Exercises third lecture

Tommaso Tarchi

Ex.1

This loop invariant can be used to prove the correctness of the algorithm.

At the beginning of the loop i=n. Previously we have called the Build-Max-Heap() function on the entire array, so it is a max-heap containing all the smallest elements of the array (since it contains all the elements). The array A[i+1, ..., n] is empty, so it trivially contains the sorted largest elements.

At each iteration of the loop the invariant is true: if at a certain iteration A[1, ..., i] is a max-heap, picking the first element of the array means picking the largest one, and if A[i+1, ..., n] contains all the largest elements sorted, appending the picked element in the first position means having the sorted largest n-i+1 elements in A[i, ..., n]. Also, at the end of the loop we call Max-Heapify() on the first element of the subarray with the first i-1 elements, so that we make A[1, ..., i-1] a max-heap (since this subarray is a max-heap except for the first element).

Ex.2

1. If the node has index i, then the parent node can be found at index $\lfloor \frac{i-2}{d} \rfloor + 1$, while the k-th child has index equal to $(i-1)\,d + 1 + k$.
2. Calling l the height of the tree, we have:

$$n = \Sigma_{i=0}^{l} d^i = \frac{1 - d^{l+1}}{1 - d} \sim d^l,$$

so $l \sim \log_d n$.

Ex.3

Considering the standard quick sort algorithm, the case in which all elements have the same value is a worst case scenario. In fact, the first call will place (by default of Partition()) the pivot in the last position of the array, then all the array will have to be scanned and all the elements will be left in their position; so that the resulting partition will consist of the entire array but the last element and an array with just the last element. The same thing will happen at each iteration.

Therefore, the time complexity will be equal to $\sim n^2$ (basically: since the condition to move an element of the array in Partition() is $\leqslant$, this case is exactly equivalent to the one in which the entire array is sorted).

Ex.4

To compute the expected time complexity we can think of a case in which the tree representation of the algorithm is (more or less) balanced. Considering that the quick sort algorithm is stopped when partitions contain less than k elements, the tree corresponding to the quick sort part of the algorithm is long $\log \frac{n}{k}$, and the corresponding complexity is $O\left(n \log \frac{n}{k}\right)$.

At this point we can think of having $\frac{n}{k}$ leaves with $\sim k$ elements each. So, remembering that the complexity of insertion sort on the average case is $O\left(n^2\right)$, the time complexity of this part is $O\left(k^2 \frac{n}{k}\right) = O\left(kn\right)$. Therefore, the total complexity is $O\left(kn + n \log \frac{n}{k}\right)$.

However, if we consider the worst case we still have a complexity of $O\left(n^2\right)$. In fact, let's consider the case in which the k smallest elements are placed in the first k positions of the array, and the following n-k elements are sorted. In this case, on the first call of Quicksort() all the array has to be scanned leaving all elements in their original position, and partitioning the array in a subarray with all elements but the last one and one with only the last element. The same goes for each iteration until the pivot is the (k+1)-th element. In this

case, independently from the complexity of the insertion sort part of the algorithm, the total complexity is $O\left(n^2\right)$, since the quick sort part has to scan $n + (n-1) + ... + (n-k) = \frac{n(n-k+1)}{2} \sim n^2$ elements.