ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Final 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 pseudocode 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.
- **Do not touch until the start of the exam.**

Good luck!

Name: _____ **N° Sciper:** _____

Problem 1 / 23 points	Problem 2 / 12 points	Problem 3 / 16 points	Problem 4 / 18 points	Problem 5 / 19 points	Problem 6 / 12 points

Total / 100

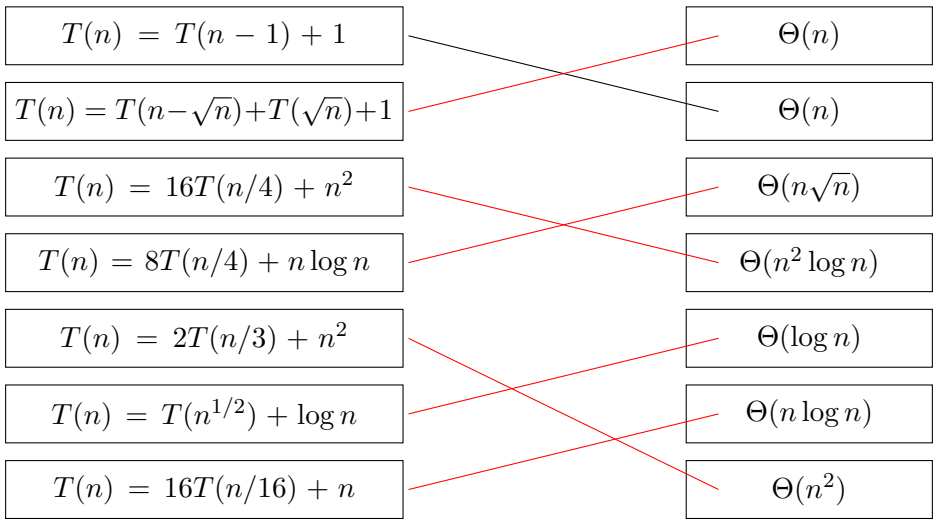
1 *(23 pts)* **Basic questions.** This problem consists of five subproblems (1a-1e) for which you **do not** need to motivate your answers.

1a *(5 pts)* Consider the task of hashing n keys into m different hash table slots $\{1, 2, \ldots, m\}$. We assume simple uniform hashing. Define a 2-*collision* as a pair of distinct keys $\{k_1, k_2\}$ that are hashed to the same hash table slot. What is the largest value of n (asymptotically as a function of m) which ensures that the expected number of 2-collisions is less than $1/10$?

Solution: There are $\binom{n}{2}$ pairs of keys. For every pair the probability of collision is $1/m$. Thus the expected number of collisions is $\binom{n}{2}/m$, so $n = \Theta(\sqrt{m})$.

1b *(6 pts, 1 pt for each correct pair)* Pair up each of the recurrence relations on the left side with the correct asymptotic growth function on the right side (assuming that $T(1) = 1$). For example the recurrence $T(n) = T(n-1) + 1$ has asymptotic growth $\Theta(n)$.

Solution:



1c *(6 pts)* Consider the code for the function UNKNOWN below:

```
UNKNOWN(n):
1. if n < 100
2.       return
3. q = ⌊n/4⌋
4. UNKNOWN(q)
6. UNKNOWN(n − 2q)
7. PRINT "Almost done!"
8. UNKNOWN(n − 2q)
```

Let $T(n)$ denote the runtime of UNKNOWN(n). Write down a recurrence relation for $T(n)$:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < 100 \\ T(n/4) + 2T(n/2) + c & \text{otherwise} \end{cases}$$

for some constant c.

Which of the following statements about the asymptotic growth of $T(n)$ as a function of n are true:

A: $T(n) = \Omega(n)$
B: $T(n) = O(n^2)$
C: $T(n) = \Omega(n^3)$
D: $T(n) = O(n)$

Solution: The correct statements are _____ *(3pts)*

Solution: To get the answer, we assume $T(n) \geq dn^a$, where d is a constant, and check for which a the assumption holds.

We have

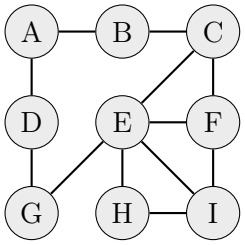
$$\begin{aligned} T(n) &\geq d\left(\frac{n}{4}\right)^a + 2d\left(\frac{n}{2}\right)^a + c \\ &= \left(\frac{d}{4^a} + \frac{2d}{2^a}\right) n^a + c \\ &= d\left(\frac{1}{4^a} + \frac{2}{2^a}\right) n^a + c. \end{aligned}$$

Now, $T(n) \geq dn^a$ holds for any n if

$$\tfrac{1}{4^a} + \tfrac{2}{2^a} \geq 1. \qquad (1)$$

(We can ignore the addition of c in this case, as for any c and sufficiently large n we have $n^f > c$ where f is any positive constant.) Inequality (1) holds for any positive $a \leq 1.1$, and even for some a larger than 1.1. However, it does not hold for $a \geq 2$ and sufficiently large n regardless of the value of c. This implies that A and B **are** correct, while C and D are **not**. Therefore, the answer is A, B.

1d *(3pts, 1.5pts for correct BFS and 1.5pts for correct DFS)* Consider the following undirected graph:



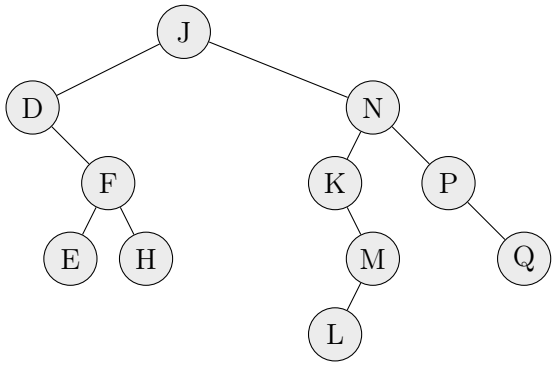
Write down the labels of the vertices in the order in which they are discovered by traversing the graph using breadth-first-search (BFS) and using depth-first-search (DFS), starting from the vertex labeled by A. Assume that the vertices are chosen in alphabetical order: that is, whenever the BFS/DFS procedure has several options, it chooses the next vertex according to the alphabetical order.

Solution:

The order in which vertices are discovered by BFS: A, B, D, C, G, E, F, H, I

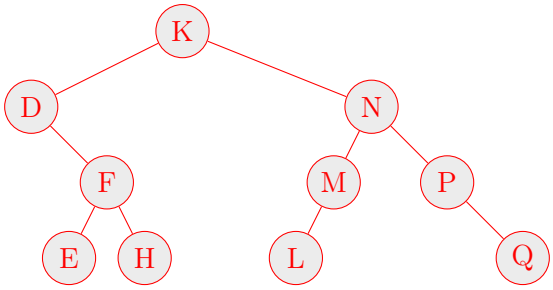
The order in which vertices are discovered by DFS: A, B, C, E, F, I, H, G, D

1e *(3 pts)* Draw the binary search tree obtained by deleting the root from the following binary search tree:



The deletion is done by the procedure TREE-DELETE explained in class.

Solution: Because the root, node J, has both left child and right child, we should find the smallest node in the right subtree, which is node K, replace the root with K and then remove node K from the right subtree. The resulting binary tree follows:



2 *(12 pts)* **Angry birds.**

A number k of birds live in a city, and need to get from their homes to their common workplace every morning (the birds choose to walk to work in the morning, as they only get their first cup of coffee at work and are not sufficiently alert for flying before then). They also insist that the paths they take to work every morning do not overlap except possibly on road intersections – they are somewhat grumpy before their morning coffee, and get angry if there is any delay on the narrow streets.

In this problem you are given the map of streets of the city (i.e. a directed graph where edges are the streets and vertices are the street intersections), the k distinct locations (vertices) $s_i, i = 1, \ldots, k$ where the birds live, and the workplace t (a vertex). Your task is to determine whether it is possible to allocate paths to birds as required, and find such k paths if possible. Your solution should run in time $O((m + k + n)k)$ where m is the number of roads in the city and n is the number of intersections.

(In this problem you are required to (a) explain a correct algorithm with the desired running time and to (b) analyze its running time.)

Solution: We add a new vertex s (not existing in G initially), and connect it to every s_i, $i = 1, \ldots, k$. That is, we add edges (s, s_i), for $i = 1, \ldots, k$. All edge capacities are exactly 1. We claim that it is possible to allocate k routes to the birds as required iff the maximum s-t flow is at least k. We prove the two directions of "iff" separately.

($\Longrightarrow$) We show that if it is possible to allocate k routes to birds as required, then the maximum s-t flow is at least k.

Let $P_1, \ldots, P_k$ be the routes for birds. Observe that P_i starts at s_i. Then, paths $sP_1, \ldots, sP_k$ are k edge disjoint flow paths between s and t. Hence, the maximum s-t flow is at least k.

($\Longleftarrow$) We prove that if the maximum s-t flow is at least k, then we can find k routes to the birds as required.

Let $F_1, \ldots, F_k$ be some k s-t flow paths. In fact, it is easy to see that there can not be more than k flow paths, as the degree of s is exactly k. Without loss of generality, assume that F_i contains edge (s, s_i), for every $i = 1, \ldots, k$. Let P_i be the path defined by the subpath of F_i starting at s_i (and also ending at t). In particular, $|F_i| - 1 = |P_i|$. Then, the paths $P_1, \ldots, P_k$ represent k routes to the birds as required.

Running time analysis. We only need to find the maximum s-t flow in the graph described above. As already noted, the maximum flow is at most k. If the maximum flow is k exactly, we should output the k flow paths, while excluding vertex s of each of the paths. If the maximum flow is less than k, then there are no k routes to the birds as required.

To find the maximum s-t flow, as seen in the class, we can run BFS as long as it increases the s-t flow. So, in this case we would run BFS at most k times. Recall that originally our graph had n vertices, and m edges. In addition, we have added one vertex s and k edges to the graph. So, the total running time of running BFS ru at most k times is $O((m+k+n+1)k) = O((m+k+n)k)$.

3 *(16 pts)* **Maintaining the median of a sequence.** Design a data structure that supports the following two operations: (a) insert a number and (b) return the median of numbers inserted so far. All operations should be performed in time $O(\log n)$, where n is the number of elements currently stored in the data structure. *Hint: use two heaps.*

Recall that the median of n numbers $a_1 \leq a_2 \leq \cdots \leq a_n$ is $a_{\lceil n/2 \rceil}$ if n is odd and $\frac{a_{(n/2)}+a_{(n/2+1)}}{2}$ if n is even.

(In this problem you are required to (a) explain how the numbers are stored in the data structure, to (b) explain the implementations of the "insert" and "median" operations with running time $O(\log n)$, and to (c) analyze the running time of your algorithms.)

Solution:

We solve this problem using two heaps. The first heap L is a max heap containing only the numbers less than equal to the median and the second heap H is a min heap containing only the numbers greater equal to the median. During all stages of the algorithm, we make sure that the size of these two heaps are almost the same, i.e., the size of L is no less than the size of H and no more than the size of H plus one.

$$H.size \leq L.size \leq H.size + 1$$

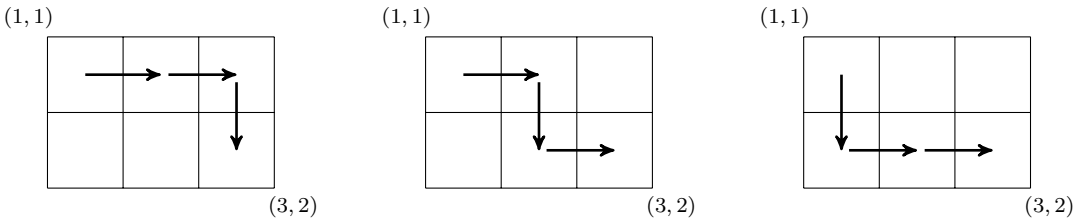
We now explain the algorithm for adding elements and finding the median and show that the above inequality holds. Notice that at the beginning, since both heaps are empty, the above Inequality holds.

- **Inserting a new number:** Given a new number we first add it two L. Then if $L.size > H.size + 1$ we remove the biggest number for L and add it to H, i.e., we remove the head of L and add it to H. This guarantees that, at the end of this step, the above inequality holds given that it was correct before this step. It is clear that the running time of this algorithm is $O(\log n)$, since we are adding and removing at most three times to a heap of size at most n.
- **Finding the median:** If n is odd, the median is the largest element in L, otherwise, the median is the average of the largest element of L and the smallest element of H. Recall that L is a max heap and H is a min heap, so we can find both these numbers in constant time.

4 *(18 pts)* **Square city.** Bob lives in a two-dimensional n by m city. He lives in cell $(1,1)$ and works in cell (n,m). He wants to get to work by passing through the city. The time he has for getting to work is T, and entering any cell (i,j) takes t_{ij} time. Also, there are c_{ij} candies in each cell (i,j) and he can collect them if he enters the cell. Bob can only go right or down through the city. **Design and analyze** an algorithm that finds the maximum number of candies that Bob can collect while still getting to work on time.

- **Input:** The input consists of integers n, m, T and t_{ij}, c_{ij} for all $1 \le i \le n$, $1 \le j \le m$.
- **Output:** Output the maximum number of candies that Bob can collect while still getting to work on time. If there is no way that he can get to work on time, output 0.
- He can move from cell (i,j) to cells $(i+1,j)$ and $(i,j+1)$.
- All the t_{ij}, c_{ij} values are positive integers.

Here are three examples of paths that Bob can take from his home to work in a 2 by 3 city.



Your solution should run in time polynomial in n, m and T.

(In this problem you are required to (a) explain a correct algorithm with the desired running time and to (b) analyze its running time.)

Solution: We solve this problem using dynamic programming. We first define a recursion that solves this problem, and then analyze the running time of a bottom-up implementation for our recursion. To that end, let $d(i,j,k)$ be the maximum amount of the candies that we can collect in time less than equal k arriving to cell (i,j). We let $d(i,j,k) = -\infty$ if there is no such path.

Since Bob is already in cell $(1,1)$, $d(1,1,k) = c_{11}$ for $0 \le k \le T$. Moreover, for any cell (1,j), such that $j > 1$, we can get to this cell only from (1,j-1), so $d(1,j,k) = d(1,j-1,k-t_{1j}) + c_{1j}$ if $k \ge t_{1j}$ and $-\infty$ otherwise. Similarly, for $i > 1$ we get $d(i,1,k) = d(i-1,0,k-t_{i1}) + c_{i1}$ if $k \ge t_{i1}$ and $-\infty$ otherwise. For any other cell (i,j), there are two options to get to this cell, i.e., cells (i-1,j) and (i,j-1) and we choose the best from this two options.

$$d(i,j,k) = \begin{cases} -\infty & \text{if } k < t_{ij} \\ c(i,j) & \text{if } i = 1, j = 1, k \ge t_{ij} \\ d(i,j-1,k-t_{ij}) + c_{ij} & \text{if } i = 1, j \ne 1, k \ge t_{ij} \\ d(i-1,j,k-t_{ij}) + c_{ij} & \text{if } j = 1, i \ne 1, k \ge t_{ij} \\ \max\left(d(i-1,j,k-t_{ij}), d(i,j-1,k-t_{ij})\right) + c_{ij} & \text{otherwise} \end{cases}$$

Now we can easily find the answer to our problem using the above recursion. The answer is 0 if $d(n,m,T) = -\infty$ and $d(n,m,T)$ otherwise.

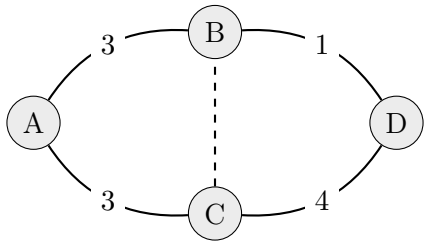
The number of the cells that we need to fill is $O(nmT)$ and we use constant time to fill each of them. Also we need constant time to find the solution after finding the value if $d(n, m, T)$. Therefore the total running time if a top-down implementation of this recursion is $O(nmT)$ which is polynomial with in n, m, T.

5 *(19 pts)* **Minimum spanning trees.** In this problem you are given an instance to the minimum spanning tree problem (i.e., a connected undirected graph with edge weights). You are further considering adding a new edge $e = \{u, v\}$ to the graph, and you want to assign it the largest possible weight while maintaining that it is part of every minimum spanning tree. More formally, your task is to **design and analyze** an efficient algorithm for the following problem:

> Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$ and an edge $e = \{u, v\} \notin E$, return the maximum weight α such that if $w(e) < \alpha$ then *every* minimum spanning tree of $(V, E \cup \{e\})$ contains e.

For full credit, your algorithm should run in time $O((|V| + |E|) \log |V|)$.

Example: Consider the following input where solid edges are those in E and the dashed edge is e.



The correct output of the algorithm is $\alpha = 3$ because of the following:

- If $w(e) < 3$, the minimum spanning trees of $(V, E \cup \{e\})$ are $\{\{B, D\}, \{B, C\}, \{A, B\}\}$ and $\{\{B, D\}, \{B, C\}, \{A, C\}\}$.
- If $w(e) = 3$, a minimum spanning tree of $(V, E \cup \{e\})$ is $\{\{B, D\}, \{A, B\}, \{A, C\}\}$ (so 3 is the maximum possible weight).

(In this problem you are required to (a) explain a correct algorithm with the desired running time, to (b) **prove that your algorithm outputs the correct value***, and to (c) analyze the running time of your algorithm.)*

Solution:

First run Kruskal's algorithm to find a minimum spanning tree on graph $G = (V, E)$. It runs in time $O(|E| \log |E|)$ or equivalently $O(|E| \log |V|)$. Name the resulting spanning tree T. Then find the unique path that connects the two nodes u and v in the tree. This can be done by running a BFS in time $O(|E|)$. The claim is that if e^* is the edge with maximum weight in this path, then we can set α to be $w(e^*)$. The proof of this follows:

If there exists a minimum spanning tree, T', that doesn't contain the newly added edge e with weight $w(e) < \alpha$ then it means that the weight of all the edges on the path between u and v in T' is at most $w(e)$. Now, note that e^* in tree T defines a cut with u on one side and v on the other side. If we look at the same cut in T', then there is only one edge passing through the cut and that edge is on the path connecting u and v and as said earlier weight of that edge is at most α and hence absolutely less than $w(e^*)$. This is a contradiction because it means that T is not a minimum spanning tree because we can remove e^* from it and add the edge of T' that passes through the cut to it.

6 *(12 pts)* **Investment strategy with advice.** In this problem you will design a strategy for playing a stock price prediction game with the help of n experts. Every morning each of the n experts makes a prediction: `+1` for **up** or `-1` for **down**, and based on their advice you make your own prediction (i.e., you output `+1` or `-1`). In the evening the true answer is revealed: `+1` if the stock actually went up and `-1` if the stock went down. It is guaranteed that one of the experts is perfect, i.e., gives the right answer every day, but other experts may behave arbitrarily and you do not know which of the experts is the perfect one. Show that it is possible to play the prediction game for any number T of rounds so as to output the wrong prediction at most $\lfloor \log_2 n \rfloor$ times.

An example timeline over $T = 4$ days with $n = 3$ experts is as follows:

morning (prediction)	evening (outcome)	morning (prediction)	evening (outcome)	morning (prediction)	evening (outcome)	morning (prediction)	evening (outcome)
`+1`		`-1`		`+1`		`-1`	
`+1`	`+1`	`-1`	`-1`	`-1`	`+1`	`+1`	`-1`
`+1`		`+1`		`+1`		`+1`	

Notice that after the first day, all experts look like they could be the perfect one. On the second day, the prediction of the third expert is wrong, so we then know that the perfect expert can only be the first or the second. By the same argument, after day 3 we know that the perfect expert is the first one. So after day 3 it is easy to always make the correct prediction. However, we are only allowed to make at most $\lfloor \log_2(n) \rfloor = 1$ erroneous prediction in total, meaning that we also have to perform fairly well even before we have identified the perfect expert (who may not be unique in general).

(In this problem you are required to (a) explain a strategy with the desired performance and to (b) analyze the number of wrong answers that your strategy can give in the worst case, i.e., prove that your algorithm only makes at most $\lfloor \log_2(n) \rfloor$ mistakes.)

Solution:

Maintain a set of experts S – initially the set of all experts. At each step, use the prediction of the majority of experts in this set to use as your answer. (If they are split evenly, output either answer.) Then, if it turns out to be a mistake, remove from S all experts which made this mistake in this round.

Note that, because the existence of a perfect expert is guaranteed, the set S will never become empty. At any time step, S contains those experts who have never yet been wrong.

Why will we make at most $\lfloor \log_2 n \rfloor$ mistakes? Notice that whenever we make a mistake, it is because we listened to the advice of at least half of the experts in S, which was wrong. These experts will now be removed from S. In short, whenever we make a mistake, this will cause S to become at most one half its size. Because initially the size of S is n and it is at least 1 at the end, there may be at most $\lfloor \log_2 n \rfloor$ mistakes. (Otherwise the size at the end would be $\leq n \cdot (1/2)^{\lfloor \log_2 n \rfloor + 1} < 1$.)