Final Exam, Algorithms 2019-2020

- 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 / 23 points	Problem 2 / 16 points	Problem 3 / 18 points	Problem 4 / 16 points	Problem 5 / 17 points	Problem 6 / 10 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)* Arrange the following functions in increasing order according to asymptotic growth.

A $\log(n!)$

B $(\log n)^{\log n}$

C 2^n

D The worst case running time of BUILD-MAX-HEAP on an array of size n.

E The worst case running time of Strassen's algorithm for multiplying two $n \times n$ matrices.

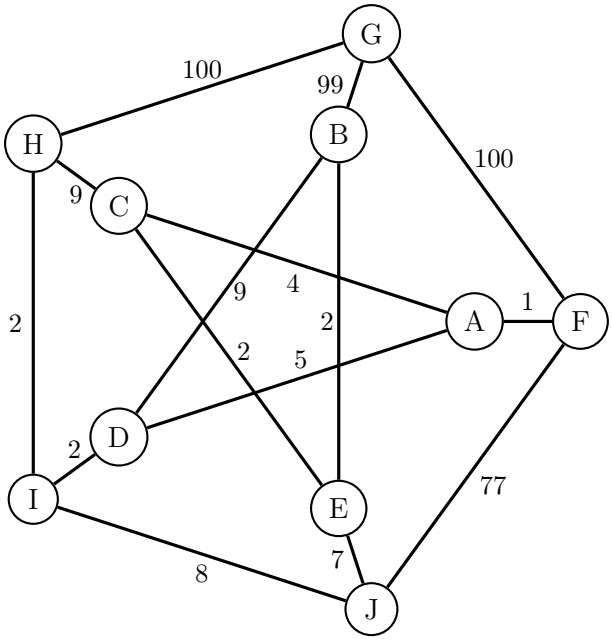
F The worst case running time of QUICKSORT on an array of size n.

Solution:

$$D < A < F < E < B < C.$$

Explanation. Note the following: $A = O(n \log n)$, $B = O(2^{\log \log n \cdot \log n}) = O(n^{\log \log n})$, $D = O(n)$, $E = O(n^{\log 7})$, and $F = O(n^2)$.

1b *(4 pts)* In the graph shown in the figure below, what order are the nodes reached in if we run Prim's algorithm from A?

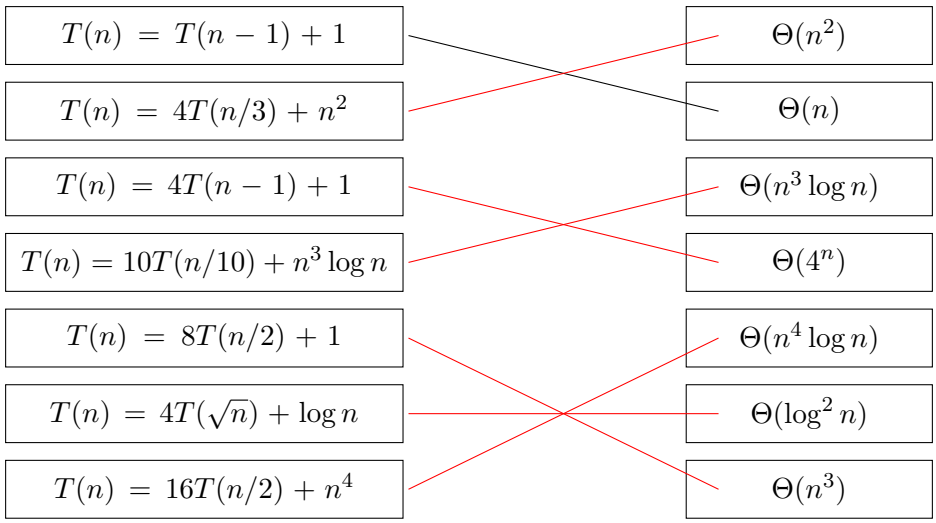


Solution:

$$A, F, C, E, B, D, I, H, J, G.$$

1c *(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:



1d *(4 pts)* Consider the recurrence $T(n) = 2T(\alpha n) + 5T(\beta n) + n^2$, $T(1) = \Theta(1)$, for some constants $\alpha, \beta \in (0, 1)$ that satisfy $2\alpha + 5\beta = 1$. Give a tight asymptotic bound for $T(n)$. You do not need to motivate your answer.

Solution:

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

1e *(4 pts)* Consider the recurrence $T(n) = 2T(\alpha n) + 5T(\beta n) + n^2$, $T(1) = \Theta(1)$, for some constants $\alpha, \beta \in (0, 1)$ that satisfy $2\alpha^2 + 5\beta^2 = 1$. Give a tight asymptotic bound for $T(n)$. You do not need to motivate your answer.

Solution:

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

2 *(16 pts)* **Binary trees.** In this problem you are given a rooted binary tree $T = (V, E)$ with n nodes, and your task is to design an efficient algorithm for finding a pair of nodes $a, b \in V$ that are at the largest possible distance from each other in T (the distance between a and b in T is the length of the path connecting a to b in T that does not have repeated vertices). For full credit your algorithm should run in $O(n)$ time.

- **Input:** A rooted binary tree $T = (V, E)$.
- **Output:** A pair of nodes $a, b \in V$ at maximum possible distance from each other in T.

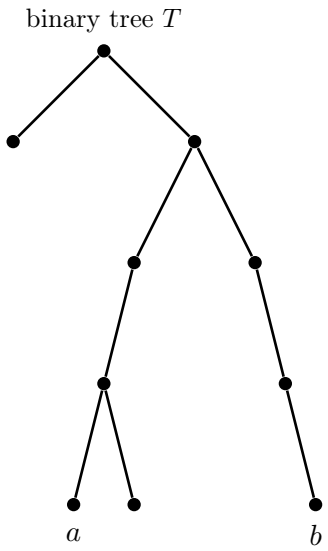


Figure 1. A binary tree T with a pair of nodes a, b at maximum possible distance in T.

Design an algorithm for this problem and analyze its runtime.

Solution:

We can solve this problem in time $O(n)$ by doing post-order traversal on the tree. First, we recursively compute the height of every node that has a value equal to the maximum of the height of the left subtree and the right subtree plus one. Then for every subtree, we compute the maximum length path that passes through the root of the subtree. The length of this path has a value equal to the sum of the height of its left subtree and right subtree plus two. Finally, the diameter of the tree is the maximum among all maximum length for all subtrees.

Algorithm 1 HEIGHT($node$)

```
1: if node = null then
2:     return 0
3: else if node.left = null and node.left = null then
4:     return 0
5: else if node.left = null then
6:     diam = max(diam, HEIGHT(node.right) + 1)
7:     return HEIGHT(node.right) + 1)
8: else if node.right = null then
9:     diam = max(diam, HEIGHT(node.left) + 1)
10:    return HEIGHT(node.left) + 1)
11: else
12:    diam = max(diam, HEIGHT(node.left) + HEIGHT(node.right) + 2)
13:    return max(HEIGHT(node.left), HEIGHT(node.right)) + 1
14: end if
```

Solution 2: First, we run BFS starting at the root to find a vertex x with the maximum distance from the root. Then we run the second BFS starting at vertex x to find the vertex y with the maximum distance from x. Then the pair (x, y) has the maximum distance among all the pairs in the tree. The runtime of the solution is the same as the runtime of the BFS algorithm i.e., $O(n)$.

Continuation of the solution to 2:

3 *(18 pts)* **Fair partitioning.** In this problem you are given n positive integers $(w_1, \ldots, w_n)$ with total sum $W = \sum_{i=1}^{n} w_i$. Your task is to determine whether it is possible to partition these integers into three disjoint subsets such that the sum of elements in every subset is exactly $W/3$.

- **Input:** An integer n and a sequence of positive integers $(w_1, \ldots, w_n)$ whose total sum is $W = \sum_{i=1}^{n} w_i$.
- **Output:** `YES` if it is possible to find a partition I_1, I_2, I_3 of the integers between 1 and n such that for every $j = 1, 2, 3$ one has $\sum_{i \in I_j} w_i = W/3$, and `NO` otherwise.

For example, suppose that the input sequence is $w_1 = 5, w_2 = 6, w_3 = 7, w_4 = 1, w_5 = 1, w_6 = 1$. Then such a partitioning exists: we let $I_1 = \{1, 4, 5\}, I_2 = \{2, 6\}, I_3 = \{3\}$. On the other hand, if the input sequence is $w_1 = 5, w_2 = 3, w_3 = 7, w_4 = 8, w_5 = 4, w_6 = 3$, then no such partitioning exists.

Design and analyze a dynamic programming solution to the problem with runtime polynomial in n and W.

3a For $i \in \{1, 2, \ldots, n\}$ and two integers $x, y \in \{0, 1, \ldots, W\}$ let $d(i, x, y) = 1$ if it is possible to partition the integers $w_1, w_2, \ldots, w_i$ (the first i integers in our sequence) into three sets whose sums are x, y and $\sum_{j=1}^{i} w_j - (x+y)$ respectively, and $d(i, x, y) = 0$ otherwise. Design a recurrence relation for $d(i, x, y)$.

Solution: The recurrence is as follows:

$$d(i, x, y) = \max\{d(i-1, x, y), d(i-1, x-w_i, y), d(i-1, x, y-w_i)\}$$

3b Analyze the runtime of a bottom up implementation of your recurrence in **3a**.

Solution:

Algorithm 2 Bottom up implementation of the recurrence

```
for i ∈ {0, 1, 2, ..., n} do
    for x ∈ {−W, ..., W/3} do
        for y ∈ {−W, ..., W/3} do
            if i = 0, x = 0, y = 0 then
                d(i, x, y) ← 1
            else if i = 0 then
                d(i, x, y) ← 0
            else if x < 0 or y < 0 then
                d(i, x, y) ← 0
            else
                d(i, x, y) ← max{d(i − 1, x, y), d(i − 1, x − w_i, y), d(i − 1, x, y − w_i)}
            end if
        end for
    end for
end for
return d(n, W/3, W/3)
```

One can see that the for loops of this algorithm incur $O(nW^2)$ run-time.

4 *(16 pts)* **Connectivity.** In this problem your task is to design cost effective infrastructure for a country consisting of n cities represented by vertices of a graph $G = (V, E)$. For every pair of cities $a, b \in V$ such that $(a, b) \in E$ you can build a road connecting the cities at cost w_{ab}. Also, for every city $a \in V$ you can build an airport at a at cost c_a. Any two cities that have an airport can reach each other. Your task is to determine which roads should be built and which airports should be constructed to ensure connectivity while minimizing cost.

Input: Connected undirected graph $G = (V, E)$ with non-negative edge weights $w : E \to \mathbb{R}_+$, for every $a \in V$ the cost c_a of building an airport at a.

Output: Minimum cost collection of roads and/or airports to build that ensures connectivity of all cities in V.

Solution: There are two possible ways of constructing the infrastructure:

- without any airports,
- with a non-zero number of airports.

If we commit to constructing no airports the problem is simply to construct a minimum cost spanning subgraph on a weighted undirected graph. Since the weights are non-negative, the optimal solution can be a spanning tree and we can simply use Prim's or Kruskal's to find the minimum cost spanning tree of G.

If we commit to constructing some airports, the problem becomes equivalent to the following: Add to G a single new vertex v_0, connected to every other vertex, with (a, v_0) having weight c_a. We can now think of building airports as building connections to v_0.

Indeed, every solution for connecting the country, which includes at least one airport, defines a spanning subgraph of the graph $G + v_0$. All pairs of vertices in V are connected (possibly through v_0 if we need to use airports), and v_0 is also connected to V, since we have built at least one airport. Similarly every spanning subgraph of $G+v_0$ defines a correct solution for connecting the cities. Again, since all edge weights are non-negative, the optimal solution can be a spanning tree which we can find using Prim's or Kruskal's.

The full algorithm runs Prim's or Kruskal's on both G and $G + v_0$ and takes the better of the two results.

Continuation of the solution to 4:

5 *(17 pts)* **Path finding.** In this problem you are given an undirected graph $G = (V, E)$ and three distinct vertices $x, y, z \in V$. Your task is to find a simple path from x to y in G that goes through the vertex z if such a path exists, and output `NONE` otherwise. Recall that a path is simple if it does not have repeated vertices. See Fig. 2 below for an example.

Input: An undirected graph $G = (V, E)$ together with three distinct vertices $x, y, z \in V$.

Output: A simple path from x to y in G that goes through z if such a path exists, and `NONE` otherwise.

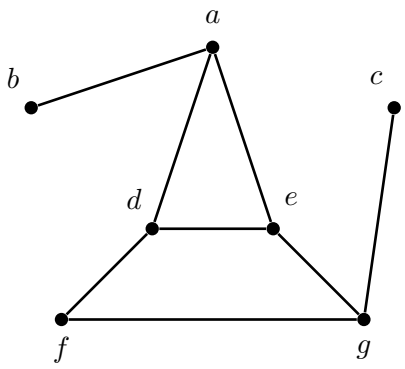


Figure 2. In the graph G above there exists a simple path from b to c that goes through e, but there does not exist a simple path from a to f that goes through b.

Design and analyze an algorithm for the problem. Your algorithm should run in time polynomial in $|V|$. *Hint: modify the graph and use max-flow.*

In this problem you are required to (a) describe an efficient algorithm and (b) analyze its runtime.

Solution: We basically need to find out if there exist vertex disjoint paths from z to x and y. This can be done by computing max-flow on an appropriate graph.
We first convert $G = (V, E)$ to a directed graph $G' = (V', E')$. For every vertex $v \in V$, we add a vertex to V'. For every edge $e = (a, b) \in E$, we add two vertices $v_{(a,b)}$ and $v_{(b,a)}$ to V' and edges $(a, v_{(a,b)}), (v_{(a,b)}, b), (b, v_{(b,a)})$ and $(v_{(b,a)}, a)$ to E'.
We now convert G' to a graph with edge capacities $G'' = (V'', E'')$. We add a source vertex v_{source} to V''. In order to simulate the vertices having capacities, for every vertex $v \in V \setminus \{z\}$, we add v, v_{in} and v_{out} to V''. We add edges (v_{in}, v) and (v, v_{out}) each of capacity 1 to E''. For simplicity of notation, let $v_{a,b} = (v_{a,b})_{in} = (v_{a,b})_{out}$ for any edge (a, b) in E and $z = z_{in} = z_{out}$. For every edge $(a, b) \in E'$, we add edges (a_{out}, b_{in}) of capacity 1. We add edges (v_{source}, x_{in}) and (v_{source}, y_{in}) of capacity 1.

It is not hard to see that testing whether there exists a simple path from x to y in G that goes through z is equivalent to testing if there exists a max-flow of value 2 on G''. We will show how to find a path in G if value of max-flow on G'' is 2 in the algorithm below. For the other direction, let there exist a path P from x to y that goes through z. Send a flow of value 1 on edges (v_{source}, x_{in}), (x_{in}, x) and (x, x_{out}). Similarly, send a flow of value 1 from v_{source} towards y_{out}. For an edge (a, b) that is on path P before vertex z is visited, add flow of 1 on edges $(a_{in}, a), (a, a_{out}), (a_{out}, v_{a,b}), (v_{a,b}, b_{in}), (b_{in}, b)$ and (b, b_{out}). For an edge (a, b) that is on path P

after vertex z is visited, add flow of 1 on edges $(b_{in}, b), (b, b_{out}), (b_{out}, v_{b,a}), (v_{b,a}, a_{in}), (a_{in}, a)$ and (a, a_{out}). The flow constraints are satisfied as the path P is simple.
For finding max-flow on G'', one can run the Ford Fulkerson method with breadth first search to find augmenting paths. If max-flow has value 2, one can run a breadth first search from vertex z on the residual graph with edges of capacity 1. Then output $rev(path(z, a) \circ path(z, b))$ where $rev(s)$ is used to represent the reverse of string s and $\circ$ is the concatenation operator. One should also remove unwanted vertices from the path found such as v_{in} and v_{out} vertices for a vertex $v \in V'$ and $v_{(a,b)}$ and $v_{(b,a)}$ for an edge $(a, b) \in E$.
For runtime analysis, building graphs G' and G'' takes $O(V + E)$ time. Running Ford Fulkerson method with bfs takes $2 \cdot O(V + E)$ as max-flow has value atmost 2. Finally, finding the path requires doing a bfs which takes time $O(V + E)$. Hence the total time taken by our algorithm is $O(V + E)$.

6 *(10 pts)* **Searching in a 2d array.** In this problem you are given an $n \times n$ array whose columns and rows contain numbers in increasing order, and a number x (see example below). Your task is to design an algorithm that locates x in the array or concludes that it is not there. For full credit your algorithm should run in $O(n)$ time.

1	2	3	5	7
3	3	4	5	8
3	4	5	5	8
4	5	6	8	10
6	7	7	9	11

Figure 3. An example 5×5 array A whose rows and columns contain numbers in increasing order.

- **Input:** Integers n and x, as well as an $n \times n$ array A of integers such that all rows of A are monotone increasing, and all columns are monotone increasing.
- **Output:** Integers $1 \leq i, j \leq n$ such that $A_{i,j} = x$ if x is in A, and $\perp$ otherwise.

Design an algorithm for this problem and analyze its runtime.

Solution: Consider the following algorithm:

$i \leftarrow n$, $j \leftarrow 1$
while $i \geq 1$ and $j \leq n$ **do**
  if $A_{i,j} = x$ **then**
    return i, j
  else if $A_{i,j} < x$ **then**
    $j \leftarrow j + 1$
  else
    $i \leftarrow i - 1$
  end if
end while
return $\perp$

In other words, we start in the bottom left corner of the array and compare this value to x. If it is x, return the coordinates. If it is smaller than x, then we know that x cannot be in the first column of the array, so we move one entry to the right. Otherwise, if it is larger than x, we know that x cannot be in the last row of the array, so we move one entry up.

Repeat this process until we either find x, or exit the boundary of the array. If we exit the array then we know that x is not in the array because we maintain that all entries to the left and below the current position cannot contain x.

Running time: In each iteration of the while loop we move one entry up or one entry to the right so there will be at most $2n - 1$ iterations, and a constant number of operations are performed in each iteration. Hence, the algorithm is linear $O(n)$ time. (Another way to think of it is that in each iteration we rule out one column or one row that cannot contain x.)