Assignment 2

Egor Kuziakov

3.1. Quicksort fused with insertion sort

1. Basically, *quick sort* runs in $O(n\,logn)$ time.
   But if we stop quick sort when subarrays are of length k, time complexity becomes $O(n\,log\frac{n}{k})$, because depth of sort becomes k times smaller.

   Basically, *insertion sort* runs in $O(n^2)$ time.
   But if we have subarrays of size k as a result of *quick sort*, now we just should sort each subarray with *insertion sort*. To do this we check each element's position in it's subarray of length k. So, time complexity becomes $O(nk)$.

   Overall time complexity is $O(n\,log\frac{n}{k}) + O(nk) = O(nk + n\,log\frac{n}{k})$.

2. **In theory**, to choose k we should make an equation where we compare quick sort time complexity and quick sort + insertion sort time complexity:
   $n\,logn \geq nk + n\,log\frac{n}{k}$
   $logn \geq k + logn - logk$
   $logk \geq k$

   In practice, we can add constants to make equation possible, because we removed O:
   $c_1 n\,logn \geq c_2 nk + c_1 n\,log\frac{n}{k}$
   $c_1 logn \geq c_2 k + c_1 logn - c_1 logk$
   $c_1 logk \geq c_2 k$

3.2. Binary Search Trees with Equal Keys

1. The element will always satisfy the one condition (entering only the right node or only the left node). Tree will just become a singly linked list. So when we insert an we always got through as many elements as tree has and insert after that.
   i.e. when we insert 1st element, we just insert, 2nd element - go through 1 element and insert, 3rd - go through 2 elements and insert…
   So the time complexity is $1 + 2 + 3 + \ldots + n = \frac{(n+1)n}{2} = \Theta(n^2)$.
2. In that case we make next level of tree only when previous is filled in.
   To insert in such tree we need $\Theta(logn)$ estimated time.
   To insert n elements we need $\Theta(nlogn)$.

3. In that case insertion of each each element needs $\Theta(1)$ time. So to insert n elements we need $\Theta(n)$ time.
4. In worst case we just get tree as singly linked list and time complexity is $\Theta(n^2)$.
   Expected time complexity is $\Theta(nlogn)$, because in that case we approach to balanced tree.

3.3. d-ary heaps

1. If indexes start from 0 then:
   Indexes of children for parent with index i are from $di + 1$ to $di + d$, because for every parent there are d children.
   Index of parent for child with index i is $\frac{i-1}{d}$.
2. Since in the binary heap, height of the heap is $logn$, height d-ary heap is $log_d n$.
3. To extract max we should also get first element and replace it with last. Next we should compare new root element with all its children and swap if needed (the heap condition is not met), repeat this while swap is needed (heapify).
   Worst case running time is the height * number of children in each followed node = height * d: $\Theta(dlog_d n)$.
4. To insert new element we should insert it at the end of the array and apply “Increase key” to it.
   Worst case running time is height of the heap, because we just go up the heap: $\Theta(log_d n)$
5. For “increaseKey” we should get parent of the element ($\frac{i-1}{d}$) and compare with it: if the heap condition is not met (element is larger than the parent), we should swap.
   Worst case running time is height of the heap: $\Theta(log_d n)$.

3.4. Bottleneck spanning tree

1. Suppose we have MST, whose bottleneck weight is greater that in bottleneck in BST. If we remove MST bottleneck (E), we will get two disconnected graphs.
   If we will add another edge X, which will connect these graphs:
   According to BST weight of X should be less or equal to weight of E.
   But according to MST weight of X should should be greater than weight of E.
   There is a contradiction, so this case is impossible.
2. We remove edges, that are larger than m and check the connectivity of the graph. If graph is connected, the condition has been preserved, if not, bottleneck of BST should be greater than m.
   Removing edges is linear time, checking connectivity for every edge is also linear.