ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Midterm Exam, Algorithms 2016-2017

- You are only allowed to have a handwritten A4 page written on both sides.
- Communication, calculators, cell phones, computers, etc... are not allowed.
- Your explanations should be clear enough and in sufficient detail that a fellow student can understand them. In particular, do not only give pseudo-code without explanations. A good guideline is that a description of an algorithm should be such that a fellow student can easily implement the algorithm following the description.
- You are allowed to refer to algorithms covered in class without reproving their properties.
- **Do not touch until the start of the exam.**

Good luck!

Name: _____ **N° Sciper:** _____

Problem 1 / 20 points	Problem 2 / 30 points	Problem 3 / 28 points	Problem 4 / 22 points

Total / 100

1 *(20 pts)* **Asymptotic growth and heaps.**

1a *(10 pts)* Arrange the following functions in increasing order of asymptotic growth:

$$2^n, \quad n^2 + n/3, \quad n \log n, \quad \log_2 \log_2 n, \quad n^{100/\log_2 n}, \quad n!, \quad 4^{\sqrt{n}}, \quad \sqrt{3^n + 2^n}$$

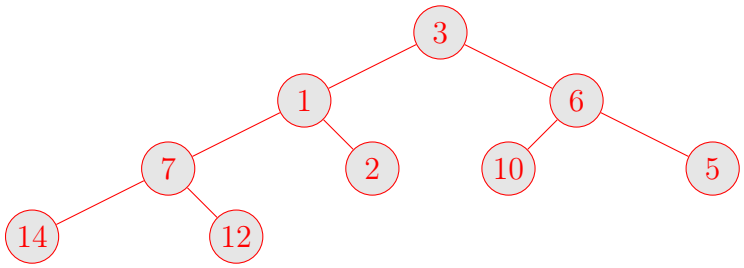
Solution:

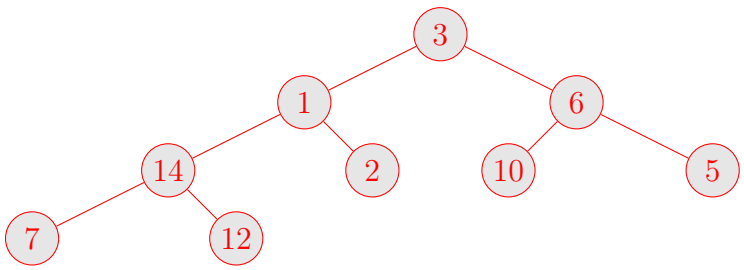
(i) $n^{100/\log_2 n} = n^{100 \log_n 2} = 2^{100}$, which is a constant,
(ii) $\log \log n$ grows slower than any polynomial or exponential function,
(iii) $n \log n$
(iv) $n^2 + n/3$
(v) $4^{\sqrt{n}}$
(vi) $\sqrt{3^n + 2^n} \leq 2(\sqrt{3})^n$
(vii) 2^n
(viii) $n!$

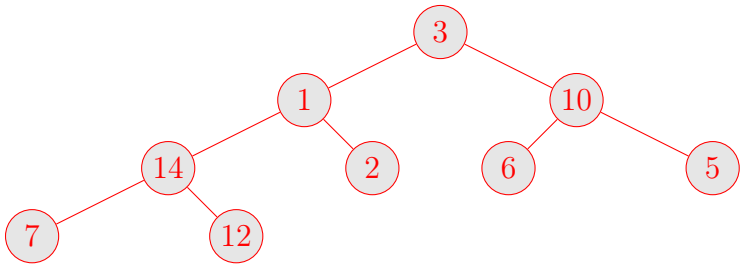
1b *(10 pts)* **Draw** the heap that results from executing BUILD-MAX-HEAP(A, n) on input

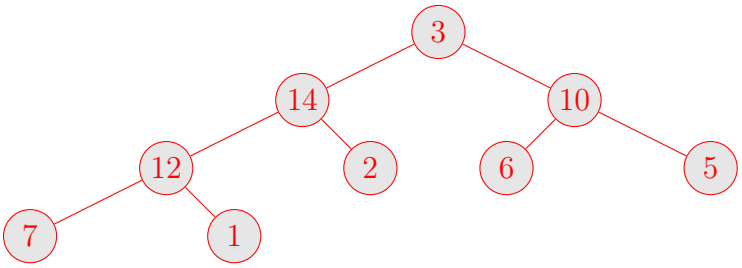
$A =$ | 3 | 1 | 6 | 7 | 2 | 10 | 5 | 14 | 12 | and $n = 9$.

Solution: The steps in the execution are:

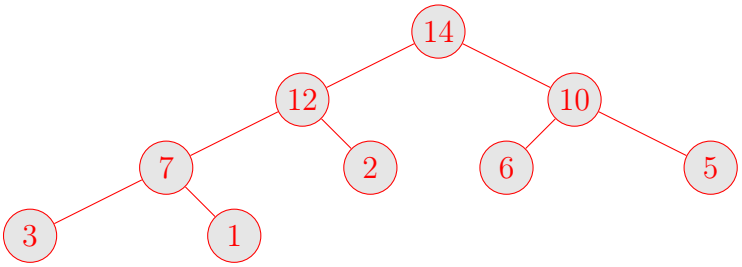








and finally,



(of course, only the last needs to be given in solution).

2 *(30 pts)* **Recurrences.** Consider the following algorithm UNKNOWN that takes as input an integer n:

```
UNKNOWN(n):
1. if n < 50
2.       return
3. q = ⌊n/3⌋
4. UNKNOWN(q)
5. UNKNOWN(n − q)
6. for i = 1 to q
7.       print "I love recurrences!"
8. UNKNOWN(n − q)
```

2a *(10 pts)* Let $T(n)$ be the time it takes to execute UNKNOWN(n). **Give the recurrence relation** for $T(n)$. To simplify notation, you may assume that $n/3$ always evaluates to an integer.

Solution: If $n < 50$, the algorithm UNKNOWN(n) takes $\Theta(1)$ time. Otherwise, it calls three instances of UNKNOWN recursively, once with argument q, and twice with argument $n - q$. In addition, on line 6 it runs a loop which takes time $O(q)$. As we may assume that $n/3$ is an integer, we have $q = n/3 = \Theta(n)$ and $n - q = 2n/3$.

This gives us the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < 50, \\ T(n/3) + 2T(2n/3) + \Theta(n) & \text{otherwise.} \end{cases}$$

2b *(20 pts)* **Prove** tight asymptotic bounds on $T(n)$. Specifically, show that $T(n) = \Theta(n^a)$ for some constant a. You may simplify your calculations by assuming that $\lfloor n/3 \rfloor = n/3$.

Solution: Let us first make an (educated) guess about a. In the inductive proof, we will have an argument roughly like:

$$T(n) = T(n/3) + 2T(2n/3) + \Theta(n) \leq (n/3)^a + 2 \cdot (2n/3)^a + \Theta(n) = n^a((1/3)^a + 2 \cdot (2/3)^a) + \Theta(n)$$

and for the inductive proof to work, we will need that

$$(1/3)^a + 2 \cdot (2/3)^a = 1$$

(with equality because this will give us a tight bound). Equivalently, this is

$$1 + 2 \cdot 2^a = 3^a.$$

After a hopefully short round of trial and error (or binary-searching on paper), we find that $a = 2$ is the solution to this equation. Now we can formally solve the problem:

We prove $T(n) = \Theta(n^2)$:

Claim 1 *There exist positive constants d and e such that $T(n) \leq dn^2 - en$ for all $n \geq 1$.*

Proof. The base case is trivial by selecting d to be larger enough than e.

Now consider the inductive step (with the induction hypothesis that $T(m) \leq dm^2 - em$ for all $m < n$):

$$\begin{aligned} T(n) &= T(n/3) + 2T(2n/3) + cn \\ &\leq d((n/3)^2 + 2(2n/3)^2) - e(n/3 + 2(2n/3)) + cn \\ &= dn^2 - 5en/3 + cn \\ &\leq dn^2 - en \qquad \text{(by selecting } 2e/3 \geq c\text{)}. \end{aligned}$$

□

Claim **??** proves $T(n) = O(n^2)$. For the lower bound, i.e. for $T(n) = \Omega(n^2)$, we show the following:

Claim 2 *There exists a positive constant d such that $T(n) \geq dn^2$ for all $n \geq 1$.*

Proof. Base case is trivial by selecting d small enough.

Now the inductive step (with the induction hypothesis that $T(m) \geq dm^2$ for all $m < n$):

$$\begin{aligned} T(n) &= T(n/3) + 2T(2n/3) + cn \\ &\geq d((n/3)^2 + 2(2n/3)^2) + cn \\ &= dn^2 + cn \\ &\geq dn^2. \end{aligned}$$

□

3 *(28 pts)* In this problem you will design an efficient algorithm for measuring distance between two strings. Your input is two strings $S = (s_1, s_2, \ldots, s_n)$ and $T = (t_1, t_2, \ldots, t_m)$, where n and m are the lengths of S and T respectively. Our measure of distance between S and T is the smallest number of deletions, insertions or substitutions that one has to make to T to make it equal to S.

For example, the distance between the strings $S =$ 'albatros' and $T =$ 'abbbas' is 5, via the sequence of operations shown below:

$$abbbas \xrightarrow{\text{delete 'b'}} abbas \xrightarrow{\text{substitute 'b' with 'l'}} albas \xrightarrow{\text{insert 't'}} albats \xrightarrow{\text{insert 'r'}} albatrs \xrightarrow{\text{insert 'o'}} albatros$$

3a *(8 pts)* Suppose that only substitutions, but no insertions and deletions, are allowed, and the strings S and T have the same length n. Give an algorithm for computing the minimum number of substitutions that are needed to turn T into S. For full credit your algorithm should run in $O(n)$ time.

Solution:

As stated in the problem statement, we are only allowed to substitute the elements of the string T. Therefore, if $s_i \neq t_i$, for some $1 \leq i \leq n$, then we have to substitute t_i; thus we need to count the number of such i. This we can do by going over the arrays once and simply counting the number of indices where they are not equal. The running time of our algorithm is $\Theta(n)$ since we traverse the array exactly once.

3b *(20 pts)* Now suppose that all three operations (insertion, deletion, substitution) are allowed, and the strings $S = (s_1, s_2, \ldots, s_n)$ and $T = (t_1, t_2 \ldots, t_m)$ are of length n and m respectively, with n possibly different from m.

For $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, m$ let $S_i = (s_1, s_2, \ldots, s_i)$ denote the prefix of S of length i, and $T_j = (t_1, t_2, \ldots, t_j)$ denote the prefix of T of length j.

Let $c[i, j]$ denote the distance between S_i and T_j. The initial conditions are

- $c[i, 0] = i$ for all $i = 0, 1, \ldots, n$;
- $c[0, j] = j$ for all $j = 0, 1, \ldots, m$.

The distance between S and T is $c[n, m]$.

Find a recurrence relation for $c[i, j]$. In addition, give a tight runtime analysis of the standard bottom-up implementation for finding the distance using your recurrence.

Solution:

In this problem we have 3 options: inserting a new element in T, removing an element from T, and substituting an element in T. The recurrence relation for $c[i, j]$ follows. Note that in the following recurrence relation, we assume that both i and j are greater than zero, since we have already discussed the case where at least one of them is zero in the problem statement.

$$c[i, j] = \min \left(c[i-1, j] + 1, c[i, j-1] + 1, \begin{cases} c[i-1, j-1] & \text{if } s_i = t_j \\ c[i-1, j-1] + 1 & \text{if } s_i \neq t_j \end{cases} \right)$$

From the recurrence relation, it is clear that we need constant time for computing each $c[i, j]$, therefore the total running time would be $\Theta(nm)$.

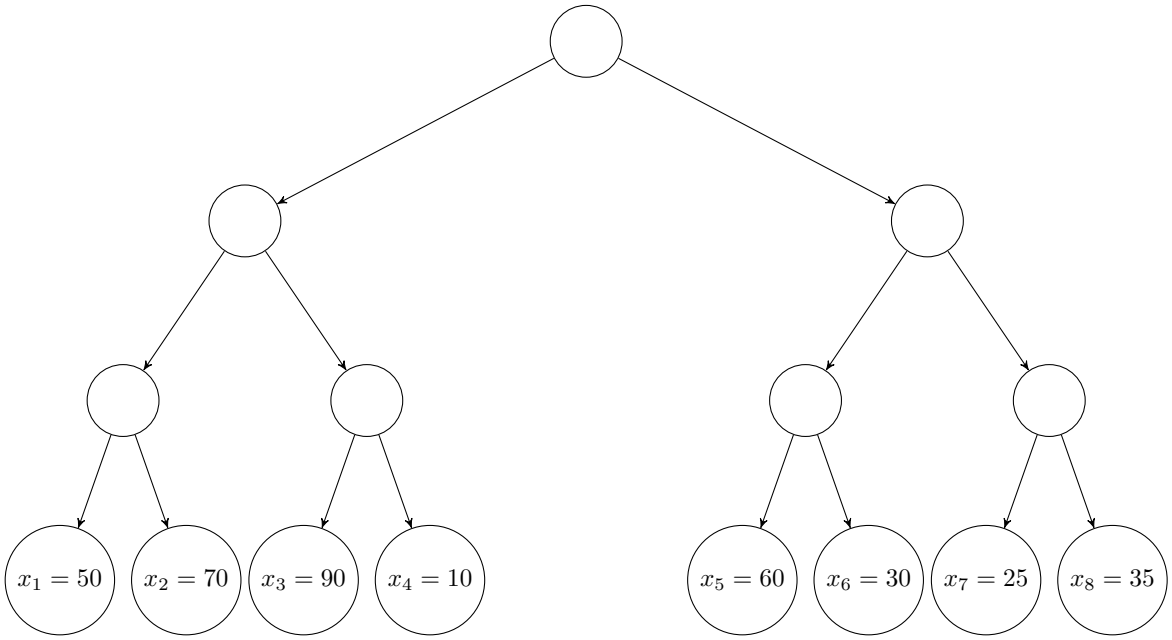


Figure 1. Illustration of the binary tree with $n = 8$ leaves annotated with numbers $x_i, i = 1, \ldots, n$.

4 *(22 pts)* **Fast interval queries.** Suppose that you are given a **complete binary tree** with numbers $1, 2, 3, \ldots, n$ at the leaves (from left to right). In particular, n is a power of 2. You are also given numbers $x_i, i = 1, \ldots, n$ associated with the leaves (see Fig. **??**). Design a data structure that stores extra information at every node of the tree and allows answering *interval queries*: for an input pair $a \le b$ of integers between 1 and n, your data structure should be able to compute $\max_{a \le i \le b} x_i$.

For full credit your solution should take $O(n)$ time to prepare the data structure (the binary tree with extra information stored at the nodes), and every query should be answered in $O(\log n)$ time in the worst case.

Solution: The full binary tree structure suggests that each node should correspond to some interval of indices, with the i-th leaf corresponding to $[i, i]$. Each internal node should correspond to the union of the two intervals that its children correspond to. Thus the root corresponds to the entire range $[1, n]$.

We are going to augment each node with three numbers:[1]

- L: the left end of the corresponding interval of indices,
- R: the right end of the corresponding interval of indices,
- M: the maximum of values over the corresponding interval, that is, $M = \max_{L \le i \le R} x_i$.

We also assume that each node has pointers to its both children. Such an augmented tree is shown in Figure **??**.

The preparation of the data structure will be done in a recursive traversal of the tree (postorder).[2] We go down to the leaves first, where it is easy to initialize these three values; then

[1] We can get rid of the first two, since they can be computed on the fly while traversing the tree later, but the third one is crucial.

[2] This can also be seen as a sort of dynamic programming on the tree.

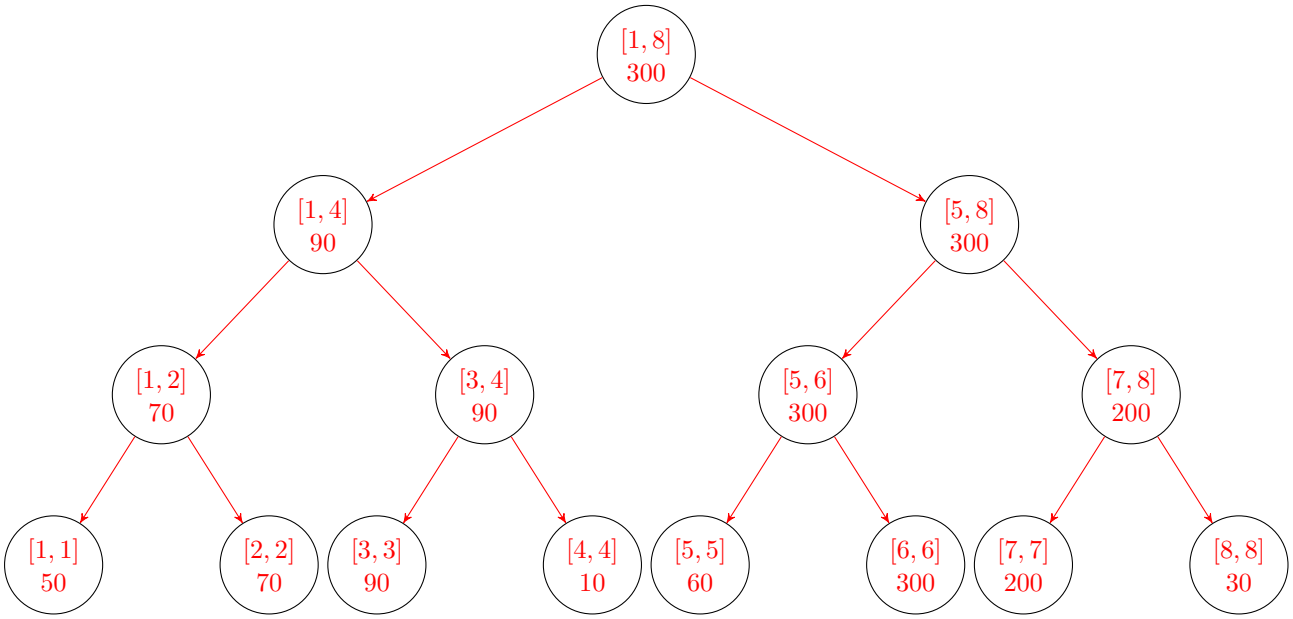


Figure 2. Augmented tree example, for $n = 8$.

we construct the three values for each internal node in $O(1)$ using the values of its children. A pseudocode follows:

```
nextLeafIndex = 1
PREPARE(node):
1. if node is a leaf then
2.     node.L = nextLeafIndex
3.     node.R = nextLeafIndex
4.     node.M = x[nextLeafIndex]
5.     nextLeafIndex = nextLeafIndex + 1
6. else
7.     PREPARE(node.left)
8.     PREPARE(node.right)
9.     node.L = node.left.L
10.    node.R = node.right.R
11.    node.M = max(node.left.M, node.right.M)
PREPARE(root)
```

Now let us see how to answer an interval query $[a, b]$. A naive algorithm would choose the maximum in time $O(b - a)$ per query (which will often be $\Omega(n)$) by looking at all the x-values in the interval $[a, b]$. This can be seen as corresponding to using only the values in the leaves: $[a, a], [a + 1, a + 1], ..., [b, b]$. However, this is wasteful: we have computed, in the internal nodes of our tree, values which should help us accelerate this process. For example, we have $[2, 8] = [2, 2] \cup [3, 4] \cup [5, 8]$, so we only need to take the maximum of three numbers in order to compute the maximum over this interval (see Figure **??**). In fact, we will be able to use only $O(\log n)$ precomputed values thanks to the structure of our tree.

We are going to develop a recursive function GETMAX which, given a node, returns the maximum element in $[a, b]$ which is in the range of that node (i.e. $[\text{node}.L, \text{node}.R]$). In other words,

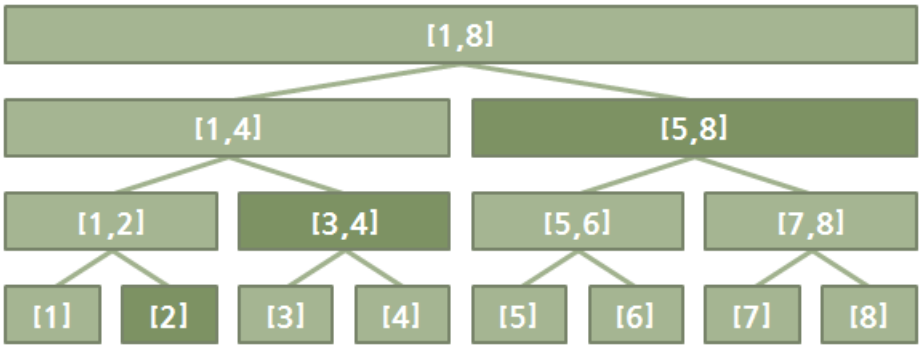


Figure 3. An example of how to use the data structure to answer the query $[2, 8]$ faster.

a call to GetMax(node) should return $\max_{i \in [a,b] \cap [\text{node}.L, \text{node}.R]} x_i$. Clearly, GetMax(root) gives the result. The recursive implementation is natural:

```
GetMax(node):
1. if node.R < a or b < node.L then        // [a,b] ∩ [node.L, node.R] = ∅
2.     return −∞
3. elseif a ≤ node.L and node.R ≤ b then    // [node.L, node.R] ⊆ [a,b]
4.     return node.M
5. else
6.     return max(GetMax(node.left), GetMax(node.right)).
```

If the current interval has an empty intersection with $[a, b]$, then there is no point continuing in that subtree (as the current interval will only shrink as we walk down the tree). On the other hand, if it is entirely included in $[a, b]$, then we should just take its M-value into account without recursing further (this is exactly the source of the speedup). Otherwise, $[a, b]$ intersects non-trivially with the current interval[3], so we continue in both subtrees, and take their maximum.

The correctness of this approach is clear, but we need to argue that the running time is only $O(\log n)$. To see this, first notice that it is enough to bound the number of executions of GetMax(node) where the **else**-branch on line 5 is chosen.[4] But whenever this is the case, we have either $a \in [\text{node}.L, \text{node}.R]$ or $b \in [\text{node}.L, \text{node}.R]$. And this can only be the case for nodes which lie on the path from either a or b up to the root, of which there are $O(\log n)$.

Alternatively, one can propose a bottom-up approach, which starts from the two leaves and traverses the tree going upwards.[5] When visiting a node, one should take care to take into account not only it (if its interval is inside $[a, b]$), but also its immediate children (for example, if coming from the left child, we must check whether to take the right child's interval into account). We skip the proof of correctness; it is a consequence of understanding what intervals $[a, b]$ can be decomposed into, and it follows along the same lines as the running time proof of the top-down approach.[6]

[3] Notice that this implies that we are not yet at a leaf.

[4] Imagine that these executions pay 3 coins, of which they pass 1 coin to each of its recursive calls; if these calls do not go to line 5, then then can pay for their constant running time with the given coin.

[5] For this, we should assume that the nodes have parent pointers; if missing, these can be constructed during the preprocessing traversal.

[6] If you want to find more information online, this problem is called Range Minimum Query (RMQ), and this data structure is sometimes called a Binary Indexed Tree. Usually it is implemented using an array (like a heap), rather than with pointers, for reasons of efficiency and simplicity. Notice that using it, it is also easy to dynamically update the x-values in the array (between the interval queries) in $O(\log n)$.

(additional space for your solutions)

(additional space for your solutions)