ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Midterm Exam, Algorithms 2014-2015

- 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. In particular, do not only give pseudocode 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 / 25 points	Problem 2 / 25 points	Problem 3 / 25 points	Problem 4 / 25 points

Total / 100

1 *(25 pts)* **Recurrences, Stacks, and Trees.**

1a *(9 pts)* Give tight asymptotic bounds for the following recurrences (assuming that $T(1) = \Theta(1)$):

(i) $T(n) = 2T(n/4) + \Theta(1)$

(ii) $T(n) = 2T(n/4) + \Theta(n)$

(iii) $T(n) = 8T(n/4) + \Theta(n)$

(iv) $T(n) = 32T(n/4) + \Theta(n^{2.5})$

Solution:

(i) $T(n) = \Theta(\sqrt{n})$

(ii) $T(n) = \Theta(n)$

(iii) $T(n) = \Theta(n^{\log_4 8})$ or equivalently $T(n) = \Theta(n^{1.5})$

(iv) $T(n) = \Theta(n^{2.5} \log n)$

1b *(8 pts)* Consider the following procedure UNKNOWN that takes as input an integer $n \geq 0$.

```
UNKNOWN(n)

1. Let S be an empty stack
2. PUSH(S, 1)
3. PUSH(S, 1)
4. while n > 1
5.      tmp1 = POP(S)
6.      tmp2 = POP(S)
7.      PUSH(S, tmp1)
8.      PUSH(S, tmp1 + tmp2)
9.      n = n − 1
10. return POP(S)
```

Write a recursive formulation of UNKNOWN(n), i.e., write UNKNOWN(n) as a function of UNKNOWN(0), UNKNOWN$(1), \ldots,$ UNKNOWN$(n-1)$ whenever $n \geq 2$. Also indicate the value of UNKNOWN(n) when $n = 0$ and $n = 1$.

Solution:

$$\text{UNKNOWN}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ \text{UNKNOWN}(n-1) + \text{UNKNOWN}(n-2) & \text{if } n \geq 2 \end{cases}$$

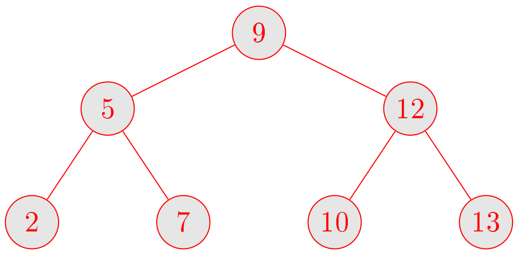
(Basically UNKNOWN calculates the Fibonacci numbers.)

1c *(8 pts)* **Illustrate/draw the binary search tree** obtained by executing

1. Let T be an empty binary search tree
2. Tree-Insert$(T, 9)$
3. Tree-Insert$(T, 5)$
4. Tree-Insert$(T, 2)$
5. Tree-Insert$(T, 12)$
6. Tree-Insert$(T, 13)$
7. Tree-Insert$(T, 7)$
8. Tree-Insert$(T, 10)$

Is it a good binary search tree for the given set of keys? Motivate your answer.

Solution:



The tree is a good binary search tree of the given set of keys as it is has logarithmic height and this is important because most operations take time proportional to the height of the tree.

2 *(25 pts)* **Divide-and-Conquer.** Consider the procedure POWER that takes as input a number a, a non-negative integer n and returns a^n:

```
POWER(a, n)
1. if $n = 0$
2.       return 1
3. if $n = 1$
4.       return a
5. $q = \lfloor \frac{n}{4} \rfloor + 1$
6. return POWER$(a, q)\cdot$ POWER$(a, n - q)$
```

2a *(10 pts)* Let $T(n)$ be the time it takes to invoke POWER(a, n). Then the recurrence relation of $T(n)$ is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 0 \text{ or } n = 1, \\ T(\lfloor n/4 \rfloor + 1) + T(n - \lfloor n/4 \rfloor - 1) + \Theta(1) & \text{if } n \geq 2. \end{cases}$$

Prove that $T(n) = O(n)$ **using the substitution method**.

In your proof you may ignore the floor function, i.e., you can replace $\lfloor n/4 \rfloor$ by $n/4$.

Solution:

We shall show that $T(n) = O(n)$.

Claim 0.1 *There exist positive constants b, b' and n_0 such that $T(n) \leq bn - b'$ for all $n \geq n_0$.*

Proof. As always in the substitution method we do not need to worry about the base case as we can always select b large enough so that the base case holds.

Now consider the inductive step, i.e., assume statement true for all $k < n$ and we prove it for n:

$$\begin{aligned} T(n) &\leq T(n/4 + 1) + T(3n/4 - 1) + c \qquad \text{for some absolute constant } c \\ &\leq b(n/4 + 1) - b' + b(3n/4 - 1) - b' + c \\ &= bn - 2b' + c \qquad \text{selecting } b' \text{ to be a greater constant than } c \text{ gives us} \\ &\leq bn - b', \end{aligned}$$

as required.

□

2b *(15 pts)* **Design** and **analyze** a modified procedure FastPower(a, n) that returns the same value a^n but runs in time $\Theta(\log n)$.

A solution that only works when n is a power of 2, i.e., $n = 2^k$ for some integer $k \geq 0$, gives partial credits.

(Note that a^n is *not* a basic instruction and should therefore not be used.)

Solution:

Note that

$$a^n = \begin{cases} 0 & \text{if } n = 0, \\ (a^{\lfloor n/2 \rfloor})^2 & \text{if } n > 0 \text{ is even}, \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{if } n > 0 \text{ is even}. \end{cases}$$

This motivates the following Divide-and-Conquer solution which modifies the previous solution:

```
FastPower($a, n$)
1. if $n = 0$
2.     return 0
3. $tmp$ =FastPower($a, \lfloor n/2 \rfloor$)
4. if $n$ is even
5.     return $tmp \cdot tmp$
6. else ($n$ is odd)
7.     return $tmp \cdot tmp \cdot a$.
```

To analyze this, let $T(n)$ be the time it takes to execute FastPower(a, n). Note that the divide and combine part both takes time $\Theta(1)$. Further we conquer one subproblem of size $\lfloor n/2 \rfloor \approx n/2$. Therefore we have that $T(n) = T(n/2) + \Theta(1)$ and $T(0) = \Theta(1)$. By the Master method we thus have that $T(n) = \Theta(\log n)$ as required.

3 *(25 pts)* **Dynamic Programming.** Consider the weighted version of the classic change making problem that addresses the following question: how can a given amount of money be made with the least weight of coins of given denominations and weights? The formal definition is as follows:

> **INPUT:** a set of n integer coin values $\{v_1, v_2, \ldots, v_n\}$ with associated weights $\{w_1, w_2, \ldots, w_n\}$ and a positive integer T. The coin values satisfy $v_1 = 1$ and $v_i \leq v_{i+1}$ for $i = 1, \ldots, n-1$.
>
> **OUTPUT:** The smallest weight W such that there exist non-negative integers $x_1, x_2, \ldots, x_n$ satisfying
>
> $$\sum_{i=1}^{n} x_i \cdot v_i = T \qquad \text{and} \qquad \sum_{i=1}^{n} x_i \cdot w_i = W.$$
>
> Here x_i stands for the number of times the coin of value v_i is used to achieve the total value T.

An example input is the following:

$T = 7$ and there are $n = 3$ coin values:

i	1	2	3
v_i	1	2	5
w_i	6	14	29

The correct output to the above input is 41 as the smallest weight change is to use $x_1 = 2$ coins of value v_1 and $x_2 = 0$ coins of value v_2 and $x_3 = 1$ coin of value v_3.

3a *(10 pts)* Let $r[t]$ equal the minimum weight W_t required to achieve a total value of t, i.e., such that there exist non-negative integers $x_1, x_2, \ldots, x_n$ satisfying

$$\sum_{i=1}^{n} x_i \cdot v_i = t \qquad \text{and} \qquad \sum_{i=1}^{n} x_i \cdot w_i = W_t.$$

Complete the recurrence relation for $r[t]$ that can be used for dynamic programming.

Solution:

$$r[t] = \begin{cases} \infty & \text{if } t < 0 \\ 0 & \text{if } t = 0 \\ \min_{i:1\leq i\leq n} r[t - v_i] + w_i & \text{if } t > 0 \end{cases}$$

3b *(15 pts)* Use either the bottom-up approach or top-down with memoization to **design an efficient algorithm** for the weighted change making problem. You should also **give a tight analysis** of the running time of your algorithm.

(write your solution to 3b on next page)

Solution:

We do bottom-up with memory r. We fill it in from index 0 to T. we then return the value $r[T]$ (the minimum weight required to achieve a total value of T)

```
BottomUp($n, v, w$)
1. Create array $r[0, \ldots, T]$
2. $r[0] = 0$
3. for $t = 1, 2 \ldots, T$
4.     $best = \infty$
5.     for $i = 1, \ldots, n$
6.         if $t - v_i \geq 0$  and  $r[t - v_i] + w_i < best$
7.             $best = r[t - v_i] + w_i$
8.     $r[t] = best$
9. return $r[T]$.
```

The running time is dominated by the operations inside the two nestled loops: they will execute nT times. Therefore our running time is $\Theta(nT)$. Another way to see it is that we have T cells and each cell takes $\Theta(n)$ time to fill in.

4 *(25 pts)* **Heaps.** Consider the following problem:

> **INPUT:** A positive integer k and an array $A[1 \ldots n]$ consisting of $n \geq k$ integers that satisfy the max-heap property, i.e., A is a max-heap.
>
> **OUTPUT:** An array $B[1 \ldots k]$ consisting of the k largest integers of A sorted in non-decreasing order.

Design and **analyze** an efficient algorithm for the above problem. Ideally your algorithm should run in time $O(k \log k)$ but the worse running time of $O(\min\{k \log n, k^2\})$ is also acceptable.

Slightly slower algorithms give partial credits.

Solution:

Let us describe an algorithm that takes $\Theta(k \log k)$ time. First some intuition. Note that the largest element of the heap is located in $A[1]$. Therefore, we know that $B[k] = A[1]$. Now where can the second largest element be? By the max-heap property it can either be $A[2]$ or $A[3]$. Suppose that it is $A[2]$ then we put $B[k-1] = A[2]$. Now the 3rd largest element can either be $A[3]$ or one of the children of $A[2]$, i.e., $A[4]$ or $A[5]$ and so on. To find the largest of these elements we need to pick the maximum. To be able to pick the maximum quickly in each iteration we will keep track of a second max-heap called H. Our algorithm now works as follows:

```
LARGESTK($A, B$)
1. Create an empty heap $H$
2. $B[k] = A[1]$
3. Insert $A[2]$ and $A[3]$ to $H$
4. for $i = k-1, k-2 \ldots, 1$
5.       tmp = EXTRACT-MAX(H)
6.       $B[i] = tmp$
7.       Insert $tmp$'s children in $A$ to $H$, i.e.,
             if $tmp$ is $A[i]$ then insert $A[2i]$ and $A[2i+1]$ to $H$.
```

Note that to implement Step 7 efficiently we need to be able to find the index i quickly so that *tmp* corresponds to i. This can easily be done by for example, storing the index along with the key in a tuple $(A[i], i)$ in the heap H (where $A[i]$ is the key).

That the above works follows from the max-heap property. Indeed, the next largest element must be a child of already selected elements. To see this, suppose toward contradiction that the next largest element is not a child. But then its parent is larger and was not yet selected which is a contradiction.

Let us now turn to the running time of our algorithm. To create an empty heap takes constant time. Similarly, Steps 2 and 3 take constant time. Step 5 takes at most $O(\log k)$ time as H will contain at most $2k$ elements. Step 6 takes constant time. Step 7 takes $O(\log k)$ time again because H contains at most $2k$ elements. Since Steps $5-7$ are executed k times the total running time will be $O(k \log k)$.

As a final comment, let me shortly express the idea of how to get an algorithm that runs in time $O(\min\{k^2, k \log n\})$ (which is worse than the previous running time). Observe that any element that we are interested in is of distance at most k from the root (this follows because all ancestors of an element are larger than that). Therefore we can restrict our attention to the heap $A[1 \ldots \min\{2^k, n\}]$. Running k steps of heap sort on this restricted heap will take time $O(\min\{k^2, k \log n\})$.