ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Midterm Exam, Algorithms 2013-2014

- 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 it. For example, a description of an algorithm should be so that a fellow student can easily implement the algorithm following the description.

Good luck!

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

Exercise 1 / 15 points	Exercise 2 / 25 points	Exercise 3 / 15 points	Exercise 4 / 20 points	Exercise 5 / 25 points

Total / 100

1 *(15pts)* **Asymptotics.**

1a *(5pts)* Arrange the following functions in increasing order according to asymptotic growth.

$$4 \cdot 16^{n/2}, \quad 10 \log^{3/2} n, \quad n^5, \sqrt{n}, \quad 10 \log n, \quad 100 \cdot 2^n, \quad n^4 \log^2 n, \quad 2 \cdot 3^n$$

Solution: $10 \log n, \quad 10 \log^{3/2} n, \quad \sqrt{n}, \quad n^4 \log^2 n, \quad n^5, \quad 100 \cdot 2^n, \quad 2 \cdot 3^n, \quad 4 \cdot 16^{n/2}$

1b *(10 pts)* Give tight asymptotic bounds for the following recurrences (assuming that $T(1) = \Theta(1)$):

(i) $T(n) = 3T(n/3) + \Theta(1)$
(ii) $T(n) = 3T(n/3) + \Theta(n)$
(iii) $T(n) = 3T(n/3) + \Theta(n^2)$
(iv) $T(n) = T(n/5) + 2T(2n/5) + \Theta(n)$
(v) $T(n) = 25T(n/5) + \Theta(n^2)$

Solution:

(i) $\Theta(n)$ (Master Theorem)

(ii) $\Theta(n \log n)$ (Master Theorem)

(ii) $\Theta(n^2)$ (Master Theorem)

(iv) $\Theta(n \log n)$ (Substitution method)

(v) $\Theta(n^2 \log n)$ (Master Theorem)

2 *(25pts)* **Divide and Conquer.** Consider the following procedure UNKNOWN that takes as input an array $A[\ell \dots r]$ of $r - \ell + 1$ numbers with the left-index ℓ and the right-index r:

```
UNKNOWN($A, \ell, r$)
1. if $\ell > r$
2.      return $-\infty$
3. else if $\ell = r$
4.      return $A[\ell]$
3. else
4.      $q \leftarrow \ell + \lfloor \frac{r-\ell}{3} \rfloor$
5.      $ansL \leftarrow$ UNKNOWN($A, \ell, q$)
6.      $ansR \leftarrow$ UNKNOWN($A, q+1, r$)
7.      if $ansL > ansR$
8.           return $ansL$
9.      else
10.          return $ansR$
```

2a *(5pts)* Let $A[1 \dots 8] = \boxed{6 \mid 4 \mid 2 \mid 9 \mid 2 \mid 8 \mid 7 \mid 5}$. What does a call to UNKNOWN(A, 1, 8) return?

Solution: 9 (The maximum value in A.)

2b *(8pts)* Let $T(n)$ be the time it takes to invoke UNKNOWN(A, ℓ, r) where $n = r - \ell + 1$ is the number of elements in the array. Give the recurrence relation of $T(n)$.

Solution: $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \Theta(1)$, $T(1) = \Theta(1)$.

2c *(12pts)* *Prove* tight asymptotic bounds of $T(n)$ using the substitution method, i.e., show that $T(n) = \Theta(f(n))$ for some function f.

Solution: *(on the next page)*

Solution of 2c: The function $T(n)$ is:

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + 1.$$

We guess that the function $T(n) = \Theta(n)$ and we prove this using the substitution method. First, we prove an upper bound on $T(n)$. We assume the following inductive hypothesis for all $m < n$:

$$T(m) \leq c_1 m - 1$$

for some constant $c_1 > 0$. We will prove that the hypothesis holds for n: (We can assume that n is divisible by 3.)

$$\begin{aligned} T(n) &= T(\frac{n}{3}) + T(2\frac{n}{3}) + 1 \\ &\leq c_1\frac{n}{3} - 1 + c_1\frac{2n}{3} - 1 + 1 \\ &= c_1 n - 1. \end{aligned}$$

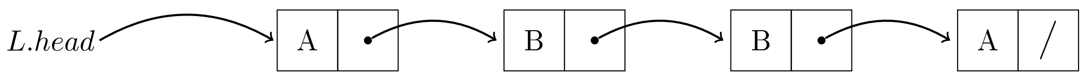
Thus, we have $T(n) = O(n)$.

For the lower bound, we assume that $T(m) \geq c_2 m$ for all $m < n$ and for some constant $c_2 > 0$. Then, we have:

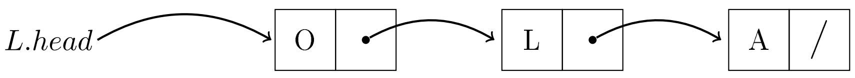
$$\begin{aligned} T(n) &= T(\frac{n}{3}) + T(\frac{2n}{3}) + 1 \\ &\geq c_2\frac{n}{3} + c_2\frac{2n}{3} \\ &= c_2 n. \end{aligned}$$

Thus, we have $T(n) = \Omega(n)$. We can conclude that $T(n) = \Theta(n)$.

3 *(15pts)* **Palindrome.** A word is a palindrome if its reverse is equal to itself. For example, ABBA is a palindrome whereas OLA is not. One way of representing a word in a computer is to have a single-linked list where we have a list-element for each letter. For example, ABBA is represented by the single-linked list



and OLA is represented by the single-linked list



Design and **analyze** an algorithm that, given a pointer to the head of a single-linked list which represents a word, outputs YES if the word is a palindrome and NO otherwise.

For full score your algorithm should run in **linear time** and **should not use any other data structures than single-linked lists**, i.e., no arrays, stacks, queues, etc..

Solution: Let L be the linked list representing our word; the idea of the algorithm is to create a new linked list Q which contains the word of L reversed, and then compare the two words. We create the linked list Q by "walking through" the list L and for each element in L, inserting this element in the *head* of the list Q. At the end of this procedure, the list Q will be equal to the reverse of L.

The following pseudocode describes this procedure. Note that the function INSERT(Q,y) inserts the element y at the head of the linked list Q.

```
PALINDROME($L$)
1. $x \leftarrow L$.head
2. $Q \leftarrow$ new linked list
3. while $x \neq$NIL
4.       $y \leftarrow$ new list element
5.       $y$.key$\leftarrow x$.key
6.       INSERT$(Q, y)$
7.       $x \leftarrow x$.next
8. end-while
9. $x \leftarrow L$.head
10. $y \leftarrow Q$.head
11. while $x \neq$NIL
12.       if $x$.key$\neq y$.key
13.             return NO
14.       $x \leftarrow x$.next
15.       $y \leftarrow y$.next
16. end-while
17. return YES
```

Both while loops will be executed n times, where n is the length of the word. Since each loop consists of a constant number of operations, each of constant cost (recall that inserting into the head of a list takes constant time), the total running time will be $\Theta(n)$.

4 *(20pts)* **Quadtrees.** A quadtree is a search tree data structure in which each internal node has exactly four children. Quadtrees are usually used to partition the two-dimensional plane by recursively subdividing it into four quadrants or regions. In this exercise, we shall use quadtrees to represent cities according to their geographical position.

Figure 1 shows an example of 4 cities in western Switzerland. It also shows the 4 quadrants induced by the city Fribourg: the first, named NW (for *north-west*), contains the city Neuchâtel; the two following (NE for *north-east* and SE for *south-east*) are empty; the last (SW for *south-west*) contains the two remaining cities, Yverdon and Lausanne.

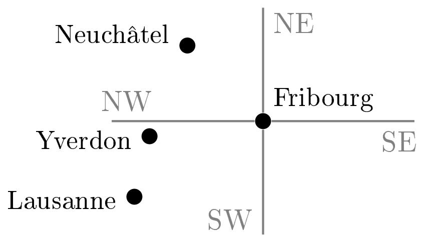


Figure 1. Geographical location of 4 cities in Switzerland

Each node v in the quadtree will store the name ($v.name$) and the coordinates ($v.x$ and $v.y$) of the cities. In addition, the node v will contain a pointer $v.p$ to its parent (or NIL if its the root) and pointers $v.NW, v.NE, v.SE, v.SW$ to its four children. Similar to binary search trees the key properties that make quadtrees useful are the following:

- if u is in the quadtree rooted by $v.NW$ then $u.x < v.x$ and $u.y \geq v.y$;
- if u is in the quadtree rooted by $v.NE$ then $u.x \geq v.x$ and $u.y > v.y$;
- if u is in the quadtree rooted by $v.SE$ then $u.x > v.x$ and $u.y \leq v.y$;
- if u is in the quadtree rooted by $v.SW$ then $u.x \leq v.x$ and $u.y < v.y$.

Figure 2 shows a possible quadtree representation of the cities in Figure 1. The root is the node corresponding to Fribourg that has x and y coordinates equal to 578461 and 183802, respectively.

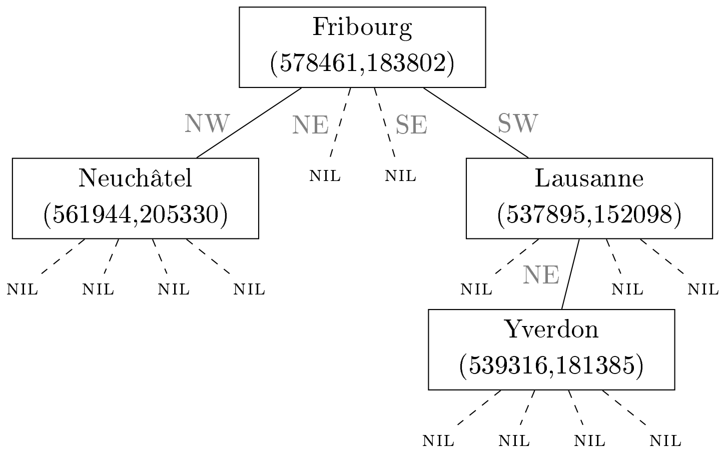


Figure 2. A possible quadtree representation of the cities in Figure 1

Design (give the pseudocode) and **analyze** the running times of the following procedures for the quadtree data structure:

(10pts) SEARCH(Q.root, x, y) — prints the name of the city located at (x, y) if such a city exists in the quadtree rooted at $Q.root$.

(10pts) PRINTSOUTHMOST(Q.root) — prints the name of the south most city (the city with the smallest y coordinate) in the quadtree rooted at $Q.root$.

Your runtime analyses should be tight for the typical case when the height of the quadtree is logarithmic in the number of cities that it contains. In addition, for full score, the running times of your implementations should not be unnecessary large.

Solution: Let us begin with the SEARCH operation; this operation will be very similar to searching in a binary tree, with the difference that now we have to operate along two dimensions. Specifically, we will design a recursive procedure, which given an input node and two coordinates x, y will compare these coordinates with the input node's coordinates, in order to decide in which direction the node we are searching for is; after this, we will apply SEARCH in the input node's child which is in the correct direction, and when we finally reach our target node, we will print its name.

```
SEARCH($Q$.root$, x, y$)
1. $u \leftarrow Q$.root
2. **if** $u$ =NIL
3.     **return** does not exist
4. **if** $u.x = x$ and $u.y = y$
5.     **return** $u$
6. **if** $u.x < x$ and $u.y \geq y$
7.     SEARCH($u$.NW,$x, y$)
8. **if** $u.x \geq x$ and $u.y > y$
9.     SEARCH($u$,NE,$x, y$)
10. **if** $u.x > x$ and $u.y \leq y$
11.     SEARCH($u$.SE, $x, y$)
12. **if** $u.x \leq x$ and $u.y < y$
13.     SEARCH($u$.SW,$x, y$)
```

Each execution of SEARCH executes a constant number of operations branches out to at most one other execution of SEARCH, and the maximum recursion depth is h, where h is the height of the tree; hence, the total running time is $O(h)$.

Continuation of solution to 4:

Let us continue with the PRINTSOUTHMOST operation: here, we will again design a recursive procedure RETURNSOUTHMOST, which returns the y-coordinate and the name of the subtree rooted at the input node. It works as follows: first, it applies RETURNSOUTHMOST to both the southwest and the southeast subtrees of the input node, and it stores the y-coordinate of the most southern node in the southwest(southeast) subtree in variable $w(e)$. Then, it compares w and e in order to determine in which of the two subtrees the most southern node lies, and then returns its y coordinate and name($wname$ for the southwest subtree and $ename$ fot the southeast tree).

```
RETURNSOUTHMOST($Q$.root)
1. $u \leftarrow Q$.root
2. $w, e \leftarrow \infty$
3. if $u$.SW$\neq$NIL
4.       $(w, wname) \leftarrow$RETURNSOUTHMOST($u$.SW)
5. if $u$.SE$\neq$NIL
6.       $(e, ename) \leftarrow$RETURNSOUTHMOST($u$.SE)
7. if $w \leq e, u.y$
8.       return $(w, wname)$
9. if $e \leq w, u.y$
10.      return $(e, ename)$
11. if $u.y \leq e, w$
12.      return $(u.y, u.name)$
```

Finally, in order to actually print the name of the most southern node, all we have to do is apply RETURNSOUTHMOST to the tree rooted at Q.root, and print out its name.

```
PRINTSOUTHMOST($Q$.root)
1. $(s, sname) \leftarrow$RETURNSOUTHMOST($Q$.root)
2. print $sname$
```

Let us analyze its running time: this recursive procedure will visit each node at most once. Therefore, its running time is $O(n)$, where n is the number of nodes in the tree.

This is tight as we do two recursive calls in each step (RETURNSOUTHMOST($u.SW$) and RETURNSOUTHMOST($u.SE$)) and all remaining cities may be located in these subtrees. That is we have in the worst case a running time proportional to $T(n) = 2T(n/2) + \Theta(1)$ which is $\Omega(n)$ even if the height of the tree is logarithmic.

5 *(25 pts)* **Restaurant placement.** Justin Bieber has surprisingly decided to open a series of restaurants along the highway between Geneva and Bern. The n possible locations are along a straight line, and the distances of these locations from the start of the highway in Geneva are, in kilometers and in arbitrary order, $m_1, m_2, \ldots, m_n$. The constraints are as follows:

- At each location, Justin may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \ldots, n$.
- Any two restaurants should be at least k kilometers apart, where k is a positive integer.

As Justin is not famous for his algorithmic skills, he needs your help to find an optimal solution, i.e., **design and analyze** an *efficient* algorithm to compute the maximum expected total profit subject to the given constraints.

Solution: Let M be the array containing the distances of the restaurants from Geneva, in increasing order; we can compute this array in $O(n \log n)$ time (e.g. using mergesort, heapsort, etc.), and let p be the array whose i-th element contains the profit we gain by opening a restaurant at position $M[i]$. Now consider the opening of the leftmost restaurant: since it is the leftmost restaurant, there will be no restaurant closer to Geneva. Furthermore, opening this restaurant restricts the set of locations where we can open other restaurants, since restaurants must be at least k kilometers apart. Thus, given that the leftmost restaurant is opened at a particular location i, we can use the following (recursive) strategy to open the most profitable set of restaurants. Let $c[i]$ be the entry in array c containing the optimal profit for the possible set of restaurant locations whose leftmost restaurant is opened at location $M[i]$. Formally we have:

$$c[i] = p[i] + \max_{i<j\le n : M[j] \ge M[i]+k} c[j].$$

We implement this recursive formulation efficiently using the Bottom-up approach:

```
RESTAURANTS($M, p$)
1. $c[n] \leftarrow p[n]$
2. for $i = n$ to 1
3.     $l \leftarrow 0$
4.     for $j = i + 1$ to $n$
5.         if $c[j] > l$ and $M[j] \ge M[i] + k$
6.             $l \leftarrow c[j]$
7.         end-if
8.     end-for
9.     $c[i] \leftarrow p[i] + l$
10. end-for
11. return $\max_{1 \le i \le n} c[i]$
```

Let us analyze the running time of the above algorithm: the two nested loops will cause lines 5-7 to be executed $\Theta(n^2)$ times. Line 11 requires $\Theta(n)$ operations, while sorting requires $O(n \log n)$ operations. Hence, the total running time is $\Theta(n^2)$.

Note that we can design an algorithm with $O(n \log n)$ running time too. As a preprocessing step, we sort the possible locations according to their distances, then, we calculate the nearest possible location on the right of m_i which is at least k kilometers away. We denote such location for each i by $b[i]$. If no such location exists, we set $b[i] = n + 1$. The algorithm below returns the array b in $\Theta(n)$ time.

```
LeftPossible($M$)
1. pointer = 1
2. for $i = 1$ to $i = n$
3.     while pointer <= n+1 and $M[pointer] - M[i] < k$
4.         $pointer \leftarrow pointer + 1$
5.     $b[i] \leftarrow pointer$
6. return $b$
```

Let $d[i]$ be the maximum profit that can be achieved from opening restaurants from m_i to m_n. At each position, we can either open a restaurant or keep it closed. If we open a restaurant, our profit would be $p[i] + d[b[i]]$. If we do not open a restaurant in m_i, then our profit would be $d[i + 1]$. Here we implement this algorithm:

```
FastRestaurant($M, p$)
1. $d[n + 1] \leftarrow 0$
2. b = LeftPossible($M$)
3. for $i = n$ to $i = 1$
4.     $d[i] = \max\ (d[i + 1], d[b[i]] + p[i])$
5. return $b[1]$
```

The total running time for this algorithm is $O(n \log n) + \Theta(n)$.