ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
Gleb Novikov, Tommaso d'Orsi, Ulysse Schaller, Rajai Nasser

25 October 2021

Algorithms & Data Structures — Exercise sheet 5 — HS 21

Exercise Class (Room & TA): ____________________

Submitted by: ____________________

Peer Feedback by: ____________________

Points: _____

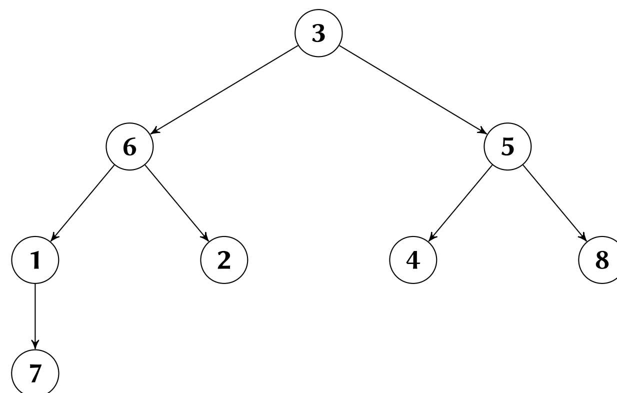
Submission: On Monday, 1 November 2021, hand in your solution to your TA *before* the exercise class starts. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 5.1 *Heapsort* **(1 point)**.

Given the array $[3, 6, 5, 1, 2, 4, 8, 7]$, we want to sort it in ascending order using Heapsort.

a) Draw the tree interpretation of the array as a heap, before any call of RestoreHeapCondition.

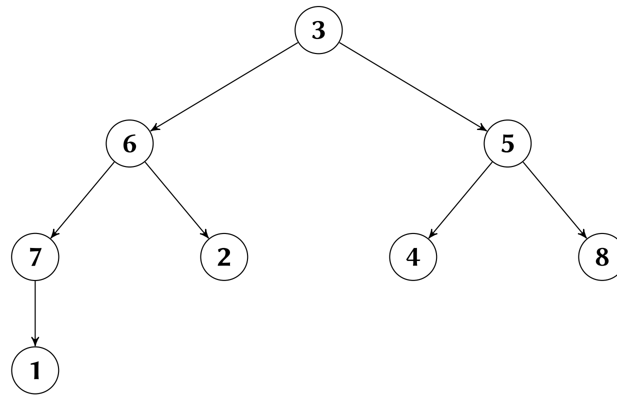
Solution:



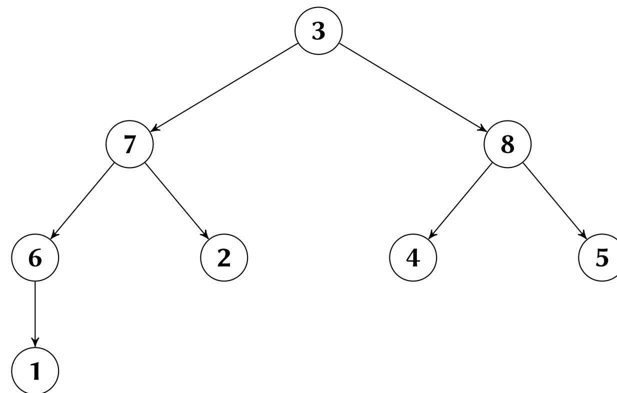
b) In the lecture you have learned a method to construct a heap from an unsorted array (see also pages 35–36 in the script). Draw the resulting max heap if this method is applied to the above array.

Solution:

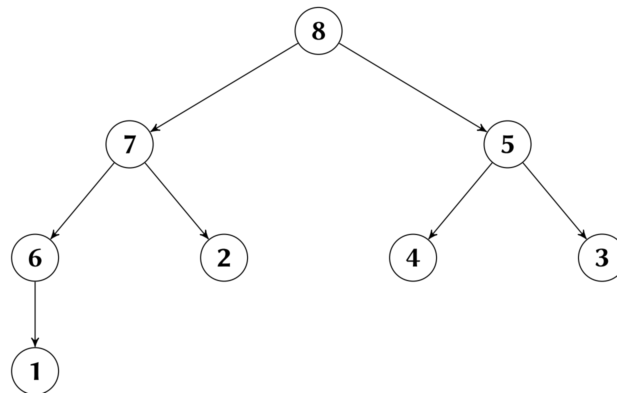
We start from the heap drawn above. The root of the heap is at level 0. Heapifying the subtree with root at level 2 yields:



Then, heapifying the subtrees with roots at level 1 yields:



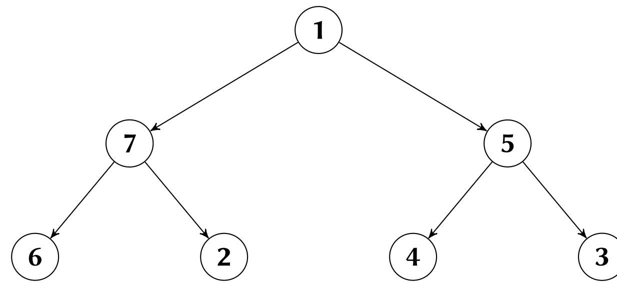
Finally, heapifying the subtree at the root node yields



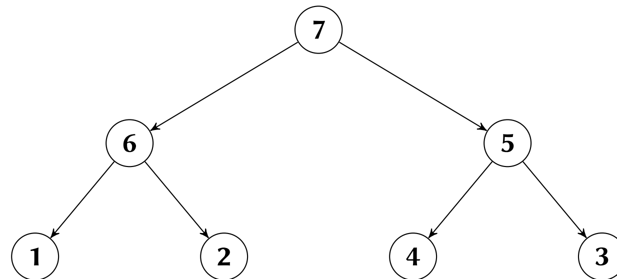
which corresponds to the array $[8, 7, 5, 6, 2, 4, 3, 1]$.

c) Sort the above array in ascending order with heapsort, beginning with the heap that you obtained in (b). Draw the array after each intermediate step in which a key is moved to its final position.

Solution: We begin with the max heap $[8, 7, 5, 6, 2, 4, 3, 1]$. We extract the root 8 and put it into the last position in the array, i.e., we swap 8 with the last element 1, removing 8 from the heap, which yields



We then sift 1 downwards until the heap condition is restored:



Now, the array is $[7, 6, 5, 1, 2, 4, 3, 8]$ and contains the one-smaller heap in the front and the sorted entries in the end.

The array after the subsequent steps are as follows. Blue letters are at their final positions.

1) Swap 7 and 3: $[3, 6, 5, 1, 2, 4, 7, 8]$
   Sift 3 down: $[6, 3, 5, 1, 2, 4, 7, 8]$
2) Swap 6 and 4: $[4, 3, 5, 1, 2, 6, 7, 8]$
   Sift 4 down: $[5, 3, 4, 1, 2, 6, 7, 8]$
3) Swap 5 and 2: $[2, 3, 4, 1, 5, 6, 7, 8]$
   Sift 2 down: $[4, 3, 2, 1, 5, 6, 7, 8]$
4) Swap 4 and 1: $[1, 3, 2, 4, 5, 6, 7, 8]$
   Sift 1 down: $[3, 1, 2, 4, 5, 6, 7, 8]$
5) Swap 3 and 2: $[2, 1, 3, 4, 5, 6, 7, 8]$
   Sift 2 down: $[2, 1, 3, 4, 5, 6, 7, 8]$
6) Swap 2 and 1: $[1, 2, 3, 4, 5, 6, 7, 8]$
   done: $[1, 2, 3, 4, 5, 6, 7, 8]$.

We are done.

Exercise 5.2 *Sorting algorithms* ***(1 point)***.

Below you see four sequences of snapshots, each obtained during the execution of one of the following algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

3	6	5	1	2	4	8	7
3	6	5	1	2	4	8	7
3	5	6	1	2	4	8	7

_____InsertionSort_____

3	6	5	1	2	4	8	7
3	5	1	2	4	6	7	8
3	1	2	4	5	6	7	8

_____BubbleSort_____

3	6	5	1	2	4	8	7
3	6	1	5	2	4	7	8
1	3	5	6	2	4	7	8

_____MergeSort_____

3	6	5	1	2	4	8	7
1	6	5	3	2	4	8	7
1	2	5	3	6	4	8	7

_____SelectionSort_____

Exercise 5.3 *Counting Operations in Loops II.*

For the following code fragments count how many times the function f is called. Report the number of calls as nested sum, and then simplify your expression in Θ-notation and prove your result.

Hint: *Note that in order to justify your Θ-notation you are required to show two parts: an upper bound on your nested sum as well as a lower bound.*

a) Consider the snippet:

Algorithm 1

```
for j = 1, ..., n do
    k ← 1
    while k ≤ j do
        m ← 1
        while m ≤ j do
            f()
            m ← 2 · m
        k ← 2 · k
```

Solution: f is called

$$\sum_{j=1}^{n}\sum_{l=0}^{\lfloor \log_2 j\rfloor}\sum_{i=0}^{\lfloor \log_2 j\rfloor} 1 = \sum_{j=1}^{n}(\lfloor \log_2 j\rfloor + 1)^2 \leq \sum_{j=1}^{n}(\lfloor \log_2 n\rfloor + 1)^2 \leq O(n \log^2 n)$$

times. Notice that, when $n \geq 4$, then $\log_2(n/2) = \log_2 n - 1 \geq (\log_2 n)/2$. Therefore, for all $n \geq 4$ we have

$$\sum_{j=1}^{n}\sum_{l=0}^{\lfloor \log_2 j\rfloor}\sum_{i=0}^{\lfloor \log_2 j\rfloor} 1 = \sum_{j=1}^{n}(\lfloor \log_2 j\rfloor + 1)^2 \geq \sum_{j=\lceil n/2\rceil}^{n}(\lfloor \log_2(n/2)\rfloor + 1)^2$$

$$\geq \sum_{j=\lceil n/2\rceil}^{n} \log_2(n/2)^2 \geq n/2 \cdot ((\log_2 n)/2)^2 \geq \Omega(n \log^2 n),$$

so actually we have

$$\sum_{j=1}^{n} \sum_{l=0}^{\lfloor \log_2 j \rfloor} \sum_{i=0}^{\lfloor \log_2 j \rfloor} 1 = \Theta(n \log^2 n).$$

b) Consider the snippet:

Algorithm 2

---

for $j = 1, \ldots, n$ **do**
  for $l = 1, \ldots, 100$ **do**
    $k \leftarrow 1$
    while $k^2 \leq j$ **do**
      $f()$
      $f()$
      $k \leftarrow k + 1$

Solution: f is called

$$\sum_{j=1}^{n} \sum_{l=1}^{100} \sum_{k=1}^{\lfloor \sqrt{j} \rfloor} 2 = \sum_{j=1}^{n} 100 \cdot \lfloor \sqrt{j} \rfloor \cdot 2 \leq 200 \sum_{j=1}^{n} \sqrt{n} = 200 n^{3/2} \leq O(n^{3/2})$$

times. Notice that, when $n \geq 24$, then $\lfloor \sqrt{n/2} \rfloor \geq \sqrt{n/4}$. Therefore, for all $n \geq 24$ we have

$$\sum_{j=1}^{n} \sum_{l=1}^{100} \sum_{k=1}^{\lfloor \sqrt{j} \rfloor} 2 \geq \sum_{j=\lceil n/2 \rceil}^{n} \lfloor \sqrt{j} \rfloor \geq \sum_{j=\lceil n/2 \rceil}^{n} \lfloor \sqrt{n/2} \rfloor$$

$$\geq \sum_{j=\lceil n/2 \rceil}^{n} \sqrt{n/4} \geq n/2 \cdot \sqrt{n/4} = n^{3/2}/4 \geq \Omega(n^{3/2}),$$

so actually we have

$$\sum_{j=1}^{n} \sum_{l=1}^{100} \sum_{k=1}^{\lfloor \sqrt{j} \rfloor} 2 = \Theta(n^{3/2}).$$

Exercise 5.4 *Bubble sort invariant.*

Consider the pseudocode of the bubble sort algorithm on an integer array $A[1, \ldots, n]$:

Algorithm 3 BUBBLESORT(A)

---

for $1 \leq i \leq n$ **do**
  for $1 \leq j \leq n - i$ **do**
    if $A[j] > A[j + 1]$ **then**
      $t \leftarrow A[j]$
      $A[j] \leftarrow A[j + 1]$
      $A[j + 1] \leftarrow t$
return A

---

a) Formulate an invariant INV(i) that holds at the end of the i-th iteration of the outer for-loop.

Solution: After i iterations of the outer for-loop, the subarray $A[n-i+1, \ldots, n]$ is sorted and each element from $A[1, \ldots, n-i]$ is not greater than each element from $A[n-i+1, \ldots, n]$.

b) Using the invariant from part (a), prove the correctness of the algorithm. Specifically, prove the following three assertions:

(i) INV(1) holds.

(ii) If INV(i) holds, then INV$(i+1)$ holds (for all $1 \leq i < n$).

(iii) INV(n) implies that BubbleSort(A) correctly sorts the array A.

Solution:

(i) INV(1) means that after the first iteration of the outer for-loop, the largest element of A is at position n. Suppose that this largest element was originally at position j for some $1 \leq j \leq n$. If $j = n$, the element will never be swapped by the first inner for-loop, and hence is still at position n at the end, as desired. For $j < n$, this largest element will be swapped to position $j+1$ in the j-th iteration of the inner for-loop, and then swapped to position $j+2$ in the next iteration, and so on until it is swapped to position n. So in both cases it is at position n at the end of the first for-loop.

(ii) Let $1 \leq i < n$. Assuming that INV(i) holds, we know that before the $(i+1)$st iteration of the outer for-loop, the i last entries of the array are the i largest entries of the input array A sorted in ascending order. Using a similar reasoning as in (i), we see that during the $(i+1)$st iteration, the largest element among the remaining part of the array (namely $A[1, \ldots, n-i]$) will be placed at the last position of this remaining part, so that now the the $i+1$ last entries of the array are the $i+1$ largest entries of the input array in ascending order. Therefore, INV$(i+1)$ holds.

(iii) INV(n) means that the "subarray" $A[1, \ldots, n]$ is sorted. But this is actually the full array (since A has length n) returned by BubbleSort(A), which shows that the algorithm correctly sorts the array A.

Exercise 5.5 *Guessing the parity of a number* **(1 point)**.

Alice and Bob are playing a game where Alice chooses a secret integer $x \in \{1, \ldots, n\}$ and Bob has to guess the parity of x, i.e., Bob has to guess whether x is even or odd. Note that n is a fixed number that Alice and Bob agree on before starting the game. Bob is allowed to ask Alice comparison questions of the form

"Is x greater than y?"

for some $y \in \{1, \ldots, n\}$. Bob is not allowed to ask other forms of questions.

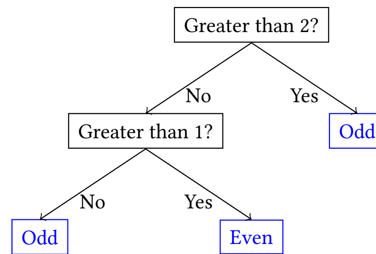
The following is an example of how the game could start:

- Alice and Bob agree on $n = 1000$.
- Alice secretly chooses $x = 541$.
- Bob asks: "Is x greater than 50?"
  - Alice answers "yes".
- Bob asks: "Is x greater than 659?"
  - Alice answers "no".

We emphasize that Bob does not have to guess the exact value of x. He only needs to find the parity of x, and he wishes to achieve this by asking as few questions as possible.

a) Assume that $n = 3$. Devise a strategy of questions for Bob and draw its decision tree

Solution:



b) Now n is arbitrary. Bob has asked i comparison questions and Alice answered these questions. Let $\mathcal{X}_i \subseteq \{1, \ldots, n\}$ be the collection of all numbers that are consistent with Alice's answers[1]. Show that $\mathcal{X}_i$ is a contiguous subset of $\{1, \ldots, n\}$, i.e., there exist $a, b \in \{1, \ldots, n\}$ such that

$$\mathcal{X} = \Big\{ y \in \{1, \ldots, n\} : \; a \leq y \leq b \Big\}.$$

***Hint:** You can prove this by induction on the number of questions i.*

Solution: If Bob has not asked any question yet, then

$$\mathcal{X}_0 = \{1, \ldots, n\} = \Big\{ y \in \{1, \ldots, n\} : \; 1 \leq y \leq n \Big\}.$$

Now let $i \geq 0$ and suppose that after asking i questions, the set $\mathcal{X}_i$ satisfies

$$\mathcal{X}_i = \Big\{ y \in \{1, \ldots, n\} : \; a \leq y \leq b \Big\},$$

for some $a, b \in \{1, \ldots, n\}$. Now assume that the $(i + 1)^{th}$ question of Bob was

"Is x greater than c?"

for some $c \in \{1, \ldots, n\}$. If the answer is yes, then

$$\mathcal{X}_{i+1} = \Big\{ y \in \{1, \ldots, n\} : \; \max\{a, c + 1\} \leq y \leq b \Big\}.$$

If the answer is no, then

$$\mathcal{X}_{i+1} = \Big\{ y \in \{1, \ldots, n\} : \; a \leq y \leq \min\{b, c\} \Big\}.$$

Therefore, in all cases, $\mathcal{X}_{i+1}$ remains a contiguous subset of $\{1, \ldots, n\}$. It follows by induction that for every $i \geq 0$, the set $\mathcal{X}_i$ is always contiguous.

c) Show that in any strategy of questions that Bob can follow, Bob cannot reliably guess the parity of x without reliably guessing the number x itself.

***Hint:** Show that after i questions, Bob cannot reliably guess the parity of x unless $\mathcal{X}_i$ contains a single number.*

Solution: Bob cannot reliably guess the parity of x unless all the numbers in $\mathcal{X}_i$ have the same parity.

[1] Consider the example above with $x = 541$. After Bob's two questions, the set $\mathcal{X}_2$ contains $x = 541$ but also contains numbers such as 51, 141, 513 and 659.

Suppose that $\mathcal{X}_i$ contains more than one element. Since $\mathcal{X}_i$ is always contiguous, then $\mathcal{X}_i$ must contain at least two consecutive numbers. By noticing that consecutive numbers have different parities, we can see that Bob cannot reliably guess the parity of x unless $\mathcal{X}_i$ contains a single number. In other words, Bob cannot reliably guess the parity of x unless Bob can reliably guess the number x itself.

d) Show that in any strategy of questions that Bob can follow, the number of questions that are required to reliably guess the parity of x is at least $\lceil \log_2(n) \rceil$ in the worst case.

Solution: We will follow an argument that is similar to the one we saw in class.

Imagine the decision tree of Bob's strategy. From c) we know that when we reach a leave of the tree, Bob must be able to determine the number x. Therefore, in any strategy, there is at least n leaves in the decision tree.

Now since the decision tree is a binary tree, its height must be at least $\lceil \log_2(n) \rceil$. Therefore, the number of questions that are required to reliably guess the parity of x is at least $\lceil \log_2(n) \rceil$ in the worst case.

Remark. This exercise shows that guessing the parity of a number using only comparison oracles is as hard as guessing the number itself. Note that if we had access to other oracles, we might be able to guess the parity of x more efficiently.