Second Round Theoretical

Solutions

Swiss Olympiad in Informatics

March 7, 2020

Evaporation

Task Idea	Daniel Rutschmann
Task Preparation	Yunshu Ouyang
Description English	Yunshu Ouyang
Description German	Benjamin Schmid
Description French	Yunshu Ouyang
Solution	Yunshu Ouyang & Daniel Rutschmann

Subtask 1: (10 Points)

The answer is 4, one solution would be to first add 3 and 4 together, then, after a minute, there will be one glass with 6, one with 1 and all others 0. Then we add 6 and 1 together to get 7. After that, 3 minutes pass so that glass with 7 will in the end only contain 4 milliliters.

Subtask 2: (20 Points)

Let us first consider the case $k \geq n$. In this case, a glass will never empty due to evaporation, so the only way of emptying a glass is to pour it into another one. (If $k = n$, some glasses might get empty right at the end.) After each minute, we loose one milliliter of water for every non-empty glass. We should hence minimize the number of non-empty glasses by always pouring an (arbitrary) non-empty glass into another non-empty glass. This reduces the number of non-empty glasses by one. If we do this, we loose $n - i - 1$ milliliters after the i-th minute, and one milliliter in the last minute (where all of the water is in a single glass and we hence can't further reduce the number of non-empty glasses). Hence after n minutes, we're left with

$$k \cdot n - \sum_{i=0}^{n-2}(n - i - 1) - 1 = k \cdot n - \frac{n(n-1)}{2} - 1$$

milliliters of water. This is optimal, as we minimize the total evaporation.

For $k < n$, we still try to minimize the total evaporation. During the first k minutes, a glass will never empty due to evaporation, so we can only reduce the number of non-empty glasses by one for each minute. After k minutes, all glasses we didn't interact with will be empty and the best thing we can do is to have all water in a single glass, in which case we loose only one milliliter during each of the remaining $n - k$ minutes. Hence after n minutes, the maximum number of milliliters we could be left with is

$$\max\left(0, k \cdot n - \sum_{i=0}^{k-1}(n - i - 1) - \sum_{i=k}^{n-1} 1\right) = \max\left(0, \frac{k(k+1)}{2} - (n - k)\right)$$

The following two-phase strategy achieves this:

1. During the first k minutes, pour any non-empty glass into the first glass.
2. After that, all the remaining water is in the first glass, so we just wait for the remaining $n - k$ minutes.

In the first phase, we reduce the number of non-empty glasses by one each minute. Since we only pour into the first glass, all other glasses will be empty after k minutes. Hence during phase two, the first glass is the only non-empty one.

Summary: The answer is given by the following formula

$$\begin{cases} k \cdot n - \frac{n(n-1)}{2} - 1 & \text{for } k \geq n \\ \max(0, \frac{k(k+1)}{2} - (n - k)) & \text{for } k < n \end{cases}$$

This can be computed in constant time.

Runtime: $O(1)$ Memory: $O(1)$ (We only need to know n and k)

Subtask 3: (30 Points)

There are a few observations to make:

- It never makes sense to first wait and then do pouring operations. Doing the pouring operations as soon as possible reduces the number of non-empty glasses and hence the evaporation. We should therefore first do all our pouring operations and then wait for the remaining minutes.
- Pouring water is never worse than not pouring water. Consider two glasses with values a, resp. b. After one minute, they will be $a-1$, resp. $b-1$, so we have lost 2 milliliters in total. However, by pouring one into the other, we will only lose 1 milliliter from the two glasses. Hence if there are two or more non-empty glasses, we should always do a pouring operation.
- Every minute, the amount of water you lose is equal to the number of non empty glasses. This is because every minute each glass with a positive number of milliliters loses one milliliter.
- Glasses that contain less water will empty sooner. The time used to empty one glass with a milliliters is a minutes.

By combining those observations, we get that it is optimal to always reduce the number of non-empty glasses. And, as glasses with less water empty sooner, it is better to leave glasses with less water to get emptied than to wait for glasses with more water. Hence, it is optimal to pour the glass with the most water into the glass with the second most water at the end of every minute.

Proof of optimality: Formally, consider this solution after m minutes. At that point, we're either left with exactly one non-empty glass, or all glasses i with at $a_i \leq m$ are empty (due to evaporation) and $m-1$ other glasses are empty (due to pouring). In either case, it is impossible to have more empty glasses after m minutes. Hence this solution maximizes the number of empty glasses at each point and is thus optimal.

Implementation: The straight-forward implementation would be to simulate the process. This consists of n steps where in each step, we first add the two maximum values together and then decrease all positive values in the list by 1. This can be sped up to $O(n \log n)$ by sorting or by using a max-heap or segment tree. Since $a_i \leq n$, you can use counting sort to get $O(n)$ time.

Since we know the integers form a permutation of $\{1, \ldots, n\}$, a better solution is to realise that the glass with n milliliters will loose exactly n milliliters due to evaporation at the end of the n minutes. Suppose glass i contains i milliliters of water. For all the other glasses, we can just add the amount that is left at the time of pouring that glass into the glass n. That means we can pour $i-(n-1-i) = 2 \cdot i - n + 1$ milliliters of water from the glass i if $2 \cdot i \geq n-1$. Hence, the answer is:

$$\sum_{i=\lceil \frac{n-1}{2} \rceil}^{n-1} (2 \cdot i - n + 1) = \lfloor \frac{n^2}{4} \rfloor$$

Runtime:

- Straight-forward implementation: $O(n^2)$ (naive), $O(n \log n)$ (with sort or heap) or $O(n)$ (with counting sort).
- Closed form solution: $O(1)$

Memory:

- Straight-forward implementation: $O(n)$
- Closed form solution: $O(1)$

Subtask 4: (40 points)

In subtask 3, we've shown that it is always optimal to pour the glass with the most water into the glass with the second most water at the end of every minute. There is again a straight-forward quadratic solution that just simulates this process.

Another approach would be to reuse the same idea as the last subtask as we know the ordering of the operations we need to do. One way would be to sort the values in increasing order. Then, similar to the closed form of the previous subtask, this time, the answer is:

$$\max\left(0, \sum_{i=0}^{n-1} \max(a_i - (n-i-2), 0) - 1 - n\right)$$

Instead of sorting the values, one could also use some data structure which give you the greatest value such as a heap.

This can be sped up to $O(n)$ with the following observation: We don't care about the order of the glasses with $a_i > n$, as those glasses will never empty. Hence we only need to sort the glasses with $a_i \le n$. This can be done in linear time with counting sort.

An alternative solution is to partition the glasses into two sets: one set containing all glasses which will be empty and the other containing those which will be used. Suppose now we lose k of the glasses. Then, we can construct the solution: we take the $n-k$ largest values and merge them one by one. By definition of k, all $n-k$ values are $\ge k$ so we will only lose

$$\frac{(n-k)\cdot(n-k-1)}{2}$$

milliliters of water in total for the $n-k$ large values. The question now becomes: which k do we want? The best k should be the greatest k such that the k largest values are all greater or equal to k. In order to find which k is the best one we can use binary search.

This idea can be further optimized in the following way: we first look at the median value m of the input, and check whether it can be the best k (the best k being the greatest value such that the k largest values are all $\ge$ to k). Two possibilities then appear:

- If less than m values are $\ge m$, than k should be smaller. In this case, the larger half values are already $\ge m/2$ so they are also $\ge k$ so we only need to check the other $n/2$ candidates.
- If more than m values are $\ge m$, then then $k \ge m$. The smaller half of the values will evaporate anyways since they evaporate in less than m minutes and $m \le k$. So we need to check only the upper half $n/2$ candidates.

We need to iterate this procedure again and again by taking the median of the median on the remaining candidates. At the k-th step, the runtime is of $O(n/2^{k-1})$so in total the runtime is:

$$O\left(\sum_{k=1}^{\lceil \log(n) \rceil} \frac{n}{2^{k-1}}\right) = O(n)$$

Runtime:

- Straight-forward implementation: $O(n^2)$
- Sorting / heap / binary search: $O(n\log(n))$
- Partition by $a_i \le n$, then counting sort: $O(n)$
- Quickselect: $O(n)$

Memory:

- Straight-forward implementation: $O(n)$
- Closed form solution: $O(n)$
- Quickselect: $O(n)$

Number Hunt

Task Idea	Monika Steinova
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Johannes Kapfhammer
Description French	Florian Gatignon
Solution	Johannes Kapfhammer

Subtask 1: Analyze Stofls Table (10 points)

In a first step, write down all possible values matching the bit pattern. This is done in column iteration 1. Then, notice the following: There are identical rows "?10", and their options are $\{2,6\}$. As both are different, all other rows can not be 2 or 6, so we can remove that option from them. We do the same for the row "1?1". The sets can't be further reduced.

Num	b_2	b_1	b_0	Iteration 1	Iteration 2
a_0		1	0	$\{2,6\}$	$\{2,6\}$
a_1	1			$\{4,5,6,7\}$	$\{4\}$
a_4	1		1	$\{5,7\}$	$\{5,7\}$
a_2		1	0	$\{2,6\}$	$\{2,6\}$
a_3			0	$\{0,2,4,6\}$	$\{0\}$
a_5	1		1	$\{5,7\}$	$\{5,7\}$
a_6				$\{\dots\}$	$\{1,3\}$

Ask query $q(6,1)$. If $q(6,1)=1$ then $a_6=3$, if $q(6,1)=0$ then $a_6=1$.

Subtask 2: Optimal Algorithm (90 points)

Num	Bits
$a_0=3$	0
$a_1=1$	0
$a_2=6$	1
$a_3=7$	1
$a_4=0$	0
$a_5=4$	1
$a_6=2$	0

Missing number must be "1??"

Num	Bits	
$a_0=3$	0	
$a_1=1$	0	
$a_2=6$	1	1
$a_3=7$	1	1
$a_4=0$	0	
$a_5=4$	1	0
$a_6=2$	0	

Number must be "10?"

Num	Bits		
$a_0=3$	0		
$a_1=1$	0		
$a_2=6$	1	1	
$a_3=7$	1	1	
$a_4=0$	0		
$a_5=4$	1	0	0
$a_6=2$	0		

Number must be "101'

The idea is the following: we query the first bit of every integer and count the number of occurrences of 0s and 1s. Since from 0 to 2^n-1 exactly 2^{n-1} numbers start with 0 and 1 and one number is missing, we either are missing a 1 or a 0. So we check which is occurring less often and then we know the first bit.

Thus we can "throw away" all numbers starting with a certain bit. The remaining numbers have the same property as before, if we ignore the first bit: They are from 0 to $2^{n-1}-1$, unique, and exactly one number is missing. So we can apply the algorithm recursively.

The following code implements this idea:

```cpp
// n: number of bits
// elements: indices of the 2^n-1 elements
int solve(vector<int> elements, int n) {
  if (n == 0) return 0;
  array<vector<int>, 2> side; // all elements starting with 0 (1) are in side[0] (side[1])
```

```
    // partition elements into the two sides
    for (auto x : elements)
      side[q(x, n-1)].push_back(x);
    // find out which side is missing a number
    int missing_bit = side[0].size() < side[1].size() ? 0 : 1;
    // recurse on the first n-1 bits in the relevant subset
    return (missing_bit<<(n-1)) | solve(side[missing_bit], n-1);
}
```

Proof of correctness

The goal is to prove the correctness by *induction* on n. In order for induction to work out, it is important to state the intended behaviour of the function (other than "it just works"). Properly defining the properties of input and output allows us to make use of those properties to prove correctness, which makes the whole proof a lot easier.

The function `solve` takes as input n, the number of bits, and `elements`, a set of size $2^n - 1$ with indices of the candidate numbers, where the that the last n bits of them are unique. Then, the function returns the last n bits of the missing number.

Base case n = 0: Because 0 is only one number with 0 bits, this case is handled correctly.

Induction step: Suppose our algorithm works for $n-1$. We have to show that it works for n. By the definition of `solve`, the vector `elements` has size $2^n - 1$. Thus if we look at the first bit, either `side[0]` or `side[1]` has size $2^{n-1} - 1$ and the other has size $2^{n-1} - 1$ (this is because the last n bits are unique and there are only 2^n possibilities). Let s be the smaller side. Observe that s has size $2^n - 1$ and the last $n-1$ bits of them are unique (as by assumption the first n bits are unique and they share the same bit at index $n-1$). Therefore we can apply the induction hypothesis and get that `solve`$(s, n-1)$ correctly identifies the last $n-1$ bits of the missing number. Because we know the $n-1$-th bit, the bitwise or of that with the result give us the value of the last n bits.

Analysis of Queries

We define $Q(n)$ as the number of queries made for a call of `solve` with parameter n (for arbitrary values of `elements`). We make one query for each element in the vector, so we do $2^n - 1$ direct calls. But we also do some indirect calls in the recurrence. Thus

$$Q(n) = 2^n - 1 + Q(n-1)$$
$$Q(0) = 0$$

Let's compute some values and try to come up with a hypothesis:

n	0	1	2	3	4	5	6	7	n
$Q(n)$	0	1	4	11	26	57	120	247	$2^{n+1} - n - 2$
$Q(n) + n + 2$	2	4	8	16	32	64	128	256	2^{n+1}

To prove this, we again use induction. We want to formally prove that $Q(n) = 2^{n+1} - n - 2$.

Base case n = 0: $Q(0) = 0 = 2^1 - 0 - 2$.

Induction step: Suppose the formula is true for $n-1$. We verify:

$$Q(n) = 2^n - 1 + Q(n-1) = 2^n - 1 + (2^n - (n-1) - 2) = 2 \cdot 2^n - n - 1 + 1 - 2 = 2^{n+1} - n - 2$$

Analysis of Queries, Time and Space

The computation of the function is proportional (for some fixed constant) to the number of queries it makes and thus the running time is $O(C \cdot 2^{n+1} - n - 2) = O(2^n)$.

We can also prove it by hand. Let $T(n)$ be the time required for a call of `solve` with parameter n (for arbitrary values of `elements`). We go through the vector once, the computation without the recursion takes at most $C \cdot 2^n + D$ time, for some constants C and D.

Important note: the wiki on "Introduction to Algorithm Design" has a similar recurrence where it states "To formally prove it, we do not want to use $\mathcal{O}$-notation because hiding constants in a recurrence can be dangerous.". While it can work out if you do it correctly, you can very easily "cheat" the notation and get incorrect results. Thus we work with constants and only at the end apply the $\mathcal{O}$-notation.

$$T(n) = C \cdot 2^n + D + T(n-1)$$
$$T(0) = D$$

This recursion can be proven again by induction (similar to the number of queries) and we get $T(n) = 2C \cdot (2^n - 1) + D \cdot n = \mathcal{O}(2^n)$. Because we can't use more memory than time, we also have $\mathcal{O}(2^n)$ memory.

Binnabike

Task Idea	Joël Mathys, Ian Boschung
Task Preparation	Yunshu Ouyang
Description English	Yunshu Ouyang
Description German	Jan Schär
Description French	Yunshu Ouyang
Solution	Johannes Kapfhammer

Subtask 1: (5 Points)

The optimal solution is the direct path $a-f-b$ without any intermediate station. This path has cost 5.

Subtask 2: (35 Points)

Since Binna has unlimited time, we only need to minimize the total amount of money she has to spend. This is not necessarily the shortest path, because coming to a station can give a discount of up to d. One idea that comes to mind would be reducing cost d from every edge going out of a station, but this has two problems: (a) the edge weights might become negative (so Dijkstra won't work anymore) and (b) if the distance to the next station is less than d Binna won't get the full discount. Any correct solution needs to find a way to cope with those two problems.

DP does not work

Let's try the following DP approach. At any position on a ride, we keep track of the "total cost" and the "time since last station". When driving, we increase the "time since last station" until we reach another station. Then we compute the cost of that travel, which is "time since last station" minus d, with a minimum of 0.

$$\begin{aligned}
\texttt{DP}[v][t_s] &= \text{minimum cost from } a \text{ to } v\text{, where the time since last station has been } t_s\\
\texttt{DP}[v][t_s] &= \min_{\substack{u\in N(v)\\ c=\texttt{cost}(u,v)\\ c>t_s}} \texttt{DP}[v][t_s-c]\\
\texttt{DP}[s][0] &= \min_{t_s} \texttt{DP}[s][t_s]+t_s \quad \text{if } s \text{ is a station}\\
\texttt{DP}[0][d] &= 0
\end{aligned}$$

As for most DPs, the argument why this is correct is that we exhaustively cover all viable paths, and such the minimum is achieved by the best path. We can find the solution at $\texttt{DP}[b][0]$.

Unfortunately, we can not compute this DP so easily, because there are cycles. It is not clear in which order we should compute the table. Always check your DP for cycles!

State Graph with full state

An easy way out of this problem is to do a state graph. We split vertex v into $t+1$ copies $v_0,\dots,v_t$, where v_i represents $\texttt{DP}[v][i]$. We draw an edge of cost 0 between v_i and u_j if (u,v) is an edge in G (the input graph) and if $\texttt{cost}(u,v)=j-i$. Also we draw an edge of cost i from v_i to v_0 for every v that is a station. Then, the shortest path from a to v_j is exactly what we want the value of $DP[i][j]$ to be, and in particular the solution to the problem is the distance from a_0 to b_0. For that we run Dijkstra.

Of course, when it "works" on the state graph we might think it could also "works" on the DP. This is true: we need to compute the values in increasing order of values and the best way to do

this is storing the values in a priority queue. But that's basically the same as Dijkstra on the state graph.
Runtime: $O(t \cdot (n \log n + m))$
Memory: $O(t \cdot n)$

State Graph with discount level

We can reduce the dependence on t to d by redefining how the costs are computed. This time, we keep track of the "total cost" and the "discount level" (where d means Binna has the full discount available, i.e. is at a station, and 0 means she has completely used it up). When driving, we pay from the discount level (i.e. subtracting the costs there) until that reaches 0. Only once it is 0, we add the driving costs directly to the total cost. When reaching a station, we recharge the available discount back to d. This is equivalent to what happens in the task statement, but it allows for easier solutions.

(Note that it the definition of a discount level is equivalent to the time since the last station, but capped at d, which leads to an equivalent solution where one dimension is reversed.)

Concretely, we split each node into $d + 1$ copies, the j-th of which means that available discount is j. Let v_j denote vertex v with discount level j. For every edge (u, v) of cost c we add edges (u_j, v_{j-d}) of cost 0 for $j \geq d$ and (u_j, v_0) of cost $d - j$. The graph now has has $n \cdot d$ vertices and $m \cdot d$ edges. We can run Dijkstra (note all edges have non-negative weight) on it to find the shortest path from s_d to t_j (for any j) and get the solution.
Runtime: $O(d \cdot (n \log n + m))$
Memory: $O(d \cdot n)$

Reducing to a Station-Station Graph

We can do better by reducing the graph G of n vertices into a graph H of k vertices consisting of only the stations. For this, we run Dijkstra (all edges have non-negative weight in the statement) from every station to compute the distance between it and any other station. Next, we subtract d from every distance to account for the discount of d, and take the maximum of 0 and the new cost since discounts doesn't accumulate.

In this new graph, we can run Dijkstra (or Floyd-Warshall; in any case the edge weights are still non-negative) to get the shortest path from a to b.
Runtime: $O(k \log n + m + k^3)$
Memory: $O(n + m + k^2)$
Since $k^3 < n$ and $k \log n < d \cdot n \log n$, this running time is better than the previous one.

Subtask 3: (60 Points)

We can't reuse the solution of the previous subtask, but we can reuse some ideas from it. As we don't have infinite time anymore, we somehow need to keep track of the time as well.

All of the solutions shown below are with "discount levels", but of course there are (suboptimal) solutions that work without those.

State Graph

We can use the state graph from before and additionally split up each vertex into $t + 1$ copies to account for the time needed to reach it. This is essentially the same solution as the DP below. With Dijkstra, it is slower by a log factor. One can get rid of that by noticing that the graph is a DAG (directed acyclic graph) and to compute it using DP.

DP with time, position and discount level

We try the following DP approach:

$$\begin{aligned}\texttt{DP}[v][t_v][\ell] &= \text{minimum cost from } a \text{ to } v \text{ within } t_v \text{ minutes and a discount level of } \ell \\ &= \min_{\substack{u\in N(v)\\ c=\texttt{cost}(u,v)\\ c\geq\ell}} \texttt{DP}[t_v-c][u][\min(d,\ell+c)]+\min(d-\ell,c)\\ \texttt{DP}[a][0][d] &= 0\end{aligned}$$

We argued in subtask 2 that the modelling with discount levels is correct. All possible paths are considered by this DP.

But why does it work this time? Because we added a parameter t_v and longer paths take longer time. So we can compute the states in increasing t_v. Because we only consider strictly smaller values of t_v (note the edge weights are non-negative), we don't have any cyclic dependencies.
Runtime: $O((n+m)\cdot t\cdot d)$
Memory: $O(n\cdot t\cdot d)$

We can make two independent optimizations on this DP. Combining them leads to the optimal solution.

DP on Station-Station Network

The first optimization is that we don't need all vertices v. Like in the second subtask, we can make a station-station network with only k vertices. Then, we do the DP from before (rather than Dijkstra) on this second network.

It is very important to note here that some of the edge costs are now 0. Then, the DP gets tricky, because there could be cycles! If vertices a and b are connected with an edge of distance 0, then both should have the same table. But if a is computed before b, and b has the smaller value, this will not reach a. There are two possible fixes: either merge all vertices with distance 0 into one mega-vertex, or do the DP twice, once in increasing order of vertex index, and once in decreasing order.
Runtime: $O(k\cdot(n\log n+m)+k^2\cdot t\cdot d)$
Memory: $O(n+m+k\cdot t\cdot d)$

DP with optimal time $\pm$dt

The second optimization is that we don't need all values t_v. We can make the following observations:

- No optimal solution will visit the same station twice. This is because the discount does not accumulate. Note that we may visit the same vertex multiple times, but not the stations.
- Look at some path that travels between stations on shortest paths. By inserting another station into the path, you can save a cost of at most d. Because the path will be longer and the discount this station gives is only d.
- Compared to the shortest path from a to b, by going over k stations, one can save at most $d\cdot k$. So if the shortest path from a to v is $d(a,v)$, the minimal cost will need at most $d(a,v)-d\cdot k$ time.
- Thus, the only times we need to consider for vertex v are between $d(a,v)-d\cdot k$ and $d(a,v)$. So only $d\cdot k$ values instead of t.

By precomputing dist(a,v) for all v with a single Dijkstra, ignoring all times outside of the viable range, and carefully computing the DP states in the correct order, we get a running time of $O((n+m)\cdot d^2\cdot k)$ instead of $O((n+m)\cdot t\cdot d)$.
Runtime: $O((n+m)\cdot d^2)$
Memory: $O(n\cdot d^2)$

DP on Station-Station Network with optimal time ±dt

We combine both approaches. First we create the station-station network using Dijkstra. Then we compute the DP where we only consider $d \cdot t$ values per vertex.
Runtime: $O(k \cdot n \log n + m + d \cdot k^3)$
Memory: $O(n + m + d \cdot k)$

Venice

Task Idea	Johannes Kapfhammer
Task Preparation	Stefanie Zbinden
Description English	Martin Raszyk
Description German	Stefanie Zbinden
Description French	Florian Gatignon
Solution	Stefanie Zbinden

Subtask 1: Example

The best solution for the example is two use two gondolas. This can be achieved by having one gondola move from sight 4 to sight 6 and the other from sight 5 to sight 0.

Subtask 2: Many gondolas

Consider the following algorithm that constructs a gondola between 0 and any sight that is a leaf:

1. Read the list of edges and store them in an adjacency `adi`.
2. iterate over every sight, then if this sight is a leaf (that is, the size of `adi` of that sight is 1) add a gondola from that sight to 0.

This algorithm runs in $O(m + n)$, where m is the number of canals and n the number of sights, but as the graph is a tree, we know that $m = n - 1$, and thus the algorithm runs in $O(n)$. The memory used is $O(n)$.

To prove the correctness of this algorithm, we have to prove the following two things:

- The set of gondolas in the output is **valid**. That is, at each stop, there is at least one gondola that stops there
- Any valid set of gondolas uses at least half the number of gondolas.

Validity Think of the input graph as a tree rooted at sight 0. Then the gondolas our algorithm outputs are exactly the gondolas from any leaf to sight 0. As any sight lies on the path between some leaf and the root of the tree, there is at least one gondola stopping at each sight.

Not too many gondolas Let k be the number of leaves in the graph. Then our algorithm uses at most k gondolas (In fact it uses exactly k if 0 is not a leaf and $k - 1$ otherwise). We will show that any valid set of gondolas uses at leas $k/2$ gondolas. Let u be a leaf. Then, a gondola only stops at u if one of its endpoints is u itself. Thus, to cover every leaf, we need at least $k/2$, which is what we wanted to prove. As $k \leq 2 \cdot (k/2)$ we satisfy the constraint about the number of gondolas we had to.

Subtask 3: Tight limit on gondolas

We saw in the previous section that we need at least $k/2$ gondolas, we will see that it is indeed possible to always find a solution which uses only $\lceil k/2 \rceil$ many gondolas. (We need to round up, as $k/2$ might not be an integer). Hence, what we have to do is pairing up the leaves in a smart way.

Consider the following algorithm: we root the tree at node 0 and enumerate the leaves in the order they were visited by the dfs. Then, we add a gondola between the i-th and the $i + \lfloor k/2 \rfloor$-th leaf. An implementation of this might look like:

```
vector <int> adi;
vector <int> vis;
vector <int> leaves;
vector <pair <int, int> > gondolas;
```

```
void dfs(int node){
    if (vis[node]) return;
    vis[node] = 1;
    if (adi[node].size() == 1){ // if node is a leaf add it to the list of leaves
        leaves.push_back(node);
    }
    for (auto next : adi[node]) dfs(next);
}
int main(){
    /* first read the input and store the graph in adi
    construct the vector vis to be zero for every node*/
    dfs(0);
    int k = leaves.size(); // k is the number of leaves
    for (int i=0; i+k/2<k; i++){
        // add a gondola from i-th leaf to the (i+k/2)-th leaf
        gondolas.push_back(leaves[i], leaves[i+k/2]);
    }
}
```

The algorithm uses $k - \lfloor k/2 \rfloor = \lceil k/2 \rceil$ gondolas, also, it runs in $O(n)$ time and uses $O(n)$ memory. The only thing we have left to do is to prove that the set of gondolas the algorithm constructs is valid.

Note that for the rest of the proof, we will consider the graph as a tree rooted at node 0. To prove the validity, the following claim is useful:

Claim: *Let u be a node. Then either, all leaves are contained in the subtree of u or there exists a gondola that has one endpoint in the subtree of u and one endpoint that is not in the subtree of u.*

Before we prove that claim, we will show why it is useful for the prove: Let u be a node and assume there exists a leaf not contained in the subtree of u. Then, the lemma implies, that there is a gondola starting in the subtree of u and ending outside of it. Note that every path from the subtree of u to a node outside of that subtree has to contain u and its parent. Or, reformulated, if the subtree of u does not contain all leaves, then there exists a gondola, that stops at u. Hence, to prove validity, we only have to worry about nodes whose subtrees contain all the leaves. However, there is only one such node, namely the root 0. Why? If the node 0 has only one child, then the node 0 is a leaf and not contained in any other subtree. If it has more than one child, any leaf form one of the children is not contained in the subtree of the other children. So the only thing we have left to prove, that there exists a gondola that stops at node 0. If 0 has only one child, then it is a leaf, and hence there even exists a gondola starting there. Thus, we can assume that 0 has at least two children say one of them is u. Applying the claim for u combined with out observation implies that there exists a gondola stopping at 0.

Finally, we have proven that the algorithms is correct, assuming the claim holds so this is all we have left to prove.