ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Midterm Exam, Algorithms 2018-2019

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

Total / 100

1 *(22 pts)* **Basic questions.**

1a *(4 pts)* Answer whether the following statements are **true** or **false**.

HeapSort requires $\Omega(n)$ extra space to sort an array of length n. True or false?
False.

If $T_H(n)$ and $T_B(n)$ denote the worst case runtime of extracting the maximum element from a max heap and a binary search tree on n nodes respectively, then $T_H(n) = \Theta(T_B(n))$. True or false?
False. The depth of a max-heap is $O(\log n)$. Hence, the operation takes $O(\log n)$ time. In a binary search tree, the depth can be $\Omega(n)$(tree was formed by insertion of elements sorted in increasing order.) In particular, the maximum element could be at the last level of the tree. Hence the operation takes time $\Omega(n)$ in the worst case.

Running HeapSort on a sorted (in the correct order) array takes $O(n)$ time. True or false?
False.

BuildMaxHeap constructs a heap from an unsorted array of length n in time $\Theta(n)$. True or false?
True.

1b *(10 pts)* Arrange the following functions in increasing order according to asymptotic growth.

$$n \log_2 \left(n^5 + n^2\right), \quad (\log_2 \log_2 n)^5, \quad 5^{n \log_2 n}, \quad n!, \quad n^2 \log n, \quad 2^n, \quad \sqrt{n^3 + n^4}$$

Solution:

$$(\log_2 \log_2 n)^5, \quad n \log_2 \left(n^5 + n^2\right), \quad \sqrt{n^3 + n^4}, \quad n^2 \log n, \quad 2^n, \quad n!, \quad 5^{n \log_2 n}$$

1c *(8 pts)* In every subproblem below you start with the same fixed binary search tree T. You should answer **yes** or **no** for every subproblem, no explanation is needed. *You should answer* **yes** *for every claim below that is true for every starting tree T, and answer* **no** *if there exists a tree T for which the claim is false.*

- Take an element z with $z.value$ distinct from all those already in the tree. Suppose we perform TreeInsert(T, z) followed immediately by TreeDelete(T, z). Is the new tree identical to T? Yes. Element z is inserted as a leaf in T. Hence, deleting z in the new tree(after insert) gives us T.

- Suppose now that z is an element already contained in T. If we perform TreeDelete(T, z) followed immediately by TreeInsert(T, z) is the new tree identical to T? No. Let T consist of two nodes with values 1 and 2 with root as 1 and it's child as 2. Now, if we perform TreeDelete$(T, 1)$ followed immediately by TreeInsert$(T, 1)$, the new tree is rooted at 2. No. Let T be an empty tree. Then in the first case the final tree is rooted at z_1, whereas in the second case the final tree is rooted at z_2.

- We now have two new elements z_1 and z_2 with values $z_1.value$ and $z_2.value$ distinct from each other and any already in the tree. Perform TREE-INSERT(T, z_1) followed by TREEINSERT(T, z_2). Do we always get the same result if we insert them in the other order? No. Let T be an empty tree. Then in the first case the final tree is rooted at z_1, whereas in the second case the final tree is rooted at z_2.

- Suppose now we are given k new elements $z_1, z_2, \ldots, z_k$ with values distinct from each other's and from any node already in the tree. We perform the following operations:

  TREEINSERT(T, z_1)
  TREEINSERT(T, z_2)
  ...
  TREEINSERT(T, z_k)
  TREEDELETE(T, z_1)
  TREEDELETE(T, z_2)
  ...
  TREEDELETE(T, z_{k-1})

  Is the resulting tree the same as if we had just inserted z_k? Yes. Consider the modified tree T' after the k inserts. Due to the property that each new node is inserted as a leaf, we get that each root to leaf path in T' is some elements of the original tree T followed by some of the newly inserted elements. Hence, if we delete all newly inserted elements but z_k, it doesn't affect the structure of the elements in the original tree T. Also observe that in both the cases, i.e. n inserts followed by $n-1$ deletes and 1 insert, the path from root to z_k is exactly the same if we only consider the part of the path up to the old nodes. Both these observations prove the claim.

2 *(30 pts)* **Recurrences.** Consider the following algorithm UNKNOWN that takes as input an array A and two indices low and $high$ in the array:

```
UNKNOWN(A, low, high)
1. for i = low to high
2.       print A[i]
3. if low = high
4.       return
5. else
6.       p = low + ⌊(high − low)/√2⌋
7.       q = ⌊(low + high)/2⌋
8.       UNKNOWN(A, low, p)
9.       UNKNOWN(A, q + 1, high)
10.      UNKNOWN(A, low, q)
11.      return
```

2a *(10 pts)* Let $T(n)$ be the time it takes to execute UNKNOWN$(A, 1, n)$. **Give the recurrence relation** for $T(n)$. To simplify notation, you may ignore floors and ceilings in your recurrence.

Solution: The for loop starting in line 2 takes $\Theta(n)$ time to run. Let $T(n)$ be the time to run UNKNOWN$(A, low, high)$ for $high - low = n$. Clearly $T(0) = \Theta(1)$. The loop starting in line 1 takes $\Theta(n)$ time to execute. The algorithm further makes recursive calls to UNKNOWN(A, low, p), UNKNOWN$(A, q + 1, high)$ and UNKNOWN(A, low, q). These take time $T(n/\sqrt{2})$, $T(n/2)$ and $T(n/2)$ respectively to run. Therefore, the recurrence relation is

$$T(n) = \Theta(n) + T(n/\sqrt{2}) + 2T(n/2).$$

2b *(20 pts)* **Prove** tight asymptotic bounds on $T(n)$. Specifically, show that $T(n) = \Theta(n^a)$ for some integer $a \geq 0$.

Solution:

Let us guess the value of a. If $T(n) \approx n^a$ then the recursive calls combined would take $\approx n^a/(\sqrt{2})^a + 2n^a/2^a = n^a \cdot \left(2^{-a/2} + 2^{1-a}\right)$ time. We want this to equal n^a, which happens exactly at $a = 2$. Furthermore, in this case the $\Theta(n)$ is dominated by the recursive part, so this seems like a good guess. Let us now prove that the guess is correct.

Let us specify that the $\Theta(n)$ term in the recurrence relation is more that bn and less than cn. We first prove by induction that $T(n) \leq dn^2 - en$ for some d and e:

$$\begin{aligned} T(n) &\leq cn + T(n/\sqrt{2}) + 2T(n/2) \\ &\leq cn + d(n/\sqrt{2})^2 - e(n/\sqrt{2}) + 2d(n/2)^2 - 2e(n/2) \\ &= cn + dn^2/2 - en/\sqrt{2} + dn^2/2 - en \\ &\leq dn^2 - en, \end{aligned}$$

when $e/\sqrt{2} \geq c$.

Proof by induction that $T(n) \geq dn^2$ for some d:

$$\begin{aligned} T(n) &\geq bn + T(b/\sqrt{2}) + 2T(n/2) \\ &\geq bn + d(n/\sqrt{2})^2 + 2d(n/2)^2 \\ &= bn + dn^2/2 + dn^2/2 \\ &\geq dn^2, \end{aligned}$$

since $b \geq 0$.

Therefore $T(n) = \Theta(n^2)$.

3 *(28 pts)* **Rod cutting revisited.** You are given a rod of length n with m weak points on it at distances $a_1, a_2, \ldots, a_m$ from one end. $a_1, a_2, \ldots, a_m$ are integers such that $0 < a_1 < a_2 < \ldots < a_m < n$. You have to cut the rod at all weak points in some order. Each time you cut a rod at some weak point it falls apart into two pieces and two smaller rods are created, so after performing m cuts you will have $m + 1$ smaller rods.

The cost of cutting a rod is equal to its length, and you want to minimize the overall cost of cutting the rod into $m+1$ pieces. In this problem your task is to design and analyze an algorithm that, given the length n of the rod and the locations of the weak points, returns the cost of the optimal solution.

> **Input:** The length n of rod (an integer), the number m of weak points, and the locations of the weak points $0 < a_1 < a_2 < \ldots < a_m < n$ (the locations $a_i, i = 1, \ldots, m$ are integers).
>
> **Output:** The smallest possible cost of cutting the rod into $m+1$ pieces at the weak points.

For example, if $n = 7$ and there are $m = 2$ weak points on the rod, with locations $a_1 = 4$ and $a_2 = 6$ respectively (see Fig. 1), the optimal solution is as follows. We first break the rod at the first weak point $a_1 = 4$, getting two rods, of length 4 and 3 respectively. We then break the rod of length 3 at the second weak point into two rods, with lengths 2 and 1 respectively. The total cost is $7 + 3 = 10$.

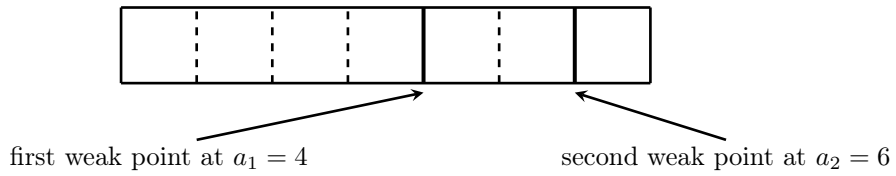

Figure 1. A rod of length $n = 7$ with $m = 2$ weak points, at locations $a_1 = 4$ and $a_2 = 6$ respectively.

Design an efficient dynamic programming algorithm for this problem and **analyze its runtime.**

Solution: For simplicity suppose that $a_0 = 0$, $a_{m+1} = N$, and for any $0 \le s \le t \le m + 1$, let $c[s][t]$ denote the smallest possible cost of cutting the rod between a_s and a_t into $t - s$ pieces. Clearly the final solution is $c[0][m + 1]$, and if $t \le s + 1$, then $c[s][t] = 0$.

If $t > s + 1$, then we consider all the subproblems of cutting the rod of endpoints a_s and a_t, into all wak points between a_s and a_t, i.e., all a_k's such that $s+1 \le k \le t-1$. The cost of $c[s][t]$ is given by $a_t - a_s$, added to the cost of the minimum subproblem. More formally, the algorithm proceeds using the following formula

$$c[s][t] = a_t - a_s + \min_{s+1 \le k \le t-1} \{c[s][k] + c[k][t]\}$$

We implement this recursive formulation using top-down with memoization approach.

The total space of the algorithm is $\Theta(m^2)$, since we need to store $c[s][t]$ for all $0 \le s \le t \le m+1$. Moreover, the runtime of the algorithm is $\Theta(m^3)$ since for any $1 \le s \le t \le m+1$, as per

Algorithm 1 COST(s, t)

```
1: if c[s][t] ≠ ∞ then
2:     return c[s][t]
3: else if t ≤ s + 1 then
4:     c[s][t] = 0
5: else
6:     for k = s + 1 to t − 1 do
7:         c[s][t] = min(c[s][t], COST(s,k) + COST(k,t) + a_t − a_s)
   return c[s][t]
```

line 6 of the algorithm, we run a for-loop over all k between s and t. Thus, the total runtime of the algorithm is $\Theta(m^3)$.

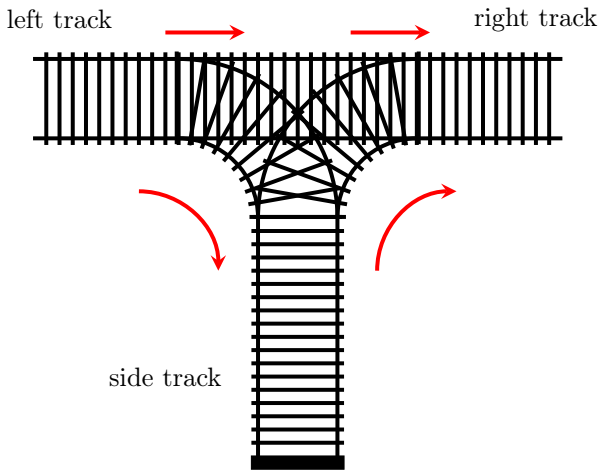

Figure 2. Lausanne Gare

4 *(20 pts)* **Train sequences at Lausanne Gare.** You and your friend are watching the trains at Lausanne Gare. The tracks at Lausanne Gare form a 'T' shape as shown in Fig. 2. You sit at the left end of the tracks and see that n trains labeled $1, 2, \ldots, n$ are lined up in increasing order. Then the trains start moving to the right one by one, first train 1, then 2 and so on. Any train may either turn onto the side track (the side track can hold an infinite number of trains) or keep moving right and leave the station. At any point in time a train on the side track can decide to leave the station through the right track if it's not blocked in. However, the trains can never go back to the left track.

Your friend sits at the right end of the tracks, observes the trains leaving one by one and writes down their numbers in the order that they leave. Your task is to develop an efficient algorithm that decides, given your friend's notes, if that order of trains is possible or if your friend had made a mistake.

Input:	A permutation $a_1, a_2, \ldots, a_n$ of integers between 1 and n.
Output:	**YES** if there exists a way for trains $1, 2, \ldots, n$ to leave Lausanne Gare in this order, and **NO** otherwise.

For example, suppose there are $n = 3$ trains. Then order $2, 1, 3$ (i.e., $a_1 = 2, a_2 = 1, a_3 = 3$) is possible: the first trains turns onto the side track, the second train leaves, the first train exits the side track and leaves and finally the third train leaves as well. However, the order $3, 1, 2$ is not possible.

Design an efficient algorithm for this problem and **analyze its runtime**.

Solution:

Let A be the list of trains as observed by your friend. Our plan is to store the positions of all trains given that A has been correct so far. If we get to a contradiction we return FALSE. More specifically we need to store the first train (that is the train labeled by the lowest number) still on the left side of the tracks (variable j in the pseudocode below) and we need to store the the content of the spur track. Notice that because trains block each other in on the spur track, only

the most recent train to have arrived can leave at any time: the spur track behaves exactly like a stack. Therefore, we store the contents of he spur track in a stack (variable s in the pseudocode below).

For simplicity, we may pretend that all trains must enter the spur tracks before leaving the station. So if the next train in the notes is $A[i]$ we have two options: If the first train in the stack is $A[i]$, we pop it out. If it isn't, we have only one choice, to put the next train from the left of the tracks (j) into the stack. If we can't do that because we're out if trains, we've reached a contradiction and the algorithm can return false.

Below is the pseudocode for this algorithm.

```
1: i ← 1
2: j ← 1
3: Initialize empty stack S
4: while j ≤ N do
5:     if S.head == A[i] then
6:         S.pop()
7:         i ← i + 1
8:     else
9:         S.push(j)
10:        j ← j + 1
11: return S.empty()
```