Problem 1

Subproblem a

Let us consider that there is a negative cycle C. Every edge has a weight of $w_{i,j} = r \cdot c_{i,j} - p_j$.

$$\Rightarrow \sum_{(i,j)\in C} r \cdot c_{i,j} - p_j < 0$$

$$\Rightarrow r \cdot \sum_{(i,j)\in C} c_{i,j} - \sum_{(i,j)\in C} p_j < 0$$

$$\Rightarrow r \cdot \sum_{(i,j)\in C} c_{i,j} < \sum_{(i,j)\in C} p_j$$

$$\Rightarrow r < \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{i,j}}$$

$$\Rightarrow \text{ found a cycle with a bigger ratio than } r$$

$$\Rightarrow r^* > r$$

Subproblem b

Let us consider that there is no negative cycle, i.e. all cycles are positive.

$$\Rightarrow \forall C \in \text{ cycles} : \sum_{(i,j)\in C} r \cdot c_{i,j} - p_j > 0$$

$$\Rightarrow \forall C \in \text{ cycles} : r > \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{i,j}}$$

$$\Rightarrow \max_{C \in \text{ cycles}} \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_(i,j)} < r$$

$$\Rightarrow \text{ even the greatest (value) cycle is smaller than } r$$

$$\Rightarrow r^* < r$$

Subproblem c

Basic Idea

- Find r^* with an accuracy of ϵ by doing binary search. Let r be the current guess.
- $R = \max_{(i,j)\in E} \frac{p_j}{c_{i,j}}$ is an upper bound for r^*, because using an additional edge can only make the ration bigger.
- Generate the weights for the graph with r.
- Run Bellman Ford to determine if there is a cycle with negative weight.
  - Run Bellman Ford.
  - If the shortest path to any vertex can be further reduced by taking an arbitrary edge, then the graph contains a negative cycle.
  - After running the outer loop n times, we know the minimal path length for every vertex.
  - After running the outer loop n more times, we can be sure that, for every vertex in a negative cycle, its predecessor points to the predecessor of the the vertex in the negative cycle.
  - Extract the negative cycle by following the predecessor pointer starting from the vertex the was most recently decreased.
- If a negative cycle is found, try a bigger value in the binary search, otherwise a smaller value.
- Runtime: $\mathcal{O}(\log \frac{R}{\epsilon})$ times runtime of Bellman Ford.
- By running Bellman Ford for every SCC, we can deal with non-connected graphs.

Pesudocode: General Algorithm

The input for the algorithm is a directed graph G. The values $c_{i,j}$ and p_j are considered part of the graph description G. The output is a cycle, i.e. a list of vertices.

```
$r \leftarrow R$
$left \leftarrow 0$
```

```
$C \leftarrow \emptyset$
while $r - \mathit{left} \leq \epsilon$ do
    $C \leftarrow \mathit{NegativeCycle}(G, r)$
    if $C = \emptyset$ then
        $r \leftarrow \frac{\mathit{left}+r}{2}$
    else
        $\mathit{left} = \frac{\mathit{left}+r}{2}$
    end if
end while
if $C = \emptyset$ then
    $r \leftarrow \max\{0, r - \epsilon\}$
    if $r = 0$ then
      return $\mathit{FindCycle}(G)$
    else
      return $\mathit{NegativeCycle}(G, r)$
    end if
end if
```

Pseudocode: Finding Negative Cycle

The basic idea of of the Bellman Ford algorithm was taken from Wikipedia. n is the number of vertices inside the current SCC. The input for the algorithm (*NegativeCycle*) is a directed graph G and a value r.

```
for all $(V, E) \in SCC(G)$ do
    $d[v] \leftarrow \infty \; \forall v \in V$
    $d[v_1] \leftarrow 0$
    $\mathit{foundNegCycle} \leftarrow \mathit{false}$
    for $k \leftarrow 1$ to $2n$ do
        for all $(i, j) \in E$ do
            $w_{i,j} \leftarrow rc_{i,j} - p_j$
            if $d[i] + w_{i,j} < d[j]$ then
                $d[j] \leftarrow d[i] + w_{i,j}$
                $\mathit{pre}[j] = i$
                $\mathit{lastDecrease} \leftarrow j$
                if $k > n$ then
                    $\mathit{foundNegCycle} \leftarrow \mathit{true}$
                end if
            end if
        end for
    end for
    if $foundNegCycle$ then
        $v \leftarrow pre[lastDecrease]$
        $L \leftarrow$ new List
        $L.add(lastDecrease)$
        while $v \neq lastDecrease$ do
            $L.add(v)$
            $v \leftarrow pre[v]$
        end while
     return $L$
    end if
end for
     return $\emptyset$
```

Proof

General Algorithm We use binary search in order to find an approximation of r^*. We know that $R = \max_{(i,j)\in C} \frac{p_j}{c_{i,j}}$ is an upper bound for r^*, because R's ratio is the maximum ratio among all edges. When we form a cycle by adding other edges to R's edge, this ratio cannot become bigger, since we only add edges whose ratio is equal or smaller. When we add an edge with a lower ratio, the overall ratio of the set of edges will become smaller.

The binary search ensures that we always take a look at smaller values of r if we did not find a negative cycle and bigger values of r otherwise. In subsections a and b, we showed that this is the correct way to get closer to r^*.

We can end the binary search, when the difference of the previous value of $r - \mathit{left} \leq \epsilon$. In this case, r^* is contained within an interval of size less than ϵ. If $C \neq \emptyset$, we found a negative cycle, therefore $r(C) < r^*$ (proof: subsection a) and $r^* - r(C) \leq \epsilon$. Therefore, $r(C) \geq r^* - \epsilon$.

In case $C = \emptyset$, we did not find a negative circle. Therefore, for every circle C, $r > r^*$, but $r - r^* \leq \epsilon$. Therefore, $r - \epsilon$ is a valid approximation of r^*, because $r^* - (r - \epsilon) \leq \epsilon$. In case, $r - \epsilon < 0$, $r = 0$ is a valid approximation. In that case, all profits are zero, $r^* = 0$, and every circle C is a valid result. In this case, we just have to find a circle in a directed graph.

Find Negative Cycle The starting vertex for the algorithm can be chosen arbitrarily, since we are not really interested in the real distance values of the vertices. We just want to find negative cycles. In case the graph is not connected, we have to run the algorithm for every strongly connected component (SCC). We can find SCCs using Tarjan's algorithm efficiently (see Wikipedia for description of the algorithm and its runtime). For every SCC, we can choose the starting point arbitrarily from the set of vertices of the SCC. By searching for a negative cycle in every SCC, we can be sure to find a negative cycle in the whole graph if one exists.

The first n runs of the outer loop (variable k), correspond to the standard Bellman Ford algorithm. We know that after that, each vertex's distance value is equal to the vertex's real distance from the starting point or maybe even smaller (in case there is a negative cycle). If there is a negative cycle in the current SCC, in every subsequent run of the outer loop, a distance value of a vertex inside a negative cycle is decreased. If we do not decrease a vertex distance value after the first n runs, we can be sure that there is not negative cycle. Otherwise, we would be able to decrease the distance values of the vertices inside the cycle by looping inside the cycle multiple times.

By running the loop n more times, we can be sure that we keep decreasing the distances values of vertices of negative cycles. After n runs, we can be sure that we decreased the distance value of every vertex inside every negative cycle. This is true, because we can always find a shorter path for a vertex v_i inside a negative cycle, if v_i's predecessors distance value was decreased. Therefore, the predecessor pointer of every vertex inside a negative cycle points to the actual predecessor in the negative cycle. In the worst case, the whole SCC is one big negative cycle. After n runs, all vertices were updated, because n is the number of vertices and in every run of the loop we updated at least one vertex inside a negative cycle.

Runtime

- Steps of the binary search starting from R with an accuracy of ϵ: $\mathcal{O}(\log R - \log \epsilon) = \mathcal{O}(\log \frac{R}{\epsilon})$
- Finding all SCCs with Tarjan's algorithm: $\mathcal{O}(|V| + |E|)$. $|V|^2$ is an upper bound for $|E|$.
- Finding a negative cycle inside an SCC with the modified Bellman Ford algorithm: the outer loop runs $2|V|$ times, and the inner loop iterates

over all edges. In addition, we have to reconstruct the loop by taking a look at the predecessors of the vertex that was decreased for the last time. This can be no more than $|V|$ vertices. $|V|^2$ is an upper bound for $|E|$ (in case there is an edge between every pair of vertices). $\mathcal{O}(2|V||E| + |V|) = \mathcal{O}(|V||E|) = \mathcal{O}(|V|^3)$.

- The total runtime is $\mathcal{O}(\log \frac{R}{\epsilon} \cdot |V|^2 \cdot |V|^3) = \mathcal{O}(\log \frac{R}{\epsilon}|V|^5$.

Problem 3

Basic Idea

- For a tree to have zero skew, its two subtrees must both have zero skew.
- The algorithm works bottom up: for every tree, run the algorithm recursively on its two subtrees. On the lowest level, a node has leaves a and b. Increase $\arg\min_{i \in \{a,b\}} l_i$, such that $l_a = l_b$.
- If the two subtrees A and B with the edges $e(A)$ and $e(B)$ pointing to them have zero skew, increase $\arg\min_{i \in \{A,B\}} h(i) + l_{e(i)}$, where $h(T)$ is the length of the root-to-tree path of tree T.
- In both cases, increase the edge by the absolute difference (between A and B) of the sum of the complete height of the subtree plus the length of the edge pointing to the subtree.

Pseudocode

The psuedocode shows the function *zeroSkew* whose input is the node T (root of the tree). We denote the child nodes of T with A and B. $l_{T,A}$ is the length of the edge between T and its child node A. We consider a node with only one child node to be zero skew. Such a child node must be a leaf, because the binary tree is complete.

```
if $T$ has no children then
    return 0
else if $T$ has only one child $A$ then
    return $zeroSkew(A) + l_{T_A}$
end if
$h_A \leftarrow zeroSkew(A)$
```

```
$h_B \leftarrow zeroSkew(B)$
$diff \leftarrow h_A + l_{T,A} - h_B - l_{T,B}$
**if** $diff < 0$ **then**
    $l_{T,A} \leftarrow l_{T,A} - diff$
**else if** $diff > 0$ **then**
    $l_{T,B} \leftarrow l_{T,B} + diff$
**end if**
        **return** $h_A + l_{T,A}$
```

Proof

We prove the correctness of the algorithm by induction. We prove that after running $zeroSkew(T)$, the tree T has zero skew.

- We notice, that for a tree T to have zero skew, both of its subtrees A and B must have zero skew, because the lengths of the edges above T affect the lengths of the paths to the leaves of both A and B in the same way.
- **Base Case:** If T is a leaf, i.e. it does not have any child nodes, the skew of T is zero. The length of the complete tree T is zero.
- **Induction Hypothesis:** Assume that *zeroSkew* reduces the skew to zero for trees A and B and returns the length of the complete tree A or B respectively.
- **Induction Step:** We prove that $zeroSkew(T)$ reduces T's skew to zero and returns T's complete lengths, where A and B are children of T. We know that A and B have zero skew and we know their complete length h_A and h_B. *diff* is the difference between the complete length plus the length of the connection edge of A and B. T has zero skew, iff $diff = 0$. By increasing the length of the connecting edge of the smaller subtree by the absolute value of *diff*, $h_A + l_{T,A}$ and $h_B + l_{T,B}$ become equal. Therefore, T has now zero skew. Also, notice that there is no better way (that involves adding a smaller number) of reducing T's skew to zero, since we are not allowed to reduce the length of an edge.
- The resulting tree has a minimal complete length, i.e. there is no other way of changing the lengths such that T has zero skew, since we are

only allowed to increase the length of a node and because every subtree must have zero skew. Therefore, the best way to reduce the skew to zero is to work bottom up and to always increase the length of at most one of T's subtrees.

Runtime

We used divide and conquer to divide the problem of reducing T's skew to zero into subproblems for each of the subtrees. Therefore, we created two subproblems and each subproblem can be solved in constant time, since we only compare two numbers amd add four numbers. In other words, we did a constant amount of computation for every node. Therefore, the runtime for the algorithm in $\mathcal{O}(n)$, where n is the number of nodes in the tree.

Problem 4

Basic Idea

- For all functions $f_e(x)$, generate a list of all pairwise intersection points x_i, resulting in a list of intervals when we sort the intersection points by x-values.
- For every interval (x_i, x_{i+1}), calculate the minimum spannning tree (Kruskal's algorithm or Prim's algorithm) for $x = \frac{x_i + x_{i+1}}{2}$, resulting in a list of edges $E_{i,i+1}$ for every interval. We denote this by the tupel $(x_i, x_{i+1}, E_{i,i+1})$.
- For every tupel $(x_i x_{i+1}, E_{i,i+1})$, calculate the minimum $m_{E_{i,i+1}}$ of the function $S_{E_{i,i+1}}(x) = \sum_{f \in E_{i,i+1}} f(x)$by calculating its first derivation.
- Output the smallest of all $m_{E_{i,i+1}}$.

Pseudocode

The algorithm receives a graph $G = (V, E)$ and a list of quadratic functions $f_e(t)$ for every $e \in E$ as an input.

```
$X \leftarrow$ new Set
**for all** $a \in E$ **do**
```

```
    for all $b \in E$ do
        $X$.add($intersection(a, b)$)
    end for
end for
sort($X$)
$X$.add($X$.first $- 10$)
$X$.add($X$.last $+ 10$)
$m_{best} \leftarrow \infty$
for all consequitive pairs $(x_i, x_{i+1}) \in X$ do
    $H \leftarrow MST(\frac{x_i + x_{i+1}}{2})$
    $M \leftarrow \sum_{f \in H} f(x)$
    $M' \leftarrow \frac{\partial M}{\partial x}$
    $m \leftarrow x$ where $M'(x) = 0$
    if $M(m) < m_{best}$ then
        $m_{best} \leftarrow M(m)$
        $x_{best} \leftarrow m$
    end if
end for
    return $x_{best}$
```

Proof

- Kruskal's algorithm and Prim's algorithm do not care about the exact values of the edges when they generate the minimum spanning tree. What matters is the relation of the edges to each other, e.g. if the weight of an edge is bigger than another one.
- When two functions $f_e(x)$ intersect, the weight of one edge gets bigger than the weight of the other edge. Therefore, when we take a look at values of x, Kruskal's/Prims's algorithm only decides to use a different set of edges when we cross an intersection point of two edge function.
- We generate a MST for every interval, i.e. for every possible combination of edge function relations. We only do this for retrieving the set of edges that are used for this interval. We can be sure that the minimum value of x is in one of the intervals, since they cover all numbers from $-\infty$ to ∞.

- For every interval, we find the extreme point of the sum of all functions of the chosen edges. Since all edge functions are quadratic functions, the sum of a set of edge functions is also a quadratic function. We can be sure that the optimum is always a minimum, because $a > 0$. One of the minimums must yield the smallest MST, because the set of edges is fixed in every interval.
- Note: the first and the last interval are open intervals, i.e. they have no boundary towards $-\infty$ and ∞. We added the intersection points $X.\text{first} - 10$ and $X.\text{last} + 10$ to have intervals for these edge cases. Note, that even these outer intervals have a minimum, i.e. the minimum can not be at $-\infty$ or ∞, because $a > 0$.

Runtime

- Generating all intersection points: $\mathcal{O}(|E|^2)$, resulting in $\mathcal{O}(|E|^2)$ intersection points and intervals.
- Sorting the intersection points: $\mathcal{O}(|E|^2 \log |E|^2)$.
- Generating the minimal spanning tree with Kruskal's algorithm: $\mathcal{O}(|E| \log |E|)$[1]. Generating the MST for all intervals requires $\mathcal{O}(|E|^3 \log |E|)$ time.
- Generating the sum of all selected edges, generating its derivative and finding its minimum takes $\mathcal{O}(|E|)$ time, because $|E|$ is the maximal number of functions that are summed up. For all intervals, this takes $\mathcal{O}(|E|^2)$ time.
- The overall runtime of the algorithm is bounded by $\mathcal{O}(|E|^4)$, i.e. it is polynomial in the the number of edges.

Problem 5

Basic Idea

- We define the function *splittable*(i, p_x, p_y) that determines whether the substring of s beginning at index i is splittable, assuming that we already saw the prefixes p_x of x and p_y of y.

[1]Source: Wikipedia

- $splittable = true \Leftrightarrow splittable(i+1, pre(x, p_x \circ s_i), p_y) \vee splittable(i+1, p_x, pre(y, p_y \circ s_i))$, with $pre(a, p_a) = \epsilon$ **if** $a = p_a$ **else** p_a[2] and $\circ$ is the concatenation of strings.
- We fill the table for *splittable* using dynamic programming, beginning with the maximum value of i (from the back of the sequence). The base cases are $splittable(n+1, p_x, p_y) = true$, for all prefixes p_x of x and all prefixes p_y of y and where $n = |s|$.
- We extract the solution to the problem (whether the string is splittable or not, an exact splitting is not requested) at $splittable(1, \epsilon, \epsilon)$.

Pseudocode

```
$splittable(n+1, p_x, p_y) \leftarrow true\ \forall p_x \in \text{prefixes}(x) \forall p_y \in \text{prefixes}(y)$
for $i \leftarrow n$ downto 1 do
    for all $p_x \in \text{prefixes}(X)$ do
        for all $p_y \in \text{prefixes}(Y)$ do
            $isSplittable \leftarrow false$
            if $p_x \circ s_i \in \text{prefixes}(x) \wedge splittable(i+1, pre(x, p_x \circ s_i), p_y)$ then
                $isSplittable \leftarrow true$
            end if
            if $p_y \circ s_i \in \text{prefixes}(y) \wedge splittable(i+1, p_x, pre(y, p_y \circ s_i))$ then
                $isSplittable \leftarrow true$
            end if
        end for
    end for
end for
    return $splittable(1, \epsilon, \epsilon)$
```

Proof

We proove by induction that $splittable(i, p_x, p_y)$ contains true iff the substring beginning at i is splittable, given that $splittable(i+1, \ldots, \ldots)$ contain the correct value.

- *Base Case:* For $i = n+1$, the substring is empty. An empty substring is always splittable regardless of the already seen prefixes, because a

[2]$pre(a, p_a)$ returns ϵ if the prefix p_a is the whole word a.

sequence of x or y does not necessarily have to be completed. Therefore, for all valid prefixes of x and y, an empty string is considered splittable.

- *Induction Hypothesis:* Let us assume that we know whether a substring beginning at index $i + 1$ is splittable, for all possbile combinations of valid prefixes of x and y.
- *Induction Step:* Let us assume that the current prefix of x, i.e. the characters of x that we already saw, is p_x and that the current prefix of y is p_y. Now we consider two cases.
  - $p_x \circ s_i$ is a prefix of x or equals x, and $\mathit{splittable}(i + 1, \mathit{pre}(x, p_x \circ s_i), p_y) = \mathit{true}$. In that case, we know that the current sequence of x, that we read so far, can be continued by reading the next character from the sequence string s_i. We also know, that the rest of the sequence string is splittable, considering the new prefix $\mathit{pre}(x, p_x \circ s_i)$ and the unchanged prefix p_y. Therefore, the string beginning at index i is splittable with the current prefixes p_x and p_y.
  - The same argument can be made for y and p_y, meaning that the sequence of currently read ys can be continued by reading the character s_i and the rest of the sequence string is also splittable with the new prefixes.
  - Note, that the previous cases can both apply at the same time.
  - Otherwise, the substring beginning at index i is not splittable with the current prefixes p_x and p_y. In case, neither $p_x \circ s_i$ nor $p_y \circ s_i$ are prefixes of x or y respectively, the character s_i can neither contribute to the sequence of already read xs (represented by p_x) nor to the sequence of already read ys. In case, $p_x \circ s_i$ or $p_y \circ s_i$ is a valid prefix but $\mathit{splittable}(i + 1, \mathit{pre}(x, p_x \circ s_i), p_y) = \mathit{false}$ or $\mathit{splittable}(i + 1, p_x, \mathit{pre}(y, p_y \circ s_i)) = \mathit{false}$ respecively, the current character could continue the list of already read x/y, but then the rest of the string is not splittable anymore, i.e. at some point, we will encounter a character that can neither continue the list of already read x nor the list of already read y. Therefore, the substring beginning at index i is not splittable with the current prefixes p_x and p_y.

Runtime

- We have $|x| \cdot |y|$ base cases at $n + 1$, for all possible combinations of prefixes of x and y. Filling the table with *true* for all base cases takes $\mathcal{O}(|x| \cdot |y|)$ time.
- The rest of the table consists of $n \cdot |x| \cdot |y|$ cells. We iterate over these cells in a specific sequence, resulting in $\mathcal{O}(n \cdot |x| \cdot |y|)$ steps.
- The checks inside the innermost loops take constant time for evaluting the table at two different positions. Checking, if the $p_x \circ s_i$ or $p_y \circ s_i$ are prefixes of x or y can also be done in constant time, because know already that p_x and p_y are prefixes of x and y. Therefore, we only have to compare the new character s_i.
- The overall runtime complexity of the algorithm is $\mathcal{O}((n+1) \cdot |x| \cdot |y|)$.

Problem 6

Subproblem a

Basic Idea

- Generate the graph $G_\cap = (V, E_\cap)$ with $E_\cap = \bigcap_{G \in \{G_0, G_1, \ldots, G_b\}} G.E$.
- Find the shortest path from s to t in $G_\cap$ with breadth-first search or another shortest path algorithm, e.g. Dijkstra.

Pseudocode

In the pesudocode, we denote the set of edges at time i with E_i. The function BFS runs the breadth-first search and outputs the shortest path between two vertices.

```
$E_\cap \leftarrow \bigcap_{E \in \{E_0, E_1, \ldots, E_b\}} E$
    return $BFS(V, E_\cap, s, t)$
```

Proof

All paths from s to t that exists in all graphs G_i, must also exist in the intersection of all graphs G_i (resulting in a new graph), and all paths from

s to t in the intersection of all G_i exist in every single graph G_i. Therefore, the shortest path from s to t that exists in all graphs G_i is the shortest path from s to t in the intersection graph of all G_i.

Since all edge weights are the same, i.e. the length is defined as the number of edges, breadth-first search can be used to find the shortest path from s to t.

Runtime

- Generating the intersection of all G_i can be done by checking if every possible edge between to vertices in V is present in every graph G_i. Since we have to check $\mathcal{O}(|V|^2)$ edges with this approach, the runtime for this step is $\mathcal{O}(b \cdot |V|^2)$. Note, that the set of vertices stays the same.
- BFS can be done in $\mathcal{O}(|V| + |E|)$. In the worst case, considering a full graph, the runtime is therefore $\mathcal{O}(|V|^2)$.
- The overall runtime complexity of the algorithm is $\mathcal{O}((b + 1)\dot{|}V|^2)$.

Subproblem b

Basic Idea

- Solve with dynamic programming.
- Let j be the time when the path selected so far changes, $cost(i) = cost(P_0, \ldots, P_i)$, and $shortest(a, b)$ be the shortest path in the intersection of all G_i with $a \leq i \leq b$. Then, $cost(i) = \min\{\min_{0<j<i}(cost(j) + (i - j) \cdot l(shortest(j + 1, i) + K)), (i + 1) \cdot l(shortest(0, i))\}$.
- In the first case of the min function, we take a look at every possibility to select a new path among the graphs G_1 up to G_{i-1}[3]. In that case, we take the same path for all graphs G_{j+1} up to G_i and solve the problem for the previous graphs recursively (that's where dynamic programming comes in). For G_{j+1} to G_i we select the shortest path that is shared with all these graphs (subproblem a) and add the cost K.

[3]For G_i, we cannote decide to take a new path from there on, because there are no more graphs after G_i.

- In the second case of the min function, we choose not to change the current path at all. Therefore, we select the minimum path that is shared with all graphs G_0 up to G_i (subproblem a).

Pseudocode

Note, that there are two different *cost* arrays in the pesudo code. One is one-dimensional, the other one is two-dimensional.

```
for all $i \leftarrow 0$ to $b$ do
    for all $j \leftarrow i$ to $b$ do
        $shortest[i, j] \leftarrow$ algorithm from subproblem a$(G_i, \dots, G_j)$
        $cost[i, j] \leftarrow l(shortest[i, j])$
    end for
end for
$cost[0] \leftarrow l($shortest path in $G_0)$
$change[0] \leftarrow -1$
for $i \leftarrow 1$ to $b$ do
    $cost_c \leftarrow \min_{0<j<i}(cost[j] + K + cost[j + 1, i] \cdot (i - j))$
    $time_c \leftarrow \arg\min_{0<j<i}(cost[j] + K + cost[j + 1, i] \cdot (i - j))$
    $cost_n \leftarrow (i + 1) \cdot cost[0, i]$
    if $cost_c < cost_n$ then
        $cost[i] \leftarrow cost_c$
        $change[i] \leftarrow time_c$
    else
        $cost[i] \leftarrow cost_n$
        $change[i] \leftarrow -1$
    end if
end for
$P \leftarrow$ new List
$i \leftarrow b$
while $i \neq -1$ do
    $nextChange \leftarrow$ change$[i]$
    for $j \leftarrow nextChange + 1$ to $i$ do
        $P$.add$(shortest[nextChange + 1, i])$
    end for
    $i \leftarrow nextChange$
end while
    return $P$.reversed()
```

Proof

We prove by induction that the correct value for $cost[i]$ is generated, given that all $cost[j]$ with $j < i$ are correct.

- **Induction Base:** For $i = 0$, we have only one graph G_0. Therefore, the sequence of paths that minimize the formula is the minimum path in G_0.
- **Induction Hypothesis:** Assume, that for an arbitrary but fixed n, all $cost[i]$ with $i \leq n$ are correct.
- **Induction Step:** We show that $cost[n + 1]$ is calculated correctly. There are two cases that we have to discuss.
  - $cost_c < cost_n$: In that case, it is better to change the path after G_j between G_1 and G_n. We take a look at every possible time value j and try to change the path directly after this point. The total cost for such a change is $cost(p) + K + (n+1-j) \cdot cost(j+1, n+1)$, i.e. the cost for the subproblem (that is smaller than $n - 1$ and therefore correctly solved according to the induction hypothesis) plus the cost for the change plus the cost for the new path for all remaining G_{j+1} to G_{n+1}. For these remaining graphs, the minimal path from s to t minimize the costs. We take change the path at the minimum value of j, if the total costs are smaller than not changing the path at all. The costs for not changing the path at all is the length of the shortest path in the intersection of all graphs, multiplied by the number of graphs.
  - $cost_n < cost_c$: In that case, it is better not to change the path at all, since the costs for that option are smaller than the costs for changing the path at any time j (for an explanation, see previous case).
  - $cost_n = cost_c$: In that case, it does not matter, which option we choose, since they result in the same total cost. In the pseudocode, we always choose not to change the path at all.

The last part of the algorithm reconstructs the path. Whenever the algorithm decides to change the path or not to change the path at all, $change[i]$ contains this decision for time/graph i. In case, the algorithm decides to change the path, *change*$[i]$ is the index of the first graph (counting starts from n) with a different path. Since this value is based on the decision that we proved correct in the induction proof, we can assume that the values for *change* are correct. We can reconstruct the paths by getting the shortest paths for every interval from the two-dimensional *costs* array containing the shortest paths for all pairs of graphs.

Runtime

- Filling *costs* with the shortest paths: We have to run the algorithm from subproblem a for every pair of graphs. This takes $\mathcal{O}(b^3 \cdot |V|^2)$ time[4].
- Filling the *cost* array with the path change points: for each number of graphs $i = 1 \dots b$, we have to take a look at the array values $cost[k]$ of all smaller $k < i$. We assume that comparing and evaluating the cost formula takes constant time. Therefore, the runtime for this step is $\mathcal{O}(b^2)$.
- For extracting the solution, we have to take a look at at most b entries of the *change* array, in case the path changes with every graph. Therefore, the runtime complexity for this step is $\mathcal{O}(b)$.
- The overall runtime complexity for the algorithm is $\mathcal{O}(b^3 \cdot |V|^2)$.

Problem 7

Basic Idea

- Solve with dynamic programming.
- For a duration of i days and an amount of stock r, we calculate the overall profit $\mathit{profit}(i,r) = \max_{0 \leq a \leq r}(\mathit{profit}(i-1, r-a) + a \cdot (p_i - f(a) - \mathit{accumulatedF}(i-1, r-a)))$ with the base cases $\mathit{profit}(1,i) = i \cdot (p_1 - f(r))$ for every $i \in 0 \dots r$ where r is the maximum amount of stock that we are considering in this problem. Note, that a is the amount stock that we sell in the last day.

[4]The value b in the runtime complexity formula for subproblem a is smaller or equal to the value of b in this problem.

- In addition, we maintain the accumulated sums of $f(a)$ for every value in *profit* with $\mathit{accumulatedF}(i, r) = \mathit{accumulatedF}(i-1, r-a) + f(a)$, where a is the minimum argument in the formula of *profit*.

Pseudocode

In the algorithm, $a_{max}[i, r]$ is the amount of stock that we sell at a given date i, given that we have r. We need this array to reconstruct the solution in the second part of the algorithm.

```
$a_{max}[1, i] \leftarrow i \ \forall i \in 0 \dots r$
$\mathit{profit}[1, i] \leftarrow i \cdot (p_1 - f(i)) \ \forall i \in 0 \dots r$
$\mathit{accumulatedF}[1, i] = f(i) \ \forall i \in 0 \dots r$
for all $i \leftarrow 2$ to $n$ do
    for all $q \leftarrow 0$ to $r$ do
        $a_{max}[i, q] \leftarrow \text{argmax}_{0 \le a \le q}(\mathit{profit}[i-1, q-a] + a \cdot (p_i - f(a) - \mathit{accumulatedF}[i-1, q-a]))$
        $\mathit{profit}[i, q] \leftarrow \mathit{profit}[i-1, q-a_{max}[i, q]] + a_{max}[i, q] \cdot (p_i - f(a_{max}[i, q]) - \mathit{accumulatedF}[i-1, q-a_{max}[i, q]])$
        $\mathit{accumulatedF}[i, q] \leftarrow \mathit{accumulatedF}[i-1, q-a_{max}[i, q]] + f(a_{max}[i, q])$
    end for
end for
$\mathit{remaining} \leftarrow r$
for $i \leftarrow n$ downto $1$ do
    $y_i \leftarrow a_{max}[i, \mathit{remaining}]$
    $\mathit{remaining} \leftarrow \mathit{remaining} - y_i$
end for
```

Proof

We prove by induction that $\mathit{profit}[i, q]$ contains the maximum achievable profit for a duration of i days and an amount q of stock, given that $\mathit{profit}(i', q')$ is correct for all $1 \le i' < i$ and all $0 \le q' \le q$.

- **Base Case:** For $i = 1$, we have to sell all stock on the first (and only) day. Therefore, the profit is $q \cdot (p_1 - f(q))$, for all positive integers q. Similarly, $\mathit{accumulatedF}[1, q] = f(q)$.

- **Induction Hypothesis:** Let us assume that $\mathit{profit}[i, q]$ contains the maximum achievable profit for an arbitrary but fixed duration of i days and all positive integers q. Let us furthermore assume that $\mathit{accumulatedF}[i, q]$ contains the accumulated sum of all $f(a_k)$ after day i, where a_k is the amount of stock that is sold on day k, with $k = 1 \dots i$.

- **Induction Step:** Let us assume that selling an amount a of stock on day i yields the maximum profit. In that case, we have to sell an amount $q - a$ of stock in the previous days. We can be sure that $\mathit{profit}[i-1, q-a']$ contains the maximum profit for any amount a' of stock (induction hypothesis) and $\mathit{accumulatedF}[i-1, q-a']$ contains the sum of all $f(a)$ for all previous days. The overall profit is the sum of the profit of the previous days plus the profit for selling an amount a' of stock on the last day i. Therefore, we can find the maximum profit by evaluating all possibilities of selling stock on the last day, result in a maximum profit of $\max_{0 \le a' \le q}(\mathit{profit}(i-1, q-a') + a' \cdot (p_i - f(a') - \mathit{accumulatedF}(i-1, q-a')))$. Therefore, we get the maximum profit at $a = a'$. Since *accumualtedF* is based on the same decision and simply sums up all values of $f(a)$ for every day and amount of stock, we can be sure that it is correct.

Runtime

- Filling the arrays for all base cases: $\mathcal{O}(r)$, assuming that function evaluations and aritmetic operations can be done in constant time.
- Filling the rest of the DP table: the outer loop iterates over all n days and the inner loop iterates over possible numbers of stock r[5]. For every run of the inner loop, we have to check all stock amounts for $i - 1$ in order to find the maximum. The total runtime complexity for filling the DP table is therefore $\mathcal{O}(n \cdot r^2)$.
- Extracting the solution: we have to check n array values. This takes $\mathcal{O}(n)$ time.
- The overall runtime complexity for the algorithm is $\mathcal{O}(n \cdot r^2)$.

[5]Note, that the amount of stock to be sold can never get greater in a later step, therefore it is sufficient to iterate up to r for every day i.

Problem 2

Basic Idea

- We use a modified version of Dijkstra's algorithm to determine the shortest path.
- Instead of distance values, we use arrival time value and evaluate the function $f_e(t)$ for $e = (v_1, v_2)$, if we consider travelling from v_1 to v_2, starting at time t.
- The rest of the algorithm works like the normal Dijkstra algorithm.

Pseudocode

The idea for the Dijkstra algorithm and its steps were taken from Wikipedia. Ine the algorithm, we assume, that there is always a path from the source to the target (otherwise it would be a strange journey). If there is no such path, the algorithm outputs an empty list (since no predecessor exists for the target vertex).

```
$dist[v] \leftarrow \infty \ \forall v \in V$
$pre[v] \leftarrow \emptyset \ \forall v \in V$
$dist[s] \leftarrow 0$
$Q \leftarrow V$
**while** $Q \neq \emptyset$ **do**
    $next \leftarrow$ vertex in $Q$ with minimal $dist$
    $Q$.remove($next$)
    **for all** $e = (next, v_2) \in E$ **do**
        $newDist \leftarrow dist[next] + f_e(dist[next])$
        **if** $newDist < dist[v_2]$ **then**
            $dist[v_2] \leftarrow newDist$
            $pre[v_2] \leftarrow next$
        **end if**
    **end for**
**end while**
$P \leftarrow$ new List
$v \leftarrow t$
**while** $pre(v) \neq \emptyset$ **do**
    $S$.add($v$)
    $v \leftarrow pre[v]$
end while
    return $S$.reversed()
```

Proof

- The algorithm is very similar to Dijkstra's algorithm. However, the edge values are no longer real number but functions of time.
- Dijkstra's algorithm cannot handle negative edge weights. Therefore, it is important, that the function $f_e(t)$ is monotone increasing. If we were able to *go back in time*, this would correspond to a negative edge weight in the graph (arrival time < starting time ⇒ time spent < 0).
- The rest of the proof is similar to the original Dijkstra proof, since the algorithm involves the same steps as Dijkstra. We can prove by induction that the algorithm finds the correct shortest path by proving that the distance values are computed correctly.

Runtime

The outer while loop visits every vertex in V exactly once. For every vertex, we take a look at all of its edges, which can be no more than $|E|$. For every such edge, we query the website once. Therefore, the runtime of the algorithm is bounded by $\mathcal{O}(|V| \cdot |E|)$ queries.