Exercise Sheet 5: Answers
COMS10017 Algorithms 2023/2024

Reminder: $\log n$ denotes the binary logarithm, i.e., $\log n = \log_2 n$.

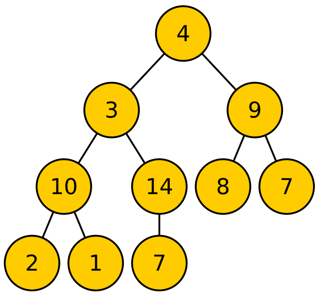
1 Heapsort

Consider the following array A:

4	3	9	10	14	8	7	2	1	7

1. Interpret A as a binary tree as in the lecture (on heaps) and draw the tree.

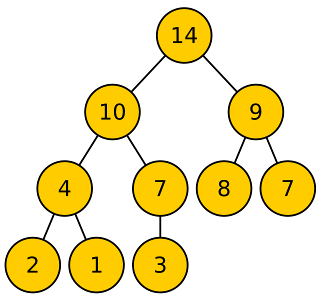
   Solution.

   ✓

2. Run Create-Heap() on the initial array. Give the sequence of node exchanges. Draw the resulting heap.

   Solution. The resulting heap looks as follows:

   The sequence of node exchanges is: $14 \leftrightarrow 3$, $3 \leftrightarrow 7$, $4 \leftrightarrow 14$, $4 \leftrightarrow 10$ ✓

3. What is the worst-case runtime of Create-Heap() and how is the runtime established?

Solution. The worst-case runtime of Create-Heap() is $O(n)$, see lectures. ✓

4. Explain how Heapsort uses the heap for sorting. Explain why the algorithm has a worst-case runtime of $O(n \log n)$.

Solution. Bookwork, see lectures. ✓

2 Heapsort: An Alternative to Create-Heap()

Let A be an integer array of length n. Heapsort interprets the input array A as a binary tree, and the Create-Heap() function shuffles the elements of A such that a valid heap is obtained, i.e., the heap property is fulfilled at every node. In this exercise, we will analyse an alternative to the Create-Heap() function that uses the auxiliary function Heapify-Up():

Heapify-Up(c) is called on a node c of the tree. It operates as follows. If the value stored at c is smaller or equal to the value stored at c's parent then do nothing. Otherwise, the value stored at c is larger than the value stored at c's parent. In this case, exchange c and c's parent. Heapify-Up() is then called recursively on the new location of c.

Based on Heapify-Up(), we now consider the function Alt-Create-Heap():

Algorithm 1 Alt-Create-Heap()

Require: Array A of n integers

```
for i = 1 to n − 1 do
    Interpret the prefix array A[0, ..., i] as a binary tree as in the lectures
    Run Heapify-Up(c) on the node c associated with A[i]
end for
```

1. Consider the prefix $A[0, \dots, i]$. What is the runtime of Heapify-Up(c) when called on the node c associated with $A[i]$?

Solution. The runtime of Heapify-Up(c) is bounded by the number of times Heapify-Up(c) is called recursively. In each recursive call, the node c is moved up one step in the tree. This process ends either when the parent node of c has a larger value than the value stored at c or when c becomes the root of the tree. In both cases, the runtime is bounded by the height of the tree, i.e., $O(\log(i))$. ✓

2. What is the runtime of Alt-Create-Heap()?

Solution. The runtime of Heapify-Up() on a node in a tree that consists of i nodes is $O(\log(i))$. The algorithm calls Heapify-Up() on nodes in trees of sizes $n = 2, \dots, n$. Thus, the runtime can be bounded by:

$$\sum_{i=2}^{n} O(\log i) \leq \sum_{i=2}^{n} O(\log n) = (n-1) \cdot O(\log n) = O(n \log n) \ .$$

One may wonder whether the inequality $\sum_{i=2}^{n} O(\log i) \leq \sum_{i=2}^{n} O(\log n)$ is not sufficiently tight and a better bound could be proved. However, as demonstrated by the following calculation, this is not the case:

$$\sum_{i=2}^{n} O(\log i) \geq \sum_{i=\lceil n/2 \rceil}^{n} O(\log i) \geq \sum_{i=\lceil n/2 \rceil}^{n} O(\log \lceil n/2 \rceil) \geq n/2 \cdot O(\log \lceil n/2 \rceil) = O(n \log n) \ .$$

✓

3. Prove the following loop-invariant:

At the beginning of iteration i, the binary tree associated with the prefix $A[0, \dots, i-1]$ constitutes a heap.

Conclude that Alt-Create-Heap() indeed creates a valid heap.

Solution.

1. *Initialization* (i=1): We need to argue that the associated tree with the array $A[0, \dots, 0]$, which is simply the single element $A[0]$, constitutes a valid heap. This is trivially true since a single node always fulfills the heap property.

2. *Maintenance:* Assume now that, before iteration i, the loop-invariant holds, i.e., the tree associated with $A[0, \dots, i-1]$ constitutes a heap. We will now show that before iteration $i+1$, or, equivalently, after iteration i, the tree associated with $A[0, \dots, i]$ constitutes a heap.

   To this end, denote by T_i the heap associated with $A[0, \dots, i-1]$ before iteration i. In iteration i, the element $A[i]$ is added to T_i at the right-most position in the lowest level and pushed upwards using the Heapify-Up() function. Denote by c the node in the tree that corresponds to $A[i]$. Furthermore, denote by $T_i = T_i^0, T_i^1, T_i^2, \dots, T_i^k$ the sequence of trees obtained by the recursive calls of Heapify-Up(), where T_i^k is the tree obtained when Heapify-Up() terminated without any further recursive calls. We will now argue that T_i^k is a valid heap. First, observe that in T_i^0, the heap property is only violated at the parent node of c. In T_i^1, the positions of c and the parent of c are exchanged. Observe that after the exchange, the resulting tree is such that the heap property is only violated at the new parent of c. More generally, we observe that in T_i^j, for $j < k$, the heap property is only violated at the parent of c. When Heapify-Up() terminates then either c became the root of the tree and c does not have a parent which could violate the heap property, or the value stored at the parent of c is larger than the value stored at c, and the parent of c does not violate the heap property. Hence, T_i^k is a valid heap, which establishes the maintenance part of the loop-invariant.

3. *Termination:* After the last iteration of the loop, which corresponds to the state before a virtual iteration $i = n$ that is never executed, we obtain from the loop-invariant that the tree associated with $A[0, \dots, n-1] = A$ constitutes a heap, which completes the proof.

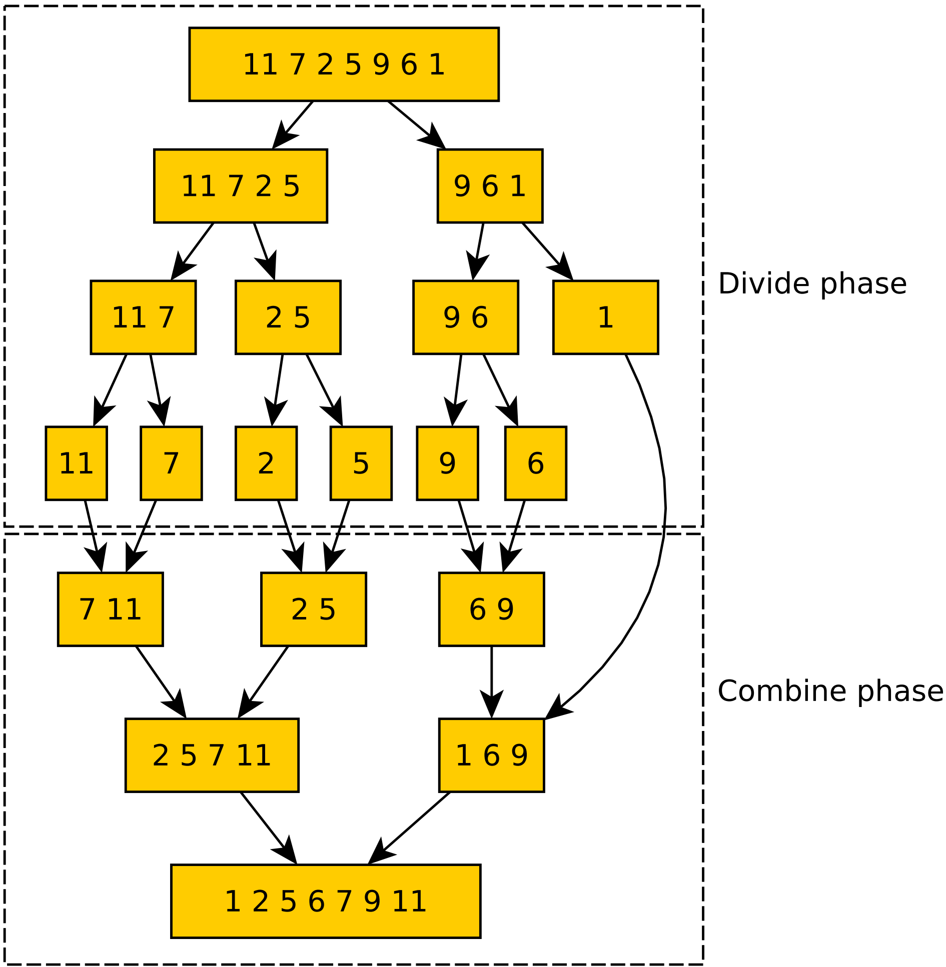
✓

3 Mergesort

Illustrate how the Mergesort algorithm sorts the following array using a recursion tree:

11 7 2 5 9 6 1

Solution.



✓

4 Circularly Shifted Arrays

Suppose you are given an array A of length n of **distinct** (all integers are different) sorted integers that has been circularly shifted by k positions to the right. For example, $[35, 42, 5, 15, 27, 29]$ is a sorted array that has been circularly shifted by $k = 2$ positions, while $[27, 29, 35, 42, 5, 15]$ has been shifted by $k = 4$ positions. Describe an $O(\log n)$ time algorithm that allows us to find the maximum element.

Solution. Before we state our algorithm we discuss a property of circularly shifted sorted arrays:

For $0 \leq q \leq n-1$, observe that $A[(q+1) \mod n] < A[q]$ holds if and only if $A[q]$ is the maximum in A. Hence, for a given position q, we can check in time $O(1)$ whether $A[q]$ constitutes the maximum.

Our algorithm is similar to a binary search. This can be implemented as follows:

1. We initialize $\ell = 0$ and $r = n - 1$ and we will make sure that the maximum will be in the subarray $A[\ell, r]$. This is trivially true after this initialization.

2. In each step of the binary search, we inspect the element in the middle between ℓ and r, i.e., at position $p = \lfloor \frac{\ell + r}{2} \rfloor$. First, we check in time $O(1)$ whether $A[p]$ constitutes the maximum. If it does then we are done. Otherwise, we compare $A[\ell]$ to $A[p]$. If $A[\ell] > A[p]$ then we know that the maximum must be contained in $A[\ell, p-1]$. We then set $r = q - 1$ and we repeat the binary search step. If $A[\ell] < A[p]$ then the maximum is necessarily located in $A[p+1, r]$. We then set $\ell = p + 1$ and repeat the binary search step.

✓

5 Optional and Difficult Questions

Exercises in this section are intentionally more difficult and are there to challenge yourself.

5.1 "Is this the simplest (and most surprising) sorting algorithm ever?", Stanley P. Y. Fung

Please read and appreciate chapters 1 and 2 of the following paper, published in 2021:

`https://arxiv.org/pdf/2110.01111.pdf`