Problem 1

Given $n$ numbers $a_1, \ldots, a_n$ find indices $i$ and $j$, $1 \leq i \leq j \leq n$, such that $\sum_{k=i}^j a_k$ is minimized.

We will develop two algorithms for this problem that run in linear time, i.e., the number of (arithmetic) operations is linear in $n$.

(a) Solve the problem using Bellman-Ford as a subroutine. In particular, construct a graph such that a shortest path in this graph yields the optimal solution to the above problem. Show that the graph can be generated in linear time and that Bellman-Ford can be implemented to run in linear time on this graph.

(b) Define $d(j) = \min_{1 \leq i \leq j} \sum_{k=i}^j a_k$. Conclude that the above problem is equivalent to computing $\min_{1 \leq j \leq n} d(j)$. Show how this can be done in linear time.

Solution:

(a) Consider the graph in Figure 1(a). The shortest path between $s$ and $t$ gives the respective result. Clearly, this graph is acyclic and it has $O(n)$ edges so by Problem 4, one can compute the shortest $s - t$ path in $O(n)$ operations.

If there is at least one negative number, we can also consider the simpler graph in Figure 1(b). Note that instances with only non-negative numbers are not very interesting. Then, the solution is $i = j$ where $a_i$ is the smallest number.

(b) It is easy to see that $d(j + 1) = \min \{d(j) + a_{j+1}, a_{j+1}\}$. Setting $d(1) = a_1$ we can thus compute $d(2), \ldots, d(n)$ subsequently. Each successor computation takes constant time, hence the total computation takes $O(n)$ time.

The optimal pair $(i, j)$ sums the numbers from $a_i$ to $a_j$. Hence, this sequence ends at $j$ and the value of $d(j) = \sum_{k=i}^j a_k$. Thus, after computing the values of $d$ we can select the minimum value of the values in $d$ which yields the optimal value. Again, this can be done in linear time.

Problem 2

Due to the decentralized nature of the global currency market, it might be the case that an individual or an organized group makes a large profit without risk. Arbitrage is a phenomenon that refers
to cases when it is possible to convert one unit of a currency into more than one unit of the same currency by using discrepancy in exchange rates. For example, consider the case that 1 CHF buys 60 RUB, 1 RUB buys 0.019 USD and 1 USD buys 0.93 CHF. This means that a trader can transform 1 CHF into $60 \cdot 0.019 \cdot 0.93 = 1.0602$ CHF gaining a profit of 6.02%.

Given a list of currencies $r_1, \ldots, r_n$ and a matrix $E \in \mathbb{R}_{>0}^{n \times n}$ where $E_{i,j}$ specifies the exchange rate between currencies $r_i$ and $r_j$, design a polynomial time algorithm to test if there is a possibility of arbitrage. While modelling the problem, bear in mind that testing if a weighted directed graph has a negative cycle can be done in polynomial time.

**Solution:**

Observe that in the case of arbitrage one makes a circular sequence of trades $t \in T$. The profit has been obtained in the case when the product of the corresponding exchange rates

$$\prod_{t \in T} E_{i(t),j(t)} > 1 \iff \prod_{t \in T} \frac{1}{E_{i(t),j(t)}} < 1 \iff \sum_{t \in T} \log \left( \frac{1}{E_{i(t),j(t)}} \right) < 0,$$

where $i(t)$ indicates the index of the currency sold in transaction $t$ and $j(t)$ is the index of the currency bought.

The above statement suggests the following natural construction. Let $G = (V,E)$ be a complete directed graph with the set of nodes $V = r_1, \ldots, r_n$. For each arc $(r_i, r_j) \in E$ set the weight $w_{i,j} = \log \left( \frac{1}{E_{i,j}} \right)$. The (possibly empty) set of arbitrages is in one-to-one correspondence with the set of negative-weight directed cycles on $G$. Existence of a negative cycle can be detected by running $n$ iterations of the Bellman-Ford algorithm. Such a cycle exists if and only if the distance matrix gets updated in the $n$th iteration as mentioned in class. It can be reconstructed by using the list of predecessors maintained by the algorithm.

**Problem 3**

Let $D = (V,A)$ be a directed graph, $w : A \to \mathbb{R}$ be arc weights and $s \in V$. Suppose that there exists a path from $s$ to each other node of $V$.

Consider the following linear program:

$$\begin{align*}
\max & \quad \sum_{v \in V \setminus \{s\}} x_v \\
\text{s.t.} & \quad x_v - x_u \leq w(u,v), \quad \forall (u,v) \in A \\
& \quad x_s \leq 0.
\end{align*}$$

(1)

Show the following:

a) This LP is feasible if and only if $D$ has no negative cycle;

b) If $D$ has no negative cycle, then (1) has a unique optimal solution.

**Solution:**

For each $v \in V$, denote with $d(s,v)$ the length of the shortest path from $s$ to $v$ in $D$, subject to $w$.

a) ($\Leftarrow$) If there are no negative cycles in $D$, then we have for each $(u,v) \in A$:

$$d(s,v) \leq d(s,u) + w(u,v) \Leftrightarrow d(s,v) - d(s,u) \leq w(u,v).$$

Thus, the assignment $x_v^* = d(s,v), \forall v \in V$ with $x_s^* = 0$ is feasible for (1).

($\Rightarrow$) Let $\bar{x}$ be a feasible solution to (1) and let $C$ be a directed cycle in $D$, denote its length with $w(C)$. One has:

$$w(C) = \sum_{(u,v) \in C} w(u,v) \geq \sum_{(u,v) \in C} (\bar{x}_v - \bar{x}_u) = 0.$$
The last equality comes from the fact that every vertex contained in $C$ arises in the summation exactly twice, once with the positive and once with the negative sign.

b) Consider an arbitrary vertex $v \in V$ and let $s = u_0 \to u_1 \to \cdots \to u_k = v$ be a shortest length path from $s$ to $v$, subject to $w$. By adding up the constraints $x_{u_1} - x_k \leq w(s, u_1)$ and $x_s \leq 0$ we obtain that $x_{u_1} \leq w(s, u_1)$ is valid for (1). Further adding $x_{u_2} - x_{u_1} \leq w(u_1, u_2)$ gives $x_{u_2} \leq w(s, u_1) + w(u_1, u_2)$. If we continue the process in the same manner, then we have that

$$x_v \leq \sum_{i=1}^{k} w(u_{i-1}, u_i) = d(s, v)$$

For each $v \in V$, this inequality has to be satisfied by every feasible solution of (1). Thus, the maximum of $\sum_{v \in V \setminus \{s\}} x_v$ is uniquely attained by the solution

$$x_v^* = d(s, v), \quad \forall v \in V \setminus \{s\}.$$ 

Note that, if there is no negative cycle in $D$, there has to be an (outgoing) arc $(s, v) \in A$ such that $d(s, v) = w(s, v)$. Therefore, $x_v^* \geq x_v^* - w(s, v) = 0$, and by $x_v^* \leq 0$ one gets $x_v^* = 0$. As seen in part (a), the solution $x^*$ is feasible for (1).

Problem 4

Design an algorithm that, a directed graph $G = (V, A)$, finds the number of shortest paths from $s$ to $t$ in time $O(|V| + |A|)$.

Solution:

We will modify the breadth-first search algorithm in a way that it also counts the number of shortest paths from $s$ to any other vertex. This is done by inheriting the number of paths from the predecessor and updating this number if other paths of the same length are found.

More formally: keep the initialization for the queue $Q$, the distance array $D$ and the predecessor array $\pi$. Further, add an array $p$ storing the number of paths to each node and initialize $p = [1, 0, \ldots, 0]$.

while $Q \neq \emptyset$ do

$\quad u := \text{head}(Q)$

for each $v \in \delta^+(u)$ do

$\quad$ if $D[v] = \infty$ then

$\quad \quad \pi[v] := u$

$\quad \quad D[v] := D[u] + 1$

$\quad \quad p[v] := p[u]$

$\quad \quad \text{enqueue}(Q, v)$

$\quad$ else if $D[v] = D[u] + 1$ then

$\quad \quad p[v] := p[v] + p[u]$

$\quad \quad \text{dequeue}(Q)$

As seen in class, the algorithm finds all shortest paths from $s$. Since each shortest path from $s$ to $v$ goes through exactly one vertex in $\delta^-(v)$, the number of shortest paths from $s$ to $v$ is exactly the sum of the number of paths of length $D[v] - 1$ from $s$ to a vertex in $\delta^-(v)$. Note that there will never be a path of length $\leq D[v] - 1$ from $s$ to a vertex in $\delta^-(v)$, otherwise $D[v]$ would not be the length of the shortest path from $s$ to $v$. Thus for each $u \in \delta^-(v)$, the number of paths of length $D[v] - 1$ is either 0 or $p[v]$.

Since we add only a constant number of calculations in each iteration to the breath-first search algorithm the running time remains $O(|V| + |A|)$. 
Problem 5
A 2-matching in a graph is a collection of disjoint cycles that covers all the vertices. Show that a 2-matching can be computed in polynomial time, if such one exists. Note that it is allowed to pick an edge twice in a 2-matching, i.e., one can have a 2-cycle.

Hint: One may reduce the problem to finding a perfect matching in a bipartite graph.

Solution:
Given the initial graph $G(V, E)$, construct a bipartite graph $G' = (V \cup V', E')$, where $V'$ is a copy of $V$ and $E'$ has the edges $\{u, v'\}$ and $\{u', v\}$ for each edge $\{u, v\} \in E$.

Now, every 2-matching $M_2$ in $G$ corresponds to a perfect matching $M'$ in $G'$. For every cycle $v_1, \ldots, v_k$ in $M_2$ put the edges $(v_1, v'_2), \ldots, (v_{k-1}, v'_k), (v_k, v'_1)$ in $M'$. Clearly, $M'$ matches all the vertices $v_1, \ldots, v_k, v'_1, \ldots, v'_k$. Doing this for all cycles in matching $M_2$ of $G$ yields the corresponding perfect matching $M'$ of $G'$.

Conversely, if there is a perfect matching $M'$ in $G'$, we can construct a 2-matching $M_2$ in $G$. For each of edges $\{u, v'\}$ and $\{u', v\}$ in $M'$, we add the edge $\{u, v\}$ to $M_2$. A perfect matching in $G'$ can be computed, or it can be detected that there is no such matching, in polynomial time.