# Practice 1

Exercise levels:
*(L1) Reproduce:* Reproduce basic facts or check basic understanding.
*(L2) Apply:* Follow step-by-step instructions.
*(L3) Reason:* Show insight using a combination of different concepts.
*(L4) Create:* Prove a non-trivial statement or create an algorithm or data structure of which the objective is formally stated.

## ▶ Lecture 1 Introduction

**Exercise 1** *(L1)* Prof. C. Monster has an array $A$ which he needs to sort. He knows that $A$ contains elements with the keys $\{1, 1, 1, 2, 2, 4\}$, but he does not know in which order. Prof. C. Monster reckons that, because he knows exactly which values are stored in the array, he can simply overwrite them with the values in the right order. Explain why satellite data are sometimes important, and why his approach would lead to problems.

**Solution:**
The keys may have associated satellite data. For example, array $A$ may store information about the grades for the first assignment in the Data Structures course, with the student IDs being the keys of the elements, and the grades being the corresponding satellite data. If the values of the keys are simply overwritten, the association between the keys and the satellite data will be lost. Students will be given incorrect grades. To avoid such problems, the complete elements (key and link to satellite data) should be copied and moved instead.

**Exercise 2** *(L2)* (This is Exercise 2.1-4 from the book.) Consider the searching problem:

**Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$ stored in array $A[1 : n]$ and a value $x$.
**Output:** An index $i$ such that $x$ equals $A[i]$ or the special value NIL if $x$ does not appear in $A$.

Write pseudocode for linear search, which scans through the array from the beginning to the end, looking for $x$. Using a loop invariant, prove that your algorithm is correct. Make sure that the loop invariant fulfills the three necessary properties.

**Solution:**

```
LINEAR-SEARCH(A, n, x)
1  i = 1
2  while i ≤ n and A[i] ≠ x
3        i = i + 1
4  if i > n
5        return NIL
6  else return i
```

The procedure checks each array element until either $i > n$ or the value $x$ is found.

**Loop invariant:** At the start of each iteration of the **while** loop, the value $x$ does not appear in the subarray $A[1 : i-1]$.
**Initialization:** Upon entering the first iteration, $i = 1$. The subarray $A[1 : i-1]$ is empty, and the loop invariant is trivially true.
**Maintenance:** At the start of an iteration for index $i$, the loop invariant says that $x$ does not appear in the subarray $A[1 : i-1]$. If $i \leq n$, then either $A[i] = x$ or $A[i] \neq x$. In the former case, the test in the **while** loop header comes up FALSE, and there is no iteration for $i+1$. In the latter case, the test comes up TRUE. Since $A[i] \neq x$ and $x$ does not appear in $A[1 : i-1]$, we have that $x$ does not appear in $A[1 : i]$. Incrementing $i$ for the next iteration preserves the loop invariant. If $i > n$, then the test in the **while** loop header comes up FALSE, and there is no iteration for $i > n$.
**Termination:** The **while** loop terminated for one of two reasons. If it terminated because $i > n$, then $i = n+1$ at that time. By the loop invariant, the value $x$ does not appear in the the subarray $A[1 : i-1]$, which is the entire array $A[1 : n]$. The procedure properly returns NIL in this case. If the loop terminated because $i \leq n$ and $A[i] == x$, then the procedure properly returns the index $i$.

**Exercise 3** *(L3)* (This is Exercise 2.3-3 from the book.) State a loop invariant for the while loop of lines 12–18 of the MERGE procedure. Show how to use it, along with the while loops of lines 20–23 and 24–27, to prove that the MERGE procedure is correct.

**Solution:**
**Loop invariant:** At the start of each iteration of the while loop of lines 12–18, the subarray $A[p : k-1]$ contains the $i+j$ smallest elements of $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.
We must show that this loop invariant holds prior to the first iteration of the **while** loop of lines 12–18, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates. In fact, we will consider as well the **while** loops of lines 20–23 and lines 24–27 to show that the MERGE procedure works correctly.
**Initialization:** Prior to the first iteration of the loop, we have $k = p$ so that the subarray $A[p : k-1]$ is empty. Since $i = j = 0$, this empty subarray contains the $i + j = 0$ smallest elements of $L$ and $R$, and both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.
**Maintenance:** To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into $A$. Because $A[p : k-1]$ contains the $i+j$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p : k]$ will contain the $i + j + 1$ smallest elements. Incrementing $i$ in line 15 and $k$ in line 18 reestablishes the loop invariant for the next iteration. If it was instead the case that $L[i] > R[j]$, then lines 16–18 perform the appropriate action to maintain the loop invariant.
**Termination:** Each iteration of the loop increments either $i$ or $j$. Eventually, either $i \geq n_L$, so that all elements in $L$ have been copied back into $A$, or $j \geq n_R$, so that all elements in $R$ have been copied back into $A$. By the loop invariant, when the loop terminates, the subarray $A[p : k-1]$ contains the $i + j$ smallest elements of $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$, in sorted

order. The subarray $A[p : r]$ consists of $r - p + 1$ positions, the last $r - p + 1 - (i + j)$ which have yet to be copied back into.
Suppose that the loop terminated because $i \geq n_L$. Then the **while** loop of lines 20–23 iterates 0 times, and the **while** loop of lines 24–27 copies the remaining $n_R - j$ elements of $R$ into the rightmost $n_r - j$ positions of $A[p : r]$. These elements of $R$ must be the $n_R - j$ greatest values in $L$ and $R$. Thus, we just need to show that the correct number of elements in $R$ are copied back into $A[p : r]$, that is, $r - p + 1 - (i + j) = n_R - j$. We use two facts to do so. First, because the number of positions in $A[p : r]$ equals the combined sizes of the $L$ and $R$ arrays, we have $r - p + 1 = n_L + n_R$, or $n_L = r - p + 1 - n_R$. Second, because $i \geq n_L$ and the **while** loop of lines 12–18 increases $i$ by at most 1 in each iteration, we must have that $i = n_L$ when this loop terminated. Thus, we have

$$\begin{aligned} r - p + 1 - (i + j) &= r - p + 1 - n_L - j \\ &= r - p + 1 - (r - p + 1 - n_R) - j \\ &= n_R - j. \end{aligned}$$

If instead the loop terminated because $j \geq n_R$, then you can show that the remaining $n_L - i$ elements of $L$ are the greatest values in $L$ and $R$, and that the **while** loop of lines 20–23 copies them into the last $r - p + 1 - (i + j)$ positions of $A[p : r]$. In either case, we have shown that the MERGE procedure merges the two sorted subarrays $A[p : q]$ and $A[q + 1 : r]$ correctly.

**Exercise 4** *(L3)* (This is part of Problem 2-2 from the book.) Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. The procedure BUBBLESORT sorts array $A[1 : n]$.

```
BUBBLESORT(A, n)
1  for i = 1 to n − 1
2      for j = n downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

(a) Let $A'$ denote the array $A$ after BUBBLESORT$(A, n)$ is executed. To prove that BUBBLESORT is correct, you need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \ldots \leq A'[n].$$

In order to show that BUBBLESORT actually sorts, what else do you need to prove?

**Solution:**
You need to show that the elements of $A'$ form a permutation of the elements of $A$.

(b) State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof in the lecture.

**Solution:**
**Loop invariant:** At the start of each iteration of the **for** loop of lines 2–4, $A[j]$ is the smallest value in the subarray $A[j : n]$, and $A[j : n]$ is a permutation of the values that were in $A[j : n]$ at the time that the loop started.
**Initialization:** Initially, $j = n$, and the subarray $A[j : n]$ consists of the single element $A[n]$. The loop invariant trivially holds.
**Maintenance:** Consider an iteration for a given value of $j$. By the loop invariant, $A[j]$ is the smallest value in $A[j : n]$. Lines 3–4 exchange $A[j]$ and $A[j - 1]$ if $A[j]$ is less than $A[j - 1]$, and so $A[j - 1]$ will be the smallest value in $A[j - 1 : n]$ afterward. Since the only change to the subarray $A[j - 1 : n]$ is this possible exchange, and the $A[j : n]$ is a permutation of the values that were in $A[j : n]$ at the time that the loop started, we see that $A[j - 1 : n]$ is a permutation of the values that were in $A[j - 1 : n]$ at the time that the loop started. Decrementing $j$ for the next iteration maintains the invariant.
**Termination:** The loop terminates when $j$ reaches $i$. By the statement of the loop invariant, $A[i]$ is the smallest value in the subarray $A[i : n]$, and $A[i : n]$ is a permutation of the values that were in $A[i : n]$ at the time that the loop started.

(c) Using the termination condition of the loop invariant proved in (b), state a loop invariant for the **for** loop in lines 1–4 that allows you to prove the inequality above. Your proof should use the structure of the loop invariant proof in the lecture.

**Solution:**
**Loop invariant:** At the start of each iteration of the **for** loop of lines 1–4, the subarray $A[1 : i - 1]$ consists of the $i - 1$ smallest values originally in $A[1 : n]$, in sorted order, and $A[i : n]$ consists of the $n - i + 1$ remaining values originally in $A[1 : n]$.
**Initialization:** Before the first iteration of the loop, $i == 1$. The subarray $A[1 : i - 1]$ is empty, and so the loop invariant vacuously holds.
**Maintenance:** Consider an iteration for a given value of $i$. By the loop invariant, $A[1 : i - 1]$ consists of the $i - 1$ smallest values in $A[1 : n]$, in sorted order. Therefore, $A[i - 1] \leq A[i]$. Part (b) showed that after executing the **for** loop of lines 2–4, $A[i]$ is the smallest value in $A[i : n]$, and so $A[1 : i]$ now consists of the $i$ smallest values originally in $A[1 : n]$, in sorted order. Moreover, since the **for** loop of lines 2–4 permutes $A[i : n]$, the subarray $A[i + 1 : n]$ consists of the $n - i$ remaining values originally in $A[1 : n]$.
**Termination:** The **for** loop of lines 1–4 terminates when $i = n$, so that $i - 1 = n - 1$. By the statement of the loop invariant, $A[1 : i - 1]$ is the subarray $A[1 : n - 1]$, and it consists of the $n - 1$ smallest values originally in $A[1 : n]$, in sorted order. The remaining element must be the largest value in $A[1 : n]$, and it is in $A[n]$. Therefore, the entire array $A[1 : n]$ is sorted.

# ▶ Lecture 2 Analysis of algorithms

## Asymptotic Notation

### Exercise 5

(a) *(L1)* Give definitions of the big-$O$, big-$\Omega$, and big-$\Theta$ notations.

**Solution:**
For a given function $g(n) : \mathbb{N} \to \mathbb{N}$,

- $O(g(n)) = \{f(n) :$ there exist positive constants $c$ and $n_0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.
- $\Omega(g(n)) = \{f(n) :$ there exist positive constants $c$ and $n_0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$.
- $\Theta(g(n)) = \{f(n) :$ there exist positive constants $c_1$, $c_2$, and $n_0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$.

(b) *(L3)* Explain using the definitions why $f(n) = \Theta(n)$ when $f(n) = \Omega(n)$ and $f(n) = O(n)$.

**Solution:**
Let $f(n) = O(n)$ and $f(n) = \Omega(n)$. Then, by definition of the big-$O$ notation, there exist positive constants $c_1$ and $n_1$ such that $0 \leq f(n) \leq c_1 n$ for all $n \geq n_1$. On the other hand, by definition of the big-$\Omega$ notation, there exist positive constants $c_2$ and $n_2$ such that $0 \leq c_2 n \leq f(n)$ for all $n \geq n_2$.
Thus, there exist positive constants $c_1$, $c_2$, and $n_0$ such that $0 \leq c_2 n \leq f(n) \leq c_1 n$ for all $n \geq n_0$, where we choose $n_0 = \max\{n_1, n_2\}$. Then, by definition of the big-$\Theta$ notation, $f(n) = \Theta(n)$.

**Exercise 6** *(L2)* For each function on the left, $p(n)$, write the letter of a function on the right, $q(n)$, such that $p(n) \in \Theta(q(n))$. If no such function $q(n)$ is listed, then choose (l).

$f(n) = \sum_{i=1}^{n}(4i-4)$ _____

$g(n) = \sum_{i=1}^{n}\sum_{j=1}^{i} i$ _____

$h(n) = \sum_{i=1}^{\log n} n$ _____

$k(n) = \sum_{i=0}^{n} \frac{4}{2^i}$ _____

(a) 1 (g) $\log n$
(b) $n$ (h) $n \log n$
(c) $n(\log n)^2$ (i) $n^2$
(d) $n^2 \log n$ (j) $n^3$
(e) $2^n$ (k) $2^{2n}$
(f) $n^n$ (l) no match

**Solution:**

(a) $f(n) = \sum_{i=1}^{n}(4i-4) = 4\sum_{i=1}^{n} i - 4\sum_{i=1}^{n} 1 = 4\frac{n(n+1)}{2} - 4n = 2n^2 - 2n = \Theta(n^2)$.

(b) $g(n) = \sum_{i=1}^{n}\sum_{j=1}^{i} i = \sum_{i=1}^{n}\left(i \cdot \sum_{j=1}^{i} 1\right) = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$.

(c) $h(n) = \sum_{i=1}^{\log n} n = n \cdot \sum_{i=1}^{\log n} 1 = n \log n = \Theta(n \log n)$.

(d) $k(n) = \sum_{i=0}^{n} \frac{4}{2^i} = 4\sum_{i=0}^{n} \left(\frac{1}{2}\right)^i$. Using the formula $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$, we get that $k(n) = 4\frac{\left(\frac{1}{2}\right)^{n+1}-1}{\frac{1}{2}-1} = 4(2 - \frac{1}{2^n}) = \Theta(1)$.

**Exercise 7** *(L2)* Rank the following functions of $n$ by order of growth (starting with the slowest growing). Functions with the same order of growth should be ranked equal.

| | | | | | |
|---|---|---|---|---|---|
| $n\sqrt[4]{n}$ | $n^{-\log 4}$ | $n^{0.00001}$ | $\sum_{i=0}^{n} 2^i$ | $4^{\log 8}$ | $n!$ |
| 1 | $\frac{\log(2n)}{2}$ | $n \log n$ | $\log \sqrt{n}$ | $\sum_{i=1}^{\log n} i$ | $n^n$ |

**Solution:**

$n^{-\log 4} = n^{-2} = \Theta(n^{-2})$

$4^{\log 8} = 4^3 = 64 = \Theta(1)$, $\quad 1 = \Theta(1)$

$\frac{\log(2n)}{2} = \frac{\log 2 + \log n}{2} = \Theta(\log n)$, $\quad \log \sqrt{n} = \log n^{0.5} = \frac{1}{2}\log n = \Theta(\log n)$

$\sum_{i=1}^{\log n} i = \frac{\log n(\log n+1)}{2} = \Theta(\log^2 n)$

$n^{0.00001} = \Theta(n^{0.00001})$

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

$n\sqrt[4]{n} = n^{\frac{5}{4}} = \Theta(n^{\frac{5}{4}})$

$\sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} = \Theta(2^n)$

$n! = \Theta(n!)$

$n^n = \Theta(n^n)$

**Exercise 8** *(L3)* Formally prove or disprove the following statements using the definitions of the respective asymptotic notations:

(a) $n^2 - 3n + 4 = \Theta(n^2)$

**Solution:**
We will prove that this statement is true by definition of the big-$\Theta$ notation. By definition, $n^2 - 3n + 4 = \Theta(n^2)$, if there exist positive constants $c_1$, $c_2$, and $n_0$ such that

$$0 \leq c_1 n^2 \leq n^2 - 3n + 4 \leq c_2 n^2 \qquad \text{for all } n \geq n_0.$$

To prove that the statement holds we need to find specific values for $c_1$, $c_2$, and $n_0$ that always satisfy the inequalities for all $n \geq n_0$.
So, we need $0 \leq c_1 n^2 \leq n^2 - 3n + 4$ and $0 \leq n^2 - 3n + 4 \leq c_2 n^2$ for all $n \geq n_0$.
**First inequality**. Choose $c_1 = 0.5$, then $0 \leq 0.5n^2$ for $n \geq 0$. Furthermore, $0.5n^2 < 0.5n^2 + 4 \leq 0.5n^2 + 4 + (0.5n^2 - 3n) = n^2 - 3n + 4$ when $0.5n^2 - 3n \geq 0$, so when $n \geq 6$.
**Second inequality**. For the second inequality, choose $c_2 = 2$, then $0 \leq n^2 - 3n + 4 \leq n^2 + 4$ for all $n \geq 0$. Furthermore $n^2 + 4 = 2n^2 + (4 - n^2) \leq 2n^2$ when $4 - n^2 \leq 0$, so when $n \geq 2$.
If we choose $n_0 = 6$ both inequalities, $0 \leq c_1 n^2 = 0.5n^2 \leq n^2 - 3n + 4$ and $0 \leq n^2 - 3n + 4 \leq 2n^2 = c_2 n^2$, will hold for all $n \geq n_0$.
Thus there exists positive constants, namely $c_1 = 0.5$, $c_2 = 2$ and $n_0 = 6$ that make the proposition true. Hence, the statement is true by definition.

(b) $n^2 \log n = \Omega(n^3)$

**Solution:**
We will prove that this statement is false by contradiction. Assume that it is true, then by the definition of big-$\Omega$ notation, there exist positive constants $c$ and $n_0$ such that

$$0 \leq cn^3 \leq n^2 \log n \qquad \text{for all } n \geq n_0.$$

Then $0 \leq cn \leq \log n$ for all $n \geq n_0$. But we know that any polynomial function grows faster than any logarithmic function. Thus, for any positive values of $c$ and $n_0$, there will exist such $n \geq n_0$ that $c > \frac{\log n}{n}$. We have reached a contradiction and the only conclusion can be that our assumption must be false. Hence, the original claim is false.

(c) $\sqrt{n} = O(\log n)$

**Solution:**
We will prove that this statement is false by contradiction. Assume that it is true, then by the definition of big-$O$ notation, there exist positive constants $c$ and $n_0$ such that

$$0 \leq \sqrt{n} \leq c \log n \qquad \text{for all } n \geq n_0.$$

But we know that any polynomial function grows faster than any logarithmic function. Thus, for any positive values of $c$ and $n_0$, there will exist such $n \geq n_0$ that $\frac{\sqrt{n}}{\log n} > c$. We have reached a contradiction and the only conclusion can be that our assumption must be false. Hence, the original claim is false.

(d) $\Omega(n \log n) \cap O(n^2) \neq \emptyset$

**Solution:**
We will prove that this statement is true by showing that there exists such function $f(n)$ that $f(n) \in \Omega(n \log n)$ and $f(n) \in O(n^2)$, and therefore, the intersection of the two sets is non-empty.
We choose $f(n) = n^2$.
First, trivially, $n^2 = O(n^2)$. Indeed, there exist positive constants $c = 1$ and $n_0 = 1$ such that $0 \leq n^2 \leq n^2$ for all $n \geq 1$.
We are left to show that $n^2 = \Omega(n \log n)$. By definition, $n^2 = \Omega(n \log n)$, if there exist positive constants $c$ and $n_0$ such that $0 \leq cn \log n \leq n^2$ for all $n \geq n_0$.
To prove that we will find values for $c$ and $n_0$ that always satisfy the inequality for all $n \geq n_0$.
After dividing the inequality by $n$ we get $0 \leq c \log n \leq n$.
Choose $c = 1$, then $0 \leq \log n \leq n = c \cdot n$ holds for all $n > 0$.
We know that $n > \log n$ for all positive $n$. So we can simply choose $n_0 = 1$.
Thus there exists positive constants, namely $c = 1$ and $n_0 = 1$ that make the inequality hold. So, we have shown that $n^2 = \Omega(n \log n)$.
We have found such $f(n)$ that $f(n) \in \Omega(n \log n)$ and $f(n) \in O(n^2)$, which implies that $\Omega(n \log n) \cap O(n^2) \neq \emptyset$.

(e) If $f(n) = \Omega(h(n))$ and $g(n) = \Omega(h(n))$, then $f(n)g(n) = \Omega(h^2(n))$.

**Solution:**
We will prove that this statement is true by definition of the big-$\Omega$ notation.
By definition, $f(n) = \Omega(h(n))$, if there exist positive constants $c_1$ and $n_1$ such that

$$0 \leq c_1 h(n) \leq f(n) \quad \text{for all } n \geq n_1.$$

Similarly, by definition, $g(n) = \Omega(h(n))$, if there exist positive constants $c_2$ and $n_2$ such that

$$0 \leq c_2 h(n) \leq g(n) \quad \text{for all } n \geq n_2.$$

After multiplying these inequalities, we get that the following inequalities hold:

$$0 \leq c_1 c_2 h^2(n) \leq f(n)g(n) \quad \text{for all } n \text{ such that } n \geq n_1 \text{ and } n \geq n_2.$$

To show that $f(n)g(n) = \Omega(h^2(n))$, we need to find positive constants $c$ and $n_0$ such that

$$0 \leq ch^2(n) \leq f(n)g(n) \quad \text{for all } n \geq n_0.$$

Then, for $c = c_1c_2$ and $n_0 = \max\{n_1, n_2\}$ the proposition is true. Hence, the statement is true by definition.

## Recurrences

**Exercise 9** *(L2)* Using the Master theorem, prove asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$.

(a) $T(n) = 16\,T(n/4) + n^2$

**Solution:**
We use the Master theorem to solve the recurrence:
$T(n) = \underset{\downarrow a}{16}T(n/\underset{\downarrow b}{4}) + \underset{\downarrow f(n)}{n^2}$.
We have $a = 16 \geq 1$, $b = 4 > 1$, and driving function $f(n) = n^2$.
The watershed function is $n^{\log_b a} = n^{\log_4 16} = n^2$. Then $f(n) = n^2 = \Theta(n^2 \log^0 n)$ so case 2 of the master theorem applies with $k = 0$. The solution to the recurrence is $T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^{\log_4 16} \log n) = \Theta(n^2 \log n)$.

(b) $T(n) = 7\,T(n/2) + n^2$

**Solution:**
We use the Master theorem to solve the recurrence:
$T(n) = \underset{\downarrow a}{7}T(n/\underset{\downarrow b}{2}) + \underset{\downarrow f(n)}{n^2}$.
We have $a = 7 \geq 1$, $b = 2 > 1$, and driving function $f(n) = n^2$.
The watershed function is $n^{\log_b a} = n^{\log 7} \approx n^{2.8}$, and $f(n) = n^2 = O(n^{\log 7 - \varepsilon})$ for any $0 < \varepsilon \leq \log 7 - 2$. We can choose, for example, $\varepsilon = 1/10$. So case 1 of the Master theorem applies, and $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log 7})$.

(c) $T(n) = 4\,T(n/2) + n^2\sqrt{n}$

**Solution:**
We use the Master theorem to solve the recurrence:
$T(n) = \underset{\downarrow a}{4}T(n/\underset{\downarrow b}{2}) + \underset{\downarrow f(n)}{n^2\sqrt{n}}$.
We have $a = 4 \geq 1$, $b = 2 > 1$, and driving function $f(n) = n^2\sqrt{n}$.
The watershed function is $n^{\log_b a} = n^{\log_2 4} = n^2$, and $f(n) = n^2\sqrt{n} = \Omega(n^{2+\varepsilon})$ for any $0 < \varepsilon \leq 1/2$. We can choose, for example, $\varepsilon = 1/2$. So, if the regularity condition holds, then case 3 of the Master theorem will apply.
Regularity condition: $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$.

$$\begin{aligned} af(n/b) &= 4f(n/2) \\ &= 4(n/2)^2\sqrt{n/2} \\ &= \tfrac{1}{\sqrt{2}}n^2\sqrt{n} \\ &\leq cf(n) \qquad (\text{for } \tfrac{1}{\sqrt{2}} \leq c < 1. \text{ We can choose, for example, } c = 0.9). \end{aligned}$$

So, indeed, case 3 of the Master theorem applies, and hence $T(n) = \Theta(f(n)) = \Theta(n^2\sqrt{n})$.

(d) *(L4)* For the solutions of the recurrences (a)–(c), give another recurrence that solves to an exactly the same asymptotic bound, but using a different case of the Master theorem.

**Solution:**

(a) We want to find a recurrence that solves to $T(n) = \Theta(n^2 \log n)$ by case 1 or 3 of the Master theorem. Case 1 gives a solution of the form $\Theta(n^{\log_b a})$, thus, it cannot give the answer that we need. We will try case 3 of the Master theorem, which gives as a solution $\Theta(f(n))$:
$T(n) = aT(n/b) + f(n)$ with $f(n) = n^2 \log n$ and

- $a \geq 1, b > 1$,
- $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some $\varepsilon > 0$,
- and the regularity condition holds, i.e., $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$.

First, we need $n^2 \log n = \Omega(n^{\log_b a+\varepsilon})$ for some positive $\varepsilon$. This will be true when $\log_b a < 2$, that is, $a < b^2$.

For the regularity condition to hold we need, for some $c < 1$,
$a(n/b)^2 \log(n/b) \leq cn^2 \log n$.

Choose $a = 2$ and $b = 2$. Then, both conditions will hold.

Specifically, $a < b^2$ holds, and $n^2 \log n = \Omega(n^{\log 2+\varepsilon})$ for some positive $\varepsilon$ (for example, $\varepsilon = 1$).

Also, $a(n/b)^2 \log(n/b) = \frac{1}{2}n^2 \log(n/2) \leq cn^2 \log n$ holds for some $c < 1$ (for example, $c = 1/2$), and all sufficiently large $n$.

Thus, we get that $T(n) = 2T(n/2) + n^2 \log n$ solves to $T(n) = \Theta(n^2 \log n)$ by case 3 of the Master theorem.

(b) We want to find a recurrence that solves to $T(n) = \Theta(n^{\log 7})$ by case 2 or 3 of the Master theorem. Case 2 gives a solution of the form $\Theta(n^{\log_b a} \log^k n)$ for some $k \geq 1$, thus, it cannot give the answer that we need. We will try again case 3 of the Master theorem, which gives as a solution $\Theta(f(n))$:
$T(n) = aT(n/b) + f(n)$ with $f(n) = n^{\log 7}$ and

- $a \geq 1, b > 1$,
- $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some $\varepsilon > 0$,
- and the regularity condition holds, i.e., $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large $n$.

First, we need $n^{\log 7} = \Omega(n^{\log_b a+\varepsilon})$ for some positive $\varepsilon$. This will be true when $\log_b a < \log 7$, that is, $a < b^{\log 7}$.

For the regularity condition to hold we need, for some $c < 1$,
$a(n/b)^{\log 7} \leq cn^{\log 7}$.

Choose $a = 4$ and $b = 2$. Then both conditions will hold.

Specifically, $a < b^{\log 7}$ holds, and $n^{\log 7} = \Omega(n^{\log 4+\varepsilon})$ for some positive $\varepsilon$ (for example, $\varepsilon = 0.1$).

Also, $a(n/b)^{\log 7} = \frac{4}{7}n^{\log 7} \leq cn^{\log 7} \log n$ holds for some $c < 1$ (for example, $c = 5/7$), and all sufficiently large $n$.

Thus, we get that $T(n) = 4T(n/2) + n^{\log 7}$ solves to $T(n) = \Theta(n^{\log 7})$ by case 3 of the Master theorem.

(c) We want to find a recurrence that solves to $T(n) = \Theta(n^2\sqrt{n})$ by case 1 or 2 of the Master theorem. Case 2 gives a solution of the form $\Theta(n^{\log_b a} \log^k n)$ for some $k \geq 1$, thus, it cannot give the answer that we need. We will try case 1 of the Master theorem, which gives as a solution $\Theta(n^{\log_b a})$:
$T(n) = aT(n/b) + f(n)$ with $n^2\sqrt{n} = n^{\log_b a}$ and

- $a \geq 1, b > 1$,
- $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$.

First, we need $n^2\sqrt{n} = n^{\log_b a}$, thus, $a = b^{5/2}$. Choose, for example, $a = 4\sqrt{2}$ and $b = 2$.

Next, we need $f(n) = O(n^{\log_b a - \varepsilon}) = O(n^{5/2-\varepsilon})$ for some positive $\varepsilon$.

Choose $f(n) = n^2$. Then, $n^2 = O(n^{5/2-\varepsilon})$ holds for some positive $\varepsilon$ (for example, for $\varepsilon = 1/4$).

Thus, we get that $T(n) = 4\sqrt{2}T(n/2) + n^2$ solves to $T(n) = \Theta(n^2\sqrt{n})$ by case 1 of the Master theorem.

**Exercise 10**

(a) *(L3)* Prove using the substitution method that the solution of $T(n) = T(\lceil n/3 \rceil) + T(\lfloor 2n/3 \rfloor) + n$ with $T(1) = 1$ is $T(n) = O(n \log n)$.

**Solution:**
We show $T(n) = O(n \log n)$ using the substitution method. That is, we will find a specific value of a positive constant $c$, for which we can prove that $T(n) \leq c \cdot n \log n$ by induction over $n$, starting at some $n_0$.
First, consider $n = 1$:
$T(1) = 1 \leq c \cdot 1 \log 1 = 0$ does not hold for any positive $c$. Thus, $n = 1$ is not a base case, and $n_0 > 1$.
Next, consider $n = 2$:
$T(2) = T(1) + T(1) + 2 = 4 \leq c \cdot 2 \log 2 = 2c$. This inequality holds for $c \geq 2$. So, we chose $n_0 = 2$, and will try to find such $c \geq 2$ for which we can prove by induction that $T(n) \leq c \cdot n \log n$. Then, $n = 2$ will be a base case.
There may be more base cases. Specifically, if there are such values of $n$ for which the recursive definition depends on $T(1)$ (for which the inequality does not hold!), these values of $n$ must be base cases themselves.
Consider $n = 3$:
$T(3) = T(1) + T(2) + 3$ depends on $T(1)$, thus $n = 3$ is a base case.
Consider $n = 4$:
$T(4) = T(2) + T(2) + 4$ does not depend on $T(1)$, so $n = 4$ does not have to be a base case.
**Base cases:**
$\boldsymbol{n = 2}$**:** $T(2) = T(1) + T(1) + 2 = 4 \leq c \cdot n \log n = c \cdot 2 \log 2 = 2c$, it holds for $c \geq 2$
$\boldsymbol{n = 3}$**:** $T(3) = T(1) + T(2) + 3 = 8 \leq c \cdot n \log n = c \cdot 3 \log 3 \approx 4.75c$, it holds for $c \geq \frac{8}{3 \log 3}$
**Step:** We will show that the inequality holds for some $n \geq n_0$.
**IH:** Assume that $T(k) \leq c \cdot k \log k$ all $n_0 \leq k < n$.

$$
\begin{aligned}
T(n) &= T(\lceil n/3 \rceil) + T(\lfloor 2n/3 \rfloor) + n && \{\text{IH on } \lceil n/3 \rceil \text{ and } \lfloor 2n/3 \rfloor\} \\
&\leq c \lceil n/3 \rceil \log \lceil n/3 \rceil + c \lfloor 2n/3 \rfloor \log \lfloor 2n/3 \rfloor + n && \left\{ \begin{array}{l} \log \lceil n/3 \rceil \leq \log (2n/3) \text{ and} \\ \log \lfloor 2n/3 \rfloor \leq \log (2n/3) \end{array} \right\} \\
&\leq c \lceil n/3 \rceil \log(2n/3) + c \lfloor 2n/3 \rfloor \log(2n/3) + n \\
&= c (\lceil n/3 \rceil + \lfloor 2n/3 \rfloor) \log(2n/3) + n && \{\text{case distinction on } n \bmod 3\} \\
&= cn \log(2n/3) + n \\
&= cn(\log n + \log 2 - \log 3) + n \\
&= cn \log n + cn(\log 2 - \log 3) + n \\
&= cn \log n + n(1 - c(\log 3 - 1)) && \{\text{if } c \geq 1/(\log 3 - 1) \approx 1.71\} \\
&\leq cn \log n
\end{aligned}
$$

Both the base cases and the induction hold when $c \geq \max\{2, \frac{8}{3 \log 3}, \frac{1}{\log 3 - 1}\}$. For example $c = 2$. As the inequality holds for $n = 2$ and $n = 3$, and for any $n > 3$ given that it holds for all $k$ such that $2 \leq k < n$, we conclude that by induction the inequality must hold for all $n \geq 2$. Thus, for $c = 2$ and $n_0 = 2$, $T(n) \leq cn \log n$ for all $n \geq n_0$, and then by definition, $T(n) = O(n \log n)$.

(b) *(L2)* Can the master method be applied to $T(n) = 4\,T(n/2) + n^2 \log\log n$? Why or why not?

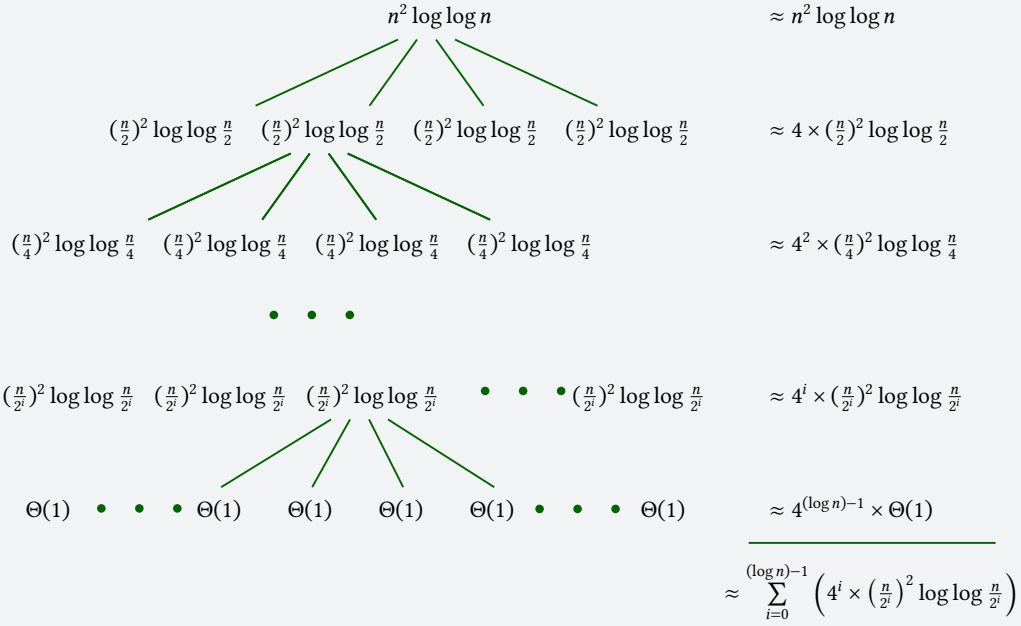
**Solution:**
For the master method to be applicable to

$$T(n) = \underset{\downarrow a}{4}T(n/\underset{\downarrow b}{2}) + n^2 \log\log \underset{\downarrow f(n)}{n}$$

(with $a = 4$, $b = 2$, $f(n) = n^2 \log\log n$), we need that $f(n) \in O(n^{\log_b a-\varepsilon}) = O(n^{2-\varepsilon})$, or $f(n) \in \Omega(n^{\log_b a+\varepsilon}) = \Omega(n^{2+\varepsilon})$ for some $\varepsilon > 0$, or that $f(n) \in \Theta(n^{\log_b a} \log^k n) = \Theta(n^2 \log^k n)$ for some $k \geq 1$. Clearly, $f(n) \notin O(n^{2-\varepsilon})$. We also do not have $f(n) \in \Theta(n^2 \log^k n)$ as $f(n) \in \omega(n^2)$ and $f(n) \in o(n^2 \log n)$ – in other words, $f(n)$ falls "between" $k = 0$ and $k = 1$. Furthermore, $f(n) \neq \Omega(n^{2+\varepsilon})$ because any polynomial function grows faster than any logarithmic function. Thus, for any positive $\varepsilon$ and any $n_0$, there exist such $n \geq n_0$ that $n^\varepsilon > \log n$. So the master theorem cannot be applied.

(c) *(L3)* Use a recursion tree to get an intuition of what the right upper asymptotic bound for $T(n) = 4\,T(n/2) + n^2 \log\log n$ might be.

**Solution:**
To make a guess of the right asymptotic bound, we build the following recursion tree:




Note that since $\log\log n$ is not defined for $n = 1$, we take as leaves of the tree when $n = 2$. As a result the height of the tree is $\log_b(n) - 1 = \log(n) - 1$ (we have $b = 2$), and each level $i$ of the tree has $a^i = 4^i$ nodes (we have $a = 4$). The work performed at each node of level $i$ is $\approx f(n/b^i) = (\frac{n}{2^i})^2 \log\log \frac{n}{2^i}$.
Thus, the total work will be

$$\approx \sum_{i=0}^{\log(n)-1} 4^i \cdot \left(\frac{n}{2^i}\right)^2 \log\log \frac{n}{2^i} = n^2 \sum_{i=0}^{\log(n)-1} \log(\log(n) - i) = n^2 \sum_{i=1}^{\log n} \log i.$$

Now recall that $\log a + \log b = \log ab$, so $\sum_{i=1}^{\log n} \log i = \log \prod_{i=1}^{\log n} i = \log((\log n)!)$. We observe that $\log(k!) = \Theta(k \log k)$ and substitute $k = \log n$ to get:

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

(d) *(L3)* Prove the upper asymptotic bound from (c) using the substitution method.

**Solution:**
The recursion implies the use of a floor or ceiling function. The asymptotic bound is not affected by this choice. Assume that $T(n) = 4T(\lfloor n/2 \rfloor) + n^2 \log\log n$. We will easily be able to adapt our solution to the case of the ceiling function.
We show $T(n) = O(n^2 \log n \log\log n)$ using the substitution method. That is, we will find a specific value of a positive constant $c$, for which we can prove that $T(n) \leq c \cdot n^2 \log n \log\log n$ by induction over $n$, starting at some $n_0$.
First, consider $n = 1$:
Let $T(1) = d$ for some positive constant $d$. $T(1) = d \leq c \cdot 1^2 \log 1 \log\log 1 = \ldots$. For $n = 1$ we cannot prove this bound as the right side is not defined.
Next, consider $n = 2$:
$T(2) = 4T(1) + 2^2 \log\log 2 = 4d + 0 \leq c \cdot 2^2 \log 2 \log\log 2 = 0$. We cannot make this equation hold for any $c$ so we will need a larger $n_0$.
Next, consider $n = 3$:
$T(3) = 4T(1) + 3^2 \log\log 3 = 4d + 9 \log\log 3 \leq c \cdot 3^2 \log 3 \log\log 3$. This inequality holds for $c \geq 1/\log 3 + 4d/(9 \log 3 \log\log 3) \approx 3.799d + 0.631$. So, we chose $n_0 = 3$, and will try to find some $c > 3.799d + 0.631$ for which we can prove by induction that $T(n) \leq c \cdot n^2 \log n \log\log n$. Then, $n = 3$ will be a base case.
There may be more base cases. Specifically, if there are such values of $n$ for which the recursive definition depends on $T(1)$ or $T(2)$ (for which the inequality does not hold!), these values of $n$ must be base cases themselves.
Consider $n = 4$:
$T(4) = 4T(2) + 4^2 \log\log 4$ depends on $T(2)$ which is not a base case, thus $n = 4$ has to be a base case itself.
Consider $n = 5$:
$T(5) = 4T(2) + 5^2 \log\log 5$ depends on $T(2)$ which is not a base case, thus $n = 5$ has to be a base case itself.
Consider $n = 6$:
$T(6) = 4T(3) + 6^2 \log\log 6$ depends on $T(3)$ which is a base case, thus $n = 6$ does not have to be a base case.
We will prove by induction that $T(n) \leq cn^2 \log n \log\log n$ for some positive $c$ for all $n \geq n_0 = 2$.
**Base cases:**
$\boldsymbol{n = 3}$**:** $T(3) = 4T(1) + 3^2 \log\log 3 = 4d + 9 \log\log 3 \leq c \cdot 3^2 \log 3 \log\log 3$ for $c \geq 1/\log 3 + 4d/(9 \log 3 \log\log 3) \approx 3.799d + 0.631$. In particular $c \geq 4d + 1$ is sufficient.
$\boldsymbol{n = 4}$**:** $T(4) = 4T(2) + 4^2 \log\log 4 = 16d + 16 \leq c \cdot 4^2 \log 4 \log\log 4 = 16c$ for $c \geq 1/2 + d/2$. In particular $c \geq 4d + 1$ is sufficient.
$\boldsymbol{n = 5}$**:** $T(5) = 4T(2) + 5^2 \log\log 5 = 16d + 25 \log\log 5 \leq c \cdot 5^2 \log 5 \log\log 5 = c \cdot$

$25 \log 5 \log\log 5$ for $c \geq 1/\log 5 + 16d/(25 \log 5 \log\log 5) \approx 0.227d + 0.431$. In particular $c \geq 4d + 1$ is sufficient.
**Step:** Let $n \geq 6$.
**IH:** Assume that, for all $n_0 \leq k < n$, the inequality $T(k) \leq c \cdot k^2 \log k \log\log k$ holds.
We will show that the inequality also holds for $n$:

$$\begin{aligned}
T(n) &= 4T(\lfloor n/2 \rfloor) + n^2 \log\log n && \{\text{IH on } \lfloor n/2 \rfloor\} \\
&\leq 4c \cdot (\lfloor n/2 \rfloor)^2 \log(\lfloor n/2 \rfloor) \log\log(\lfloor n/2 \rfloor) + n^2 \log\log n && \{\lfloor n/2 \rfloor \leq n/2\} \\
&\leq 4c \cdot (n/2)^2 \log(n/2) \log\log(n/2) + n^2 \log\log n \\
&= c \cdot n^2 \Big( \log(n/2) \log\log(n/2) + \tfrac{1}{c} \log\log n \Big) \\
&= c \cdot n^2 \Big( (\log n - 1) \cdot \log\log(n/2) + \tfrac{1}{c} \log\log n \Big) \\
&= c \cdot n^2 \Big( \log n \log\log(n/2) - \log\log(n/2) + \tfrac{1}{c} \log\log n \Big) \\
&\leq c \cdot n^2 \Big( \log n \log\log n - \log\log(n/2) + \tfrac{1}{c} \log\log n \Big).
\end{aligned}$$

From here we can find the bound we are aiming for if we can show that $-\log\log(n/2) + \frac{1}{c} \log\log n \leq 0$, so that $\frac{1}{c} \log\log n \leq \log\log(n/2)$. Intuitively the $1/c$ outside of the log should have more impact than the $1/2$ inside the log. Let us prove the above claim for $c \geq 2$:

$$\begin{aligned}
\tfrac{1}{c} \log\log n &\leq \tfrac{1}{2} \log\log n \\
&= \log \sqrt{\log n} && \{\log n \geq 2\} \\
&\leq \log \left( \tfrac{1}{2} \log n \right) \\
&= \log\log \sqrt{n} && \{n \geq 2\} \\
&\leq \log\log n/2.
\end{aligned}$$

Note that both conditions here are satisfied in our step case as $n \geq 6$. So when $c \geq 2$ we can indeed conclude that

$$\begin{aligned}
T(n) &\leq c \cdot n^2 \Big( \log n \log\log n - \log\log(n/2) + \tfrac{1}{c} \log\log n \Big) \\
&\leq c \cdot n^2 \log n \log\log n.
\end{aligned}$$

As the inequality holds for $n = 3, n = 4$ and $n = 5$, and for any $n > 5$ given that it holds for all $k$ such that $3 \leq k < n$, we conclude that by induction the inequality must hold for all $n \geq 3$. Thus, for $c = d + 2 \geq \max\{d + 1, 2\}$ and $n_0 = 3$, $T(n) \leq c \cdot n^2 \log n \log\log n$ for all $n \geq n_0$, and by definition, $T(n) = O(n^2 \log n \log\log n)$.

## Analysis of Algorithms

**Exercise 11** For each of the algorithms below answer the following questions.

```
YUKKURI(m, n)
    Input: m, n ∈ ℕ
1   r = 1
2   k = n
3   while k > 0
4       k = k − 1
5       r = r · m
6   return r
```

```
HAYAKU(m, n)
    Input: m, n ∈ ℕ
1   k = n
2   r = 1
3   h = m
4   while k > 0
5       if k even
6           k = k/2
7           h = h · h
8       else k = k − 1
9           r = r · h
10  return r
```

(a) *(L3)* What does the algorithm compute?

**Solution:**
Both algorithms compute $m^n$.

(b) *(L2/3)* Prove your claim using a proof by loop invariant.

**Solution:**

(a) YUKKURI:

**Loop Invariant**: $r = m^{n-k}$.

**Initialization:**
At the start of the first iteration of the while loop we have $k = n$ and $r = 1 = m^{n-n}$, thus the loop invariant holds at the start of the first iteration.

**Maintenance:**
Assume the loop invariant holds at the start of an arbitrary iteration. Denote the value of $k$ at the start of that iteration as $k_i$. Then, our assumption is that $k = k_i$ and $r = m^{n-k_i}$. We need to prove that the loop invariant will also hold at the start of the next iteration.

Within an iteration, $k$ is decreased by 1. The new value of $k$ is $k_i - 1$. Within an iteration, $r$ is multiplied by $m$. Thus, at the start of the next iteration $k = k_i - 1$ and $r = m^{n-k_i}m = m^{n-(k_i-1)}$. We indeed have that the loop invariant holds at the start of the next iteration. Therefore, the loop invariant holds throughout the whole while-loop.

**Termination:**
The loop terminates when $k = 0$. By the loop invariant, at the start of the iteration with $k = 0$, $r$ equals $m^{n-0} = m^n$.

This is exactly what the algorithm returns.

(b) Hayaku:

Consider the binary representation of $n$, and denote the number of digits in it as $d_n$.

**Loop Invariant**: $r = m^n/h^k$.
**Initialization:**
At the start of the first iteration of the while loop we have $k = n$, $h = m$, and $r = 1 = m^n/h^k = 1$, thus the loop invariant holds at the start of the first iteration.

**Maintenance:**
Assume the loop invariant holds at the start of an arbitrary iteration. Denote the value of $k$ at the start of this iteration as $k_i$, and the value of $h$ at the start of this iteration as $h_i$. Then, our assumption is that $r = m^n/{h_i}^{k_i}$. We need to prove that the loop invariant will also hold at the start of the next iteration.

Consider the case when $k_i$ is even. Within the iteration, $k$ is decreased by a factor of 2, $h$ is squared, and $r$ remains unchanged. Then, at the start of the next iteration, $k = k_i/2$, $h = h_i^2$, and $r = m^n/{h_i}^{k_i} = m^n/(h_i^2)^{k_i/2} = m^n/h^k$. Thus, at the start of the next iteration the loop invariant holds.

Consider the case when $k_i$ is odd. Within an iteration, $k$ is decreased by 1, $r$ is multiplied by $h$, and $h$ remains unchanged. Thus, at the start of the next iteration, $k = k_i - 1$, $h = h_i$, $r = h_i \cdot m^n/{h_i}^{k_i} = m^n/{h_i}^{k_i-1} = m^n/h^k$. Thus, at the start of the next iteration the loop invariant holds. Therefore, the loop invariant holds throughout the whole while-loop.

**Termination:**
The loop terminates when $k = 0$. By the loop invariant, at the start of the iteration with $k = 0$, $r = m^n/h^0 = m^n$.

This is exactly what the algorithm returns.

(c) *(L2)* Analyze the asymptotic running time of the algorithm.

**Solution:**

(a) Yukkuri: Lines 1, 2, and 6 take in $O(1)$ time. Each line of the while-loop also takes $O(1)$ time each. The while-loop starts with $k = n$, in every iteration $k$ is decreased by one, and the loop terminates when $k = 0$. Thus the loop is executed $n$ times and takes $O(n)$ time in total. Therefore, the total running time of the algorithm is $O(n)$.

(b) Hayaku: Lines 1, 2, 3, and 10 take in $O(1)$ time. Each line of the while-loop also takes $O(1)$ time each. The loop is executed $O(\log n)$ times. To prove that, observe, that $k$ decreases by the factor of 2 at least every second iteration of the loop. Therefore, the total running time of the algorithm is $O(\log n)$.

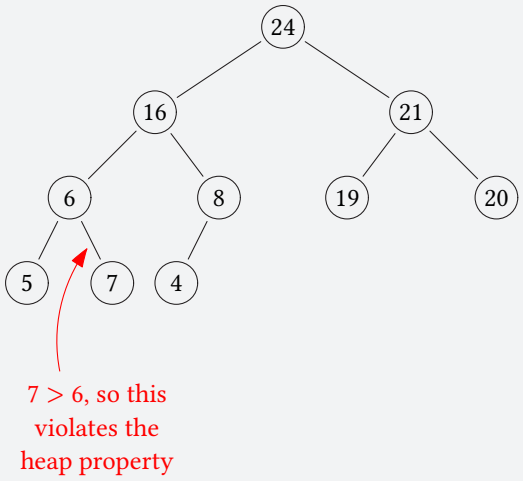
# ▶ Lecture 3 Heaps

**Exercise 12**

(a) *(L1)* Give the definition of a max-heap.

> **Solution:**
> A max-heap is a nearly complete binary tree, filled on all levels except possibly the lowest, with the lowest level filled from left to right, that satisfies the max-heap property: for every node $i$ other than the root $key[Parent(i)] \geq key[i]$.

(b) *(L2)* Is the sequence $\langle 24, 16, 21, 6, 8, 19, 20, 5, 7, 4 \rangle$ a max-heap?

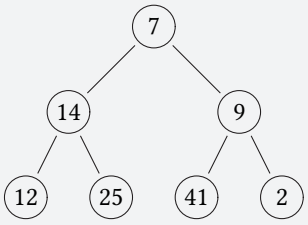
> **Solution:**
> No, it is not. To see this, let us draw the array as a heap:
>
> 
> 

(c) *(L3)* We can store a max-heap in an array. Argue that the $k$-th node on level $j$ of a max-heap is stored at position $2^j + k - 1$.

> **Solution:**
> A heap, when stored in an array, is stored level by level, from left to right. The root occupies the first cell of the array $A[1]$. The left child of the root occupies $A[2]$, and the right child of the root occupies $A[3]$. There are $2^i$ nodes at each level $i$. Thus, the first node of the $j$th level will occupy $A[\sum_{i=0}^{j-1} 2^i + 1]$, and the $k$-th node on level $j$ will occupy $A[\sum_{i=0}^{j-1} 2^i + k]$. The sum $\sum_{i=0}^{j-1} 2^i = 2^j - 1$, therefore, the $k$-th node on level $j$ is stored at position $2^j + k - 1$.

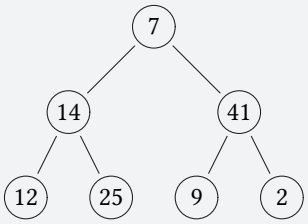
**Exercise 13** *(L2)* Illustrate the execution of HeapSort on the following input: $\langle 7, 14, 9, 12, 25, 41, 2\rangle$. Show every step that changes the array. (*One call of Max-Heapify counts as one step.*)

**Solution:**
The input list:



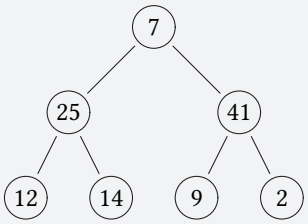

[7, 14, 9, 12, 25, 41, 2]

We start with the Build-Heap phase.
Max-Heapify($A$, 3):



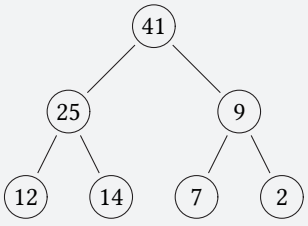

[7, 14, 41, 12, 25, 9, 2]

Max-Heapify($A$, 2):



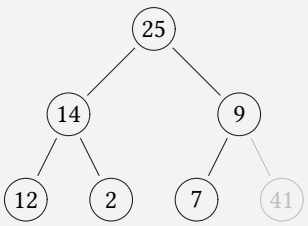

[7, 25, 41, 12, 14, 9, 2]

Max-Heapify($A$, 1):



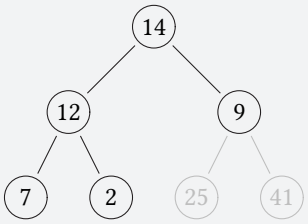

[41, 25, 9, 12, 14, 7, 2]

Now we do the actual sorting.
Swap 2 and 41 and Max-Heapify($A$, 1):



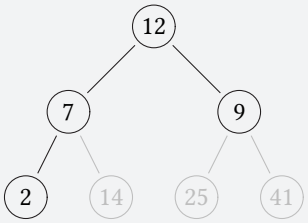

[25, 14, 9, 12, 2, 7, 41]

Swap 7 and 25 and MAX-HEAPIFY($A$, 1):



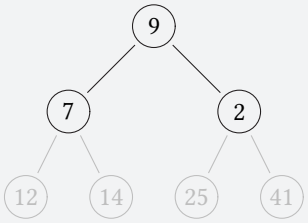

[14, 12, 9, 7, 2, 25, 41]

Swap 2 and 14 and MAX-HEAPIFY($A$, 1):



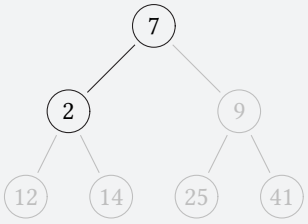

[12, 7, 9, 2, 14, 25, 41]

Swap 2 and 12 and MAX-HEAPIFY($A$, 1):



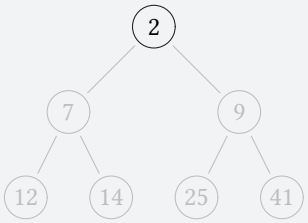

[9, 7, 2, 12, 14, 25, 41]

Swap 2 and 9 and MAX-HEAPIFY($A$, 1):



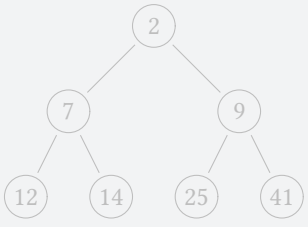

[7, 2, 9, 12, 14, 25, 41]

Swap 2 and 7 and MAX-HEAPIFY($A$, 1):




[2, 7, 9, 12, 14, 25, 41]

Now we are done:




[2, 7, 9, 12, 14, 25, 41]

**Exercise 14** *(L4)* Describe a linear time algorithm MaxHeapVerify($A$) that checks whether a given array $A$ is a max-heap. Prove its correctness using the loop invariant, and analyze the running time.

**Solution:**
To verify if a given array is a max-heap, we need to check whether each node satisfies the max-heap property, that is, that $key[Parent(i)] \geq key[i]$ for all nodes $i$ except for the root. In a max-heap, the parent of node $i$ is stored at the position $\lfloor i/2 \rfloor$. Thus, we can linearly scan the array from right to left and test for each node whether the max-heap property is satisfied.

```
MaxHeapVerify(A)
    Input: array A
1   for i = A.length downto 2
2       if A[⌊i/2⌋].key < A[i].key
3           return FALSE
4   return TRUE
```

We will prove the correctness of the algorithm using the following loop invariant:
**Loop Invariant:** for each element in $A[i+1 : A.length]$ the key of the parent of the element is larger or equal than the key of the element itself.
**Initialization:**
At the start we have $i = A.length$. The subarray $A[i+1 : A.length] = A[A.length + 1 : A.length] = \emptyset$. As there are no elements in the empty set, trivially the claim holds for each element in this set. Thus the loop invariant holds.
**Maintenance:**
Assume the loop invariant holds at the start of an arbitrary iteration $i$. Then for each element in $A[i+1 : A.length]$ the key of the parent of the element is larger or equal than the key of the element itself. We need to prove that the loop invariant also holds at the start of the next iteration.
Within an iteration we check if the key of the parent of the element $A[i]$ is greater or equal to $A[i].key$. If it does not hold then the algorithm terminates and reports FALSE. Trivially this is correct as at least for one element the max-heap property does not hold.
If the algorithm does not terminate, then the key of the parent of $A[i]$ is larger or equal than the key of $A[i]$ itself. Combining this with the our initial observation by the LI we get that for each element in $A[i : A.length]$ the key of the parent of the element is larger or equal than the key of the element itself. Thus, at the start of the next iteration (with $i-1$) indeed the LI holds.
**Termination:**
The loop terminates when $i \geq 2$ no longer holds, so $i = 1$. Combining this with the loop invariant we get that for each element in $A[2 : A.length]$ the key of the parent of the element is larger or equal than the key of the element itself. Thus, the max-heap property holds for all the non-root nodes of the heap. (As the root has no parent, we do not need to check the claim for the root.) Thus the heap is a max-heap and the algorithm correctly returns TRUE.

**Running time:**
Each line of the algorithm takes $O(1)$ time, and the loop makes $n-1$ iterations. Thus, the total running time of the algorithm is $O(n)$.