Tutorial 3

Exercise 1: Let $a, b > 1$. Show that:

- $a^{\log_b n} = n^{\log_b a}$ (hint: apply the function $\log_b$ to both sides of the equation)
- $\exists c : \forall x : \log_a x = c \cdot \log_b x$ (which implies $\log_a n \in \Theta(\log_b n)$)

Exercise 2: Recall the precise meaning of the notation $\mathcal{O}$, Ω, Θ, o, ω and define it formally. Then order the following functions according how fast they grow and determine, which of the relationships of the form $f \in \mathcal{O}(g)$, $f \in \Omega(g)$, $f \in \Theta(g)$, $f \in o(g)$, and $f \in \omega(g)$, hold for the given functions and which do not.

a) n
b) n^2
c) n^3
d) $\sqrt{n} \cdot 3n$
e) $n^2 \log_2 n$
f) $\log_2 n$
g) $\log_{10} n$
h) $(\log_2 n)^2$
i) 2^n
j) n^n
k) $n^{\log_2 n}$
l) $(\log_2 n)^n$
m) $2^{\sqrt{n}}$
n) 2^{2^n}
o) $2^{2^{n+1}}$
p) $\begin{cases} n^2 & \text{if } n \text{ is odd} \\ 2^n & \text{if } n \text{ is even} \end{cases}$

Exercise 3: Describe some polynomial algorithm solving the following problem:

INPUT: A directed graph $G = (V, E)$ and a pair of nodes $s, t \in V$.

OUTPUT: A shortest path from node s to node t, or information that there is no such path.

Remark: There can be more than one shortest path in the graph. The output can be any of these shortest paths.

Write down the algorithm that solves this problem in a form of a pseudocode, and analyze its time and space complexity as precisely as possible.

Exercise 4: Recall that a ***walk*** is an undirected graph $G = (V, E)$ is a sequence of nodes and edges

$$v_0, e_1, v_1, e_2, v_2, \ldots, v_{r-1}, e_r, v_r,$$

where $v_0, \ldots, v_r \in V$ and $e_1, \ldots, e_r \in E$, where each e_i (where $1 \leq i \leq r$) is as edge from node v_{i-1} to node v_i. (Both nodes and edges can repeat in this sequence.)

A ***trail*** is a special case of a walk where nodes can repeat but edges should not repeat. (I.e., all edges $e_1, \ldots, e_r$ in this sequence are distinct.)

A trail is called ***Eulerian*** if it contains all edges of graph G. A name ***Eulerian path*** is often used instead of ***Eulerian trail***, even if it is in fact a trail, not a path. So an Eulerian path describes a way how to go through the graph in such a way that each edge is visited exactly once.

Consider the following problem:

NAME: EULER-PATH

INPUT: An undirected graph $G = (V, E)$.

OUTPUT: An Eulerian path going through all edges of the graph G, or information that there is no such path.

(*Remark:* There can exist more than one Eulerian path in a given graph G. The output can be any of them.)

a) Show that the problem EULER-PATH is algorithmically solvable. What is the computational complexity of the proposed algorithm?

b) Show that there is a polynomial algorithm solving the problem EULER-PATH.

Exercise 5: Consider the problem of finding the ***maximum flow*** throught a network (MAX-FLOW).

A ***network*** here means a directed graph $G = (V, E)$where every edge $(u, v) \in E$ has an assigned ***capacity*** $c(u, v)$, which is an non-negative number. For simplicity, we can consider it in such a way that if there is no edge from one node to another, it is the same as if there would be an edge with capacity 0, i.e., if $(u, v) \notin E$, then $c(u, v) = 0$. So formally, the capacities can be defined by a function $c : V \times V \to \mathbb{R}_{\geq 0}$. There are two distinguished nodes s (***source***) and t (***sink***) in the network.

A ***flow*** in a network G with capacities of edges c and with source t and sing t, is a function $f : V \times V \to \mathbb{R}$ satisfying two following conditions:

- The flow flows only through edge of the given graph, and a capacity of any of the edges is exceeded, i.e., for each $u, v \in V$ it holds
$$0 \leq f(u, v) \leq c(u, v)$$
(If $(u, v) \notin E$, then $c(u, v) = 0$, and so necessarily $f(u, v) = 0$.)

- For each node with exception of the source and the sink, it holds that what flows in into the node, it also flows out of this node, i.e., for each $u \in (V - \{s, t\})$ it holds that
$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

The ***value*** of a flow f, denoted as $|f|$, is the sum of what flows out of the source s, minus what flows in, i.e.,
$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

A flow f is ***maximum*** if for each other flow f' we have $|f| \geq |f'|$.

The problem MAX-FLOW is defined as follows:

NAME: MAX-FLOW

INPUT: A network $G = (V, E)$ with capacities of edges $c : V \times V \to \mathbb{R}_{\geq 0}$, source s and sink t (where $s, t \in V$).

OUTPUT: A maximum flow f in the given network.

(*Remark:* There can be more than one maximum flow in a given network. The output can be any of them.)

For simplicity you can assume that the capacities of edges are given as natural numbers.

a) Show that the MAX-FLOW problem is algorithmically solvable, i.e., describe an algorithm solving this problem.

Hint: For a given network G with capacities of edges c and a given flow f, we can compute ***residual network*** G' with capacities of edges c' that express how much the flow through each edge of the original network can be changed in one or the other direction (i.e., how much it can be increased or decreased) without exceeding capacities of edges. It can be shown that a given flow f is maximum iff the residual network corresponding to this flow contains no path from s to t.

b) Show that if we would use a simple algorithm, based on the above idea, where we would just check whether there is an augmenting path from s to t, and it is the case, the flow would be increased along this path, and we would repeat this step as long as there would be some augmenting path, then there could be some bad choice of these augmenting paths that could lead to the number of iterations that could correspond the ***value*** of the maximum flow, which is a value that could be much bigger than the number of nodes and edges of the given network.

c) Propose a modification of the approach described above that will ensure that the number of iterations is polynomial with respect to the number of nodes and edges of a given network, and so it is bounded from above by some value that does not depend on particular values of capacities of edges.

What is the computation complexity of the proposed algorithm?

Exercise 6: Consider the problem of finding a ***maximum matching in a bipartite graph*** (i.e., a matching with maximal number of edges).

A ***bipartite graph*** $G = (U, V, E)$ consists of two disjoint sets of nodes $U = \{u_1, \dots, u_n\}$ and $V = \{v_1, \dots, v_m\}$ (where $U \cap V = \emptyset$) and a set of edges $E \subseteq U \times V$. A ***matching*** M in graph G is a subset of edges $M \subseteq E$where for every two different edges $(u, v), (u', v') \in M$ we have $u \neq u'$ and $v \neq v'$. A maximum matching M is a matching where for each other matching M' we have $|M'| \leq |M|$.

NAME: Maximum matching in a bipartite graph

INPUT: A bipartite graph $G = (U, V, E)$.

OUTPUT: A maximum matching in the graph G.

(*Remark:* There can be more than one maximum matching in a graph G. The output can be any of them.)

Propose a polynomial algorithm for this problem.

Hint: This problem can be reduced to the problem of finding maximum network flow.

Exercise 7: Recall the ***travelling salesmen problem*** (***TSP***).

An instance of this problem is a set of n cities $V = \{1, \ldots, n\}$ A distances for every pair of cities. Let $d(i, j)$ denote the distance from city i to city j. We can assume that for every $i \in V$ is $d(i, i) = 0$ and that for each pair $i, j \in V$ we have $d(i, j) = d(j, i)$.

The goal is to find a closed path that goes through each city exactly once and ends in the same city where it started, such that the total length of this path will be the smallest possible between all such closed paths.

It is obvious that such path can start and end in city 1. So we can restrict our attantion to this case. In this variant of the problem, it is not allowed to visit cities repeatedly (with the exception of city 1, which is viseted twice — at the beginning and at the end). So the goal is to find a permutation $(i_1, \ldots, i_n)$ of cities $\{1, \ldots, n\}$ where $i_1 = 1$ and where the sum

$$d(i_1, i_2) + d(i_2, i_3) + \cdots + d(i_{n-1}, i_n) + d(i_n, i_1)$$

is as small as possible.

a) Propose an algorithm solving this problem. What is the time and space complexity of your algorithm?

b) Design an algorithm based on the following idea:

   For each subset $S \subseteq (V - \{1\})$ and each $j \in S$, we can define $c[S, j]$ as the length of a shortest path that starts in city 1, visits all cities from S exactly once (and does not visit any other cities), and ends in city j.

   All values $c[S, j]$ can be computed using ***dynamic programming*** where these values can be computed in the order determined by the size of set S (from the smallest to the biggest), where for the computing of values for bigger sets we can use already computed values for smaller sets.

   Show how this algorithm can be implemented in such a way that its time complexity is $\mathcal{O}(n^2 2^n)$. What is the space complexity of this algorithm?

   Compare the time complexity and the space complexity of this algorithm and the previous algorithm your have proposed. Which of them is better?

Exercise 8: Use the master theorem to derive the time complexity of recursive algorithms, for which heir running times are given by the following recurrences:

a) $T(n) = 3T(n/2) + n$

b) $T(n) = 4T(n/2) + n^2$

c) $T(n) = 5T(n/2) + n^3$