Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department Informatik
Markus Püschel
David Steurer

Johannes Lengler
Gleb Novikov
Chris Wendler

Repetition Exam

Algorithmen und Datenstrukturen

SUMMER 2020

DO NOT OPEN!

Last name, first name: ______________________

Student number: ______________________

With my signature I confirm that I can participate in the exam under regular conditions. I will act honestly during the exam, and I will not use any forbidden means.

Signature: ______________________

Good luck!

	P1: 24P	P2: 16P	T1: 19P	T2: 21P	T3: 8P	T4: 12P	Σ: 100P
Score							
Sign.							

/ 19 P

Theory Task T1.

Notes:

1) In this problem, you have to provide **solutions only**. You do not need to justify your answer.
2) We assume letters to be ordered alphabetically and numbers to be ordered ascendingly, according to their values.

/ 4 P

a) *Landau notation:* Fill out the quiz about asymptotic notation below. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

claim	**true**	**false**
$(2n + n^2 + 3)^2 = \Theta(n^4)$	☒	☐
$\frac{n}{\log n} \leq \mathcal{O}(\sqrt{n})$	☐	☒
$\log(n!) = \Theta(n \log n)$	☒	☐
$\sum_{i=1}^{\log_5 n} 5^i \geq \Omega(n \log n)$	☐	☒

/ 6 P

b) *Graph quiz:* In the following, for a given (undirected) graph $G = (V, E)$, let $n = |V|$, $m = |E|$, and let $A \in \{0, 1\}^{n \times n}$ be the adjacency matrix of G. Indicate whether the following statements are true for all graphs G.

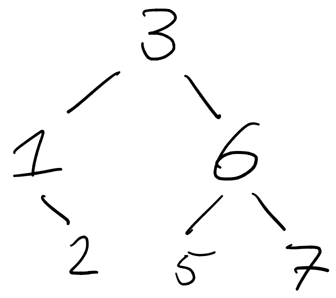
You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

claim	**true**	**false**
A tree with n vertices must have $n - 1$ edges.	☒	☐
The complete graph K_d with $n = d$ vertices cannot be coloured with less than d colours.	☒	☐
An Eulerian walk visits every edge exactly once.	☒	☐
Let $B = A^k$ be the adjacency matrix raised to the k-th power. If $B_{ij} > 0$ then there is a *path* of length k from i to j.	☐	☒
In an unweighted graph, both BFS and DFS can be used to determine shortest paths.	☐	☒
A binary tree of height h (the root has height 0) has at most 2^h leaves.	☒	☐

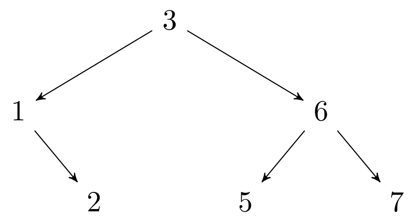
/ 4 P

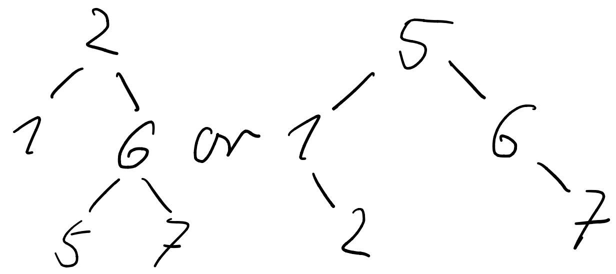
c) *Search trees:*

i) Draw the binary search tree that is obtained when inserting the keys $3, 6, 1, 5, 7, 2$ in this order into an empty tree.

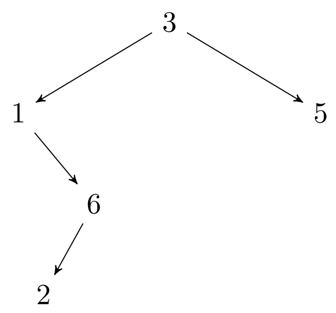


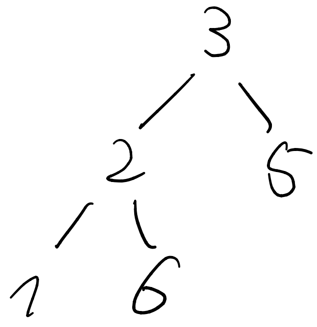
ii) Draw the binary search tree obtained from the following tree by performing the operation `DELETE`(3).



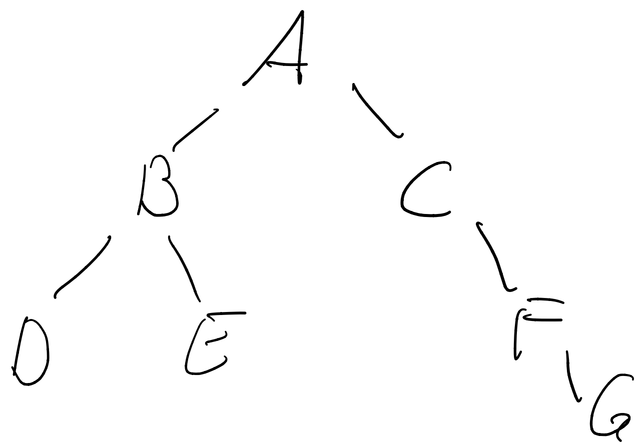


iii) Draw the **AVL tree** that is obtained from the following tree by restoring the AVL-condition.



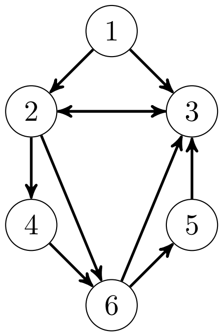


iv) Draw the binary tree that has the pre-order traversal A, B, D, E, C, F, G and the post-order traversal D, E, B, G, F, C, A.



/ **3 P**

d) *Depth-first search / breadth first search:* Consider the following directed graph:



i) Draw the depth-first tree resulting from a depth-first search starting from vertex 1. When processing the neighbors of a vertex, process them in increasing order.

$$\begin{tikzcd}
 & & 1 \arrow[dl, no head] & & \\
 & 2 \arrow[dl, no head] \arrow[dr, no head] & & & \\
3 & & 4 \arrow[dr, no head] & & \\
 & & & 6 \arrow[dr, no head] & \\
 & & & & 5
\end{tikzcd}$$

ii) Draw the breadth-first search tree resulting from a breadth-first search starting from vertex 1. When processing the neighbors of a vertex, process them in increasing order.

$$\begin{tikzcd}
 & & 1 \arrow[dl, no head] \arrow[dr, no head] & \\
 & 2 \arrow[dl, no head] \arrow[dr, no head] & & 3 \\
4 & & 6 \arrow[dr, no head] & \\
 & & & 5
\end{tikzcd}$$

iii) Remove one edge from the graph such that it can be topologically sorted, and give a topological ordering of the resulting graph.

Remove (3,2) i.e. $3 \rightarrow 2$,

1 2 4 6 5 3

/ 2 P

e) *Sorting algorithms:*

Below you see four sequences of snapshots, each obtained during the execution of one of the following five algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

8	6	4	2	5	1	3	7
6	4	2	5	1	3	7	8
4	2	5	1	3	6	7	8

Algorithm: bubblesort

8	6	4	2	5	1	3	7
1	6	4	2	5	8	3	7
1	2	4	6	5	8	3	7

Algorithm: selectionsort

8	6	4	2	5	1	3	7
6	8	2	4	1	5	3	7
2	4	6	8	1	3	5	7

Algorithm: merge sort

8	6	4	2	5	1	3	7
6	8	4	2	5	1	3	7
4	6	8	2	5	1	3	7

Algorithm: insertion sort

/ 21 P

Theory Task T2.

Notes: In this part, you should **justify your answers briefly**.

/ 2 P

a) *Induction:* Prove by mathematical induction that for any positive integer $n \geq 3$,

$$n^2 \geq 2n + 3\,.$$

Solution:

Base case For $n = 3$ we have $9 = 6 + 3$.

Induction hypothesis (I.H.) Assume $k^2 \geq 2k + 3$ holds for some $k \geq 3$.

Induction step ($k \to k + 1$)

$$\begin{aligned}(k+1)^2 &= k^2 + 2k + 1 \\ &\geq 2k + 3 + 2k + 1 \quad , \text{[by I.H.]} \\ &\geq 2(k+1) + 3 \quad , [2k + 1 \geq 2].\end{aligned} \tag{1}$$

/ 3 P

b) *Factorial:*

i) Provide pseudo code using a `for`-loop for the computation of the factorial of a positive integer n:

$$n! = n(n-1)(n-2)\cdots 1.$$

Solution:

Algorithm 1 factorial(n)

```
res = 1
for $k = 1, \ldots, n$ do
    res = $k \cdot$ res
return res
```

ii) Prove the correctness of your algorithm via mathematical induction for all $n \in \mathbb{N}$. **Solution:**We prove `factorial(n)` $= n!$ by mathematical induction.

Base case For $n = 1$ we have `factorial(n)`=1=1! because then the `for`-loop only does one iteration and $res = 1 \cdot 1 = 1$ afterwards.

Induction hypothesis (I.H.) Assume `factorial(m)` $= m!$ holds for some $m \in \mathbb{N}$.

Induction step ($m \to m + 1$) The `for`-loop in `factorial(m + 1)` computes `factorial(m)` in the first m iterations and then multiplies it by $m + 1$ in the last iteration. Thus, factorial(m + 1) = $(m+1)\cdot$`factorial(m)` $= (m+1)!$ by the induction hypothesis.

/ 3 P

c) *Recurrence relations:*

For this exercise, you may use the following master theorem from exercise sheet 4:

Theorem 1 (Master theorem) *Let $T : \mathbb{N} \to \mathbb{R}^+$ be a non-decreasing function such that for all $k \in \mathbb{N}$ and $n = 2^k$,*

$$T(n) \leq aT(n/2) + \mathcal{O}(n^b)$$

for some constants $a > 0$ and $b \geq 0$. Then

- *If $b > \log_2 a$, $T(n) \in \mathcal{O}(n^b)$.*
- *If $b = \log_2 a$, $T(n) \in \mathcal{O}(n^{\log_2 a} \cdot \log n)$.*
- *If $b < \log_2 a$, $T(n) \in \mathcal{O}(n^{\log_2 a})$.*

i) Consider the following recursive function that takes as an input a positive integer m that is a power of 2. Further, let $c \geq 4$ be a power of two.

Algorithm 2 $g(m)$

```
if $m > 1$ then
    for $i = 1, 2, 3, 4, \ldots, c$ do
        $g(m/2)$
        for $j = 1, 2, 4, 8, \ldots, c$ do
            $f()$
else
    return 0
```

Let $T(m)$ be the number of calls of the function f in $g(m)$. Give a recursive formula for $T(m)$.

Solution:

$$T(m) = cT(m/2) + \log_2(c)$$

Determine $T(m)$ in $\mathcal{O}$-notation.

Solution: We have $T(m) \in \mathcal{O}(n^{\log_2 c})$, which follows from the master theorem with $a = c$ and $b = 0$ (observe that $\log_2(c) \in \mathcal{O}(1) = \mathcal{O}(n^0)$) and $0 < \log_2(c)$ for $c \geq 4$.

/ 6 P

d) *Shortest Paths with BFS:* Consider a directed weighted graph G with vertices V, edges E and positive integer edge weights $w : E \to \mathbb{N}$.

i) State the name of an efficient algorithm for computing the lengths of all shortest paths starting from $s \in V$.

Solution: Dijkstra

ii) What is the asymptotic running time of the algorithm?

Solution: $\mathcal{O}((|V| + |E|) \log |V|)$. (Or $\mathcal{O}(|V| \log |V| + |E|)$ with Fibonacci heaps)

When the edge weights are small positive integers, i.e., $w\colon E \to \{1, 2, \ldots, d\}$, it is possible to modify the graph such that the shortest path problem can be solved using BFS on the modified graph $G' = (V', E')$.

iii) Explain how to modify the graph G with small positive integer weights such that the lengths of the shortest paths starting from $s \in V$ can be computed using BFS. How do you read off the length of a shortest path from s to v in G from the BFS in G'?

Solution: We construct a new graph $G' = (V', E')$ by first adding all vertices V to V'. Then, for each edge $(u, v) \in E$ we add $w(e) - 1$ new intermediate nodes $x_1, \ldots, x_{w(e)-1}$ to V' and add the edges $(u, x_1), (x_1, x_2), (x_2, x_3), \ldots, (x_{w(e)-1}, v)$ to E'.

iv) What is the number of vertices and edges in the modified graph G'?

Solution: $|E'| = \sum_{e \in E} w(e)$ and $|V'| = |V| + \sum_{e \in E}(w(e) - 1) = |V| + |E'| - |E|$

v) What is the running time of BFS on G'?

Solution: $O(|V'| + |E'|) = O(|V| + |E'|)$

/ 7 P e) *Top k Elements:* You are given an array of n integers. The goal is to extract the k largest elements from that sequence. Assume that both $k, n \in \mathbb{N}$ are known and that n is significantly larger than k.

i) Which data structure presented in the lecture can be used to solve this task efficiently?

Solution: Max-heap

ii) Provide an algorithm (using pseudo code) that utilizes the data structure from i) to extract the k largest elements from a given input array $x_1, \ldots, x_n$ of integers.

Solution:

Algorithm 3 k-largest($x_1, \ldots, x_n$)

```
maxheap ← heapify($x_1, \ldots, x_n$)
for $i = 1, \ldots, k$ do
    $m$ ← extract maximum from maxheap
    print out $m$
```

iii) Analyze the running time (asymptotic amount of operations) and memory requirements of your solution.

Solution: First, `heapify` creates a Max-heap from n elements in linear time. Then, we perform k times an extraction of the maximum on a heap of size $\approx n$. The extraction of the maximum entails removing the root and fixing the heap condition afterwards which takes $\log n$ time. Thus, overall our solution requires $\mathcal{O}(n + k \log n)$ time.

iv) Now, consider the modified problem: Instead of a finite array $x_1, x_2, x_3, \ldots, x_n$, you now have to process an infinite sequence $x_1, x_2, x_3, \ldots$. Formally, at each time step $t = 1, 2, 3, \ldots$, you get a new integer x_t. Thus, n is not known anymore but k is still known. At some point, you will be asked to provide the k largest elements of $x_1, \ldots, x_t$. But you do not know in advance when that request will come. You may assume that the first request occurs for some $t > k$.

Give an efficient solution to this task by completing the following pseudo code such that each call of the function `TopK` prints the k currently largest elements, i.e., at time t the k largest elements within $x_1, \dots, x_t$. Your solution should require at most $\mathcal{O}(k)$ memory.

Solution:

Algorithm 4 processElement(datastructure, x)

```
// Process the integer $x$. Update your data structure if necessary.
if datastructure.size() $< k$ then
    datastructure.add(x)
else
    $y \leftarrow$ datastructure.getMinimum()
    // getMinimum retrieves the minimum in constant time without deleting it.
    if $y < x$ then
        datastructure.extractMinimum()
        datastructure.add(x)
  return datastructure
```

Algorithm 5 InfiniteTopK(k, $x_1, x_2, x_3, \dots$)

```
// Process the integers $x_1, x_2, x_3, \dots$ such that TopK exhibits the desired behavior.
// Initialize your data structure:
datastructure $\leftarrow$ initialize empty Min-heap of size $k$

for $t = 1, 2, \dots$ do
    datastructure $\leftarrow$ processElement(datastructure, $x_t$)
```

Algorithm 6 TopK(datastructure, k)

```
// Print the $k$ largest elements that have been processed so far.
$l \leftarrow$datastructure.size()
for $i = 1, \dots, l$ do
    print out datastructure.extractMinimum()
```

Theory Task T3. / 8 P

Consider the following procedure that takes two sorted (in ascending order) integer arrays $A = A[1, \dots, n]$ and $B = B[1, \dots, m]$ as input and merges them:

```
procedure MERGE($A, B$)
    $M[1], \dots, M[n], M[n+1], \dots, M[n+m] \leftarrow 0, \dots, 0$
    $i, j \leftarrow 1, 1$
    for $1 \le k \le n+m$ do
        if $i \le n$ and ($j > m$ or $A[i] \le B[j]$) then
            $M[k] \leftarrow A[i]$
            $i \leftarrow i + 1$
        else if $j \le m$ and ($i > n$ or $A[i] > B[j]$) then
            $M[k] \leftarrow B[j]$
            $j \leftarrow j + 1$
    return $M$
```

Note: if $j > m$, then ($j > m$ or $A[i] \le B[j]$) always evaluates to true, no matter whether $B[j]$ is well-defined, and, similarly for ($i > n$ or $A[i] > B[j]$).

Recall that an array C is the result of merging the sorted arrays A and B if C is sorted and contains the same values as A and B (preserving duplicates). For example, if we merge $[1, 2, 4, 6]$ and $[1, 3, 6, 7, 10]$, we obtain $[1, 1, 2, 3, 4, 6, 6, 7, 10]$.

Show that the pseudocode above satisfies the following loop invariant INV(ℓ) for $1 \le \ell \le n + m$: After ℓ iterations of the for-loop,

1) the subarray $M[1, \dots, \ell]$ is the result of merging the subarrays $A[1, \dots, i-1]$ and $B[1, \dots, j-1]$.
2) all values in $M[1, \dots, \ell]$ are at most as large as any of the values in $A[i, \dots, n]$ and $B[j, \dots, m]$.

Specifically, prove the following 3 assertions.

i) INV(1) holds.
ii) If INV(ℓ) holds, then INV($\ell + 1$) holds (for all $1 \le \ell < n + m$).
iii) INV($n + m$) implies that the algorithm correctly merges A and B.

Finally, state the running time of the procedure Merge described above in Θ-notation in terms of n and m (as simplified as possible).

Proof of i). If $A[1] \le B[1]$, then $M[1] = A[1]$ and $i = 2$, $j = 1$ after the first iteration, otherwise $M[1] = B[1]$ and $i = 1$, $j = 2$ after the first iteration. In the first case $M[1]$ is the result of merging $A[1]$ and empty $B[0]$, in the second case $M[1]$ is the result of merging $B[1]$ and empty $A[0]$, so 1) holds.

Condition 2) holds since $M[1] = \min\{B[1], A[1]\}$ which cannot exceed any element of A and B since both A and B are sorted.

Proof of ii). Let's denote the values of i and j after ℓ iterations of the foor-loop by $i(\ell)$ and $j(\ell)$. Notice that condition 1) of INV(ℓ) imlies $i(\ell) - 1 + j(\ell) - 1 = \ell < n + m$, hence either $i(\ell) \leq n$ or $j(\ell) \leq m$.

Consider the case $i(\ell) \leq n$ and ($j(\ell) > m$ or $A[i(\ell)] \leq B[j(\ell)]$). In this case $M[\ell + 1] = A[i(\ell)]$, $i(\ell + 1) = i(\ell) + 1$ and $j(\ell + 1) = j(\ell)$.

Condition 1) of INV($\ell + 1$) holds since $M[1, \ldots, \ell]$ is the result of merging $A[1, \ldots, i(\ell + 1) - 2]$ and $B[1, \ldots, j(\ell+1)]$, and since all elements of $M[1, \ldots, \ell]$ are not greater than $M[\ell+1] = A[i(\ell+1) - 1]$.

If $j(\ell) > m$, condition 1 of INV(ℓ) implies that $j(\ell) = m + 1$. In this case condition 2) of INV($\ell + 1$) holds since $B[m + 1, \ldots, m]$ is empty, any element from $M[1, \ldots, \ell + 1]$ is at most $M[\ell + 1] = A[i(\ell + 1) - 1]$, and A is sorted.

If $j(\ell) \leq m$ and $A[i(\ell)] \leq B[j(\ell)]$, condition 2) holds since any element from $M[1, \ldots, \ell + 1]$ is at most $M[\ell + 1] = A[i(\ell + 1) - 1] \leq B[j(\ell + 1)]$, and A and B are sorted.

The case $j(\ell) \leq m$ and ($i(\ell) > n$ or $A[i(\ell)] > B[j(\ell)]$) is very similar (the solution can be obtained from the previous one by exchanging m with n, A with B and i with j).

Proof of iii). Notice that if $i - 1 \leq n$, $j - 1 \leq m$ and $i - 1 + j - 1 = n + m$, then $i - 1 = n$ and $j - 1 = m$. Hence condition 1) of INV($n + m$) implies that $M[1, \ldots, n + m]$ is the result of merging $A[1, \ldots, n]$ and $B[1, \ldots m]$.

Running time: Each iteration requires $\Theta(1)$ steps, and there are $n + m$ iterations, so the running time is $\Theta(n + m)$.

/ 12 P

Theory Task T4.

Assume that there are n towns $T_1, \dots, T_n$ in the country Examistan. For each pair of distinct towns T_i and T_j, there is exactly one road from T_i to T_j. All of the roads in Examistan are one-way. This implies that there is always a road from T_i to T_j and another road from T_j to T_i. Each road has a nonnegative integer cost that you need to pay to use this road.

For simplicity you can assume that each town T_i is represented by its index i.

/ 1 P

a) Model the n towns, the roads and their costs as a directed weighted graph: give a precise description of the vertices, edges and the weights of the edges of the graph $G = (V, E, w)$ involved (if possible, in words and not formal). What are $|V|$ and $|E|$ in terms of n?

Solution: The towns are modeled as the vertices $V = \{1, \dots, n\}$ of the graph G. The roads are modeled as directed edges $E = \{(i, j) \mid i \neq j, i, j = 1, \dots, n\}$. The costs that you need to pay to use the roads are modeled as the weights w of the respective edges.

The number of vertices is thus $|V| = n$ and the number of edges $|E| = n^2 - n$, since n^2 is the number of possible ordered pairs (i, j) and we have to subtract the n self-edges represented by (i, i) as they are not part of our graph.

Alternative way to get the number of edges: you choose 2 out of n to get the number of unordered sets $\{i, j\}$ with $i \neq j$, resulting in $\binom{n}{2} = \frac{1}{2}(n-1)n$. But we care for the different directions so we have to multiply this number by 2 (for (i, j) and (j, i)) resulting in $|E| = n^2 - n$.

In the following subtasks b) and c), you can assume that the directed graph in a) is represented by a data structure that allows you to traverse the direct successors and direct predecessors of a vertex u in time $\mathcal{O}(\deg_+(u))$ and $\mathcal{O}(\deg_-(u))$ respectively, where $\deg_-(u)$ is the in-degree of vertex u and $\deg_+(u)$ is the out-degree of vertex u.

/ 6 P b) Due to the epidemiological situation in Examistan, the authorities decided to reduce the number of trips between different towns. Now the only way to get from one town to another is to use the roads. Moreover, if you want to travel from town T_i to the other town T_j, you must visit a test center during your trip (in T_i or T_j or elsewhere with a detour). Since test centers are expensive, there are only $k < n$ of them, and they are located only in the first k towns $T_1, \dots, T_k$ (i.e., one test center in each of these towns).

Assume that you need to fill the table of minimal costs required to travel between all pairs of towns, which takes into account the new rules of travelling. Provide an as efficient as possible algorithm that takes as input a graph G from task a) and a number k, and outputs a table C such that $C[i][j]$ is the minimal total cost of roads that one can use to get from T_i to T_j while also visiting a test center. You can assume that for all $1 \leq i \leq n$, $C[i][i] = 0$.

What is the running time of your algorithm in concise Θ-notation in terms of n and k? Justify your answer.

Solution: First note that we can *not* use a simple breadth-first search here, as we have a weighted graph with non-uniform weights (just a disclaimer, because in student solutions this appeared a lot).

For the calculation of the shortest paths between all pairs we make the following observations: We have $|E| \in \Theta(|V|^2)$, hence Bellman-Ford leads to a runtime of $O(|V|^4)$. Johnsons algorithm leads here to $O(|V|^2 \log |V| + |V|^3)$, the same runtime as running Dijkstra $|V|$-times. Floyd-Warshall on the other hand needs $O(|V|^3)$, hence for Θ-tightness we modify the Floyd-Warshall algorithm. The idea is to find the shortest paths from each node to the nodes $T_1, \dots, T_k$ and from the nodes $T_1, \dots, T_k$ to all nodes. Then we use these results to obtain the cheapest paths from T_i to T_j via at least one of the $T_1, \dots, T_k$ as the minimal sum of the cheapest-from-paths and the cheapest-to-paths.

```
function ModifiedFloydWarshall($G$)
    $C_{\text{from}}, C_{\text{to}}$                ▷ Cheapest paths from/to $T_1, \ldots, T_k$, initial infinity
    for $\ell$ from 1 to $n$ do
        for $i$ from 1 to $n$ do
            for $j$ from 1 to $k$ do
                if $C_{\text{from}}[i][j] > w_{i,\ell} + w_{\ell,j}$ then
                    $C_{\text{from}}[i][j] \leftarrow w_{i,\ell} + w_{\ell,j}$
                if $C_{\text{to}}[j][i] > w_{\ell,i} + w_{j,\ell}$ then
                    $C_{\text{to}}[j][i] \leftarrow w_{\ell,i} + w_{j,\ell}$
    $C[i][i] \leftarrow 0$                ▷ Path to itself is zero for each node
    for $\ell$ from 1 to $k$ do
        for $i$ from 1 to $n$ do
            for $j$ from 1 to $n$ do
                if $C[i][j] > C_{\text{from}}[i][\ell] + C_{\text{to}}[\ell][j]$ then
                    $C[i][j] \leftarrow C_{\text{from}}[i][\ell] + C_{\text{to}}[\ell][j]$
```

The algorithm is basically two times Floyd-Warshall, hence we obtain a running time of $\Theta(n^2k)$, as in our modified version one loop only goes until k and not n.

/ 5 P

c) Now suppose that before building the test centers in towns $T_1, \dots, T_k$, the authorities had made the roads between all different T_i and T_j among $T_1, \dots, T_k$ free of charge (i.e. their cost is now 0). Solve the problem from subtask b) assuming this condition.

That is, provide an as efficient as possible algorithm that takes as input a graph G from task a) and a number k, and outputs a table C such that $C[i][j]$ is the minimal total cost of roads that one should use to get from T_i to T_j with visiting a test center.

What is the running time of your algorithm in concise Θ-notation in terms of n and k? Justify your answer.

Solution: Since the paths between the test centers are now free of charge (i.e $w_{i,j} = 0$), we can consider the test centers as a super-vertex (i.e. all the test centers merged to one vertex and connected to each other vertex with the minimal cost of all costs from the test centers). This graph has $n - (k - 1)$ vertices. Then we run Dijsktra algorithm one time to get the shortest paths from each vertex to the super-vertex in an array $C_1[i]$. Then we reverse the edges and run Dijkstra another time to get the shortest paths from the super-vertex to each vertex in an array $C_2[j]$ (this is required as the weights of the graph do not need to be symmetric). The shortest path from vertex i to vertex j passing through one test center is then $C[i][j] = C_1[i] + C_2[j]$.

Creating the super-vertex is $O(n - k)$ if we assume that the array containing the edge weights for each vertex are sorted, otherwise it is $O((n - k) \cdot k)$. The runtime for Dijkstra algorithm implemented with a Fibonacci heap on the modified graph with super-vertex is $O((n-k)\log(n-k) + (n-k)^2 - (n-k))$, which does not change if we run it two times. Filling the final table is $O((n - k)^2)$, so the overall runtime is $\Theta((n - k)^2)$.

Extra space. Please indicate clearly to which task your notes belong. Please cross out all notes that you do not want to be graded.

Extra space. Please indicate clearly to which task your notes belong. Please cross out all notes that you do not want to be graded.

Extra space. Please indicate clearly to which task your notes belong. Please cross out all notes that you do not want to be graded.

Extra space. Please indicate clearly to which task your notes belong. Please cross out all notes that you do not want to be graded.