Exercise Sheet 5
COMS10017 Algorithms 2020/2021

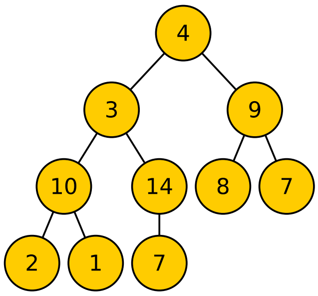
1 Heap Sort

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).

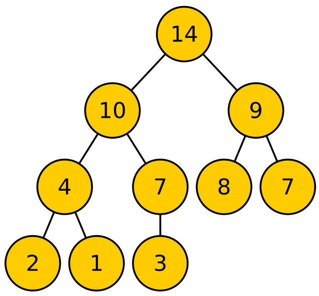
   Solution.



✓

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 are: $14 \leftrightarrow 3$, $3 \leftrightarrow 7$, $4 \leftrightarrow 14$, $4 \leftrightarrow 10$ ✓

3. What is the worst-case runtime of Heapify()?

Solution. As discussed in the lecture, Heapify() runs in time $O(\log n)$. This corresponds to the maximum height of a complete binary tree on n elements. ✓

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

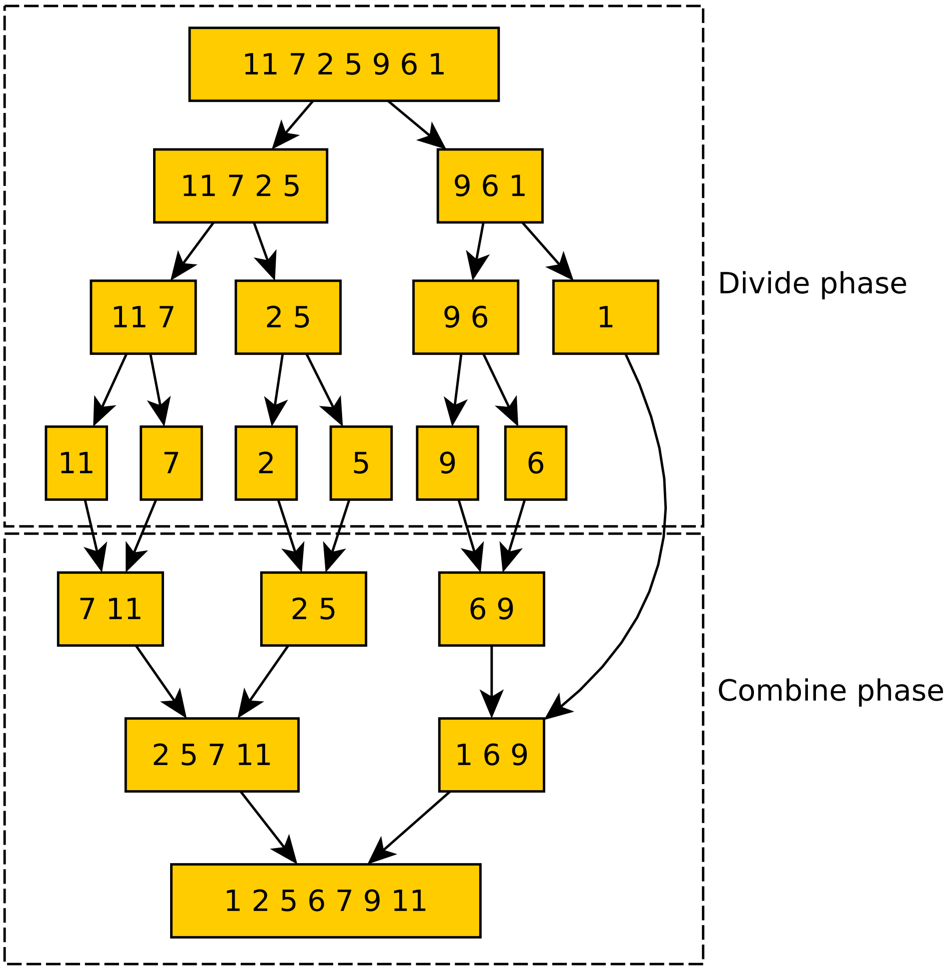
Solution. See lecture. ✓

2 Merge Sort

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

11 7 2 5 9 6 1

Solution.



✓

3 Quick Sort

Consider an array A of length n so that $A[i] = n - i$. For example, for $n = 10$ we are given the following array:

$$A = 10 \quad 9 \quad 8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1\,.$$

The goal is to sort A in non-decreasing order which in this case is equivalent to reversing it. The pivot plays a central role in Quicksort. Consider the following options as a choice for the pivot:

1. The right-most position.
2. The element at position $\lceil n/2 \rceil$.
3. The left-most position.

For each of these options, what is the runtime of Quicksort on A? State your answers using $\Theta(.)$-notation. Justify your answers.

Solution.

1. In this case, the pivot is always the smallest element of the subarray. Every array of length k considered is then split into an array of length $k - 1$, the pivot, and an empty array. This yields a runtime of $\Theta(n^2)$.
2. This is a very good split as every array of length k is split roughly two equal halves. This yields a runtime of $\Theta(n \log n)$.
3. Similar to the first case, this leads to one empty subarray. The runtime is therefore $\Theta(n^2)$.

✓

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 $A[q]$. If $A[\ell] > A[q]$ then we know that the maximum must be contained in $A[\ell, q-1]$. We then set $r = q - 1$ and we repeat the binary search step. If $A[\ell] < A[q]$ then the maximum is necessarily located in $A[q+1, r]$. We then set $\ell = q + 1$ 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 Closest Pair of Points (hard!)

The input consists of two arrays of n real numbers X, Y and represent n points with coordinates $(X[0], Y[0]), (X[1], Y[1]), \ldots, (X[n-1], Y[n-1])$. Give a divide-and-conquer algorithm that finds the pair of points that are closest to each other, i.e., the output consists of a two indices i, j such that $(X[i], Y[i])$ and $(X[j], Y[j])$ are the two closest points.

Hint: This algorithm is similar to the algorithm given for the Maximum Subarray problem. The combine step is tricky here. It is easy to give a combine step that runs in $O(n^2)$ time. How can we get a combine step that runs in $O(n)$ time?