ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Midterm Exam, Algorithms 2015-2016

- 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 so 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 so that a fellow student can easily implement the algorithm following the description.
- **Do not touch until the start of the exam.**

Good luck!

Name: ______________________ **N° Sciper:** ______________

Problem 1 / 16 points	Problem 2 / 15 points	Problem 3 / 23 points	Problem 4 / 23 points	Problem 5 / 23 points

Total / 100

1 *(16 pts)* Recurrences and Stacks.

1a *(8 pts)* Give tight asymptotic bounds for the following recurrences (assuming that $T(1) = \Theta(1)$). You need not justify your answers.

(i) $T(n) = 2T(n/4) + \Theta(\sqrt{n})$

(ii) $T(n) = 10T(n/3) + \Theta(n)$

(iii) $T(n) = T(n/2) + T(n/3) + T(n/3) + \Theta(n^2)$

Solution:

(i) $T(n) = 2T(n/4) + \Theta(\sqrt{n}) = \Theta(\sqrt{n}\log n)$

(ii) $T(n) = 10T(n/3) + \Theta(n) = \Theta(n^{\log_3(10)})$

(iii) $T(n) = T(n/2) + T(n/3) + T(n/3) + \Theta(n^2) = \Theta(n^2)$

1b *(8 pts)* Consider the following procedure UNKNOWN that takes as input an array $A[1 \ldots n]$ consisting of n letters and returns true or false.

```
UNKNOWN($A, n$)
1. Let $S$ be an empty stack
4. for $i = 1$ to $\lfloor n/2 \rfloor$
5.      PUSH($S, A[i]$)
6. for $j = \lceil n/2 \rceil + 1$ to $n$
7.      if $A[j] \neq$ POP($S$)
8.           return false
9. return true
```

What does UNKNOWN(A, n) return on input $A =$ | A | B | B | A | and $n = 4$?

Solution: true

What does UNKNOWN(A, n) return on input $A =$ | O | L | A | and $n = 3$?

Solution: false

In general, give a succinct characterization of the inputs for which the procedure returns true.

Solution: The procedure returns true if and only if the word/phrase represented by A is a palindrome, i.e., it reads the same backward and forward.

2 *(15 pts)* **Divide and Conquer.** The increasing popularity of the MERGE-SORT algorithm is largely due to it being parallelizable. This is a significant advantage when dealing with large data sets. Here, we will analyze a new variant of merge-sort, called MERGE-SORT-DELUX, that could potentially be even better for parallelization. Indeed, instead of partitioning the array recursively into two subproblems, we will partition the array into $\sqrt{n}$ subproblems (all of which could potentially be sorted recursively on different computers). The pseudo-code is as follows.

```
MERGE-SORT-DELUX($A, p, r$)
1. Let $n = r - p + 1$
2. if $n > 1$
3.     $k = \lceil \sqrt{n} \rceil$
4.     for $i = 1$ to $k$
5.         MERGE-SORT-DELUX($A, p + \lfloor \frac{n}{k} \cdot (i-1) \rfloor, p + \lfloor \frac{n}{k} \cdot i \rfloor - 1$)
6. Merge the $k$ sorted arrays

        $A\left[p \ldots (p + \lfloor \frac{n}{k} \rfloor - 1)\right]$
        $A\left[(p + \lfloor \frac{n}{k} \rfloor) \ldots (p + \lfloor \frac{n}{k} \cdot 2 \rfloor - 1)\right]$
            $\vdots$
        $A\left[(p + \lfloor \frac{n}{k} \cdot (k-1) \rfloor) \ldots r\right]$

   into one sorted array $A[p \ldots r]$.
```

Recall from the exercises that k sorted arrays containing n elements in total can be merged in time $\Theta(n \log k)$ and thus Step 6 takes time $\Theta(n \log k)$. We also assume that $\sqrt{n}$ can be calculated in constant time.

2a *(6 pts)* Let $T(n)$ be the time it takes to execute MERGE-SORT-DELUX(A, p, r) on a single computer, where $n = r - p + 1$ is the number of elements in the array. **Give the recurrence relation** for $T(n)$. To simplify notation, you may assume that $\sqrt{n}$ and n/k are integers.

Solution: The divide and combine step is dominated by Step 6 which takes time $\Theta(n \log k)$ where $k = \sqrt{n} = n^{1/2}$ so this is $\Theta(n \log n)$. In the conquer step, $k = \sqrt{n}$ subproblems of size $n/\sqrt{n}$ are solved recursively.

This gives us the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \text{ (or a constant)} \\ \sqrt{n} \cdot T(n/\sqrt{n}) + \Theta(n \log n) & \text{otherwise} \end{cases}$$

2b *(9 pts)* **Prove** that $T(n) = O(n \log n)$ using the substitution method.

Solution: We prove $T(n) = O(n \log n)$ by showing that there exists a constant $d > 0$ such that $T(n) \leq dn \log n$ for all $n \geq 1$. As always, the base case is trivial by selecting d large enough.

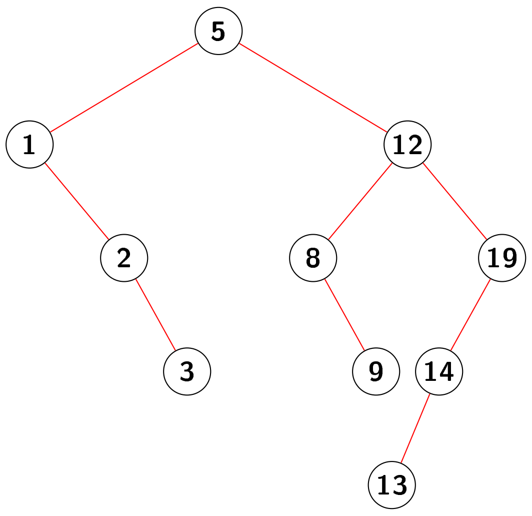
Now consider the inductive step (with the induction hypothesis that $T(n') \leq dn' \log n'$ for all $n' < n$). For some constant $c > 0$ we have that

$$\begin{aligned}
T(n) &= \sqrt{n}T(\sqrt{n}) + c \cdot n \log n \\
&\leq \sqrt{n} d \sqrt{n} \log(\sqrt{n}) + c \cdot n \log n \\
&= dn \log(n^{1/2}) + c \cdot n \log n \\
&= \frac{d}{2} n \log n + cn \log n \\
&\leq dn \log n \qquad \text{(by selecting } d \geq 2c\text{)}.
\end{aligned}$$

3 *(23 pts)* **Binary Search Trees.**

3a *(7 pts)* We consider the task of *reconstructing* a binary search tree from the output of a postorder walk. **Illustrate/draw** the binary search tree T for which the output of POSTORDER-TREE-WALK($T.root$) is $3, 2, 1, 9, 8, 13, 14, 19, 12, 5$.

Solution:



3b *(16 pts)* **Design** and **analyze** an algorithm for the following problem:

Input: An array $A[1 \ldots n]$ consisting of n *different* integers.

Output: A binary search tree T of height $\lfloor \log_2(n) \rfloor$ with the integers in A as keys (the tree should contain exactly one key for each element of A).

Your algorithm should run in time $O(n \log n)$.

Solution: In order to construct our tree, we first sort A in non-decreasing order using Mergesort. Then, we will apply a recursive procedure REC to the sorted array A; REC receives as input an array, and outputs the root node of a balanced binary search tree whose keys are the elements of A. REC works as follows: given array B of length k as input, REC constructs a root node r whose key is $B[\lfloor k/2 \rfloor + 1]$; then, the left child of r is set to be the root of the tree constructed by REC($B[1, \ldots \lfloor k/2 \rfloor]$), and the right child of r is set to be the tree constructed by REC($B[\lfloor k/2 \rfloor + 2, k]$).

Let us now analyze our algorithm; first, we analyze its running time. The sorting takes time $O(n \log n)$, and the running time of constructing the tree is expressed by the following recurrence:

$$T(n) = T(n/2) + O(1)$$

which means (using the Master Theorem) that $T(n) = O(n)$. Hence, the total running time is $O(n \log n)$.

Now, let us see why the algorithm works: first of all, the fact that the output is a binary search tree can be easily proved by observing that (due to the fact that A is sorted) the left child of a node of a node r has always smaller key than r, and the right child of a node r has always larger key than r. To see that the resulting tree has height $\lfloor \log n \rfloor$, it suffices to observe that every level i of the tree has exactly 2^{i-1} nodes, except possibly for the last one.

Here is the complete description of the algorithm:

```
function MAIN($B, x, y$)
    Sort $B$ using Merge-sort
    $l \leftarrow x - y + 1$
    Create node $r$ with $r$.key $= B[x + \lfloor l/2 \rfloor]$
    if $l \geq 2$ then
        $r$.left $\leftarrow$ REC($B, x, x + \lfloor l/2 \rfloor - 1$)
    end if
    if $l \geq 3$ then
        $r$.right $\leftarrow$ REC($B, x + \lfloor l/2 \rfloor + 1, y$)
    end if
    return $r$
end function
function REC($B, x, y$)
    $l \leftarrow x - y + 1$
    Create node $r$ with $r$.key $= B[x + \lfloor l/2 \rfloor]$
    if $l \geq 2$ then
        $r$.left $\leftarrow$ REC($B, x, x + \lfloor l/2 \rfloor - 1$)
    end if
    if $l \geq 3$ then
        $r$.right $\leftarrow$ REC($B, x + \lfloor l/2 \rfloor + 1, y$)
    end if
    return $r$
end function
```

To solve the problem, we run MAIN($A, 1, n$).

4 *(23 pts)* **Dynamic Programming.** In this problem you are going to help Mourinho get the Chelsea football club back on track. In particular, you should design an algorithm for buying the cheapest set of players so that they form a "good team": their sum of skills should be at least a threshold T. In our abstract model, we assume that a player's skill can be characterized by a single integer. The formal definition of our problem is as follows:

INPUT: A set $\{1, 2, \dots, n\}$ of n players where each player i is characterized by the following data:

- $s_i \geq 0$ — an integer describing player i's skill,
- $p_i \geq 0$ — an integer describing the price of buying player i.

In addition, we are given a non-negative integer T, which is the required sum of skills of the new players.

OUTPUT: The smallest cost C such that there exists a subset $P \subseteq \{1, 2, \dots, n\}$ of players satisfying

$$\sum_{i \in P} p_i = C \qquad \text{and} \qquad \sum_{i \in P} s_i \geq T.$$

If no subset P with the required sum of skills exists, the algorithm should output ∞.

4a *(13 pts)* Let $c[i, t]$ be the minimum cost of a solution to the instance consisting only of the first i players $\{1, 2, \dots, i\}$ and with total skill requirement t. In other words, $c[i, t]$ is the smallest cost such that there exists a subset $P \subseteq \{1, 2, \dots, i\}$ satisfying

$$\sum_{i \in P} p_i = c[i, t] \qquad \text{and} \qquad \sum_{i \in P} s_i \geq t.$$

(Or, simply ∞ if no such subset P exists.)

Complete the recurrence relation for $c[i, t]$ that can be used for dynamic programming. Also motivate your answers by explaining your reasoning.

Solution:

$$c[i,t] = \begin{cases} \infty & \text{if } i = 0 \text{ and } t > 0 \\ 0 & \text{if } i = 0 \text{ and } t = 0 \\ \min \left\{ \begin{array}{l} c[i-1, t], \\ p_i + c[i-1, (t - s_i)^+] \end{array} \right\} & \text{if } i > 0 \end{cases}$$

Here, we used the notation $(a)^+$ to denote $\max\{a, 0\}$. The motivation of our solution is that the interesting case (when $i > 0$) is the minimum of two choices: either we buy player i or not. If we don't buy him then we need to meet the requirement t with players $\{1, \dots, i-1\}$ in an optimal way which costs $c[i-1, t]$. Otherwise, if we buy player i then we need to pay p_i and meet the requirement $(t - s_i)^+$ with players $\{1, \dots, i-1\}$ in an optimal way which costs $c[i-1, (t-s_i)^+]$. For the base cases, assume we have no player. By choosing none of them, we can get a team of cost zero with total sum of skills equal to zero, so the cost is zero for the case when $i = 0, t = 0$. On the other hand, it is not possible to have a team with positive skills, therefore, the cost is ∞ in the case when $i = 0, t > 0$.

4b *(10 pts)* Consider the following instance of our problem:

i	1	2	3	4	5	6
s_i	2	1	3	4	4	2
p_i	3	1	7	7	6	2

$n = 6$ and $T = 5$.

Use the recurrence relation to return the optimal solution by **filling in the table** of $c[i, t]$ values below (in a bottom-up dynamic programming fashion). Also, in general, **what is the running time** (in Θ-notation) of a bottom-up dynamic programming implementation as a function of n and T?

Solution:

Table:

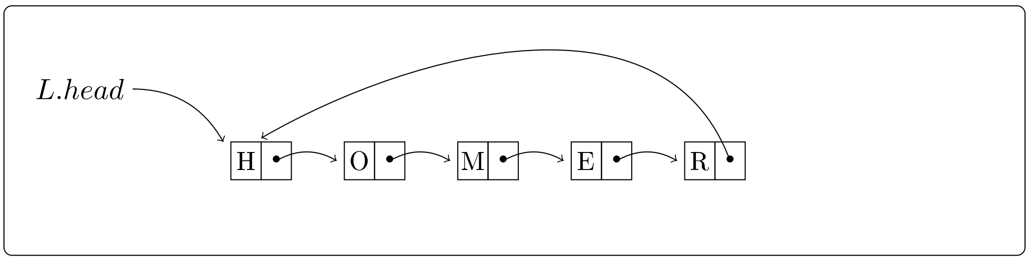
$i \backslash T$	0	1	2	3	4	5
0	0	∞	∞	∞	∞	∞
1	0	3	3	∞	∞	∞
2	0	1	3	4	∞	∞
3	0	1	3	4	8	10
4	0	1	3	4	7	8
5	0	1	3	4	6	7
6	0	1	2	3	5	6

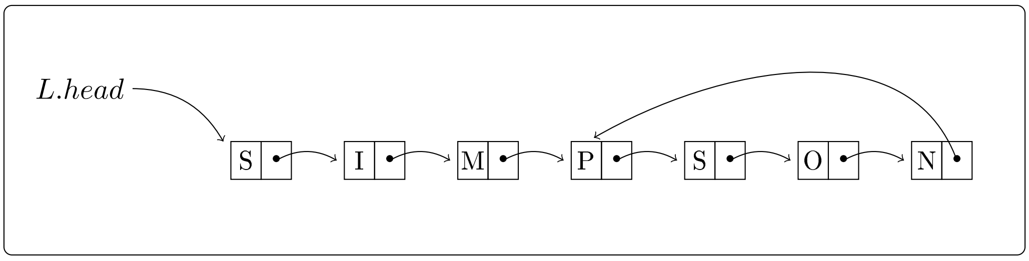
The asymptotic running time (as a function of n and T) of solving the problem in a bottom-up fashion is

$$\Theta\left(nT\right)$$

for the recurrence given in 4a. Other solutions/recurrences may have a different running time.

5 *(23 pts)* **Homer Simpson's crazy-lists.** Homer Simpson is not exactly known as the Einstein of Springfield. In spite of this and unfortunately for us, he has decided to design a linked-list data structure, which we call crazy-lists. A crazy-list is like a single-linked list with the following important exception: the last element's pointer points to a previous element in the list instead of being nil. Two examples of crazy-lists are as follows:





Design and **analyze** an algorithm that takes as input a crazy-list (i.e., a pointer $L.head$) and outputs the number n of elements in that list. The algorithm is *not* allowed to modify the input. In addition, your algorithm should **run in time** $O(n)$ and **use a constant amount of *extra* space** (not counting the memory for storing the list).

If you do not solve the whole problem, you are encouraged to write down your best (partial) solution.

Hints: Use the "tortoise/turtle (slow) and hare (fast)" technique to detect the cycle. Then calculate the length of the cycle. After that calculate the length of the path up to the cycle.

Solution:

1. Let $slow$ and $fast$ be two pointers, initially initialised to $L.head$ and $L.head.next$ respectively, such that $slow$ makes one move at a time (i.e., $slow = slow.next$) and $fast$ makes two moves (i.e., $fast = fast.next$).
2. We detect the cycle the first time $slow$ and $fast$ meet, but we still need to calculate the length of this cycle. To do this we create a counter $cLen$ that we set to 1.
3. We walk around the cycle again with both pointers while incrementing $cLen$ at each time. When they meet again, $cLen$ will be exactly the length of the cycle.
4. We now create a counter $LenToC = 1$, and 2 new pointers, call them $p1$ and $p2$, such that $p1 = L.head$ and $p2$ is $cLen$ steps away from $L.head$. We move each pointer 1 step at a time while incrementing $LenToC$, and when they meet is easy to see that $LenToC$ will be exactly the length from head up until the start of the cycle.
5. The total number of elements in the list is now simply $n = cLen + LenToC$.

Space Requirement: O(1), as we only create 4 pointers (we can infact reuse *slow* and *fast* instead of creating $p1$ and $p2$) and 2 counters.

Time Requirement: Let n be the total number of elements in the list, and c the length of the cycle. It is easy to see that step 2 takes c moves, and step 3 takes n moves. It remains to see how much does step 1 take.

We know that after $n-c$ moves, both *slow* and *fast* are *inside* the cycle, which means that they are at distance at most c from each others. Note that at each step, since t increases by 1, and h increases by 2, the hare will be one step closer to the turtle, and hence they will collide after at most c moves after entering the cycle.

Hence 1 takes at most $n - c + c = n$ steps. Therefore the overall running time is $O(n)$.