ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Final 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 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.
- Attached at the end of the exam is a French translation.
- **Do not touch until the start of the exam.**

Good luck!

Name: _____ **N° Sciper:** _____

Problem 1 / 22 points	Problem 2 / 14 points	Problem 3 / 18 points	Problem 4 / 18 points	Problem 5 / 14 points	Problem 6 / 14 points

Total / 100

1 *(22 pts)* **Basic questions.** This problem consists of five subproblems (1a-1e) for which you **do not** need to motivate your answers.

1a *(6 pts)* **Sorting**

Insertion Sort and Quick Sort have the same worst case running time. True or False? True

Both Heap Sort and Merge Sort require linear extra space to sort, i.e. are **not** in place. True or False? False

Let $A[1 \dots 6] =$ | 1 | 3 | 7 | 4 | 5 | 6 | be an array consisting of 6 numbers. If we use randomized quick sort to sort A, the probability that $A[2] = 3$ and $A[5] = 5$ are compared is $1/2$. True or False? False

1b *(4 pts)* **Uniform Hashing**
Suppose you are hashing n elements into m slots using uniform hashing. Asymptotically, what is the value of m in terms of n that ensures that

(A) Expected number of collisions is $\Theta(\sqrt{n})$.

$m = \Theta(n^{3/2})$

(B) Expected number of elements mapped to any given slot of the table is $\Theta(1)$.

$m = \Theta(n)$

1c *(8 pts)* **Recurrences**

Consider the functions FOO(n) and BAR(n), whose pseudocodes are given below. The functions take as input an integer n.

```
FOO(n)
1. if n > 100
2.       u= ⌊n/3⌋
3.       FOO(u)
4.       FOO(n − u)
5.       BAR(⌊√n⌋)

BAR(n)
1. if n > 1000
2.       BAR(n − 1)
3.       for i = 1 to n
4.             PRINT ('Ok')
```

Denote the running time of FOO(n) by $T(n)$, denote the running time of BAR(n) by $S(n)$.

Write down the recurrence relation for $T(n)$ in terms of $S(n)$.

$T(n) = T(n/3) + T(2n/3) + S(\sqrt{n}), T(1) = \Theta(1)$

Write down the recurrence relation for $S(n)$.

$S(n) = S(n-1) + \Theta(n), S(1) = \Theta(1)$

What is the runtime of $\textsc{Bar}(n)$ asymptotically as a function of n?

$S(n) = \Theta(n^2)$

What is the runtime of $\textsc{Foo}(n)$ asymptotically as a function of n?

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

1d *(2 pts)* Consider the recurrence $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + n^3$, $T(1) = \Theta(1)$, for some $\alpha, \beta, \gamma \in (0, 1)$ that satisfy $\alpha^3 + \beta^3 + \gamma^3 = 1$. Give a tight asymptotic bound for $T(n)$. You do not need to motivate your answer.

Solution: The answer is $T(n) = \Theta(n^3 \log n)$

1e *(2 pts)* Consider the recurrence $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + n^2$, $T(1) = \Theta(1)$, for some $\alpha, \beta, \gamma \in (0, 1)$ that satisfy $\alpha^2 + \beta^2 + \gamma^2 = 0.999$. Give a tight asymptotic bound for $T(n)$. You do not need to motivate your answer.

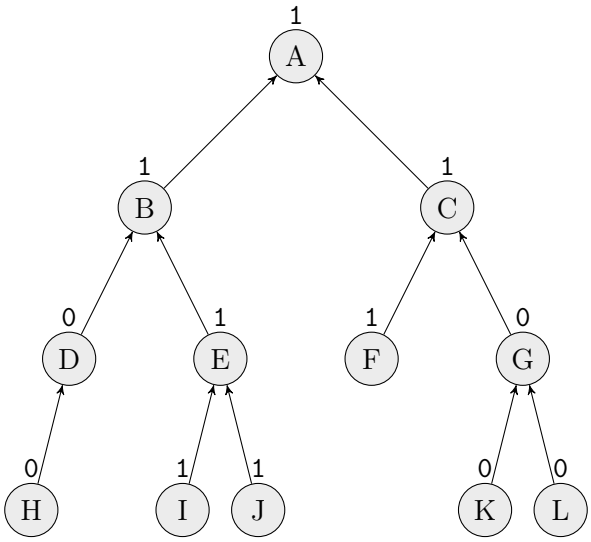
Solution: The answer is $T(n) = \Theta(n^2)$

2 *(14 pts)* **Monochromatic subtrees.** For a rooted binary tree $T = (V, E)$ with nodes labeled with zeros and ones we would like to count the number of nodes $u \in V$ whose subtrees are monochromatic (i.e. either all nodes in their subtree are labelled zero or all nodes in their subtree are labelled one).

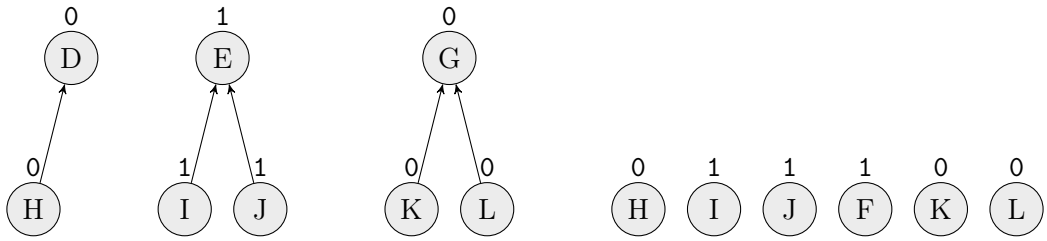
Input: a rooted binary tree $T = (V, E)$ with a label $u.label \in \{0, 1\}$ for every node $u \in T$.

Output: number of nodes $u \in V$ whose subtrees are monochromatic.

For example, consider the tree below (the arrows indicate the child-parent relation):



There are 9 nodes with monochromatic subtrees, namely `D, E, G, F, H, I, J, K, L`. The subtrees are shown below:



In this problem you are required to (a) explain a correct algorithm with the desired running time and to (b) analyze its running time. For full credit your algorithm should run in $O(V)$ time. You may assume that for every node u of T you have access to left child, right child and parent pointers $u.left, u.right, u.parent$ respectively, and $T.root$ is the root of T.

Solution: We use divide and conquer to solve this problem. To count the number of monochromatic subtrees of node u, we count the number of monochromatic subtrees of $u.right$ and $u.left$ and we add them. We also check if the entire subtrees $u.right$ and $u.right$ are monochromatic and the same color as u, in which case the entire subtree u is monochromatic and we increment the number of monochromatic subtrees of u by 1.

The runtime analysis is as follows. In each iteration of the algorithm we recurse on the left subtree as well as the right subtree and then merge the information about the subtrees in

```
1: procedure MONOCHROM(u)
2:     if u.right ≠ ∅ then
3:         (N_r, mono_r, label_r) ←MONOCHROM(u.right)
4:         if u.left ≠ ∅ then
5:             (N_l, mono_l, label_l) ←MONOCHROM(u.left)
6:         else
7:             (N_l, mono_l, label_l) ← (0, true, u.label)
8:         end if
9:     else
10:        (N_r, mono_r, label_r) ← (0, true, u.label)
11:    end if
12:    if mono_l = true and mono_r = true and label_l = label_r = u.label then
13:        label ← u.label
14:        mono ← true
15:    else
16:        mono ← false
17:    end if
18:    N ← N_l + N_r + 1
19:    return (N, mono, label)
20: end procedure
```

constant time. Therefore we spend a constant amount of time per each node of the tree. Hence the runtime is proportional to the number of nodes of the tree, $O(n)$.

3 *(18 pts)* **Packing dominos.** You bought a new board game: an $n \times n$ grid with every cell either square shaped or circular or inactive, together with an unlimited supply of dominos of a rather non-standard form: a circle attached to a square (see Fig. 1 below; inactive cells are shaded in grey). The domino fits onto the board if its circle part is in a circular cell of the board and its square is in a square cell of the board to the left/right, or above/below the circular part. Design an algorithm that determines, given the shape of the board, the maximum number of non-overlapping dominos that can be simultaneously placed on the board. You cannot use inactive cells.

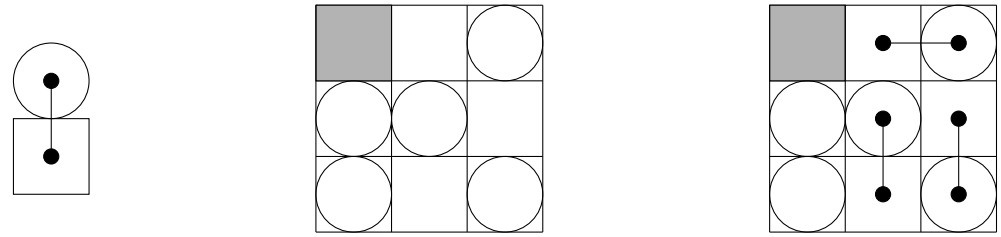


Figure 1. Illustration of the domino (left), a 3×3 board (center) and an optimal packing of 3 dominos onto the board (right).

- **Input:** an integer $n \geq 1$, an $n \times n$ array A with $A_{ij} = 1$ if the (i, j)-th cell is circular and $A_{ij} = 2$ if the (i, j)-th cell is square and $A_{ij} = 0$ if the (i, j)-the cell is inactive.
- **Output:** largest number of non-overlapping dominos that can be placed on the board.

In this problem you are required to (a) describe an efficient algorithm and (b) analyze its runtime. Hint: convert the problem to a graph problem and use an algorithm seen in class.

Solution: Create a bipartite graph $G = (P, Q, E)$, where P is the set of square shapes, Q is the set of circle shapes on the board, and a square shape $u \in P$ is connected by an edge to a circle shape $v \in Q$ if and only if they are neighbors on the board (i.e., if and only if the circle shape v is to the left/right or above/below the square shape u). Then our task is to find a maximum matching in the graph G. This can be done using a reduction to the maximum flow problem shown in class: orient all edges in G to go form P to Q, attach a source s connecting with unit capacity edges to every vertex in P (directed out of s), attach a sink t connecting with unit capacity edges to every vertex in Q (directed into t), and find the maximum $s - t$ flow. Using the Ford-Fulkerson method with BFS to find augmenting paths we get runtime of $O(f_{max} \cdot |E|)$, which is $O(n^4)$ since the maximum flow at most n^2 and the number of edges in the constructed graph is $O(n^2)$.

4 *(18 pts)* **Archipelago.** You're trying to navigate a strange city. The city's map is given by a graph $G = (V, E)$ specifying the intersections and streets of the city. Each edge $e \in E$ has a positive integer weight w_e specifying the time it takes to walk along the street. You are currently located in vertex a and want to get to vertex b as quickly as possible.

Luckily we've seen many algorithms in class that are able to tell you the shortest path to b. Unluckily, the city is located on an archipelago, and the graph G is not connected.

To help the citizens get around, there are a number of ferries connecting the islands (or sometimes two points of the same island). You have access to their schedule: It lists T ferries, specifying for each the place of departure (a vertex), the place of arrival (also a vertex), the departure time and the arrival time. Times are given as non-negative integers; it is currently time 0. See Fig. 2 for an illustration.

Design and analyze an algorithm that, given the graph G and the ferry schedule, returns the shortest time needed to get from a to b if you start at a at time 0.

- **Input:** a graph $G = (V, E)$ with nonnegative weights $w : E \to \mathbb{Z}$ given in adjacency list representation. You have access to a function FERRYSCHEDULE. For every $u \in V$ and integer $i \geq 1$ a call to FERRYSCHEDULE(u, i) in $O(1)$ time returns a triple (v, s, t), where $v \in V$ is the destination, s the departure time and t the arrival time of the i-th ferry out of u, and NULL if there are fewer than i ferries out of u.
- **Output:** shortest time needed to get from a to b if you start at a at time 0.

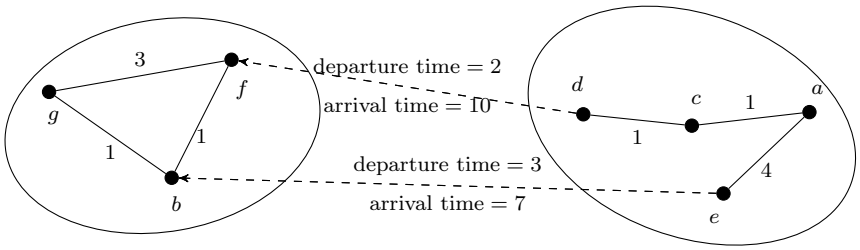


Figure 2. A graph G with two connected components (islands) and two ferries. The ferry from e to be b is faster, but one cannot get to vertex e from vertex a before the ferry's departure time, so the route a, c, d, f, b is optimal.

In this problem you are required to (a) describe an efficient algorithm and (b) analyze its runtime. For full credit your algorithm should run in time $O((E + T) \log V)$ time, where T is the total number of ferries. Hint: adapt an algorithm seen in class.

Solution: We adapt Dijkstra's algorithm. Note that, in order to reach the destination with the shortest path in this question, you may need to wait for the ferries. Also, if you reach a starting point of a ferry after its departure, you will not be able to take that ferry. We need to apply the following changes to Dijkstra's algorithm.

1. In the initialization phase of the algorithm, additional to edges corresponding to roads on the islands, we add **ferry edges** to the graph, by giving them weight equal to their arrival time, i.e., for every $u \in V$, for every i, we add an edge from u to v_i with weight t_i (v_i is

the point of arrival of the i-th ferry out of u, s_i and t_i are the departure and arrival times respectively; note that such edges are directed)

2. After each round of Dijkstra's algorithm, i.e., adding a new vertex u, we update graph as follows: First, for every **ferry edge** originating from u update the weight of the edge by subtracting the shortest path distance $u.d$ of u from a, from the length of the edge, if the departure time of the ferry is no earlier than $u.d$, and make the weight of the edge infinity otherwise (equivalently, remove the edge).

The correctness proof is similar to that for Dijkstra's algorithm.

Runtime analysis: We have at most $|E| + T$ edges at each step of Dijkstra's algorithm. Also, updating part of the algorithm, described above, takes linear time in T. So, in total the runtime of this algorithm is $O((|E| + T) \log |V|)$.

5 *(14 pts)* **Maintaining a minimum spanning tree in a changing graph.** Suppose that you are given a graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$ on edges, and a minimum spanning tree T in G. Now the graph $G' = (V', E')$ is obtained from G by adding a new vertex v together with incident edges (see Fig. 3 below). All edge weights in G' are distinct. Give an efficient algorithm for computing a minimum spanning tree T' in G'.

- **Input:** a graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$ on the edges, a minimum spanning tree T in G. A new vertex v together with incident edges (with weights). All edge weights are distinct.
- **Output:** a minimum spanning tree in the graph G' obtained by adding v with all its edges to G.

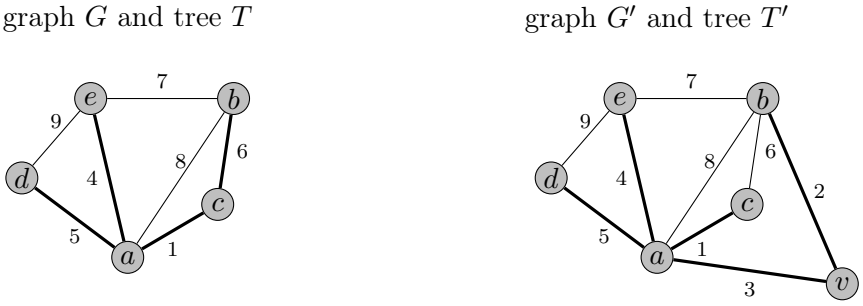


Figure 3. A graph G with its minimum spanning tree T (left) and the graph G' with its minimum spanning tree T' (right). The edges of the spanning trees are shown in bold.

In this problem you are expected to (a) design an efficient algorithm and (b) prove its correctness. For full credit your algorithm should run in $O(V \log V)$ time.

Solution:

Our algorithm is going to be very simple. Take the graph $G'' = (V', T \cup \delta(w))$, where $\delta(w)$ is the set of edges adjacent on w. Then find a minimum spanning tree of G'' using Kruskal or Prim.

We just need to prove that any edge that wasn't in T wasn't going to be in T' either so we don't destroy T' by discarding edges like this.

Consider the minimum spanning tree of G', T'. Recall that for any cut $A \cup B = V'$ the minimum edge crossing the cut must be in T'. Let edge $e \in T'$. Consider the cut A, B created by removing e from T'. e must be the minimum edge crossing this cut in G'. Therefore, either e is not in G (because it is adjacent to w) or e was also the minimum edge crossing the same cut in G. In either case $e \in T \cup \delta(w)$ and is not discarded by our algorithm.

The algorithm runs in time $O(V \log V)$ since the size of the edge set of G'' is $O(V)$.

6 *(14 pts)* **Largest square submatrix of ones.** Given an $m \times n$ matrix of zeros and ones, find the size of the largest **square** sub-matrix of 1's present in it.

- **Input:** an $m \times n$ matrix C with $C_{ij} \in \{0, 1\}$ for all $i = 1, \ldots, m$ and $j = 1, \ldots, n$.
- **Output:** Your task is to compute the size of the largest **square** sub-matrix of 1's present in it. Formally, you should find the largest k such that there exist $1 \leq i \leq m - k + 1, 1 \leq j \leq n - k + 1$ such that $C_{i+a,j+b} = 1$ for all $a \in \{0, 1, \ldots, k-1\}$ and $b \in \{0, 1, \ldots, k-1\}$.

For example, the size of largest **square** sub-matrix of 1's is 3×3 in the matrix below.

1	0	0	1	1	1	0
1	1	0	1	1	0	1
0	0	1	1	1	1	0
1	1	1	1	1	0	1
1	0	1	1	1	1	0
1	1	0	1	0	0	1

Give a dynamic programming solution to this problem.

In this problem you are required to (a) describe the subproblems, (b) give a recurrence relation and (c) analyze the runtime of a bottom-up implementation of your recurrence. For full credit your solution should run in $O(mn)$ time.

Solution: We define the following subproblems :

1. $L_{i,j}$ stores the length of the longest continuous sequence of 1's ending in $C_{i,j}$ in the row C_i.
2. $T_{i,j}$ stores the length of the longest continuous sequence of 1's ending in $C_{i,j}$ in the column C_j.
3. $S_{i,j}$ stores the size of the largest square sub matrix of 1's whose bottom right corner is $C_{i,j}$.

We have the following recurrence relations :-

$$L_{i,j} = \begin{cases} 0, & \text{if } C_{i,j} = 0. \\ L_{i,j-1} + 1, & \text{if } C_{i,j} = 1 \ \& \ j > 1. \\ 1, & \text{if } C_{i,j} = 1 \ \& \ j = 1. \end{cases} \tag{1}$$

$$T_{i,j} = \begin{cases} 0, & \text{if } C_{i,j} = 0. \\ T_{i-1,j} + 1, & \text{if } C_{i,j} = 1 \ \& \ i > 1. \\ 1, & \text{if } C_{i,j} = 1 \ \& \ i = 1. \end{cases} \tag{2}$$

$$S_{i,j} = \begin{cases} L_{i,j}, & \text{if } j = 1. \\ T_{i,j}, & \text{if } i = 1. \\ S_{i-1,j-1} + 1, & \text{if } i > 1 \ \& \ j > 1 \ \& \ \min(T_{i,j}, L_{i,j}) > S_{i-1,j-1} \ . \\ min(T_{i,j}, L_{i,j}), & \text{if } i > 1 \ \& \ j > 1 \ \& \ \min(T_{i,j}, L_{i,j}) \leq S_{i-1,j-1} \ . \end{cases} \tag{3}$$

It is straightforward to come up with a bottom up implementation which runs in $O(mn)$ time. We give the psuedo code below:

MaxSubSquare(C, m, n)
1. Initialize 3 $m \times n$ arrays L, T and S.
2. ans=0.
3. **for** $i = 1$ to m
4. **for** $j = 1$ to n
5. Compute $L_{i,j}$, $T_{i,j}$ and $S_{i,j}$ acc. to recurrences defined above.
6. ans = max(ans,$S_{i,j}$)
7. RETURN ans

Steps (5) and (6) both take constant time as all the formulas require a constant time to evaluate and require values of variables that have already been calculated in the previous iterations. Hence the algorithm runs in $O(mn)$ time.