \log n$.

5. $T(n) = \Theta(n \log n)$.

Example 16.11: $T(n) = T(2n/3) + 1$

1. $a = 1, b = \frac{3}{2}$ (indeed $\frac{2n}{3} = \frac{n}{3/2}$), $f(n) = 1$. Note that b does not need to be an integer.
2. $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
3. 1 and $f(n) = 1$ have the same order $\rightarrow$ **case 2**.

4. $f(n) = 1 = \Theta(n^0 \log^k n)$ with $k = 0$.
5. $T(n) = \Theta(\log n)$.

- A **numerical check**. For each of the recurrences above with b integer we computed $T(n)$ exactly from the recurrence, with $T(1) = 1$ and $n = b^L$, and divided it by the order of growth $g(n)$ given by the theorem (logarithms in base 2). The ratio stays bounded between two positive constants, as the theorem guarantees, and approaches the limit in the last column.

Recurrence	$g(n)$	$L = 5$	$L = 10$	$L = 20$	limit
$16T(n/4) + n$	n^2	1.3330	1.3333	1.3333	4/3
$8T(n/2) + n^3$	$n^3 \log n$	1.2000	1.1000	1.0500	1
$8T(n/3) + n^2$	n^2	4.5606	6.5364	8.2414	9
$T(n/2) + 1$	$\log n$	1.2000	1.1000	1.0500	1
$2T(n/2) + n$	$n \log n$	1.2000	1.1000	1.0500	1
$2T(n/2) + 1$	n	1.9688	1.9990	2.0000	2
$3T(n/2) + n$	$n^{\log_2 3}$	2.7366	2.9653	2.9994	3
$7T(n/2) + n^2$	$n^{\log_2 7}$	2.2521	2.3284	2.3333	7/3
$2T(n/2) + n \log n$	$n \log^2 n$	0.6400	0.5600	0.5275	1/2
$3T(n/4) + n \log n$	$n \log n$	2.1933	2.8704	3.4020	4

- For merge sort, for instance, $T(2^L) = 2^L(L+1)$, so $T(n) = n \log_2 n + n$ and $\frac{T(n)}{n \log_2 n} = 1 + \frac{1}{L}$. In case 3 the convergence is slow: the ratio tends to $\frac{1}{1-c}$ (9 for $c = \frac{8}{9}$, 4 for $c = \frac{3}{4}$), the bound that appears in the proof of the theorem.

The theorem gives the **order of growth**, not the constant: in the table the ratios tend to different constants, but they never grow or vanish.

16.4 When the Master Theorem does not apply

- Before applying the theorem, always check that **all** its hypotheses hold. Here are typical situations in which the theorem **cannot** be applied.

Example 16.12: A gap between case 1 and case 2: $T(n) = 2T(n/2) + n/\log_2 n$

- $a = 2$, $b = 2$, $f(n) = n/\log_2 n$, and $n^{\log_b a} = n$.
- **Not case 1:** $f(n)$ is smaller than n , but not polynomially smaller. For every $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n/\log_2 n}{n^{1-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log_2 n} = +\infty,$$

hence $f(n) \neq O(n^{1-\epsilon})$.

- **Not case 2:** $f(n) = n \log^{-1} n$ would require $k = -1 < 0$.
- **Not case 3:** $f(n)$ is smaller than n .

- **Solution with the recursion tree** (Lemma 16.3), for $n = 2^L$ and $T(1) = 1$: level j costs

$$2^j \frac{n/2^j}{\log_2(n/2^j)} = \frac{n}{L-j}, \quad \text{so} \quad T(n) = n + \sum_{j=0}^{L-1} \frac{n}{L-j} = n + n \sum_{i=1}^L \frac{1}{i} = n(1 + H_L),$$

where $H_L = \sum_{i=1}^L \frac{1}{i}$ is the L -th harmonic number. Since $\ln(L+1) \leq H_L \leq 1 + \ln L$ (comparison with $\int \frac{dx}{x}$), $H_L = \Theta(\log L)$ and

$$T(n) = \Theta(n \log \log n).$$

Example 16.13: Not in standard form: $T(n) = T(n-1) + n$

- The subproblem has size $n-1$, not n/b with a constant $b > 1$: the recurrence is **not in standard form**.
- Solution by iteration (with $T(1) = 1$):

$$T(n) = n + (n-1) + \cdots + 2 + T(1) = \sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2).$$

Example 16.14: Subproblems of different sizes: $T(n) = T(n/3) + T(2n/3) + n$

- The two subproblems have **different sizes** $n/3$ and $2n/3$: the recurrence is not of the form $aT(n/b) + f(n)$.
- Recursion tree: every complete level has total size n , hence cost n . The shortest root-to-leaf path (always $n/3$) has length $\log_3 n$, the longest (always $2n/3$) has length $\log_{3/2} n$. Therefore the first $\log_3 n$ levels are complete (cost $\geq n \log_3 n$), and there are at most $\log_{3/2} n + 1$ levels, each of cost at most n . Hence

$$T(n) = \Theta(n \log n).$$

Example 16.15: a is not constant: $T(n) = 2^n T(n/2) + n^n$

- The number of subproblems $a = 2^n$ **depends on** n : the Master Theorem requires a to be a constant.

Example 16.16: $a < 1$: $T(n) = 0.5T(n/2) + 1/n$

- Here $a = 0.5 < 1$: the theorem requires $a \geq 1$ (there must be at least one subproblem).

Example 16.17: The regularity condition fails: $T(n) = T(n/2) + n(2 - \cos n)$

- $a = 1$, $b = 2$, $n^{\log_b a} = n^0 = 1$, $f(n) = n(2 - \cos n)$. Since $1 \leq 2 - \cos n \leq 3$, we have $n \leq f(n) \leq 3n$, so

$$f(n) = \Theta(n) = \Omega(n^{0+\epsilon}) \quad \text{with } \epsilon = 1 :$$

$f(n)$ is polynomially larger than $n^{\log_b a}$, and we would expect case 3.

- **The regularity condition fails.** Take an even $n = 2m$. Then

$$\frac{a f(n/2)}{f(n)} = \frac{m(2 - \cos m)}{2m(2 - \cos 2m)} = \frac{2 - \cos m}{2(2 - \cos 2m)}.$$

If $\cos m \leq -0.9$, then $2 - \cos m \geq 2.9$ and $\cos 2m = 2\cos^2 m - 1 \geq 2 \cdot 0.81 - 1 = 0.62$, so $2(2 - \cos 2m) \leq 2.76$ and the ratio is > 1 . Since π is irrational, the values $m \bmod 2\pi$ ($m \in \mathbb{N}$) are dense in $[0, 2\pi)$, so there are **infinitely many** m with $\cos m \leq -0.9$ (e.g., $m = 3, 9, 16, 22, 28, \dots$). For $m = 22$: $\cos 22 \approx -0.99996$, $\cos 44 \approx 0.99984$, and the ratio is ≈ 1.4997 .

Hence there is no constant $c < 1$ such that $af(n/2) \leq cf(n)$ for all large n : **case 3 cannot be applied.**

- Nevertheless the recurrence can be solved directly (with $n/2$ meaning $\lfloor n/2 \rfloor$): unfolding it,

$$n \leq f(n) \leq T(n) \leq \sum_{j \geq 0} 3 \frac{n}{2^j} + T(1) = 6n + T(1), \quad \text{so} \quad T(n) = \Theta(n).$$

16.5 Proof of the Master Theorem

- We follow the classical proof, which has two parts:
 1. we first assume that n is an **exact power of b** , i.e., $n = b^L$ for some $L \in \mathbb{N}$; then all the subproblem sizes $n/b, n/b^2, \dots$ are exact powers of b and no floor or ceiling is needed (Lemmas 16.3, 16.4 and 16.5);
 2. we then extend the result to every natural number n , when n/b is $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$ (Section 16.5.5).

16.5.1 Setting

- Throughout the proof we set

$$p = \log_b a, \quad \text{so that} \quad b^p = a \quad \text{and} \quad p \geq 0 \quad (\text{since } a \geq 1).$$

- For n exact power of b we consider the recurrence

$$T(n) = \begin{cases} d & \text{if } n = 1, \\ aT(n/b) + f(n) & \text{if } n = b^i, i \geq 1, \end{cases} \quad (16.1)$$

where $d > 0$ is a constant (this is the meaning of $T(1) = \Theta(1)$).

- We assume $f(m) > 0$ for every $m \geq 1$. This is always true when f counts steps of an algorithm; in general, since f is asymptotically positive, only finitely many values are affected and only the constants of the proof change.
- The hypotheses of the three cases hold “for all sufficiently large n ”, i.e., for $n \geq n_0$. The following observation allows us to use them for **every** exact power of b at least b .

Observation 16.2: from large powers to all powers

Let $f(m) > 0$ and $h(m) > 0$ for every exact power $m \geq b$ of b . If there are constants $c > 0$ and n_0 such that $f(m) \leq ch(m)$ (resp. $f(m) \geq ch(m)$) for all exact powers $m \geq n_0$, then there is a constant $c' > 0$ such that $f(m) \leq c'h(m)$ (resp. $f(m) \geq c'h(m)$) for **all** exact powers $m \geq b$ of b .

Proof. There are only finitely many exact powers of b in the interval $[b, n_0)$. Let

$$c' = \max \left\{ c, \max_{b \leq m < n_0} \frac{f(m)}{h(m)} \right\} \quad \left(\text{resp. } c' = \min \left\{ c, \min_{b \leq m < n_0} \frac{f(m)}{h(m)} \right\} > 0 \right),$$

where m ranges over exact powers of b . Then $f(m) \leq c'h(m)$ (resp. $\geq$) both for $m < n_0$, by the choice of c' , and for $m \geq n_0$, since $c \leq c'$ (resp. $c \geq c'$). $\square$

- We will apply it with $h(m) = m^{p-\epsilon}$ and $h(m) = m^p \log_b^k m$, which are positive for $m \geq b$ (note $\log_b m \geq 1$ for $m \geq b$).

16.5.2 The recursion tree

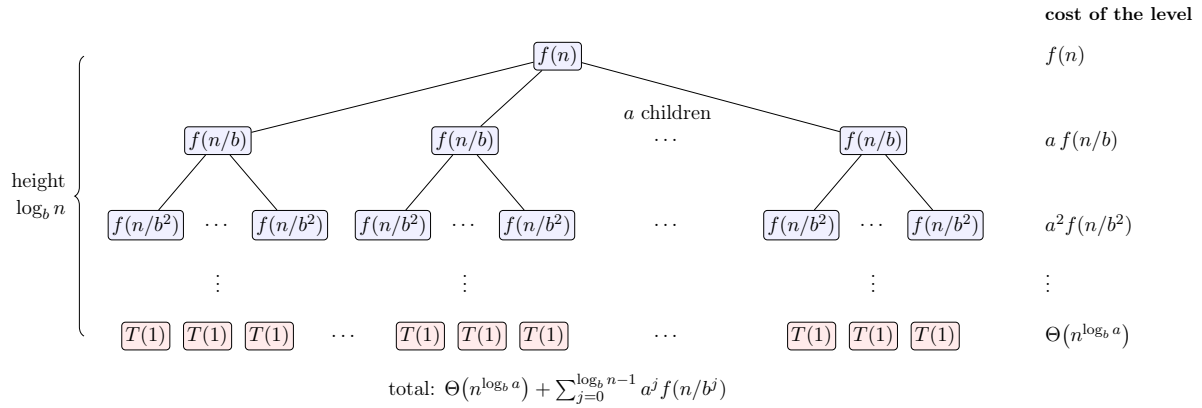


Figure 16.1: Recursion tree of $T(n) = aT(n/b) + f(n)$ for n exact power of b . Each internal node has a children; level j has a^j nodes, each a subproblem of size n/b^j with cost $f(n/b^j)$; the leaves are at depth $\log_b n$.

- The root (level 0) is a problem of size n ; it costs $f(n)$ and generates a subproblems of size n/b .
- By induction, level j contains a^j nodes, each a subproblem of size n/b^j and cost $f(n/b^j)$: the total cost of level j is $a^j f(n/b^j)$.
- The subproblem size reaches 1 when $n/b^j = 1$, i.e., at depth $j = \log_b n$: the leaves are the $a^{\log_b n}$ nodes of level $\log_b n$, each with cost $T(1) = d$.

Lemma 16.3: Cost of the recursion tree

Let $a \geq 1$ and $b > 1$ be constants and let $f(n) > 0$ for $n \geq 1$. For every n exact power of b , the solution of recurrence (16.1) is

$$T(n) = d n^{\log_b a} + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right).$$

Proof. Let $n = b^L$. We first prove by induction on $i \in \{0, 1, \dots, L\}$ that

$$T(n) = a^i T\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j f\left(\frac{n}{b^j}\right). \quad (16.2)$$

For $i = 0$ the sum is empty and (16.2) reads $T(n) = T(n)$. Assume (16.2) holds for some $i < L$. Since $n/b^i = b^{L-i}$ with $L - i \geq 1$, the recurrence gives $T(n/b^i) = a T(n/b^{i+1}) + f(n/b^i)$, so

$$T(n) = a^i \left(a T\left(\frac{n}{b^{i+1}}\right) + f\left(\frac{n}{b^i}\right) \right) + \sum_{j=0}^{i-1} a^j f\left(\frac{n}{b^j}\right) = a^{i+1} T\left(\frac{n}{b^{i+1}}\right) + \sum_{j=0}^i a^j f\left(\frac{n}{b^j}\right),$$

which is (16.2) for $i + 1$. For $i = L = \log_b n$ we have $T(n/b^L) = T(1) = d$, hence

$$T(n) = d a^{\log_b n} + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right).$$

It remains to show that $a^{\log_b n} = n^{\log_b a}$. Writing $a = b^{\log_b a}$ and $n = b^{\log_b n}$,

$$a^{\log_b n} = (b^{\log_b a})^{\log_b n} = b^{\log_b a \cdot \log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}.$$

Finally, $d > 0$ is a constant, so $d n^{\log_b a} = \Theta(n^{\log_b a})$. □

- The first term is the cost of the **leaves**: there are exactly $n^{\log_b a}$ leaves. The sum is the cost of the **internal nodes** (levels $0, \dots, \log_b n - 1$).

Example 16.18: Lemma 16.3 for $T(n) = 2T(n/2) + n$, $T(1) = 1$, $n = 8$

Here $a = 2$, $b = 2$, $f(n) = n$, $d = 1$, $\log_2 8 = 3$.

level j	nodes a^j	size n/b^j	cost per node	cost of the level
0	1	8	$f(8) = 8$	8
1	2	4	$f(4) = 4$	8
2	4	2	$f(2) = 2$	8
3 (leaves)	$8 = 8^{\log_2 2}$	1	$T(1) = 1$	8
total				32

Lemma 16.3 gives $T(8) = 1 \cdot 8^{\log_2 2} + (8 + 8 + 8) = 8 + 24 = 32$. Directly from the recurrence:

$$T(2) = 2 \cdot 1 + 2 = 4, \quad T(4) = 2 \cdot 4 + 4 = 12, \quad T(8) = 2 \cdot 12 + 8 = 32. \quad \checkmark$$

16.5.3 Bounding the sum

- By Lemma 16.3, to solve the recurrence we need the order of growth of

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right).$$

- A key identity, used many times: since $b^p = a$,

$$a^j \left(\frac{n}{b^j}\right)^p = \frac{a^j}{(b^p)^j} n^p = \frac{a^j}{a^j} n^p = n^p \quad \text{for every } j. \quad (16.3)$$

- In the sum, $j \leq \log_b n - 1$, so the sizes $m = n/b^j$ are exact powers of b with $m \geq b$: by Observation 16.2, the bounds on f of cases 1 and 2 can be used for all of them.

Lemma 16.4: Bounds on the sum

Let $a \geq 1$ and $b > 1$ be constants, $p = \log_b a$, let $f(m) > 0$ for $m \geq 1$, and let $g(n)$ be defined, for n exact power of b , as above. Then:

1. if $f(n) = O(n^{p-\epsilon})$ for some constant $\epsilon > 0$, then $g(n) = O(n^p)$;
2. if $f(n) = \Theta(n^p \log^k n)$ for some constant $k \geq 0$, then $g(n) = \Theta(n^p \log^{k+1} n)$;
3. if $f(n) = \Omega(n^{p+\epsilon})$ for some constant $\epsilon > 0$, and $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all exact powers $n \geq n_0$, then $g(n) = \Theta(f(n))$.

Proof of case 1. By Observation 16.2 there is a constant $c > 0$ such that $f(m) \leq c m^{p-\epsilon}$ for every exact power $m \geq b$. Hence, using (16.3),

$$g(n) \leq c \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{p-\epsilon} = c \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^p \left(\frac{b^j}{n}\right)^\epsilon = c n^{p-\epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j.$$

This is a finite geometric series with ratio $b^\epsilon > 1$ (since $b > 1$ and $\epsilon > 0$):

$$\sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j = \frac{(b^\epsilon)^{\log_b n} - 1}{b^\epsilon - 1} = \frac{(b^{\log_b n})^\epsilon - 1}{b^\epsilon - 1} = \frac{n^\epsilon - 1}{b^\epsilon - 1}.$$

Therefore

$$0 < g(n) \leq c n^{p-\epsilon} \frac{n^\epsilon - 1}{b^\epsilon - 1} \leq \frac{c}{b^\epsilon - 1} n^p,$$

and since $\frac{c}{b^\epsilon - 1}$ is a positive constant, $g(n) = O(n^p)$. □

Example 16.19: Case 1 of Lemma 16.4: $T(n) = 16T(n/4) + n$, $n = 64$

Here $p = \log_4 16 = 2$, $f(m) = m = m^{2-1}$, so $c = 1$, $\epsilon = 1$. The internal levels cost

$$g(64) = 1 \cdot 64 + 16 \cdot 16 + 256 \cdot 4 = 64 + 256 + 1024 = 1344 :$$

the costs grow by the factor $b^\epsilon = 4$ from one level to the next (they are dominated by the deepest levels). The formula of the proof gives exactly $n^{p-\epsilon} \frac{n^\epsilon - 1}{b^\epsilon - 1} = 64 \cdot \frac{63}{3} = 1344$, and the bound $\frac{c}{b^\epsilon - 1} n^p = \frac{4096}{3} \approx 1365.3$ holds. Adding the leaves, $T(64) = 4096 + 1344 = 5440$ with $T(1) = 1$.

Proof of case 2. Since $\log^k n$ and $\log_b^k n$ differ by a constant factor, the hypothesis is $f(n) = \Theta(n^p \log_b^k n)$. By Observation 16.2 there are constants $c_1, c_2 > 0$ such that

$$c_1 m^p \log_b^k m \leq f(m) \leq c_2 m^p \log_b^k m \quad \text{for every exact power } m \geq b.$$

Let $L = \log_b n$. For the term j of the sum, $m = n/b^j$ and $\log_b m = L - j$; by (16.3),

$$a^j \left(\frac{n}{b^j}\right)^p \log_b^k \frac{n}{b^j} = n^p (L - j)^k.$$

Hence, setting $i = L - j$ (as j goes from 0 to $L - 1$, i goes from L down to 1),

$$c_1 n^p S_k(L) \leq g(n) \leq c_2 n^p S_k(L), \quad \text{where} \quad S_k(L) = \sum_{i=1}^L i^k.$$

We now prove that $(\frac{L}{2})^{k+1} \leq S_k(L) \leq L^{k+1}$, i.e., $S_k(L) = \Theta(L^{k+1})$.

- *Upper bound:* each of the L terms satisfies $i^k \leq L^k$ (since $k \geq 0$), so $S_k(L) \leq L \cdot L^k = L^{k+1}$.
- *Lower bound:* keep only the terms with $i \geq \lceil L/2 \rceil$ (all terms are nonnegative). There are $L - \lceil L/2 \rceil + 1 \geq L/2$ of them and each is at least $(L/2)^k$. Hence $S_k(L) \geq \frac{L}{2} (\frac{L}{2})^k = (\frac{L}{2})^{k+1}$.

Therefore

$$\frac{c_1}{2^{k+1}} n^p \log_b^{k+1} n \leq g(n) \leq c_2 n^p \log_b^{k+1} n,$$

that is, $g(n) = \Theta(n^p \log_b^{k+1} n) = \Theta(n^p \log^{k+1} n)$. □

Example 16.20: Case 2 of Lemma 16.4: $2T(n/2) + n$ and $2T(n/2) + n \log_2 n$, $n = 16$

Here $p = \log_2 2 = 1$ and $L = \log_2 16 = 4$.

- $f(n) = n$ ($k = 0$): every level costs $2^j \cdot \frac{16}{2^j} = 16 = n^p$, so $g(16) = 4 \cdot 16 = 64 = n S_0(4)$, with $S_0(4) = 4 = L$.
- $f(n) = n \log_2 n$ ($k = 1$): level j costs $n^p(L - j) = 16(4 - j)$:

$$g(16) = 64 + 48 + 32 + 16 = 160 = 16(4 + 3 + 2 + 1) = n S_1(4).$$

The bounds of the proof give $(\frac{L}{2})^2 = 4 \leq S_1(4) = 10 \leq L^2 = 16$, hence $64 \leq g(16) \leq 256$.

Proof of case 3. Lower bound. All terms of $g(n)$ are positive and the term $j = 0$ is $f(n)$: hence $g(n) \geq f(n)$.

Upper bound. Let $L = \log_b n$. We first prove, by induction on j , that

$$a^j f\left(\frac{n}{b^j}\right) \leq c^j f(n) \quad \text{for every } j \geq 0 \text{ such that } \frac{n}{b^{j-1}} \geq n_0. \quad (16.4)$$

For $j = 0$ it is an equality. Let $j \geq 1$ with $n/b^{j-1} \geq n_0$. Then also $n/b^{j-2} \geq n_0$, so the induction hypothesis holds for $j - 1$; applying the regularity condition to the exact power $n/b^{j-1} \geq n_0$,

$$a^j f\left(\frac{n}{b^j}\right) = a^{j-1} \left[a f\left(\frac{n/b^{j-1}}{b}\right) \right] \leq a^{j-1} c f\left(\frac{n}{b^{j-1}}\right) \leq c \cdot c^{j-1} f(n) = c^j f(n).$$

We split $g(n)$ into the terms that satisfy $n/b^{j-1} \geq n_0$ (*top levels*) and the others (*bottom levels*).

- Top levels: by (16.4) and the infinite geometric series with ratio $0 < c < 1$,

$$\sum_{\text{top}} a^j f\left(\frac{n}{b^j}\right) \leq f(n) \sum_{j=0}^{\infty} c^j = \frac{f(n)}{1-c}.$$

- Bottom levels: if $n/b^{j-1} < n_0$, then $m = n/b^j < n_0$ is an exact power of b with $m \geq b$, so $f(m) \leq M$, where M is the maximum of f over the finitely many exact powers of b in $[b, n_0]$. Moreover $a^j \leq a^L = n^p$ (Lemma 16.3), and the number of bottom levels is at most $h = \lceil \log_b n_0 \rceil$ (they correspond to $m = b, b^2, \dots$ with $m < n_0$). Hence

$$\sum_{\text{bottom}} a^j f\left(\frac{n}{b^j}\right) \leq h M n^p.$$

Finally, $f(n) = \Omega(n^{p+\epsilon})$ implies $n^p = O(f(n))$ (indeed $n^p \leq n^{p+\epsilon}$). Therefore $g(n) \leq \frac{f(n)}{1-c} + h M n^p = O(f(n))$, and together with the lower bound $g(n) = \Theta(f(n))$. $\square$

Example 16.21: Case 3 of Lemma 16.4: $T(n) = 8T(n/3) + n^2$, $n = 27$

Here $p = \log_3 8 \approx 1.893$, $c = \frac{8}{9}$ and the regularity condition holds for every n (so $n_0 = 1$, and there are no bottom levels). Level j costs $8^j \left(\frac{27}{3^j}\right)^2 = 729 \left(\frac{8}{9}\right)^j$:

$$g(27) = 729 + 648 + 576 = 1953, \quad f(27) = 729 \leq g(27) \leq \frac{f(27)}{1-8/9} = 6561.$$

The level costs decrease geometrically (they are dominated by the root). With $T(1) = 1$: $T(27) = 8^3 + 1953 = 512 + 1953 = 2465$.

16.5.4 Proof for exact powers of b

Lemma 16.5: Master Theorem for exact powers

The three cases of Theorem 16.1 hold for recurrence (16.1), i.e., when n ranges over the exact powers of b .

Proof. By Lemma 16.3, $T(n) = \Theta(n^p) + g(n)$, with $g(n) > 0$. We use Lemma 16.4.

Case 1: $g(n) = O(n^p)$, hence $\Theta(n^p) \leq T(n) \leq \Theta(n^p) + O(n^p)$, i.e., $T(n) = \Theta(n^p) = \Theta(n^{\log_b a})$.

Case 2: $g(n) = \Theta(n^p \log^{k+1} n)$. Since $n^p = O(n^p \log^{k+1} n)$, the leaves are negligible: $T(n) = \Theta(n^p \log^{k+1} n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Case 3: $g(n) = \Theta(f(n))$. Since $f(n) = \Omega(n^{p+\epsilon})$, we have $n^p = O(f(n))$, so again the leaves are negligible: $T(n) = \Theta(f(n))$. $\square$

16.5.5 Floors and ceilings

- We now consider every natural number n . We treat in detail the recurrence with ceilings,

$$T(n) = aT(\lceil n/b \rceil) + f(n) \quad \text{for } n \geq n', \quad d_1 \leq T(n) \leq d_2 \quad \text{for } 1 \leq n < n', \quad (16.5)$$

where n' and $0 < d_1 \leq d_2$ are constants, and $f(n) > 0$ for every $n \geq 1$. In case 3 the regularity condition reads $a f(\lceil n/b \rceil) \leq c f(n)$ for all $n \geq n_0$.

- **Enlarging the threshold.** The values $T(m)$ for $1 \leq m < N$ are finitely many positive numbers for every constant $N \geq n'$. Hence, replacing d_1 by their minimum and d_2 by their maximum, we may assume that the recurrence is used only for $n \geq N$, where we choose

$$N \geq \max \left\{ n', n_0, \frac{2b-1}{b-1} \right\}.$$

Then the hypotheses on f (and the regularity condition) hold for every argument $\geq N$.

- **Sizes of the subproblems.** At depth j of the recursion tree all subproblems have the same size n_j , where

$$n_0^* = n, \quad n_j = \lceil n_{j-1}/b \rceil \quad (j \geq 1)$$

(we write n_0^* for the size at depth 0 to avoid confusion with the threshold n_0).

Lemma 16.6: Sizes with ceilings

For every $j \geq 0$:

$$\frac{n}{b^j} \leq n_j < \frac{n}{b^j} + \frac{b}{b-1}.$$

Proof. By induction on j . For $j = 0$, $n_0^* = n$. For $j \geq 1$, using $x \leq \lceil x \rceil < x + 1$ and the induction hypothesis,

$$\begin{aligned} n_j &\geq \frac{n_{j-1}}{b} \geq \frac{n}{b^j}, \\ n_j &< \frac{n_{j-1}}{b} + 1 < \frac{1}{b} \left(\frac{n}{b^{j-1}} + \frac{b}{b-1} \right) + 1 = \frac{n}{b^j} + \frac{1}{b-1} + 1 = \frac{n}{b^j} + \frac{b}{b-1}. \quad \square \end{aligned}$$

Lemma 16.7: Master Theorem with ceilings

The three cases of Theorem 16.1 hold for recurrence (16.5), for every natural number n .

Proof. Let $n \geq N$ and $L = \log_b n$ (a real number, not necessarily an integer).

Step 1: depth of the tree. Let D be the smallest j such that $n_j < N$ (the depth at which the recursion stops). For $j \geq L$, Lemma 16.6 gives $n_j < 1 + \frac{b}{b-1} = \frac{2b-1}{b-1} \leq N$; hence $D \leq \lceil L \rceil < L + 1$. For $j \leq L - \log_b N$, $n_j \geq n/b^j \geq N$; hence $D > L - \log_b N$. Thus

$$L - \log_b N < D < L + 1.$$

Step 2: unfolding. Exactly as in the proof of Lemma 16.3 (the recurrence holds at every size $n_j \geq N$, i.e., for $j < D$),

$$T(n) = a^D T(n_D) + \sum_{j=0}^{D-1} a^j f(n_j).$$

Since $d_1 \leq T(n_D) \leq d_2$ and, by Step 1, $\frac{n^p}{N^p} = a^{L-\log_b N} \leq a^D \leq a^{L+1} = a n^p$, the first term is $\Theta(n^p)$.

Step 3: bounding $g(n) = \sum_{j=0}^{D-1} a^j f(n_j)$. Let $m_j = n/b^j$. For $0 \leq j \leq D-1$ we have $j < L$, so $m_j > 1$, and by Lemma 16.6

$$m_j \leq n_j < m_j + \frac{b}{b-1} < C m_j, \quad C = \frac{2b-1}{b-1}, \quad (16.6)$$

while $n_j \geq N$, so the bounds on f hold at n_j . Recall $a^j m_j^p = n^p$ by (16.3).

Case 1: $f(n_j) \leq c n_j^{p-\epsilon} \leq c C' m_j^{p-\epsilon}$ with $C' = \max\{1, C^{p-\epsilon}\}$ (if $p-\epsilon \geq 0$ use $n_j < C m_j$, otherwise $n_j \geq m_j$). As in Lemma 16.4,

$$g(n) \leq c C' n^{p-\epsilon} \sum_{j=0}^{D-1} (b^\epsilon)^j \leq c C' n^{p-\epsilon} \frac{b^{\epsilon D}}{b^\epsilon - 1} \leq c C' n^{p-\epsilon} \frac{b^\epsilon n^\epsilon}{b^\epsilon - 1} = O(n^p).$$

Case 2: Upper bound: $\log_b n_j < \log_b C + \log_b m_j \leq \log_b C + L$, so

$$a^j f(n_j) \leq c_2 a^j C^p m_j^p (L + \log_b C)^k = c_2 C^p n^p (L + \log_b C)^k,$$

and summing the $D < L + 1$ terms, $g(n) \leq c_2 C^p n^p (L + 1)(L + \log_b C)^k = O(n^p \log^{k+1} n)$.

Lower bound: $n_j \geq m_j > 1$ gives $a^j f(n_j) \geq c_1 a^j m_j^p \log_b^k m_j = c_1 n^p (L - j)^k$. For n large enough $D - 1 \geq L/2$ (because $D > L - \log_b N$), so keeping only the terms $0 \leq j \leq L/2$ (at least $L/2$ terms, each $\geq (L/2)^k$),

$$g(n) \geq c_1 n^p \left(\frac{L}{2}\right)^{k+1} = \Omega(n^p \log^{k+1} n).$$

Case 3: $g(n) \geq f(n)$ (term $j = 0$). For $1 \leq j \leq D-1$ we have $n_{j-1} \geq N \geq n_0$, so the regularity condition gives $a f(n_j) = a f(\lceil n_{j-1}/b \rceil) \leq c f(n_{j-1})$; by induction (as in (16.4)) $a^j f(n_j) \leq c^j f(n)$, hence $g(n) \leq \frac{f(n)}{1-c}$ and $g(n) = \Theta(f(n))$.

Step 4: conclusion. $T(n) = \Theta(n^p) + g(n)$ with the same bounds on $g(n)$ as in Lemma 16.4; the three cases follow exactly as in the proof of Lemma 16.5. $\square$

• **Floors.** For $T(n) = a T(\lfloor n/b \rfloor) + f(n)$ the proof is the same, with $n_j = \lfloor n_{j-1}/b \rfloor$ and the bounds

$$\frac{n}{b^j} - \frac{b}{b-1} < n_j \leq \frac{n}{b^j}$$

(proved as Lemma 16.6, using $x-1 < \lfloor x \rfloor \leq x$). Choosing also $N \geq \max\{b, \frac{2b}{b-1}\}$, for $j < D$ we get $m_j \geq n_j \geq N$, hence $\frac{b}{b-1} \leq \frac{m_j}{2}$ and $\frac{m_j}{2} < n_j \leq m_j$, which replaces (16.6);

moreover $L - \log_b(N + \frac{b}{b-1}) \leq D \leq L - \log_b N + 1$. Only the constants change.

- This completes the proof of Theorem 16.1. ■

Example 16.22: $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$, $T(1) = 1$

This is the exact recurrence of merge sort on an array of any length n . It is not of the form $aT(\lceil n/b \rceil) + f(n)$, but it is **sandwiched** between two recurrences of that form:

$$U(n) = 2U(\lceil n/2 \rceil) + n, \quad W(n) = 2W(\lfloor n/2 \rfloor) + n, \quad U(1) = W(1) = 1.$$

- U and W are nondecreasing (by strong induction: if $n < n''$ then $\lceil n/2 \rceil \leq \lceil n''/2 \rceil$ and the added term n is smaller).
- By strong induction, $W(n) \leq T(n) \leq U(n)$: for $n \geq 2$, using that U is nondecreasing and $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil$,

$$T(n) \leq U(\lceil n/2 \rceil) + U(\lfloor n/2 \rfloor) + n \leq 2U(\lceil n/2 \rceil) + n = U(n),$$

and symmetrically $T(n) \geq 2W(\lfloor n/2 \rfloor) + n = W(n)$.

- By Lemma 16.7 (case 2, $k = 0$) and its floor version, $U(n) = \Theta(n \log n)$ and $W(n) = \Theta(n \log n)$. Hence $T(n) = \Theta(n \log n)$ for **every** n , not only for powers of 2.

Numerically: $T(1000) = 10976$ and $T(10^6) = 20951424$, with $\frac{T(n)}{n \log_2 n} \approx 1.101$ and ≈ 1.051 , respectively.

16.5.6 The regularity condition

- In case 3 the hypothesis $f(n) = \Omega(n^{p+\epsilon})$ is in fact implied by the regularity condition (for positive f). This explains why, in practice, one only has to check the regularity condition.

Observation 16.8: regularity implies polynomial growth

Let $a \geq 1$, $b > 1$, $p = \log_b a$, and let $f(m) > 0$ for every exact power m of b . If $a f(n/b) \leq c f(n)$ for some constant $0 < c < 1$ and for all exact powers $n \geq n_0$ of b , then

$$f(n) = \Omega(n^{p+\epsilon}) \quad \text{with} \quad \epsilon = \log_b \frac{a}{c} - \log_b a = -\log_b c > 0.$$

Proof. Since $0 < c < 1$, $\epsilon = -\log_b c > 0$ and $\frac{a}{c} > a \geq 1$. The regularity condition can be rewritten as

$$f(n) \geq \frac{a}{c} f\left(\frac{n}{b}\right) \quad \text{for every exact power } n \geq n_0.$$

Let $n \geq n_0$ be an exact power of b , and let $j \geq 0$ be the number of times we can apply this inequality, i.e., the smallest j such that $m = n/b^j$ satisfies $m < bn_0$ (so $n_0 \leq m < bn_0$ when $n \geq n_0$; every intermediate size n/b^i , $i < j$, is $\geq bn_0 \geq n_0$). Applying the inequality j times,

$$f(n) \geq \left(\frac{a}{c}\right)^j f\left(\frac{n}{b^j}\right) \geq \left(\frac{a}{c}\right)^j \mu,$$

where $\mu > 0$ is the minimum of f over the finitely many exact powers of b in $[n_0, bn_0]$. Since $b^j = n/m > \frac{n}{bn_0}$ and $\frac{a}{c} = b^{\log_b(a/c)}$ with $\log_b(a/c) > 0$,

$$\left(\frac{a}{c}\right)^j = (b^j)^{\log_b(a/c)} > \left(\frac{n}{bn_0}\right)^{\log_b(a/c)}.$$

Finally $\log_b(a/c) = \log_b a - \log_b c = p + \epsilon$, hence

$$f(n) \geq \frac{\mu}{(bn_0)^{p+\epsilon}} n^{p+\epsilon} \quad \text{for every exact power } n \geq n_0,$$

that is, $f(n) = \Omega(n^{p+\epsilon})$. □

- For example, in Example 16.3 ($8T(n/3) + n^2$, $c = 8/9$) the observation gives $\epsilon = -\log_3 \frac{8}{9} = 2 - \log_3 8 \approx 0.107$, exactly the largest ϵ found there.
- The converse is false: in Example 16.17, $f(n) = \Omega(n^{p+1})$ but the regularity condition does not hold.

16.6 Summary

How to solve a recurrence $T(n)$.

1. Is it in standard form $T(n) = aT(n/b) + f(n)$ with constants $a \geq 1$, $b > 1$ and $f(n)$ asymptotically positive?
 - **No** $\rightarrow$ use another method: iteration, recursion tree, substitution.
 - **Yes** $\rightarrow$ compute $n^{\log_b a}$ and compare it with $f(n)$.
2. $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$ (polynomially smaller)?
 $\rightarrow$ **case 1:** $T(n) = \Theta(n^{\log_b a})$.
3. $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$?
 $\rightarrow$ **case 2:** $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
4. $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ (polynomially larger)? Check the regularity condition $af(n/b) \leq cf(n)$ with $c < 1$:
 - holds $\rightarrow$ **case 3:** $T(n) = \Theta(f(n))$;
 - fails $\rightarrow$ the theorem does not apply.
5. None of the above (a **gap**, e.g., $f(n) = n^{\log_b a} / \log n$) $\rightarrow$ the theorem does not apply: use the recursion tree $T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$ (Lemma 16.3).

Base of the logarithms: irrelevant in $\log^k n$, essential in $n^{\log_b a}$.

Recursive algorithms

Introduction

In the chapters «Recursion», «Recurrences» and «The Master Theorem» we have learnt how a recursive algorithm is executed, how its number of steps is described by a recurrence, and how such a recurrence is solved. In this chapter we put these tools to work on four classical problems on arrays:

- the **sum** of the elements of an array, the simplest possible problem, on which we compare an iterative and two recursive algorithms;
- **sorting**, with **Merge Sort**, the first algorithm of the course that beats the $\Theta(n^2)$ sorting algorithms;
- **searching** in a sorted array, with the **Recursive Binary Search**;
- the **maximum subarray** problem, where a divide and conquer algorithm improves on brute force, and a clever iterative algorithm improves on both.

All the algorithms are studied with the same scheme, which is the scheme of the analysis of every recursive algorithm.

For each recursive algorithm we:

1. write the pseudo-code and run it on a small example, drawing the **call tree**;
2. prove that it is **correct**, by (strong) induction on the size of the sub-problem;
3. describe its number of steps by a **recurrence** and solve it, with the recursion tree and with the Master Theorem;
4. express the worst-case **running time** as a function of the input size $|I|$;
5. measure the **extra memory**, which for a recursive algorithm includes the call stack;
6. give a C++ implementation (arrays indexed from 0, while in the pseudo-code they are indexed from 1).

Three of the four algorithms follow the **divide and conquer** paradigm: the problem on a sub-array $a[l..r]$ is split into sub-problems on the two halves $a[l..m]$ and $a[m+1..r]$ (*divide*), the

sub-problems are solved recursively (*conquer*), and their solutions are put together (*combine*). The Recursive Binary Search, instead, makes a single recursive call on one half.

17.1 The sum of the elements of an array

17.1.1 The problem

Given an array $a[1..n]$ of $n \geq 1$ integer numbers, we consider the problem of computing the sum of its elements:

$$\sum_{i=1}^n a[i] = a[1] + a[2] + \cdots + a[n].$$

- This is one of the simplest problems on arrays, and it is the ideal problem to compare an **iterative** algorithm and a **recursive** algorithm that solve the same problem.
- We write the recursive algorithm, we prove that it is correct, we describe its number of steps by a recurrence and we solve the recurrence, using the tools of the chapters «Recursion», «Recurrences» and «The Master Theorem».

Example 17.1: sum of an array

We consider the following array of $n = 7$ integer numbers:

$a[i]$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
	9	8	12	3	5	14	6

The sum of its elements is

$$9 + 8 + 12 + 3 + 5 + 14 + 6 = 57.$$

17.1.2 Iterative algorithm

- We start from the obvious iterative algorithm **SumIterative**, which scans the array from left to right and accumulates the partial sums in the variable s :

Algorithm: SumIterative

Input : An array $a[1..n]$ of n integer numbers

Output: The sum $a[1] + a[2] + \cdots + a[n]$

```

1 function SumIterative( $a, n$ )
2    $s := 0$ 
3    $i := 1$ 
4   while  $i \leq n$  do
5      $s \leftarrow s + a[i]$ 
6      $i \leftarrow i + 1$ 
7   end
8   return  $s$ 
9 end
```

- Let $S(n)$ denote the number of steps executed by **SumIterative** on an array of n elements (there is no difference between best and worst case: the algorithm always performs the same operations). The number of times each line of code is executed is:
 - **Line 2** is executed 1 time
 - **Line 3** is executed 1 time
 - **Line 4** is executed $n + 1$ times
 - **Line 5** is executed n times
 - **Line 6** is executed n times
 - **Line 8** is executed 1 time

Therefore,

$$S(n) = 1 + 1 + (n + 1) + n + n + 1 = 3n + 4 = \Theta(n),$$

$$\text{since } \lim_{n \rightarrow \infty} \frac{3n + 4}{n} = \lim_{n \rightarrow \infty} \left(3 + \frac{4}{n} \right) = 3 \in \mathbb{R}_{>0}.$$

- Besides the input array, the algorithm uses only the two variables s and i : the **extra memory** is $\Theta(1)$.

17.1.3 Recursive algorithm (divide and conquer)

- The same sum can be computed recursively. The sum of the elements of a sub-array $a[l..r]$ is obtained by **divide and conquer**:
 1. **Divide**: split $a[l..r]$ in two halves $a[l..m]$ and $a[m + 1..r]$, where $m = \lfloor (l + r)/2 \rfloor$;
 2. **Conquer**: compute recursively the sum s_1 of $a[l..m]$ and the sum s_2 of $a[m + 1..r]$;
 3. **Combine**: return $s_1 + s_2$.
- The **base case** is a sub-array with a single element ($l = r$): its sum is the element $a[l]$ itself.
- This is the pseudo-code of the recursive algorithm **RecursiveSum**. To sum the whole array we call **RecursiveSum**($a, 1, n$).

Algorithm: RecursiveSum

Input : An array a and two indices $l \leq r$

Output: The sum $a[l] + a[l + 1] + \dots + a[r]$

```

1 function RecursiveSum( $a, l, r$ )
2   if  $l = r$  then
3     return  $a[l]$                                 // Base case
4   end
5    $m := \lfloor (l + r)/2 \rfloor$                           // Divide
6    $s_1 := \text{RecursiveSum}(a, l, m)$                 // Conquer (left half)
7    $s_2 := \text{RecursiveSum}(a, m + 1, r)$             // Conquer (right half)
8   return  $s_1 + s_2$                                 // Combine
9 end

```

- Let $k = r - l + 1$ be the number of elements of $a[l..r]$. If $k \geq 2$, the two halves have

$$m - l + 1 = \left\lceil \frac{k}{2} \right\rceil \quad \text{and} \quad r - m = \left\lfloor \frac{k}{2} \right\rfloor$$

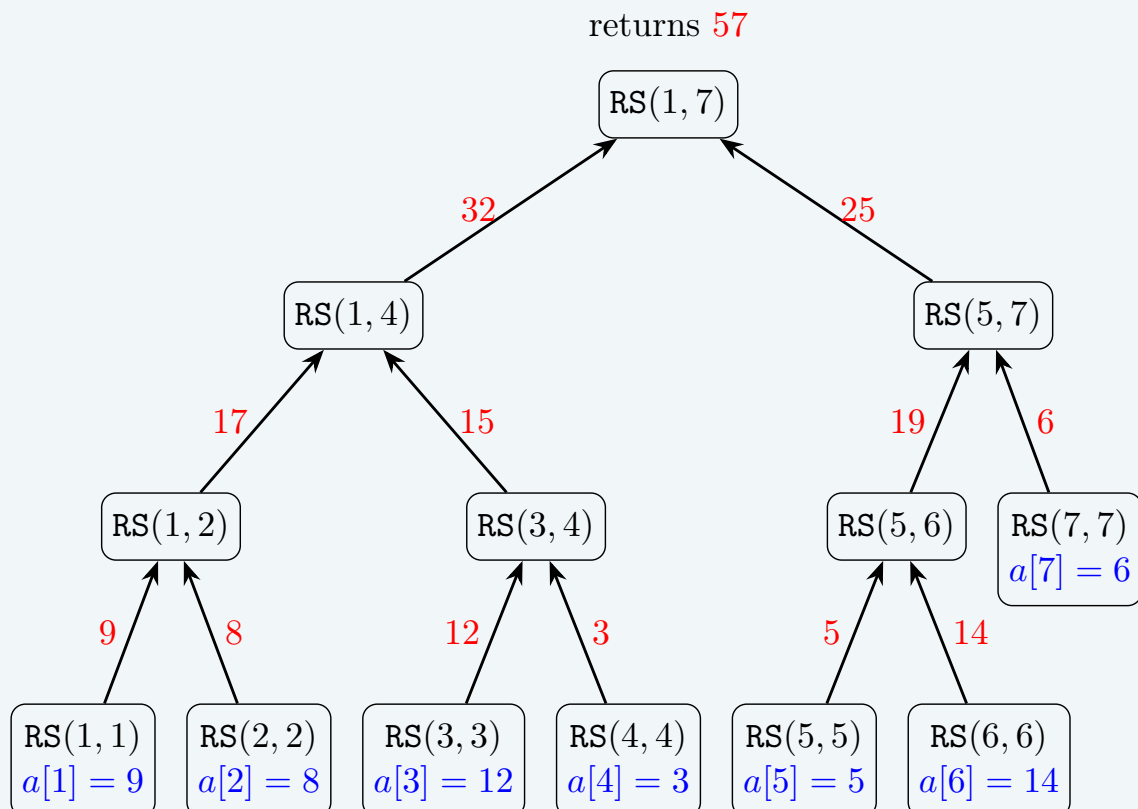
elements, respectively. Both numbers are at least 1 and strictly smaller than k : every recursive call works on a smaller sub-array, and the recursion always reaches the base case.

17.1.4 Call tree

- The execution of a recursive algorithm is represented by its **call tree**: every node is a call, and the children of a node are the calls it makes. The value returned by a call is sent back to its parent.

Example 17.2: call tree of RecursiveSum

We consider the array of Example 17.1, i.e., $a = [9, 8, 12, 3, 5, 14, 6]$ and $n = 7$. This is the call tree of `RecursiveSum(a, 1, 7)`, where each node $RS(l, r)$ is a call on the sub-array $a[l..r]$. In blue the elements of the array returned by the base cases, in red the values returned by the calls to their parents (along the arrows):



- The root $RS(1, 7)$ computes $m = \lfloor 8/2 \rfloor = 4$ and calls $RS(1, 4)$ and $RS(5, 7)$.

- $\text{RS}(1, 4)$ computes $m = 2$ and returns $17 + 15 = 32$; $\text{RS}(5, 7)$ computes $m = 6$ and returns $19 + 6 = 25$.
- The root returns $32 + 25 = 57$, which is the sum of the elements of the array.
- There are $13 = 2 \cdot 7 - 1$ calls: 7 leaves (base cases, one for each element) and 6 internal nodes, each of which performs one addition, so there are $6 = n - 1$ additions in total.

- The calls are executed in **depth-first** order: a call is completed only after both its children are completed. In the example the calls terminate in the order

$(1, 1), (2, 2), (1, 2), (3, 3), (4, 4), (3, 4), (1, 4), (5, 5), (6, 6), (5, 6), (7, 7), (5, 7), (1, 7)$.

17.1.5 Correctness

- The correctness of a recursive algorithm is proved by **strong induction** on the size of the sub-problem: we assume that the recursive calls, which work on smaller sub-arrays, are correct, and we show that the call combines their results correctly.

Observation 17.1: correctness of RecursiveSum

For every pair of indices $l \leq r$, the call $\text{RecursiveSum}(a, l, r)$ terminates and returns

$$\sum_{i=l}^r a[i].$$

Proof. We use strong induction on the number $k = r - l + 1 \geq 1$ of elements of the sub-array.

Base case ($k = 1$). Then $l = r$, the condition of line 2 is true and the algorithm returns $a[l] = \sum_{i=l}^l a[i]$.

Inductive step ($k \geq 2$). Assume that the statement holds for all sub-arrays with fewer than k elements (*inductive hypothesis*). Since $l < r$, the algorithm computes $m = \lfloor (l + r)/2 \rfloor$, with $l \leq m < r$. The sub-arrays $a[l..m]$ and $a[m + 1..r]$ have $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$ elements, both between 1 and $k - 1$. By the inductive hypothesis, the two recursive calls terminate and

$$s_1 = \sum_{i=l}^m a[i], \quad s_2 = \sum_{i=m+1}^r a[i].$$

Therefore, the algorithm returns

$$s_1 + s_2 = \sum_{i=l}^m a[i] + \sum_{i=m+1}^r a[i] = \sum_{i=l}^r a[i].$$

□

- With $l = 1$ and $r = n$, the call $\text{RecursiveSum}(a, 1, n)$ returns the sum of all the elements of the array.

17.1.6 Number of steps of RecursiveSum

17.1.6.1 The recurrence

- Let $S(n)$ denote the number of steps executed by **RecursiveSum** on a sub-array of $n = r - l + 1$ elements. As for the iterative algorithm, the number of steps depends only on n (and not on the values of the elements).
- We count the lines of code executed **by a single call**. A line containing a recursive call counts as 1 step (the call itself) **plus** all the steps executed by the called function, which we express again using S .
- If $n = 1$, **lines** 2 and 3 are executed 1 time each, and therefore $S(1) = 2$.
- If $n \geq 2$:
 - **Line** 2 is executed 1 time
 - **Line** 3 is not executed
 - **Line** 5 is executed 1 time
 - **Line** 6 is executed 1 time, plus the $S(\lceil n/2 \rceil)$ steps of the call on $a[l..m]$
 - **Line** 7 is executed 1 time, plus the $S(\lfloor n/2 \rfloor)$ steps of the call on $a[m+1..r]$
 - **Line** 8 is executed 1 time

and therefore $S(n) = S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + 5$.

The number of steps of **RecursiveSum** is described by the recurrence

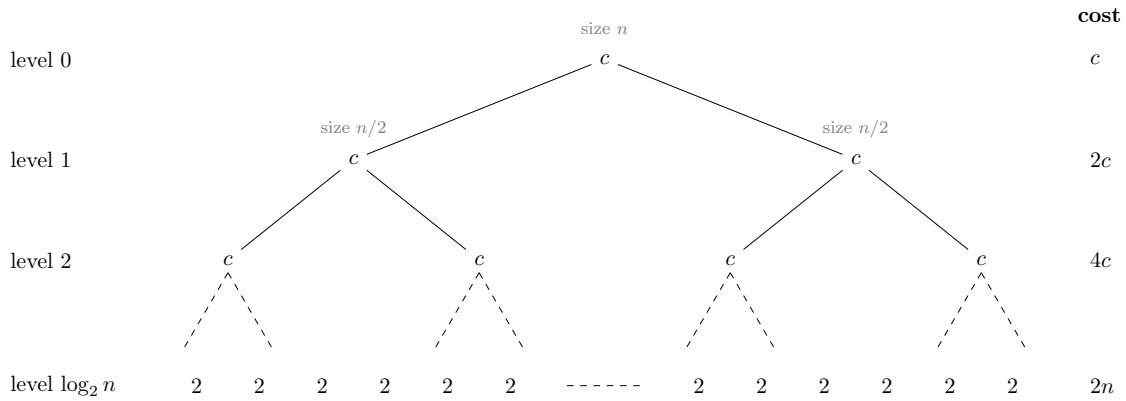
$$S(n) = \begin{cases} 2 & \text{if } n = 1, \\ S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + c & \text{if } n \geq 2, \end{cases}$$

with $c = 5$. As usual, in the asymptotic analysis we write it simply as

$$S(n) = 2 S(n/2) + c.$$

17.1.6.2 Solution with the recursion tree

- In the **recursion tree** every node represents a call and contains the number of steps executed by that call *excluding* the steps of its recursive calls: c for an internal node, 2 for a leaf. $S(n)$ is the sum of the values of all the nodes.
- For simplicity, assume first that $n = 2^h$ is an exact power of 2. Then every sub-array is split into two halves of equal size, and the tree is:



- Level i contains 2^i calls on sub-arrays of $n/2^i$ elements. The leaves are at level $\log_2 n$, where the sub-arrays have $n/2^{\log_2 n} = 1$ element: there are $2^{\log_2 n} = n$ leaves.
- The internal levels $0, 1, \dots, \log_2 n - 1$ contain

$$\sum_{i=0}^{\log_2 n - 1} 2^i = 2^{\log_2 n} - 1 = n - 1$$

nodes (sum of a geometric series). Therefore the total number of steps is

$$S(n) = c(n - 1) + 2n = 5(n - 1) + 2n = 7n - 5.$$

- The cost of the levels **doubles** from one level to the next one: the total is dominated by the leaves, which are n . This is exactly the situation of case 1 of the Master Theorem.
- The same counting holds for **every** n , not only for powers of 2: the key fact is that the tree has exactly n leaves and $n - 1$ internal nodes, as the next observation shows.

Observation 17.2: calls and steps of `RecursiveSum`

For every $n \geq 1$, the call `RecursiveSum`(a, l, r) on a sub-array of $n = r - l + 1$ elements generates exactly

$$2n - 1$$

calls (including the first one): n of them are base cases and $n - 1$ of them perform one addition. Moreover, $S(n) = 7n - 5$.

Proof. Let $C(n)$ be the number of calls. We have $C(1) = 1$ and, for $n \geq 2$,

$$C(n) = 1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor).$$

We prove $C(n) = 2n - 1$ by strong induction on n . For $n = 1$: $C(1) = 1 = 2 \cdot 1 - 1$. For $n \geq 2$, by the inductive hypothesis applied to $\lceil n/2 \rceil < n$ and $\lfloor n/2 \rfloor < n$,

$$C(n) = 1 + (2\lceil n/2 \rceil - 1) + (2\lfloor n/2 \rfloor - 1) = 2(\lceil n/2 \rceil + \lfloor n/2 \rfloor) - 1 = 2n - 1,$$

because $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$ for every natural number n .

Each base case corresponds to exactly one element of the sub-array (the sub-arrays of the leaves are disjoint and cover $a[l..r]$), so there are n base cases and $2n - 1 - n = n - 1$ calls that execute line 8.

In the same way, $S(1) = 2 = 7 \cdot 1 - 5$ and, for $n \geq 2$,

$$S(n) = (7\lceil n/2 \rceil - 5) + (7\lfloor n/2 \rfloor - 5) + 5 = 7n - 5.$$

□

- Accordingly,

$$S(n) = 7n - 5 = \Theta(n), \quad \text{since} \quad \lim_{n \rightarrow \infty} \frac{7n - 5}{n} = 7 \in \mathbb{R}_{>0}.$$

For the array of Example 17.2, $S(7) = 44$: 7 leaves with 2 steps each and 6 internal nodes with 5 steps each.

17.1.6.3 Solution with the Master Theorem

- We can obtain the order of growth of $S(n)$ directly from the Master Theorem, without computing $S(n)$ exactly. We recall case 1 of the Master Theorem:

Let $S(n) = aS(n/b) + f(n)$, with $a \geq 1$ and $b > 1$ constants and $f(n)$ asymptotically positive. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then

$$S(n) = \Theta(n^{\log_b a}).$$

- $S(n) = 2S(n/2) + c \rightarrow a = 2, b = 2, f(n) = c.$
- $n^{\log_b a} = n^{\log_2 2} = n^1 = n.$
- Comparison: $f(n) = c$ grows polynomially slower than $n \rightarrow$ case 1.
- Hypothesis: $f(n) = c = O(n^{1-\varepsilon})$ with $\varepsilon = 1$, since $c = c \cdot n^0$.
- Conclusion: $S(n) = \Theta(n^{\log_2 2}) = \Theta(n).$

As expected, the result agrees with the exact solution $S(n) = 7n - 5$.

17.1.6.4 Worst-case running time

- As in the chapter «Running time», we make the hypothesis that every entry of the array is stored using a fixed constant number of bits. Then $|I| = \Theta(n)$, i.e., there exist constants $c_1, c_2 > 0$ such that $c_1 n \leq |I| \leq c_2 n$ for all sufficiently large n , and

$$\frac{1}{c_2}|I| \leq n \leq \frac{1}{c_1}|I|.$$

Since $S(n) = 7n - 5 = \Theta(n)$, there exist constants $a_1, a_2 > 0$ such that $a_1 n \leq S(n) \leq a_2 n$ for all sufficiently large n , and hence

$$\frac{a_1}{c_2}|I| \leq S(n) \leq \frac{a_2}{c_1}|I|.$$

If all input values use a fixed constant number of bits, then the worst-case running time of the `RecursiveSum` algorithm (and also of the `SumIterative` algorithm) is

$$T(|I|) = \Theta(|I|).$$

Thus, both algorithms are linear in the input size.

- Recursion does not make the algorithm faster: both algorithms read every element once and perform $n - 1$ (recursive) or n (iterative) additions. The recursive version even executes more steps ($7n - 5$ against $3n + 4$), because of the overhead of the calls.

17.1.7 Extra memory: the call stack

- Time is not the only resource. Each active call of a recursive function needs its own **activation record** on the **call stack**, containing its parameters, its local variables and the return address. In `RecursiveSum` the parameters and local variables are l , r , m , s_1 , s_2 : every activation record has constant size.
- A call is active from the moment it starts until it returns. Since the calls are executed depth-first, the active calls at any moment form a path from the root of the call tree to the current node. Therefore:

The maximum number of simultaneously active calls (the **recursion depth**) equals the height of the call tree plus one. The extra memory used by a recursive algorithm is (at least) proportional to the recursion depth.

Observation 17.3: recursion depth of `RecursiveSum`

For every $n \geq 1$, the call tree of `RecursiveSum` on a sub-array of n elements has height $\lceil \log_2 n \rceil$. Hence the recursion depth is $\lceil \log_2 n \rceil + 1$ and the extra memory used by `RecursiveSum` is $\Theta(\log_2 n)$.

Proof. Let $h(n)$ be the height of the call tree. Then $h(1) = 0$ and, for $n \geq 2$,

$$h(n) = 1 + \max \{h(\lceil n/2 \rceil), h(\lfloor n/2 \rfloor)\}.$$

We prove $h(n) = \lceil \log_2 n \rceil$ by strong induction. For $n = 1$, $\lceil \log_2 1 \rceil = 0$. Let $n \geq 2$. Since $\lceil \log_2 x \rceil$ is non-decreasing in x , by the inductive hypothesis the maximum is attained by the larger half, and $h(n) = 1 + \lceil \log_2 \lceil n/2 \rceil \rceil$.

Let $k = \lceil \log_2 n \rceil \geq 1$, i.e., $2^{k-1} < n \leq 2^k$. Dividing by 2, we obtain $2^{k-2} < n/2 \leq 2^{k-1}$; since 2^{k-1} is an integer, also $2^{k-2} < \lceil n/2 \rceil \leq 2^{k-1}$, that is $\lceil \log_2 \lceil n/2 \rceil \rceil = k - 1$. Therefore $h(n) = 1 + (k - 1) = k = \lceil \log_2 n \rceil$.

Each activation record has constant size, so the extra memory is $\Theta(\lceil \log_2 n \rceil + 1) = \Theta(\log_2 n)$. $\square$

- In Example 17.2, $n = 7$ and the recursion depth is $\lceil \log_2 7 \rceil + 1 = 3 + 1 = 4$: for instance, when `RS(1, 1)` is executed, the active calls are `RS(1, 7)`, `RS(1, 4)`, `RS(1, 2)` and `RS(1, 1)`.

algorithm	number of steps	extra memory
SumIterative	$3n + 4 = \Theta(n)$	$\Theta(1)$
RecursiveSum	$7n - 5 = \Theta(n)$	$\Theta(\log_2 n)$

17.1.8 A different recursion: decrease by one

- The halving is not the only way to sum an array recursively. The array can also be split in an **unbalanced** way: the sum of $a[1..n]$ is the sum of $a[1..n-1]$ plus $a[n]$. This is a *decrease by one* recursion:

Algorithm: RecursiveSumLinear

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The sum $a[1] + a[2] + \dots + a[n]$

```

1 function RecursiveSumLinear(a, n)
2   if n = 1 then
3     return a[1]                                // Base case
4   end
5   s := RecursiveSumLinear(a, n - 1)
6   return s + a[n]
7 end

```

- Correctness follows by (simple) induction on n : for $n = 1$ the algorithm returns $a[1]$; for $n \geq 2$, by the inductive hypothesis $s = a[1] + \dots + a[n-1]$, and the algorithm returns $s + a[n] = a[1] + \dots + a[n]$.
- If $n = 1$, lines 2 and 3 are executed once: $S(1) = 2$. If $n \geq 2$, lines 2, 5 and 6 are executed once, plus the $S(n-1)$ steps of the recursive call:

$$S(n) = \begin{cases} 2 & \text{if } n = 1, \\ S(n-1) + 3 & \text{if } n \geq 2. \end{cases}$$

- The Master Theorem **cannot** be applied, because the size of the sub-problem is $n-1$ and not n/b for a constant $b > 1$. We solve the recurrence by iteration (unrolling):

$$S(n) = S(n-1) + 3 = S(n-2) + 2 \cdot 3 = \dots = S(1) + (n-1) \cdot 3 = 3n - 1 = \Theta(n).$$

- The call tree is a **path** of n nodes: **RecursiveSumLinear**(a, n) calls **RecursiveSumLinear**($a, n-1$), which calls **RecursiveSumLinear**($a, n-2$), and so on, until $n = 1$. When the base case is reached, all the n calls are active.

RecursiveSumLinear executes $S(n) = 3n - 1 = \Theta(n)$ steps, but its recursion depth is n : it needs $\Theta(n)$ extra memory for the call stack (against $\Theta(\log_2 n)$ of **RecursiveSum** and $\Theta(1)$ of **SumIterative**). In practice, for large n (e.g. $n = 10^7$) the decrease-by-one version may exhaust the call stack (*stack overflow*), while **RecursiveSum** needs only $\lceil \log_2 10^7 \rceil + 1 = 25$ active calls.

17.1.9 C++ implementation

- This is a C++ implementation of the iterative and of the recursive algorithms. In C++ arrays are indexed from 0: the sum of the whole array is computed by the call `RecursiveSum(a, 0, n-1)`.

```

1 // Iterative sum of a[0..n-1]
2 int SumIterative(int n, int *a)
3 {
4     int s = 0;
5     for (int i = 0; i < n; i++)
6     {
7         s = s + a[i];
8     }
9     return s;
10 }
11
12 // Recursive sum of a[l..r] (0-indexed, l <= r)
13 int RecursiveSum(int *a, int l, int r)
14 {
15     if (l == r)
16     {
17         return a[l]; // Base case: one element
18     }
19
20     int m = l + (r - l) / 2; // Equivalent to (l + r) / 2
21
22     int s1 = RecursiveSum(a, l, m); // Sum of the left half
23     int s2 = RecursiveSum(a, m + 1, r); // Sum of the right half
24
25     return s1 + s2;
26 }

```

- The expression $l + (r - l) / 2$ computes $\lfloor (l + r) / 2 \rfloor$ (for $l, r \geq 0$) without the risk of overflow of the sum $l + r$, as in the Binary Search algorithm.

Example 17.3: execution of the C++ implementation of RecursiveSum

We execute the call `RecursiveSum(a, 0, 4)` on the array $a = [9, 8, 12, 3, 5]$ ($n = 5$):

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$l = 0$	$r = 4$
	9	8	12	3	5		

The column *stack* reports the active calls, from the first one to the current one (we write (l, r) for `RecursiveSum(a, l, r)`).

#	stack	action
1	(0, 4)	$l \neq r, m = 0 + (4 - 0)/2 = 2$; call (0, 2)
2	(0, 4), (0, 2)	$l \neq r, m = 0 + (2 - 0)/2 = 1$; call (0, 1)
3	(0, 4), (0, 2), (0, 1)	$l \neq r, m = 0 + (1 - 0)/2 = 0$; call (0, 0)
4	(0, 4), (0, 2), (0, 1), (0, 0)	$l = r \Rightarrow \text{return } a[0] = 9$
5	(0, 4), (0, 2), (0, 1)	$s1 = 9$; call (1, 1)
6	(0, 4), (0, 2), (0, 1), (1, 1)	$l = r \Rightarrow \text{return } a[1] = 8$
7	(0, 4), (0, 2), (0, 1)	$s2 = 8$; return $9 + 8 = 17$
8	(0, 4), (0, 2)	$s1 = 17$; call (2, 2)
9	(0, 4), (0, 2), (2, 2)	$l = r \Rightarrow \text{return } a[2] = 12$
10	(0, 4), (0, 2)	$s2 = 12$; return $17 + 12 = 29$
11	(0, 4)	$s1 = 29$; call (3, 4)
12	(0, 4), (3, 4)	$l \neq r, m = 3 + (4 - 3)/2 = 3$; call (3, 3)
13	(0, 4), (3, 4), (3, 3)	$l = r \Rightarrow \text{return } a[3] = 3$
14	(0, 4), (3, 4)	$s1 = 3$; call (4, 4)
15	(0, 4), (3, 4), (4, 4)	$l = r \Rightarrow \text{return } a[4] = 5$
16	(0, 4), (3, 4)	$s2 = 5$; return $3 + 5 = 8$
17	(0, 4)	$s2 = 8$; return $29 + 8 = 37$

The algorithm terminates with result $37 = 9 + 8 + 12 + 3 + 5$. There are $9 = 2 \cdot 5 - 1$ calls, and the maximum number of active calls is $4 = \lceil \log_2 5 \rceil + 1$ (rows 4 and 6).

Key points on the sum of an array.

- **RecursiveSum** is correct (strong induction), makes $2n - 1$ calls and executes $7n - 5 = \Theta(n)$ steps: recursion does not reduce the work, which is linear because every element must be read.
- The recurrence $S(n) = 2S(n/2) + c$ falls in case 1 of the Master Theorem: the cost is dominated by the n leaves.
- The call stack costs memory: $\Theta(\log_2 n)$ for the halving recursion, $\Theta(n)$ for the decrease-by-one recursion, $\Theta(1)$ for the loop.

17.2 Merge Sort

For the sum of an array, divide and conquer gave no gain in time. We now meet a problem where it does make a difference: sorting.

17.2.1 The sorting problem

Given an array $a[1..n]$ of n natural numbers, we consider the problem of rearranging the elements of the array in non-decreasing order such that:

$$a[1] \leq a[2] \leq \dots \leq a[n].$$

- The problem has been introduced in the chapter «Sorting algorithms», where we studied Insertion Sort, Exchange Sort and Selection Sort. In the chapter «Running time» we proved that the worst-case number of steps of **InsertionSort** is $S(n) = \frac{3n^2+9n-8}{2} = \Theta(n^2)$.
- In this section we study **Merge Sort**, a recursive comparison-based sorting algorithm based on the **divide and conquer** paradigm, whose number of steps is $\Theta(n \log_2 n)$ in the worst case.

Example 17.4: a sorting instance

We consider the following array of $n = 7$ natural numbers:

$a[i]$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
	9	12	3	16	19	11	12

The sorted array is:

$a[i]$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
	3	9	11	12	12	16	19

17.2.2 The Merge Sort algorithm

17.2.2.1 Divide and conquer

- Merge Sort sorts a sub-array $a[l..r]$ as follows:
 1. **Divide**: split $a[l..r]$ in two halves $a[l..m]$ and $a[m+1..r]$, where $m = \lfloor (l+r)/2 \rfloor$;
 2. **Conquer**: sort recursively the two halves;
 3. **Combine**: **merge** the two sorted halves into a single sorted sub-array $a[l..r]$.
- The **base case** is a sub-array with a single element ($l = r$), which is already sorted: nothing has to be done.
- The whole work is done in the **combine** step, by the procedure **Merge**. The key observation is that two *sorted* sequences can be merged very efficiently: the smallest element of the result is the smaller of the two first elements, and so on.

17.2.2.2 Pseudo-code

- This is the pseudo-code of the **MergeSort** algorithm. To sort the whole array $a[1..n]$, we call **MergeSort**($a, 1, n$).

Algorithm: MergeSort**Input** : An array a and two indices $l \leq r$ **Output:** The sub-array $a[l..r]$ sorted in non-decreasing order

```

1 function MergeSort( $a, l, r$ )
2   if  $l < r$  then
3      $m := \lfloor (l + r) / 2 \rfloor$                                 // Divide
4     MergeSort( $a, l, m$ )                                  // Conquer (left half)
5     MergeSort( $a, m + 1, r$ )                              // Conquer (right half)
6     Merge( $a, l, m, r$ )                                    // Combine
7   end
8 end

```

- If $k = r - l + 1 \geq 2$, the two halves $a[l..m]$ and $a[m + 1..r]$ have $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$ elements, both between 1 and $k - 1$: the recursion always reaches the base case.
- The procedure **Merge**(a, l, m, r) assumes that $a[l..m]$ and $a[m + 1..r]$ are sorted. It copies them into two auxiliary arrays L and R , and then it repeatedly moves the smaller of the first two elements not yet copied back into a :

Algorithm: Merge

Input : An array a and indices $l \leq m < r$ such that $a[l..m]$ and $a[m+1..r]$ are sorted

Output: The sub-array $a[l..r]$ sorted in non-decreasing order

```

1 function Merge( $a, l, m, r$ )
2    $n_1 := m - l + 1$ 
3    $n_2 := r - m$ 
4    $L := \text{new array}[1..n_1], R := \text{new array}[1..n_2]$ 
5   for  $i := 1$  to  $n_1$  do
6      $L[i] := a[l + i - 1]$ 
7   end
8   for  $j := 1$  to  $n_2$  do
9      $R[j] := a[m + j]$ 
10  end
11   $i := 1$ 
12   $j := 1$ 
13   $k := l$ 
14  while  $i \leq n_1$  and  $j \leq n_2$  do
15    if  $L[i] \leq R[j]$  then
16       $a[k] \leftarrow L[i]$ 
17       $i \leftarrow i + 1$ 
18    else
19       $a[k] \leftarrow R[j]$ 
20       $j \leftarrow j + 1$ 
21    end
22     $k \leftarrow k + 1$ 
23  end
24  while  $i \leq n_1$  do // Copy the rest of  $L$ 
25     $a[k] \leftarrow L[i]$ 
26     $i \leftarrow i + 1$ 
27     $k \leftarrow k + 1$ 
28  end
29  while  $j \leq n_2$  do // Copy the rest of  $R$ 
30     $a[k] \leftarrow R[j]$ 
31     $j \leftarrow j + 1$ 
32     $k \leftarrow k + 1$ 
33  end
34 end

```

- Explanation of Merge:

- Lines 2–9 copy $a[l..m]$ into $L[1..n_1]$ and $a[m+1..r]$ into $R[1..n_2]$. The copies are necessary, because the positions $a[l..r]$ are overwritten during the merge.
- The indices i and j point to the first elements of L and R not yet copied back, and k points to the next position of a to be filled.

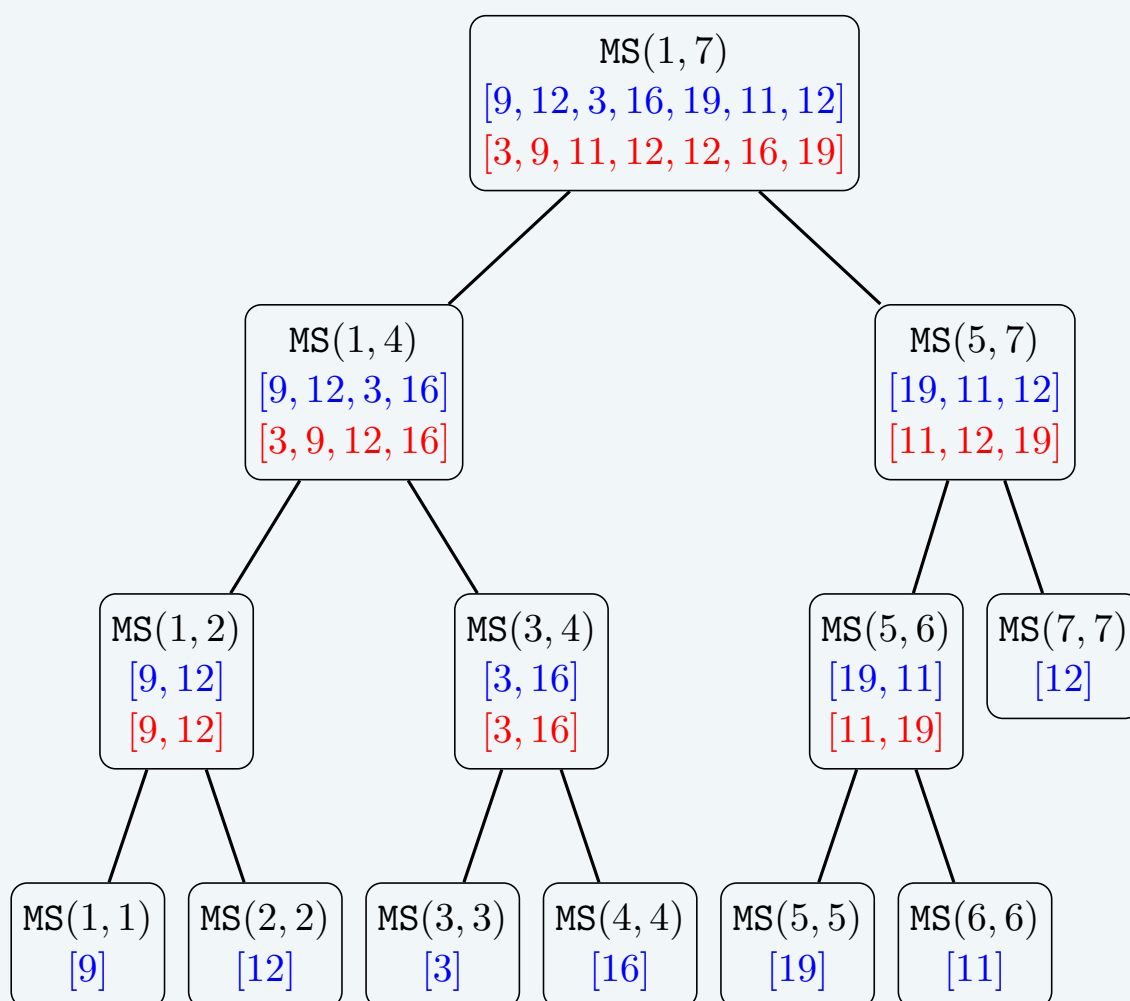
- The first while loop (line 14) compares $L[i]$ and $R[j]$ and copies the smaller one into $a[k]$ (in case of a tie, the element of L is copied first: this makes Merge Sort **stable**).
- As soon as one of the two arrays is exhausted, the remaining elements of the other one (which are already sorted, and not smaller than the ones already copied) are copied into a by the while loop of line 24 or by the one of line 29. Only one of these two loops executes at least one iteration.

17.2.2.3 Examples

We now run Merge Sort on the array of Example 17.4: first we look at the whole call tree, then at the two phases of the algorithm, and finally at a single call of **Merge**.

Example 17.5: call tree of MergeSort

We consider the array of Example 17.4, i.e., $a = [9, 12, 3, 16, 19, 11, 12]$. This is the call tree of $\text{MergeSort}(a, 1, 7)$, where each node $\text{MS}(l, r)$ is a call on the sub-array $a[l..r]$. In each node, in **blue** the sub-array received by the call and in **red** the sorted sub-array produced by the call (after its **Merge**):

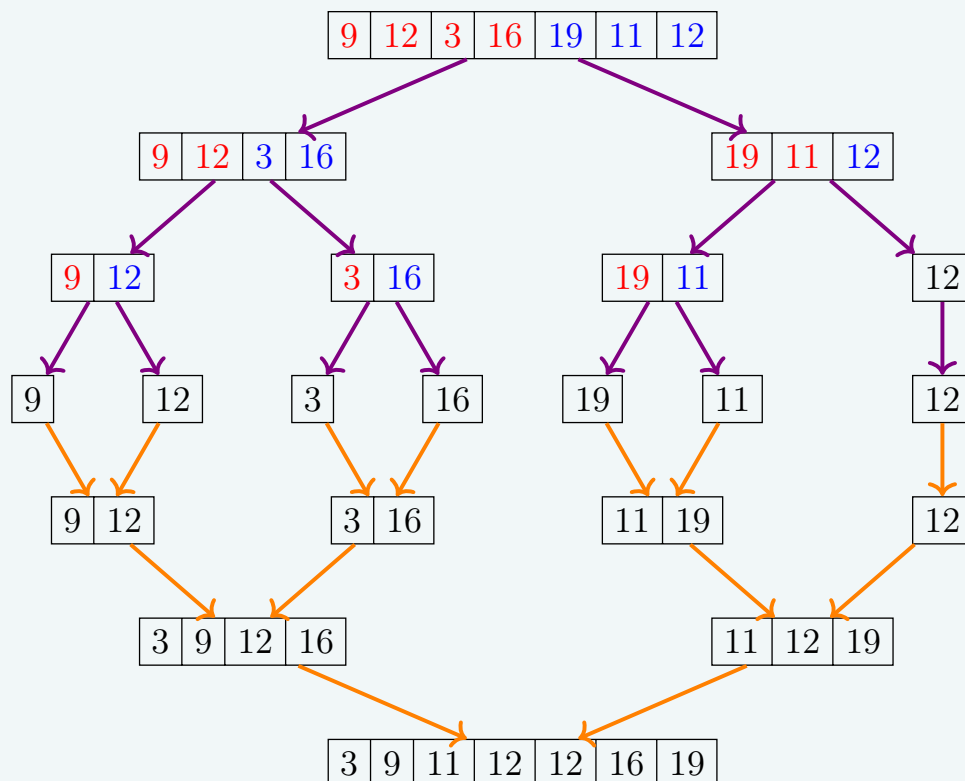


The calls of **Merge** are executed in the following order (depth-first), and this is the content of the array after each of them (the sub-array modified by each call is in **red**):

call	merged sub-arrays	array <i>a</i> after the call
Merge(<i>a</i> , 1, 1, 2)	[9] and [12]	[9 , 12 , 3, 16, 19, 11, 12]
Merge(<i>a</i> , 3, 3, 4)	[3] and [16]	[9, 12, 3 , 16 , 19, 11, 12]
Merge(<i>a</i> , 1, 2, 4)	[9, 12] and [3, 16]	[3 , 9 , 12 , 16 , 19, 11, 12]
Merge(<i>a</i> , 5, 5, 6)	[19] and [11]	[3, 9, 12, 16, 11 , 19 , 12]
Merge(<i>a</i> , 5, 6, 7)	[11, 19] and [12]	[3, 9, 12, 16, 11 , 12 , 19]
Merge(<i>a</i> , 1, 4, 7)	[3, 9, 12, 16] and [11, 12, 19]	[3 , 9 , 11 , 12 , 12 , 16 , 19]

Example 17.6: divide and combine phases of MergeSort

We consider again the array of Example 17.4. The upper part of the figure shows the **divide** phase (**violet** arrows): each sub-array is split into its left half (in **red**) and its right half (in **blue**), until sub-arrays of one element are obtained. The lower part shows the **combine** phase (**orange** arrows): pairs of sorted sub-arrays are merged into a sorted sub-array.



Example 17.7: trace of Merge

We consider the last call `Merge(a, 1, 4, 7)` of Example 17.5. Before the call $a = [3, 9, 12, 16, 11, 12, 19]$, with $a[1..4]$ and $a[5..7]$ sorted. Then $n_1 = 4$, $n_2 = 3$ and

$$L = [3, 9, 12, 16], \quad R = [11, 12, 19].$$

This is the execution of the while loops (one row per element copied into a):

k	i	j	$L[i]$	$R[j]$	comparison	action
1	1	1	3	11	$3 \leq 11$	$a[1] \leftarrow 3, i = 2$
2	2	1	9	11	$9 \leq 11$	$a[2] \leftarrow 9, i = 3$
3	3	1	12	11	$12 > 11$	$a[3] \leftarrow 11, j = 2$
4	3	2	12	12	$12 \leq 12$	$a[4] \leftarrow 12$ (from L), $i = 4$
5	4	2	16	12	$16 > 12$	$a[5] \leftarrow 12$ (from R), $j = 3$
6	4	3	16	19	$16 \leq 19$	$a[6] \leftarrow 16, i = 5$
7	5	3	–	19	L exhausted	$a[7] \leftarrow 19, j = 4$

After 6 iterations of the first while loop (and 6 comparisons), $i = 5 > n_1$: the array L is exhausted, and the last element of R is copied by the while loop of line 29. The result is $a = [3, 9, 11, 12, 12, 16, 19]$.

17.2.3 Correctness

The correctness of Merge Sort rests on two facts: **Merge** correctly merges two sorted sub-arrays (a property of a loop, proved with a loop invariant), and the recursion is correct (proved by strong induction, as for **RecursiveSum**).

17.2.3.1 Correctness of Merge

- We prove the correctness of **Merge** by a **loop invariant**, i.e., a property that holds at the beginning of every iteration of a loop. As in the chapter «Correctness», we show that the invariant holds before the first iteration (*initialization*), that if it holds before an iteration it also holds before the next one (*maintenance*), and what it gives us when the loop ends (*termination*).

Observation 17.4: loop invariant of Merge

Let $a[l..m]$ and $a[m + 1..r]$ be sorted. At the beginning of every iteration of the while loop of line 14 of **Merge**:

- (I1) the sub-array $a[l..k - 1]$ contains the $k - l$ elements of $L[1..i - 1]$ and $R[1..j - 1]$, in non-decreasing order;
- (I2) every element of $a[l..k - 1]$ is smaller than or equal to every element of $L[i..n_1]$ and of $R[j..n_2]$.

At the end, **Merge** terminates and $a[l..r]$ contains the elements originally in $a[l..r]$, sorted in non-decreasing order.

Proof. After lines 2–9, $L[1..n_1]$ and $R[1..n_2]$ are copies of the sorted sub-arrays $a[l..m]$ and $a[m+1..r]$, hence they are sorted.

Initialization. Before the first iteration $k = l$, $i = 1$, $j = 1$: the sub-array $a[l..k-1]$ is empty, and (I1) and (I2) hold trivially.

Maintenance. Assume (I1) and (I2) at the beginning of an iteration, and suppose $L[i] \leq R[j]$ (the other case is symmetric). Since L and R are sorted, $L[i] \leq L[i+1..n_1]$ and $L[i] \leq R[j] \leq R[j+1..n_2]$: $L[i]$ is the smallest element not yet copied. By (I2), $L[i]$ is greater than or equal to all the elements of $a[l..k-1]$, so after $a[k] \leftarrow L[i]$ the sub-array $a[l..k]$ is sorted and contains $L[1..i]$ and $R[1..j-1]$. After $i \leftarrow i+1$ and $k \leftarrow k+1$, (I1) holds again. (I2) holds as well, because the elements not yet copied are a subset of the previous ones, and the new element $a[k-1] = L[i-1]$ is smaller than or equal to all of them.

Termination. Every iteration increases $i+j$ by one, so the loop ends after at most $n_1 + n_2$ iterations, with $i = n_1 + 1$ or $j = n_2 + 1$. Suppose $i = n_1 + 1$ (the other case is symmetric). The loop of line 24 is skipped, and the loop of line 29 copies $R[j..n_2]$, in this order, into $a[k..r]$. Since $R[j..n_2]$ is sorted and, by (I2), its elements are not smaller than those of $a[l..k-1]$, the whole sub-array $a[l..r]$ is sorted. By (I1), it contains exactly the elements of L and R , i.e., the elements originally in $a[l..r]$. $\square$

17.2.3.2 Correctness of MergeSort

Observation 17.5: correctness of MergeSort

For every pair of indices $l \leq r$, the call `MergeSort( $a, l, r$ )` terminates and rearranges the elements of $a[l..r]$ in non-decreasing order (without modifying the rest of the array).

Proof. We use strong induction on the number $k = r - l + 1 \geq 1$ of elements of the sub-array.

Base case ($k = 1$). Then $l = r$, the condition of line 2 is false and nothing is done: a sub-array of one element is sorted.

Inductive step ($k \geq 2$). Assume that the statement holds for all sub-arrays with fewer than k elements. Since $l < r$, the algorithm computes $m = \lfloor (l+r)/2 \rfloor$, with $l \leq m < r$. The sub-arrays $a[l..m]$ and $a[m+1..r]$ have $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$ elements, both between 1 and $k-1$. By the inductive hypothesis, after lines 4 and 5 both $a[l..m]$ and $a[m+1..r]$ are sorted (and contain the same elements as before). Therefore the hypothesis of `Merge` is satisfied, and by Observation 17.4, after line 6 the sub-array $a[l..r]$ is sorted. $\square$

17.2.4 Number of steps

17.2.4.1 Number of steps of Merge

- Let $n = r - l + 1 = n_1 + n_2$ be the number of elements to be merged, and let t be the number of iterations of the while loop of line 14. Among them, t_L copy an element of L and t_R copy an element of R , with $t_L + t_R = t$. The header of a for loop from 1 to n_1 is executed $n_1 + 1$ times (the last test fails). The line **else** is part of the **if** of line 15 and is not counted separately.

The number of times each line of code is executed is:

- Lines 2, 3, 4 are executed 1 time each
- Line 5 is executed $n_1 + 1$ times and line 6 n_1 times
- Line 8 is executed $n_2 + 1$ times and line 9 n_2 times
- Lines 11, 12, 13 are executed 1 time each
- Line 14 is executed $t + 1$ times
- Line 15 is executed t times
- Lines 16, 17 are executed t_L times each
- Lines 19, 20 are executed t_R times each
- Line 22 is executed t times
- Line 24 is executed $(n_1 - t_L) + 1$ times and lines 25, 26, 27 $n_1 - t_L$ times each
- Line 29 is executed $(n_2 - t_R) + 1$ times and lines 30, 31, 32 $n_2 - t_R$ times each

Therefore, the total number of steps of Merge is

$$\begin{aligned}
 & 3 + (2n_1 + 1) + (2n_2 + 1) + 3 + (t + 1) + t + 2t_L + 2t_R + t \\
 & \quad + 4(n_1 - t_L) + 1 + 4(n_2 - t_R) + 1 \\
 & = 11 + 2n + 5t + 4(n - t) = 6n + t + 11.
 \end{aligned}$$

- Every iteration of the first while loop copies one element into a , so $t \leq n$. More precisely:
 - the loop stops as soon as one of the two arrays is exhausted, so $t \geq \min\{n_1, n_2\}$;
 - when the loop stops, the other array still contains at least one element, so $t \leq n - 1$.

Hence the number of steps of Merge on n elements is between $6n + 11 + \min\{n_1, n_2\}$ and $7n + 10$; in particular, it is between $6n + 11$ and $7n + 10$. The worst case $t = n - 1$ happens, for instance, in Example 17.7 ($n = 7$, $t = 6$, $6 \cdot 7 + 6 + 11 = 59$ steps).

The number of steps of Merge on $n = r - l + 1$ elements is $6n + t + 11$ and satisfies

$$6n + 11 \leq S_{\text{Merge}}(n) \leq 7n + 10, \quad \text{hence} \quad S_{\text{Merge}}(n) = \Theta(n).$$

The number of comparisons between elements (line 15) is t , with $\min\{n_1, n_2\} \leq t \leq n - 1$.

17.2.4.2 The recurrence of MergeSort

- Let $S(n)$ denote the number of steps executed by **MergeSort** in the worst case on a subarray of $n = r - l + 1$ elements. As for the sum of an array, a line containing a call counts 1 step plus the steps of the called function.
- If $n = 1$, only **line 2** is executed: $S(1) = 1$.
- If $n \geq 2$:
 - **Line 2** is executed 1 time
 - **Line 3** is executed 1 time
 - **Line 4** is executed 1 time, plus the $S(\lceil n/2 \rceil)$ steps of the call
 - **Line 5** is executed 1 time, plus the $S(\lfloor n/2 \rfloor)$ steps of the call
 - **Line 6** is executed 1 time, plus the $S_{\text{Merge}}(n)$ steps of **Merge**

and therefore

$$S(n) = S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + f(n), \quad f(n) = 5 + S_{\text{Merge}}(n).$$

- From the analysis of **Merge**, $6n + 16 \leq f(n) \leq 7n + 15$. Since $7n + 15 \leq 15n$ for $n \geq 2$, we have

$$6n \leq f(n) \leq 15n \quad \text{for every } n \geq 2, \quad \text{i.e.,} \quad f(n) = \Theta(n).$$

The number of steps of **MergeSort** is described by the recurrence

$$S(n) = \begin{cases} 1 & \text{if } n = 1, \\ S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2, \end{cases}$$

which, ignoring floors and ceilings, we write as

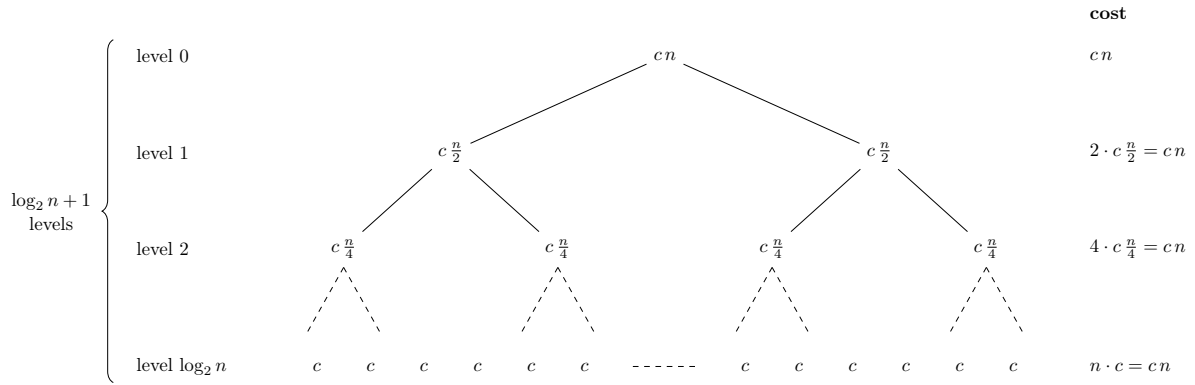
$$S(n) = 2S(n/2) + \Theta(n).$$

17.2.4.3 Solution with the recursion tree

- Assume that $n = 2^h$ is an exact power of 2, and let $c = 15$ be the constant such that $f(n) \leq cn$ (with $S(1) = 1 \leq c$). We first bound $S(n)$ from above by the recursion tree of

$$S(n) \leq 2S(n/2) + cn,$$

where each node contains the cost of a call, excluding the cost of its recursive calls:



- The top level has cost cn ; the next level has cost $c \frac{n}{2} + c \frac{n}{2} = cn$; the level after has cost $4 \cdot c \frac{n}{4} = cn$, and so on. In general, level i has 2^i nodes, each corresponding to a sub-array of $n/2^i$ elements and contributing a cost of at most $cn/2^i$, so each level costs at most

$$2^i \cdot c \frac{n}{2^i} = cn.$$

- The bottom level $\log_2 n$ has n leaves (sub-arrays of one element), each with cost $S(1) = 1 \leq c$: its cost is at most cn .
- The levels are $0, 1, \dots, \log_2 n$, i.e., $\log_2 n + 1$. Summing the costs of all the levels:

$$S(n) \leq cn(\log_2 n + 1) = cn \log_2 n + cn.$$

- In the same way, using $f(n) \geq 6n$, each internal level costs at least $6n$, and there are $\log_2 n$ internal levels: $S(n) \geq 6n \log_2 n$.
- Therefore, for n a power of 2,

$$6n \log_2 n \leq S(n) \leq 15n \log_2 n + 15n, \quad \text{i.e.,} \quad S(n) = \Theta(n \log_2 n),$$

since

$$\lim_{n \rightarrow \infty} \frac{15n \log_2 n + 15n}{n \log_2 n} = \lim_{n \rightarrow \infty} \left(15 + \frac{15}{\log_2 n} \right) = 15 \in \mathbb{R}_{>0}.$$

- Contrary to the recursion tree of the sum of an array, where the cost of the levels doubles at each level, here **all the levels have the same cost** cn : the total is the cost of one level times the number of levels. This is exactly the situation of case 2 of the Master Theorem.

17.2.4.4 Solution with the Master Theorem

- We recall case 2 of the Master Theorem:

Let $S(n) = aS(n/b) + f(n)$, with $a \geq 1$ and $b > 1$ constants and $f(n)$ asymptotically positive. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then

$$S(n) = \Theta(n^{\log_b a} \log^{k+1} n).$$

- $S(n) = 2S(n/2) + \Theta(n) \rightarrow a = 2, b = 2, f(n) = \Theta(n).$
- $n^{\log_b a} = n^{\log_2 2} = n.$
- Comparison: $f(n)$ and $n^{\log_b a}$ grow in the same way $\rightarrow$ case 2.
- Hypothesis: $f(n) = \Theta(n) = \Theta(n^{\log_2 2} \log^0 n)$ with $k = 0.$
- Conclusion: $S(n) = \Theta(n^{\log_2 2} \log^1 n) = \Theta(n \log n).$

The base of the logarithm is irrelevant inside Θ , because $\log_b n = \frac{\log_2 n}{\log_2 b}$ differs from $\log_2 n$ by a constant factor. Hence $S(n) = \Theta(n \log_2 n)$, in agreement with the recursion tree.

17.2.4.5 Worst-case running time

- We make the hypothesis that every entry of the array is stored using a fixed constant number of bits. Then $|I| = \Theta(n)$: there exist constants $c_1, c_2 > 0$ such that, for all sufficiently large n ,

$$\frac{1}{c_2}|I| \leq n \leq \frac{1}{c_1}|I|.$$

Taking logarithms, $\log_2 |I| - \log_2 c_2 \leq \log_2 n \leq \log_2 |I| - \log_2 c_1$, hence, for sufficiently large $|I|$,

$$\frac{1}{2} \log_2 |I| \leq \log_2 n \leq 2 \log_2 |I|.$$

Multiplying the two chains of inequalities (all terms are positive),

$$\frac{1}{2c_2} |I| \log_2 |I| \leq n \log_2 n \leq \frac{2}{c_1} |I| \log_2 |I|,$$

that is, $n \log_2 n = \Theta(|I| \log_2 |I|)$. Since $S(n) = \Theta(n \log_2 n)$, we obtain $S(n) = \Theta(|I| \log_2 |I|)$.

If all input values use a fixed constant number of bits, then the worst-case running time of the MergeSort algorithm is

$$T(|I|) = \Theta(|I| \log |I|).$$

17.2.5 Extra memory

- **Merge**(a, l, m, r) allocates the auxiliary arrays L and R , with $n_1 + n_2 = r - l + 1$ elements in total, which are released when **Merge** returns.
- **Merge** is called only after both recursive calls have returned, so at any moment at most one **Merge** is active, and the largest auxiliary arrays are the ones of the first call **MergeSort**(a, l, n), with n elements in total.
- In addition, the call stack contains at most $\lceil \log_2 n \rceil + 1$ activation records of constant size (the call tree has the same shape as the one of **RecursiveSum**, Observation 17.3): $\Theta(\log_2 n)$ memory.

MergeSort uses $\Theta(n) + \Theta(\log_2 n) = \Theta(n)$ extra memory: it is **not** an in-place sorting algorithm. On the contrary, **InsertionSort** sorts in place, with $\Theta(1)$ extra memory.

17.2.6 Comparison with Insertion Sort

- In the worst case, **InsertionSort** executes $\Theta(n^2)$ steps, while **MergeSort** executes $\Theta(n \log_2 n)$ steps. Since

$$\lim_{n \rightarrow \infty} \frac{n \log_2 n}{n^2} = \lim_{n \rightarrow \infty} \frac{\log_2 n}{n} = 0,$$

Merge Sort is asymptotically much faster.

- To get a feeling of the difference, we ignore the multiplicative constants and assume that a computer executes 10^9 steps per second:

n	n^2	$n \log_2 n$	time for n^2 steps	time for $n \log_2 n$ steps
10	100	≈ 33	10^{-7} s	$\approx 3.3 \cdot 10^{-8}$ s
10^3	10^6	≈ 9966	10^{-3} s	$\approx 10^{-5}$ s
10^6	10^{12}	$\approx 1.99 \cdot 10^7$	10^3 s ≈ 16.7 min	≈ 0.02 s

- For small arrays the difference is negligible (and, because of the constants and of the cost of the recursive calls, **InsertionSort** can even be faster). For large arrays, Merge Sort is dramatically faster: with $n = 10^6$ elements it is about 50 000 times faster.

17.2.7 C++ implementation

- This is a C++ implementation of the Merge Sort algorithm. Arrays are indexed from 0: the whole array is sorted by the call `MergeSort(a, 0, n-1)`. The auxiliary arrays are `std::vectors`, which are automatically released at the end of `Merge`.

```

1  #include <vector>
2
3  // Merges the sorted sub-arrays a[l..m] and a[m+1..r] (0-indexed)
4  void Merge(int *a, int l, int m, int r)
5  {
6      int n1 = m - l + 1; // Size of the left part
7      int n2 = r - m;      // Size of the right part
8
9      std::vector<int> L(n1), R(n2); // Auxiliary arrays
10
11     for (int i = 0; i < n1; i++)
12     {
13         L[i] = a[l + i];
14     }
15     for (int j = 0; j < n2; j++)
16     {
17         R[j] = a[m + 1 + j];
18     }
19
20     int i = 0, j = 0, k = l;
21
22     while (i < n1 && j < n2) // Merge L and R back into a[l..r]
23     {
24         if (L[i] <= R[j])
25         {
26             a[k] = L[i];
27             i++;
28         }
29         else
30         {
31             a[k] = R[j];
32             j++;
33         }
34         k++;
35     }
36
37     while (i < n1) // Copy the remaining elements of L
38     {
39         a[k] = L[i];
40         i++;
41         k++;
42     }
43
44     while (j < n2) // Copy the remaining elements of R
45     {
46         a[k] = R[j];
47         j++;
48         k++;
49     }
50 }
51

```

```

52 // Sorts a[l..r] in non-decreasing order (0-indexed)
53 void MergeSort(int *a, int l, int r)
54 {
55     if (l < r)
56     {
57         int m = l + (r - l) / 2; // Equivalent to (l + r) / 2
58
59         MergeSort(a, l, m);
60         MergeSort(a, m + 1, r);
61         Merge(a, l, m, r);
62     }
63 }

```

Example 17.8: execution of the C++ implementation of Merge

We execute the call `Merge(a, 0, 1, 3)` on the array $a = [5, 8, 2, 7]$: the sub-arrays $a[0..1] = [5, 8]$ and $a[2..3] = [2, 7]$ are sorted.

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$l = 0$	$m = 1$	$r = 3$
	5	8	2	7			

After the copies,

$$n1 = 1 - 0 + 1 = 2, \quad n2 = 3 - 1 = 2, \quad L = [5, 8], \quad R = [2, 7], \quad i = 0, \quad j = 0, \quad k = 0.$$

- **Iteration 1** (first while loop). $L[0] = 5 \leq R[0] = 2$? No $\Rightarrow a[0] = R[0] = 2, j = 1, k = 1$; now $a = [2, 8, 2, 7]$.
- **Iteration 2** (first while loop). $L[0] = 5 \leq R[1] = 7$? Yes $\Rightarrow a[1] = L[0] = 5, i = 1, k = 2$; now $a = [2, 5, 2, 7]$.
- **Iteration 3** (first while loop). $L[1] = 8 \leq R[1] = 7$? No $\Rightarrow a[2] = R[1] = 7, j = 2, k = 3$; now $a = [2, 5, 7, 7]$.
- The condition $j < n2$ is false ($j = 2$): R is exhausted and the first while loop ends.
- **Iteration 1** (copy the rest of L). $a[3] = L[1] = 8, i = 2, k = 4$; now $a = [2, 5, 7, 8]$.
- The conditions $i < n1$ and $j < n2$ are both false: **Merge** terminates.

The algorithm terminates with $a = [2, 5, 7, 8]$, after $3 = n - 1$ comparisons. Note that the original values $a[0] = 5$ and $a[1] = 8$ are overwritten during the execution: this is why the copies L and R are necessary.

Key points on Merge Sort.

- All the work is done by **Merge**, which merges two sorted sub-arrays of total size n in $6n + t + 11 = \Theta(n)$ steps.
- The recurrence $S(n) = 2S(n/2) + \Theta(n)$ falls in case 2 of the Master Theorem: $\log_2 n + 1$ levels of cost $\Theta(n)$ each, $S(n) = \Theta(n \log_2 n)$.
- The price is memory: $\Theta(n)$ for the auxiliary arrays.

algorithm	worst-case number of steps	extra memory
InsertionSort	$\Theta(n^2)$	$\Theta(1)$
MergeSort	$\Theta(n \log_2 n)$	$\Theta(n)$

17.3 Recursive Binary Search

Merge Sort makes two recursive calls on the two halves. The next algorithm makes only **one**: after a single comparison, one of the two halves can be discarded.

17.3.1 The algorithm

Given an array $a[1..n]$ of n natural numbers sorted in non-decreasing order and a target natural number m , we consider the problem of determining if m is one of the elements of the array.

- In the chapter «Linear and binary search» we solved this problem with the **iterative** algorithm **BinarySearch**: the algorithm maintains a search interval $a[left..right]$, initially $a[1..n]$, and at each iteration it compares the target m with the middle element

$$a[mid], \quad mid = left + \left\lfloor \frac{right - left}{2} \right\rfloor.$$

If $a[mid] = m$ the element is found; otherwise, since the array is sorted, half of the interval can be discarded.

- The key observation is that, after the comparison with $a[mid]$, the problem that remains to be solved is **the same problem** (search m in a sorted array) on a **smaller** sub-array:
 - if $a[mid] < m$, then m can only be in $a[mid + 1..right]$;
 - if $a[mid] > m$, then m can only be in $a[left..mid - 1]$.
- This is exactly the structure required by a recursive algorithm: we solve the problem on $a[left..right]$ by solving the same problem on a sub-array of (about) half the size. The **base cases** are the empty interval ($left > right$), in which m is certainly not present, and the case $a[mid] = m$, in which m is found.
- Differently from the iterative version, the boundaries $left$ and $right$ are now **parameters** of the function. To search m in the whole array, we call

$$\text{RecursiveBinarySearch}(a, 1, n, m).$$

- This is the pseudo-code of the Recursive Binary Search algorithm:

Algorithm: RecursiveBinarySearch

Input : An array $a[1..n]$ of n natural numbers sorted in non-decreasing order, two indices $left$ and $right$, and a target value m

Output: An index $i \in \{left, \dots, right\}$ such that $a[i] = m$, or -1 if m is not in $a[left..right]$

```

1 function RecursiveBinarySearch(a, left, right, m)
2   if left > right then
3     | return -1                                // Empty interval: not found
4   end
5   mid := left + ⌊(right - left)/2⌋              // Middle index
6   if a[mid] = m then
7     | return mid                              // Element found
8   end
9   if a[mid] < m then
10    | return RecursiveBinarySearch(a, mid + 1, right, m)
11  end
12  return RecursiveBinarySearch(a, left, mid - 1, m)
13 end

```

- Explanation of the algorithm:

- **Lines 2-3:** if the interval is empty ($left > right$), the element is not present and -1 is returned (**base case**).
- **Line 5:** the middle index mid is computed with the same formula as in the iterative algorithm.
- **Lines 6-7:** if $a[mid] = m$, the element is found and its index is returned (**base case**).
- **Lines 9-10:** if $a[mid] < m$, the search continues **recursively** in the right half $a[mid+1..right]$.
- **Line 12:** otherwise $a[mid] > m$, and the search continues **recursively** in the left half $a[left..mid-1]$.
- The value returned by the recursive call is returned unchanged to the caller.

- Each call of **RecursiveBinarySearch** performs **at most one** recursive call. Therefore, the **call tree** is a **path**: it has no branching. This is very different from Merge Sort, where every call on an array with at least two elements generates two recursive calls.

17.3.2 Examples

- We consider the same searches performed in the chapter «Linear and binary search» with the iterative algorithm. For each search we report one row for each call: the values of *left*, *right*, *mid*, $a[mid]$ and the action executed. Then, we draw the call tree: a black arrow is a recursive call, a blue arrow is the value returned to the caller.

Example 17.9: Recursive Binary Search on a sorted array

We search four values in the sorted array $a = [3, 9, 11, 12, 12, 16, 19]$ ($n = 7$).

- Search for $m = 12$.** First call `RecursiveBinarySearch(a, 1, 7, 12)`:

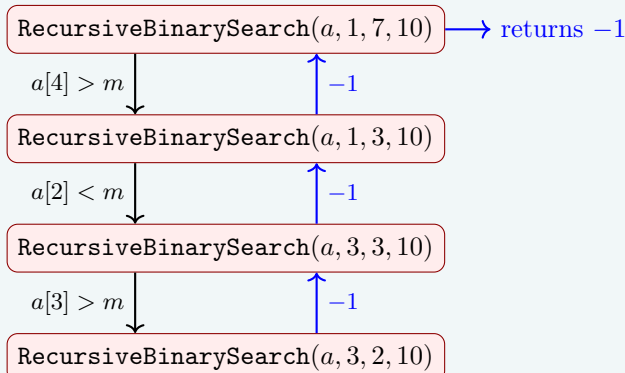
call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	7	4	12	$a[mid] = m \Rightarrow$ Found! Return 4

`RecursiveBinarySearch(a, 1, 7, 12)` → returns 4

Element 12 is found at index 4 with a single call (no recursion).

- Search for $m = 10$.** First call `RecursiveBinarySearch(a, 1, 7, 10)`:

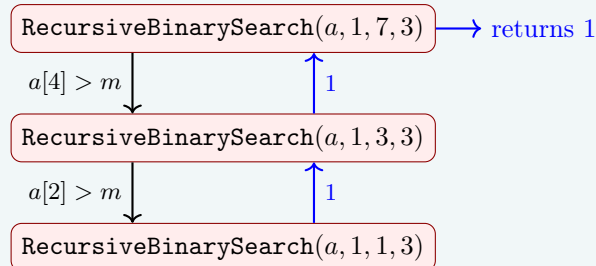
call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	7	4	12	$a[mid] > m \Rightarrow$ call on $left = 1, right = 3$
2	1	3	2	9	$a[mid] < m \Rightarrow$ call on $left = 3, right = 3$
3	3	3	3	11	$a[mid] > m \Rightarrow$ call on $left = 3, right = 2$
4	3	2	–	–	$left > right \Rightarrow$ Not found! Return -1



Element 10 is not in the array: the last call is on an empty interval and returns -1 , and this value is propagated back to the first call.

- Search for $m = 3$.** First call `RecursiveBinarySearch(a, 1, 7, 3)`:

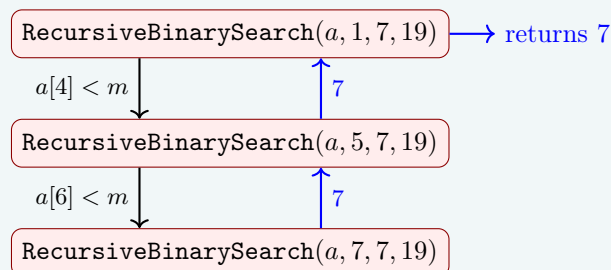
call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	7	4	12	$a[mid] > m \Rightarrow$ call on $left = 1, right = 3$
2	1	3	2	9	$a[mid] > m \Rightarrow$ call on $left = 1, right = 1$
3	1	1	1	3	$a[mid] = m \Rightarrow$ Found! Return 1



Element 3 is found at index 1.

- **Search for $m = 19$.** First call `RecursiveBinarySearch(a, 1, 7, 19)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	7	4	12	$a[mid] < m \Rightarrow$ call on $left = 5, right = 7$
2	5	7	6	16	$a[mid] < m \Rightarrow$ call on $left = 7, right = 7$
3	7	7	7	19	$a[mid] = m \Rightarrow$ Found! Return 7



Element 19 is found at index 7.

- The sequence of values $(left, right, mid)$ is **exactly the same** as the sequence produced by the iterative algorithm: each call of the recursive algorithm corresponds to one iteration of the **while** loop of the iterative algorithm.
- Note that the second occurrence of 12 (index 5) is never found when searching $m = 12$: as in the iterative version, the algorithm returns **one** index i with $a[i] = m$, not necessarily the first one.

17.3.3 Correctness

- As for the previous algorithms, the correctness is proved by **strong induction** on the size of the input of the call. The size of the call `RecursiveBinarySearch(a, left, right, m)` is the number of elements of the interval:

$$k = \text{right} - \text{left} + 1.$$

For an empty interval ($\text{left} = \text{right} + 1$) we have $k = 0$.

Theorem 17.6: correctness of Recursive Binary Search

Let $a[1..n]$ be sorted in non-decreasing order and let $1 \leq \text{left} \leq \text{right} + 1 \leq n + 1$. Then `RecursiveBinarySearch(a, left, right, m)` terminates and returns:

- an index $i \in \{\text{left}, \dots, \text{right}\}$ such that $a[i] = m$, if m is one of the elements of $a[\text{left}..\text{right}]$;
- -1 , otherwise.

Proof. We proceed by strong induction on $k = \text{right} - \text{left} + 1 \geq 0$.

Base case $k = 0$. Then $\text{left} > \text{right}$, the interval is empty and m is not one of its elements. The algorithm executes **lines 2–3** and returns -1 , which is correct.

Inductive step. Let $k \geq 1$ and assume (inductive hypothesis) that the statement holds for every call whose interval has size $k' < k$. Since $\text{left} \leq \text{right}$, the algorithm computes

$$\text{mid} = \text{left} + \left\lfloor \frac{\text{right} - \text{left}}{2} \right\rfloor, \quad \text{left} \leq \text{mid} \leq \text{right}.$$

- If $a[\text{mid}] = m$, the algorithm returns $\text{mid} \in \{\text{left}, \dots, \text{right}\}$, which is correct.
- If $a[\text{mid}] < m$, then, since the array is sorted, $a[i] \leq a[\text{mid}] < m$ for every $i \in \{\text{left}, \dots, \text{mid}\}$. Hence m is in $a[\text{left}..\text{right}]$ if and only if it is in $a[\text{mid} + 1..\text{right}]$. This interval has size $\text{right} - \text{mid} < k$ (because $\text{mid} \geq \text{left}$), so by the inductive hypothesis the call `RecursiveBinarySearch(a, mid + 1, right, m)` terminates and returns a correct answer, which is returned unchanged.
- If $a[\text{mid}] > m$, then $a[i] \geq a[\text{mid}] > m$ for every $i \in \{\text{mid}, \dots, \text{right}\}$. Hence m is in $a[\text{left}..\text{right}]$ if and only if it is in $a[\text{left}..\text{mid} - 1]$, an interval of size $\text{mid} - \text{left} < k$ (because $\text{mid} \leq \text{right}$). By the inductive hypothesis, the recursive call returns a correct answer.

In all cases the call terminates and returns a correct answer. □

- Applying Theorem 17.6 with $\text{left} = 1$ and $\text{right} = n$, we obtain that `RecursiveBinarySearch(a, 1, n, m)` solves the problem.

17.3.4 Worst-case running time

17.3.4.1 The recurrence

- Let $S(n)$ denote the number of steps executed by `RecursiveBinarySearch` in the worst case on an interval of $n = right - left + 1$ elements (including the steps executed by all the recursive calls).
- The number of steps executed by a single call, **without** counting the steps of the recursive call, is:
 - empty interval: **lines** 2 and 3 $\Rightarrow$ 2 steps;
 - element found: **lines** 2, 5, 6, 7 $\Rightarrow$ 4 steps;
 - recursive call: **lines** 2, 5, 6, 9 and **line** 10 (or 12) $\Rightarrow$ 5 steps.
- **Size of the sub-problems.** Let $n = right - left + 1 \geq 1$. Since $mid - left = \lfloor (n-1)/2 \rfloor$:
 - the left interval $a[left..mid - 1]$ has $mid - left = \lfloor \frac{n-1}{2} \rfloor$ elements;
 - the right interval $a[mid + 1..right]$ has $right - mid = (n-1) - \lfloor \frac{n-1}{2} \rfloor = \lceil \frac{n-1}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ elements.

For example, for $n = 7$ both halves have 3 elements; for $n = 8$ the left half has 3 elements and the right half has 4.

- In the worst case the element is not found before the interval becomes empty, and the recursion continues on the larger half, which has $\lfloor n/2 \rfloor$ elements (one can easily check by induction that S is non-decreasing). Therefore, $S(n)$ satisfies the recurrence

$$S(n) = \begin{cases} 2 & \text{if } n = 0, \\ S(\lfloor n/2 \rfloor) + 5 & \text{if } n \geq 1. \end{cases}$$

This worst case happens, for example, when $m > a[n]$: the algorithm always continues in the right half, until the interval becomes empty.

- In general, we write

$$S(n) = S(\lfloor n/2 \rfloor) + c,$$

where c is a constant (here $c = 5$): a recursive algorithm that reduces the problem to **one** sub-problem of **half** the size, doing a **constant** amount of extra work.

17.3.4.2 Solution by expansion

- We expand the recurrence, using the property $\lfloor \lfloor n/2^i \rfloor / 2 \rfloor = \lfloor n/2^{i+1} \rfloor$:

$$\begin{aligned} S(n) &= S(\lfloor n/2 \rfloor) + 5 \\ &= S(\lfloor n/4 \rfloor) + 5 + 5 \\ &= S(\lfloor n/8 \rfloor) + 5 \cdot 3 \\ &\vdots \\ &= S(\lfloor n/2^j \rfloor) + 5j. \end{aligned}$$

- The expansion stops at the first j such that $\lfloor n/2^j \rfloor = 0$, that is $2^j > n$. For $n \geq 1$, this is

$$j = \lfloor \log_2 n \rfloor + 1,$$

because $2^{\lfloor \log_2 n \rfloor} \leq n < 2^{\lfloor \log_2 n \rfloor + 1}$. Hence, for every $n \geq 1$,

$$S(n) = 5(\lfloor \log_2 n \rfloor + 1) + 2.$$

- For example, $S(7) = 5 \cdot (2 + 1) + 2 = 17$: in the search for $m = 10$ in Example 17.9, the three calls on non-empty intervals execute 5 steps each and the last call (empty interval) executes 2 steps.
- Since $\log_2 n - 1 < \lfloor \log_2 n \rfloor \leq \log_2 n$, we have

$$\lim_{n \rightarrow \infty} \frac{S(n)}{\log_2 n} = \lim_{n \rightarrow \infty} \frac{5\lfloor \log_2 n \rfloor + 7}{\log_2 n} = 5 \in \mathbb{R}_{>0}, \quad \text{therefore} \quad S(n) = \Theta(\log_2 n).$$

17.3.4.3 Solution by the Master Theorem

- The same result is obtained immediately with the Master Theorem, applied to

$$S(n) = S(n/2) + c$$

(as usual, floors do not change the asymptotic solution).

- **Identify** $a, b, f(n)$: $a = 1, b = 2, f(n) = c$.
- **Compute** $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$.
- **Compare**: $f(n) = c = \Theta(1) = \Theta(n^{\log_2 1} \log^0 n)$.
- **Verify the hypothesis**: we are in case 2 of the Master Theorem with $k = 0$.
- **Conclude**:

$$S(n) = \Theta(n^{\log_2 1} \log^{0+1} n) = \Theta(\log n) = \Theta(\log_2 n).$$

17.3.4.4 Number of recursive calls

- Each call on a non-empty interval compares m with exactly one element $a[\text{mid}]$. Therefore, the number of calls on a non-empty interval is the number of elements of the array examined by the algorithm (exactly as the number of iterations of the **while** loop of the iterative algorithm).

Observation 17.7: number of calls of Recursive Binary Search

For $n \geq 1$, the call `RecursiveBinarySearch(a, 1, n, m)` executes at most

$$\lfloor \log_2 n \rfloor + 1$$

calls on a non-empty interval, and this bound is attained (for example when $m > a[n]$). Counting also the final call on an empty interval, the total number of calls is at most $\lfloor \log_2 n \rfloor + 2$.

Proof. Let n_i be the size of the interval of the i -th call ($i = 1$ is the first call, so $n_1 = n$). By the analysis of the sub-problems, $n_{i+1} \leq \lfloor n_i/2 \rfloor$. By induction on i ,

$$n_i \leq \left\lfloor \frac{n}{2^{i-1}} \right\rfloor :$$

it holds for $i = 1$, and if it holds for i then $n_{i+1} \leq \lfloor n_i/2 \rfloor \leq \lfloor \lfloor n/2^{i-1} \rfloor / 2 \rfloor = \lfloor n/2^i \rfloor$.

If the i -th call is on a non-empty interval, then $n_i \geq 1$, hence $n/2^{i-1} \geq 1$, i.e., $2^{i-1} \leq n$ and $i - 1 \leq \log_2 n$. Since $i - 1$ is an integer, $i - 1 \leq \lfloor \log_2 n \rfloor$, i.e.,

$$i \leq \lfloor \log_2 n \rfloor + 1.$$

Thus, at most $\lfloor \log_2 n \rfloor + 1$ calls are on a non-empty interval, and at most one further call (on an empty interval, which makes no recursive call) can follow.

If $m > a[n]$, the algorithm always continues in the right half, whose size is exactly $\lfloor n_i/2 \rfloor$. Hence $n_i = \lfloor n/2^{i-1} \rfloor$, which is ≥ 1 for $i = \lfloor \log_2 n \rfloor + 1$: the bound is attained, and it is followed by one call on an empty interval. $\square$

- For example, for $n = 7$ we have $\lfloor \log_2 7 \rfloor + 1 = 3$: in the search of $m = 10$ there are 3 calls on non-empty intervals, plus the final call on the empty interval *left* = 3, *right* = 2.
- The same reasoning shows that also the iterative algorithm executes at most $\lfloor \log_2 n \rfloor + 1$ iterations of the **while** loop.

17.3.4.5 Worst-case running time as a function of the input size

- The input of the algorithm consists of the array $a[1..n]$ and of the target m . Under the hypothesis that every value uses a constant number of bits, the input size satisfies

$$|I| = \Theta(n),$$

that is, there exist constants $c_1, c_2 > 0$ such that $c_1 n \leq |I| \leq c_2 n$ for all sufficiently large n . Taking logarithms,

$$\log_2 |I| - \log_2 c_2 \leq \log_2 n \leq \log_2 |I| - \log_2 c_1,$$

hence $\log_2 n = \Theta(\log_2 |I|)$ (the additive constants are negligible with respect to $\log_2 |I|$).

- Since $S(n) = \Theta(\log_2 n)$, we obtain $S(n) = \Theta(\log_2 |I|)$.

If all input values use a fixed constant number of bits, then the worst-case running time of the **RecursiveBinarySearch** algorithm is

$$T(|I|) = \Theta(\log |I|).$$

Thus, as the iterative version, the algorithm is logarithmic in the input size: it is **not** necessary to read the whole input to solve the problem, because the array is sorted.

17.3.5 Space: recursive versus iterative Binary Search

- The recursive and the iterative algorithms execute the same comparisons and have the same worst-case running time. They differ in the **memory** used.
- Every call that has not yet terminated keeps in memory its own copy of the parameters and local variables (*left*, *right*, *m*, *mid*, a reference to the array, and the return address). This information is stored in the **call stack**, and it occupies a constant amount of memory per call.
- In `RecursiveBinarySearch` no call terminates before its recursive call has terminated. Therefore, when the last call is executing, **all** the calls of the path are in the call stack. By Observation 17.7, in the worst case there are $\lfloor \log_2 n \rfloor + 2$ of them, and the extra memory used is

$$\Theta(\log n).$$

- The iterative algorithm uses only the variables *left*, *right* and *mid*: its extra memory is

$$\Theta(1).$$

Definition 17.8: tail recursion

A recursive call is a **tail call** if it is the last operation executed by the calling function, i.e., the value it returns is directly returned by the caller without any further computation. A function whose recursive calls are all tail calls is called **tail recursive**.

- `RecursiveBinarySearch` is tail recursive: in **lines** 10 and 12 the value of the recursive call is returned as it is.
- A tail call can be replaced by updating the parameters and jumping back to the beginning of the function:
 - **line** 10 becomes $left \leftarrow mid + 1$;
 - **line** 12 becomes $right \leftarrow mid - 1$;
 - the test of **line** 2 becomes the condition of a **while** loop “**while** $left \leq right$ ”.

The result of this transformation is **exactly** the iterative `BinarySearch` algorithm. This is why the two algorithms produce the same sequence of values of *left*, *right* and *mid*.

- Many compilers perform this transformation automatically (**tail-call elimination**), so that the recursive version uses only $\Theta(1)$ extra memory. For instance, the compiler g++ 15.2 removes the recursive calls of the C++ implementation below with the option `-O2`, but not with `-O0`. However, this is not guaranteed by the C++ standard.

Recursive and iterative Binary Search have the same worst-case running time $\Theta(\log n)$, but the recursive version uses $\Theta(\log n)$ extra memory for the call stack, while the iterative version uses $\Theta(1)$ extra memory. Every tail-recursive algorithm can be transformed into an iterative one.

17.3.6 C++ implementation

- This is a C++ implementation of the Recursive Binary Search algorithm. As usual in C++, the array is indexed from 0: to search m in the whole array of n elements we call `RecursiveBinarySearch(a, 0, n-1, m)`.

```

1 // Returns an index of m in a[left..right], or -1 if not found
2 // The array a must be sorted in non-decreasing order
3
4 int RecursiveBinarySearch(int *a, int left, int right, int m)
5 {
6     if (left > right)
7     {
8         return -1; // Empty interval: element not found
9     }
10
11     int mid = left + (right - left) / 2;
12
13     if (a[mid] == m)
14     {
15         return mid; // Element found at index mid
16     }
17
18     if (a[mid] < m)
19     {
20         return RecursiveBinarySearch(a, mid + 1, right, m); // Right half
21     }
22
23     return RecursiveBinarySearch(a, left, mid - 1, m); // Left half
24 }
25
26 int main()
27 {
28     int a[] = {3, 9, 11, 12, 12, 16, 19};
29     int n = 7;
30     int result = RecursiveBinarySearch(a, 0, n - 1, 10); // result = -1
31     return 0;
32 }

```

Example 17.10: execution of the C++ implementation of RecursiveBinarySearch

We search $m = 10$ and $m = 19$ in the sorted array $a = [3, 9, 11, 12, 12, 16, 19]$:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
	3	9	11	12	12	16	19

Search for $m = 10$.

- **Call 1**, `RecursiveBinarySearch(a, 0, 6, 10)`: $left = 0 \leq right = 6$; $mid = 0 + (6 - 0)/2 = 3$; $a[3] = 12 > 10 \Rightarrow$ call `RecursiveBinarySearch(a, 0, 2, 10)`.
- **Call 2**, `RecursiveBinarySearch(a, 0, 2, 10)`: $left = 0 \leq right = 2$; $mid = 0 + (2 - 0)/2 = 1$; $a[1] = 9 < 10 \Rightarrow$ call `RecursiveBinarySearch(a, 2, 2, 10)`.

- **Call 3**, `RecursiveBinarySearch(a, 2, 2, 10)`: $left = 2 \leq right = 2$; $mid = 2 + (2 - 2)/2 = 2$; $a[2] = 11 > 10 \Rightarrow$ call `RecursiveBinarySearch(a, 2, 1, 10)`.
- **Call 4**, `RecursiveBinarySearch(a, 2, 1, 10)`: $left = 2 > right = 1$: the interval is empty $\Rightarrow$ return -1 .

The value -1 is returned to call 3, which returns it to call 2, which returns it to call 1, which returns it to `main`. At the moment of the deepest call, 4 calls are simultaneously in the call stack.

Search for $m = 19$.

- **Call 1**, `RecursiveBinarySearch(a, 0, 6, 19)`: $mid = 3$; $a[3] = 12 < 19 \Rightarrow$ call `RecursiveBinarySearch(a, 4, 6, 19)`.
- **Call 2**, `RecursiveBinarySearch(a, 4, 6, 19)`: $mid = 4 + (6 - 4)/2 = 5$; $a[5] = 16 < 19 \Rightarrow$ call `RecursiveBinarySearch(a, 6, 6, 19)`.
- **Call 3**, `RecursiveBinarySearch(a, 6, 6, 19)`: $mid = 6$; $a[6] = 19 = m \Rightarrow$ return 6.

The value 6 is returned to call 2, which returns it to call 1, which returns it to `main`. The element is found at index 6 (position 7 in the pseudo-code, where the array is indexed from 1).

Key points on the Recursive Binary Search.

- One recursive call on (at most) half of the interval: the call tree is a path, with at most $\lfloor \log_2 n \rfloor + 2$ calls.
- $S(n) = S(\lfloor n/2 \rfloor) + 5$, i.e., $S(n) = 5(\lfloor \log_2 n \rfloor + 1) + 2 = \Theta(\log n)$ (case 2 of the Master Theorem with $a = 1$).
- The algorithm is tail recursive: removing the recursion gives the iterative `BinarySearch`, with $\Theta(1)$ extra memory instead of $\Theta(\log n)$.

17.4 The maximum subarray problem

The last problem of the chapter shows divide and conquer at work on a less obvious problem, where the *combine* step is the interesting part. It also shows that a recursive algorithm that improves on brute force is not necessarily the best possible one.

17.4.1 Motivation: buying and selling a stock

- Suppose we know in advance the price of a stock for the next days. We may buy one unit of stock **once** and sell it **once**, at a later day. We want to maximise the profit (sell price minus buy price).
- This is an example¹: $p[d]$ is the price at day d , and $a[d] = p[d] - p[d-1]$ is the **daily change** of the price.

day d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
price $p[d]$	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
change $a[d]$		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- If we buy at day $l-1$ and sell at day $r \geq l$, the profit is

$$p[r] - p[l-1] = (p[l] - p[l-1]) + (p[l+1] - p[l]) + \cdots + (p[r] - p[r-1]) = a[l] + a[l+1] + \cdots + a[r],$$

i.e., the sum of the consecutive daily changes $a[l..r]$. Hence, maximising the profit is equivalent to finding the consecutive portion of the array $a[1..16]$ with the largest sum.

- Note that it is **not** enough to buy at the lowest price or to sell at the highest price: the lowest price (63, day 7) comes after the highest price (113, day 1). In this example the best choice is to buy at day 7 (price 63) and sell at day 11 (price 106), with profit

$$106 - 63 = 43 = a[8] + a[9] + a[10] + a[11] = 18 + 20 - 7 + 12.$$

17.4.2 The problem

Given an array $a[1..n]$ of n integer numbers, consider the problem of finding two indices $l \leq r$ such that the sum

$$a[l] + a[l+1] + \cdots + a[r] = \sum_{k=l}^r a[k]$$

is maximum. The sub-array $a[l..r]$ is called a **maximum subarray** of a .

- A subarray is always **non-empty** ($l \leq r$) and made of **consecutive** elements.
- The problem is interesting only if the array contains negative numbers: if all the entries are non-negative, the whole array $a[1..n]$ is a maximum subarray. If all the entries are negative, a maximum subarray is made of a single element, the largest one.

¹The example is taken from the textbook of T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Section 4.1.

- A maximum subarray is not necessarily unique: we look for **one** of them.
- The number of subarrays of $a[1..n]$ is the number of pairs (l, r) with $1 \leq l \leq r \leq n$:

$$\sum_{l=1}^n (n - l + 1) = n + (n - 1) + \dots + 1 = \frac{n(n + 1)}{2}.$$

17.4.3 A brute force algorithm

- The simplest algorithm considers all the $n(n + 1)/2$ subarrays $a[l..r]$ and keeps the best one. For a fixed l , the sums of $a[l..r]$ for $r = l, l + 1, \dots, n$ are computed incrementally: the sum of $a[l..r]$ is the sum of $a[l..r - 1]$ plus $a[r]$.
- This is the pseudo-code of the `BruteForceMaximumSubarray` algorithm:

Algorithm: `BruteForceMaximumSubarray`

Input : An array $a[1..n]$ of n integer numbers

Output: Indices $bl \leq br$ of a maximum subarray and its sum $best$

```

1 function BruteForceMaximumSubarray( $a, n$ )
2    $best := a[1]$ 
3    $bl := 1$ 
4    $br := 1$ 
5    $l := 1$ 
6   while  $l \leq n$  do
7      $sum := 0$ 
8      $r := l$ 
9     while  $r \leq n$  do
10       $sum \leftarrow sum + a[r]$                                 // sum of  $a[l..r]$ 
11      if  $sum > best$  then
12         $best \leftarrow sum$ 
13         $bl \leftarrow l$ 
14         $br \leftarrow r$ 
15      end
16       $r \leftarrow r + 1$ 
17    end
18     $l \leftarrow l + 1$ 
19  end
20  return ( $bl, br, best$ )
21 end

```

Example 17.11: brute force algorithm

Consider $a = [-2, 5, -1, 3]$. The algorithm starts with $best = a[1] = -2$ and examines the $4 \cdot 5/2 = 10$ subarrays in this order:

l	r	sum	action
1	1	-2	$sum \leq best = -2$
1	2	3	$sum > best \Rightarrow best = 3, bl = 1, br = 2$
1	3	2	
1	4	5	$sum > best \Rightarrow best = 5, bl = 1, br = 4$
2	2	5	$sum \leq best = 5$
2	3	4	
2	4	7	$sum > best \Rightarrow best = 7, bl = 2, br = 4$
3	3	-1	
3	4	2	
4	4	3	

The algorithm returns (2, 4, 7): the maximum subarray is $a[2..4] = [5, -1, 3]$ with sum 7.

17.4.3.1 Worst-case running time

- Let $S(n)$ denote the number of steps executed by **BruteForceMaximumSubarray** in the worst case, where n is the size of the array.
- The execution of **lines** 12–14 depends on the input. We define a function $t(l, r) \in \{0, 1\}$, for $1 \leq l \leq r \leq n$, where $t(l, r) = 1$ if **lines** 12–14 are executed when the inner loop examines the pair (l, r) , and $t(l, r) = 0$ otherwise.
- For a fixed l , the inner loop executes $n - l + 1$ iterations, and its condition is tested $n - l + 2$ times. Hence

$$\sum_{l=1}^n (n - l + 1) = \frac{n(n+1)}{2}, \quad \sum_{l=1}^n (n - l + 2) = \frac{n(n+1)}{2} + n.$$

- The number of times each line of code is executed is:
 - **Lines** 2, 3, 4, 5 are executed 1 time each
 - **Line** 6 is executed $n + 1$ times
 - **Lines** 7 and 8 are executed n times each
 - **Line** 9 is executed $\frac{n(n+1)}{2} + n$ times
 - **Lines** 10, 11 and 16 are executed $\frac{n(n+1)}{2}$ times each
 - **Lines** 12, 13, 14 are executed $\sum_{l=1}^n \sum_{r=l}^n t(l, r)$ times each
 - **Line** 18 is executed n times
 - **Line** 20 is executed 1 time
- Therefore, the total number of steps is

$$\begin{aligned}
 & 4 + (n + 1) + 2n + \frac{n(n+1)}{2} + n + 3 \frac{n(n+1)}{2} + n + 1 + 3 \sum_{l=1}^n \sum_{r=l}^n t(l, r) \\
 &= 2n^2 + 7n + 6 + 3 \sum_{l=1}^n \sum_{r=l}^n t(l, r).
 \end{aligned}$$

In Example 17.11 there are 3 updates and the algorithm executes $2 \cdot 16 + 7 \cdot 4 + 6 + 3 \cdot 3 = 75$ steps.

- Since $0 \leq t(l, r) \leq 1$, the double sum is between 0 and $n(n+1)/2$. Hence, for every input of size n , the number of steps is between

$$2n^2 + 7n + 6 \quad \text{and} \quad 2n^2 + 7n + 6 + \frac{3n(n+1)}{2} = \frac{7}{2}n^2 + \frac{17}{2}n + 6,$$

and so is the worst-case number of steps $S(n)$. Both bounds are $\Theta(n^2)$; for example,

$$\lim_{n \rightarrow \infty} \frac{2n^2 + 7n + 6}{n^2} = 2 \in \mathbb{R}_{>0}, \quad \lim_{n \rightarrow \infty} \frac{\frac{7}{2}n^2 + \frac{17}{2}n + 6}{n^2} = \frac{7}{2} \in \mathbb{R}_{>0}.$$

Therefore,

$$S(n) = \Theta(n^2).$$

Note that, for **every** input (not only in the worst case), the algorithm executes $\Theta(n^2)$ steps, since it always examines all the $n(n+1)/2$ subarrays.

- Under the hypothesis that every entry uses a constant number of bits, $|I| = \Theta(n)$, and therefore $S(n) = \Theta(|I|^2)$.

If all input values use a fixed constant number of bits, then the worst-case running time of the `BruteForceMaximumSubarray` algorithm is

$$T(|I|) = \Theta(|I|^2).$$

Thus, the brute force algorithm is quadratic in the input size.

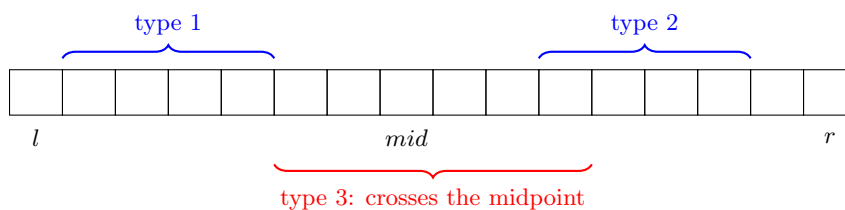
17.4.4 A divide and conquer algorithm

- We now design a recursive algorithm based on the **divide and conquer** approach, as for Merge Sort. We solve the problem on a sub-array $a[l..r]$, and we initially call it with $l = 1$ and $r = n$.
- **Divide**: split $a[l..r]$ into two halves $a[l..mid]$ and $a[mid+1..r]$, where

$$mid = \left\lfloor \frac{l+r}{2} \right\rfloor \quad \left(= l + \left\lfloor \frac{r-l}{2} \right\rfloor \right).$$

- Every subarray $a[i..j]$ of $a[l..r]$ (with $l \leq i \leq j \leq r$) is in exactly one of these three situations:
 1. it lies entirely in the left half: $l \leq i \leq j \leq mid$;
 2. it lies entirely in the right half: $mid+1 \leq i \leq j \leq r$;
 3. it **crosses the midpoint**: $l \leq i \leq mid < j \leq r$.
- **Conquer**: a maximum subarray of type 1 and of type 2 are found **recursively**, since they are maximum subarrays of $a[l..mid]$ and $a[mid+1..r]$, respectively.

- **Combine:** a maximum subarray of type 3 is found by a non-recursive procedure, `FindMaxCrossingSubarray`, described below. The answer is the best of the three subarrays.



17.4.4.1 Maximum subarray crossing the midpoint

- A subarray crossing the midpoint is made of two parts: $a[i..mid]$, which **ends** at mid , and $a[mid + 1..j]$, which **starts** at $mid + 1$:

$$\sum_{k=i}^j a[k] = \sum_{k=i}^{mid} a[k] + \sum_{k=mid+1}^j a[k].$$

The first term depends only on i , the second only on j : the two parts can be chosen **independently**. Therefore, it is enough to find the best i (scanning the left half from mid down to l) and the best j (scanning the right half from $mid + 1$ up to r).

- This is the pseudo-code of the `FindMaxCrossingSubarray` algorithm (from the same text-book as the stock example):

Algorithm: FindMaxCrossingSubarray**Input** : An array a and three indices $l \leq mid < r$ **Output**: Indices $maxleft \leq mid < maxright$ of a maximum subarray crossing the midpoint, and its sum

```

1 function FindMaxCrossingSubarray( $a, l, mid, r$ )
2    $leftsum := -\infty$ 
3    $sum := 0$ 
4   for  $i := mid$  downto  $l$  do
5      $sum \leftarrow sum + a[i]$                                 // sum of  $a[i..mid]$ 
6     if  $sum > leftsum$  then
7        $leftsum \leftarrow sum$ 
8        $maxleft \leftarrow i$ 
9     end
10  end
11   $rightsum := -\infty$ 
12   $sum \leftarrow 0$ 
13  for  $j := mid + 1$  to  $r$  do
14     $sum \leftarrow sum + a[j]$                                 // sum of  $a[mid + 1..j]$ 
15    if  $sum > rightsum$  then
16       $rightsum \leftarrow sum$ 
17       $maxright \leftarrow j$ 
18    end
19  end
20  return ( $maxleft, maxright, leftsum + rightsum$ )
21 end

```

Lemma 17.9: crossing subarray

Let $l \leq mid < r$. FindMaxCrossingSubarray(a, l, mid, r) returns ($maxleft, maxright, s$) with $l \leq maxleft \leq mid < maxright \leq r$ and

$$s = \sum_{k=maxleft}^{maxright} a[k] = \max \left\{ \sum_{k=i}^j a[k] : l \leq i \leq mid < j \leq r \right\}.$$

Moreover, it executes $\Theta(n)$ steps, where $n = r - l + 1$.

Proof. Left loop. At the end of the iteration with index i , the variable sum is equal to $L(i) = \sum_{k=i}^{mid} a[k]$, and $leftsum$ is the maximum of $L(mid), L(mid-1), \dots, L(i)$, attained at $maxleft$ (by induction on the iterations: at each iteration sum is compared with the best value found so far, which is initially $-\infty$, so the first comparison always succeeds). At the end of the loop,

$$leftsum = \max_{l \leq i \leq mid} L(i) = L(maxleft).$$

Right loop. In the same way, with $R(j) = \sum_{k=mid+1}^j a[k]$,

$$rightsum = \max_{mid+1 \leq j \leq r} R(j) = R(maxright).$$

Combination. For every crossing subarray, $\sum_{k=i}^j a[k] = L(i) + R(j)$. Since i and j can be chosen independently,

$$\max_{l \leq i \leq mid < j \leq r} (L(i) + R(j)) = \max_{l \leq i \leq mid} L(i) + \max_{mid < j \leq r} R(j) = leftsum + rightsum,$$

which is the value returned, attained by $a[maxleft..maxright]$.

Running time. The two loops execute $(mid - l + 1) + (r - mid) = r - l + 1 = n$ iterations in total, each with a constant number of steps (between 3 and 5 lines, counting the loop test), plus a constant number of steps outside the loops. Hence the number of steps is between $c_1 n$ and $c_2 n$ for two constants $c_1, c_2 > 0$, i.e., $\Theta(n)$. $\square$

- Note that the crossing subarray always contains both $a[mid]$ and $a[mid + 1]$, even if they are negative: the subarrays of type 1 and 2 take care of the other cases.

17.4.4.2 The recursive algorithm

- This is the pseudo-code of the `FindMaximumSubarray` algorithm. To solve the problem on the whole array we call `FindMaximumSubarray(a, 1, n)`.

Algorithm: FindMaximumSubarray

Input : An array a and two indices $l \leq r$

Output: Indices $i \leq j$ of a maximum subarray of $a[l..r]$ and its sum

```

1 function FindMaximumSubarray(a, l, r)
2   if l = r then
3     return (l, r, a[l])                                // Base case: one element
4   end
5   mid := ⌊(l + r)/2⌋
6   (ll, lr, lsum) := FindMaximumSubarray(a, l, mid)
7   (rl, rr, rsum) := FindMaximumSubarray(a, mid + 1, r)
8   (cl, cr, csum) := FindMaxCrossingSubarray(a, l, mid, r)
9   if lsum ≥ rsum and lsum ≥ csum then
10    return (ll, lr, lsum)
11  end
12  if rsum ≥ lsum and rsum ≥ csum then
13    return (rl, rr, rsum)
14  end
15  return (cl, cr, csum)
16 end

```

- Explanation of the algorithm:

- **Lines 2–3:** if the sub-array has only one element, its only subarray is $a[l..l]$ (**base case**).
 - **Line 5:** **divide** the sub-array into $a[l..mid]$ and $a[mid + 1..r]$.
 - **Lines 6–7:** **conquer**, i.e., find recursively a maximum subarray in each half.
 - **Line 8:** find a maximum subarray crossing the midpoint.
 - **Lines 9–15:** **combine**, i.e., return the best of the three (in case of ties, the left one is preferred, then the right one).
- Differently from the Recursive Binary Search, each call on at least two elements makes **two** recursive calls: the call tree is a binary tree with n leaves (one for each element) and $n - 1$ internal nodes, i.e., $2n - 1$ calls in total, exactly as for **RecursiveSum** (Observation 17.2).

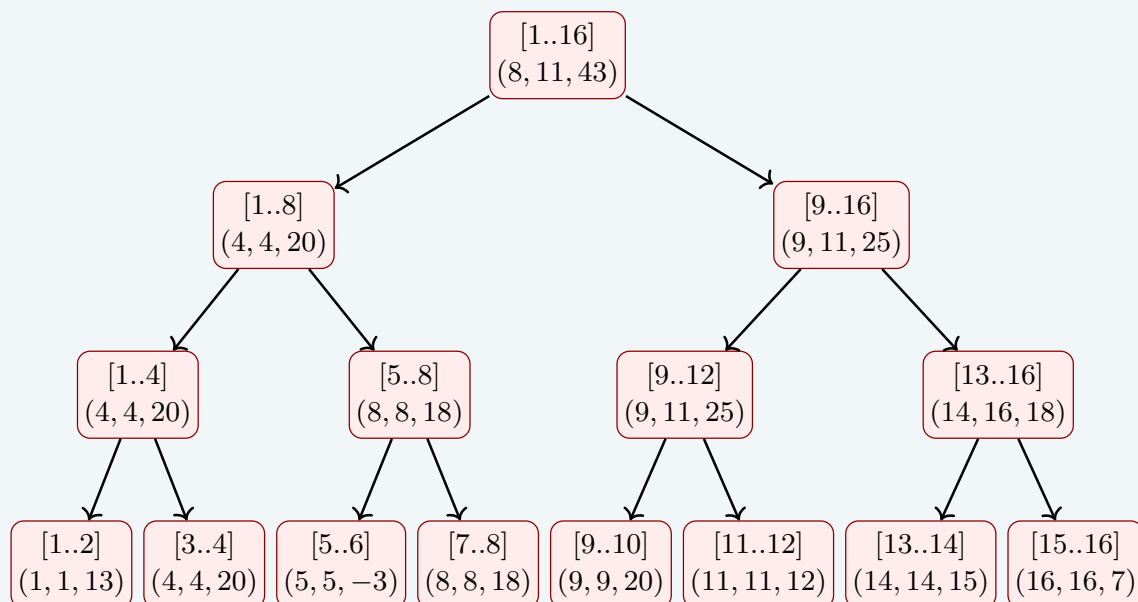
17.4.4.3 Example

Example 17.12: divide and conquer on the stock example

Consider the array of the daily changes of the stock example:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$a[i]$	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- **Step 1: the call tree.** We call $\text{FindMaximumSubarray}(a, 1, 16)$. In the following tree, each node is a call on $a[l..r]$, written as $[l..r]$, together with the triple (i, j, s) it returns. The 16 leaves (calls on one element $[k..k]$, returning $(k, k, a[k])$) are not drawn.



- **Step 2: the results of the calls.** For each call on at least two elements, we report the triples returned by the left recursive call, by the right recursive call and by $\text{FindMaxCrossingSubarray}$, and the triple returned (the best of the three).

call	<i>mid</i>	left	right	crossing	returned
[1..16]	8	(4, 4, 20)	(9, 11, 25)	(8, 11, 43)	(8, 11, 43)
[1..8]	4	(4, 4, 20)	(8, 8, 18)	(4, 5, 17)	(4, 4, 20)
[9..16]	12	(9, 11, 25)	(14, 16, 18)	(9, 16, 16)	(9, 11, 25)
[1..4]	2	(1, 1, 13)	(4, 4, 20)	(1, 4, 5)	(4, 4, 20)
[5..8]	6	(5, 5, -3)	(8, 8, 18)	(6, 8, -21)	(8, 8, 18)
[9..12]	10	(9, 9, 20)	(11, 11, 12)	(9, 11, 25)	(9, 11, 25)
[13..16]	14	(14, 14, 15)	(16, 16, 7)	(14, 16, 18)	(14, 16, 18)
[1..2]	1	(1, 1, 13)	(2, 2, -3)	(1, 2, 10)	(1, 1, 13)
[3..4]	3	(3, 3, -25)	(4, 4, 20)	(3, 4, -5)	(4, 4, 20)
[5..6]	5	(5, 5, -3)	(6, 6, -16)	(5, 6, -19)	(5, 5, -3)
[7..8]	7	(7, 7, -23)	(8, 8, 18)	(7, 8, -5)	(8, 8, 18)
[9..10]	9	(9, 9, 20)	(10, 10, -7)	(9, 10, 13)	(9, 9, 20)
[11..12]	11	(11, 11, 12)	(12, 12, -5)	(11, 12, 7)	(11, 11, 12)
[13..14]	13	(13, 13, -22)	(14, 14, 15)	(13, 14, -7)	(14, 14, 15)
[15..16]	15	(15, 15, -4)	(16, 16, 7)	(15, 16, 3)	(16, 16, 7)

- **Step 3: the crossing computations of the two calls of level 1.** Call [1..8], *mid* = 4:

<i>i</i>	<i>a</i> [<i>i</i>]	<i>sum</i>	<i>leftsum</i>	<i>maxleft</i>	<i>j</i>	<i>a</i> [<i>j</i>]	<i>sum</i>	<i>rightsum</i>	<i>maxright</i>
4	20	20	20	4	5	-3	-3	-3	5
3	-25	-5	20	4	6	-16	-19	-3	5
2	-3	-8	20	4	7	-23	-42	-3	5
1	13	5	20	4	8	18	-24	-3	5

The crossing subarray is $a[4..5]$ with $\text{sum } 20 + (-3) = 17$. Call [9..16], *mid* = 12:

<i>i</i>	<i>a</i> [<i>i</i>]	<i>sum</i>	<i>leftsum</i>	<i>maxleft</i>	<i>j</i>	<i>a</i> [<i>j</i>]	<i>sum</i>	<i>rightsum</i>	<i>maxright</i>
12	-5	-5	-5	12	13	-22	-22	-22	13
11	12	7	7	11	14	15	-7	-7	14
10	-7	0	7	11	15	-4	-11	-7	14
9	20	20	20	9	16	7	-4	-4	16

The crossing subarray is $a[9..16]$ with $\text{sum } 20 + (-4) = 16$.

- **Step 4: the crossing computation of the first call [1..16], *mid* = 8:**

<i>i</i>	<i>a</i> [<i>i</i>]	<i>sum</i>	<i>leftsum</i>	<i>maxleft</i>	<i>j</i>	<i>a</i> [<i>j</i>]	<i>sum</i>	<i>rightsum</i>	<i>maxright</i>
8	18	18	18	8	9	20	20	20	9
7	-23	-5	18	8	10	-7	13	20	9
6	-16	-21	18	8	11	12	25	25	11
5	-3	-24	18	8	12	-5	20	25	11
4	20	-4	18	8	13	-22	-2	25	11
3	-25	-29	18	8	14	15	13	25	11
2	-3	-32	18	8	15	-4	9	25	11
1	13	-19	18	8	16	7	16	25	11

The crossing subarray is $a[8..11]$ with sum $18 + 25 = 43$.

In the first call, the left half gives 20, the right half gives 25 and the crossing subarray gives 43: the algorithm returns $(8, 11, 43)$, i.e., buy at day 7 and sell at day 11, with profit 43. Note that the optimal subarray is found only at the last step, by `FindMaxCrossingSubarray`: it is neither in the left half nor in the right half.

17.4.5 Correctness

Theorem 17.10: correctness of `FindMaximumSubarray`

For every $l \leq r$, `FindMaximumSubarray`(a, l, r) returns (i, j, s) with $l \leq i \leq j \leq r$ and

$$s = \sum_{k=i}^j a[k] = \max \left\{ \sum_{k=i'}^{j'} a[k] : l \leq i' \leq j' \leq r \right\}.$$

Proof. We proceed by strong induction on the number of elements $n = r - l + 1 \geq 1$.

Base case $n = 1$. Then $l = r$, the only subarray is $a[l..l]$, and the algorithm returns $(l, l, a[l])$, which is correct.

Inductive step. Let $n \geq 2$ and assume (inductive hypothesis) that the statement holds for every sub-array with $n' < n$ elements. Since $l < r$, we have $l \leq \text{mid} < r$, and the two halves $a[l..\text{mid}]$ and $a[\text{mid} + 1..r]$ have $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ elements, both at least 1 and less than n . By the inductive hypothesis:

- $lsum$ is the maximum sum of a subarray of type 1 (contained in $a[l..\text{mid}]$), attained by $a[l..lr]$;
- $rsum$ is the maximum sum of a subarray of type 2 (contained in $a[\text{mid} + 1..r]$), attained by $a[rl..rr]$.

By Lemma 17.9, $csum$ is the maximum sum of a subarray of type 3 (crossing the mid-point), attained by $a[cl..cr]$.

Every subarray of $a[l..r]$ is of type 1, 2 or 3. Hence the maximum sum of a subarray of $a[l..r]$ is

$$\max\{lsum, rsum, csum\},$$

and lines 9–15 return exactly a triple attaining this maximum. $\square$

17.4.6 Worst-case running time

17.4.6.1 The recurrence

- Let $S(n)$ denote the number of steps executed by `FindMaximumSubarray` in the worst case on a sub-array of $n = r - l + 1$ elements.
- If $n = 1$, only **lines** 2–3 are executed: $S(1) = c_1$, a constant.
- If $n \geq 2$:
 - **lines** 2, 5 and 9–15 execute a constant number of steps;
 - the recursive calls of **lines** 6 and 7 execute $S(\lceil n/2 \rceil)$ and $S(\lfloor n/2 \rfloor)$ steps;
 - **line** 8 executes $\Theta(n)$ steps (Lemma 17.9).
- Therefore,

$$S(n) = \begin{cases} c_1 & \text{if } n = 1, \\ S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2. \end{cases}$$

Ignoring floors and ceilings (which do not change the asymptotic solution), we obtain the same recurrence as Merge Sort:

$$S(n) = 2S(n/2) + \Theta(n).$$

- Note that the running time does not depend on the values in the array, but only on n : every input of size n is a worst case.

17.4.6.2 Solution by the Master Theorem

- **Identify** $a, b, f(n)$: $a = 2, b = 2, f(n) = \Theta(n)$.
- **Compute** $n^{\log_b a} = n^{\log_2 2} = n$.
- **Compare**: $f(n) = \Theta(n) = \Theta(n^{\log_2 2} \log^0 n)$.
- **Verify the hypothesis**: we are in case 2 of the Master Theorem with $k = 0$.
- **Conclude**:

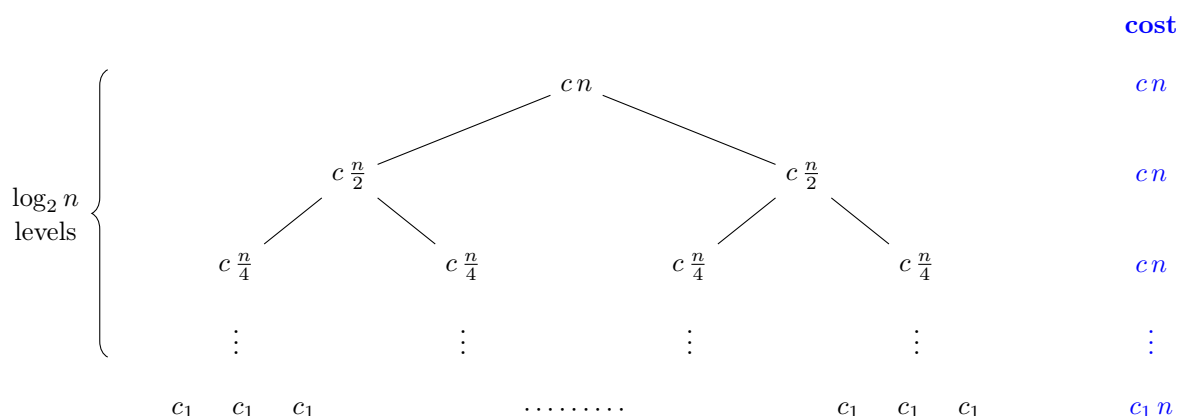
$$S(n) = \Theta(n^{\log_2 2} \log^{0+1} n) = \Theta(n \log n).$$

17.4.6.3 Solution by the recursion tree

- Let $n = 2^k$ be a power of 2, and let cn be the cost of the non-recursive part of a call on $n \geq 2$ elements. In the recursion tree, the level h (for $h = 0, 1, \dots, k-1$) contains 2^h calls on $n/2^h$ elements each, with total cost

$$2^h \cdot c \frac{n}{2^h} = cn.$$

The last level ($h = k = \log_2 n$) contains the n leaves, with total cost $c_1 n$.



- Summing over all the levels:

$$S(n) = \underbrace{cn + cn + \dots + cn}_{\log_2 n \text{ times}} + c_1 n = cn \log_2 n + c_1 n = \Theta(n \log n).$$

- The recursion tree also explains the result: there are $\log_2 n$ levels and each level costs $\Theta(n)$, because the crossing computations of the calls of a level scan, all together, the whole array once.

17.4.6.4 Worst-case running time as a function of the input size

- Under the hypothesis that every entry uses a constant number of bits, $|I| = \Theta(n)$, and also $\log n = \Theta(\log |I|)$ (as shown for the Recursive Binary Search). Hence

$$S(n) = \Theta(n \log n) = \Theta(|I| \log |I|).$$

If all input values use a fixed constant number of bits, then the worst-case running time of the FindMaximumSubarray algorithm is

$$T(|I|) = \Theta(|I| \log |I|).$$

Thus, the divide and conquer algorithm is asymptotically faster than the brute force algorithm, whose worst-case running time is $\Theta(|I|^2)$.

17.4.7 Remark: an iterative linear algorithm

- The divide and conquer algorithm is much better than the brute force algorithm. However, the maximum subarray problem can be solved even faster, with a simple **iterative** algorithm due to J. Kadane, which scans the array only **once**.
- **Idea:** for each position i , let $M(i)$ be the maximum sum of a subarray **ending at** position i . A subarray ending at i is either $a[i]$ alone, or a subarray ending at $i - 1$ extended with $a[i]$. Therefore,

$$M(1) = a[1], \quad M(i) = a[i] + \max\{0, M(i-1)\} \quad (i \geq 2).$$

In words: if the best subarray ending at $i - 1$ has a negative sum, it is better to **restart** from $a[i]$; otherwise, it is better to **extend** it. The answer to the problem is the maximum of $M(1), \dots, M(n)$.

- This is the pseudo-code of the **Kadane** algorithm. The variable *cur* stores $M(i)$ and *cl* the first index of the corresponding subarray.

Algorithm: Kadane

Input : An array $a[1..n]$ of n integer numbers

Output: Indices $bl \leq br$ of a maximum subarray and its sum *best*

```

1 function Kadane( $a, n$ )
2    $best := a[1]; bl := 1; br := 1$ 
3    $cur := a[1]; cl := 1$ 
4    $i := 2$ 
5   while  $i \leq n$  do
6     if  $cur < 0$  then
7        $cur \leftarrow a[i]; cl \leftarrow i$            // restart
8     else
9        $cur \leftarrow cur + a[i]$                  // extend
10    end
11    if  $cur > best$  then
12       $best \leftarrow cur; bl \leftarrow cl; br \leftarrow i$ 
13    end
14     $i \leftarrow i + 1$ 
15  end
16  return ( $bl, br, best$ )
17 end
```

Lemma 17.11: loop invariant of Kadane

At the beginning of each iteration of the **while** loop (and when the loop terminates), with current index i :

1. $cur = \sum_{k=cl}^{i-1} a[k]$ is the maximum sum of a subarray **ending at** position $i - 1$;
2. $best = \sum_{k=bl}^{br} a[k]$ is the maximum sum of a subarray of $a[1..i - 1]$.

Proof. Initialization ($i = 2$). The only subarray ending at 1 is $a[1..1]$, and it is also the only subarray of $a[1..1]$: $cur = best = a[1]$ with $cl = bl = br = 1$.

Maintenance. Assume the invariant holds at the beginning of the iteration with index i . A subarray ending at i is either $a[i..i]$, with sum $a[i]$, or $a[k..i]$ with $k < i$, whose sum is $\sum_{h=k}^{i-1} a[h] + a[i] \leq cur + a[i]$ (by part 1), with equality for $k = cl$. Hence the maximum sum of a subarray ending at i is $\max\{a[i], cur + a[i]\}$, which is $a[i]$ if $cur < 0$ and $cur + a[i]$ otherwise: this is exactly what **lines 6–9** compute. So part 1 holds for $i + 1$.

A subarray of $a[1..i]$ either is a subarray of $a[1..i - 1]$ (best sum: $best$, by part 2) or ends at i (best sum: the new cur). **Lines 11–12** keep the larger of the two, so part 2 holds for $i + 1$.

Termination. The loop terminates with $i = n + 1$: by part 2, $best$ is the maximum sum of a subarray of $a[1..n]$, attained by $a[bl..br]$. $\square$

- **Running time.** The **while** loop executes $n - 1$ iterations, each with a constant number of steps. Hence $S(n) = \Theta(n)$ and, if every entry uses a constant number of bits, $T(|I|) = \Theta(|I|)$. Since any algorithm must read all the entries of the array, no algorithm can be asymptotically faster.

Example 17.13: Kadane algorithm on the stock example

Execution of Kadane on

$$a = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7].$$

Each row reports the values at the end of the iteration with index i (row $i = 1$: initialization).

i	$a[i]$	action	cur	cl	$best$	(bl, br)
1	13	initialization	13	1	13	(1, 1)
2	-3	extend	10	1	13	(1, 1)
3	-25	extend	-15	1	13	(1, 1)
4	20	restart ($cur < 0$)	20	4	20	(4, 4)
5	-3	extend	17	4	20	(4, 4)
6	-16	extend	1	4	20	(4, 4)
7	-23	extend	-22	4	20	(4, 4)
8	18	restart ($cur < 0$)	18	8	20	(4, 4)
9	20	extend	38	8	38	(8, 9)
10	-7	extend	31	8	38	(8, 9)
11	12	extend	43	8	43	(8, 11)
12	-5	extend	38	8	43	(8, 11)
13	-22	extend	16	8	43	(8, 11)
14	15	extend	31	8	43	(8, 11)
15	-4	extend	27	8	43	(8, 11)
16	7	extend	34	8	43	(8, 11)

The algorithm returns (8, 11, 43), the same maximum subarray found by the divide and conquer algorithm, after a single scan of the array.

17.4.8 C++ implementation

- This is a C++ implementation of the divide and conquer algorithm. The array is indexed from 0, and the three values returned by the functions are stored in a **struct**. The value $-\infty$ is replaced by `INT_MIN`, the smallest `int`. To solve the problem on an array of n elements we call `FindMaximumSubarray(a, 0, n-1)`.

```

1  #include <climits>
2
3  struct Subarray
4  {
5      int low;    // first index
6      int high;   // last index
7      int sum;    // sum of a[low..high]
8  };
9
10 Subarray FindMaxCrossingSubarray(int *a, int l, int mid, int r)
11 {
12     int leftSum = INT_MIN, sum = 0, maxLeft = mid;
13     for (int i = mid; i >= l; i--)
14     {
15         sum = sum + a[i];
16         if (sum > leftSum)
17         {
18             leftSum = sum;
19             maxLeft = i;
20         }
21     }
22
23     int rightSum = INT_MIN, maxRight = mid + 1;
24     sum = 0;
25     for (int j = mid + 1; j <= r; j++)
26     {
27         sum = sum + a[j];
28         if (sum > rightSum)
29         {
30             rightSum = sum;
31             maxRight = j;
32         }
33     }
34
35     Subarray result = {maxLeft, maxRight, leftSum + rightSum};
36     return result;
37 }
38
39 Subarray FindMaximumSubarray(int *a, int l, int r)
40 {
41     if (l == r)
42     {
43         Subarray result = {l, r, a[l]}; // Base case: one element
44         return result;
45     }
46
47     int mid = (l + r) / 2;
48
49     Subarray left  = FindMaximumSubarray(a, l, mid);
50     Subarray right = FindMaximumSubarray(a, mid + 1, r);

```

```

51 Subarray cross = FindMaxCrossingSubarray(a, l, mid, r);
52
53 if (left.sum >= right.sum && left.sum >= cross.sum)
54 {
55     return left;
56 }
57 if (right.sum >= left.sum && right.sum >= cross.sum)
58 {
59     return right;
60 }
61 return cross;
62 }

```

Example 17.14: execution of the C++ implementation of FindMaximumSubarray

We execute the first call `FindMaximumSubarray(a, 0, 3)` on the array $a = [-2, 5, -1, 3]$:

$a[j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$
	-2	5	-1	3

- **Call `FindMaximumSubarray(a, 0, 3)`:** $l = 0 \neq r = 3$, $mid = (0 + 3)/2 = 1$. Call on the left half.
 - `FindMaximumSubarray(a, 0, 0)` returns $\{0, 0, -2\}$ (base case)
 - `FindMaximumSubarray(a, 1, 1)` returns $\{1, 1, 5\}$ (base case)
 - `FindMaxCrossingSubarray(a, 0, 0, 1)`: left loop $i = 0$: $sum = -2$, $leftSum = -2$, $maxLeft = 0$; right loop $j = 1$: $sum = 5$, $rightSum = 5$, $maxRight = 1$. Returns $\{0, 1, 3\}$
 - Compare: $left.sum = -2$, $right.sum = 5$, $cross.sum = 3 \Rightarrow$ **return** $\{1, 1, 5\}$
- **Call `FindMaximumSubarray(a, 2, 3)`:** $l = 2 \neq r = 3$, $mid = (2 + 3)/2 = 2$.
 - `FindMaximumSubarray(a, 2, 2)` returns $\{2, 2, -1\}$ (base case)
 - `FindMaximumSubarray(a, 3, 3)` returns $\{3, 3, 3\}$ (base case)
 - `FindMaxCrossingSubarray(a, 2, 2, 3)`: left loop $i = 2$: $sum = -1$, $leftSum = -1$, $maxLeft = 2$; right loop $j = 3$: $sum = 3$, $rightSum = 3$, $maxRight = 3$. Returns $\{2, 3, 2\}$
 - Compare: $left.sum = -1$, $right.sum = 3$, $cross.sum = 2 \Rightarrow$ **return** $\{3, 3, 3\}$
- **Back to the call `FindMaximumSubarray(a, 0, 3)`:** $left = \{1, 1, 5\}$, $right = \{3, 3, 3\}$.
 - `FindMaxCrossingSubarray(a, 0, 1, 3)`: left loop $i = 1$: $sum = 5$, $leftSum = 5$, $maxLeft = 1$; $i = 0$: $sum = 3 \leq 5$, no update; right loop $j = 2$: $sum = -1$, $rightSum = -1$, $maxRight = 2$; $j = 3$: $sum = 2 > -1$, $rightSum = 2$, $maxRight = 3$. Returns $\{1, 3, 5 + 2\} = \{1, 3, 7\}$
 - Compare: $left.sum = 5$, $right.sum = 3$, $cross.sum = 7 \Rightarrow$ **return** $\{1, 3, 7\}$

The algorithm terminates with result $\{1, 3, 7\}$: the maximum subarray is $a[1..3] = [5, -1, 3]$ (indices 2..4 in the pseudo-code, where the array is indexed from 1), with sum 7. In total, $2n - 1 = 7$ calls of `FindMaximumSubarray` are executed.

Key points on the maximum subarray problem.

- brute force algorithm: $\Theta(n^2)$;
- divide and conquer algorithm (recursive): $\Theta(n \log n)$, with the same recurrence as Merge Sort;
- Kadane algorithm (iterative): $\Theta(n)$.

A recursive divide and conquer algorithm is not necessarily the best possible one: here, a simple iterative algorithm that exploits the structure of the problem ($M(i)$ depends only on $M(i - 1)$) is asymptotically faster. We will come back to this lesson in the chapter «Recursive versus iterative algorithms».

17.5 Summary

The four problems of the chapter cover the three typical shapes of a recursive algorithm on arrays: two calls on the halves with constant combination work (sum), two calls on the halves with linear combination work (Merge Sort, maximum subarray), one call on a half (binary search).

recursive algorithm	recurrence	number of steps	extra memory
<code>RecursiveSum</code>	$S(n) = 2S(n/2) + c$	$7n - 5 = \Theta(n)$	$\Theta(\log_2 n)$
<code>RecursiveSumLinear</code>	$S(n) = S(n - 1) + 3$	$3n - 1 = \Theta(n)$	$\Theta(n)$
<code>MergeSort</code>	$S(n) = 2S(n/2) + \Theta(n)$	$\Theta(n \log_2 n)$	$\Theta(n)$
<code>RecursiveBinarySearch</code>	$S(n) = S(\lfloor n/2 \rfloor) + 5$	$5(\lfloor \log_2 n \rfloor + 1) + 2 = \Theta(\log_2 n)$	$\Theta(\log_2 n)$
<code>FindMaximumSubarray</code>	$S(n) = 2S(n/2) + \Theta(n)$	$\Theta(n \log_2 n)$	$\Theta(\log_2 n)$

- Correctness: always by strong induction on the size of the sub-problem; the non-recursive parts (`Merge`, `FindMaxCrossingSubarray`, `Kadane`) by loop invariants.
- Recursion gives an asymptotic gain only when the work of the combine step, summed over a level, is small compared with what an iterative algorithm does (Merge Sort against the $\Theta(n^2)$ sorting algorithms). For the sum and the search it gives no gain in time and costs memory; for the maximum subarray the best algorithm is iterative.

Recursive versus iterative algorithms

Introduction

In the previous chapters we have studied several recursive algorithms (chapter «Recursion»: factorial, Fibonacci numbers, recursive GCD, fast power, Tower of Hanoi; chapter «Recursive algorithms»: sum of an array, Merge Sort, Recursive Binary Search, maximum subarray) and we have analysed them by means of recurrences and of the Master Theorem. In this chapter we answer a very practical question:

when is it convenient to design a recursive algorithm, and when is it not?

We will see that the answer does not depend on recursion itself, but on **how** the problem is decomposed into subproblems:

- if the problem is split into a few **independent** subproblems of size n/b (divide and conquer), recursion may give a large asymptotic gain;
- if the problem is only reduced from size n to size $n - 1$, recursion gives no gain in time and it costs memory;
- if the subproblems **overlap**, a naive recursive algorithm may be exponentially slower than an iterative one.

The chapter first explains the costs that every recursive call carries, then compares recursive and iterative algorithms for the problems of the course, studies the three situations above one by one, and closes with measured running times of the C++ implementations and with some guidelines.

18.1 Recursion and iteration

Every recursive algorithm can be transformed into an iterative one (that uses an **explicit stack** to store the pending calls), and every iterative algorithm can be written recursively (every loop can be replaced by a function that calls itself).

Recursion and iteration have the **same expressive power**: what changes from one to the other is the **number of steps** and the **amount of memory** needed to solve the problem.

18.1.1 The hidden costs of recursion

- Each time a function is called, the computer creates a **stack frame** (or *activation record*) on the **call stack**, a memory area reserved to the program. The stack frame contains:
 - the parameters of the call;
 - the local variables of the call;
 - the return address, i.e., the point of the calling function from which the execution resumes when the call terminates.
- The frame is removed from the stack only when the call terminates. Accordingly, during the execution of a recursive algorithm, all the calls that have started and not yet terminated keep their frame on the stack.
- This produces three hidden costs, which do not appear in a loop:
 1. **Call overhead**: creating and removing a frame, copying the parameters and jumping to the function and back require a constant number of extra steps for each call. This does not change the order of growth of the running time, but it makes a recursive algorithm slower than an equivalent loop by a constant factor.
 2. **Memory**: the memory used by the call stack is proportional to the maximum number of calls that are *simultaneously* active, i.e., to the **depth** of the recursion (the height of the recursion tree plus one, studied in detail later in this chapter).
 3. **Stack overflow**: the call stack has a fixed and rather small size. On Linux the default size is typically 8 MB (command `ulimit -s` $\Rightarrow$ 8192 KB), on Windows it is typically 1 MB. If the depth of the recursion is too large, the stack is exhausted and the program **crashes** (in C++ with a *segmentation fault*). In Python the default limit is only 1000 nested calls.

Example 18.1: a recursive linear sum that crashes

- Consider the decrease-by-one recursive algorithm that computes the sum of an array $a[1..n]$ as $\text{SumDecrease}(a, n) = \text{SumDecrease}(a, n - 1) + a[n]$, with base case $\text{SumDecrease}(a, 0) = 0$ (the decrease-by-one sum of the chapter «Recursive algorithms», with base case $n = 0$ instead of $n = 1$).
- The calls $\text{SumDecrease}(a, n), \text{SumDecrease}(a, n - 1), \dots, \text{SumDecrease}(a, 0)$ are all active at the same time: the depth of the recursion is $n + 1$.

- With $n = 10^7$ and a stack frame of about 32 bytes, the stack should contain about

$$10^7 \cdot 32 = 3.2 \cdot 10^8 \text{ bytes} \approx 320 \text{ MB},$$

i.e., 40 times the default 8 MB.

- The program crashes, although the array itself ($4 \cdot 10^7$ bytes = 40 MB) fits easily in memory and the iterative sum computes the result in about 3 milliseconds. In the experiment at the end of the chapter we measure that, with a stack of 8 MB, the crash occurs already for $n \approx 7 \cdot 10^5$.

18.1.2 From recursion to iteration with an explicit stack

- A recursive algorithm can always be transformed into an iterative one by managing **explicitly** a stack that contains the parameters of the calls still to be executed.
- As an example, this is an iterative version of the halving recursive sum `RecursiveSum` of the chapter «Recursive algorithms», which computes the sum of $a[l..r]$ as the sum of $a[l..mid]$ plus the sum of $a[mid + 1..r]$:

Algorithm: SumExplicitStack

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The sum $a[1] + a[2] + \dots + a[n]$

```

1 function SumExplicitStack( $a, n$ )
2    $s := 0$ 
3    $P := \text{empty stack}$                                 // pending sub-arrays
4   push  $(1, n)$  on  $P$ 
5   while  $P$  is not empty do
6      $(l, r) := \text{pop from } P$ 
7     if  $l = r$  then
8        $s \leftarrow s + a[l]$                                 // base case
9     else
10       $mid := \lfloor (l + r) / 2 \rfloor$ 
11      push  $(mid + 1, r)$  on  $P$ 
12      push  $(l, mid)$  on  $P$ 
13    end
14  end
15  return  $s$ 
16 end
```

- The stack P plays exactly the role of the call stack: it contains the sub-arrays still to be summed. Its maximum size is of the same order as the depth of the recursion, i.e., $\Theta(\log_2 n)$.

Example 18.2: SumExplicitStack on four elements

We execute `SumExplicitStack(a, 4)` with $a = [4, 7, 1, 9]$. Each item reports the pair popped from P , what is done, and the content of P afterwards (the top of the stack is on the right):

- pop (1, 4): $mid = 2$, push (3, 4) and (1, 2) $\Rightarrow P = [(3, 4), (1, 2)]$, $s = 0$;
- pop (1, 2): $mid = 1$, push (2, 2) and (1, 1) $\Rightarrow P = [(3, 4), (2, 2), (1, 1)]$, $s = 0$;
- pop (1, 1): base case, $s = 4 \Rightarrow P = [(3, 4), (2, 2)]$;
- pop (2, 2): base case, $s = 11 \Rightarrow P = [(3, 4)]$;
- pop (3, 4): $mid = 3$, push (4, 4) and (3, 3) $\Rightarrow P = [(4, 4), (3, 3)]$, $s = 11$;
- pop (3, 3): base case, $s = 12 \Rightarrow P = [(4, 4)]$;
- pop (4, 4): base case, $s = 21 \Rightarrow P$ is empty.

The algorithm returns $21 = 4 + 7 + 1 + 9$. The sub-arrays are processed in the same order as the calls of `RecursiveSum(a, 1, 4)`, and P never contains more than 3 pairs.

Simulating the recursion with an explicit stack does **not** change the order of growth of the number of steps and of the memory: it only removes the call overhead and moves the stack from the (small) call stack to the (large) main memory.

A real gain in time or in memory can only come from a **different algorithm**, not from the mechanical replacement of recursion by iteration (or vice versa).

18.2 Comparison at a glance

- The following table compares, for the problems studied in the course, the best iterative algorithm of the course with the recursive one.
- **Time** is the worst-case number of steps $S(n)$ (arithmetic operations on numbers are counted as one step, as in the chapter «Running time»); **space** is the memory used **in addition to the input**, including the call stack.
- The verdict refers to the recursive algorithm: **GAIN** if it is asymptotically faster, **SAME** if it has the same running time, **LOSS** if it is worse in time or in space.

Problem	Iterative	time	space	Recursive	time	space	Verdict
Sorting	Insertion, Selection, Exchange Sort	$\Theta(n^2)$	$\Theta(1)$	Merge Sort	$\Theta(n \log_2 n)$	$\Theta(n)$	GAIN in time (divide and conquer), paid with $\Theta(n)$ memory
Power x^n	n multiplications	$\Theta(n)$	$\Theta(1)$	Fast power	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	GAIN , which comes from halving n : the iterative square-and-multiply is also $\Theta(\log_2 n)$, with $\Theta(1)$ space
Maximum subarray	Brute force	$\Theta(n^2)$	$\Theta(1)$	Divide and conquer	$\Theta(n \log_2 n)$	$\Theta(\log_2 n)$	GAIN over brute force, but the iterative Kadane algorithm, $\Theta(n)$ time and $\Theta(1)$ space, beats both
Product of n -digit integers	Schoolbook	$\Theta(n^2)$	$\Theta(n)$	Karatsuba	$\Theta(n^{\log_2 3})$	$\Theta(n)$	GAIN : $S(n) = 3S(n/2) + \Theta(n)$, $n^{\log_2 3} \approx n^{1.585}$
Array sum	Loop	$\Theta(n)$	$\Theta(1)$	Halving Decr. by one	$\Theta(n)$ $\Theta(n)$	$\Theta(\log_2 n)$ $\Theta(n)$	SAME time, LOSS in space
Binary search	Loop	$\Theta(\log_2 n)$	$\Theta(1)$	Recursive Binary Search	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	SAME time, LOSS in space; tail recursion, it can be removed
Factorial, linear search	Loop	$\Theta(n)$	$\Theta(1)$	Decrease by one	$\Theta(n)$	$\Theta(n)$	SAME time, LOSS in space, risk of stack overflow
GCD (Euclid)	Loop	$\Theta(\log m)$	$\Theta(1)$	Recursive GCD	$\Theta(\log m)$	$\Theta(\log m)$	SAME : tail recursion, the depth is only logarithmic
Fibonacci numbers	Loop with two variables	$\Theta(n)$	$\Theta(1)$	Naive recursion	$\Theta(\varphi^n)$	$\Theta(n)$	huge LOSS : overlapping subproblems; memoization brings it back to $\Theta(n)$
Tower of Hanoi	Iterative versions exist	$\Theta(2^n)$	$\Theta(n)$	Recursive Hanoi	$\Theta(2^n)$	$\Theta(n)$	SAME : $2^n - 1$ moves are needed anyway; recursion is the natural description

- In the table, $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio and $m = \min(a, b)$ for the GCD of a and b .
- The four rows of the first block are the cases where recursion **pays off**: in all of them the problem is split into subproblems of size $n/2$ (next section).
- The rows of the second block are the cases where recursion only **replaces a loop**: no gain in time, a loss in memory (section «Space»).
- The Fibonacci row is the case where naive recursion is a **disaster**, because the same subproblems are solved over and over again (section «When subproblems overlap»).

18.3 Divide and conquer is what gives the gain, not recursion itself

- The number of steps of a recursive algorithm is described by a recurrence. Two families of recurrences appear in the course:

- **decrease by one**: the problem of size n is reduced to **one** problem of size $n - 1$,

$$S(n) = S(n - 1) + f(n);$$

- **divide and conquer**: the problem of size n is split into a problems of size n/b ,

$$S(n) = a S(n/b) + f(n), \quad a \geq 1, b > 1.$$

18.3.1 Decrease by one: recursion is a loop in disguise

Observation 18.1: decrease-by-one recurrences

Let $S(n) = S(n - 1) + f(n)$ for $n \geq 1$, with $S(0) = d$. Then

$$S(n) = d + \sum_{i=1}^n f(i).$$

In particular, if $f(n) = c$ for a constant $c > 0$, then $S(n) = cn + d = \Theta(n)$.

Proof. By induction on n . For $n = 0$ the sum is empty and $S(0) = d$. If the formula holds for $n - 1$, then

$$S(n) = S(n - 1) + f(n) = d + \sum_{i=1}^{n-1} f(i) + f(n) = d + \sum_{i=1}^n f(i).$$

If $f(i) = c$ for every i , then $\sum_{i=1}^n c = cn$ and $S(n) = cn + d$. Since $\lim_{n \rightarrow \infty} \frac{cn+d}{n} = c \in \mathbb{R}_{>0}$, we have $S(n) = \Theta(n)$. $\square$

- The formula $S(n) = d + \sum_{i=1}^n f(i)$ is **exactly** the number of steps of a loop that executes, for $i = 1, 2, \dots, n$, a body of cost $f(i)$. The decrease-by-one recursion does the same work as a loop:
 - factorial, linear search and the decrease-by-one array sum: $S(n) = S(n - 1) + \Theta(1) = \Theta(n)$, like the corresponding loop;
 - a recursive insertion sort (sort $a[1..n - 1]$ recursively, then insert $a[n]$): $S(n) = S(n - 1) + \Theta(n) = \Theta(n^2)$, like the iterative **InsertionSort**.
- In all these cases the recursive algorithm has the same order of growth of the running time as the iterative one, plus the call overhead and a depth of recursion $\Theta(n)$.

18.3.2 Divide and conquer: where the gain comes from

- Merge Sort splits the array into two halves, sorts them recursively and merges them in $\Theta(n)$ steps:

$$S(n) = 2S(n/2) + \Theta(n) = \Theta(n \log_2 n)$$

by case 2 of the Master Theorem. The gain with respect to $\Theta(n^2)$ comes from the fact that the two halves are **balanced** and **independent**: the recursion tree has only $\Theta(\log_2 n)$ levels, and each level costs $\Theta(n)$.

- The same splitting can be organised **without recursion**: we first merge pairs of sub-arrays of length 1, then pairs of sub-arrays of length 2, then of length 4, and so on. This is the **bottom-up** Merge Sort. It uses the procedure `Merge(a, l, m, r)` of the chapter «Recursive algorithms», which merges the sorted sub-arrays $a[l..m]$ and $a[m + 1..r]$ in $\Theta(r - l + 1)$ steps.

Algorithm: MergeSortBottomUp

Input : An array $a[1..n]$ of n natural numbers

Output: The array a sorted in non-decreasing order

```

1 function MergeSortBottomUp(a, n)
2   width := 1
3   while width < n do
4     l := 1
5     while l + width ≤ n do
6       m := l + width - 1
7       r := min(l + 2 · width - 1, n)
8       Merge(a, l, m, r)
9       l ← l + 2 · width
10    end
11    width ← 2 · width
12  end
13 end

```

Example 18.3: bottom-up Merge Sort

Consider $a = [5, 2, 4, 6, 1, 3]$ ($n = 6$).

- $width = 1$: merge $a[1..1]$ with $a[2..2]$, $a[3..3]$ with $a[4..4]$, $a[5..5]$ with $a[6..6] \Rightarrow a = [2, 5, 4, 6, 1, 3]$.
- $width = 2$: merge $a[1..2]$ with $a[3..4]$; for $l = 5$ we have $l + width = 7 > 6$, so $a[5..6]$ is left as it is $\Rightarrow a = [2, 4, 5, 6, 1, 3]$.
- $width = 4$: merge $a[1..4]$ with $a[5..6] \Rightarrow a = [1, 2, 3, 4, 5, 6]$.
- $width = 8 \geq 6$: stop.

The algorithm executes $3 = \lceil \log_2 6 \rceil$ passes.

Observation 18.2: bottom-up Merge Sort

For $n \geq 2$, `MergeSortBottomUp` executes $\lceil \log_2 n \rceil$ passes of the outer loop and its number of steps in the worst case is

$$S(n) = \Theta(n \log_2 n),$$

the same as the recursive Merge Sort. Moreover, it does not use the call stack.

Proof. Let $n \geq 2$. In the outer loop *width* takes the values $1, 2, 4, \dots, 2^{p-1}$, where p is the smallest integer with $2^p \geq n$, i.e., $p = \lceil \log_2 n \rceil$.

Upper bound. In a pass, the calls to `Merge` (line 8) work on disjoint sub-arrays, so their total length is at most n and their total cost is $O(n)$. The inner loop is executed at most $\frac{n}{2 \cdot \text{width}} + 1 \leq n$ times, each time with a constant number of other steps. Hence every pass costs $O(n)$ and $S(n) = O(n \lceil \log_2 n \rceil) = O(n \log_2 n)$.

Lower bound. The inner loop stops when $l + \text{width} > n$, so the elements that are not merged in a pass are at most *width*. In the passes with $\text{width} \leq n/2$, at least $n - \text{width} \geq n/2$ elements are merged, with $\Omega(n)$ steps. These passes are those with $\text{width} = 2^t$, $t = 0, 1, \dots, \lceil \log_2 n \rceil - 1$, i.e., $\lceil \log_2 n \rceil$ passes. Hence $S(n) = \Omega(n \lceil \log_2 n \rceil) = \Omega(n \log_2 n)$.

Therefore $S(n) = \Theta(n \log_2 n)$. The algorithm contains no recursive call, so the only extra memory is the one used by `Merge`, i.e., $\Theta(n)$. $\square$

The gain of Merge Sort over the $\Theta(n^2)$ sorting algorithms is a consequence of the **divide and conquer** strategy (balanced halves, merged in linear time), not of recursion: the iterative bottom-up version has the same running time $\Theta(n \log_2 n)$.

18.3.3 Fast power: the gain comes from halving

- The recursive fast power of the chapter «Recursion» computes $x^n = (x^{\lfloor n/2 \rfloor})^2$ if n is even, and $x^n = (x^{\lfloor n/2 \rfloor})^2 \cdot x$ if n is odd, with $S(n) = S(\lfloor n/2 \rfloor) + \Theta(1) = \Theta(\log_2 n)$.
- Halving the exponent can also be done with a loop, by reading the bits of n from the least significant one (**square-and-multiply**):

Algorithm: PowerIt**Input** : A number x and a natural number n **Output**: The power x^n

```

1 function PowerIt( $x, n$ )
2    $p := 1$ 
3    $y := x$ 
4    $k := n$ 
5   while  $k > 0$  do
6     if  $k$  is odd then
7        $p \leftarrow p \cdot y$ 
8     end
9      $y \leftarrow y \cdot y$ 
10     $k \leftarrow \lfloor k/2 \rfloor$ 
11  end
12  return  $p$ 
13 end

```

- At the beginning of each iteration we have $p \cdot y^k = x^n$ (it holds at the start with $p = 1$, $y = x$, $k = n$, and it is preserved by every iteration); when $k = 0$ we get $p = x^n$.

Example 18.4: square-and-multiply with $n = 13$

We execute **PowerIt**($x, 13$), where $13 = (1101)_2$. Each item reports the values at the end of an iteration:

- $k = 13$ is odd: $p = x$, $y = x^2$, $k = 6$;
- $k = 6$ is even: $p = x$, $y = x^4$, $k = 3$;
- $k = 3$ is odd: $p = x \cdot x^4 = x^5$, $y = x^8$, $k = 1$;
- $k = 1$ is odd: $p = x^5 \cdot x^8 = x^{13}$, $y = x^{16}$, $k = 0$.

The iterations multiply p by x , x^4 and x^8 (the bits of 13 equal to 1), and $p = x^{1+4+8} = x^{13}$ after $4 = \lfloor \log_2 13 \rfloor + 1$ iterations.

Observation 18.3: iterations of PowerIt

For $n \geq 1$, **PowerIt** executes exactly $\lfloor \log_2 n \rfloor + 1$ iterations, hence $S(n) = \Theta(\log_2 n)$, and it uses $\Theta(1)$ extra memory.

Proof. Every iteration replaces k by $\lfloor k/2 \rfloor$, i.e., it removes the least significant bit of the binary representation of k . Hence the number of iterations is the number of binary digits of n , which is $\lfloor \log_2 n \rfloor + 1$ (chapter «Representation of numbers»). Each iteration executes a constant number of steps, so $S(n) = \Theta(\log_2 n)$. Only the three variables p , y , k are used. $\square$

18.3.4 Maximum subarray: a better iterative algorithm

- For the maximum subarray problem (chapter «Recursive algorithms») the divide and conquer algorithm, with $S(n) = 2S(n/2) + \Theta(n) = \Theta(n \log_2 n)$, improves on the $\Theta(n^2)$ brute force algorithm.
- However, a smarter **iterative** algorithm, due to Kadane, solves the problem in $\Theta(n)$ steps. For each position i it maintains the maximum sum cur of a sub-array **ending** at position i : such a sub-array either is $a[i]$ alone, or extends the best sub-array ending at $i - 1$. This is a compact version of the algorithm **Kadane** of the chapter «Recursive algorithms», which returns only the maximum sum:

Algorithm: Kadane

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The maximum sum of a non-empty sub-array $a[l..r]$

```

1 function Kadane(a, n)
2   best := a[1]
3   cur := a[1]                                // best sum of a sub-array ending at i
4   i := 2
5   while i ≤ n do
6     cur ← max(a[i], cur + a[i])
7     best ← max(best, cur)
8     i ← i + 1
9   end
10  return best
11 end

```

- The loop is executed $n - 1$ times with a constant number of steps: $S(n) = \Theta(n)$, with $\Theta(1)$ extra memory.
- Divide and conquer is a powerful **design technique**, but it is not always the best one: a gain with respect to brute force does not mean that the recursive algorithm is optimal.

18.3.5 Integer multiplication: Karatsuba

- Two n -digit integers are multiplied with the schoolbook method in $\Theta(n^2)$ steps on the digits.
- Splitting each number into two halves of $n/2$ digits, the product requires four products of $n/2$ -digit numbers. Karatsuba observed that **three** products are enough (plus additions and shifts, costing $\Theta(n)$):

$$S(n) = 3S(n/2) + \Theta(n).$$

- Here $a = 3$, $b = 2$, $f(n) = \Theta(n)$ and $n^{\log_2 3} \approx n^{1.585}$; since $f(n) = O(n^{\log_2 3 - \varepsilon})$ for $\varepsilon = 0.5$, case 1 of the Master Theorem gives $S(n) = \Theta(n^{\log_2 3})$, asymptotically better than $\Theta(n^2)$.

Key points. A decrease-by-one recursion does the same work as a loop. A divide and conquer recursion can be asymptotically faster than the simple iterative algorithms (Merge Sort, fast power, Karatsuba), but the gain comes from the decomposition into balanced independent subproblems, and the same decomposition can often be organised with a loop (bottom-up Merge Sort, square-and-multiply). The best algorithm for a problem may even be iterative (Kadane).

18.4 When subproblems overlap

- In divide and conquer the subproblems are **disjoint**: the two halves of Merge Sort have no element in common. When the subproblems **overlap**, a naive recursive algorithm solves the same subproblem many times.
- The Fibonacci numbers are defined by $F(0) = 0$, $F(1) = 1$ and $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$. The direct translation of the definition is:

Algorithm: FibRec

Input : A natural number n

Output: The Fibonacci number $F(n)$

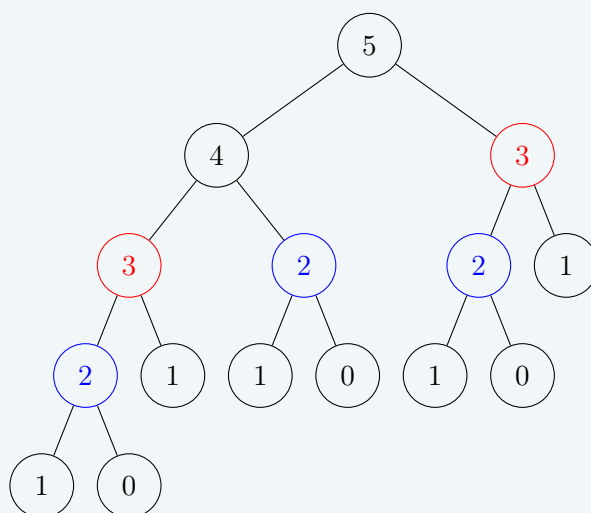
```

1 function FibRec( $n$ )
2   if  $n \leq 1$  then
3     return  $n$ 
4   end
5   return FibRec( $n - 1$ ) + FibRec( $n - 2$ )
6 end

```

Example 18.5: the recursion tree of FibRec(5)

This is the recursion tree of FibRec(5) (each node is a call, labelled with its parameter). The sub-tree of FibRec(3) is computed **twice**, the call FibRec(2) **three times**:



The tree has 15 nodes, while only 6 different values $F(0), F(1), \dots, F(5)$ are needed.

Observation 18.4: number of calls of FibRec

For every $n \geq 0$, the number of calls executed by `FibRec`(n) is

$$C(n) = 2F(n+1) - 1.$$

Proof. The calls satisfy $C(0) = C(1) = 1$ and $C(n) = 1 + C(n-1) + C(n-2)$ for $n \geq 2$. We prove the formula by strong induction. Base cases: $C(0) = 1 = 2F(1) - 1$ and $C(1) = 1 = 2F(2) - 1$. If the formula holds for $n-1$ and $n-2$, then

$$C(n) = 1 + (2F(n) - 1) + (2F(n-1) - 1) = 2(F(n) + F(n-1)) - 1 = 2F(n+1) - 1. \quad \square$$

Observation 18.5: growth of the Fibonacci numbers

Let $\varphi = \frac{1+\sqrt{5}}{2}$. For every $n \geq 1$,

$$\varphi^{n-2} \leq F(n) \leq \varphi^{n-1}.$$

Hence $F(n) = \Theta(\varphi^n)$ and the number of steps of `FibRec` is $S(n) = \Theta(C(n)) = \Theta(\varphi^n)$.

Proof. The number φ satisfies $\varphi^2 = \varphi + 1$. Base cases: $\varphi^{-1} \approx 0.618 \leq F(1) = 1 \leq \varphi^0 = 1$ and $\varphi^0 = 1 \leq F(2) = 1 \leq \varphi \approx 1.618$. For $n \geq 3$, by strong induction,

$$F(n) = F(n-1) + F(n-2) \geq \varphi^{n-3} + \varphi^{n-4} = \varphi^{n-4}(\varphi + 1) = \varphi^{n-4}\varphi^2 = \varphi^{n-2},$$

and in the same way $F(n) \leq \varphi^{n-2} + \varphi^{n-3} = \varphi^{n-3}(\varphi + 1) = \varphi^{n-1}$. Accordingly, $\varphi^{-2}\varphi^n \leq F(n) \leq \varphi^{-1}\varphi^n$, i.e., $F(n) = \Theta(\varphi^n)$. Each call executes a constant number of steps (besides its sub-calls), so $S(n) = \Theta(C(n)) = \Theta(2F(n+1) - 1) = \Theta(\varphi^n)$. $\square$

- For example, `FibRec`(40) executes $2F(41) - 1 = 331\,160\,281$ calls to compute $F(40) = 102\,334\,155$, and `FibRec`(50) executes $2F(51) - 1 = 40\,730\,022\,147$ calls.

18.4.1 Memoization

- The waste of `FibRec` is eliminated by storing in a table `memo[0..n]` the values already computed (**memoization**): before computing $F(k)$ we check whether it is already in the table.

Algorithm: FibMemo

Input : A natural number n and a table $memo[0..n]$ whose entries are all *undefined*

Output: The Fibonacci number $F(n)$

```

1 function FibMemo( $n$ ,  $memo$ )
2   if  $n \leq 1$  then
3     return  $n$ 
4   end
5   if  $memo[n]$  is undefined then
6      $memo[n] \leftarrow \text{FibMemo}(n-1, memo) + \text{FibMemo}(n-2, memo)$ 
7   end
8   return  $memo[n]$ 
9 end

```

Observation 18.6: number of calls of FibMemo

For every $n \geq 2$, $\text{FibMemo}(n, memo)$ (with all entries of $memo$ undefined) executes exactly $2n - 1$ calls. Hence $S(n) = \Theta(n)$; the memory used is $\Theta(n)$ (the table and the call stack).

Proof. Let $M(k)$ be the number of calls executed by $\text{FibMemo}(k, memo)$ when the entries $memo[0..k]$ are undefined. For $k = 2$ the call makes two calls, both base cases: $M(2) = 3$. For $k \geq 3$ the call makes the call $\text{FibMemo}(k-1, memo)$, which executes $M(k-1)$ calls and fills $memo[2..k-1]$; then it makes the call $\text{FibMemo}(k-2, memo)$, which finds $memo[k-2]$ already defined (or is a base case if $k-2 \leq 1$) and returns immediately. Hence $M(k) = 1 + M(k-1) + 1 = M(k-1) + 2$, and by Observation 18.1 $M(n) = M(2) + 2(n-2) = 2n - 1$. Each call executes a constant number of steps, so $S(n) = \Theta(n)$. The table has $n + 1$ entries and the depth of the recursion is n . $\square$

18.4.2 The iterative algorithm

- Computing the values $F(2), F(3), \dots, F(n)$ in increasing order, each value only needs the two previous ones. We do not even need the table: two variables are enough.

Algorithm: FibIt**Input** : A natural number n **Output**: The Fibonacci number $F(n)$

```

1 function FibIt( $n$ )
2   if  $n \leq 1$  then
3     return  $n$ 
4   end
5    $prev := 0$                                 //  $F(i - 2)$ 
6    $curr := 1$                                 //  $F(i - 1)$ 
7    $i := 2$ 
8   while  $i \leq n$  do
9      $next := prev + curr$ 
10     $prev \leftarrow curr$ 
11     $curr \leftarrow next$ 
12     $i \leftarrow i + 1$ 
13  end
14  return  $curr$ 
15 end

```

- The loop is executed $n - 1$ times: $S(n) = \Theta(n)$, with $\Theta(1)$ extra memory.

Algorithm	time	space
FibRec (naive recursion)	$\Theta(\varphi^n)$	$\Theta(n)$
FibMemo (recursion + memoization)	$\Theta(n)$	$\Theta(n)$
FibIt (iteration, bottom-up)	$\Theta(n)$	$\Theta(1)$

When the subproblems overlap, the right technique is to solve each subproblem **once** and store its value: top-down with memoization, or bottom-up with a loop. This is the idea of **Dynamic Programming**, which will be studied later in the course.

18.5 Space

The previous sections compared numbers of steps. We now turn to the second resource, the memory, which for a recursive algorithm is governed by the depth of the recursion.

18.5.1 Depth of the recursion

Definition 18.7: depth of the recursion

The **depth of the recursion** $D(n)$ of a recursive algorithm on an input of size parameter n is the maximum number of calls that are active at the same time during the execution. It is the number of nodes of the longest path from the root to a leaf of the recursion tree, i.e., the height of the tree plus one.

- When the calls are executed one after the other, only the calls on the path from the root to the current call are active. Hence, if every frame has constant size, the memory used by the call stack is $\Theta(D(n))$.
- The depth is **not** the number of calls: **FibRec**(n) executes $\Theta(\varphi^n)$ calls, but at most n of them are active at the same time.

Algorithm	base case	depth $D(n)$	$D(10^6)$
Factorial, linear sum (decrease by one)	$n = 0$	$n + 1$	1 000 001
Tower of Hanoi with n disks	$n = 0$	$n + 1$	1 000 001
FibRec (n), $n \geq 1$	$n \leq 1$	n	1 000 000
Sum by halving, Merge Sort, Maximum Subarray	$l = r$	$\lceil \log_2 n \rceil + 1$	21
Recursive Binary Search (worst case)	$l > r$	$\lfloor \log_2 n \rfloor + 2$	21
Fast power x^n , $n \geq 1$	$n = 0$	$\lfloor \log_2 n \rfloor + 2$	21
Recursive GCD(a, b)	$b = 0$	$\Theta(\log \min(a, b))$	–

- For the halving algorithms, the largest sub-array at level t of the recursion tree has $\lceil n/2^t \rceil$ elements, and it reaches size 1 for $t = \lceil \log_2 n \rceil$; for the fast power, the exponent goes through the values $n, \lfloor n/2 \rfloor, \dots, 1, 0$, which are $\lfloor \log_2 n \rfloor + 2$.
- A depth $\Theta(\log_2 n)$ is never a problem in practice (21 frames for $n = 10^6$, 31 for $n = 10^9$). A depth $\Theta(n)$ is a real risk of stack overflow for large n .

18.5.2 Tail recursion and its elimination

Definition 18.8: tail recursion

A recursive call is a **tail call** if it is the last operation executed by the function, and its result is returned as it is, without further computation. A recursive algorithm is **tail recursive** if all its recursive calls are tail calls.

- The recursive GCD (return $\text{GCD}(b, a \bmod b)$) and the Recursive Binary Search (return the result of the search in one half) are tail recursive.
- The factorial ($n \cdot \text{Fact}(n-1)$), the decrease-by-one sum ($\text{SumDecrease}(a, n-1) + a[n]$) and **FibRec** are **not** tail recursive: after the recursive call returns there is still a multiplication or an addition to do, so the frame must be kept.
- In a tail call, the frame of the calling function is no longer needed: nothing remains to be done after the call. Therefore the call can be replaced by an **update of the parameters** followed by a jump to the beginning of the function, i.e., by a loop.

Mechanical elimination of tail recursion

1. Enclose the body of the function in a **while** loop whose condition is the negation of the base-case condition.
2. Replace every tail call $f(e_1, \dots, e_k)$ by the **simultaneous** assignment of the new values $e_1, \dots, e_k$ to the parameters (use temporary variables when a new value depends on a parameter that is modified).
3. Move the return of the base case after the loop.

Example 18.6: Recursive GCD $\Rightarrow$ iterative GCD**Algorithm: GCDRec**

```

1 function GCDRec(a, b)
2   if b = 0 then
3     return a
4   end
5   return GCDRec(b, a mod b)
6 end

```

Algorithm: GCDIt

```

1 function GCDIt(a, b)
2   while b  $\neq$  0 do
3     r := a mod b
4     a  $\leftarrow$  b
5     b  $\leftarrow$  r
6   end
7   return a
8 end

```

- Base condition $b = 0 \Rightarrow$ loop condition $b \neq 0$.
- Tail call $\text{GCDRec}(b, a \bmod b) \Rightarrow (a, b) \leftarrow (b, a \bmod b)$; the temporary variable r is needed because $a \bmod b$ must be computed with the *old* value of a .
- With $(a, b) = (252, 105)$ both algorithms go through the pairs $(252, 105)$, $(105, 42)$, $(42, 21)$, $(21, 0)$ and return 21: four calls versus three iterations.

Example 18.7: Recursive Binary Search $\Rightarrow$ iterative Binary Search

Algorithm: RecBinarySearch

```

1 function RecBS( $a, l, r, m$ )
2   if  $l > r$  then
3     return -1
4   end
5    $mid := l + \lfloor (r - l)/2 \rfloor$ 
6   if  $a[mid] = m$  then
7     return  $mid$ 
8   end
9   if  $a[mid] < m$  then
10    return RecBS( $a, mid + 1, r, m$ )
11  else
12    return RecBS( $a, l, mid - 1, m$ )
13  end
14 end

```

Algorithm: ItBinarySearch

```

1 function ItBS( $a, l, r, m$ )
2   while  $l \leq r$  do
3      $mid := l + \lfloor (r - l)/2 \rfloor$ 
4     if  $a[mid] = m$  then
5       return  $mid$ 
6     end
7     if  $a[mid] < m$  then
8        $l \leftarrow mid + 1$ 
9     else
10       $r \leftarrow mid - 1$ 
11    end
12  end
13  return -1
14 end

```

- Base condition $l > r \Rightarrow$ loop condition $l \leq r$.
- The second base case $(a[mid] = m)$ stays inside the loop as a **return**.
- Tail call $\text{RecBS}(a, mid + 1, r, m) \Rightarrow l \leftarrow mid + 1$; tail call $\text{RecBS}(a, l, mid - 1, m) \Rightarrow$

$r \leftarrow \text{mid} - 1$ (the parameters that do not change are not assigned).

- Calling `ItBS(a, 1, n, m)` we obtain exactly the iterative `BinarySearch` of the chapter «Linear and binary search».

Observation 18.9: elimination of tail recursion

Let f be a tail recursive algorithm whose calls execute a constant number of steps (besides the recursive call) and use frames of constant size, and let g be the iterative algorithm obtained by the mechanical elimination of the tail recursion. Then g returns the same result as f , its number of steps is of the same order as that of f , and it uses $\Theta(1)$ extra memory instead of $\Theta(D(n))$.

Proof. Let $p_1, p_2, \dots, p_k$ be the values of the parameters in the successive calls of f , where p_k satisfies a base condition. The j -th iteration of the loop of g starts with the parameters equal to p_j and executes the same operations as the call of f with parameters p_j , except that the tail call is replaced by the assignment of p_{j+1} . Hence g goes through the same values $p_1, \dots, p_k$ and returns, in the end, what the call with parameters p_k returns; since every call of f returns as it is the value of the following one, this is also the result of f . The k calls of f and the $k - 1$ iterations of g (plus the final test and return) execute the same steps up to a constant per call, so the two numbers of steps are of the same order. Finally, g only uses the parameters and a constant number of temporary variables. $\square$

- A function that is not tail recursive can sometimes be made tail recursive with an extra parameter (an **accumulator**) that carries the partial result: for example `FactAcc(n, acc)` returns acc if $n = 0$ and `FactAcc(n - 1, n · acc)` otherwise, and $n! = \text{FactAcc}(n, 1)$. Its elimination gives the usual loop $acc \leftarrow acc \cdot n, n \leftarrow n - 1$.
- Optimizing compilers often eliminate tail calls automatically: for instance, `g++ 15.2` with the option `-O2` removes the recursive calls of the C++ functions `gcdRec` and `binarySearchRec` of the course code, while with `-O0` it keeps them. However, the C++ standard does not guarantee it, and Python and Java never do it. To be sure that the memory is $\Theta(1)$, write the loop.

Key points on space. The call stack uses memory proportional to the depth of the recursion, not to the number of calls. A logarithmic depth is harmless; a linear depth (decrease by one) risks a stack overflow. Tail recursion can be removed mechanically, obtaining a loop with $\Theta(1)$ extra memory.

18.6 C++ implementation

- All the algorithms of this chapter are implemented in the course code, in the file `Recursion.cpp` (with `Recursion.h`); the experiments of the next section are in the file `Experiments.cpp`, and the sorting algorithms `insertionSort` and `mergeSort` are in the file `Sortings.cpp`. Arrays are indexed from 0.
- These are the three implementations of the array sum:

```

1 long long sumIt(int *a, int n)
2 {
3     long long s = 0;
4     for (int i = 0; i < n; i++)
5     {
6         s = s + a[i];
7     }
8     return s;
9 }
10
11 // sum of a[l..r] by halving: depth Theta(log n)
12 long long sumRecHalving(int *a, int l, int r)
13 {
14     if (l == r)
15     {
16         return a[l];
17     }
18     int mid = l + (r - l) / 2;
19     return sumRecHalving(a, l, mid) + sumRecHalving(a, mid + 1, r);
20 }
21
22 // sum of a[0..n-1] by decrease by one: depth Theta(n)
23 long long sumRecDecrease(int *a, int n)
24 {
25     if (n == 0)
26     {
27         return 0;
28     }
29     return sumRecDecrease(a, n - 1) + a[n - 1];
30 }

```

Example 18.8: the call stack of `sumRecDecrease`

This is the step-by-step execution of `sumRecDecrease(a, 3)` on the array $a = [4, 7, 1]$, showing the content of the call stack (the top of the stack is on the right; `s` stands for `sumRecDecrease` with the array `a` omitted).

$a[j]$	$j = 0$	$j = 1$	$j = 2$
	4	7	1

step	call stack (active calls)	action
1	s(3)	$n = 3 \neq 0$: call s(2), then add $a[2]$
2	s(3) s(2)	$n = 2 \neq 0$: call s(1), then add $a[1]$
3	s(3) s(2) s(1)	$n = 1 \neq 0$: call s(0), then add $a[0]$
4	s(3) s(2) s(1) s(0)	$n = 0$: return 0
5	s(3) s(2) s(1)	return $0 + a[0] = 4$
6	s(3) s(2)	return $4 + a[1] = 11$
7	s(3)	return $11 + a[2] = 12$

At step 4 there are $4 = n + 1$ active calls: the depth of the recursion is $n + 1$. The additions are executed only **after** the base case is reached, while the stack is emptied. For comparison, `sumIt(a, 3)` uses the two variables `s` and `i` only: $s = 0 \rightarrow 4 \rightarrow 11 \rightarrow 12$.

- These are the three implementations of the Fibonacci numbers:

```

1 long long fibonacciRec(int n)
2 {
3     if (n <= 1)
4     {
5         return n;
6     }
7     return fibonacciRec(n - 1) + fibonacciRec(n - 2);
8 }
9
10 // memo must have n+1 entries, all initialised to -1
11 long long fibonacciMemo(int n, long long *memo)
12 {
13     if (n <= 1)
14     {
15         return n;
16     }
17     if (memo[n] == -1)
18     {
19         memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
20     }
21     return memo[n];
22 }
23
24 long long fibonacciIt(int n)
25 {
26     if (n <= 1)
27     {
28         return n;
29     }
30     long long prev = 0; // F(i-2)
31     long long curr = 1; // F(i-1)
32     for (int i = 2; i <= n; i++)
33     {
34         long long next = prev + curr;
35         prev = curr;

```

```

36     curr = next;
37 }
38 return curr;
39 }

```

18.7 Experiment

The asymptotic analysis predicts how the number of steps grows with n . We now check these predictions on a real computer.

18.7.1 Experimental setting

- The running times were measured on 11 October 2026 on a computer with an Intel Core i5-14500T processor (20 logical CPUs, maximum frequency 4.8 GHz), with the operating system Ubuntu 26.04 LTS (Linux kernel 7.0), using the C++ code of the course compiled with g++ 15.2.0 and the option `-O2`. The default stack size is 8 MB (`ulimit -s ⇒ 8192 KB`).
- The function `runRecursionExperiments` of the file `Experiments.cpp` measures the wall-clock time of each computation and repeats it until at least 0.2 seconds (and at least 3 repetitions) have elapsed; each time in the tables is the average over the repetitions.
 - **Sorting:** arrays of $n = 10^3$, 10^4 and 10^5 random integers between 1 and 10^6 (seed 12345), sorted by `insertionSort`, by the recursive `mergeSort` and by `mergeSortBottomUp`.
 - **Fibonacci:** `fibonacciRec(n)` and `fibonacciIt(n)` for $n = 20, 25, 30, 35, 40$; the calls of `fibonacciRec` are counted by a counter in the code; `fibonacciIt` is repeated in blocks of 10^5 calls.
 - **Array sum:** `sumIt`, `sumRecHalving` and `sumRecDecrease` on arrays of $n = 10^5$, 10^6 and 10^7 random integers between 1 and 100, with the stack limit removed (`ulimit -s unlimited`); a separate run checks whether `sumRecDecrease` terminates with the default stack of 8 MB.

Measured times depend on the machine, on the compiler and on its options, on the other programs running, and on the cache memory. They must not be compared in absolute terms: what matters is **how they grow** with n , which is predicted by the asymptotic analysis and is the same on every machine.

18.7.2 Sorting: Insertion Sort versus Merge Sort

n	insertionSort	mergeSort (recursive)	mergeSortBottomUp
10^3	0.0695 ms	0.0361 ms	0.0332 ms
10^4	5.38 ms	0.654 ms	0.629 ms
10^5	525 ms	7.76 ms	7.43 ms

- From $n = 10^4$ to $n = 10^5$ the time of Insertion Sort is multiplied by about 98, close to the factor 100 predicted by $\Theta(n^2)$, while the time of Merge Sort is multiplied by about 12, close to the factor predicted by $\Theta(n \log_2 n)$:

$$10 \cdot \frac{\log_2 10^5}{\log_2 10^4} = 12.5.$$

(From $n = 10^3$ to $n = 10^4$ the factors are smaller, about 77 and 18: for small arrays the constant costs and the cache memory still weigh on the measure.)

- For $n = 10^5$ Merge Sort is about 70 times faster. The recursive and the bottom-up versions have practically the same time: the gain is due to divide and conquer, not to recursion.

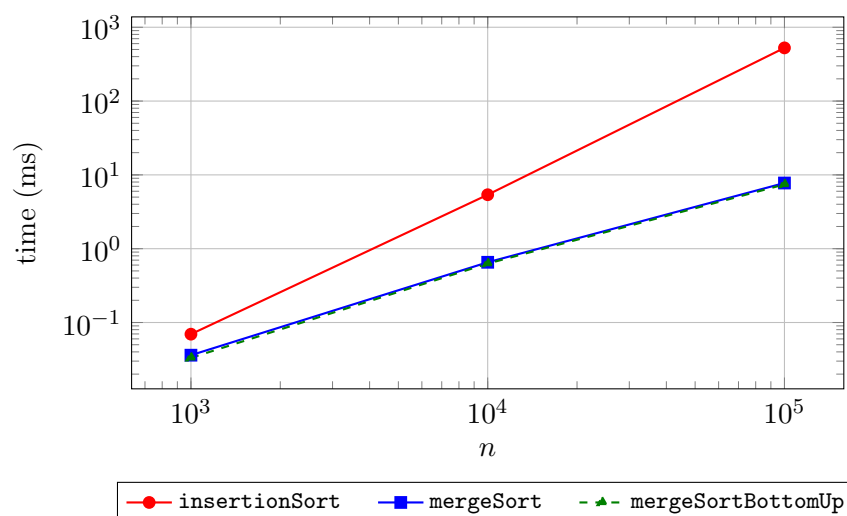


Figure 18.1: Measured running times of the three sorting algorithms (logarithmic scales on both axes).

- In a plot with logarithmic scales on both axes a power n^k is a straight line with slope k : the line of Insertion Sort has slope close to 2, the lines of Merge Sort slope slightly larger than 1.

18.7.3 Fibonacci: naive recursion versus iteration

n	$F(n)$	calls of <code>fibonacciRec</code>	<code>fibonacciRec</code>	<code>fibonacciIt</code>
20	6 765	21 891	0.00808 ms	4.0 ns
25	75 025	242 785	0.0699 ms	4.8 ns
30	832 040	2 692 537	0.764 ms	7.4 ns
35	9 227 465	29 860 703	9.15 ms	8.4 ns
40	102 334 155	331 160 281	93.8 ms	9.5 ns

- The numbers of calls were counted with a counter in the code and coincide with $2F(n+1) - 1$ (Observation 18.4).
- When n increases by 5, the time of `fibonacciRec` is multiplied by a factor between about 9 and 12, in agreement with $\varphi^5 \approx 11.09$. The time of `fibonacciIt` remains of a few nanoseconds.
- Extrapolating with the factor φ per unit of n from the measured time for $n = 40$:
 - `fibonacciRec`(50) needs about $93.8 \cdot \varphi^{10} \approx 11\,500$ ms, i.e., about 12 seconds;
 - `fibonacciRec`(60) needs about $93.8 \cdot \varphi^{20}$ ms, i.e., about 24 minutes;
 - `fibonacciRec`(100) needs about $93.8 \cdot \varphi^{60}$ ms, i.e., about 10 000 years.
- The compiler optimizes the code (with `-O2` it merges several levels of the recursion into a single function call), so the time per call, about $93.8 \text{ ms} / 331\,160\,281 \approx 0.28 \text{ ns}$, is smaller than one would expect: this changes the constant, not the exponential growth.

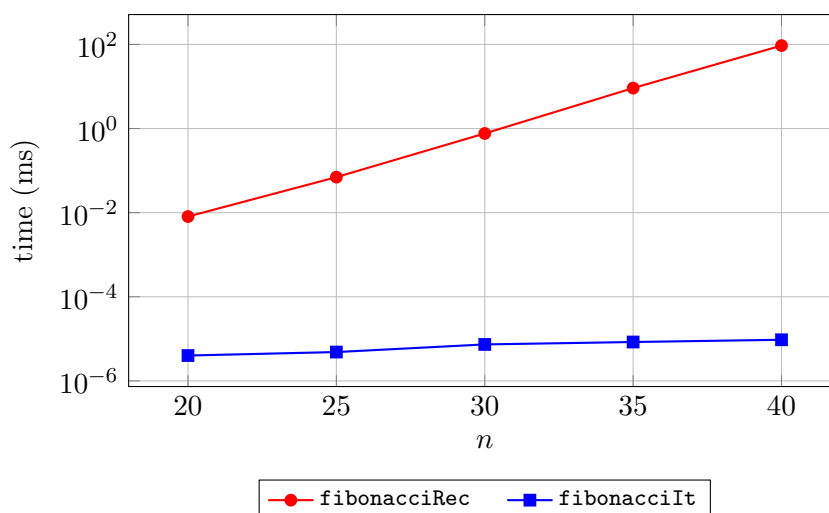


Figure 18.2: Measured running times of the recursive and of the iterative Fibonacci algorithms (logarithmic scale on the vertical axis).

- In a semi-logarithmic plot an exponential function is a straight line: the slope of the red line is about $\log_{10} \varphi \approx 0.21$ per unit of n .

18.7.4 Array sum: time and stack

n	sumIt	sumRecHalving	sumRecDecrease	sumRecDecrease, stack 8 MB
10^5	0.0215 ms	0.0960 ms	0.207 ms	OK
10^6	0.210 ms	0.910 ms	2.24 ms	crash
10^7	2.69 ms	9.01 ms	34.9 ms	crash

- The three algorithms have the same order of growth $\Theta(n)$: when n is multiplied by 10, all the times are multiplied by about 10 (between 9.5 and 16; from $n = 10^6$ to $n = 10^7$ the array no longer fits in the cache memory and the factors increase).
- The recursive versions are slower by a constant factor (about 3–4.5 for the halving version, about 10–13 for the decrease-by-one version): this is the call overhead.
- With the default stack of 8 MB, **sumRecDecrease** crashes (segmentation fault) for $n = 10^6$ and $n = 10^7$; the times of the table were measured by removing the limit with the command `ulimit -s unlimited`.
- By bisection on n (with a precision of 10^3) we found that, with 8 MB, the largest n that does not crash is about $6.97 \cdot 10^5$ with `-O2`, and about $2.62 \cdot 10^5$ without optimization (`-O0`). Without optimization every call uses a frame of 32 bytes:

$$\frac{8 \cdot 2^{20}}{32} = 262\,144.$$

With `-O2` the compiler merges four levels of the recursion into a single call with a frame of 48 bytes, so that

$$\frac{8 \cdot 2^{20}}{48} \cdot 4 \approx 6.99 \cdot 10^5 :$$

optimization changes the constant, not the $\Theta(n)$ depth.

- **sumRecHalving** uses only $\lceil \log_2 n \rceil + 1 \leq 25$ frames for $n \leq 10^7$ and never crashes; **sumIt** uses no frame at all.

18.8 Guidelines

Use recursion when:

- the problem splits into a few **independent** subproblems of size n/b (divide and conquer), so that the Master Theorem gives a better running time than the iterative algorithms (Merge Sort, fast power, Karatsuba);
- the recursive description is the natural one and no asymptotic gain is possible anyway (Tower of Hanoi, and later visits of trees and graphs);
- the depth of the recursion is small, typically $\Theta(\log_2 n)$.

Avoid recursion when:

- the problem only decreases from n to $n - 1$ with constant extra work: the recursion is a loop in disguise, with the same running time, a call overhead, $\Theta(n)$ stack memory and a risk of stack overflow (factorial, linear sum, linear search);
- the subproblems **overlap**: the naive recursion repeats the same computations and can be exponential (Fibonacci); use memoization or, better, a bottom-up loop (Dynamic Programming).

Always consider the space: the call stack uses memory proportional to the depth of the recursion. Tail recursion can be eliminated mechanically, obtaining a loop with $\Theta(1)$ extra memory.

Always compare with the best iterative algorithm: a recursive algorithm can improve on brute force and still be beaten by a smarter loop (Kadane's algorithm for the maximum subarray).