Tuan Nguen

Design and Analysis of Algorithms - Problem Sheet 2

1.

a) Sort (A, p, r) // Initial call will be Sort(A, 0, A.size)

```
if (p+1 < r)
    med = Median'(A, p, r)
    index = Partition'(A, p, r, med)
    Sort(A, p, index)
    Sort(A, index + 1, r)
```

Because the Median' function returns the $\lceil (r-p)/2 \rceil$-th order statistic, both subarrays will be of comparable sizes, and at most $\lfloor \frac{r-p}{2} \rfloor$.

$\Rightarrow T(n) \le 2T(\frac{n}{2}) + O(n)$ (Median' and Partition' are both linear)

$\Rightarrow$ By the Master theorem, the algorithm will run in $O(n \log n)$, because $1 = \log_2 2$.

b) Selection (A, p, r, i) // Initial call will be Selection(A, 0, size, i)

```
x = Median'(A, p, r)
index = Partition(A, p, r, x)
if (index + 1 == i)
    return A(index)
else
    if (i < index + 1)
        Selection(A, p, index, i)
    else
        Selection(A, index + 1, r, i)
```

Median' and Partition are both linear $\Rightarrow O(n)$

$\Rightarrow T(n) \le T(\frac{n}{2}) + O(n)$

$\Rightarrow$ By the master theorem, the program will run in $O(n)$

c) If i is less than $(r-p)/2$, then we want to pad the array with $(-\infty)$. We want to pad with $r-p-2i$ number of $(-\infty)$ in order the i-th order statistic to be the median. After we have paded we apply the Median' function. That will be the desidered answer because we need to find the $i+r-p-2i = r-p-i$-th element, which is the median. Similarly if $i > (r-p)/2$, we can pad with $2i - r + p$ times $(+\infty)$ and apply the Median' function.

2.

The standard algorithm, we need to do $2 \cdot 800^3 - 800^2 = 1023360000$.

Strassen's method will take $(18 \cdot 400^2) + (7 \cdot 18 \cdot 200^2) + (7^2 \cdot 18 \cdot 100^2) + (7^3 \cdot 18 \cdot 50^2) + (7^4 \cdot 18 \cdot 25^2) + (7^5 \cdot (2 \cdot 25^3 - 25^2)) = 573900625$, which is less than the standard algorithm.

3.

a) For this algorithm we need 32 block multiplications, and each block is of size $\frac{n}{4}$. $\Rightarrow$

$$T(n) \leq 32T\left(\frac{n}{4}\right) + O(n^2)$$

$\Rightarrow$ By the master theorem, because $\log_4 32 > 2$, then the algorithm is assymptotically bounded by $O(n^{2.5})$.

~~From the question we know that $T(4^k) = 32T(4^{k-1}) + 144(4^{k-1})^2$. But $144(4^{k-1})^2 = 144 \cdot \frac{4^{2k}}{16} = 9 \cdot 4^{2k}$~~

~~$\Rightarrow T(4^k) = 32T(4^{k-1}) + 9 \cdot 4^{2k} =$~~
~~$= 32\{32T(4^{k-2}) + 9 \cdot 4^{2(k-1)}\} + 9 \cdot 4^{2k} =$~~
~~$= 32^k \cdot 9 \cdot 4^{2 \cdot 0} + 32^{k-1} \cdot 9 \cdot 4^2 + \ldots + 32 \cdot 9 \cdot 4^{2(k-1)} + 9 \cdot 4^{2k} =$~~
~~$= 9 \cdot 32^k \{4^0 + 32^{-1} \cdot 4^2 + \ldots + 32^{-k+1} \cdot 4^{2k-2} + 32^{-k} \cdot 4^{2k}\} =$~~

We want to find c, such that for every integer $n \geq 1$, the following is true: $T(n) \leq c \cdot n^{2.5} = c \cdot 2^{5k}$

We know that $T(4^k) = 32T(4^{k-1}) + 144 \cdot (4^{k-1})^2 =$

$= 32T(4^{k-1}) + \cancel{144}^{9} \cdot \frac{4^{2k}}{\cancel{16}} = 32T(4^{k-1}) + 9 \cdot 4^{2k} =$

$= 9 \cdot 4^{2k} + 2^5 \cdot 9 \cdot 4^{2(k-1)} + \dots + (2^5)^{k-1} \cdot 9 \cdot 4^2 + 32^k \cdot T(1) =$

$= 9 \cdot 2^{4k} + 9 \cdot 2^{4k+1} + \dots + 9 \cdot 2^{5k-1} + 2^{5k} =$

$= 9 \cdot 2^{4k}[1 + 2 + \dots + 2^{k-1}] + 2^{5k} =$

$= 9 \cdot 2^{4k} \cdot (2^k - 1) + 2^{5k} =$

$= 9 \cdot 2^{5k} - 9 \cdot 2^{4k} + 2^{5k} =$

$= 10 \cdot 2^{5k} - 9 \cdot 2^{4k} \leq 10 \cdot 2^{5k}$

$\Rightarrow$ the upper bound for c is 10.

$\Rightarrow T(4^k) \leq 10 \cdot 2^{5k}$

b) We want to solve the inequality:

$10 \cdot 2^{5k} \leq 2 \cdot 2^{6k} - 2^{4k} \quad /: 2^{4k} \neq 0$

$2^k \cdot 10 \leq 2 \cdot 2^{2k} - 1$

Let $x = 2^k \Rightarrow$ the inequality $\Leftrightarrow 0 \leq 2 \cdot x^2 - 10x - 1$

$x_{1,2} = \frac{5 \pm \sqrt{25+2}}{2} \in \{5,1;\ -0,1\}$

$\Rightarrow$ for $x \in (-\infty; -0,1) \cup (5,1, +\infty)$ the inequality is true. But $x > 0 \Rightarrow x \geq 5,1$

$\Rightarrow 2^k \geq 5,1$

$\Rightarrow k \geq \log_2 5,1 = 2,35$

$\Rightarrow n$ has to be at least 26 for this method to be more efficient than the conventional one. But because n is a power of 4, then n has to be more or equal to 64.

4.

a) So the minimum number of elements of a heap of height h is when for all levels $j \in [1, h-1]$ it is filled, and on the last level there is only one node.

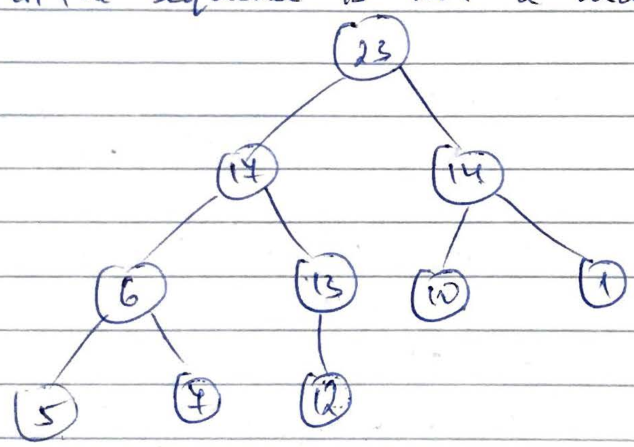
=> the size will be $2^0 + 2^1 + \dots + 2^{k-2} + 1 =$
$= 2^{k-1} - 1 + 1 =$
$= 2^{k-1}$

The maximum number of elements is:
$2^0 + 2^1 + \dots + 2^{k-2} + 2^{k-1} = 2^k - 1$

b) The smallest element resides in one of the leaves. This can be proven by counterpositive. If it is not one of the leaves, then it has a child node. But because it is a max heap that would mean the smallest element is not actually the smallest.

c) A min-heap satisfies the property: for every $i \leq A.heap.size$ is true that $A[i] \geq A[\lfloor \frac{i}{2} \rfloor]$ and $A[i] \geq A[\lfloor \frac{i}{2} \rfloor + 1]$, which is true because the array is ordered.

d) The sequence is not a max-heap, because $A[9] > A[4]$



5.
The worst case for the function "max-heapify" is when the heap is a min-heap. For each of the nodes at height h, the function will run at least $\Omega(h)$, where height is the length of the longest path from the node to a leaf. => $h = \log$(size of the subtree) => for the root, "max-heapify" is $\Omega(\log n)$.

6.

When we remove an element from the heap there will be a gap. Take the subtree, ~~whose root is the~~ ~~node~~ which is the smallest and contains both the node that we want to remove and the last node. If the element we want removed is the root of that subtree, we just replace each of the removed nodes with of the children and ~~the~~ on the last change we take the last node of the tree. If the element is not the root we replace the removed node with ~~one with~~ its parent node until we reach the root and then repeat the same procedure as before. Maximum number of swaps is $2\log_2 n$ => $O(\log n)$.

7.

The running time is $O(n\log n)$ because if the array is sorted, we still have to run the "max-heapify" function at each iteration. Analogically, if the array is sorted in decreasing order it would still take $O(n\log n)$ because at each iteration we need to call "max-heapify".

8.

We take the first entry of each of the sorted arrays and build a ~~max~~ min-heap. Delete the root and print the root - that will be the smallest element. Insert the direct successor of the deleted element from that array. Run „min-heapify" to build the min-heap and repeat the process.