Algorithms and Data Structures (IBC027)
Solutions and Correction Guidelines

January 16, 2019

You are allowed to answer in Dutch. Whenever an algorithm is required, it can be given in pseudocode or plain English (or Dutch), and its running time and correctness must **always** *be justified (even informally, but in a clear way!). For problems 2, 3, 4, 5 and 7.1, where an algorithm must be defined, 20% of the points is for the correctness argument, and 20% for the analysis of the running time. If the sum of the scores for problems 1-7 is x and the bonus score for the homework is b, then the grade for the exam is* $\min(10, b + 0.5 + \frac{9.5}{8}x)$. *Success!*

1 Quiz (1 point)

Solutions: Questions have been taken from `https://www.geeksforgeeks.org/algorithms-gq/`. Each correct answer is worth 0.25points. Students do not have to provide any justification for their answer.

Question 1 Which of the following is not $\mathcal{O}(n^2)$?

(A) $(15^{10}) * n + 12099$

(B) $n^{1.98}$

(C) $n^3/\sqrt{n}$

(D) $(2^{20}) * n$

Solution: C

Question 2 Is following statement true or false? If a DFS of a directed graph contains a back edge, any other DFS of the same graph will also contain at least one back edge.

(A) true

(B) false

Solution: A (DFS will run into a back edge iff the graph has a cycle.)

Question 3 An undirected graph G has n nodes. Its adjacency matrix is given by an $n \times n$ matrix whose diagonal elements are 0's, and non-diagonal elements are 1's. Which one of the following is true?

(A) Graph G has no minimum spanning tree (MST)

(B) Graph G has a unique MST of cost $n - 1$

(C) Graph G has multiple distinct MSTs, each of cost $n - 1$

(D) Graph G has multiple spanning trees of different costs

Solution: C

Question 4 Consider the following algorithms:

1. Breadth First Search
2. Depth First Search
3. Prim's Minimum Spanning Tree
4. Kruskal' Minimum Spanning Tree

What are appropriate data structures to use in an implementation of these algorithms?

(A) 1) Stack 2) Queue 3) Priority Queue 4) Union Find

(B) 1) Queue 2) Stack 3) Priority Queue 4) Union Find

(C) 1) Stack 2) Queue 3) Union Find 4) Priority Queue

(D) 1) Priority Queue 2) Queue 3) Stack 4) Union Find

Solution: B

2 Divide and conquer (1.5 points)

You are given a one dimensional array that may contain both positive and negative integers. Describe an $\mathcal{O}(n \log n)$ divide and conquer algorithm to find the sum of the contiguous subarray of numbers which has the largest sum. For example, if the given array is $[-2, -5, \mathbf{6}, \mathbf{-2}, \mathbf{-3}, \mathbf{1}, \mathbf{5}, -6]$, then the maximum subarray sum is 7 (see highlighted elements). The sum of the empty subarray is defined to be 0.

Solution The general strategy is this:

1. If the array contains a single value v then return $\texttt{max}(0, v)$.
2. Otherwise, divide the array in two halves. Then there are three possibilities:
   a) Maximum subarray entirely contained in left half
   b) Maximum subarray entirely contained in right half
   c) Maximum subarray overlaps with both halves

   For cases 2.a and 2.b we do a recursive call. For case 2.c we use a simple linear algorithm.
3. Return the maximum of the results for 2.a, 2.b and 2.c.

For case 2.c, the idea is to find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending at some point to the right of mid + 1, and then add the two and return. Below is the pseudocode is shown.

Correctness: If I is an array and i and j are indices with $i \leq j$, then we must compute

$$\mathsf{MSA}(I, i, j) \quad = \quad \texttt{max}(0, \texttt{max}_{i \leq i' \leq j' \leq j} \sum_{k=i'}^{j'} I[k]).$$

In the special case where $i = j$ this simplifies to

$$\mathsf{MSA}(I, i, j) \quad = \quad \texttt{max}(0, I[i]),$$

Algorithm 1: Computation of maximimum subarray value.

```
Function MSA(I,i,j)
    if $i = j$ then
        return max(0,I[i])
    $m \leftarrow \lfloor (i+j)/2 \rfloor$
    $left \leftarrow$ MSA(I,i,m)
    $right \leftarrow$ MSA(I,m+1,j)
    $middle \leftarrow$ MSAbegin(I,i,m) + MSAend(I,m+1,j)
    return max(left, right, middle)
Function MSAbegin(I,i,m)
    $current \leftarrow -\infty$
    $sum \leftarrow 0$
    for k from m to i do
        $sum \leftarrow sum + I[k]$
        if $current < sum$ then
            $current \leftarrow sum$
    return current
Function MSAend(I,m,j)
    $current \leftarrow -\infty$
    $sum \leftarrow 0$
    for k from m to j do
        $sum \leftarrow sum + I[k]$
        if $current < sum$ then
            $current \leftarrow sum$
    return current
```

as computed by our algorithm. If $i < j$ and m is any value with $i \leq m < j$ then

$$\begin{aligned}
\mathsf{MSA}(I,i,j) &= \mathtt{max}(0, \mathtt{max}_{i \leq i' \leq j' \leq m} \sum_{k=i'}^{j'} I[k], \mathtt{max}_{m+1 \leq i' \leq j' \leq j} \sum_{k=i'}^{j'} I[k], \mathtt{max}_{i \leq i' \leq m < j' \leq j} \sum_{k=i'}^{j'} I[k]) \\
&= \mathtt{max}(\mathsf{MSA}(I,i,m), \mathsf{MSA}(I,m+1,j), \mathtt{max}_{i \leq i' \leq m < j' \leq j} \sum_{k=i'}^{j'} I[k]) \\
&= \mathtt{max}(\mathsf{MSA}(I,i,m), \mathsf{MSA}(I,m+1,j), (\mathtt{max}_{i \leq i' \leq m} \sum_{k=i'}^{m} I[k]) + (\mathtt{max}_{m < j' \leq j} \sum_{k=m+1}^{j'} I[k])).
\end{aligned}$$

Now the correctness of our algorithm follows from the following observations:

- $i \leq \lfloor (i+j)/2 \rfloor < j$
- function $\mathsf{MSAbegin}(I,i,m)$ computes $\mathtt{max}_{i \leq i' \leq m} \sum_{k=i'}^{m} I[k]$
- function $\mathsf{MSAend}(I,m+1,j)$ computes $\mathtt{max}_{m < j' \leq j} \sum_{k=m+1}^{j'} I[k]$.

Complexity: The recursion equation describing the number of computation steps of the algorithm is (with $n = j - 1 + 1$ the size of the input):

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

This is exactly the same recurrence as the one for the mergesort algorithm that was discussed during the lecture, and for which it was proved that $T(n) \in \mathcal{O}(n \log n)$.

Students may earn 0.9points for the algorithm, 0.3 points for the correctness argument, and 0.3points for the complexity argument. (For the algorithm: 0.3 for divide and conquer idea, 0.1 for identification of three cases, deduct 0.2 if algorithm is correct but has the wrong complexity, deduct 0.1 for mistake with empty subarray.) When students only give a textual explanation as above and an informal argument for correctness this is sufficient. For the complexity, mentioning of the recurrence equation and a reference to mergesort suffices. There is one subtle corner case that occurs when the array only contains negative numbers. Given that the sum of the empty subarray is 0, I would argue that the maximum subarray sum is 0 in this case. However, if an empty subarray is not allowed then the maximum subarray sum will be negative in this case. This solution is also considered to be correct. The maximum subarray problem can be solved in $\mathcal{O}(n)$ using Kadane's algorithm, see `https://en.wikipedia.org/wiki/Maximum_subarray_problem`. If students come up with a correct $\mathcal{O}(n \log n)$ algorithm that does not use divide and conquer this is fine.

3 Graph algorithms (1.5 points)

After a wine tasting event organized by Thalia in the Mercator building, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus that you could possibly use. Unfortunately, no single bus visits both the bus stop near the Mercator building and your home; you must change buses at least once. There are exactly B different buses. Each bus starts at 12:00am (midnight), makes exactly N stops, and finally stops running at 11:59:59pm. You are in time to catch the first bus that leaves from the bus stop closest to the Mercator building. Buses always run exactly on schedule, and you have an accurate watch. Finally, you are far too tired to walk between bus stops.

1. Describe and analyze an algorithm to determine the sequence of bus rides that gets you home as early as possible. Your goal is to minimize your arrival time, not the time you spend traveling.

2. Oh, no! The wine tasting event was held on Halloween, and the streets are infested with zombies! The bus company doesn't have the funding to add additional buses or install zombie-proof bus stops, especially for only one night a year. Describe and analyze an algorithm to determine a sequence of bus rides that minimizes the total time you spend waiting at bus stops; you don't care how late you get home or how much time you spend on buses. (Assume you can wait inside the Mercator building until your first bus is just about to leave.)

Solution We have B different buses, say $\{1, \ldots, B\}$, which each stop N times. Since we know the complete schedule of all the buses, we may assume a function *time* that specifies, for each bus $1 \leq b \leq B$ and for each $1 \leq s \leq N$, the time of arrival $time(b, s)$ of bus b at its s-th stop. We also know a function *loc* that specifies, for each bus $1 \leq b \leq B$ and for each $1 \leq s \leq N$, the location $loc(b, s)$ of the s-th stop of b. We write M for the location of the bus stop closest to the Mercator building, and H for the location of the bus stop closest to home. Different solutions for the problem are possible, depending on the graph that we define. We describe two solutions, one $\mathcal{O}(n^2)$ and another $\mathcal{O}(n^2)$, where n is the size of the input.

1) In the first solution, we construct a directed graph G in which the vertices are pairs (b, s) of a bus b and a stop s. In addition, graph G contains a single start vertex v_0. Graph G contains the following edges:

1. An edge from v_0 to any vertex (b, s) with $loc(b, s) = M$. This expresses that you can take any bus that leaves from the bus stop closest to the Mercator building.

2. An edge from (b, s) to $(b, s + 1)$ for each b and each $s < N$. This expresses that bus b will bring you from the s-th stop to the $s + 1$-th stop.

3. An edge from (b, s) to (b', s') for any pair of vertices with $loc(b, s) = loc(b', s')$ and $time(b, s) < time(b', s')$. This expresses when you can switch from bus b to bus b'.

Graph G has $n = (B \times N) + 1$ vertices and $\mathcal{O}(n^2)$ edges. (Basically, n is the size of the input of our problem, which consists of the complete bus schedule.) In order find the fastest route home, we perform a BFS starting from node v_0 and then, amongst the nodes (b, s) with $loc(b, s) = H$ reachable from v_0, we select the one with $time(b, s)$ minimal. By following the predecessor edges starting from this selected vertex, we obtain the sequence of bus rides that gets us home as quickly as possible. Our algorithm is correct since graph G encodes the routes of all buses, and all possible ways to switch from one bus to another. We may construct graph G in time $\mathcal{O}(n^2)$. The BFS can also be carried out in time $\mathcal{O}(n^2)$, after that we need $\mathcal{O}(n)$ time to find the minimal reachable node, and $\mathcal{O}(n^2)$ time to retrieve the sequence of bus rides. Thus the overall time complexity of our algorithm is $\mathcal{O}(n^2)$.

2) We now assign weights to the edges of G: edges of type (1) and (2) get weight 0, and we assign weight $time(b', s') - time(b, s)$ to each type-(3) edge from (b, s) to (b', s'). The weights describe how long we have to wait at a bus stop. Next we use Dijkstra's algorithm to compute the shortest path from source vertex v_0 to any other vertex, that is, the sequence of bus rides that minimizes the total time we need to wait for transfers. Then we pick the vertex (b, s) with $loc(b, s) = H$ for which the length of the shortest path $d(b, s)$ is minimal, and we track the predecessor edges to retrieve the sequence of bus rides. Again, correctness is obvious from the definitions: the length of the shortest path is exactly the total time we need to wait at bus stops. Again, we may construct graph G in time $\mathcal{O}(n^2)$. If we use an implementation of Dijkstra's algorithm with arrays, this takes $\mathcal{O}(n^2)$ time. After that we need $\mathcal{O}(n)$ time to find the minimal reachable node, and finally $\mathcal{O}(n^2)$ time to retrieve the sequence of bus rides. Thus the overall time complexity is $\mathcal{O}(n^2)$.

1') In 2nd more efficient solution, we construct a directed graph $G = (V, E)$ where the vertices are all the pairs (l, t), such that there is some bus that halts at location l at time t:

$$V = \{(l, t) \mid \exists b\ \exists s : l = loc(b, s) \text{ and } t = time(b, s)\}.$$

Set E containts two types of edges. First we have edges that correspond to *bus rides* from one stop to the next. For each bus b and for each $s < N$, E contains a transition from $(loc(b, s), time(b, s))$ to $(loc(b, s + 1), time(b, s + 1))$. In addition, E contains edges that correspond to *waiting* for the arrival of the next bus at some bus stop: we have an edge from vertex (l, t) to vertex (l, t') if $t' > t$ and there is no vertex $(l, t'') \in V$ with $t < t'' < t'$. Note that each vertex has at most B outgoing edges corresponding to bus rides, and at most one outgoing waiting-edge. Moreover, both V and E are in $\mathcal{O}(n)$. Let t_0 the earliest time that a bus stops at M. In order find the fastest route home, we perform a BFS starting from vertex (M, t_0) and then, amongst all nodes (H, t) reachable from (M, t_0), we select the one with t minimal. By following the predecessor edges starting from this selected vertex, we obtain the sequence of bus rides that gets us home as quickly as possible. Our algorithm is correct since graph G encodes the routes of all buses, and all possible ways to switch from one bus to another. We may construct graph G in time $\mathcal{O}(n \log n)$ by first sorting all the vertices in V according to lexicographic order: this allows us to find for each vertex (l, t) the time t' at which the next bus arrives at l. The BFS can also be carried out in time $\mathcal{O}(n)$, after that we need $\mathcal{O}(n)$ time to find the minimal reachable node, and $\mathcal{O}(n)$ time to retrieve the sequence of bus rides. Thus the overall time complexity of our second algorithm is $\mathcal{O}(n \log n)$.

2') We give weight 0 to all the bus ride edges and weight $t' - t$ to each waiting edge from (l, t) to (l, t').[1] Next we use Dijkstra's algorithm to compute the shortest path from source vertex (M, t_0) to any other vertex, that is, the sequence of bus rides that minimizes the total time we need to wait for transfers. Then we pick the vertex (H, t) for which the length of the shortest path $d(H, t)$ is minimal, and we track the predecessor edges to retrieve the sequence of bus rides. Again, correctness is obvious from the definitions: the length of the shortest path is exactly the total time we need to wait at bus stops. Again, we may construct graph G in time $\mathcal{O}(n \log n)$.

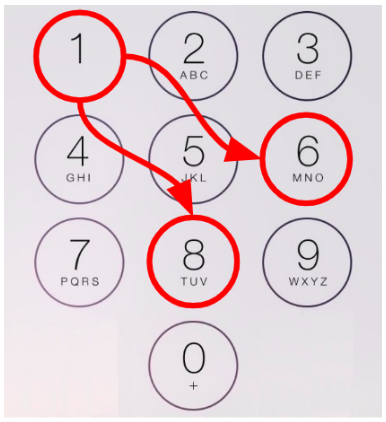
[1]Technically, it is possible that an edge is both a bus ride edge and a waiting edge: this happens when a bus b stops at a location l, and the next stop of b is also at location l. In realistic bus schedules this never happens, but if your goal is to avoid zombies you want to be inside a bus than outside. Thus we assign wait 0 to these edges.

If we use an implementation of Dijkstra's algorithm with priority queues, this takes $\mathcal{O}(n \log n)$ time. After that we need $\mathcal{O}(n)$ time to find the minimal reachable node, and finally $\mathcal{O}(n)$ time to retrieve the sequence of bus rides. Thus the overall time complexity is $\mathcal{O}(n \log n)$.

Students may earn 0.75points for both parts, and for each part 0.45point for the algorithm, 0.15point for the correctness argument, and 0.15point for the complexity. There are a couple of variations possible in the solution. In our solutions, we assume that you can swith from bus (b, s) to bus (b', s') if $loc(b, s) = loc(b', s')$ and $time(b, s) < time(b', s')$. Students may be more optimistic and only require $time(b, s) \leq time(b', s')$, or more pessimistic and require $time(b, s) + O \leq time(b', s')$, where O is some minimal transfer time. This is all considered to be correct. In part 2, one may use implementations of Dijkstra with different data structures. This is considered correct, even though it may lead to different complexity bounds. The problem statement does not require that solutions must be efficient, so algorithms that e.g. use Floyd-Warshall or even exponential algorithms are all fine, as long as they are correct.

4 Dynamic programming (1.5 points)

Imagine you place a knight chess piece on a phone dial pad. This chess piece moves in an uppercase "L" shape: two steps horizontally followed by one vertically, or one step horizontally then two vertically:



Suppose you dial keys on the keypad using only hops a knight can make. Every time the knight lands on a key, we dial that key and make another hop. The starting position counts as being dialed. Give an efficient dynamic programming algorithm for the following problem: How many distinct numbers can you dial in N hops from a particular starting position? Specify the recursion equations on which your algorithm is based.

Solution. This question is used by Google during interviews, for a discussion see `https://hackernoon.com/google-interview-questions-deconstructed-the-knights-dialer-f780d516f029`.

For any key $k \in \{0, 1, 2, \ldots, 9\}$, let $neighbors(k)$ be the set of keys that you can reach from k in a single hop. So we have, for instance,

$$\begin{aligned} neighbors(1) &= \{6, 8\} \\ neighbors(4) &= \{0, 3, 9\} \\ neighbors(5) &= \emptyset \end{aligned}$$

Function $neighbor$ may be computed in constant time using e.g., a linked list representation. If we dial a number in n hops, then this number will comprise $n + 1$ digits. The following recursion equations specify how many numbers we can dial in n hops from starting position p:

$$\begin{aligned} T(p,0) &= 1 \\ T(p,n) &= \sum_{q \in neighbors(p)} T(q,n-1) \quad \text{if } n>0 \end{aligned}$$

(As usual, the sum over the empty index set is defined to be 0. Indeed, if $N > 1$ there is no number with N digits that you can dial when you start in 5.) The recurrence equations suggest a simple bottom up dynamic programming algorithm for solving our problem. We fill a $10 \times (N+1)$ integer array A by first filling the first column $A(p,0)$ with 1's for each key p, the fill the second column using the recursion equation, then fill the third column, etc. Finally, given starting position q, the algorithm outputs the value of $A(q,N)$. The complexity of this algorithm is $\mathcal{O}(N)$.

A variant of the above solution may be obtained by viewing the 10 keys as points in a two-dimensional grids. In a two dimensional grid, the knight can always make 8 moves but only some of them will end on a key again. This idea leads to slightly different recursion equations, but the resulting algorithm is equivalent.

Students get 0.9points for the correct recursion equations, 0.3points for description of the algorithm based on these equations (either bottom up or top down), and 0.3points for the complexity argument. Correctness trivially follows from the equations. We deduct 0.4points in case the recursion equations are not given explicitly, but are hidden in the code for the algorithm.

Some students misinterpreted the assignment and thought they had to compute the number of different keys that can be reached in N hops. In English, to "dial a number" means dialing a phone number that consists of a sequence of keys/digits. Thus, for native speakers the problem description is unambiguous. For non-native speakers confusion is possible and so we give some points to students who solved the alternative problem. However, computing the number of keys that you can reach in N hops is trivial and can be done in constant time using DFS (even manually), so students should have become suspicious. Students can get at most 0.75points for such a solution.

5 Network flow (1.5 points)

You are organizing a dance event, to be held all day Friday, Saturday, and Sunday. Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints:

1. Exactly k sets of music must be played each day, and thus $3k$ sets altogether.
2. Each set must be played by a single DJ in a consistent music genre (ambient, bubblegum, dubstep, horrorcore, K-pop, Kwaito, mariachi, straight-ahead jazz, trip-hop, Nashville country, parapara, ska,...).
3. Each genre must be played at most once per day.
4. Each DJ has given you a list of genres they are willing to play.
5. Each DJ can play multiple sets per day, but at most three sets during the entire event.

Suppose there are n candidate DJs and g different musical genres available. Give an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.

Solution. We construct a flow graph with a source s, a sink t, a vertex for each of the three days, a vertex for each music genre, and a vertex for each DJ. We add the following edges:

- An edge with capacity k from source s to each of the three nodes that represent days. The idea is that each music set corresponds to a flow of size 1 in the network. This encodes that at most k sets can be played each day (first constraint).

- An edge from each day to each music genre with capacity 1. This encodes that each genre can be played at most once per day (third constraint).
- An edge from each music genre to each DJ that is willing to play that genre with capacity ∞ (second and fourth constraint).
- An edge from each DJ to the sink t with capacity 3 (fifth constraint).

If there exists an assignment of a DJ and a genre to each of the $3k$ sets that is consistent with all the constraints, we can easily define a flow of size $3k$ in the flow graph. Conversely, whenever the network as an integer flow of size $f > 0$, we may construct a corresponding assignment for f music sets iteratively by "following the flow": just pick any day d such that the flow from s to d is positive, then pick any genre gn such that the flow from d to gn is positive (exists by flow preservation), and then pick any DJ dj such that the flow from gn to dj is positive (exists by flow preservation). Add triple (d, gn, dj) to the list of scheduled sets, reduce to flow along the path $s \to d \to gn \to dj \to t$ by one, etc. Thus the flow graph has a maxflow of $3k$ iff there exists an assignment of a DJ and a genre to each of the $3k$ sets that is consistent with all the constraints. Since the maxflow of the network is $3n$ (along incoming edges of t), our algorithm reports that no assignment is possible if $k > n$. Otherwise, we compute the maxflow using the Ford Fulkerson algorithm. If the maxflow is less than $3k$, we report that no assignment is possible, otherwise we iteratively generate an assignment using the follow-the-flow approach sketched above.

Suppose that together the DJs have m preferences. Then $m \in \mathcal{O}(gn)$. The number of vertices in the flow graph is $5 + g + n$ and the number of edges is $3 + 3g + n + m$. We may thus construct the flow graph in $\mathcal{O}(g + n + m)$ time. The running time of Ford Fulkerson is $\mathcal{O}(Mf)$, where M is the number of edges and f is the value of the maximal flow. Thus the time complexity of running Ford-Fulkerson (with $k \leq n$) is $\mathcal{O}((g + n + m) \times n)$. Once we have computed the max flow, we may extract the assignment in $\mathcal{O}(n)$ time. This the overall complexity of our algorithm is $\mathcal{O}((g + n + m) \times n)$.

The solution we present here is not unique: one may for instance swap the role of source and sink, and reverse all the edges in the network. Also, duplicate or redudant nodes may be added to the network. Of course there are many incorrect solutions. Deduct 0.3points for every constraint that is violated by the proposed flow network. It is ok if students use other max flow algorithms instead of FF, even if the complexity is worse. However, we deduct 0.1 points if the algorithm is not polynomial in the size of the input (which occurs if FF is used without the preceding test for $k > n$). We also deduct 0.1 points if students do not explain how an assignment is computed from the maximal flow. Students may earn 0.9 points for the algorithm, 0.3points for the correctness argument, and 0.3points for the complexity argument.

6 Greedy (1.5 points)

We have n skiers with heights $p_1, \ldots, p_n$ and n pairs of skis with heights $s_1, \ldots, s_n$. We want to minimize the average difference between the height of a skier and the skis that are assigned to him/her. That is, if skier p_i is assigned pair of skis $s_{a(i)}$ then we want to minimize:

$$Cost \quad = \quad \frac{1}{n} \sum_{i=1}^{n} (\mid p_i - s_{a(i)} \mid).$$

1. Consider the following greedy algorithm. Find the skier and the pair of skis whose height difference is minimal. Assign this skier this pair of skis. Repeat the process until every skier has skis. Prove or disprove that this algorithm is correct.
2. Consider the following greedy algorithm. Give the shortest skier the shortest pair of skis, the second shortest skier the second shortest pair of skis, the third shortest skier the third shortest pair of skis, etc. Prove or disprove that this algorithm is correct

Hint: One of the above greedy algorithms is correct and the other is not. (Of course you are not allowed to use this hint in your proof, otherwise there wouldn't be much to prove once you find a counterexample for one of the algorithms.)

Solution. 1) Algorithm 1 is incorrect. Suppose we have 2 skiers with lengths 160 and 190, and 2 pairs of skis with length 140 and 170. Then Algorithm 1 will first match 160 and 170, and then 140 and 190, leading to an average difference of $(10 + 50)/2 = 30$. This is not optimal because if we match 160 with 140, and 190 with 170, we obtain an average difference of $(20 + 20)/2 = 20$, which is smaller.

2) This is probably the most difficult question from the exam. We prove by contradiction that Algorithm 2 is optimal. If Algorithm 2 is not optimal, then there exists some input $p_1, \ldots, p_n, s_1, \ldots, s_n$ for which it does not produce an optimal solution. Without loss of generality we may assume that sequences $p_1, \ldots, p_n$ and $s_1, \ldots, s_n$ are sorted in increasing order. Then the output of greedy Algorithm 2 is $G = \{(p_1, s_1), \ldots, (p_n, s_n)\}$.

Let the optimal solution be $T = \{(p_1, s_{a(1)}), \ldots, (p_n, s_{a(n)})\}$. Beginning with p_1, we compare T and G. Let p_i be the first person who is assigned different skis in G than in T. Without loss of generality we may assume that T is the optimal solution for which i is maximal, that is, the optimal solution that agrees with G maximally long. Let s_j be the pair of skis assigned to p_i in T, and let p_k be the person who gets skis s_i in T. Then $i < k$ and $i < j$, and thus $p_i \leq p_k$ and $s_i \leq s_j$. Now create a solution T' by switching the ski assignments of p_i and p_j. Then we have

$$Cost(T') - Cost(T) \quad = \quad \frac{1}{n}(\mid p_i - s_i \mid + \mid p_k - s_j \mid - \mid p_i - s_j \mid - \mid p_k - s_i \mid).$$

The next lemma implies that $Cost(T') \leq Cost(T)$, and thus T' is also optimal. This contradicts our assumption that T is the optimal solution that agrees with G maximally long.

Lemma. Suppose $p \leq p'$ and $s \leq s'$. Then $\mid p - s \mid + \mid p' - s' \mid \leq \mid p - s' \mid + \mid p' - s \mid$.

Proof: Since $\mid x - y \mid = \mid y - x \mid$, the lemma is fully symmetric and we can swap the roles of p, p' and s, s'. Thus we may assume without loss of generality that $p \leq s$. There are three cases:

- $p \leq p' \leq s \leq s'$. Then
$$\mid p - s \mid + \mid p' - s' \mid = s - p + s' - p' = s' - p + s - p' = \mid p - s' \mid + \mid p' - s \mid .$$
- $p \leq s \leq p' \leq s'$. Then, using that $s - p' \leq 0 \leq p' - s$,
$$\mid p - s \mid + \mid p' - s' \mid = s - p + s' - p' \leq s' - p + p' - s = \mid p - s' \mid + \mid p' - s \mid .$$
- $p \leq s \leq s' \leq p'$. Then, using that $s - s' \leq 0 \leq s' - s$,
$$\mid p - s \mid + \mid p' - s' \mid = s - p + p' - s' \leq s' - p + p' - s = \mid p - s' \mid + \mid p' - s \mid .$$

Hence the lemma follows.

Students may earn 0.75points both for part 1 and for part 2. For part 2, any sensible beginning of a correctness argument is worth 0.3points. Students who have the complete proof except for the proof of the lemma get 0.5points.

7 NP completeness (1 points)

The problem AllOrNothing3Sat asks, given a Boolean expression that is the conjunction of clauses such that each clause contains exactly three literals, whether there is an assignment to the variables such that each clause either has three True literals or has three False literals.

1. Describe a polynomial-time algorithm to solve AllOrNothing3Sat.
2. But 3Sat is NP-hard! Why doesnt the existence of this algorithm prove that P=NP?

Solution. 1) This is the butterfly problem from the Exam of 2017 in disguise!

We construct an undirected graph $G = (V, E)$ as follows. Each variable is a vertex. There is an edge between x and y if there is a clause in which both variables occur. An edge is labelled "same" if either both x and y occur as literals in some clause, and "different" if either x and $\neg y$ occur in a clause, or $\neg x$ and y occur in a clause.

Let's say an assignment to the variables is *consistent with* G when variables connected with a "same" edge have the same value, and variables connected with a "different" edge have a different value. By construction, an assignment is consistent with G iff each clause has either three True literals or three False literals. If there is both a "same" and a "different" edge between two variables, then there is no consistent assignment possible, and we are done. We use DFS to determine whether a consistent assignment exists.

We arbitrarily designate some variable as the starting node. Graph G need not be connected in which case we choose starting nodes for each component. For each component G_i label the starting node with True. (We can do this without loss of generality because if an assignment is consistent on G_i then the assignment in which we negate the value of all variables from G_i is also consistent.) Now perform BFS on G_i starting at the selected starting node. For each node y_k that is visited from y_j, consider the label of edge (y_j, y_k). If y_k has not been visited before, we assign a label to it. If the label of the edge is "same," we label y_k the same as y_j, and if the label of the edge is "different," we label y_k with the negation of y_j. If y_k has been labeled before we check whether the edge label is consistent: if the edge label is "same," labels of y_k and y_j must be the same, and otherwise labels of v_k and v_j must be different. If we discover an inconsistency we stop the algorithm and return No. After succesful termination of the BFS we return Yes.

Suppose we have n clauses. Construction of graph G will take $\mathcal{O}(n)$ time. Since the number of edges of graph G is $\mathcal{O}(n)$, the running time of the BFS algorithm is also $O(n)$. Thus the overall complexity of our algorithm is $\mathcal{O}(n)$.

Students earn 0.7pt for a correct algorithm, 0.1pt for the complexity argument (also if the algorithm is incorrect; it is sufficient to say the algorithm is polynomial), and 0.1pt for the correctness argument. Students get no points for an incorrect algorithm, but may get some points for an algorithm that is correct but not fully specified.

2) The input of AllOrNothing3Sat and 3Sat is the same, but besides that the problems are completely different. In particular, an efficient algorithm for AllOrNothing3Sat cannot be used to construct an efficient algorithm for 3SAT (we have not constructed a polynomial reduction from 3SAT to AllOrNothing3Sat). Thus, unfortunately, we haven't proved P=NP. For this insight, students get 0.3points.