



SAPIENZA
UNIVERSITÀ DI ROMA

Algorithms

Exercises 5 — Recursion

Worked exercises on chapters 14–18: recursion, recurrence relations, the Master Theorem, recursive algorithms, recursive versus iterative. Every exercise has a complete solution.

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	2
15	Recurrences	19
16	The Master Theorem	34
17	Recursive algorithms	47
18	Recursive versus iterative algorithms	86



Recursion

In the exercises of this chapter the steps of a recursive algorithm are counted with the conventions of the chapters «Running time» and «Recursion», and the recurrences are solved by simple unrolling.

- Every execution of a numbered 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 call.
- $C(n)$ denotes the number of **calls** (nodes of the tree of recursive calls) and $S(n)$ the number of **steps**; the **depth of the recursion** is the maximum number of nodes on a path from the root to a leaf of the tree, i.e., the maximum number of stack frames simultaneously in memory.
- **Strong induction**: if $P(n_0)$ is true and, for every $n > n_0$, the truth of $P(k)$ for every k with $n_0 \leq k < n$ implies the truth of $P(n)$, then $P(n)$ is true for every $n \geq n_0$.
- **Unrolling** a recurrence $S(n) = S(n-1) + c$ with constant c :

$$S(n) = S(n-1) + c = S(n-2) + 2c = \dots = S(n-k) + kc,$$

and the base case is reached for the value of k that makes $n-k$ a base case.

Exercise 14.1 — Recursion (E1)

Consider the following recursive algorithm.

Algorithm: Mystery**Input** : A natural number $n \geq 1$ **Output**: A natural number

```

1 function Mystery( $n$ )
2   if  $n = 1$  then
3     return 1
4   end
5   return Mystery( $\lfloor n/2 \rfloor$ ) + Mystery( $\lceil n/2 \rceil$ )
6 end

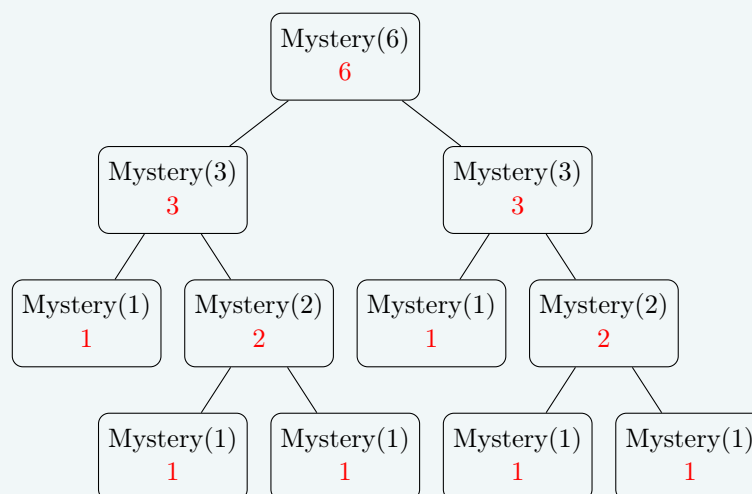
```

Questions

1. Draw the tree of recursive calls of **Mystery**(6), writing next to each call the value it returns.
2. What does the algorithm compute? Prove your claim by strong induction on n .
3. Let $C(n)$ be the number of calls executed by **Mystery**(n). Write a recurrence for $C(n)$ and prove that $C(n) = 2n - 1$ for every $n \geq 1$. Check the formula on the tree of question 1.
4. What is the depth of the recursion of **Mystery**(6), i.e., the maximum number of stack frames simultaneously in memory?

Solution

1. We have $\lfloor 6/2 \rfloor = \lceil 6/2 \rceil = 3$, $\lfloor 3/2 \rfloor = 1$, $\lceil 3/2 \rceil = 2$ and $\lfloor 2/2 \rfloor = \lceil 2/2 \rceil = 1$. The tree of recursive calls (returned values in red) is:



Mystery(2) returns $1 + 1 = 2$, **Mystery(3)** returns $1 + 2 = 3$ and **Mystery(6)** returns $3 + 3 = 6$.

2. The algorithm returns n itself (it computes n as a sum of n ones).

Proof by strong induction on $n \geq 1$.

- **Base case** ($n = 1$): the algorithm returns 1 at line 3.
- **Inductive step** ($n \geq 2$): assume that **Mystery**(k) returns k for every $1 \leq k < n$. For $n \geq 2$ we have $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$, so by the inductive hypothesis line 5 returns

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n.$$

Strong induction is necessary because the recursive calls are on $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, not on $n - 1$.

3. Each call with $n \geq 2$ makes two recursive calls, hence

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

Proof of $C(n) = 2n - 1$ by strong induction on n .

- **Base case** ($n = 1$): $C(1) = 1 = 2 \cdot 1 - 1$.
- **Inductive step** ($n \geq 2$): by the inductive hypothesis,

$$C(n) = 1 + (2\lfloor n/2 \rfloor - 1) + (2\lceil n/2 \rceil - 1) = 2(\lfloor n/2 \rfloor + \lceil n/2 \rceil) - 1 = 2n - 1.$$

For $n = 6$: $C(6) = 11$, and indeed the tree of question 1 has 11 nodes. (The tree has $n = 6$ leaves, one for each 1 in the sum, and $n - 1 = 5$ internal nodes, one for each addition.)

4. The longest paths from the root to a leaf are, e.g., **Mystery(6)**, **Mystery(3)**, **Mystery(2)**, **Mystery(1)**. Hence the depth of the recursion is 4: at most 4 stack frames are simultaneously in memory, although 11 calls are executed. (In general the depth is $\lceil \log_2 n \rceil + 1$.)

Exercise 14.2 — Recursion (E2)

Given an array $a[1..n]$ of n integer numbers, consider the problems of computing

- the **sum** $a[1] + a[2] + \dots + a[n]$ (equal to 0 if $n = 0$);
- the **maximum** value $\max\{a[1], a[2], \dots, a[n]\}$ (for $n \geq 1$).

Use the array $a = [3, 8, 1, 6]$ for the numerical questions.

Questions

1. Write a recursive algorithm $\text{RecSum}(a, n)$ that computes the sum of $a[1..n]$ by reducing the problem to the sub-array $a[1..n-1]$ (decrease by one).
2. Write a recursive algorithm $\text{RecMax}(a, n)$ that computes the maximum of $a[1..n]$ by reducing the problem to the sub-array $a[1..n-1]$.
3. Show the sequence of calls and returns of $\text{RecSum}(a, 4)$ and of $\text{RecMax}(a, 4)$ for $a = [3, 8, 1, 6]$.
4. Prove the correctness of RecMax by induction on n .
5. Write the recurrences of the number of steps $S(n)$ of the two algorithms, solve them by unrolling, and give the depth of the recursion.

Solution

1. The sum of $a[1..n]$ is the sum of $a[1..n-1]$ plus $a[n]$, and the sum of an empty array is 0:

Algorithm: RecSum

Input : An array $a[1..n]$ of $n \geq 0$ integer numbers

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

```

1 function RecSum( $a, n$ )
2   if  $n = 0$  then
3     return 0
4   end
5   return RecSum( $a, n - 1$ ) +  $a[n]$ 
6 end

```

2. The maximum of $a[1..n]$ is the larger between the maximum of $a[1..n-1]$ and $a[n]$; the maximum of a single element $a[1]$ is $a[1]$:

Algorithm: RecMax**Input** : An array $a[1..n]$ of $n \geq 1$ integer numbers**Output**: The maximum value in $a[1..n]$

```

1 function RecMax( $a, n$ )
2   if  $n = 1$  then
3     return  $a[1]$ 
4   end
5    $m := \text{RecMax}(a, n - 1)$ 
6   if  $a[n] > m$  then
7     return  $a[n]$ 
8   end
9   return  $m$ 
10 end

```

3. For $a = [3, 8, 1, 6]$:

RecSum($a, 4$): calls

$\text{RecSum}(a, 4) \rightarrow \text{RecSum}(a, 3) \rightarrow \text{RecSum}(a, 2) \rightarrow \text{RecSum}(a, 1) \rightarrow \text{RecSum}(a, 0)$,

then the returns:

call	returns
$\text{RecSum}(a, 0)$	0 (base case)
$\text{RecSum}(a, 1)$	$0 + a[1] = 0 + 3 = 3$
$\text{RecSum}(a, 2)$	$3 + a[2] = 3 + 8 = 11$
$\text{RecSum}(a, 3)$	$11 + a[3] = 11 + 1 = 12$
$\text{RecSum}(a, 4)$	$12 + a[4] = 12 + 6 = 18$

RecMax($a, 4$): calls $\text{RecMax}(a, 4) \rightarrow \text{RecMax}(a, 3) \rightarrow \text{RecMax}(a, 2) \rightarrow \text{RecMax}(a, 1)$,

then the returns:

call	comparison	returns
$\text{RecMax}(a, 1)$	– (base case)	$a[1] = 3$
$\text{RecMax}(a, 2)$	$m = 3, a[2] = 8 > 3$	8
$\text{RecMax}(a, 3)$	$m = 8, a[3] = 1 \leq 8$	8
$\text{RecMax}(a, 4)$	$m = 8, a[4] = 6 \leq 8$	8

The sum is 18 and the maximum is 8.

4. **Claim:** for every $n \geq 1$, **RecMax**(a, n) returns $\max\{a[1], \dots, a[n]\}$.

- **Base case** ($n = 1$): the algorithm returns $a[1] = \max\{a[1]\}$ at line 3.
- **Inductive step** ($n \geq 2$): by the inductive hypothesis, line 5 assigns

$$m = \max\{a[1], \dots, a[n-1]\}.$$

If $a[n] > m$, then $a[n]$ is larger than all the elements of $a[1..n-1]$ and the algorithm returns $a[n] = \max\{a[1], \dots, a[n]\}$ (line 7). Otherwise $a[n] \leq m$ and the algorithm returns $m = \max\{a[1], \dots, a[n]\}$ (line 9).

5. **RecSum.** $S(0) = 2$ (lines 2–3) and, for $n \geq 1$, lines 2 and 5 plus the recursive call:

$$S(n) = S(n-1) + 2 = S(n-2) + 2 \cdot 2 = \dots = S(0) + 2n = 2n + 2 = \Theta(n).$$

RecMax. $S(1) = 2$ (lines 2–3) and, for $n \geq 2$, lines 2, 5, 6 and one of the lines 7, 9, plus the recursive call:

$$S(n) = S(n-1) + 4 = S(n-2) + 2 \cdot 4 = \dots = S(1) + 4(n-1) = 4n - 2 = \Theta(n).$$

For $n = 4$: $S(4) = 10$ for **RecSum** and $S(4) = 14$ for **RecMax**.

The depth of the recursion is $n + 1$ for **RecSum** (calls on $n, n-1, \dots, 0$) and n for **RecMax** (calls on $n, \dots, 1$): both use $\Theta(n)$ memory for the call stack, while the iterative versions (as **FindMaximum** in the chapter «Running time») use constant memory.

If all the entries use a constant number of bits ($|I| = \Theta(n)$), the worst-case running time of both **RecSum** and **RecMax** is $T(|I|) = \Theta(|I|)$.

Exercise 14.3 — Recursion (E3)

An array $a[1..n]$ is a **palindrome** if it reads the same from left to right and from right to left, i.e., $a[i] = a[n+1-i]$ for every $i \in \{1, 2, \dots, n\}$. Consider the following recursive algorithm, called on the whole array as **IsPalindrome**($a, 1, n$).

Algorithm: IsPalindrome

Input : An array a and two indices l, r

Output: **true** if the sub-array $a[l..r]$ is a palindrome, **false** otherwise

```

1 function IsPalindrome( $a, l, r$ )
2   if  $l \geq r$  then
3     return true
4   end
5   if  $a[l] \neq a[r]$  then
6     return false
7   end
8   return IsPalindrome( $a, l+1, r-1$ )
9 end
```

Questions

1. Show the sequence of calls of `IsPalindrome(a, 1, 5)` for the arrays $a = [1, 2, 3, 2, 1]$ and $a = [1, 2, 3, 3, 1]$.
2. Prove by induction on the length $k = r - l + 1$ of the sub-array that the algorithm is correct.
3. Determine the worst-case number of steps $S(n)$ of `IsPalindrome(a, 1, n)` and its depth of recursion.
4. Is the algorithm tail recursive? If yes, write the equivalent iterative algorithm.
5. Write a recursive algorithm `Reverse(a, l, r)` that reverses the sub-array $a[l..r]$ in place, and show its execution on $a = [5, 7, 2, 9, 4, 6]$ with $l = 1, r = 6$.

Solution

1. For $a = [1, 2, 3, 2, 1]$:

call	$a[l]$	$a[r]$	action
<code>IsPalindrome(a, 1, 5)</code>	1	1	equal $\Rightarrow$ calls <code>IsPalindrome(a, 2, 4)</code>
<code>IsPalindrome(a, 2, 4)</code>	2	2	equal $\Rightarrow$ calls <code>IsPalindrome(a, 3, 3)</code>
<code>IsPalindrome(a, 3, 3)</code>	–	–	$l \geq r \Rightarrow$ returns true

The value **true** is returned unchanged by all the calls (3 calls).

For $a = [1, 2, 3, 3, 1]$:

call	$a[l]$	$a[r]$	action
<code>IsPalindrome(a, 1, 5)</code>	1	1	equal $\Rightarrow$ calls <code>IsPalindrome(a, 2, 4)</code>
<code>IsPalindrome(a, 2, 4)</code>	2	3	different $\Rightarrow$ returns false

The value **false** is returned (2 calls).

2. Let $k = r - l + 1$ be the length of $a[l..r]$ ($k \leq 0$ means an empty sub-array). We prove by strong induction on k that the algorithm returns **true** if and only if $a[l..r]$ is a palindrome.
 - **Base cases** ($k \leq 1$, i.e., $l \geq r$): an empty array or an array with one element is a palindrome, and the algorithm returns **true** at line 3.
 - **Inductive step** ($k \geq 2$): $a[l..r]$ is a palindrome if and only if $a[l] = a[r]$ and the inner sub-array $a[l + 1..r - 1]$ is a palindrome. If $a[l] \neq a[r]$, the algorithm correctly returns **false** (line 6). Otherwise it returns the result of the call on

$a[l + 1..r - 1]$, whose length $k - 2 < k$; by the inductive hypothesis, this result is **true** if and only if $a[l + 1..r - 1]$ is a palindrome, hence if and only if $a[l..r]$ is a palindrome.

The length decreases by 2 at each call, which is why both $k = 0$ and $k = 1$ must be base cases.

3. The worst case is when the array is a palindrome (the recursion is never stopped at line 6). Every call with $l < r$ executes lines 2, 5 and 8 (3 steps), and the final call executes lines 2–3 (2 steps). In terms of the length k :

$$S(k) = S(k - 2) + 3 \quad (k \geq 2), \quad S(0) = S(1) = 2.$$

Unrolling, after $\lfloor k/2 \rfloor$ steps we reach $S(0)$ or $S(1)$:

$$S(n) = 3\lfloor n/2 \rfloor + 2 = \Theta(n).$$

The algorithm makes $\lfloor n/2 \rfloor + 1$ calls, which is also the depth of the recursion (the tree is a path). For $n = 5$: 3 calls, as in question 1, and $S(5) = 8$.

4. Yes: the recursive call at line 8 is the last operation and its result is returned unchanged. Replacing the tail call by the update $(l, r) \leftarrow (l + 1, r - 1)$ we obtain:

Algorithm: IterativeIsPalindrome

Input : An array $a[1..n]$

Output: **true** if a is a palindrome, **false** otherwise

```

1 function IterativeIsPalindrome( $a, n$ )
2    $l := 1$ 
3    $r := n$ 
4   while  $l < r$  do
5     if  $a[l] \neq a[r]$  then
6       return false
7     end
8      $l \leftarrow l + 1$ 
9      $r \leftarrow r - 1$ 
10  end
11  return true
12 end
```

It uses constant memory instead of a stack of depth $\lfloor n/2 \rfloor + 1$.

5. To reverse $a[l..r]$: swap the two extreme elements, then reverse the inner sub-array $a[l + 1..r - 1]$. A sub-array with at most one element is already reversed (base case, nothing to do).

Algorithm: Reverse**Input** : An array a and two indices l, r **Output**: The sub-array $a[l..r]$ reversed in place

```

1 function Reverse( $a, l, r$ )
2   if  $l < r$  then
3     swap  $a[l]$  and  $a[r]$ 
4     Reverse( $a, l + 1, r - 1$ )
5   end
6 end

```

Execution on $a = [5, 7, 2, 9, 4, 6]$:

call	action	array after the call's swap
Reverse($a, 1, 6$)	swap $a[1] = 5, a[6] = 6$	[6, 7, 2, 9, 4, 5]
Reverse($a, 2, 5$)	swap $a[2] = 7, a[5] = 4$	[6, 4, 2, 9, 7, 5]
Reverse($a, 3, 4$)	swap $a[3] = 2, a[4] = 9$	[6, 4, 9, 2, 7, 5]
Reverse($a, 4, 3$)	$l \geq r$: base case, nothing to do	[6, 4, 9, 2, 7, 5]

The final array is [6, 4, 9, 2, 7, 5], after $\lfloor 6/2 \rfloor + 1 = 4$ calls and 3 swaps. The correctness follows by induction on $k = r - l + 1$ exactly as in question 2.

Exercise 14.4 — Recursion (E4)

Consider the naive recursive algorithm for the Fibonacci numbers ($F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$ for $n \geq 2$).

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  $n$ 
4   end
5   return Fib( $n - 1$ ) + Fib( $n - 2$ )
6 end

```

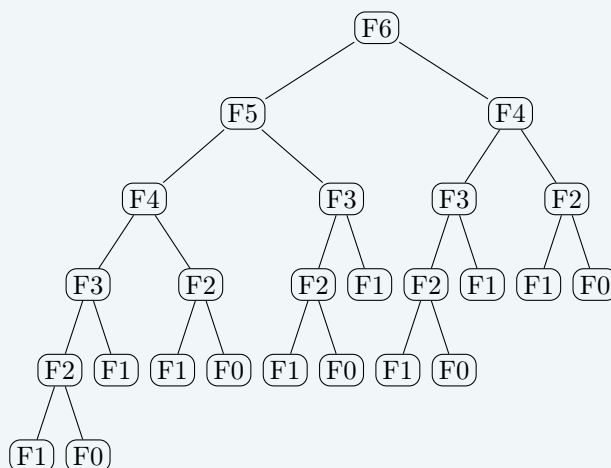
Questions

1. Draw the tree of recursive calls of Fib(6) and compute the value returned.

2. Count the number of calls $C(6)$ and check the formula $C(n) = 2F(n+1) - 1$.
3. How many times is $\text{Fib}(k)$ called, for each $k = 0, 1, \dots, 6$, during the computation of $\text{Fib}(6)$? What do you observe?
4. Compare the number of steps of Fib ($S(n) = 4F(n+1) - 2$) with the number of steps of IterativeFib ($S(n) = 5n + 1$) for $n = 6$ and for $n = 20$.
5. What is the depth of the recursion of $\text{Fib}(6)$?

Solution

1. The tree of recursive calls of $\text{Fib}(6)$ (we write Fk for $\text{Fib}(k)$):



Returning from the leaves to the root: $\text{Fib}(2) = 1 + 0 = 1$, $\text{Fib}(3) = 1 + 1 = 2$, $\text{Fib}(4) = 2 + 1 = 3$, $\text{Fib}(5) = 3 + 2 = 5$, $\text{Fib}(6) = 5 + 3 = 8$. Hence $F(6) = 8$.

2. Counting the nodes of the tree level by level: $1 + 2 + 4 + 8 + 8 + 2 = 25$. The formula gives

$$C(6) = 2F(7) - 1 = 2 \cdot 13 - 1 = 25.$$

Equivalently, from the recurrence $C(n) = C(n-1) + C(n-2) + 1$ with $C(0) = C(1) = 1$: $C(2) = 3$, $C(3) = 5$, $C(4) = 9$, $C(5) = 15$, $C(6) = 15 + 9 + 1 = 25$.

3. Counting the nodes with the same label:

k	6	5	4	3	2	1	0	total
number of calls $\text{Fib}(k)$	1	1	2	3	5	8	5	25

The numbers of calls are themselves Fibonacci numbers: $\text{Fib}(k)$ is called $F(7 - k)$ times for $k \geq 1$ (and $\text{Fib}(0)$ is called $F(5)$ times, once for each call $\text{Fib}(2)$). The same subproblems are solved again and again: for example $F(2)$ is computed 5 times, while it would be enough to compute it once and store it. This is the motivation of **Dynamic Programming**.

4. The number of calls and of steps of the two algorithms are:

n	calls $C(n) = 2F(n+1) - 1$	steps of Fib : $4F(n+1) - 2$	steps of IterativeFib : $5n + 1$
6	$2 \cdot 13 - 1 = 25$	$4 \cdot 13 - 2 = 50$	31
20	$2 \cdot 10946 - 1 = 21891$	$4 \cdot 10946 - 2 = 43782$	101

For $n = 6$ the two algorithms are comparable, but for $n = 20$ the naive algorithm executes more than 400 times the number of steps of the iterative one: the first grows exponentially, the second linearly.

5. The longest path from the root to a leaf is $\text{Fib}(6)$, $\text{Fib}(5)$, $\text{Fib}(4)$, $\text{Fib}(3)$, $\text{Fib}(2)$, $\text{Fib}(1)$: the depth of the recursion is 6. Although 25 calls are executed, at most 6 stack frames are simultaneously in memory.

Exercise 14.5 — Recursion (E5)

Consider the Towers of Hanoi with $n = 4$ disks, numbered 1, 2, 3, 4 from the smallest to the largest, initially on peg A , to be moved to peg C using peg B as auxiliary peg, with the recursive algorithm:

Algorithm: Hanoi

Input : The number of disks $n \geq 1$, the pegs $from$, to , 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  $to$ 
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

```

Questions

1. List all the moves executed by $\text{Hanoi}(4, A, C, B)$, in order.

2. Check that the number of moves is $M(4) = 2^4 - 1$.
3. How many times is each disk moved? Prove that, with n disks, disk k is moved exactly 2^{n-k} times ($\forall k \in \{1, 2, \dots, n\}$), and check that $\sum_{k=1}^n 2^{n-k} = 2^n - 1$.
4. How many calls of `Hanoi` are executed for $n = 4$, and what is the depth of the recursion?
5. According to the legend, $n = 64$ golden disks are moved at a speed of one move per second. How long does it take?

Solution

1. `Hanoi(4, A, C, B)` executes `Hanoi(3, A, B, C)`, then moves disk 4 from A to C , then executes `Hanoi(3, B, C, A)`. Each call with $n = 3$ moves 7 disks (in the same pattern as `Hanoi(3, A, C, B)` in the chapter, with the pegs renamed):

move	disk	from	to	move	disk	from	to
1	1	A	B	9	1	B	C
2	2	A	C	10	2	B	A
3	1	B	C	11	1	C	A
4	3	A	B	12	3	B	C
5	1	C	A	13	1	A	B
6	2	C	B	14	2	A	C
7	1	A	B	15	1	B	C
8	4	A	C				

Moves 1–7 transfer disks 1, 2, 3 from A to B (call `Hanoi(3, A, B, C)`), move 8 moves the largest disk, and moves 9–15 transfer disks 1, 2, 3 from B to C (call `Hanoi(3, B, C, A)`).

2. There are 15 moves and $2^4 - 1 = 15$. Equivalently, $M(1) = 1$, $M(2) = 2 \cdot 1 + 1 = 3$, $M(3) = 2 \cdot 3 + 1 = 7$, $M(4) = 2 \cdot 7 + 1 = 15$.
3. From the table: disk 1 is moved 8 times, disk 2 is moved 4 times, disk 3 is moved 2 times and disk 4 is moved once; these are $2^{4-1}, 2^{4-2}, 2^{4-3}, 2^{4-4}$.

Proof by induction on n that, in `Hanoi`($n, \cdot, \cdot, \cdot$), disk k is moved 2^{n-k} times for every $k \in \{1, 2, \dots, n\}$.

- **Base case** ($n = 1$): disk 1 is moved once $= 2^{1-1}$.
- **Inductive step** ($n \geq 2$): disk n is moved only by the call itself, once $= 2^{n-n}$. A disk $k \leq n - 1$ is moved only inside the two recursive calls with $n - 1$ disks, and by the inductive hypothesis it is moved $2^{(n-1)-k}$ times in each of them: in total $2 \cdot 2^{n-1-k} = 2^{n-k}$ times.

Summing over all the disks (geometric sum),

$$\sum_{k=1}^n 2^{n-k} = 2^{n-1} + 2^{n-2} + \dots + 2 + 1 = 2^n - 1 = M(n).$$

For $n = 4$: $8 + 4 + 2 + 1 = 15$.

4. Every call executes exactly one move (the base case moves disk 1, the other calls move disk n), hence the number of calls is equal to the number of moves: 15 calls. The tree of recursive calls is a complete binary tree with 4 levels ($1 + 2 + 4 + 8 = 15$ nodes), so the depth of the recursion is 4.
5. $M(64) = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ moves, i.e., the same number of seconds. One year has about $365.25 \cdot 24 \cdot 3600 = 31\,557\,600$ seconds, so the time is

$$\frac{2^{64} - 1}{31\,557\,600} \approx 5.85 \cdot 10^{11} \text{ years,}$$

about 585 billion years (more than 40 times the age of the universe). Since the number of moves of `Hanoi` is optimal, no algorithm can do better.

Exercise 14.6 — Recursion (E6)

Consider the following two recursive algorithms: `FastPower` computes x^n by squaring, `ToBinary` prints the binary representation of a natural number n .

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
4   end
5    $y := \text{FastPower}(x, \lfloor n/2 \rfloor)$ 
6   if  $n \bmod 2 = 0$  then
7     return  $y \cdot y$ 
8   else
9     return  $x \cdot y \cdot y$ 
10  end
11 end

```

Algorithm: ToBinary**Input** : A natural number n **Output**: The binary digits of n , printed from the most significant one

```

1 function ToBinary( $n$ )
2   if  $n \leq 1$  then
3     print  $n$ 
4   else
5     ToBinary( $\lfloor n/2 \rfloor$ )
6     print  $n \bmod 2$ 
7   end
8 end

```

Questions

1. Show the sequence of calls of **FastPower**($x, 25$) and count the number of multiplications. Compare with the naive algorithm, which executes n multiplications.
2. Compute the number of multiplications and the number of steps $S(n) = 4\lfloor \log_2 n \rfloor + 6$ of **FastPower** for $n = 31$, $n = 32$ and $n = 100$.
3. A student replaces the line $y := \text{FastPower}(x, \lfloor n/2 \rfloor)$ and the following lines by

return **FastPower**($x, \lfloor n/2 \rfloor$) · **FastPower**($x, \lfloor n/2 \rfloor$) (times x if n is odd).

Write the recurrence of the number of calls $C(n)$ of this version, compute $C(13)$ and $C(16)$, and show that $C(n) = \Theta(n)$.

4. Show the execution of `ToBinary(37)`: calls, printed digits, and number of calls. Why are the digits printed in the correct order, while the algorithm `ChangeOfBase` (chapter «Numerical algorithms») produces them from the least significant one?
5. Prove by strong induction that `ToBinary( $n$ )` prints the binary representation of n , and that it executes $\lfloor \log_2 n \rfloor + 1$ calls for $n \geq 1$.

Solution

1. The exponents of the calls are 25, 12, 6, 3, 1, 0. Going back from the base case:

call	y	returns	multiplications
<code>FastPower($x, 0$)</code>	–	1 (base case)	0
<code>FastPower($x, 1$)</code>	1	$x \cdot 1 \cdot 1 = x$ (odd)	2
<code>FastPower($x, 3$)</code>	x	$x \cdot x \cdot x = x^3$ (odd)	2
<code>FastPower($x, 6$)</code>	x^3	$x^3 \cdot x^3 = x^6$ (even)	1
<code>FastPower($x, 12$)</code>	x^6	$x^6 \cdot x^6 = x^{12}$ (even)	1
<code>FastPower($x, 25$)</code>	x^{12}	$x \cdot x^{12} \cdot x^{12} = x^{25}$ (odd)	2

In total $2 + 2 + 1 + 1 + 2 = 8$ multiplications, against 25 for the naive algorithm. Note that $25 = (11001)_2$: 5 binary digits, three of which equal to 1, and $5 + 3 = 8$.

2. The number of multiplications is the number of binary digits of n plus the number of digits equal to 1 (each call with $n \geq 1$ executes 1 multiplication if n is even, 2 if n is odd, and the parity of the successive exponents gives the binary digits of n):

n	binary	multiplications	$S(n) = 4\lfloor \log_2 n \rfloor + 6$	naive
31	11111	$5 + 5 = 10$	$4 \cdot 4 + 6 = 22$	31
32	10000	$6 + 1 = 7$	$4 \cdot 5 + 6 = 26$	32
100	1100100	$7 + 3 = 10$	$4 \cdot 6 + 6 = 30$	100

Note that x^{32} requires fewer multiplications than x^{31} , although $32 > 31$.

3. Each call with $n \geq 1$ now makes **two** recursive calls on $\lfloor n/2 \rfloor$:

$$C(n) = 2C(\lfloor n/2 \rfloor) + 1 \quad (n \geq 1), \quad C(0) = 1.$$

Hence $C(1) = 3$, $C(3) = 7$, $C(6) = 15$, $C(13) = 31$, and $C(2) = 7$, $C(4) = 15$, $C(8) = 31$, $C(16) = 63$.

The tree of recursive calls is now a complete binary tree. Its levels correspond to the exponents $n, \lfloor n/2 \rfloor, \dots, 1, 0$, i.e., there are $L = \lfloor \log_2 n \rfloor + 2$ levels, with $1, 2, 4, \dots, 2^{L-1}$ nodes. Therefore

$$C(n) = 1 + 2 + \dots + 2^{L-1} = 2^L - 1 = 4 \cdot 2^{\lfloor \log_2 n \rfloor} - 1.$$

(Check: $C(13) = 4 \cdot 8 - 1 = 31$, $C(16) = 4 \cdot 16 - 1 = 63$.) Since $n/2 < 2^{\lfloor \log_2 n \rfloor} \leq n$, we get $2n - 1 < C(n) \leq 4n - 1$, i.e., $C(n) = \Theta(n)$: the version with two identical recursive calls is **no better than the naive algorithm**. Computing the value once and storing it in y is essential.

4. Execution of ToBinary(37):

Calls:

ToBinary(37) $\rightarrow$ ToBinary(18) $\rightarrow$ ToBinary(9) $\rightarrow$
ToBinary(4) $\rightarrow$ ToBinary(2) $\rightarrow$ ToBinary(1)

Prints (in the order in which the calls end):

call	action	printed so far
ToBinary(1)	base case, prints 1	1
ToBinary(2)	prints $2 \bmod 2 = 0$	10
ToBinary(4)	prints $4 \bmod 2 = 0$	100
ToBinary(9)	prints $9 \bmod 2 = 1$	1001
ToBinary(18)	prints $18 \bmod 2 = 0$	10010
ToBinary(37)	prints $37 \bmod 2 = 1$	100101

Hence $37 = (100101)_2$ (indeed $32 + 4 + 1 = 37$), with 6 calls.

The remainders $n \bmod 2$ are computed in the same order as in **ChangeOfBase** (repeated division: 1, 0, 1, 0, 0, 1, from the least significant digit). However, each call prints its remainder **after** its recursive call has returned, i.e., after all the more significant digits have been printed: the call stack reverses the order of the digits, and no array is needed.

5. Correctness, by strong induction on n .

- **Base cases** ($n \in \{0, 1\}$): the binary representation of n is the single digit n , which is printed.
- **Inductive step** ($n \geq 2$): we have $n = 2\lfloor n/2 \rfloor + (n \bmod 2)$ with $1 \leq \lfloor n/2 \rfloor < n$. If $\lfloor n/2 \rfloor = (d_{k-1} \dots d_0)_2$, then

$$n = 2 \sum_{i=0}^{k-1} d_i 2^i + (n \bmod 2) = (d_{k-1} \dots d_0 (n \bmod 2))_2,$$

i.e., the binary representation of n is that of $\lfloor n/2 \rfloor$ followed by the digit $n \bmod 2$. By the inductive hypothesis the recursive call prints $d_{k-1} \dots d_0$, and then the algorithm prints $n \bmod 2$.

Number of calls. Let $C(n)$ be the number of calls for $n \geq 1$: $C(1) = 1$ and $C(n) = C(\lfloor n/2 \rfloor) + 1$ for $n \geq 2$. We prove $C(n) = \lfloor \log_2 n \rfloor + 1$ by strong induction: $C(1) = 1 = \lfloor \log_2 1 \rfloor + 1$; for $n \geq 2$, using $\lfloor \log_2 \lfloor n/2 \rfloor \rfloor = \lfloor \log_2 n \rfloor - 1$,

$$C(n) = (\lfloor \log_2 n \rfloor - 1 + 1) + 1 = \lfloor \log_2 n \rfloor + 1.$$

This is the number of binary digits of n (one digit printed per call). For $n = 37$: $\lfloor \log_2 37 \rfloor + 1 = 5 + 1 = 6$.

Recurrences

In the exercises of this chapter recurrences are solved with the three methods of the chapter «Recurrences» (expansion, substitution, recursion tree), using the following formulas.

For every $n, m \in \mathbb{N}$, $n, m \geq 1$, every $k \in \mathbb{N}$ and every real number $x \neq 1$:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad (\text{arithmetic series})$$

$$\sum_{j=1}^m j^2 = \frac{m(m+1)(2m+1)}{6} \quad (\text{sum of the squares})$$

$$\sum_{i=0}^k x^i = \frac{x^{k+1} - 1}{x - 1} = \frac{1 - x^{k+1}}{1 - x} \quad (\text{geometric series})$$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x} \quad \text{if } |x| < 1 \quad (\text{infinite decreasing geometric series})$$

For every $a, b, n > 0$ with $b \neq 1$, and every real number x :

$$a^{\log_b n} = n^{\log_b a}, \quad \log_b n = \frac{\log_2 n}{\log_2 b},$$

$$x - 1 < \lfloor x \rfloor \leq x, \quad 2\lfloor n/2 \rfloor \geq n - 1 \quad (n \in \mathbb{N}).$$

Exercise 15.1 — Recurrences (E1)

Consider the following recurrences.

- (a) $T(n) = T(n-1) + 2n - 1$ for $n > 1$, with $T(1) = 1$.
- (b) $T(n) = 3T(n-1)$ for $n > 1$, with $T(1) = 2$.
- (c) $T(n) = T\left(\frac{n}{3}\right) + 1$ for $n > 1$, with $T(1) = 1$, where n is an exact power of 3.
- (d) $T(n) = 2T\left(\frac{n}{2}\right) + n^2$ for $n > 1$, with $T(1) = 1$, where n is an exact power of 2.

Questions

1. For each recurrence, compute the first four values of $T(n)$ (for (c) and (d), on the first four admissible values of n).
2. Solve each recurrence with the expansion method, that is, determine a closed form of $T(n)$.
3. Check that the closed forms are consistent with the values computed in question 1.
4. Determine the order of growth of each recurrence (Θ -notation).

Solution

1. First values:

- (a) $T(1) = 1$, $T(2) = 1 + 3 = 4$, $T(3) = 4 + 5 = 9$, $T(4) = 9 + 7 = 16$.
- (b) $T(1) = 2$, $T(2) = 6$, $T(3) = 18$, $T(4) = 54$.
- (c) $T(1) = 1$, $T(3) = T(1) + 1 = 2$, $T(9) = T(3) + 1 = 3$, $T(27) = T(9) + 1 = 4$.
- (d) $T(1) = 1$, $T(2) = 2 \cdot 1 + 4 = 6$, $T(4) = 2 \cdot 6 + 16 = 28$, $T(8) = 2 \cdot 28 + 64 = 120$.

2. Expansion.

(a) Expanding,

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

With $k = n-1$ and using the arithmetic series $\sum_{i=1}^n i = \frac{n(n+1)}{2}$:

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

(b) Expanding,

$$T(n) = 3T(n-1) = 3^2 T(n-2) = 3^3 T(n-3) = \dots = 3^k T(n-k).$$

With $k = n-1$: $T(n) = 3^{n-1} T(1) = 2 \cdot 3^{n-1}$.

(c) Let $n = 3^k$. Expanding,

$$T(n) = T\left(\frac{n}{3}\right) + 1 = T\left(\frac{n}{3^2}\right) + 2 = \dots = T\left(\frac{n}{3^j}\right) + j.$$

The boundary condition is reached when $\frac{n}{3^j} = 1$, that is, $j = \log_3 n$:

$$T(n) = T(1) + \log_3 n = \log_3 n + 1.$$

(d) Expanding,

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n^2 \\
 &= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n^2}{4}\right) + n^2 = 2^2 T\left(\frac{n}{2^2}\right) + n^2 \left(1 + \frac{1}{2}\right) \\
 &= 2^2 \left(2T\left(\frac{n}{2^3}\right) + \frac{n^2}{16}\right) + n^2 \left(1 + \frac{1}{2}\right) = 2^3 T\left(\frac{n}{2^3}\right) + n^2 \left(1 + \frac{1}{2} + \frac{1}{4}\right) \\
 &\vdots \\
 &= 2^j T\left(\frac{n}{2^j}\right) + n^2 \sum_{i=0}^{j-1} \left(\frac{1}{2}\right)^i.
 \end{aligned}$$

With $j = \log_2 n$ (so $2^j = n$) and the geometric series $\sum_{i=0}^{j-1} x^i = \frac{1-x^j}{1-x}$:

$$T(n) = nT(1) + n^2 \frac{1 - \left(\frac{1}{2}\right)^{\log_2 n}}{1 - \frac{1}{2}} = n + 2n^2 \left(1 - \frac{1}{n}\right) = 2n^2 - n.$$

3. Check:

- (a) n^2 : 1, 4, 9, 16. ✓
- (b) $2 \cdot 3^{n-1}$: 2, 6, 18, 54. ✓
- (c) $\log_3 n + 1$ for $n = 1, 3, 9, 27$: 1, 2, 3, 4. ✓
- (d) $2n^2 - n$ for $n = 1, 2, 4, 8$: 1, $8 - 2 = 6$, $32 - 4 = 28$, $128 - 8 = 120$. ✓

4. Orders of growth:

- (a) $T(n) = n^2 = \Theta(n^2)$.
- (b) $T(n) = 2 \cdot 3^{n-1} = \frac{2}{3} 3^n = \Theta(3^n)$ (exponential).
- (c) $T(n) = \log_3 n + 1 = \Theta(\log_2 n)$ (the base of the logarithm is irrelevant inside Θ).
- (d) $T(n) = 2n^2 - n = \Theta(n^2)$, since $\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0}$. Note that the cost is dominated by the root (n^2).

Exercise 15.2 — Recurrences (E2)

Consider the following recurrences, defined for every $n \in \mathbb{N}$, $n \geq 1$.

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

Questions

1. Using the substitution method, prove that $T(n) \leq n^2$ for all $n \geq 1$.
2. Using the substitution method, prove that $T(n) \geq \frac{1}{2}n^2$ for all $n \geq 1$. Conclude that $T(n) = \Theta(n^2)$.
3. Compute $U(n)$ for $n = 1, 2, \dots, 8$.
4. Prove that $U(n) = O(\log_2 n)$, by showing that $U(n) \leq c \log_2 n$ for a suitable constant c and all $n \geq 2$. Explain why the bound cannot hold for $n = 1$ and which base cases are needed.
5. Prove that $U(n) \geq \log_2(n + 1)$ for all $n \geq 1$. Conclude that $U(n) = \Theta(\log_2 n)$.

Solution

1. We guess $T(n) \leq cn^2$ and look for c . Assume (inductive hypothesis) $T(n - 1) \leq c(n - 1)^2$. Then

$$T(n) \leq c(n - 1)^2 + n = cn^2 - 2cn + c + n = cn^2 - ((2c - 1)n - c).$$

We need $(2c - 1)n - c \geq 0$. With $c = 1$ this is $n - 1 \geq 0$, true for every $n \geq 1$.

Base case: $T(1) = 1 \leq 1^2$. ✓

Inductive step ($n \geq 2$): $T(n) \leq (n - 1)^2 + n = n^2 - n + 1 \leq n^2$. ✓

Hence $T(n) \leq n^2$ for all $n \geq 1$, that is, $T(n) = O(n^2)$.

2. We guess $T(n) \geq dn^2$. Assume $T(n - 1) \geq d(n - 1)^2$. Then

$$T(n) \geq d(n - 1)^2 + n = dn^2 + ((1 - 2d)n + d).$$

We need $(1 - 2d)n + d \geq 0$: with $d = \frac{1}{2}$ it is $\frac{1}{2} \geq 0$. ✓

Base case: $T(1) = 1 \geq \frac{1}{2}$. ✓

Hence $\frac{1}{2}n^2 \leq T(n) \leq n^2$ for all $n \geq 1$, and $T(n) = \Theta(n^2)$. (Indeed, the exact solution is $T(n) = \frac{n(n+1)}{2}$.)

3.

n	1	2	3	4	5	6	7	8
$U(n)$	1	2	2	3	3	3	3	4

For example, $U(6) = U(3) + 1 = U(1) + 2 = 3$ and $U(8) = U(4) + 1 = U(2) + 2 = U(1) + 3 = 4$. (In general $U(n) = \lfloor \log_2 n \rfloor + 1$, the number of bits of n .)

4. We guess $U(n) \leq c \log_2 n$.

Inductive step. Assume $U(\lfloor n/2 \rfloor) \leq c \log_2 \lfloor n/2 \rfloor$. Then

$$U(n) \leq c \log_2 \lfloor n/2 \rfloor + 1 \leq c \log_2 \frac{n}{2} + 1 = c \log_2 n - c + 1 \leq c \log_2 n,$$

as long as $c \geq 1$.

Base cases. For $n = 1$: $U(1) = 1 > c \log_2 1 = 0$ for every c , so the bound cannot hold for $n = 1$. 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 uses only values ≥ 2 . For $n = 2, 3$ we have $\lfloor n/2 \rfloor = 1$: these are the base cases. We need

$$U(2) = 2 \leq c \log_2 2 = c, \quad U(3) = 2 \leq c \log_2 3 \approx 1.58 c.$$

Both hold with $c = 2$.

Hence $U(n) \leq 2 \log_2 n$ for all $n \geq 2$, and $U(n) = O(\log_2 n)$.

5. By strong induction on n .

Base case: $U(1) = 1 = \log_2 2$. ✓

Inductive step ($n \geq 2$): since $1 \leq \lfloor n/2 \rfloor < n$, the inductive hypothesis gives

$$U(\lfloor n/2 \rfloor) \geq \log_2 (\lfloor n/2 \rfloor + 1).$$

Then

$$\begin{aligned} U(n) &\geq \log_2 (\lfloor n/2 \rfloor + 1) + 1 = \log_2 (2\lfloor n/2 \rfloor + 2) \\ &\geq \log_2 ((n-1) + 2) = \log_2 (n+1), \end{aligned}$$

where we used $2\lfloor n/2 \rfloor \geq n-1$.

Note that the natural guess $U(n) \geq \log_2 n$ does **not** work directly: for odd n the inductive step only gives

$$U(n) \geq \log_2 \lfloor n/2 \rfloor + 1 = \log_2 (2\lfloor n/2 \rfloor) = \log_2 (n-1).$$

The slightly stronger statement $U(n) \geq \log_2 (n+1)$ makes the induction work.

Since $\log_2 (n+1) \geq \log_2 n$, we have $U(n) = \Omega(\log_2 n)$. Together with question 4,

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

Exercise 15.3 — Recurrences (E3)

Consider the recurrence

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

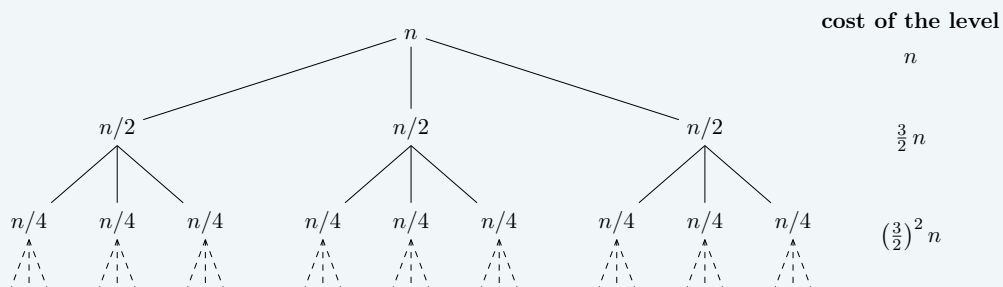
where n is an exact power of 2.

Questions

1. Draw the first three levels of the recursion tree (depths 0, 1, 2) and indicate the cost of each node.
2. Determine, for a generic depth i , the number of nodes, the cost of each node and the cost of the level. Determine the depth of the tree and the number of leaves.
3. Sum the costs of all levels and obtain a closed form of $T(n)$.
4. Determine the order of growth of $T(n)$. Which part of the tree dominates the total cost?
5. Verify the closed form obtained in question 3 by induction (substitution method), and check it numerically for $n = 1, 2, 4, 8$.

Solution

1. The root has cost n and three children, each with cost $\frac{n}{2}$; each of them has three children with cost $\frac{n}{4}$.



2. At depth i :

- number of nodes: 3^i ;
- size of each subproblem: $\frac{n}{2^i}$, hence cost of each node: $\frac{n}{2^i}$;
- cost of the level: $3^i \frac{n}{2^i} = \left(\frac{3}{2}\right)^i n$.

The subproblem size reaches 1 when $\frac{n}{2^i} = 1$, that is, at depth $k = \log_2 n$. The number of leaves is

$$3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585},$$

each with cost $T(1) = 1$.

3. Let $k = \log_2 n$. Using the geometric series $\sum_{i=0}^{k-1} x^i = \frac{x^k - 1}{x - 1}$ with $x = \frac{3}{2}$:

$$\sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i n = n \frac{\left(\frac{3}{2}\right)^k - 1}{\frac{3}{2} - 1} = 2n \left(\frac{3^k}{2^k} - 1\right) = 2n \left(\frac{3^k}{n} - 1\right) = 2 \cdot 3^k - 2n.$$

Adding the cost of the leaves, $3^k \cdot 1$:

$$T(n) = 2 \cdot 3^k - 2n + 3^k = 3 \cdot 3^{\log_2 n} - 2n = 3n^{\log_2 3} - 2n.$$

4. Since $\log_2 3 > 1$, the term $n^{\log_2 3}$ dominates n :

$$\lim_{n \rightarrow \infty} \frac{3n^{\log_2 3} - 2n}{n^{\log_2 3}} = \lim_{n \rightarrow \infty} \left(3 - \frac{2}{n^{\log_2 3 - 1}} \right) = 3 \in \mathbb{R}_{>0}.$$

Hence $T(n) = \Theta(n^{\log_2 3})$. The costs of the levels increase geometrically (ratio $\frac{3}{2} > 1$): the total cost is dominated by the **leaves**.

5. We prove $T(n) = 3n^{\log_2 3} - 2n$ for every $n = 2^k$, by induction on k .

Base case $k = 0$ ($n = 1$): $3 \cdot 1 - 2 = 1 = T(1)$. ✓

Inductive step: assume the formula for $\frac{n}{2}$. Since $\left(\frac{n}{2}\right)^{\log_2 3} = \frac{n^{\log_2 3}}{2^{\log_2 3}} = \frac{n^{\log_2 3}}{3}$,

$$T(n) = 3 \left(3 \frac{n^{\log_2 3}}{3} - 2 \frac{n}{2} \right) + n = 3n^{\log_2 3} - 3n + n = 3n^{\log_2 3} - 2n. \quad \checkmark$$

Numerical check ($n^{\log_2 3} = 3^k$):

n	$T(n)$ from the recurrence	$3 \cdot 3^k - 2n$
1	1	$3 - 2 = 1$
2	$3 \cdot 1 + 2 = 5$	$9 - 4 = 5$
4	$3 \cdot 5 + 4 = 19$	$27 - 8 = 19$
8	$3 \cdot 19 + 8 = 65$	$81 - 16 = 65$

Exercise 15.4 — Recurrences (E4)

Consider the following recurrences.

$$(A) \quad T(n) = \begin{cases} 1 & \text{if } 1 \leq n < 4 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n & \text{if } n \geq 4 \end{cases} \quad (\text{defined for every real } n \geq 1)$$

$$(B) \quad U(n) = \begin{cases} 0 & \text{if } n = 0 \\ U(n-2) + n^2 & \text{if } n \geq 2 \end{cases} \quad (\text{defined for every even } n \in \mathbb{N})$$

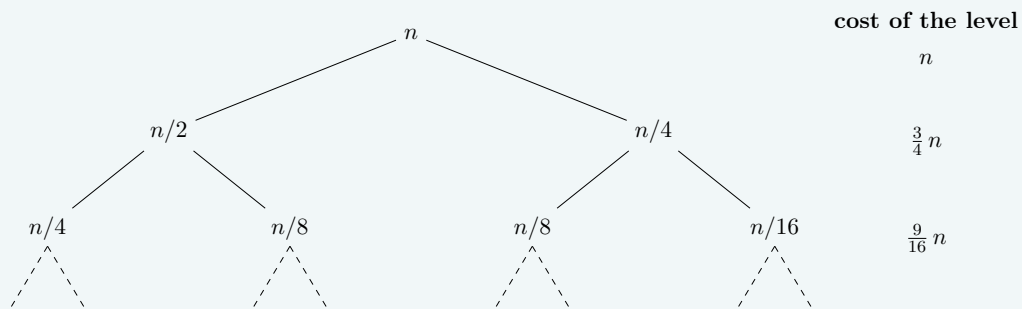
Questions

1. Draw the first three levels of the recursion tree of (A) and compute the cost of each of these levels.

2. Show that the cost of the level at depth i of the tree of (A) is at most $(\frac{3}{4})^i n$, and use this fact to guess an upper bound on $T(n)$.
3. Prove, with the substitution method, that $T(n) \leq 4n$ for all $n \geq 1$. Conclude that $T(n) = \Theta(n)$.
4. Draw the recursion tree of (B). What is its shape? What is its depth?
5. Determine a closed form of $U(n)$ and its order of growth. (Recall that $\sum_{j=1}^m j^2 = \frac{m(m+1)(2m+1)}{6}$.)

Solution

1. Each node of size $s \geq 4$ has two children, of sizes $\frac{s}{2}$ and $\frac{s}{4}$.



The costs of the first three levels (for n large enough, so that all these nodes are internal) are:

$$n, \quad \frac{n}{2} + \frac{n}{4} = \frac{3}{4}n, \quad \frac{n}{4} + \frac{n}{8} + \frac{n}{8} + \frac{n}{16} = \frac{9}{16}n.$$

2. If a node has size s , its children have total size $\frac{s}{2} + \frac{s}{4} = \frac{3}{4}s$. Hence, by induction on i , the nodes at depth i have total size at most $(\frac{3}{4})^i n$ (exactly, if the level is complete), and the internal nodes at depth i have total cost at most $(\frac{3}{4})^i n$.

Summing over all levels and using the infinite decreasing geometric series $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$:

$$\sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i n = \frac{1}{1-\frac{3}{4}} n = 4n.$$

Since the leaves have constant cost, we **guess** $T(n) = O(n)$, more precisely $T(n) \leq 4n$. (The leaves are not easy to count, so we prove the guess with the substitution method.)

3. We prove $T(n) \leq 4n$ for every real $n \geq 1$, by induction on $k = \lfloor \log_2 n \rfloor$.

Base cases $k \in \{0, 1\}$, that is, $1 \leq n < 4$: $T(n) = 1 \leq 4 \leq 4n$. ✓

Inductive step $k \geq 2$, that is, $n \geq 4$: the arguments $\frac{n}{2} \geq 2$ and $\frac{n}{4} \geq 1$ are in the domain and have $\lfloor \log_2(\cdot) \rfloor < k$, so by the inductive hypothesis

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n \leq 4 \frac{n}{2} + 4 \frac{n}{4} + n = 2n + n + n = 4n. \quad \checkmark$$

Hence $T(n) = O(n)$.

Lower bound: for $n \geq 4$, all the terms are non-negative and the root alone costs n , so $T(n) \geq n$. Hence $T(n) = \Omega(n)$ and

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

(The cost is dominated by the root: the level costs decrease geometrically with ratio $\frac{3}{4}$.)

4. Each call generates only *one* recursive call, so the recursion tree is a **path**:

$$n^2 \longrightarrow (n-2)^2 \longrightarrow (n-4)^2 \longrightarrow \cdots \longrightarrow 2^2 \longrightarrow U(0) = 0.$$

The node at depth i has size $n - 2i$ and cost $(n - 2i)^2$. The size reaches 0 at depth $\frac{n}{2}$, which is the depth of the tree.

5. Summing the costs of all levels, with $m = \frac{n}{2}$ and $j = m - i$ (so that $n - 2i = 2j$):

$$U(n) = \sum_{i=0}^{m-1} (n - 2i)^2 = \sum_{j=1}^m (2j)^2 = 4 \sum_{j=1}^m j^2 = 4 \frac{m(m+1)(2m+1)}{6}.$$

Substituting $m = \frac{n}{2}$:

$$U(n) = 4 \frac{\frac{n}{2} \cdot \frac{n+2}{2} \cdot (n+1)}{6} = \frac{n(n+1)(n+2)}{6}.$$

Check: $U(2) = 4$ and $\frac{2 \cdot 3 \cdot 4}{6} = 4$; $U(4) = 4 + 16 = 20$ and $\frac{4 \cdot 5 \cdot 6}{6} = 20$; $U(6) = 20 + 36 = 56$ and $\frac{6 \cdot 7 \cdot 8}{6} = 56$. ✓

Order of growth:

$$\lim_{n \rightarrow \infty} \frac{n(n+1)(n+2)/6}{n^3} = \frac{1}{6} \in \mathbb{R}_{>0} \quad \implies \quad U(n) = \Theta(n^3).$$

Note that, unlike in (A), the cost of the levels decreases only slowly (polynomially, not geometrically) and the depth is linear in n : no level dominates, and the total is $\Theta(n)$ levels times $\Theta(n^2)$ per level (at least half of the levels cost at least $\frac{n^2}{4}$).

Exercise 15.5 — Recurrences (E5)

Consider the following two recursive algorithms. Assume that every multiplication is a primitive operation (one step), and that x is stored with a constant number of bits.

Algorithm: FastPower

Input : An integer number x and a natural number n

Output: An integer number

```

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

```

Algorithm: SlowPower

Input : An integer number x and a natural number n

Output: An integer number

```

1 function SlowPower( $x, n$ )
2   if  $n = 0$  then
3     return 1
4   end
5   if  $n$  is even then
6     return  $\text{SlowPower}(x, n/2) \cdot \text{SlowPower}(x, n/2)$ 
7   else
8     return  $x \cdot \text{SlowPower}(x, \lfloor n/2 \rfloor) \cdot \text{SlowPower}(x, \lfloor n/2 \rfloor)$ 
9   end
10 end

```

Questions

1. What do the two algorithms compute? Justify your answer.
2. Let $S_F(n)$ be the number of steps executed by **FastPower**. Write a recurrence for $S_F(n)$, counting the executions of each line, and compute $S_F(n)$ for $n = 0, 1, \dots, 8$.
3. Solve the recurrence for $S_F(n)$ and determine its order of growth.

4. Let $S_S(n)$ be the number of steps executed by **SlowPower**. Write a recurrence for $S_S(n)$ and solve it with the expansion method when n is an exact power of 2. Determine its order of growth.
5. Determine the worst-case running time of the two algorithms as a function of the input size $|I|$, and compare them.

Solution

1. Both algorithms compute x^n . Indeed, writing $n = 2\lfloor n/2 \rfloor + r$ with $r \in \{0, 1\}$,

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

and $x^0 = 1$. By strong induction on n , if the recursive calls return $x^{\lfloor n/2 \rfloor}$, both algorithms return x^n . The only difference is that **FastPower** computes $x^{\lfloor n/2 \rfloor}$ *once* and stores it in y , while **SlowPower** computes it *twice*.

FastPower is the function `powerRec` of the code of the course (in C++, `n / 2` is the integer division $\lfloor n/2 \rfloor$ for $n \geq 0$):

```

1 long long powerRec(long long x, int n)
2 {
3     if (n == 0)
4     {
5         return 1;
6     }
7
8     long long h = powerRec(x, n / 2);
9
10    if (n % 2 == 0)
11    {
12        return h * h;
13    }
14
15    return h * h * x;
16 }
```

For $x = 3$ and $n = 0, 1, \dots, 10$ it returns 1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, that is, 3^n .

2. **FastPower**.

- If $n = 0$: lines 2 and 3 are executed once: $S_F(0) = 2$.
- If $n \geq 1$: lines 2, 5 and 6 are executed once, and exactly one of lines 7 and 9 is executed once: 4 steps. In addition, the recursive call at line 5 executes $S_F(\lfloor n/2 \rfloor)$ steps.

Therefore

$$S_F(n) = \begin{cases} 2 & \text{if } n = 0 \\ S_F(\lfloor n/2 \rfloor) + 4 & \text{if } n \geq 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8
$S_F(n)$	2	6	10	10	14	14	14	14	18

3. Expanding,

$$S_F(n) = S_F(\lfloor n/2 \rfloor) + 4 = S_F(\lfloor n/4 \rfloor) + 8 = \dots = S_F(\lfloor n/2^j \rfloor) + 4j.$$

For $n \geq 1$, $\lfloor n/2^j \rfloor = 0$ for the first time when $2^j > n$, that is, for $j = \lfloor \log_2 n \rfloor + 1$ (the number of bits of n). Hence

$$S_F(n) = 4(\lfloor \log_2 n \rfloor + 1) + 2 \quad (n \geq 1).$$

Check: $S_F(1) = 4 + 2 = 6$, $S_F(3) = 8 + 2 = 10$, $S_F(8) = 16 + 2 = 18$. ✓

Since $\log_2 n - 1 < \lfloor \log_2 n \rfloor \leq \log_2 n$, we have $4 \log_2 n + 2 < S_F(n) \leq 4 \log_2 n + 6$, and

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

4. SlowPower.

- If $n = 0$: **lines** 2 and 3: $S_S(0) = 2$.
- If $n \geq 1$: **lines** 2, 5 and one of 6, 8: 3 steps, plus *two* recursive calls on $\lfloor n/2 \rfloor$.

Therefore

$$S_S(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 S_S(\lfloor n/2 \rfloor) + 3 & \text{if } n \geq 1 \end{cases}$$

so that $S_S(1) = 2 \cdot 2 + 3 = 7$. Let $n = 2^k$. Expanding down to $S_S(1)$:

$$\begin{aligned} S_S(n) &= 2 S_S\left(\frac{n}{2}\right) + 3 \\ &= 2^2 S_S\left(\frac{n}{2^2}\right) + 3(2 + 1) \\ &= 2^3 S_S\left(\frac{n}{2^3}\right) + 3(2^2 + 2 + 1) \\ &\vdots \\ &= 2^k S_S(1) + 3 \sum_{i=0}^{k-1} 2^i = 7 \cdot 2^k + 3(2^k - 1) = 10n - 3. \end{aligned}$$

Check: $S_S(1) = 7$, $S_S(2) = 2 \cdot 7 + 3 = 17 = 20 - 3$, $S_S(4) = 2 \cdot 17 + 3 = 37 = 40 - 3$. ✓

Hence $S_S(n) = \Theta(n)$ for n power of 2. (For a generic $n \geq 1$, $S_S(n)$ depends only on the number of bits of n , and $S_S(n) = 10 \cdot 2^{\lfloor \log_2 n \rfloor} - 3$, which is between $5n - 3$ and $10n - 3$: still $\Theta(n)$.)

5. The input consists of x (a constant number of bits) and of the natural number n , so

$$|I| = \Theta(\log_2 n), \quad \text{that is,} \quad n = 2^{\Theta(|I|)}.$$

- **FastPower**: $S_F(n) = \Theta(\log_2 n) = \Theta(|I|)$.
- **SlowPower**: $S_S(n) = \Theta(n) = 2^{\Theta(|I|)}$.

The worst-case running time of **FastPower** is $T(|I|) = \Theta(|I|)$ (linear in the input size), while the worst-case running time of **SlowPower** is $T(|I|) = 2^{\Theta(|I|)}$ (exponential in the input size).

Computing the same sub-result twice changes the recurrence from $S(n) = S(n/2) + c$ to $S(n) = 2S(n/2) + c$, and the running time from logarithmic to linear in n .

Exercise 15.6 — Recurrences (E6)

Consider the following recurrences, where n is an exact power of 2.

$$(A) \quad T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 4T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{cases} \quad (B) \quad U(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8U\left(\frac{n}{2}\right) + n^2 & \text{if } n > 1 \end{cases}$$

Questions

1. Try to prove $T(n) \leq cn^2$ with the substitution method. Show that the induction fails for every choice of the constant c .
2. Strengthen the guess by subtracting a lower-order term: prove that $T(n) \leq cn^2 - dn$ for suitable constants $c, d > 0$, handling the base case.
3. Prove that $T(n) \geq n^2$ and conclude that $T(n) = \Theta(n^2)$.
4. Repeat questions 1–3 for $U(n)$, with the guess $U(n) \leq cn^3$.
5. Using the recursion-tree method, compute the exact value of $U(n)$ and compare it with the bound obtained in question 4.

Solution

1. Assume $T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^2$. Then

$$T(n) \leq 4c \frac{n^2}{4} + n = cn^2 + n.$$

Since $n > 0$, $cn^2 + n > cn^2$ for every c : we cannot conclude $T(n) \leq cn^2$. The

induction fails, by the lower-order term n . (Note: concluding “ $T(n) \leq cn^2 + n = O(n^2)$ ” would be **wrong**: the induction must prove exactly the inductive hypothesis, with the same constant c .)

2. Guess $T(n) \leq cn^2 - dn$. Assume $T\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^2 - d\frac{n}{2}$. Then

$$T(n) \leq 4\left(c\frac{n^2}{4} - d\frac{n}{2}\right) + n = cn^2 - 2dn + n = cn^2 - dn - (d-1)n \leq cn^2 - dn,$$

as long as $d \geq 1$.

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

Choosing $c = 2$ and $d = 1$: $T(n) \leq 2n^2 - n$ for every n power of 2. Hence $T(n) = O(n^2)$. (In fact, $T(n) = 2n^2 - n$ exactly, as can be verified with the recursion-tree method.)

3. Assume $T\left(\frac{n}{2}\right) \geq \left(\frac{n}{2}\right)^2$. Then

$$T(n) \geq 4\frac{n^2}{4} + n = n^2 + n \geq n^2.$$

Base case: $T(1) = 1 \geq 1$. Hence $n^2 \leq T(n) \leq 2n^2 - n$, and $T(n) = \Theta(n^2)$.

4. *Naive guess.* Assume $U\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^3$. Then

$$U(n) \leq 8c\frac{n^3}{8} + n^2 = cn^3 + n^2,$$

and the induction fails (by the lower-order term n^2).

Strengthened guess. Assume $U\left(\frac{n}{2}\right) \leq c\left(\frac{n}{2}\right)^3 - d\left(\frac{n}{2}\right)^2$. Then

$$U(n) \leq 8\left(c\frac{n^3}{8} - d\frac{n^2}{4}\right) + n^2 = cn^3 - 2dn^2 + n^2 = cn^3 - dn^2 - (d-1)n^2 \leq cn^3 - dn^2,$$

as long as $d \geq 1$. *Base case:* $U(1) = 1 \leq c - d$, satisfied by $c = 2, d = 1$. Hence

$$U(n) \leq 2n^3 - n^2.$$

Lower bound. Assume $U\left(\frac{n}{2}\right) \geq \left(\frac{n}{2}\right)^3$. Then $U(n) \geq 8\frac{n^3}{8} + n^2 \geq n^3$; *base case* $U(1) = 1 \geq 1$. Hence $n^3 \leq U(n) \leq 2n^3 - n^2$, and $U(n) = \Theta(n^3)$.

5. At depth i the recursion tree of (B) has 8^i nodes, each of size $\frac{n}{2^i}$ and cost $\left(\frac{n}{2^i}\right)^2$. The cost of the level is

$$8^i \frac{n^2}{4^i} = 2^i n^2.$$

The depth of the tree is $\log_2 n$, and there are $8^{\log_2 n} = n^{\log_2 8} = n^3$ leaves, each with cost $U(1) = 1$. Using $\sum_{i=0}^{k-1} 2^i = 2^k - 1$:

$$U(n) = \sum_{i=0}^{\log_2 n - 1} 2^i n^2 + n^3 = n^2 (2^{\log_2 n} - 1) + n^3 = n^2(n - 1) + n^3 = 2n^3 - n^2.$$

The bound of question 4 is therefore *exact*: the strengthened guess $cn^3 - dn^2$ with $c = 2, d = 1$ is precisely the solution of the recurrence.

Numerical check: $U(2) = 8 + 4 = 12 = 16 - 4$, $U(4) = 8 \cdot 12 + 16 = 112 = 128 - 16$, $U(8) = 8 \cdot 112 + 64 = 960 = 1024 - 64$. ✓

The Master Theorem

In the exercises of this chapter recurrences in standard form $T(n) = aT(n/b) + f(n)$, with constants $a \geq 1$, $b > 1$ and $f(n)$ asymptotically positive, are solved with the Master Theorem of the chapter «The Master Theorem», following the five-step scheme (identify, compute $n^{\log_b a}$, compare, verify, conclude). The following formulas are used.

Warning

The three cases of the Master Theorem:

- **Case 1:** if $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- **Case 2:** if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$, 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 $\epsilon > 0$ and $a f(n/b) \leq c f(n)$ for some $c < 1$ and all large n (regularity condition), then $T(n) = \Theta(f(n))$.

Cost of the recursion tree, for n exact power of b :

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

Harmonic numbers: $H_L = \sum_{i=1}^L \frac{1}{i}$ satisfies $\ln(L+1) \leq H_L \leq 1 + \ln L$; moreover $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$.

Exercise 16.1 — The Master Theorem (E1)

Consider the following recurrences ($\log$ denotes $\log_2$):

- (a) $T(n) = 9T(n/3) + n$;
- (b) $T(n) = 2T(n/4) + \sqrt{n}$;
- (c) $T(n) = 2T(n/2) + n^2$;
- (d) $T(n) = 4T(n/2) + n^2 \log n$;

(e) $T(n) = 6T(n/3) + n^2 \log n$.

Questions

1. For each recurrence, identify a , b , $f(n)$ and compute $n^{\log_b a}$.
2. For each recurrence, determine which case of the Master Theorem applies, verify all its hypotheses (ϵ , k , regularity condition) and give the order of growth of $T(n)$.
3. For recurrence (c), compute $T(n)$ exactly for $n = 2^L$ assuming $T(1) = 1$, and check the result of the previous question.

Solution

	a	b	$f(n)$	$n^{\log_b a}$	
1.	(a)	9	3	n	$n^{\log_3 9} = n^2$
	(b)	2	4	$\sqrt{n}$	$n^{\log_4 2} = n^{1/2} = \sqrt{n}$
	(c)	2	2	n^2	$n^{\log_2 2} = n$
	(d)	4	2	$n^2 \log n$	$n^{\log_2 4} = n^2$
	(e)	6	3	$n^2 \log n$	$n^{\log_3 6} \approx n^{1.631}$

2. We follow the scheme: identify $\rightarrow$ compute $n^{\log_b a} \rightarrow$ compare $\rightarrow$ verify $\rightarrow$ conclude.

- (a) n^2 is polynomially larger than $f(n) = n \rightarrow$ **case 1**. Indeed $f(n) = n = O(n^{2-\epsilon})$ with $\epsilon = 1$. Hence

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

- (b) $n^{1/2}$ and $f(n) = \sqrt{n}$ have the same order $\rightarrow$ **case 2**: $f(n) = \Theta(n^{1/2} \log^k n)$ with $k = 0$. Hence

$$T(n) = \Theta(\sqrt{n} \log n).$$

- (c) n is polynomially smaller than $f(n) = n^2 \rightarrow$ **case 3**. $f(n) = n^2 = \Omega(n^{1+\epsilon})$ with $\epsilon = 1$. Regularity condition:

$$2f\left(\frac{n}{2}\right) = 2 \cdot \frac{n^2}{4} = \frac{1}{2} n^2 = c f(n) \quad \text{with } c = \frac{1}{2} < 1,$$

for every n . Hence

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

- (d) $f(n) = n^2 \log n$ is larger than n^2 only by a factor $\log n$ (not polynomially) and it has the form $n^{\log_b a} \log^k n \rightarrow$ **case 2** with $k = 1$. Hence

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

(e) $n^{1.631}$ is polynomially smaller than $f(n) = n^2 \log n \rightarrow$ **case 3**. Since $n^2 \log n \geq n^2$ for $n \geq 2$, $f(n) = \Omega(n^{\log_3 6 + \epsilon})$ with $\epsilon = 2 - \log_3 6 \approx 0.369 > 0$. Regularity condition: for $n \geq 3$, since $\log(n/3) \leq \log n$,

$$6f\left(\frac{n}{3}\right) = 6 \cdot \frac{n^2}{9} \log \frac{n}{3} \leq \frac{2}{3} n^2 \log n = c f(n) \quad \text{with } c = \frac{2}{3} < 1.$$

Hence

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

3. By the recursion tree (Lemma of the cost of the recursion tree), with $n = 2^L$, $a = 2$, $T(1) = 1$:

$$\begin{aligned} T(n) &= 2^L \cdot 1 + \sum_{j=0}^{L-1} 2^j \left(\frac{n}{2^j}\right)^2 = n + n^2 \sum_{j=0}^{L-1} \left(\frac{1}{2}\right)^j \\ &= n + n^2 \frac{1 - (1/2)^L}{1 - 1/2} = n + 2n^2 \left(1 - \frac{1}{n}\right) = 2n^2 - n. \end{aligned}$$

Check with the recurrence: $T(2) = 2 + 4 = 6 = 2 \cdot 4 - 2$, $T(4) = 12 + 16 = 28 = 2 \cdot 16 - 4$, $T(8) = 56 + 64 = 120 = 2 \cdot 64 - 8$. Since

$$\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0},$$

we have $T(n) = \Theta(n^2)$, as found with case 3. Note that the level costs $n^2(1/2)^j$ decrease geometrically: the root dominates.

Exercise 16.2 — The Master Theorem (E2)

Consider the following recurrences ($\log$ denotes $\log_2$):

- (a) $T(n) = 4T(n/2) + n^2 / \log n$;
- (b) $T(n) = nT(n/2) + n$;
- (c) $T(n) = 3T(n/3) + n/2$;
- (d) $T(n) = T(n-2) + \log n$;
- (e) $T(n) = 64T(n/8) - n^2 \log n$;
- (f) $T(n) = \sqrt{2}T(n/2) + \log n$.

Questions

1. For each recurrence, say whether the Master Theorem can be applied. If not, explain precisely which hypothesis fails.
2. For the recurrences to which the theorem applies, give the order of growth of $T(n)$, following the usual scheme.
3. For recurrence (a), determine the order of growth of $T(n)$ for $n = 2^L$ and $T(1) = 1$ using the recursion tree.

Solution

1. (a) **Not applicable.** $a = 4$, $b = 2$, $n^{\log_2 4} = n^2$, $f(n) = n^2 / \log n$. $f(n)$ is smaller than n^2 but not polynomially smaller: for every $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n^2 / \log n}{n^{2-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log n} = +\infty,$$

so case 1 fails; case 2 would need $k = -1 < 0$; case 3 fails since $f(n)$ is smaller than n^2 . This is a **gap** between case 1 and case 2.

- (b) **Not applicable.** The number of subproblems $a = n$ is not a constant.
 - (c) **Applicable:** $a = 3$, $b = 3$, $f(n) = n/2$ (see question 2).
 - (d) **Not applicable.** The subproblem has size $n - 2$, not n/b : the recurrence is not in standard form.
 - (e) **Not applicable.** $f(n) = -n^2 \log n < 0$ for every $n \geq 2$: f is not asymptotically positive.
 - (f) **Applicable:** $a = \sqrt{2} \geq 1$, $b = 2$, $f(n) = \log n$ (see question 2). Note that a does not have to be an integer.
2. (c) $n^{\log_b a} = n^{\log_3 3} = n$. $f(n) = n/2$ has the same order as $n \rightarrow$ **case 2**: $f(n) = \Theta(n \log^k n)$ with $k = 0$ (constant factors do not matter). Hence

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

- (f) $n^{\log_b a} = n^{\log_2 \sqrt{2}} = n^{1/2}$. $f(n) = \log n$ is polynomially smaller than $n^{1/2} \rightarrow$ **case 1**. Indeed, with $\epsilon = \frac{1}{4}$,

$$\lim_{n \rightarrow \infty} \frac{\log n}{n^{1/2-1/4}} = \lim_{n \rightarrow \infty} \frac{\log n}{n^{1/4}} = 0^+,$$

so $f(n) = O(n^{1/2-\epsilon})$. Hence

$$T(n) = \Theta(\sqrt{n}).$$

Even if the Master Theorem does not apply to (d), it is easy to solve: $T(n)$ is the sum of about $n/2$ terms $\log n, \log(n-2), \dots$, each at most $\log n$, and the first $n/4$ of them are at least $\log(n/2)$; hence $T(n) = \Theta(n \log n)$.

3. Recursion tree with $n = 2^L$: level j has 4^j nodes of size $n/2^j$, and $\log(n/2^j) = L - j$. The cost of level j is

$$4^j \frac{(n/2^j)^2}{L - j} = \frac{n^2}{L - j}.$$

The leaves are $4^L = n^2$, each with cost $T(1) = 1$. Hence

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

where H_L is the L -th harmonic number. Comparing the sum with the integral of $1/x$,

$$\ln(L + 1) = \int_1^{L+1} \frac{dx}{x} \leq H_L \leq 1 + \int_1^L \frac{dx}{x} = 1 + \ln L,$$

so $H_L = \Theta(\log L) = \Theta(\log \log n)$, and

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

For example, $T(16) = 256(1 + H_4) = 256(1 + \frac{25}{12}) = \frac{2368}{3} \approx 789.33$, which agrees with the recurrence: $T(2) = 8$, $T(4) = 40$, $T(8) = \frac{544}{3}$, $T(16) = 4 \cdot \frac{544}{3} + \frac{256}{4} = \frac{2368}{3}$.

Exercise 16.3 — The Master Theorem (E3)

Consider the following families of recurrences, depending on a real parameter (log denotes $\log_2$):

- (A) $T(n) = aT(n/4) + n^2$, with $a \geq 1$;
- (B) $T(n) = 8T(n/2) + n^d$, with $d \geq 0$;
- (C) $T(n) = 4T(n/b) + n \log n$, with $b > 1$.

Questions

1. For family (A), determine for which values of a each case of the Master Theorem applies, and give the corresponding order of growth of $T(n)$. Verify the regularity condition when needed.
2. Same question for family (B), as a function of d .
3. Same question for family (C), as a function of b .
4. In family (A), what is the largest integer a such that $T(n) = O(n^2)$? What happens for $a = 17$?

Solution

1. Family (A): $b = 4$, $f(n) = n^2$, $n^{\log_b a} = n^{\log_4 a}$. We compare the exponents $\log_4 a$ and 2; note that $\log_4 a = 2 \iff a = 16$.

- $a > 16$: $\log_4 a > 2 \rightarrow$ **case 1**, since $f(n) = n^2 = O(n^{\log_4 a - \epsilon})$ with $\epsilon = \log_4 a - 2 > 0$. Hence $T(n) = \Theta(n^{\log_4 a})$.
- $a = 16$: $n^{\log_4 16} = n^2 \rightarrow$ **case 2** with $k = 0$. Hence $T(n) = \Theta(n^2 \log n)$.
- $1 \leq a < 16$: $\log_4 a < 2 \rightarrow$ **case 3**, with $f(n) = \Omega(n^{\log_4 a + \epsilon})$, $\epsilon = 2 - \log_4 a > 0$. Regularity condition:

$$a f\left(\frac{n}{4}\right) = a \frac{n^2}{16} = \frac{a}{16} n^2 = c f(n) \quad \text{with } c = \frac{a}{16} < 1.$$

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

2. Family (B): $a = 8$, $b = 2$, $n^{\log_2 8} = n^3$, $f(n) = n^d$.

- $0 \leq d < 3$: **case 1** with $\epsilon = 3 - d > 0$. Hence $T(n) = \Theta(n^3)$.
- $d = 3$: **case 2** with $k = 0$. Hence $T(n) = \Theta(n^3 \log n)$.
- $d > 3$: **case 3** with $\epsilon = d - 3 > 0$. Regularity condition:

$$8 \left(\frac{n}{2}\right)^d = \frac{8}{2^d} n^d = 2^{3-d} n^d = c f(n) \quad \text{with } c = 2^{3-d} < 1.$$

Hence $T(n) = \Theta(n^d)$.

3. Family (C): $a = 4$, $f(n) = n \log n$, $n^{\log_b a} = n^{\log_b 4}$. We compare $\log_b 4$ with the exponent 1 of n in $f(n)$; note that $\log_b 4 = 1 \iff b = 4$, and $\log_b 4 > 1 \iff b < 4$.

- $1 < b < 4$: $p = \log_b 4 > 1 \rightarrow$ **case 1**. Choose $\epsilon = \frac{p-1}{2} > 0$; then $p - \epsilon = 1 + \frac{p-1}{2} > 1$ and

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^{p-\epsilon}} = \lim_{n \rightarrow \infty} \frac{\log n}{n^{(p-1)/2}} = 0^+,$$

so $f(n) = O(n^{p-\epsilon})$. Hence $T(n) = \Theta(n^{\log_b 4})$.

- $b = 4$: $n^{\log_4 4} = n$ and $f(n) = n \log n = \Theta(n \log^k n)$ with $k = 1 \rightarrow$ **case 2**. Hence $T(n) = \Theta(n \log^2 n)$.
- $b > 4$: $p = \log_b 4 < 1 \rightarrow$ **case 3**: since $n \log n \geq n$ for $n \geq 2$, $f(n) = \Omega(n^{p+\epsilon})$ with $\epsilon = 1 - p > 0$. Regularity condition, for $n \geq b$:

$$4 \frac{n}{b} \log \frac{n}{b} \leq \frac{4}{b} n \log n = c f(n) \quad \text{with } c = \frac{4}{b} < 1.$$

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

4. From question 1, $T(n) = \Theta(n^2)$ for $1 \leq a < 16$, while $T(n) = \Theta(n^2 \log n)$ for $a = 16$, which is not $O(n^2)$ because $\lim_{n \rightarrow \infty} \frac{n^2 \log n}{n^2} = +\infty$. Hence the largest integer is $a = 15$.

For $a = 17$ we are in case 1: $T(n) = \Theta(n^{\log_4 17}) \approx \Theta(n^{2.044})$. A small change in the number of subproblems changes the exponent of the running time.

Exercise 16.4 — The Master Theorem (E4)

Consider the following two recursive algorithms. In **PairSum** the length $n = r - l + 1$ of the sub-array is a power of 2; in **TernaryMax** it is a power of 3. The first call is on the whole array $a[1..n]$, i.e., with $l = 1$ and $r = n$.

Algorithm: PairSum

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

Output: An integer number s

```

1 function PairSum( $a, l, r$ )
2   if  $l = r$  then
3     return 0
4   end
5    $m := \lfloor (l + r) / 2 \rfloor$ 
6    $s := \text{PairSum}(a, l, m) + \text{PairSum}(a, m + 1, r)$ 
7    $i := l$ 
8   while  $i \leq m$  do
9      $j := m + 1$ 
10    while  $j \leq r$  do
11       $s \leftarrow s + a[i] \cdot a[j]$ 
12       $j \leftarrow j + 1$ 
13    end
14     $i \leftarrow i + 1$ 
15  end
16  return  $s$ 
17 end

```

Algorithm: TernaryMax

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

Output: An integer number

```

1 function TernaryMax( $a, l, r$ )
2   if  $l = r$  then
3     return  $a[l]$ 
4   end
5    $t := (r - l + 1) / 3$ 
6    $x := \text{TernaryMax}(a, l, l + t - 1)$ 
7    $y := \text{TernaryMax}(a, l + t, l + 2t - 1)$ 
8    $z := \text{TernaryMax}(a, l + 2t, r)$ 
9   return  $\max\{x, y, z\}$ 
10 end

```

Questions

1. What do the two algorithms compute?
2. Write the recurrence for the number of steps $S(n)$ executed by **PairSum**, counting the executions of each line, and solve it with the Master Theorem. Determine the worst-case running time $T(|I|)$.
3. Same question for **TernaryMax**. Then compute $S(n)$ exactly with the recursion tree and verify it for $n = 9$.

Solution

1. **PairSum**($a, 1, n$) returns the sum of the products of all pairs of distinct entries,

$$\sum_{1 \leq i < j \leq n} a[i] a[j] :$$

each pair is either inside the left half, or inside the right half (recursive calls), or has i in the left half and j in the right half (the two nested loops).

TernaryMax($a, 1, n$) returns the maximum value of the array: it splits the array into three parts of equal length, computes the maximum of each part recursively and returns the largest of the three.

2. Let $n = r - l + 1 \geq 2$ and $h = n/2$ (the length of each half). The number of times each line is executed, **excluding the steps executed inside the recursive calls**, is:

- **Line 2** is executed 1 time
- **Line 3** is executed 0 times
- **Line 5** is executed 1 time
- **Line 6** is executed 1 time (plus the two recursive calls on h entries)
- **Line 7** is executed 1 time
- **Line 8** is executed $h + 1$ times
- **Line 9** is executed h times
- **Line 10** is executed $h(h + 1)$ times
- **Line 11** is executed h^2 times
- **Line 12** is executed h^2 times
- **Line 14** is executed h times
- **Line 16** is executed 1 time

In total $1 + 1 + 1 + 1 + (h + 1) + h + h(h + 1) + h^2 + h^2 + h + 1 = 3h^2 + 4h + 6$. With $h = n/2$, and since for $n = 1$ only lines 2 and 3 are executed,

$$S(n) = \begin{cases} 2 & \text{if } n = 1, \\ 2S(n/2) + \frac{3}{4}n^2 + 2n + 6 & \text{if } n \geq 2. \end{cases}$$

(For example $S(2) = 2 \cdot 2 + \frac{3}{4} \cdot 4 + 4 + 6 = 17$.) We apply the Master Theorem:

- (a) $a = 2, b = 2, f(n) = \frac{3}{4}n^2 + 2n + 6$.
 (b) $n^{\log_b a} = n^{\log_2 2} = n$.
 (c) $f(n) = \Theta(n^2)$ is polynomially larger than $n \rightarrow$ **case 3**.
 (d) $f(n) \geq \frac{3}{4}n^2$, hence $f(n) = \Omega(n^{1+\epsilon})$ with $\epsilon = 1$. Regularity condition with $c = \frac{3}{4}$:

$$2f\left(\frac{n}{2}\right) = \frac{3}{8}n^2 + 2n + 12 \leq \frac{3}{4}\left(\frac{3}{4}n^2 + 2n + 6\right) = \frac{9}{16}n^2 + \frac{3}{2}n + \frac{9}{2}$$

$$\iff 0 \leq \frac{3}{16}n^2 - \frac{1}{2}n - \frac{15}{2} \iff 3n^2 - 8n - 120 \geq 0,$$

which holds for every $n \geq 8$ (the positive root is $\frac{8+\sqrt{1504}}{6} \approx 7.8$).

- (e) $S(n) = \Theta(f(n)) = \Theta(n^2)$.

Under the hypothesis that every entry uses a constant number of bits, $|I| = \Theta(n)$, hence $n = \Theta(|I|)$.

The worst-case running time of **PairSum**, as a function of the input size $|I|$, is

$$T(|I|) = \Theta(|I|^2).$$

3. For **TernaryMax** with $n \geq 3$ (a power of 3), excluding the recursive calls, lines 2, 5, 6, 7, 8, 9 are executed once each and line 3 is not executed: 6 steps. For $n = 1$, lines 2 and 3 are executed: 2 steps. Each of the three calls is on $t = n/3$ entries. Hence

$$S(n) = \begin{cases} 2 & \text{if } n = 1, \\ 3S(n/3) + 6 & \text{if } n \geq 3. \end{cases}$$

- (a) $a = 3, b = 3, f(n) = 6$.
 (b) $n^{\log_b a} = n^{\log_3 3} = n$.
 (c) n is polynomially larger than $f(n) = 6 \rightarrow$ **case 1**.
 (d) $f(n) = 6 = O(n^{1-\epsilon})$ with $\epsilon = 1$.
 (e) $S(n) = \Theta(n)$.

The worst-case running time of **TernaryMax**, as a function of the input size $|I|$, is

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

Exact value with the recursion tree, $n = 3^L$: level j has 3^j internal nodes of cost 6, and there are $3^L = n$ leaves of cost 2:

$$S(n) = 2n + \sum_{j=0}^{L-1} 6 \cdot 3^j = 2n + 6 \frac{3^L - 1}{3 - 1} = 2n + 3(n - 1) = 5n - 3.$$

Check for $n = 9$: $S(1) = 2, S(3) = 3 \cdot 2 + 6 = 12, S(9) = 3 \cdot 12 + 6 = 42 = 5 \cdot 9 - 3$. ✓

Exercise 16.5 — The Master Theorem (E5)

A student solves three recurrences as follows ($\log$ denotes $\log_2$, $T(1) = 1$):

- (a) $T(n) = 2T(n/2) + n \log n$. “ $f(n) = n \log n$ is larger than $n^{\log_2 2} = n$, so by case 3 $T(n) = \Theta(n \log n)$.”
- (b) $T(n) = 2T(n/2) + n/\log n$. “ $f(n) = n/\log n$ is smaller than n , so by case 1 $T(n) = \Theta(n)$.”
- (c) $T(n) = 2T(n/2) + n/\log^2 n$. “ $f(n) = n/\log^2 n$ is smaller than n , so by case 1 $T(n) = \Theta(n)$.”

Questions

- Find the mistake in each of the three solutions.
- Solve (a) correctly with the Master Theorem.
- Solve (b) and (c) for $n = 2^L$ with the recursion tree. Is the final answer of the student correct for (b)? And for (c)?
- Verify your formulas for (b) and (c) computing $T(16)$ directly from the recurrences.

Solution

- The student compares $f(n)$ with $n^{\log_b a} = n$ but forgets that the comparison must be **polynomial**.

- (a) Case 3 requires $f(n) = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$. But for every $\epsilon > 0$

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

so $f(n) \neq \Omega(n^{1+\epsilon})$: case 3 does not apply.

- (b) and (c) Case 1 requires $f(n) = O(n^{1-\epsilon})$ for some $\epsilon > 0$. But for every $\epsilon > 0$

$$\lim_{n \rightarrow \infty} \frac{n/\log n}{n^{1-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log n} = +\infty, \quad \lim_{n \rightarrow \infty} \frac{n/\log^2 n}{n^{1-\epsilon}} = \lim_{n \rightarrow \infty} \frac{n^\epsilon}{\log^2 n} = +\infty,$$

so case 1 does not apply. Case 2 would require $k = -1$ and $k = -2$, respectively, but $k \geq 0$. Both recurrences fall in the **gap** between case 1 and case 2: the Master Theorem cannot be used.

2. (a) $a = 2$, $b = 2$, $f(n) = n \log n$, $n^{\log_2 2} = n$. $f(n) = \Theta(n \log^k n)$ with $k = 1 \rightarrow$ **case 2:**

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

3. For $n = 2^L$: level j has 2^j nodes of size $n/2^j$, with $\log(n/2^j) = L - j$; there are $2^L = n$ leaves of cost $T(1) = 1$.

- (b) Level j costs $2^j \frac{n/2^j}{L-j} = \frac{n}{L-j}$, hence

$$T(n) = n + \sum_{j=0}^{L-1} \frac{n}{L-j} = n + n \sum_{i=1}^L \frac{1}{i} = n(1 + H_L).$$

Since $\ln(L+1) \leq H_L \leq 1 + \ln L$, $H_L = \Theta(\log L)$ and

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

The student's answer $\Theta(n)$ is **wrong**: $\lim_{n \rightarrow \infty} \frac{n \log \log n}{n} = +\infty$.

- (c) Level j costs $2^j \frac{n/2^j}{(L-j)^2} = \frac{n}{(L-j)^2}$, hence

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

The series $\sum_{i \geq 1} \frac{1}{i^2}$ converges (to $\frac{\pi^2}{6} \approx 1.645$), so

$$n \leq T(n) \leq n \left(1 + \frac{\pi^2}{6}\right), \quad \text{i.e.,} \quad T(n) = \Theta(n).$$

The student's **answer is correct, but the reasoning is wrong**: the Master Theorem does not apply, and the result must be proved with the recursion tree. Here the level costs decrease so fast towards the leaves that the leaves (n of them) dominate.

4. For $n = 16$ ($L = 4$):

- (b) $T(2) = 2 + \frac{2}{1} = 4$, $T(4) = 8 + \frac{4}{2} = 10$, $T(8) = 20 + \frac{8}{3} = \frac{68}{3}$, $T(16) = \frac{136}{3} + \frac{16}{4} = \frac{148}{3}$.
Formula: $16(1 + H_4) = 16(1 + \frac{25}{12}) = \frac{148}{3} \approx 49.33$. ✓

- (c) $T(2) = 2 + \frac{2}{1} = 4$, $T(4) = 8 + \frac{4}{4} = 9$, $T(8) = 18 + \frac{8}{9} = \frac{170}{9}$, $T(16) = \frac{340}{9} + \frac{16}{16} = \frac{349}{9}$.
Formula: $16(1 + 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16}) = 16(2 + \frac{61}{144}) = \frac{349}{9} \approx 38.78$. ✓

Exercise 16.6 — The Master Theorem (E6)

Consider the recurrences, defined for n power of 2:

$$T_1(n) = 3T_1(n/2) + n^2, \quad T_2(n) = 4T_2(n/2) + n, \quad T_1(1) = T_2(1) = 1.$$

Questions

1. For $n = 8$, draw the table of the recursion tree of T_1 (level, number of nodes, size, cost of the level) and verify the lemma, comparing with the value of $T_1(8)$ computed directly from the recurrence.
2. Same question for T_2 and $n = 8$.
3. Using the lemma, find a closed formula for $T_1(n)$ and $T_2(n)$, $n = 2^L$.
4. Solve both recurrences with the Master Theorem and check the result with the closed formulas.
5. For T_1 , check numerically, for $n = 8$, the bound $f(n) \leq g(n) \leq \frac{f(n)}{1-c}$ of case 3 of the lemma on the sum $g(n)$.

Solution

1. T_1 : $a = 3$, $b = 2$, $f(n) = n^2$, $\log_2 8 = 3$.

level j	nodes 3^j	size $8/2^j$	cost per node	cost of the level
0	1	8	64	64
1	3	4	16	48
2	9	2	4	36
3 (leaves)	27	1	$T_1(1) = 1$	27
total				175

The number of leaves is $3^3 = 27 = 8^{\log_2 3}$ (indeed $8^{\log_2 3} = (2^3)^{\log_2 3} = (2^{\log_2 3})^3 = 3^3$).
Directly:

$$T_1(2) = 3 + 4 = 7, \quad T_1(4) = 21 + 16 = 37, \quad T_1(8) = 111 + 64 = 175. \quad \checkmark$$

2. T_2 : $a = 4$, $b = 2$, $f(n) = n$.

level j	nodes 4^j	size $8/2^j$	cost per node	cost of the level
0	1	8	8	8
1	4	4	4	16
2	16	2	2	32
3 (leaves)	$64 = 8^{\log_2 4}$	1	$T_2(1) = 1$	64
total				120

Directly: $T_2(2) = 4 + 2 = 6$, $T_2(4) = 24 + 4 = 28$, $T_2(8) = 112 + 8 = 120$. ✓

3. With $n = 2^L$:

$$T_1(n) = n^{\log_2 3} + \sum_{j=0}^{L-1} 3^j \frac{n^2}{4^j} = n^{\log_2 3} + n^2 \frac{1 - (3/4)^L}{1 - 3/4} = n^{\log_2 3} + 4n^2 - 4n^2 \frac{3^L}{4^L}.$$

Since $4^L = n^2$ and $3^L = n^{\log_2 3}$,

$$T_1(n) = 4n^2 - 3n^{\log_2 3}.$$

(Check: $4 \cdot 64 - 3 \cdot 27 = 256 - 81 = 175$.) Similarly

$$T_2(n) = n^2 + \sum_{j=0}^{L-1} 4^j \frac{n}{2^j} = n^2 + n(2^L - 1) = 2n^2 - n.$$

(Check: $128 - 8 = 120$.)

4. • T_1 : $n^{\log_2 3} \approx n^{1.585}$ is polynomially smaller than $f(n) = n^2 \rightarrow$ **case 3**, with $\epsilon = 2 - \log_2 3 \approx 0.415$. Regularity: $3\left(\frac{n}{2}\right)^2 = \frac{3}{4}n^2$, so $c = \frac{3}{4} < 1$. Hence $T_1(n) = \Theta(n^2)$. Closed formula:

$$\lim_{n \rightarrow \infty} \frac{4n^2 - 3n^{\log_2 3}}{n^2} = \lim_{n \rightarrow \infty} \left(4 - \frac{3}{n^{2 - \log_2 3}}\right) = 4 \in \mathbb{R}_{>0}. \quad \checkmark$$

- T_2 : $n^{\log_2 4} = n^2$ is polynomially larger than $f(n) = n \rightarrow$ **case 1**, with $\epsilon = 1$. Hence $T_2(n) = \Theta(n^2)$. Closed formula: $\lim_{n \rightarrow \infty} \frac{2n^2 - n}{n^2} = 2 \in \mathbb{R}_{>0}$. ✓

Both recurrences are $\Theta(n^2)$, but for different reasons: in T_1 the cost is dominated by the root (level costs $n^2(3/4)^j$ decrease), in T_2 by the leaves (level costs $n2^j$ increase).

5. For T_1 and $n = 8$: $g(8) = 64 + 48 + 36 = 148$, $f(8) = 64$, $c = \frac{3}{4}$:

$$64 \leq 148 \leq \frac{64}{1 - 3/4} = 256. \quad \checkmark$$

Recursive algorithms

In the exercises of this chapter the steps are counted as in the chapter «Recursive algorithms»: every line of pseudo-code executed counts as one step, and a line containing a call counts one step plus the steps of the called function. The following results of the chapter are used.

Warning

Number of steps on an array (or sub-array) of n elements:

<code>SumIterative(a, n)</code>	$3n + 4$
<code>RecursiveSum($a, 1, n$)</code>	$7n - 5$, with $2n - 1$ calls and depth $\lceil \log_2 n \rceil + 1$
<code>RecursiveSumLinear(a, n)</code>	$3n - 1$, with n calls and depth n
<code>Merge(a, l, m, r), $n = r - l + 1$</code>	$6n + t + 11$, where t is the number of comparisons and $\min\{n_1, n_2\} \leq t \leq n - 1$
<code>RecursiveBinarySearch($a, 1, n, m$)</code>	$S(0) = 2$, $S(n) = S(\lfloor n/2 \rfloor) + 5$ for $n \geq 1$
<code>BruteForceMaximumSubarray(a, n)</code>	$2n^2 + 7n + 6 + 3U$, where U is the number of updates of <i>best</i>

Warning

Master Theorem. Let $S(n) = a S(n/b) + f(n)$, with $a \geq 1$ and $b > 1$ constants and $f(n)$ asymptotically positive.

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $S(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then $S(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n , then $S(n) = \Theta(f(n))$.

Exercise 17.1 — Recursive algorithms (E1)

Consider the recursive algorithm `RecursiveSum`:

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  $a[l]$ 
4   end
5    $m := \lfloor (l + r)/2 \rfloor$ 
6    $s_1 := \text{RecursiveSum}(a, l, m)$ 
7    $s_2 := \text{RecursiveSum}(a, m + 1, r)$ 
8   return  $s_1 + s_2$ 
9 end

```

and the array of $n = 9$ elements

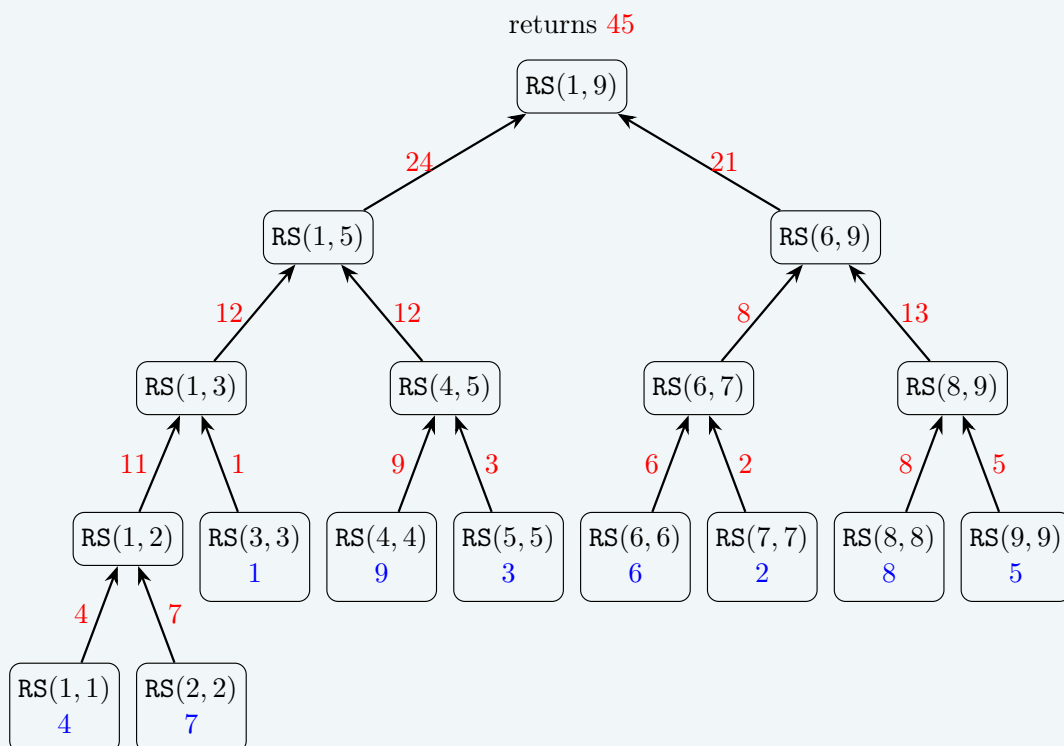
$$a = [4, 7, 1, 9, 3, 6, 2, 8, 5].$$

Questions

1. Draw the call tree of **RecursiveSum**($a, 1, 9$), indicating for each call the indices (l, r) and the value returned to its parent.
2. List the calls in the order in which they **terminate**.
3. How many calls are executed in total? How many additions? How many steps (count each line executed as one step)? Compare with the general formulas $2n - 1$, $n - 1$ and $7n - 5$.
4. What is the maximum number of calls that are simultaneously active (the recursion depth)? Which calls are active at that moment? Compare with the formula $\lceil \log_2 n \rceil + 1$.

Solution

1. The root **RS**(1, 9) computes $m = \lfloor 10/2 \rfloor = 5$; the left half $a[1..5]$ has $\lfloor 9/2 \rfloor = 5$ elements and the right half $a[6..9]$ has $\lfloor 9/2 \rfloor = 4$ elements. Proceeding in the same way for every call, we obtain the following call tree (in **blue** the elements returned by the base cases, in **red** the values returned to the parents):



The values of m computed by the internal calls are:

call	(1, 9)	(1, 5)	(1, 3)	(1, 2)	(4, 5)	(6, 9)	(6, 7)	(8, 9)
m	5	3	2	1	4	7	6	8
returned value	45	24	12	11	12	21	8	13

The algorithm returns $45 = 4 + 7 + 1 + 9 + 3 + 6 + 2 + 8 + 5$.

- The calls are executed depth-first, and a call terminates only after both its children have terminated. The order of termination is:

(1, 1), (2, 2), (1, 2), (3, 3), (1, 3), (4, 4), (5, 5), (4, 5), (1, 5),
(6, 6), (7, 7), (6, 7), (8, 8), (9, 9), (8, 9), (6, 9), (1, 9).

- The call tree has 9 leaves (one for each element) and 8 internal nodes:

- number of calls: $17 = 2 \cdot 9 - 1$;
- number of additions: one for each internal node, i.e., $8 = 9 - 1$;
- number of steps: each leaf executes 2 lines and each internal node 5 lines, so the total is $9 \cdot 2 + 8 \cdot 5 = 58 = 7 \cdot 9 - 5$.

- The call tree has height 4 (the leaves $RS(1, 1)$ and $RS(2, 2)$ are at level 4), so the maximum number of simultaneously active calls is 5. For instance, when $RS(1, 1)$ is executed, the active calls are

$RS(1, 9)$, $RS(1, 5)$, $RS(1, 3)$, $RS(1, 2)$, $RS(1, 1)$.

This agrees with the formula: $\lceil \log_2 9 \rceil + 1 = 4 + 1 = 5$, since $2^3 = 8 < 9 \leq 16 = 2^4$.

Note that the leaves are not all at the same level: $\text{RS}(3, 3)$, for instance, is at level 3. The deepest leaves are reached by always following the larger (left) half: $9 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1$.

Exercise 17.2 — Recursive algorithms (E2)

Consider the three algorithms that compute the sum of the elements of an array $a[1..n]$, $n \geq 1$, studied in the chapter «Recursive algorithms»:

- **SumIterative**(a, n) (a loop);
- **RecursiveSum**(a, l, n) (divide and conquer, $m = \lfloor (l + r)/2 \rfloor$, base case $l = r$);
- **RecursiveSumLinear**(a, n) (decrease by one: returns **RecursiveSumLinear**($a, n-1$) + $a[n]$, base case $n = 1$).

Their numbers of steps are recalled in the introduction to the exercises.

Questions

1. For **RecursiveSum**, let $A(n)$ be the number of additions executed on an array of n elements. Write a recurrence for $A(n)$ and prove by strong induction that $A(n) = n-1$.
2. Complete the following table, where *calls* is the total number of calls (for **SumIterative** write $-$) and *depth* is the maximum number of simultaneously active calls:

	$n = 10$			$n = 100$			$n = 1000$		
	steps	calls	depth	steps	calls	depth	steps	calls	depth
SumIterative									
RecursiveSum									
RecursiveSumLinear									

3. Consider the following variant of **RecursiveSum**, which also treats sub-arrays of two elements as a base case: if $r - l + 1 \leq 2$ then it returns $a[l]$ (if $l = r$) or $a[l] + a[r]$ (if $r = l + 1$); otherwise it proceeds as **RecursiveSum**. How many calls and how many additions does it execute for $n = 8$? And for $n = 2^k$, $k \geq 1$?
4. Explain why *every* algorithm that computes the sum of n numbers by adding them two at a time executes exactly $n - 1$ additions.

Solution

1. A call on one element executes no addition; a call on $n \geq 2$ elements executes one

addition $(s_1 + s_2)$ plus the additions of its two recursive calls:

$$A(n) = \begin{cases} 0 & \text{if } n = 1, \\ A(\lceil n/2 \rceil) + A(\lfloor n/2 \rfloor) + 1 & \text{if } n \geq 2. \end{cases}$$

Base case ($n = 1$): $A(1) = 0 = 1 - 1$.

Inductive step ($n \geq 2$): assume $A(k) = k - 1$ for every $1 \leq k < n$. Since $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$,

$$A(n) = (\lceil n/2 \rceil - 1) + (\lfloor n/2 \rfloor - 1) + 1 = \lceil n/2 \rceil + \lfloor n/2 \rfloor - 1 = n - 1.$$

2. We use the formulas of the chapter «Recursive algorithms» (see also the introduction to the exercises):

- **SumIterative**: $3n + 4$ steps, no recursive call;
- **RecursiveSum**: $7n - 5$ steps, $2n - 1$ calls, depth $\lceil \log_2 n \rceil + 1$;
- **RecursiveSumLinear**: $3n - 1$ steps, n calls (on $n, n - 1, \dots, 1$ elements), depth n (all the calls are active when the base case is reached).

Since $2^3 < 10 \leq 2^4$, $2^6 < 100 \leq 2^7$ and $2^9 < 1000 \leq 2^{10}$, we have $\lceil \log_2 10 \rceil = 4$, $\lceil \log_2 100 \rceil = 7$ and $\lceil \log_2 1000 \rceil = 10$. Therefore:

	$n = 10$			$n = 100$			$n = 1000$		
	steps	calls	depth	steps	calls	depth	steps	calls	depth
SumIterative	34	—	—	304	—	—	3004	—	—
RecursiveSum	65	19	5	695	199	8	6995	1999	11
RecursiveSumLinear	29	10	10	299	100	100	2999	1000	1000

All the three algorithms execute $\Theta(n)$ steps, but the extra memory (proportional to the depth) is $\Theta(1)$, $\Theta(\log_2 n)$ and $\Theta(n)$, respectively.

3. For $n = 8$ the sub-arrays have sizes $8 \rightarrow 4, 4 \rightarrow 2, 2, 2, 2$: the calls on 2 elements are base cases. The call tree has $1 + 2 + 4 = 7$ calls: 3 internal calls (one addition each) and 4 base cases on two elements (one addition each), for a total of $3 + 4 = 7$ additions.

For $n = 2^k$, the tree has levels $0, 1, \dots, k - 1$, with 2^i calls at level i on sub-arrays of 2^{k-i} elements; the calls at level $k - 1$ are the base cases on 2 elements. The number of calls is

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1 = n - 1$$

(about half of the $2n - 1$ calls of **RecursiveSum**). The number of additions is still $(2^{k-1} - 1) + 2^{k-1} = 2^k - 1 = n - 1$: one for each of the $2^{k-1} - 1$ internal calls and one for each of the 2^{k-1} base cases.

4. Every addition takes two numbers (elements of the array or partial sums) and replaces them by a single number (their sum). At the beginning there are n numbers, at the end there is only one number (the total sum), and each addition decreases the count by exactly one. Therefore exactly $n - 1$ additions are needed, regardless of the order in which they are performed (iterative, divide and conquer, decrease by one, ...).

Exercise 17.3 — Recursive algorithms (E3)

Given an array $a[1..n]$ of $n \geq 1$ integer numbers, consider the problem of determining the maximum value in the array. We want to solve it with a recursive **divide and conquer** algorithm $\text{RecursiveMax}(a, l, r)$, following the same scheme as $\text{RecursiveSum}(a, l, r)$, and to apply it to the array

$$a = [5, 17, 2, 11, 17, 8].$$

Questions

1. Write the pseudo-code of $\text{RecursiveMax}(a, l, r)$, which returns the maximum of $a[l..r]$ (base case $l = r$, middle index $m = \lfloor (l + r)/2 \rfloor$).
2. Draw the call tree of $\text{RecursiveMax}(a, 1, 6)$ for the given array, with the value returned by each call. How many calls and how many comparisons between elements are executed?
3. Prove by strong induction that $\text{RecursiveMax}(a, l, r)$ returns $\max\{a[l], \dots, a[r]\}$.
4. Write the recurrence for the number of steps $S(n)$ of RecursiveMax and solve it with the Master Theorem. Then compute $S(n)$ exactly.
5. Determine the worst-case running time $T(|I|)$, assuming that every entry uses a constant number of bits, and compare the algorithm with the iterative algorithm FindMaximum of the chapter «Running time».

Solution

1. The maximum of $a[l..r]$ is the larger of the maxima of the two halves:

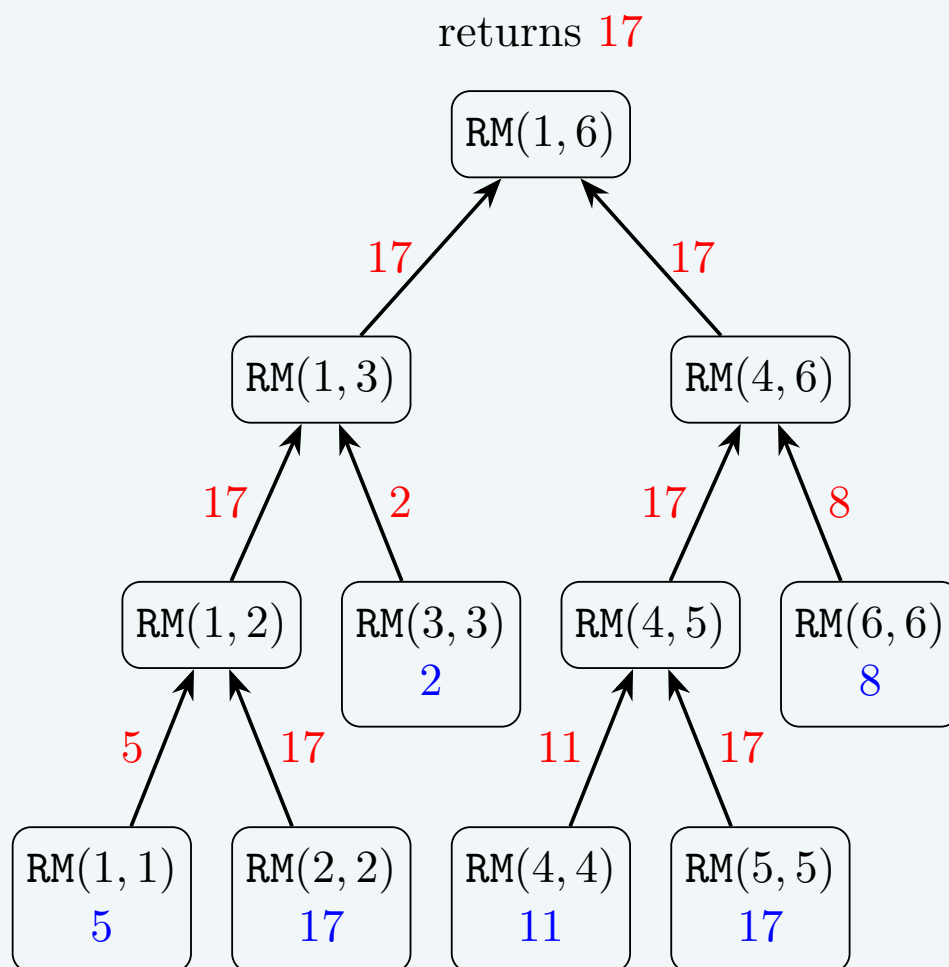
Algorithm: RecursiveMax**Input** : An array a and two indices $l \leq r$ **Output:** The maximum value in $a[l..r]$

```

1 function RecursiveMax( $a, l, r$ )
2   if  $l = r$  then
3     return  $a[l]$ 
4   end
5    $m := \lfloor (l + r)/2 \rfloor$ 
6    $x_1 := \text{RecursiveMax}(a, l, m)$ 
7    $x_2 := \text{RecursiveMax}(a, m + 1, r)$ 
8   if  $x_1 \geq x_2$  then
9     return  $x_1$ 
10  else
11    return  $x_2$ 
12  end
13 end

```

2. The root computes $m = \lfloor 7/2 \rfloor = 3$. This is the call tree, where $\text{RM}(l, r)$ stands for the call $\text{RecursiveMax}(a, l, r)$ and the returned values are in red:



The comparisons executed at line 8 are:

call	x_1	x_2	returned value
RM(1, 2)	5	17	$5 < 17 \Rightarrow 17$
RM(1, 3)	17	2	$17 \geq 2 \Rightarrow 17$
RM(4, 5)	11	17	$11 < 17 \Rightarrow 17$
RM(4, 6)	17	8	$17 \geq 8 \Rightarrow 17$
RM(1, 6)	17	17	$17 \geq 17 \Rightarrow 17$

There are $11 = 2 \cdot 6 - 1$ calls and $5 = 6 - 1$ comparisons (one for each internal call). The maximum is 17.

3. We use strong induction on $k = r - l + 1 \geq 1$.

Base case ($k = 1$): $l = r$ and the algorithm returns $a[l] = \max\{a[l]\}$.

Inductive step ($k \geq 2$): assume that the statement holds for all sub-arrays with fewer than k elements. 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 $k-1$; by the inductive hypothesis

$$x_1 = \max\{a[l], \dots, a[m]\}, \quad x_2 = \max\{a[m+1], \dots, a[r]\}.$$

Every element of $a[l..r]$ belongs to one of the two halves, so it is at most $\max\{x_1, x_2\}$; moreover $\max\{x_1, x_2\}$ is an element of $a[l..r]$. Therefore $\max\{a[l], \dots, a[r]\} = \max\{x_1, x_2\}$, which is exactly the value returned by lines 8–11.

4. If $n = 1$, lines 2 and 3 are executed: $S(1) = 2$. If $n \geq 2$:

- Lines 2, 5, 8 are executed 1 time each
- Line 6 is executed 1 time, plus the $S(\lceil n/2 \rceil)$ steps of the call
- Line 7 is executed 1 time, plus the $S(\lfloor n/2 \rfloor)$ steps of the call
- exactly one of lines 9 and 11 is executed 1 time

Therefore

$$S(n) = \begin{cases} 2 & \text{if } n = 1, \\ S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) + 6 & \text{if } n \geq 2, \end{cases} \quad \text{i.e.,} \quad S(n) = 2S(n/2) + c.$$

Master Theorem:

- $a = 2$, $b = 2$, $f(n) = c$;
- $n^{\log_b a} = n^{\log_2 2} = n$;
- $f(n) = c$ grows polynomially slower than $n \rightarrow$ case 1;
- $f(n) = c = O(n^{1-\varepsilon})$ with $\varepsilon = 1$;
- $S(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Exact value: as for **RecursiveSum**, the call tree has n leaves (cost 2 each) and $n - 1$ internal nodes (cost 6 each), hence

$$S(n) = 2n + 6(n - 1) = 8n - 6.$$

(It can be verified by strong induction: $S(1) = 2 = 8 - 6$ and $S(n) = (8\lceil n/2 \rceil - 6) + (8\lfloor n/2 \rfloor - 6) + 6 = 8n - 6$.) For the given array, $S(6) = 42$.

5. Since $|I| = \Theta(n)$ and $S(n) = 8n - 6 = \Theta(n)$, exactly as in the chapter «Recursive algorithms» we obtain

The worst-case running time of **RecursiveMax** is $T(|I|) = \Theta(|I|)$.

FindMaximum executes $4n - 1$ steps in the worst case: both algorithms are linear and perform $n - 1$ comparisons between elements. The recursive algorithm executes more steps ($8n - 6$) and uses $\Theta(\log_2 n)$ extra memory for the call stack, against $\Theta(1)$ of **FindMaximum**.

Exercise 17.4 — Recursive algorithms (E4)

Consider the following recursive algorithm, which splits the sub-array $a[l..r]$ in **three** parts:

Algorithm: RecursiveSum3**Input** : An array a and two indices $l \leq r$ **Output**: The sum $a[l] + a[l+1] + \dots + a[r]$

```

1 function RecursiveSum3( $a, l, r$ )
2   if  $r - l + 1 \leq 2$  then
3     if  $l = r$  then return  $a[l]$ 
4     return  $a[l] + a[r]$ 
5   end
6    $s := \lfloor (r - l + 1)/3 \rfloor$ 
7    $m_1 := l + s - 1$ 
8    $m_2 := m_1 + s$ 
9    $x_1 := \text{RecursiveSum3}(a, l, m_1)$ 
10   $x_2 := \text{RecursiveSum3}(a, m_1 + 1, m_2)$ 
11   $x_3 := \text{RecursiveSum3}(a, m_2 + 1, r)$ 
12  return  $x_1 + x_2 + x_3$ 
13 end

```

and the array of $n = 9$ elements $a = [2, 7, 1, 8, 2, 8, 1, 8, 2]$.

Questions

1. Show that, for $n = r - l + 1 \geq 3$, the three parts are non-empty and have fewer than n elements. What are their sizes?
2. Draw the call tree of `RecursiveSum3( $a, 1, 9$ )` with the value returned by each call.
3. Write the recurrence for the number of steps $S(n)$ and solve it with the Master Theorem.
4. Assume $n = 3^k$, $k \geq 1$. Compute the number of calls, the number of additions and the recursion depth, and compare them with those of `RecursiveSum`.

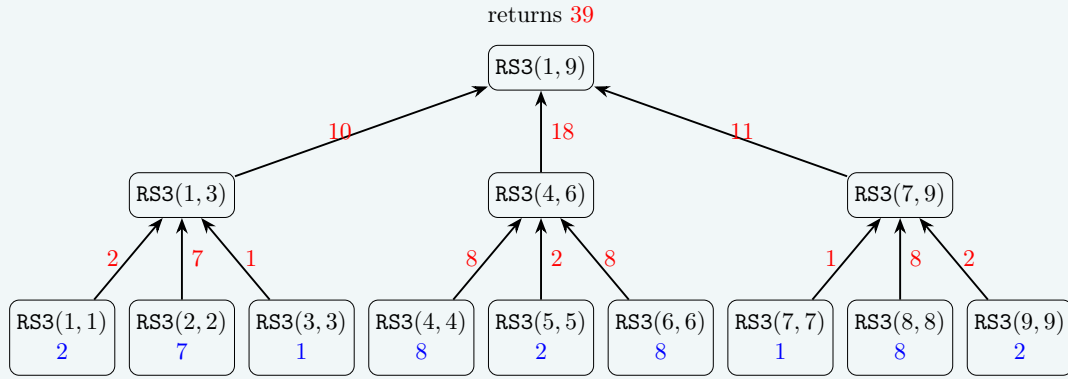
Solution

1. Let $n = r - l + 1 \geq 3$ and $s = \lfloor n/3 \rfloor \geq 1$. The three parts are

$$a[l..m_1] \text{ (} s \text{ elements), } a[m_1 + 1..m_2] \text{ (} s \text{ elements), } a[m_2 + 1..r] \text{ (} n - 2s \text{ elements).}$$

Since $s \geq 1$, the first two parts are non-empty and have $s \leq n/3 < n$ elements. Writing $n = 3s + \rho$ with $\rho \in \{0, 1, 2\}$, the third part has $n - 2s = s + \rho$ elements, with $1 \leq s \leq s + \rho < 3s + \rho = n$. So the third part is also non-empty and smaller than n , and all the sizes are about $n/3$ (they differ by at most 2).

2. The root has $n = 9$, $s = 3$, $m_1 = 3$, $m_2 = 6$; every part has 3 elements, which are split again into parts of one element:



For instance, $\text{RS3}(4, 6)$ computes $s = 1$, $m_1 = 4$, $m_2 = 5$ and returns $8 + 2 + 8 = 18$. The algorithm returns $10 + 18 + 11 = 39$, which is the sum of the elements. There are 13 calls and $4 \cdot 2 = 8$ additions.

3. For $n \leq 2$, lines 2, 3 and (if $n = 2$) 4 are executed: $S(1) = 2$, $S(2) = 3$. For $n \geq 3$, lines 2, 6, 7, 8, 12 are executed once, and lines 9, 10, 11 once each plus the steps of the three calls:

$$S(n) = S(s) + S(s) + S(n - 2s) + 8, \quad s = \lfloor n/3 \rfloor,$$

which, ignoring floors, we write as

$$S(n) = 3S(n/3) + c.$$

Master Theorem:

- $a = 3$, $b = 3$, $f(n) = c$;
- $n^{\log_b a} = n^{\log_3 3} = n$;
- $f(n) = c$ grows polynomially slower than $n \rightarrow$ case 1;
- $f(n) = c = O(n^{1-\varepsilon})$ with $\varepsilon = 1$;
- $S(n) = \Theta(n^{\log_3 3}) = \Theta(n)$.

As expected, splitting in three parts instead of two does not change the order of growth: every element must be read at least once.

4. For $n = 3^k$, every call on 3^j elements, $j \geq 1$, makes three calls on 3^{j-1} elements, and the calls on $3^0 = 1$ element are base cases. Level i of the call tree contains 3^i calls, for $i = 0, 1, \dots, k$:

- **calls:** $\sum_{i=0}^k 3^i = \frac{3^{k+1} - 1}{2} = \frac{3n - 1}{2}$ (for $n = 9$: 13, as in question 2), against $2n - 1$ of **RecursiveSum**;
- **additions:** each of the $\frac{3n-1}{2} - n = \frac{n-1}{2}$ internal calls executes 2 additions, for a total of $n - 1$, as **RecursiveSum** (every algorithm that adds two numbers at a time needs $n - 1$ additions);
- **recursion depth:** the height of the tree is $k = \log_3 n$, so the depth is $\log_3 n + 1$ (for $n = 9$: 3), against $\lceil \log_2 n \rceil + 1$ of **RecursiveSum**. Since $\log_3 n = \frac{\log_2 n}{\log_2 3} \approx 0.63 \log_2 n$, the depth is smaller by a constant factor, but it is still $\Theta(\log n)$.

Exercise 17.5 — Recursive algorithms (E5)

Consider the following array of $n = 8$ natural numbers:

$$a = [38, 27, 43, 3, 9, 82, 10, 15].$$

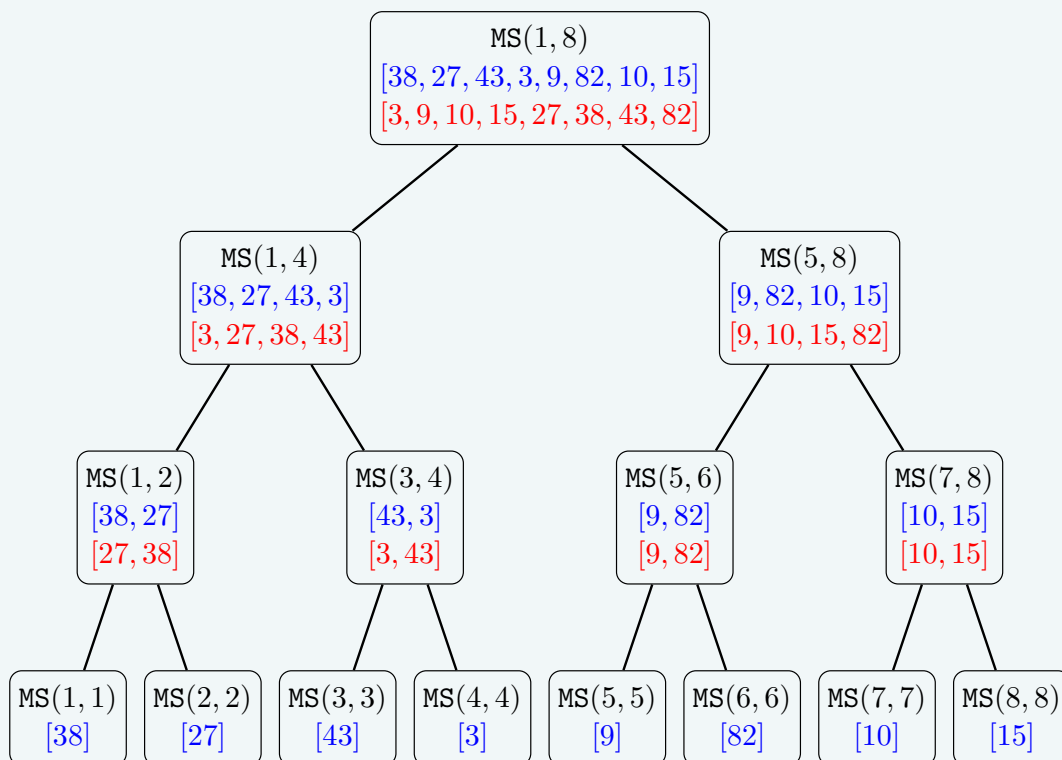
The array is sorted by the call `MergeSort(a, 1, 8)` of the algorithm studied in the chapter «Recursive algorithms» (with $m = \lfloor (l + r)/2 \rfloor$ and the procedure `Merge(a, l, m, r)`).

Questions

1. Draw the call tree of `MergeSort(a, 1, 8)`, indicating for each call the indices (l, r) , the sub-array received and the sorted sub-array produced.
2. List the calls of `Merge` in the order in which they are executed, and give the content of the array a after each of them.
3. For each call of `Merge`, give the number of comparisons between elements (line $L[i] \leq R[j]$). What is the total number of comparisons?
4. How many calls of `MergeSort` and how many calls of `Merge` are executed? Generalise to an array of $n = 2^k$ elements.

Solution

1. Since $n = 8$ is a power of 2, every sub-array is split into two halves of the same size. In each node, in **blue** the sub-array received and in **red** the sorted sub-array produced:



2. The calls of **Merge** are executed depth-first, after the two recursive calls of the corresponding **MergeSort** (in **red** the modified sub-array):

call	merged sub-arrays	array a after the call
$\text{Merge}(a, 1, 1, 2)$	[38], [27]	[27, 38 , 43, 3, 9, 82, 10, 15]
$\text{Merge}(a, 3, 3, 4)$	[43], [3]	[27, 38, 3, 43 , 9, 82, 10, 15]
$\text{Merge}(a, 1, 2, 4)$	[27, 38], [3, 43]	[3, 27, 38, 43 , 9, 82, 10, 15]
$\text{Merge}(a, 5, 5, 6)$	[9], [82]	[3, 27, 38, 43, 9, 82 , 10, 15]
$\text{Merge}(a, 7, 7, 8)$	[10], [15]	[3, 27, 38, 43, 9, 82, 10, 15]
$\text{Merge}(a, 5, 6, 8)$	[9, 82], [10, 15]	[3, 27, 38, 43, 9, 10, 15, 82]
$\text{Merge}(a, 1, 4, 8)$	[3, 27, 38, 43], [9, 10, 15, 82]	[3, 9, 10, 15, 27, 38, 43, 82]

3. The comparisons of each **Merge** are:

- $\text{Merge}(a, 1, 1, 2)$: $38 \leq 27$? No. Then L is copied. **1** comparison.
- $\text{Merge}(a, 3, 3, 4)$: $43 \leq 3$? No. **1** comparison.
- $\text{Merge}(a, 1, 2, 4)$, $L = [27, 38]$, $R = [3, 43]$: $27 \leq 3$? No; $27 \leq 43$? Yes; $38 \leq 43$? Yes; then L is exhausted and 43 is copied. **3** comparisons.
- $\text{Merge}(a, 5, 5, 6)$: $9 \leq 82$? Yes. **1** comparison.
- $\text{Merge}(a, 7, 7, 8)$: $10 \leq 15$? Yes. **1** comparison.
- $\text{Merge}(a, 5, 6, 8)$, $L = [9, 82]$, $R = [10, 15]$: $9 \leq 10$? Yes; $82 \leq 10$? No; $82 \leq 15$? No; then R is exhausted and 82 is copied. **3** comparisons.
- $\text{Merge}(a, 1, 4, 8)$, $L = [3, 27, 38, 43]$, $R = [9, 10, 15, 82]$:

k	i	j	$L[i]$	$R[j]$	action
1	1	1	3	9	$3 \leq 9$: $a[1] \leftarrow 3$
2	2	1	27	9	$27 > 9$: $a[2] \leftarrow 9$
3	2	2	27	10	$27 > 10$: $a[3] \leftarrow 10$
4	2	3	27	15	$27 > 15$: $a[4] \leftarrow 15$
5	2	4	27	82	$27 \leq 82$: $a[5] \leftarrow 27$
6	3	4	38	82	$38 \leq 82$: $a[6] \leftarrow 38$
7	4	4	43	82	$43 \leq 82$: $a[7] \leftarrow 43$
8	5	4	–	82	L exhausted: $a[8] \leftarrow 82$

7 comparisons.

The total number of comparisons is $1 + 1 + 3 + 1 + 1 + 3 + 7 = 17$. Each **Merge** of n' elements performs at most $n' - 1$ comparisons; here every **Merge** reaches this maximum $(1, 1, 3, 1, 1, 3, 7)$, so this array is a **worst case** for the number of comparisons of Merge Sort with $n = 8$: $17 = 8 \log_2 8 - 8 + 1$.

4. The call tree has 8 leaves and 7 internal nodes: there are $15 = 2 \cdot 8 - 1$ calls of **MergeSort** and $7 = 8 - 1$ calls of **Merge** (one for each internal node).

For $n = 2^k$, level i of the call tree contains 2^i calls, for $i = 0, 1, \dots, k$, so the number of calls of **MergeSort** is $\sum_{i=0}^k 2^i = 2^{k+1} - 1 = 2n - 1$. The leaves are the n calls on one element (which do not call **Merge**), so the number of calls of **Merge** is $2n - 1 - n = n - 1$.

Exercise 17.6 — Recursive algorithms (E6)

Consider the array of $n = 9$ elements

$$a = [2, 5, 8, 13, 20, 3, 4, 9, 21],$$

in which $a[1..5]$ and $a[6..9]$ are sorted, and the call **Merge**($a, 1, 5, 9$) of the procedure **Merge** studied in the chapter «Recursive algorithms». The number of steps of **Merge** on n elements is recalled in the introduction to the exercises (t is the number of iterations of the first while loop).

Questions

1. Give the values of n_1 , n_2 , L and R .
2. Trace the execution of **Merge**($a, 1, 5, 9$) in a table with columns k , i , j , $L[i]$, $R[j]$, comparison and action.
3. How many comparisons between elements are executed? How many elements are copied by the first while loop and how many by the other two loops? How many steps are executed?
4. Repeat question 3 (without the table) if the right part is replaced by $[22, 23, 24, 25]$ and by $[0, 1, 1, 1]$, i.e., for $a = [2, 5, 8, 13, 20, 22, 23, 24, 25]$ and for $a = [2, 5, 8, 13, 20, 0, 1, 1, 1]$.

Solution

1. $n_1 = m - l + 1 = 5 - 1 + 1 = 5$ and $n_2 = r - m = 9 - 5 = 4$. The two for loops of Merge copy the two sorted parts:

$$L = [2, 5, 8, 13, 20], \quad R = [3, 4, 9, 21].$$

2. Initially $i = 1, j = 1, k = 1$. One row per element copied into a :

k	i	j	$L[i]$	$R[j]$	comparison	action
1	1	1	2	3	$2 \leq 3$	$a[1] \leftarrow 2, i = 2$
2	2	1	5	3	$5 > 3$	$a[2] \leftarrow 3, j = 2$
3	2	2	5	4	$5 > 4$	$a[3] \leftarrow 4, j = 3$
4	2	3	5	9	$5 \leq 9$	$a[4] \leftarrow 5, i = 3$
5	3	3	8	9	$8 \leq 9$	$a[5] \leftarrow 8, i = 4$
6	4	3	13	9	$13 > 9$	$a[6] \leftarrow 9, j = 4$
7	4	4	13	21	$13 \leq 21$	$a[7] \leftarrow 13, i = 5$
8	5	4	20	21	$20 \leq 21$	$a[8] \leftarrow 20, i = 6$
9	6	4	–	21	L exhausted	$a[9] \leftarrow 21, j = 5$

After the 8-th iteration $i = 6 > n_1$: the first while loop ends, the loop that copies the rest of L is skipped, and the loop that copies the rest of R copies $R[4] = 21$. The result is

$$a = [2, 3, 4, 5, 8, 9, 13, 20, 21].$$

3. There are $t = 8$ comparisons, which is the maximum possible value $n - 1 = 8$. The first while loop copies 8 elements, the loop on L copies 0 elements and the loop on R copies 1 element. The number of steps is

$$6n + t + 11 = 6 \cdot 9 + 8 + 11 = 73.$$

4. **Right part** [22, 23, 24, 25]. All the elements of L are smaller than $R[1] = 22$: the first while loop copies 2, 5, 8, 13, 20 with the comparisons $2 \leq 22, 5 \leq 22, 8 \leq 22, 13 \leq 22, 20 \leq 22$, and stops because L is exhausted. The loop on R copies 22, 23, 24, 25. Hence $t = 5$ comparisons (5 elements copied by the first loop, 4 by the loop on R) and $6 \cdot 9 + 5 + 11 = 70$ steps.

Right part [0, 1, 1, 1]. All the elements of R are smaller than $L[1] = 2$: the first while loop copies 0, 1, 1, 1 with the comparisons $2 \leq 0, 2 \leq 1, 2 \leq 1, 2 \leq 1$ (all false), and stops because R is exhausted. The loop on L copies 2, 5, 8, 13, 20. Hence $t = 4$ comparisons (4 elements copied by the first loop, 5 by the loop on L) and $6 \cdot 9 + 4 + 11 = 69$ steps. This is the minimum possible number of comparisons, $\min\{n_1, n_2\} = 4$.

In all the cases the number of steps is between $6n + 11 = 65$ and $7n + 10 = 73$: Merge is linear in the number of elements, whatever their values.

Exercise 17.7 — Recursive algorithms (E7)

Consider the procedure $\text{Merge}(a, l, m, r)$ studied in the chapter «Recursive algorithms», which merges the sorted sub-arrays $a[l..m]$ (n_1 elements) and $a[m + 1..r]$ (n_2 elements), with $n_1, n_2 \geq 1$. Let C be the number of **comparisons between elements** ($L[i] \leq R[j]$) executed by Merge , and let $C_{\text{MS}}(n)$ be the total number of comparisons executed by MergeSort on an array of n elements.

Questions

1. Prove that $\min\{n_1, n_2\} \leq C \leq n_1 + n_2 - 1$.
2. Show that both bounds are tight: for $n_1 = n_2 = 4$, give two sorted sub-arrays for which Merge performs 4 comparisons and two for which it performs 7 comparisons.
3. Let $n = 2^k$. Write recurrences for the minimum $B(n)$ and for the maximum $W(n)$ of $C_{\text{MS}}(n)$ over all the arrays of n elements, and solve them.
4. For $n = 8$, verify the formulas with the arrays $[1, 2, 3, 4, 5, 6, 7, 8]$ and $[1, 5, 3, 7, 2, 6, 4, 8]$. What happens with the array $[8, 7, 6, 5, 4, 3, 2, 1]$?

Solution

1. Every iteration of the first while loop executes exactly one comparison and copies exactly one element into a ; the other two loops do not execute comparisons. Hence $C = t$, the number of iterations of the first while loop.
 - **Lower bound.** The loop ends only when $i > n_1$ or $j > n_2$, i.e., when all the elements of L or all the elements of R have been copied by the loop. Therefore $t \geq n_1$ or $t \geq n_2$, and in both cases $t \geq \min\{n_1, n_2\}$.
 - **Upper bound.** Each iteration increases i or j by one, and before the last iteration both $i \leq n_1$ and $j \leq n_2$ hold. When the loop ends, exactly one of the two arrays is exhausted, while the other one still contains at least one element (it was not exhausted before the last iteration, and the last iteration copied an element of the first array). Therefore at most $n_1 + n_2 - 1$ elements are copied by the loop: $t \leq n_1 + n_2 - 1$.
2. **4 comparisons:** all the elements of one part are smaller than all the elements of the other one, e.g., $L = [1, 2, 3, 4]$ and $R = [5, 6, 7, 8]$. The comparisons are $1 \leq 5$, $2 \leq 5$, $3 \leq 5$, $4 \leq 5$; then L is exhausted and $5, 6, 7, 8$ are copied without comparisons.
7 comparisons: the elements of the two parts alternate, e.g., $L = [1, 3, 5, 7]$ and $R = [2, 4, 6, 8]$. The comparisons are

$$1 \leq 2, \quad 3 \leq 2, \quad 3 \leq 4, \quad 5 \leq 4, \quad 5 \leq 6, \quad 7 \leq 6, \quad 7 \leq 8,$$
 after which L is exhausted and only 8 is copied without comparisons.
3. Let $n = 2^k$. MergeSort executes the comparisons of the two recursive calls on $n/2$

elements plus those of the final **Merge** with $n_1 = n_2 = n/2$, and the calls on one element execute no comparison.

- **Minimum.** By question 1, the final **Merge** executes at least $n/2$ comparisons, and exactly $n/2$ if the two halves are, e.g., one entirely smaller than the other one (which is the case if the array is sorted, at all levels of the recursion). Hence

$$B(1) = 0, \quad B(n) = 2B(n/2) + \frac{n}{2}.$$

- **Maximum.** The final **Merge** executes at most $n - 1$ comparisons, and exactly $n - 1$ if the elements of the two sorted halves alternate. Hence

$$W(1) = 0, \quad W(n) = 2W(n/2) + n - 1.$$

(That the maximum is reached simultaneously at all levels is shown by the example of question 4.)

We solve the recurrences with the recursion tree: level i , $i = 0, \dots, k - 1$, has 2^i calls on $n/2^i$ elements.

$$B(n) = \sum_{i=0}^{k-1} 2^i \cdot \frac{n}{2^{i+1}} = \sum_{i=0}^{k-1} \frac{n}{2} = \frac{n}{2} \log_2 n,$$

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

In both cases $C_{MS}(n) = \Theta(n \log_2 n)$: also the best case of **MergeSort** is $\Theta(n \log n)$ (on the contrary, the best case of **InsertionSort** is linear).

4. For $n = 8$ ($k = 3$) the formulas give $B(8) = 4 \cdot 3 = 12$ and $W(8) = 24 - 8 + 1 = 17$.

- $[1, 2, 3, 4, 5, 6, 7, 8]$ (sorted): the four merges of pairs execute 1 comparison each, the two merges of quadruples ($[1, 2]$ with $[3, 4]$, $[5, 6]$ with $[7, 8]$) 2 each, and the final merge of $[1, 2, 3, 4]$ with $[5, 6, 7, 8]$ executes 4 comparisons: $4 \cdot 1 + 2 \cdot 2 + 4 = 12 = B(8)$.
- $[1, 5, 3, 7, 2, 6, 4, 8]$: the pairs $[1, 5]$, $[3, 7]$, $[2, 6]$, $[4, 8]$ are already sorted (1 comparison each); then $[1, 5]$ and $[3, 7]$ alternate, giving $[1, 3, 5, 7]$ with 3 comparisons, and $[2, 6]$ and $[4, 8]$ alternate, giving $[2, 4, 6, 8]$ with 3 comparisons; finally $[1, 3, 5, 7]$ and $[2, 4, 6, 8]$ alternate (7 comparisons, see question 2). Total: $4 \cdot 1 + 2 \cdot 3 + 7 = 17 = W(8)$.
- $[8, 7, 6, 5, 4, 3, 2, 1]$ (reversed): every merge combines two halves in which all the elements of R are smaller than all the elements of L . Each merge executes $\min\{n_1, n_2\}$ comparisons, exactly as for the sorted array: $4 \cdot 1 + 2 \cdot 2 + 4 = 12 = B(8)$. The reversed array, which is the worst case of **InsertionSort**, is a best case for the number of comparisons of **MergeSort**.

Exercise 17.8 — Recursive algorithms (E8)

Consider the following **3-way** variant of Merge Sort, which splits $a[l..r]$ in three parts and uses the procedure **Merge** of the chapter «Recursive algorithms» twice (**Merge** works correctly also when its right part is empty, i.e., $n_2 = 0$):

Algorithm: MergeSort3

Input : An array a and two indices l, r

Output: The sub-array $a[l..r]$ sorted in non-decreasing order

```

1 function MergeSort3( $a, l, r$ )
2   if  $l < r$  then
3      $m_1 := \lfloor (2l + r)/3 \rfloor$ 
4      $m_2 := \lfloor (l + 2r)/3 \rfloor$ 
5     MergeSort3( $a, l, m_1$ )
6     MergeSort3( $a, m_1 + 1, m_2$ )
7     MergeSort3( $a, m_2 + 1, r$ )
8     Merge( $a, l, m_1, m_2$ )
9     Merge( $a, l, m_2, r$ )
10  end
11 end

```

Consider also the array $a = [24, 5, 17, 9, 31, 2, 12, 20, 7]$ of $n = 9$ elements.

Questions

1. Apply **MergeSort3**($a, 1, 9$) to the given array: give the three parts of the first call, the sorted parts returned by the three recursive calls, and the result of each of the two final calls of **Merge**.
2. Explain what lines 8 and 9 do, and show that together they execute $\Theta(n)$ steps, where $n = r - l + 1$.
3. Write the recurrence for the number of steps $S(n)$ of **MergeSort3** and solve it with the Master Theorem. Is **MergeSort3** asymptotically faster than **MergeSort**?
4. Let $n = 3^k$. Compare the maximum number of comparisons between elements executed at each level of the recursion and the number of levels with those of **MergeSort**. Which algorithm executes fewer comparisons in the worst case, for large n ?

Solution

1. In the first call $l = 1, r = 9$: $m_1 = \lfloor 11/3 \rfloor = 3$ and $m_2 = \lfloor 19/3 \rfloor = 6$. The three parts are

$$a[1..3] = [24, 5, 17], \quad a[4..6] = [9, 31, 2], \quad a[7..9] = [12, 20, 7].$$

Each part has 3 elements; for instance, for $a[1..3]$: $m_1 = \lfloor 5/3 \rfloor = 1$ and $m_2 = \lfloor 7/3 \rfloor =$

2, so it is split into three parts of one element. The three recursive calls return

$$a[1..3] = [5, 17, 24], \quad a[4..6] = [2, 9, 31], \quad a[7..9] = [7, 12, 20],$$

i.e., $a = [5, 17, 24, 2, 9, 31, 7, 12, 20]$. Then:

- **Merge**($a, 1, 3, 6$) merges $[5, 17, 24]$ and $[2, 9, 31]$ into $a[1..6] = [2, 5, 9, 17, 24, 31]$; now $a = [2, 5, 9, 17, 24, 31, 7, 12, 20]$;
- **Merge**($a, 1, 6, 9$) merges $[2, 5, 9, 17, 24, 31]$ and $[7, 12, 20]$: $a = [2, 5, 7, 9, 12, 17, 20, 24, 31]$.

2. After the three recursive calls, $a[l..m_1]$, $a[m_1 + 1..m_2]$ and $a[m_2 + 1..r]$ are sorted. Line 8 merges the first two parts, so that $a[l..m_2]$ is sorted; line 9 merges $a[l..m_2]$ with the third part, so that $a[l..r]$ is sorted.

Merge on n' elements executes between $6n' + 11$ and $7n' + 10$ steps. The first call works on $n' = m_2 - l + 1 \leq n$ elements and the second one on $n' = n$ elements, so the two calls together execute at least $6n + 11$ and at most $2(7n + 10) = 14n + 20$ steps: $\Theta(n)$.

3. For $n \geq 2$, lines 2–9 are executed once each (8 steps), plus the steps of three recursive calls on about $n/3$ elements, plus the $\Theta(n)$ steps of the two calls of **Merge**:

$$S(n) = 3S(n/3) + \Theta(n).$$

Master Theorem:

- $a = 3, b = 3, f(n) = \Theta(n)$;
- $n^{\log_b a} = n^{\log_3 3} = n$;
- $f(n)$ and $n^{\log_b a}$ grow in the same way $\rightarrow$ case 2;
- $f(n) = \Theta(n) = \Theta(n^{\log_3 3} \log^0 n)$ with $k = 0$;
- $S(n) = \Theta(n^{\log_3 3} \log n) = \Theta(n \log n)$.

MergeSort3 is **not** asymptotically faster than **MergeSort**: both execute $\Theta(n \log n)$ steps. The recursion tree of **MergeSort3** has only $\log_3 n + 1$ levels instead of $\log_2 n + 1$, but

$$\log_3 n = \frac{\log_2 n}{\log_2 3},$$

and the two logarithms differ only by a constant factor, which disappears inside Θ .

4. For $n = 3^k$, every call on $n' = 3^j \geq 3$ elements splits it into three parts of $n'/3$ elements. The first **Merge** works on $2n'/3$ elements and executes at most $2n'/3 - 1$ comparisons, the second one works on n' elements and executes at most $n' - 1$ comparisons: at most $\frac{5}{3}n' - 2$ comparisons per call. Summing over the 3^i calls of level i (with $n' = n/3^i$), each level executes fewer than $\frac{5}{3}n$ comparisons, and there are $k = \log_3 n$ levels with **Merge**. In the worst case:

- **MergeSort3**: about $\frac{5}{3} n \log_3 n = \frac{5}{3 \log_2 3} n \log_2 n \approx 1.05 n \log_2 n$ comparisons;

- **MergeSort**: about $n \log_2 n$ comparisons

(the exact value for **MergeSort** is $n \log_2 n - n + 1$ when n is a power of 2). The levels are fewer, but each level is more expensive: for large n , **MergeSort3** executes about 5% **more** comparisons in the worst case. Both are $\Theta(n \log n)$.

Exercise 17.9 — Recursive algorithms (E9)

For small sub-arrays **InsertionSort** is faster in practice than **MergeSort**. Consider the following **hybrid** algorithm, where $k \geq 1$ is a parameter and **InsertionSort**(a, l, r) is Insertion Sort applied to the sub-array $a[l..r]$ (it executes $\Theta(p^2)$ steps in the worst case on p elements):

Algorithm: HybridMergeSort

Input : An array a , two indices $l \leq r$ and a parameter $k \geq 1$

Output: The sub-array $a[l..r]$ sorted in non-decreasing order

```

1 function HybridMergeSort( $a, l, r, k$ )
2   if  $r - l + 1 \leq k$  then
3     InsertionSort( $a, l, r$ )
4   else
5      $m := \lfloor (l + r) / 2 \rfloor$ 
6     HybridMergeSort( $a, l, m, k$ )
7     HybridMergeSort( $a, m + 1, r, k$ )
8     Merge( $a, l, m, r$ )
9   end
10 end
```

Questions

1. Apply **HybridMergeSort**($a, 1, 8, 2$) to the array $a = [7, 2, 9, 4, 3, 8, 6, 1]$: list, in order, the calls of the procedures **InsertionSort** and **Merge**, with their results.
2. Let $n = k \cdot 2^h$ for a natural number h . Show that the calls of **InsertionSort** are executed on n/k sub-arrays of exactly k elements, and that they execute $\Theta(nk)$ steps in total in the worst case.
3. Show that the calls of **Merge** execute $\Theta(n \log_2(n/k))$ steps in total.
4. Conclude that the worst-case number of steps of **HybridMergeSort** is

$$S(n) = \Theta(nk + n \log(n/k)),$$

and discuss the extreme cases $k = 1$ and $k = n$.

5. What is the largest order of growth of k , as a function of n , for which **HybridMergeSort** still executes $\Theta(n \log n)$ steps?

Solution

1. With $k = 2$, the sub-arrays with at most 2 elements are sorted by **InsertionSort**. The call on $a[1..8]$ ($m = 4$) calls $a[1..4]$ ($m = 2$) and $a[5..8]$ ($m = 6$), which call the sub-arrays of two elements. In order:

call	sub-array	result
InsertionSort ($a, 1, 2$)	[7, 2]	[2, 7]
InsertionSort ($a, 3, 4$)	[9, 4]	[4, 9]
Merge ($a, 1, 2, 4$)	[2, 7], [4, 9]	[2, 4, 7, 9]
InsertionSort ($a, 5, 6$)	[3, 8]	[3, 8]
InsertionSort ($a, 7, 8$)	[6, 1]	[1, 6]
Merge ($a, 5, 6, 8$)	[3, 8], [1, 6]	[1, 3, 6, 8]
Merge ($a, 1, 4, 8$)	[2, 4, 7, 9], [1, 3, 6, 8]	[1, 2, 3, 4, 6, 7, 8, 9]

There are $4 = n/k$ calls of **InsertionSort** and 3 calls of **Merge**, against the 7 calls of **Merge** of **MergeSort** on 8 elements.

2. Let $n = k \cdot 2^h$. A call on $k \cdot 2^j$ elements with $j \geq 1$ has more than k elements, so it splits the sub-array into two halves of $k \cdot 2^{j-1}$ elements each. Therefore the recursion tree has levels $0, 1, \dots, h$: level i contains 2^i calls on $n/2^i = k \cdot 2^{h-i}$ elements, and the calls of level h , on exactly k elements, are the leaves, which call **InsertionSort**. There are $2^h = n/k$ leaves.

In the worst case **InsertionSort** on k elements executes $\frac{3k^2+9k-8}{2} = \Theta(k^2)$ steps (see the chapter «Running time»). This happens on every leaf at the same time, for instance if the whole array is sorted in decreasing order (then every sub-array of the leaves is in decreasing order). In total:

$$\frac{n}{k} \cdot \Theta(k^2) = \Theta(nk).$$

3. The calls of **Merge** are executed by the internal nodes, i.e., at levels $0, 1, \dots, h-1$. At level i there are 2^i calls of **Merge**, each on $n/2^i$ elements. Since **Merge** on n' elements executes between $6n' + 11$ and $7n' + 10$ steps, the calls at level i execute between

$$2^i \left(6 \frac{n}{2^i} + 11 \right) \geq 6n \quad \text{and} \quad 2^i \left(7 \frac{n}{2^i} + 10 \right) = 7n + 10 \cdot 2^i \leq 17n$$

steps (since $2^i \leq 2^h = n/k \leq n$). Each level costs $\Theta(n)$ and there are $h = \log_2(n/k)$ levels with **Merge**, so the total is

$$\Theta(n \log_2(n/k)).$$

4. The steps of **HybridMergeSort** are those of the leaves (question 2), plus those of the internal nodes (question 3), plus a constant number of steps (lines 2–8) for each of the $2^{h+1} - 1 < 2n$ calls, which is $O(n)$ and is absorbed by the other terms. In the worst case,

$$S(n) = \Theta(nk) + \Theta(n \log(n/k)) = \Theta(nk + n \log(n/k)).$$

- $k = 1$: the leaves contain one element, **InsertionSort** does nothing significant and the algorithm is **MergeSort**: $S(n) = \Theta(n + n \log n) = \Theta(n \log n)$.
 - $k = n$: there is a single leaf, the whole array is sorted by **InsertionSort** and there is no merge: $S(n) = \Theta(n^2 + n \log 1) = \Theta(n^2)$.
5. Since $\log(n/k) \leq \log n$, we always have $n \log(n/k) = O(n \log n)$. Hence $S(n) = \Theta(n \log n)$ if and only if, in addition, $nk = O(n \log n)$, i.e.,

$$k = O(\log n).$$

Indeed, if $k = \Theta(\log n)$, then $nk = \Theta(n \log n)$ and $n \log(n/k) = n \log n - n \log k \leq n \log n$, so $S(n) = \Theta(n \log n)$. If k grows faster than $\log n$ (for instance $k = \sqrt{n}$), the term nk dominates and $S(n)$ grows faster than $n \log n$.

Therefore the largest admissible order of growth is $k = \Theta(\log n)$. In practice, k is chosen experimentally (typical values are between 8 and 32), as the largest size for which, on the given computer, Insertion Sort is faster than Merge Sort.

Exercise 17.10 — Recursive algorithms (E10)

Consider the following sorted array of $n = 10$ elements:

$$a = [1, 4, 7, 10, 13, 17, 21, 25, 30, 34]$$

and the algorithm **RecursiveBinarySearch**($a, left, right, m$), with first call **RecursiveBinarySearch**($a, 1, 10, m$) and middle index

$$mid = left + \left\lfloor \frac{right - left}{2} \right\rfloor.$$

Questions

1. Apply the Recursive Binary Search algorithm for the following values of m :

- (a) $m = 21$
- (b) $m = 1$
- (c) $m = 34$
- (d) $m = 15$
- (e) $m = 40$

For each search, report one row for each call, indicating the values of $left$, $right$, mid , $a[mid]$ and the action executed, and give the returned value.

2. Draw the tree of calls for $m = 15$, indicating the values returned by each call.
3. For each search, count the number of calls on a non-empty interval and the total number of calls. Compare these numbers with the bound $\lceil \log_2 n \rceil + 1$ on the number of calls on a non-empty interval.

Solution

1. (a) Search for $m = 21$, first call `RecursiveBinarySearch(a, 1, 10, 21)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	10	5	13	$a[mid] < m \Rightarrow$ call on $left = 6, right = 10$
2	6	10	8	25	$a[mid] > m \Rightarrow$ call on $left = 6, right = 7$
3	6	7	6	17	$a[mid] < m \Rightarrow$ call on $left = 7, right = 7$
4	7	7	7	21	$a[mid] = m \Rightarrow$ Found! Return 7

The value 7 is returned back along the calls: element 21 is found at index 7.

- (b) Search for $m = 1$, first call `RecursiveBinarySearch(a, 1, 10, 1)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	10	5	13	$a[mid] > m \Rightarrow$ call on $left = 1, right = 4$
2	1	4	2	4	$a[mid] > m \Rightarrow$ call on $left = 1, right = 1$
3	1	1	1	1	$a[mid] = m \Rightarrow$ Found! Return 1

Element 1 is found at index 1.

- (c) Search for $m = 34$, first call `RecursiveBinarySearch(a, 1, 10, 34)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	10	5	13	$a[mid] < m \Rightarrow$ call on $left = 6, right = 10$
2	6	10	8	25	$a[mid] < m \Rightarrow$ call on $left = 9, right = 10$
3	9	10	9	30	$a[mid] < m \Rightarrow$ call on $left = 10, right = 10$
4	10	10	10	34	$a[mid] = m \Rightarrow$ Found! Return 10

Element 34 is found at index 10.

- (d) Search for $m = 15$, first call `RecursiveBinarySearch(a, 1, 10, 15)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	10	5	13	$a[mid] < m \Rightarrow$ call on $left = 6, right = 10$
2	6	10	8	25	$a[mid] > m \Rightarrow$ call on $left = 6, right = 7$
3	6	7	6	17	$a[mid] > m \Rightarrow$ call on $left = 6, right = 5$
4	6	5	-	-	$left > right \Rightarrow$ Not found! Return -1

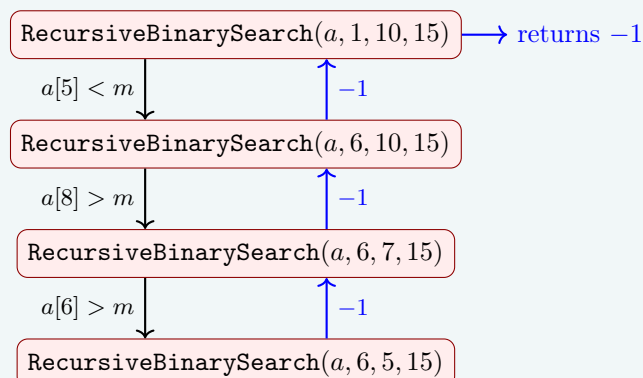
Element 15 is not in the array (return -1).

- (e) Search for $m = 40$, first call `RecursiveBinarySearch(a, 1, 10, 40)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	10	5	13	$a[mid] < m \Rightarrow$ call on $left = 6, right = 10$
2	6	10	8	25	$a[mid] < m \Rightarrow$ call on $left = 9, right = 10$
3	9	10	9	30	$a[mid] < m \Rightarrow$ call on $left = 10, right = 10$
4	10	10	10	34	$a[mid] < m \Rightarrow$ call on $left = 11, right = 10$
5	11	10	-	-	$left > right \Rightarrow$ Not found! Return -1

Element 40 is not in the array (return -1).

2. Each call makes at most one recursive call, hence the tree of calls is a path. For $m = 15$ (black arrows: recursive calls; blue arrows: returned values):



The last call is on the empty interval $left = 6$, $right = 5$ and returns -1 ; every other call returns, unchanged, the value returned by its recursive call.

3. For $n = 10$ we have $\lfloor \log_2 10 \rfloor + 1 = 3 + 1 = 4$, since $2^3 = 8 \leq 10 < 16 = 2^4$.

m	calls on a non-empty interval	total calls
21	4	4
1	3	3
34	4	4
15	3	4
40	4	5

In every search the number of calls on a non-empty interval is at most 4, as predicted by the bound. For $m = 40 > a[10]$ the algorithm always continues in the right half, the bound 4 is attained, and one more call on the empty interval follows: the total number of calls is $\lfloor \log_2 10 \rfloor + 2 = 5$, which is the maximum possible for $n = 10$.

Exercise 17.11 — Recursive algorithms (E11)

Consider the algorithm `RecursiveBinarySearch(a, left, right, m)` applied to a sorted array $a[1..n]$, with first call `RecursiveBinarySearch(a, 1, n, m)`. Its worst-case number of steps $S(n)$ satisfies the recurrence recalled in the introduction to the exercises.

Questions

- Determine the maximum number of calls on a non-empty interval and the maximum total number of calls (including the final call on an empty interval) for an array of $n = 1000$ elements. Give an example of a target value m for which the maximum is attained, if $a[i] = 2i$ for $i = 1, \dots, 1000$.
- Answer the same questions for $n = 10^6$.

3. Compute $S(1000)$ and $S(10^6)$.
4. Determine the smallest n for which, in the worst case, the algorithm makes 21 calls on a non-empty interval.
5. What is the maximum number of calls that are simultaneously in the call stack for $n = 10^6$? Compare with the number of elements examined by a linear search in the worst case.

Solution

1. The maximum number of calls on a non-empty interval is $\lfloor \log_2 n \rfloor + 1$, and the maximum total number of calls is $\lfloor \log_2 n \rfloor + 2$. For $n = 1000$:

$$2^9 = 512 \leq 1000 < 1024 = 2^{10} \implies \lfloor \log_2 1000 \rfloor = 9.$$

Therefore:

- maximum number of calls on a non-empty interval: $9 + 1 = 10$;
- maximum total number of calls: $9 + 2 = 11$.

The maximum is attained, for example, by any $m > a[1000] = 2000$, e.g. $m = 2001$: the algorithm always continues in the right half, whose sizes are

$$1000, 500, 250, 125, 62, 31, 15, 7, 3, 1, 0,$$

that is, 10 non-empty intervals followed by the empty one.

2. For $n = 10^6$:

$$2^{19} = 524\,288 \leq 10^6 < 1\,048\,576 = 2^{20} \implies \lfloor \log_2 10^6 \rfloor = 19.$$

Therefore:

- maximum number of calls on a non-empty interval: $19 + 1 = 20$;
- maximum total number of calls: $19 + 2 = 21$.

Again, the maximum is attained by any $m > a[n]$.

3. By expansion, $S(n) = 5(\lfloor \log_2 n \rfloor + 1) + 2$ for $n \geq 1$. Hence

$$S(1000) = 5 \cdot 10 + 2 = 52, \quad S(10^6) = 5 \cdot 20 + 2 = 102.$$

Multiplying the size of the array by 1000 only doubles (approximately) the number of steps.

4. We need $\lfloor \log_2 n \rfloor + 1 = 21$, i.e., $\lfloor \log_2 n \rfloor = 20$, i.e.,

$$2^{20} \leq n < 2^{21}.$$

The smallest such n is $n = 2^{20} = 1\,048\,576$. Indeed, for $n = 2^{20} - 1$ we have $\lfloor \log_2 n \rfloor = 19$ and only 20 calls on a non-empty interval.

5. No call terminates before its recursive call has terminated; hence, when the deepest call is executing, all the calls of the path are in the call stack. For $n = 10^6$ there are at most 21 calls simultaneously in the call stack (extra memory $\Theta(\log n)$).

A linear search examines, in the worst case, all the 10^6 elements of the array, while the Recursive Binary Search examines at most 20 elements (one for each call on a non-empty interval).

Exercise 17.12 — Recursive algorithms (E12)

Consider the following **ternary search** algorithm, which splits the interval $a[\text{left}..\text{right}]$ into three parts using two indices m_1 and m_2 .

Algorithm: TernarySearch

Input : An array $a[1..n]$ sorted in non-decreasing order, two indices left , right , a target value m

Output: An index i with $a[i] = m$, or -1 if m is not in $a[\text{left}..\text{right}]$

```

1 function TernarySearch( $a$ ,  $\text{left}$ ,  $\text{right}$ ,  $m$ )
2   if  $\text{left} > \text{right}$  then
3     | return  $-1$ 
4   end
5    $d := \lfloor (\text{right} - \text{left}) / 3 \rfloor$ 
6    $m_1 := \text{left} + d$ 
7    $m_2 := \text{right} - d$ 
8   if  $a[m_1] = m$  then
9     | return  $m_1$ 
10  end
11  if  $a[m_2] = m$  then
12    | return  $m_2$ 
13  end
14  if  $m < a[m_1]$  then
15    | return TernarySearch( $a$ ,  $\text{left}$ ,  $m_1 - 1$ ,  $m$ )
16  end
17  if  $m > a[m_2]$  then
18    | return TernarySearch( $a$ ,  $m_2 + 1$ ,  $\text{right}$ ,  $m$ )
19  end
20  return TernarySearch( $a$ ,  $m_1 + 1$ ,  $m_2 - 1$ ,  $m$ )
21 end

```

Questions

1. Let $n = \text{right} - \text{left} + 1 \geq 1$. Show that each of the three sub-intervals on which a recursive call can be made has at most $\lfloor n/3 \rfloor$ elements, and write a recurrence for the worst-case number of steps $S(n)$.

2. Solve the recurrence $S(n) = S(n/3) + c$ with the Master Theorem.
3. Show that the number of calls on a non-empty interval is at most $\lfloor \log_3 n \rfloor + 1$. Compute this bound for $n = 10^6$ and compare it with the corresponding bound for the Recursive Binary Search.
4. Each call of the Recursive Binary Search compares m with array elements at most 2 times ($a[mid] = m$ and $a[mid] < m$), while each call of the ternary search does it at most 4 times. Compare the maximum number of comparisons of the two algorithms for $n = 10^6$ and for $n \rightarrow \infty$. Is ternary search faster than binary search?

Solution

1. Let $n = \text{right} - \text{left} + 1 \geq 1$ and write $n - 1 = \text{right} - \text{left} = 3d + s$ with $d = \lfloor (n-1)/3 \rfloor$ and $s \in \{0, 1, 2\}$. The three sub-intervals have the following sizes:
 - $a[\text{left}..m_1 - 1]$: $m_1 - \text{left} = d$ elements;
 - $a[m_2 + 1..\text{right}]$: $\text{right} - m_2 = d$ elements;
 - $a[m_1 + 1..m_2 - 1]$: $m_2 - m_1 - 1 = (n - 1) - 2d - 1 = d + s - 1$ elements (if this number is negative, the interval is empty).

Since $d = \lfloor (n - 1)/3 \rfloor \leq \lfloor n/3 \rfloor$, the first two sizes are at most $\lfloor n/3 \rfloor$. For the middle interval:

- if $s \in \{0, 1\}$, then $d + s - 1 \leq d \leq \lfloor n/3 \rfloor$;
- if $s = 2$, then $n = 3d + 3$, so $\lfloor n/3 \rfloor = d + 1 = d + s - 1$.

Hence every sub-interval has at most $\lfloor n/3 \rfloor$ elements. For example, for $n = 9$: $d = 2$, $s = 2$, and the three sub-intervals have 2, 3, 2 elements, with $\lfloor 9/3 \rfloor = 3$.

Each call executes a constant number of steps besides the recursive call, and makes at most one recursive call. Therefore, the worst-case number of steps satisfies

$$S(n) = S(\lfloor n/3 \rfloor) + c \quad (n \geq 1), \quad S(0) = c_0,$$

for some constants $c, c_0 > 0$.

2. We apply the Master Theorem to $S(n) = S(n/3) + c$:

- **Identify:** $a = 1$, $b = 3$, $f(n) = c$.
- **Compute:** $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$.
- **Compare:** $f(n) = c = \Theta(1) = \Theta(n^{\log_3 1} \log^0 n)$.
- **Verify:** case 2 of the Master Theorem holds with $k = 0$.
- **Conclude:**

$$S(n) = \Theta(n^{\log_3 1} \log^1 n) = \Theta(\log n).$$

Ternary search has the same order of growth as binary search: the base of the logarithm is irrelevant inside Θ , since $\log_3 n = \log_2 n / \log_2 3$.

3. Let n_i be the size of the interval of the i -th call ($n_1 = n$). By question 1, $n_{i+1} \leq \lfloor n_i/3 \rfloor$, and by induction $n_i \leq \lfloor n/3^{i-1} \rfloor$ (using $\lfloor \lfloor x/3^{i-1} \rfloor / 3 \rfloor = \lfloor x/3^i \rfloor$). If the i -th call is on a non-empty interval, then $n_i \geq 1$, so $3^{i-1} \leq n$, i.e., $i-1 \leq \log_3 n$ and, since $i-1$ is an integer,

$$i \leq \lfloor \log_3 n \rfloor + 1.$$

For $n = 10^6$: $3^{12} = 531\,441 \leq 10^6 < 1\,594\,323 = 3^{13}$, hence $\lfloor \log_3 10^6 \rfloor = 12$ and the ternary search makes at most

$$12 + 1 = 13$$

calls on a non-empty interval (and this value is attained for $n = 10^6$). The Recursive Binary Search makes at most $\lfloor \log_2 10^6 \rfloor + 1 = 19 + 1 = 20$ such calls. Ternary search makes fewer calls.

4. Maximum number of comparisons with array elements for $n = 10^6$:

$$\text{binary search: } 2 \cdot 20 = 40, \quad \text{ternary search: } 4 \cdot 13 = 52.$$

For $n \rightarrow \infty$ the two bounds behave as $2 \log_2 n$ and $4 \log_3 n$, and

$$\lim_{n \rightarrow \infty} \frac{4 \log_3 n}{2 \log_2 n} = \frac{4}{2 \log_2 3} = \frac{2}{\log_2 3} \approx 1.26.$$

Although ternary search makes fewer calls, each call is more expensive, and in total it performs about 26% more comparisons than binary search in the worst case. Both algorithms are $\Theta(\log n)$: ternary search is **not** asymptotically faster, and in practice it is slightly slower.

Exercise 17.13 — Recursive algorithms (E13)

Consider the sorted array with duplicates

$$a = [2, 4, 4, 4, 4, 7, 9, 9, 9, 12]$$

and the following algorithm, whose first call is `FirstOccurrence(a, 1, n, m)`.

Algorithm: FirstOccurrence

Input : An array $a[1..n]$ sorted in non-decreasing order, two indices $left$, $right$, a target value m

Output: The smallest index $i \in \{left, \dots, right\}$ with $a[i] = m$, or -1 if m is not in $a[left..right]$

```

1 function FirstOccurrence(a, left, right, m)
2   if left > right then
3     return -1
4   end
5   mid := left + ⌊(right - left)/2⌋
6   if a[mid] < m then
7     return FirstOccurrence(a, mid + 1, right, m)
8   end
9   if a[mid] > m then
10    return FirstOccurrence(a, left, mid - 1, m)
11  end
12  r := FirstOccurrence(a, left, mid - 1, m)           // a[mid] = m
13  if r = -1 then
14    return mid
15  end
16  return r
17 end

```

Questions

1. Apply `RecursiveBinarySearch(a, 1, 10, m)` for $m = 4$ and $m = 9$. Does it return the first occurrence of m ?
2. Apply `FirstOccurrence(a, 1, 10, m)` for $m = 4$, $m = 9$ and $m = 5$. For each search, report one row for each call and the value returned by each call.
3. Prove by strong induction on $k = right - left + 1$ that `FirstOccurrence` is correct.
4. Write the recurrence for the worst-case number of steps $S(n)$ and solve it.
5. A student proposes a different algorithm: find any occurrence i of m with `RecursiveBinarySearch`, then move left while $i > 1$ and $a[i - 1] = m$. Determine its worst-case number of steps.
6. Is `FirstOccurrence` tail recursive?

Solution

1. Search for $m = 4$ with `RecursiveBinarySearch(a, 1, 10, 4)`:

call	$left$	$right$	mid	$a[mid]$	action
1	1	10	5	4	$a[mid] = m \Rightarrow$ Found! Return 5

Search for $m = 9$ with `RecursiveBinarySearch(a, 1, 10, 9)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action
1	1	10	5	4	$a[mid] < m \Rightarrow$ call on $left = 6, right = 10$
2	6	10	8	9	$a[mid] = m \Rightarrow$ Found! Return 8

The returned indices are 5 and 8, while the first occurrences are at index 2 (for $m = 4$) and 7 (for $m = 9$). `RecursiveBinarySearch` returns **an** occurrence, not necessarily the first one.

2. Search for $m = 4$ with `FirstOccurrence(a, 1, 10, 4)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action	returns
1	1	10	5	4	$a[mid] = m \Rightarrow r :=$ call on 1, 4	$r = 2 \Rightarrow 2$
2	1	4	2	4	$a[mid] = m \Rightarrow r :=$ call on 1, 1	$r = -1 \Rightarrow mid = 2$
3	1	1	1	2	$a[mid] < m \Rightarrow$ call on 2, 1	-1
4	2	1	-	-	$left > right$	-1

The returned values are computed from the bottom to the top: call 4 returns -1, call 3 returns -1, call 2 receives $r = -1$ and returns its own $mid = 2$, call 1 receives $r = 2 \neq -1$ and returns 2. The first occurrence of 4 is at index 2.

Search for $m = 9$ with `FirstOccurrence(a, 1, 10, 9)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action	returns
1	1	10	5	4	$a[mid] < m \Rightarrow$ call on 6, 10	7
2	6	10	8	9	$a[mid] = m \Rightarrow r :=$ call on 6, 7	$r = 7 \Rightarrow 7$
3	6	7	6	7	$a[mid] < m \Rightarrow$ call on 7, 7	7
4	7	7	7	9	$a[mid] = m \Rightarrow r :=$ call on 7, 6	$r = -1 \Rightarrow mid = 7$
5	7	6	-	-	$left > right$	-1

The first occurrence of 9 is at index 7.

Search for $m = 5$ with `FirstOccurrence(a, 1, 10, 5)`:

call	<i>left</i>	<i>right</i>	<i>mid</i>	$a[mid]$	action	returns
1	1	10	5	4	$a[mid] < m \Rightarrow$ call on 6, 10	-1
2	6	10	8	9	$a[mid] > m \Rightarrow$ call on 6, 7	-1
3	6	7	6	7	$a[mid] > m \Rightarrow$ call on 6, 5	-1
4	6	5	-	-	$left > right$	-1

Element 5 is not in the array (return -1).

3. We prove that, for $1 \leq \text{left} \leq \text{right} + 1 \leq n + 1$, **FirstOccurrence**($a, \text{left}, \text{right}, m$) returns the smallest index $i \in \{\text{left}, \dots, \text{right}\}$ with $a[i] = m$, or -1 if there is none. Strong induction on $k = \text{right} - \text{left} + 1$.

Base case $k = 0$: the interval is empty, m does not occur, and the algorithm returns -1 .

Inductive step $k \geq 1$: assume the statement for all sizes $k' < k$. We have $\text{left} \leq \text{mid} \leq \text{right}$, and every recursive call is on an interval of size $\text{right} - \text{mid}$ or $\text{mid} - \text{left}$, both smaller than k .

- If $a[\text{mid}] < m$: all the elements $a[\text{left}..\text{mid}]$ are $< m$ (sorted array), so the occurrences of m in $a[\text{left}..\text{right}]$ are exactly those in $a[\text{mid} + 1..\text{right}]$, and the smallest one is returned by the recursive call (inductive hypothesis).
- If $a[\text{mid}] > m$: all the elements $a[\text{mid}..\text{right}]$ are $> m$, so the occurrences are exactly those in $a[\text{left}..\text{mid} - 1]$, and we conclude as above.
- If $a[\text{mid}] = m$: mid is an occurrence, so the first occurrence is either in $a[\text{left}..\text{mid} - 1]$ or it is mid . By the inductive hypothesis, r is the first occurrence in $a[\text{left}..\text{mid} - 1]$, or -1 if there is none. In the first case the algorithm returns r (line 16), in the second case it returns mid (line 14). Both are correct.

4. Let $n = \text{right} - \text{left} + 1$. Each call executes a constant number of steps (at most 7: lines 2, 5, 6, 9, 12, 13 and 14 or 16) and makes **at most one** recursive call, on an interval of $\lfloor (n - 1)/2 \rfloor$ or $\lfloor n/2 \rfloor$ elements. Hence

$$S(n) = S(\lfloor n/2 \rfloor) + c \quad (n \geq 1), \quad S(0) = c_0.$$

By case 2 of the Master Theorem ($a = 1$, $b = 2$, $f(n) = c = \Theta(n^{\log_2 1} \log^0 n)$, $k = 0$):

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

As for **RecursiveBinarySearch**, there are at most $\lfloor \log_2 n \rfloor + 1$ calls on a non-empty interval (here $\lfloor \log_2 10 \rfloor + 1 = 4$; the search of $m = 9$ attains it). Note that **FirstOccurrence** never stops when it finds m : the recursion always ends with a call on an empty interval.

5. Consider the array $a[1..n]$ with all elements equal to m . **RecursiveBinarySearch** immediately returns $i = \text{mid} = 1 + \lfloor (n - 1)/2 \rfloor$. Then the algorithm moves left $i - 1 = \lfloor (n - 1)/2 \rfloor$ times, until $i = 1$. Each move costs a constant number of steps, so the worst-case number of steps is

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

The proposed algorithm is linear in the worst case, while **FirstOccurrence** is logarithmic.

6. No. The recursive calls of lines 7 and 10 are tail calls, but the call of line 12 is not: after it terminates, the algorithm still has to test $r = -1$ (line 13). The algorithm can be made tail recursive by passing the best candidate found so far as an extra parameter *best* (initially -1): when $a[\text{mid}] = m$, it calls **FirstOccurrence**($a, \text{left}, \text{mid} - 1, m, \text{mid}$), and the empty interval returns *best* instead of -1 .

Exercise 17.14 — Recursive algorithms (E14)

Consider the array of $n = 5$ integer numbers

$$a = [2, -4, 3, -1, 2]$$

and the algorithm `BruteForceMaximumSubarray(a, n)`, which examines all the pairs (l, r) with $1 \leq l \leq r \leq n$ in the order $(1, 1), (1, 2), \dots, (1, n), (2, 2), \dots, (n, n)$, starting from $best = a[1]$ and updating $best, bl, br$ only when $sum > best$. Its number of steps on an array of n elements, as a function of the number U of updates of $best$, is recalled in the introduction to the exercises.

Questions

1. How many subarrays does a have?
2. Apply the brute force algorithm to a . For each pair (l, r) , report the sum of $a[l..r]$ and whether $best$ is updated. Which triple $(bl, br, best)$ is returned?
3. Compute the exact number of steps executed by the algorithm on a .
4. For $n = 1000$, give a lower bound and an upper bound on the number of steps executed by the algorithm on any input of size n .

Solution

1. The subarrays are the pairs (l, r) with $1 \leq l \leq r \leq 5$:

$$\frac{n(n+1)}{2} = \frac{5 \cdot 6}{2} = 15.$$

2. Initially $best = a[1] = 2, bl = br = 1$.

l	r	sum of $a[l..r]$	action
1	1	2	$sum \leq best = 2$
1	2	-2	
1	3	1	
1	4	0	
1	5	2	$sum \leq best = 2$
2	2	-4	
2	3	-1	
2	4	-2	
2	5	0	
3	3	3	$sum > best \Rightarrow best = 3, bl = 3, br = 3$
3	4	2	
3	5	4	$sum > best \Rightarrow best = 4, bl = 3, br = 5$
4	4	-1	
4	5	1	
5	5	2	

The algorithm returns (3, 5, 4): the maximum subarray is $a[3..5] = [3, -1, 2]$ with sum 4. Note that the whole array has sum 2 and the subarray $a[1..1]$ has sum 2: since the comparison is strict, they never cause an update.

3. There are $U = 2$ updates (pairs (3, 3) and (3, 5)). With $n = 5$:

$$2 \cdot 5^2 + 7 \cdot 5 + 6 + 3 \cdot 2 = 50 + 35 + 6 + 6 = 97 \text{ steps.}$$

4. The number of updates satisfies $0 \leq U \leq n(n+1)/2$. Hence the number of steps is at least

$$2 \cdot 1000^2 + 7 \cdot 1000 + 6 = 2\,007\,006$$

and at most

$$2\,007\,006 + 3 \cdot \frac{1000 \cdot 1001}{2} = 2\,007\,006 + 1\,501\,500 = 3\,508\,506.$$

In both cases the number of steps is about 2–3.5 million: the algorithm is $\Theta(n^2)$ on every input, not only in the worst case.

Exercise 17.15 — Recursive algorithms (E15)

Consider the array of $n = 8$ integer numbers

$$a = [-1, 3, -2, 4, -6, 1, 5, -3]$$

and the algorithm $\text{FindMaxCrossingSubarray}(a, l, mid, r)$, which scans $i = mid, mid - 1, \dots, l$ computing the sums of $a[i..mid]$, and $j = mid + 1, \dots, r$ computing the sums of $a[mid + 1..j]$, keeping the best of each (strict comparison, initial values $-\infty$).

Questions

1. Compute $\text{FindMaxCrossingSubarray}(a, 1, 4, 8)$ step by step.
2. Compute $\text{FindMaxCrossingSubarray}(a, 5, 6, 8)$ step by step.
3. Is the subarray found in question 1 a maximum subarray of a ? Justify the answer.
4. How many iterations do the loops execute in questions 1 and 2? Why is the subarray returned in question 1 forced to contain $a[4]$ and $a[5]$, even if $a[5] = -6$?

Solution

1. $l = 1, mid = 4, r = 8$.

Left loop (i from 4 down to 1):

i	$a[i]$	sum	$leftsum$	$maxleft$
4	4	4	4	4
3	-2	2	4	4
2	3	5	5	2
1	-1	4	5	2

Right loop (j from 5 to 8):

j	$a[j]$	sum	$rightsum$	$maxright$
5	-6	-6	-6	5
6	1	-5	-5	6
7	5	0	0	7
8	-3	-3	0	7

The algorithm returns $(maxleft, maxright, leftsum + rightsum) = (2, 7, 5 + 0) = (2, 7, 5)$: the subarray $a[2..7] = [3, -2, 4, -6, 1, 5]$ with sum 5.

2. $l = 5, mid = 6, r = 8$.

i	$a[i]$	sum	$leftsum$	$maxleft$
6	1	1	1	6
5	-6	-5	1	6
j	$a[j]$	sum	$rightsum$	$maxright$
7	5	5	5	7
8	-3	2	5	7

The algorithm returns $(6, 7, 1 + 5) = (6, 7, 6)$: the subarray $a[6..7] = [1, 5]$ with sum 6.

3. No. The subarray $a[6..7]$ found in question 2 has sum $6 > 5$. In fact, $a[6..7]$ is a maximum subarray of a : the maximum subarray lies entirely in the right half $a[5..8]$ and does **not** cross the midpoint $mid = 4$.

FindMaxCrossingSubarray only finds the best subarray **among those crossing the midpoint**. This is why **FindMaximumSubarray** also solves recursively the two halves and returns the best of the three candidates: here the left half $a[1..4]$ gives $a[2..4]$ with sum $3 - 2 + 4 = 5$, the right half $a[5..8]$ gives $a[6..7]$ with sum 6, the crossing subarray has sum 5, and the answer is $a[6..7]$ with sum 6.

4. In question 1 the loops execute $(mid - l + 1) + (r - mid) = 4 + 4 = 8 = r - l + 1$ iterations; in question 2 they execute $2 + 2 = 4 = r - l + 1$ iterations. In general, **FindMaxCrossingSubarray** executes exactly $n = r - l + 1$ iterations, hence $\Theta(n)$ steps.

By definition, a subarray crossing the midpoint is $a[i..j]$ with $i \leq mid < j$: it always contains both $a[mid]$ and $a[mid + 1]$. Indeed, the left loop starts from $i = mid$ and the right loop from $j = mid + 1$, and both *leftsum* and *rightsum* are updated at the first iteration (initial values $-\infty$). The subarrays that do not contain $a[5] = -6$ are considered by the recursive calls on the two halves.

Exercise 17.16 — Recursive algorithms (E16)

Consider the array of $n = 8$ integer numbers

$$a = [-2, 4, -1, 3, -5, 2, 6, -4]$$

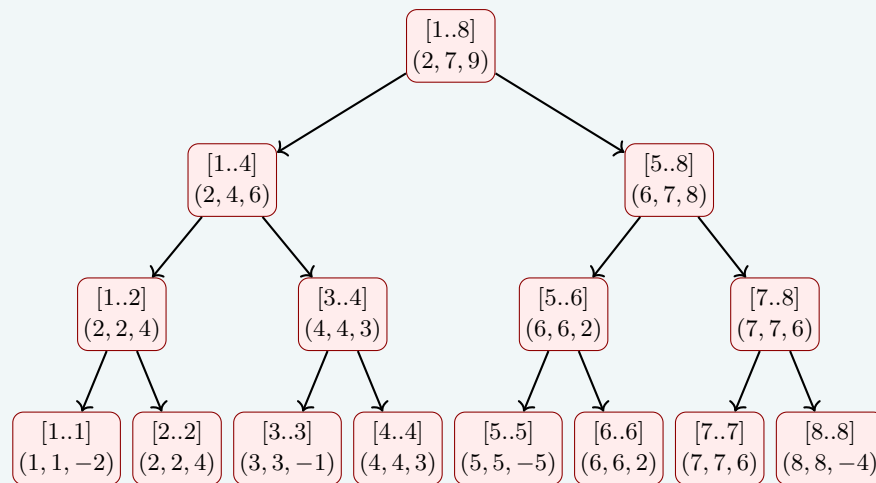
and the divide and conquer algorithm **FindMaximumSubarray**(a, l, r), with first call **FindMaximumSubarray**($a, 1, 8$), $mid = \lfloor (l + r)/2 \rfloor$, and ties broken in favour of the left half, then of the right half.

Questions

1. Draw the tree of calls, indicating for each call $[l..r]$ the triple (i, j, s) it returns.
2. For each call on at least two elements, report the triples returned by the two recursive calls and by **FindMaxCrossingSubarray**, and the returned triple.
3. Show the computation of **FindMaxCrossingSubarray** in the first call.
4. How many calls of **FindMaximumSubarray** are executed? How many iterations are executed, in total, by the loops of all the calls of **FindMaxCrossingSubarray**? Compare with $n \log_2 n$.

Solution

1. Tree of calls (each node: the sub-array $[l..r]$ and the returned triple (i, j, s)):



2. Results of the calls on at least two elements (left, right and crossing candidates, and the returned triple):

call	<i>mid</i>	left	right	crossing	returned
[1..8]	4	(2, 4, 6)	(6, 7, 8)	(2, 7, 9)	(2, 7, 9)
[1..4]	2	(2, 2, 4)	(4, 4, 3)	(2, 4, 6)	(2, 4, 6)
[5..8]	6	(6, 6, 2)	(7, 7, 6)	(6, 7, 8)	(6, 7, 8)
[1..2]	1	(1, 1, -2)	(2, 2, 4)	(1, 2, 2)	(2, 2, 4)
[3..4]	3	(3, 3, -1)	(4, 4, 3)	(3, 4, 2)	(4, 4, 3)
[5..6]	5	(5, 5, -5)	(6, 6, 2)	(5, 6, -3)	(6, 6, 2)
[7..8]	7	(7, 7, 6)	(8, 8, -4)	(7, 8, 2)	(7, 7, 6)

For example, in the call $[1..4]$ the crossing subarray is computed as follows: left loop $i = 2$: $sum = 4$; $i = 1$: $sum = 2$, so $leftsum = 4$, $maxleft = 2$; right loop $j = 3$: $sum = -1$; $j = 4$: $sum = 2$, so $rightsum = 2$, $maxright = 4$; result $(2, 4, 6)$. In the call $[5..8]$: $leftsum = 2$ ($maxleft = 6$), $rightsum = 6$ ($maxright = 7$), result $(6, 7, 8)$.

3. First call: $l = 1$, $mid = 4$, $r = 8$.

<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>
4	3	3	3	4	5	-5	-5	-5	5
3	-1	2	3	4	6	2	-3	-3	6
2	4	6	6	2	7	6	3	3	7
1	-2	4	6	2	8	-4	-1	3	7

The crossing subarray is $a[2..7] = [4, -1, 3, -5, 2, 6]$ with sum $6 + 3 = 9$. Since $9 > 6$ (left) and $9 > 8$ (right), the algorithm returns $(2, 7, 9)$: the maximum subarray is $a[2..7]$ with sum 9. Note that it contains the negative element $a[5] = -5$.

4. The tree of calls has $n = 8$ leaves and $n - 1 = 7$ internal nodes: $2n - 1 = 15$ calls of **FindMaximumSubarray**, and 7 calls of **FindMaxCrossingSubarray**.

A call of **FindMaxCrossingSubarray** on $a[l..r]$ executes $r - l + 1$ iterations. The calls of each level of the tree cover the whole array once:

- level 0: $[1..8] \Rightarrow 8$ iterations;
- level 1: $[1..4], [5..8] \Rightarrow 4 + 4 = 8$ iterations;
- level 2: $[1..2], [3..4], [5..6], [7..8] \Rightarrow 2 + 2 + 2 + 2 = 8$ iterations.

In total $3 \cdot 8 = 24$ iterations, which is exactly $n \log_2 n = 8 \cdot 3 = 24$. This is the term $cn \log_2 n$ of the recursion tree, which gives $S(n) = \Theta(n \log n)$.

Exercise 17.17 — Recursive algorithms (E17)

Consider the two arrays

$$a = [-2, 1, -3, 4, -1, 2, 1, -5, 4] \quad \text{and} \quad b = [-8, -3, -6, -2, -5, -4]$$

and the iterative algorithm **Kadane**(a, n): it starts with $best := a[1]$, $bl := br := 1$, $cur := a[1]$, $cl := 1$ and, for $i = 2, \dots, n$, it executes

- if $cur < 0$: $cur \leftarrow a[i]$, $cl \leftarrow i$ (restart); otherwise $cur \leftarrow cur + a[i]$ (extend);
- if $cur > best$: $best \leftarrow cur$, $bl \leftarrow cl$, $br \leftarrow i$.

It returns $(bl, br, best)$.

Questions

1. Apply **Kadane** to the array a , reporting for each i the action executed and the values of cur , cl , $best$, bl , br .
2. Apply **Kadane** to the array b , whose entries are all negative.
3. What do the brute force algorithm and the divide and conquer algorithm **FindMaximumSubarray** return on b ? (For the divide and conquer algorithm, report the three candidates of the first call.) Prove that, if all the entries of an array are negative, every maximum subarray consists of a single element.
4. A popular variant of the algorithm initialises $cur := 0$ and $best := 0$ and, for $i = 1, \dots, n$, executes $cur \leftarrow \max\{0, cur + a[i]\}$ and $best \leftarrow \max\{best, cur\}$. What does it return on a and on b ? Which problem does it solve?

Solution

1. Execution on $a = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$ (row $i = 1$: initialisation; the other rows report the values at the end of the iteration with index i):

i	$a[i]$	action	cur	cl	$best$	(bl, br)
1	-2	initialisation	-2	1	-2	(1, 1)
2	1	restart ($cur = -2 < 0$)	1	2	1	(2, 2)
3	-3	extend	-2	2	1	(2, 2)
4	4	restart ($cur = -2 < 0$)	4	4	4	(4, 4)
5	-1	extend	3	4	4	(4, 4)
6	2	extend	5	4	5	(4, 6)
7	1	extend	6	4	6	(4, 7)
8	-5	extend	1	4	6	(4, 7)
9	4	extend	5	4	6	(4, 7)

The algorithm returns (4, 7, 6): the maximum subarray is $a[4..7] = [4, -1, 2, 1]$ with sum 6.

2. Execution on $b = [-8, -3, -6, -2, -5, -4]$:

i	$b[i]$	action	cur	cl	$best$	(bl, br)
1	-8	initialisation	-8	1	-8	(1, 1)
2	-3	restart	-3	2	-3	(2, 2)
3	-6	restart	-6	3	-3	(2, 2)
4	-2	restart	-2	4	-2	(4, 4)
5	-5	restart	-5	5	-2	(4, 4)
6	-4	restart	-4	6	-2	(4, 4)

Since cur is always negative, the algorithm restarts at every iteration: $cur = b[i]$, and $best$ is simply the maximum element seen so far. The algorithm returns (4, 4, -2): the maximum subarray is the single element $b[4] = -2$.

3. The brute force algorithm examines all the subarrays and returns (4, 4, -2) as well.

The divide and conquer algorithm, in the first call [1..6] with $mid = 3$, obtains:

- left half [1..3]: (2, 2, -3);
- right half [4..6]: (4, 4, -2);
- crossing subarray: left loop $i = 3$: $sum = -6$, $i = 2$: -9 , $i = 1$: -17 , so $leftsum = -6$ ($maxleft = 3$); right loop $j = 4$: $sum = -2$, $j = 5$: -7 , $j = 6$: -11 , so $rightsum = -2$ ($maxright = 4$); result (3, 4, -8).

The best of the three is (4, 4, -2). All three algorithms return $b[4..4]$ with sum -2.

Proof of the property. Let all entries be negative, and let $a[l..r]$ be a subarray with at least two elements ($l < r$). Then

$$\sum_{k=l}^r a[k] = a[l] + \sum_{k=l+1}^r a[k] < a[l],$$

because $\sum_{k=l+1}^r a[k]$ is a sum of negative numbers. Hence the single-element subarray $a[l..l]$ has a larger sum, and $a[l..r]$ is not a maximum subarray. Thus every maximum subarray has one element, and it is a maximum element of the array.

4. The variant computes the maximum sum of a subarray **allowing the empty subarray** (whose sum is 0): resetting *cur* to 0 means “start a new, empty subarray”.

- On *a*: *cur* takes the values 0, 1, 0, 4, 3, 5, 6, 1, 5, and the algorithm returns *best* = 6, as before, since the maximum subarray has a positive sum.
- On *b*: *cur* is reset to 0 at every iteration, and the algorithm returns *best* = 0, which is the sum of the empty subarray. This is **not** a correct answer to our problem, where a subarray must be non-empty ($l \leq r$): the correct answer is -2 .

In the stock interpretation, the variant answers the question “what is the maximum profit if we are also allowed **not to buy** at all?”: if the price decreases every day, the best choice is not to buy (profit 0). The two versions coincide whenever the array contains at least one non-negative entry.

Recursive versus iterative algorithms

Exercise 18.1 — Recursive versus iterative algorithms (E1)

Consider the following two recursive algorithms.

Algorithm: GCDRec

Input : Two natural numbers a and b , not both equal to 0

Output: The greatest common divisor of a and b

```
1 function GCDRec( $a, b$ )
2   if  $b = 0$  then
3     return  $a$ 
4   end
5   return GCDRec( $b, a \bmod b$ )
6 end
```

Algorithm: RecBinarySearch**Input** : A sorted sub-array $a[l..r]$ and a target value m **Output**: An index of m in $a[l..r]$, or -1 if m is not present

```

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

```

Questions

1. Are GCDRec and RecBS tail recursive? Justify the answer.
2. Transform GCDRec into an equivalent iterative algorithm GCDIt, applying the mechanical elimination of the tail recursion step by step. Then execute both algorithms with $(a, b) = (1071, 462)$, listing the values of the parameters in each call and in each iteration.
3. Transform RecBS into an equivalent iterative algorithm ItBS. Then execute both algorithms on the array $a = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]$ (with $l = 1, r = 10$) for $m = 23$ and for $m = 4$.
4. Compare the recursive and the iterative versions of the two algorithms in terms of time and space. In particular, determine the depth of the recursion of GCDRec(89, 55) and the maximum depth of the recursion of RecBS on an array of $n = 10^6$ elements.

Solution

1. Both algorithms are tail recursive.
 - In GCDRec the only recursive call is at line 5: its result is returned as it is, without any further operation.
 - In RecBS the recursive calls are at lines 10 and 12: in both cases the result of the call is returned as it is.

Hence, after each recursive call, nothing remains to be done in the calling function:

its frame is not needed anymore, and the call can be replaced by an update of the parameters.

2. We apply the three rules of the mechanical elimination of tail recursion.

- The base-case condition is $b = 0$ (line 2): the loop condition is $b \neq 0$.
- The tail call $\text{GCDRec}(b, a \bmod b)$ becomes the simultaneous assignment $(a, b) \leftarrow (b, a \bmod b)$. Since the new value of b must be computed with the old value of a , we use a temporary variable r .
- The return of the base case (**return** a , line 3) is moved after the loop.

Algorithm: GCDIt

```

1 function GCDIt( $a, b$ )
2   while  $b \neq 0$  do
3      $r := a \bmod b$ 
4      $a \leftarrow b$ 
5      $b \leftarrow r$ 
6   end
7   return  $a$ 
8 end

```

Execution with $(a, b) = (1071, 462)$:

GCDRec				GCDIt			
call	a	b	$a \bmod b$	iteration	a	b	r
1	1071	462	147	1	1071	462	147
2	462	147	21	2	462	147	21
3	147	21	0	3	147	21	0
4	21	0	–	end	21	0	–

Both algorithms go through the same pairs $(1071, 462)$, $(462, 147)$, $(147, 21)$, $(21, 0)$ and return 21. **GCDRec** executes 4 calls (all active at the same time when the base case is reached); **GCDIt** executes 3 iterations.

3. The base-case condition $l > r$ becomes the loop condition $l \leq r$; the second base case $(a[\text{mid}] = m)$ stays inside the loop as a return; the tail call at line 10 becomes $l \leftarrow \text{mid} + 1$ and the tail call at line 12 becomes $r \leftarrow \text{mid} - 1$ (the parameters a and m do not change); the return of -1 is moved after the loop.

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

```

The two algorithms go through the same values of (l, r) ; for **RecBS** each row is a call, for **ItBS** each row is an iteration (the last row with $l > r$ is the final test of the loop).

- $m = 23$:

call / iteration	l	r	mid	$a[mid]$
1	1	10	5	$16 < 23 \Rightarrow l = 6$
2	6	10	8	$56 > 23 \Rightarrow r = 7$
3	6	7	6	$23 = m \Rightarrow \text{return } 6$

- $m = 4$:

call / iteration	l	r	mid	$a[mid]$
1	1	10	5	$16 > 4 \Rightarrow r = 4$
2	1	4	2	$5 > 4 \Rightarrow r = 1$
3	1	1	1	$2 < 4 \Rightarrow l = 2$
4	2	1	–	$l > r \Rightarrow \text{return } -1$

For $m = 23$ **RecBS** executes 3 calls and **ItBS** 3 iterations; for $m = 4$ **RecBS** executes 4 calls and **ItBS** 3 iterations plus the final test.

4. **Time.** The iterative versions go through exactly the same values of the parameters as the recursive ones, with a constant number of steps for each call / iteration. Hence the number of steps has the same order of growth: $\Theta(\log \min(a, b))$ in the worst case for the GCD and $\Theta(\log_2 n)$ in the worst case for the binary search. The recursive versions only pay the (constant) overhead of the calls.

Space. The iterative versions use $\Theta(1)$ extra memory (the parameters and the variables r or mid). In the recursive versions all the calls stay active until the base case is reached, so the stack contains as many frames as the depth of the recursion.

- $\text{GCDRec}(89, 55)$ goes through the pairs

$(89, 55), (55, 34), (34, 21), (21, 13), (13, 8), (8, 5), (5, 3), (3, 2), (2, 1), (1, 0),$

so the depth of the recursion is 10. Consecutive Fibonacci numbers are the worst case of Euclid's algorithm: the depth is $\Theta(\log \min(a, b))$.

- For RecBS on n elements the maximum depth is $\lfloor \log_2 n \rfloor + 2$ (the sizes of the sub-arrays are at most $n, \lfloor n/2 \rfloor, \dots, 1$, plus the final call on the empty sub-array). For $n = 10^6$: $\lfloor \log_2 10^6 \rfloor + 2 = 19 + 2 = 21$.

For both algorithms, recursion gives **no gain** in time and a (small, logarithmic) **loss** in space. Since they are tail recursive, the recursion can be eliminated mechanically, obtaining loops with $\Theta(1)$ extra memory.

Exercise 18.2 — Recursive versus iterative algorithms (E2)

Consider the following recursive algorithms and input sizes.

- $\text{Fact}(n)$: returns 1 if $n = 0$, otherwise $n \cdot \text{Fact}(n - 1)$; with $n = 1000$.
- $\text{SumDecrease}(a, n)$: returns 0 if $n = 0$, otherwise $\text{SumDecrease}(a, n - 1) + a[n]$; with $n = 10^6$.
- $\text{SumHalving}(a, l, r)$: returns $a[l]$ if $l = r$, otherwise $\text{SumHalving}(a, l, \text{mid}) + \text{SumHalving}(a, \text{mid} + 1, r)$ with $\text{mid} = \lfloor (l + r)/2 \rfloor$; on an array of $n = 10^6$ elements.
- Merge Sort on an array of $n = 1000$ elements (base case: sub-arrays of one element).
- Tower of Hanoi with $n = 20$ disks (base case: 0 disks).
- $\text{FibRec}(n)$: returns n if $n \leq 1$, otherwise $\text{FibRec}(n - 1) + \text{FibRec}(n - 2)$; with $n = 30$.
- Recursive fast power x^n (base case: $n = 0$; recursive call on $\lfloor n/2 \rfloor$); with $n = 10^9$.

Assume that every stack frame occupies 48 bytes and that the call stack has a size of 8 MB $= 8 \cdot 2^{20} = 8\,388\,608$ bytes.

Questions

1. Determine the depth of the recursion of each algorithm for the given input size.
2. Determine the memory occupied by the call stack at its maximum, and say which executions cause a stack overflow.
3. Determine the largest n for which $\text{SumDecrease}(a, n)$ does not cause a stack overflow, and the depth of SumHalving for the same n .
4. For $\text{FibRec}(30)$, compare the depth of the recursion with the total number of calls. Is the problem of this algorithm the space or the time?

Solution

1. The depth of the recursion is the maximum number of calls active at the same time, i.e., the number of calls on the longest path from the root to a leaf of the recursion tree.
 - (a) The active calls are **Fact**(1000), **Fact**(999), ..., **Fact**(0): depth $n + 1 = 1001$.
 - (b) The active calls are **SumDecrease**($a, 10^6$), ..., **SumDecrease**($a, 0$): depth $n + 1 = 1\,000\,001$.
 - (c) The largest sub-array at level t of the tree has $\lceil n/2^t \rceil$ elements, which becomes 1 for $t = \lceil \log_2 n \rceil$. Depth $\lceil \log_2 n \rceil + 1$; since $2^{19} = 524\,288 < 10^6 \leq 2^{20} = 1\,048\,576$, we have $\lceil \log_2 10^6 \rceil = 20$ and the depth is 21.
 - (d) As in (c): $\lceil \log_2 1000 \rceil + 1 = 10 + 1 = 11$ (since $2^9 = 512 < 1000 \leq 2^{10} = 1024$).
 - (e) The active calls are **Hanoi**(20), **Hanoi**(19), ..., **Hanoi**(0): depth $n + 1 = 21$.
 - (f) The longest path is **FibRec**(30), **FibRec**(29), ..., **FibRec**(1): depth $n = 30$.
 - (g) The exponent goes through the values $n, \lfloor n/2 \rfloor, \dots, 1, 0$, i.e., the depth is $\lfloor \log_2 n \rfloor + 2$. Since $2^{29} = 536\,870\,912 \leq 10^9 < 2^{30}$, we have $\lfloor \log_2 10^9 \rfloor = 29$ and the depth is 31.
2. The memory of the stack at its maximum is 48 bytes times the depth:

	algorithm	depth	stack (bytes)	overflow?
(a)	Fact (1000)	1 001	48 048	no
(b)	SumDecrease , $n = 10^6$	1 000 001	48 000 048	yes
(c)	SumHalving , $n = 10^6$	21	1 008	no
(d)	Merge Sort, $n = 1000$	11	528	no
(e)	Hanoi , $n = 20$	21	1 008	no
(f)	FibRec (30)	30	1 440	no
(g)	fast power, $n = 10^9$	31	1 488	no

Only (b) causes a stack overflow: $48\,000\,048 > 8\,388\,608$ bytes (about 46 MB against 8 MB).

3. The stack contains at most $\lfloor 8\,388\,608/48 \rfloor = 174\,762$ frames. **SumDecrease**(a, n) needs $n + 1$ frames, so it does not overflow if and only if

$$n + 1 \leq 174\,762 \iff n \leq 174\,761.$$

For $n = 174\,761$, since $2^{17} = 131\,072 < n \leq 2^{18} = 262\,144$, **SumHalving** has depth $\lceil \log_2 n \rceil + 1 = 18 + 1 = 19$, i.e., 912 bytes of stack. Both algorithms execute $\Theta(n)$ steps: the halving version uses an exponentially smaller stack.

4. `FibRec(30)` has depth 30 (only 1 440 bytes of stack), but it executes

$$2F(31) - 1 = 2 \cdot 1\,346\,269 - 1 = 2\,692\,537$$

calls. The problem of `FibRec` is the **time**, $\Theta(\varphi^n)$, not the space, $\Theta(n)$: the depth of the recursion and the number of calls are two very different quantities.

A depth of the recursion $\Theta(\log_2 n)$ is never a problem in practice; a depth $\Theta(n)$ (decrease by one) causes a stack overflow already for n of the order of 10^5 – 10^6 , while the corresponding loop uses $\Theta(1)$ memory.

Exercise 18.3 — Recursive versus iterative algorithms (E3)

Consider the following four recursive algorithms.

Algorithm: A: MaxDecrease

```

1 function MaxDec(a, n)
2   if n = 1 then
3     return a[1]
4   end
5   return max(MaxDec(a, n - 1), a[n])
6 end
```

Algorithm: B: MaxHalving

```

1 function MaxHalf(a, l, r)
2   if l = r then
3     return a[l]
4   end
5   mid := ⌊(l + r)/2⌋
6   return max(MaxHalf(a, l, mid),
              MaxHalf(a, mid + 1, r))
7 end
```

Algorithm: C: PowerDecrease

```

1 function PowDec(x, n)
2   if n = 0 then
3     return 1
4   end
5   return x · PowDec(x, n - 1)
6 end
```

Algorithm: D: Tribonacci

```

1 function Trib(n)
2   if n ≤ 2 then
3     return ⌊n/2⌋
4   end
5   return Trib(n − 1) + Trib(n − 2)
      + Trib(n − 3)
6 end

```

MaxDec and MaxHalf compute the maximum of an array $a[1..n]$ (MaxHalf is called as MaxHalf($a, 1, n$)); PowDec computes x^n ; Trib computes the Tribonacci numbers $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n-1) + T(n-2) + T(n-3)$.

Questions For each of the four algorithms:

1. write the recurrence of the number of steps $S(n)$ and determine its order of growth;
2. determine the depth of the recursion and the extra memory used;
3. indicate the best iterative algorithm seen in the course for the same problem, with its time and space;
4. decide whether recursion gains, is equivalent, or loses, justifying the answer.

Solution**A: MaxDecrease**

1. Each call executes a constant number c of steps besides the recursive call on $n-1$ elements:

$$S(n) = S(n-1) + c, \quad S(1) = d.$$

Unrolling, $S(n) = d + c(n-1) = \Theta(n)$.

2. The calls MaxDec(a, n), ..., MaxDec($a, 1$) are all active at the same time: depth n , extra memory $\Theta(n)$. The algorithm is not tail recursive (the maximum is computed after the call returns).
3. FindMaximum (chapter «Running time»): a loop with $S(n) = 4n - 1 = \Theta(n)$ steps and $\Theta(1)$ extra memory.
4. **SAME** time, **LOSS** in space: the decrease-by-one recursion is a loop in disguise, with a call overhead and a risk of stack overflow for large n .

B: MaxHalving

1. Two recursive calls on halves, plus a constant number of steps:

$$S(n) = 2S(n/2) + \Theta(1).$$

Master Theorem: $a = 2$, $b = 2$, $f(n) = \Theta(1)$; $n^{\log_b a} = n^{\log_2 2} = n$. Since $f(n) = \Theta(1) = O(n^{1-\varepsilon})$ with $\varepsilon = 1$, case 1 gives $S(n) = \Theta(n)$.

2. Depth $\lceil \log_2 n \rceil + 1$, extra memory $\Theta(\log_2 n)$.
3. Again **FindMaximum**: $\Theta(n)$ time, $\Theta(1)$ space.
4. **SAME** time, small **LOSS** in space ($\Theta(\log_2 n)$). Divide and conquer does not help here: the recursion tree has $2n - 1$ nodes, each with constant work, so the total work is still $\Theta(n)$. No algorithm can do better, because every element must be examined at least once. Divide and conquer gives a gain only when it reduces the total work, as in Merge Sort, where the combination of the halves costs $\Theta(n)$ instead of the $\Theta(n^2)$ of the iterative algorithms.

C: PowerDecrease

1. $S(n) = S(n - 1) + c$, $S(0) = d \Rightarrow S(n) = cn + d = \Theta(n)$.
2. Depth $n + 1$ (calls with exponent $n, n - 1, \dots, 0$), extra memory $\Theta(n)$.
3. The loop with n multiplications has $\Theta(n)$ time and $\Theta(1)$ space; the iterative square-and-multiply algorithm **PowerIt** has $\Theta(\log_2 n)$ time and $\Theta(1)$ space.
4. **LOSS**: same time as the simple loop and $\Theta(n)$ memory; compared with the square-and-multiply loop it is even exponentially slower in terms of $\log_2 n$. Recursion is convenient for the power only when it **halves** the exponent (fast power): $S(n) = S(n/2) + \Theta(1) = \Theta(\log_2 n)$.

D: Tribonacci

1. Let $C(n)$ be the number of calls; then $C(0) = C(1) = C(2) = 1$ and

$$C(n) = 1 + C(n - 1) + C(n - 2) + C(n - 3), \quad n \geq 3,$$

and $S(n) = \Theta(C(n))$, since each call executes a constant number of steps. The function C is non-decreasing, so for $n \geq 3$

$$C(n) \geq 3C(n - 3) \geq 3^2C(n - 6) \geq \dots \geq 3^{\lfloor n/3 \rfloor}.$$

Hence $S(n) = \Omega(3^{n/3}) = \Omega(1.442^n)$: the algorithm is exponential. (For example $C(10) = 289$, $C(20) = 128\,287$, $C(25) = 2\,700\,421$.) In the same way, $C(n) \leq 3C(n - 1) + 1$ gives $S(n) = O(3^n)$.

2. The longest path in the recursion tree is **Trib**(n), **Trib**($n - 1$), $\dots$, **Trib**(2): depth $n - 1$, extra memory $\Theta(n)$.
3. A loop that computes $T(3), T(4), \dots, T(n)$ keeping only the last three values in three variables: $\Theta(n)$ time and $\Theta(1)$ space.

4. Huge **LOSS**: the subproblems **overlap** (for example $\text{Trib}(n-1)$ and $\text{Trib}(n-2)$ both call $\text{Trib}(n-3)$), so the same values are recomputed an exponential number of times. Memoization or the bottom-up loop (Dynamic Programming) bring the time back to $\Theta(n)$.

	recursive		best iterative	verdict
	time	space	time, space	
A	$\Theta(n)$	$\Theta(n)$	$\Theta(n), \Theta(1)$	same time, loss in space
B	$\Theta(n)$	$\Theta(\log_2 n)$	$\Theta(n), \Theta(1)$	same time, small loss
C	$\Theta(n)$	$\Theta(n)$	$\Theta(\log_2 n), \Theta(1)$	loss
D	exponential	$\Theta(n)$	$\Theta(n), \Theta(1)$	huge loss

Exercise 18.4 — Recursive versus iterative algorithms (E4)

The binomial coefficients satisfy, for $0 < k < n$, the recurrence (Pascal's rule)

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad \binom{n}{0} = \binom{n}{n} = 1.$$

Consider the following recursive algorithm.

Algorithm: BinomRec

Input : Two natural numbers n and k with $0 \leq k \leq n$

Output: The binomial coefficient $\binom{n}{k}$

```

1 function BinomRec( $n, k$ )
2   if  $k = 0$  or  $k = n$  then
3     return 1
4   end
5   return BinomRec( $n-1, k-1$ ) + BinomRec( $n-1, k$ )
6 end
```

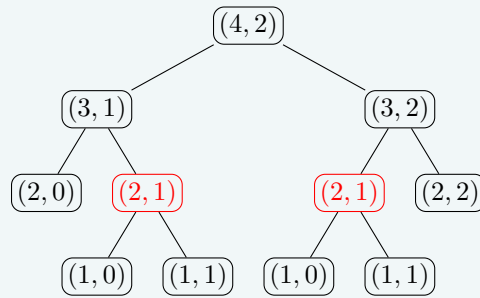
Questions

1. Draw the recursion tree of $\text{BinomRec}(4, 2)$, count the calls and indicate the subproblems that are solved more than once.
2. Prove that the number of calls executed by $\text{BinomRec}(n, k)$ is $N(n, k) = 2\binom{n}{k} - 1$.
3. Compute $N(10, 5)$, $N(30, 15)$ and $N(40, 20)$, and estimate the time needed by $\text{BinomRec}(40, 20)$ on a computer that executes 10^9 calls per second.

4. Write a memoized version **BinomMemo** and prove that, for $0 < k < n$, it executes exactly $2k(n - k) + 1$ calls. Compute this number for the three cases of question 3.
5. Write an iterative algorithm that computes $\binom{n}{k}$ row by row (Pascal's triangle), and give its time and space.

Solution

1. Each node is a call, labelled with (n, k) ; the leaves are base cases ($k = 0$ or $k = n$) and return 1.



The tree has 11 nodes, i.e., 11 calls, and $6 = \binom{4}{2}$ leaves. The subproblem $(2, 1)$ is solved **twice** (and so are its sub-calls $(1, 0)$ and $(1, 1)$).

2. Every call is either a leaf (base case, returns 1) or an internal node with exactly two children. The value returned by a call is the sum of the values returned by the leaves of its sub-tree, i.e., the number of its leaves. Hence the tree of **BinomRec** (n, k) has $\binom{n}{k}$ leaves.

In a tree in which every internal node has exactly two children, the number of internal nodes is the number of leaves minus one (by induction: a single leaf has 0 internal nodes; joining two such trees with L_1 and L_2 leaves under a new root gives $(L_1 - 1) + (L_2 - 1) + 1 = (L_1 + L_2) - 1$ internal nodes). Therefore

$$N(n, k) = \binom{n}{k} + \left(\binom{n}{k} - 1 \right) = 2 \binom{n}{k} - 1.$$

For example $N(4, 2) = 2 \cdot 6 - 1 = 11$, as in question 1.

- 3.

$$N(10, 5) = 2 \cdot 252 - 1 = 503, \quad N(30, 15) = 2 \cdot 155\,117\,520 - 1 = 310\,235\,039,$$

$$N(40, 20) = 2 \cdot 137\,846\,528\,820 - 1 = 275\,693\,057\,639.$$

At 10^9 calls per second, **BinomRec** $(40, 20)$ needs about 275.7 seconds, i.e., more than 4.5 minutes, to compute a number that can be obtained with a few hundred additions (see questions 4 and 5). Since $\binom{2m}{m} \geq \frac{4^m}{2m+1}$, the number of calls grows exponentially.

4. We store the values already computed in a table *memo*, indexed by the pair (i, j) :

Algorithm: BinomMemo

Input : Natural numbers $0 \leq k \leq n$ and a table *memo* whose entries are all *undefined*

Output: The binomial coefficient $\binom{n}{k}$

```

1 function BinomMemo(n, k, memo)
2   if k = 0 or k = n then
3     return 1
4   end
5   if memo[n][k] is undefined then
6     memo[n][k] ←
        BinomMemo(n − 1, k − 1, memo) + BinomMemo(n − 1, k, memo)
7   end
8   return memo[n][k]
9 end

```

Number of calls. Starting from (n, k) , a call reaches the pairs $(n - s, k - t)$, where s is the number of steps down and $t \leq s$ is the number of these steps in which k decreases. Let $u = s - t$ be the number of steps in which k does not decrease. A pair is **internal** (not a base case) if $k - t \geq 1$ and $(n - s) - (k - t) \geq 1$, i.e.,

$$0 \leq t \leq k - 1 \quad \text{and} \quad 0 \leq u \leq n - k - 1,$$

so there are exactly $k(n - k)$ internal subproblems, and all of them are reached. The body of each internal subproblem (the two recursive calls) is executed only the **first** time it is called; the following times the value is read from *memo*. Every call, except the first one, is made by the body of an internal subproblem, which makes exactly 2 calls. Therefore the number of calls is

$$1 + 2k(n - k).$$

For the three cases:

$$(10, 5) : 1 + 2 \cdot 25 = 51, \quad (30, 15) : 1 + 2 \cdot 225 = 451, \quad (40, 20) : 1 + 2 \cdot 400 = 801,$$

against 503, 310 235 039 and 275 693 057 639 calls of **BinomRec**. The time is $\Theta(k(n - k))$ and the memory is $\Theta(k(n - k))$ for the table plus $\Theta(n)$ for the stack (depth $\leq n$).

5. Row i of Pascal's triangle is computed from row $i - 1$. Keeping a single array $c[0..k]$ and updating it from right to left (so that $c[j - 1]$ still contains the value of the previous row when $c[j]$ is updated):

Algorithm: BinomIt**Input** : Two natural numbers n and k with $0 \leq k \leq n$ **Output**: The binomial coefficient $\binom{n}{k}$

```

1 function BinomIt( $n, k$ )
2    $c[0] := 1; c[1..k] := 0$ 
3    $i := 1$ 
4   while  $i \leq n$  do
5      $j := \min(i, k)$ 
6     while  $j \geq 1$  do
7        $c[j] \leftarrow c[j] + c[j - 1]$ 
8        $j \leftarrow j - 1$ 
9     end
10     $i \leftarrow i + 1$ 
11  end
12  return  $c[k]$ 
13 end

```

After the i -th iteration of the outer loop, $c[j] = \binom{i}{j}$ for $0 \leq j \leq \min(i, k)$. The inner loop is executed at most k times for each of the n rows: time $O(nk)$ (and $\Theta(nk)$ when $k \leq n/2$), extra memory $\Theta(k)$, no recursion. For $(40, 20)$ the algorithm executes $\sum_{i=1}^{40} \min(i, 20) = 210 + 400 = 610$ additions.

The naive recursion is exponential because the subproblems overlap; memoization (top-down) and the iterative Pascal's triangle (bottom-up) solve each subproblem once, with a polynomial number of steps. This is the idea of Dynamic Programming.

Exercise 18.5 — Recursive versus iterative algorithms (E5)

Consider the two following algorithms for the Fibonacci numbers ($F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$).

Algorithm: FibRec

```

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

```

Algorithm: FibIt

```

1 function FibIt(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 ← curr
11    curr ← next
12    i ← i + 1
13  end
14  return curr
15 end

```

Assume that every line executed counts as one step (line 5 counts as one step, besides the steps of the two sub-calls) and that the computer executes 10^9 steps per second.

Questions

1. Determine the exact number of steps $S_{it}(n)$ executed by **FibIt** for $n \geq 2$.
2. Determine the exact number of steps $S_{rec}(n)$ executed by **FibRec**, using the fact that **FibRec**(n) executes $2F(n+1) - 1$ calls.
3. Estimate the running times of the two algorithms for $n = 50$, knowing that

$$F(51) = 20\,365\,011\,074.$$

4. Estimate the running time of **FibRec** for $n = 60$ and $n = 100$, knowing that

$$F(61) = 2\,504\,730\,781\,961 \quad \text{and} \quad F(101) \approx 5.73 \cdot 10^{20}.$$

5. Determine the largest n for which **FibRec** terminates within one hour, knowing that

$$F(58) = 591\,286\,729\,879 \quad \text{and} \quad F(59) = 956\,722\,026\,041.$$

Solution

1. For $n \geq 2$ the number of times each line of **FibIt** is executed is:
 - **Line 2** is executed 1 time (the return in the next line is not executed)

- Lines 5, 6, 7 are executed 1 time each
- **Line** 8 is executed n times ($i = 2, 3, \dots, n + 1$)
- **Lines** 9, 10, 11, 12 are executed $n - 1$ times each
- **Line** 14 is executed 1 time

Therefore

$$S_{it}(n) = 1 + 3 + n + 4(n - 1) + 1 = 5n + 1 = \Theta(n).$$

2. Every call of **FibRec** executes exactly 2 steps of its own: line 2 and then either line 3 (base case) or line 5. Since **FibRec**(n) executes $2F(n + 1) - 1$ calls,

$$S_{rec}(n) = 2(2F(n + 1) - 1) = 4F(n + 1) - 2 = \Theta(\varphi^n).$$

3. For $n = 50$:

$$S_{it}(50) = 5 \cdot 50 + 1 = 251 \text{ steps} \implies \frac{251}{10^9} \text{ s} = 251 \text{ ns},$$

$$S_{rec}(50) = 4 \cdot 20\,365\,011\,074 - 2 = 81\,460\,044\,294 \text{ steps} \implies \approx 81.5 \text{ s}.$$

The recursive algorithm is about $3 \cdot 10^8$ times slower.

4. For $n = 60$:

$$S_{rec}(60) = 4 \cdot 2\,504\,730\,781\,961 - 2 = 10\,018\,923\,127\,842 \text{ steps} \\ \implies \approx 10\,019 \text{ s} \approx 2.8 \text{ hours}.$$

For $n = 100$:

$$S_{rec}(100) \approx 4 \cdot 5.73 \cdot 10^{20} \approx 2.29 \cdot 10^{21} \text{ steps} \implies \approx 2.29 \cdot 10^{12} \text{ s}.$$

Since one year has about $3.156 \cdot 10^7$ seconds, this is about $7.3 \cdot 10^4$ years. Meanwhile, **FibIt**(100) executes 501 steps, i.e., about half a microsecond.

5. One hour corresponds to $3600 \cdot 10^9 = 3.6 \cdot 10^{12}$ steps. We need

$$4F(n + 1) - 2 \leq 3.6 \cdot 10^{12} \iff F(n + 1) \leq 9 \cdot 10^{11} + 0.5.$$

Since $F(58) = 591\,286\,729\,879 \leq 9 \cdot 10^{11} < F(59) = 956\,722\,026\,041$, the condition holds for $n + 1 \leq 58$. The largest n is $n = 57$, with

$$S_{rec}(57) = 4 \cdot 591\,286\,729\,879 - 2 = 2\,365\,146\,919\,514 \text{ steps} \approx 39 \text{ minutes},$$

while $n = 58$ already needs $4F(59) - 2 \approx 3.83 \cdot 10^{12}$ steps, i.e., about 64 minutes. Every time n increases by 1, the time of **FibRec** is multiplied by about $\varphi \approx 1.618$: a computer 10 times faster would only allow to increase n by $\log_{\varphi} 10 \approx 4.8$.

With an exponential algorithm, faster computers do not help: the size of the problems that can be solved grows only by an additive constant. The iterative algorithm, with $5n + 1$ steps, computes $F(57)$ in less than 0.3 microseconds.