Second Round Theoretical

Solutions

Swiss Olympiad in Informatics

March 9, 2019

Cheese Recommendation

Task Idea	Joël Mathys
Task Preparation	André Ryser
Description English	André Ryser
Description German	André Ryser
Description French	Yunshu Ouyang
Solution	André Ryser
Correction	André Ryser and others

Subtask 1: (10 points)

Your task was to find a pair out of 9 types of cheese that maximises the following expression:

$$\frac{\left|a_i - a_j\right| \cdot b_i \cdot b_j}{\max(|b_i|, \left|b_j\right|)} \tag{1}$$

Note that the expression (i.e. the recommendation score) is negative if b_i and b_j have a different sign. For this reason we only have to compare types of cheese that have the same sign in the characteristic b.

In this case the expression is maximal with $i = 5$, $j = 6$, since $\frac{|5-(-6)| \cdot -3 \cdot -2}{\max(|-3|,|-2|)} = 22$. This solution runs in $O(n^2)$.

Subtask 2: (60 points)

For this subtask we can assume that b_i is positive everywhere.

The easiest solution is a brute-force solution where we pair each type of cheese with every other type, calculate the expression (1) and compare it to the maximal value of the expression seen so far.

There is a more efficient solution that uses the following observation. Suppose we only compare the type of cheese i only with other types such that $b_i \leq b_j$. We are still comparing every type of cheese with every other type since we have either $b_i \leq b_j$ or $b_i \geq b_j$. The expression (1) now simplifies to $\left|a_i - a_j\right| \cdot b_i$. The dependency on b_j disappears. The only term that still depends on type of cheese j is $\left|a_i - a_j\right|$.

We can sort all the types of cheese in ascending order according to their b characteristic. Now we can traverse the list from the back and keep track of the biggest $a_{i,\max}$ and smallest $a_{i,\min}$ value of a so far. One of the two values will maximize $\left|a_i - a_j\right|$ ($a_j \in \{a_{i,\max}, a_{i,\min}\}$). Therefore it is sufficient to traverse the list once to calculate the maximum recommendation score.

The runtime of this solution is $O(n \log n)$. We need to sort all the types of cheese once. The calculation of the recommendation score afterwards is linear, sicne we traverse the list just once.

It is relatively easy to see why this algorithm is correct. It is obviously correct to just look at each possible pair of types of cheese and pick the one with the maximum score. To sort the list by the b property and then to only compare cheese i with j when $b_i \leq b_j$ is equivalent, since we still look at every pair of type of cheese. The expression is now $|a_i - a_j| \cdot b_i$ and only the term a_j depends on cheese j. $|a_i - a_j|$ is maximal if either a_j is the maximal or minimal value of an a that appears in the list after cheese i. We keep track of the minimum as well as the maximum of a and get the maximum recommendation score for cheese i when calculating $\max |a_i - a_{i,max}|, |a_i - a_{i,min}| * b_i$ when only considering types of cheese j with $b_i \leq b_j$.

A full c++ implementation of the algorithm, which we didn't ask you to do, would look something like this:

```
struct cheese {
  int a, b;
};

int main() {
  vector<cheese> k = read_input();
  int n = k.size();

  sort(begin(k),end(k),
       [](struct cheese x, struct cheese y) { return x.b < y.b; });

  int max_a, min_a;
  int max_score = 0;
  max_a = min_a = k.back().a;

  for(int i = n-2; i >= 0; i--) {
    int score = k[i].b * max(abs(k[i].a - max_a),abs(k[i].a - min_a));
    max_score = max(max_score, score);
    max_a = max(max_a, k[i].a);
    min_a = min(min_a, k[i].a);
  }
  cout << "Maximal Score: " << max_score << "\n";
}
```

Subtask 3: (20 points)

Now the characteristic b can also be negative. The maximal recommendation score will never be smaller than 0 since $n \geq 1$ and we can always combine a type of cheese with itself, which gives a recommendation score of 0.

For the score to have a positive sign, b_i and b_j have to have the same sign. We can split the types of cheese in two groups, the ones with $b_i < 0$ and the ones with $b_i \geq 0$. The two types of cheese that have together the highest score must then be in the same group.

We already found an algorithm that solves the problem for the group with $b_i \geq 0$ in the previous subtask.

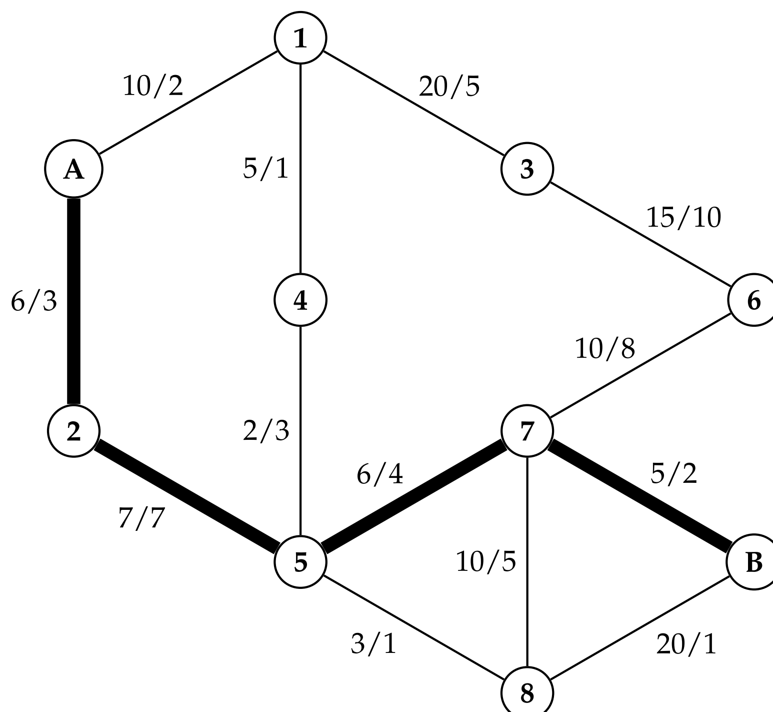
In the other group we can simply switch the sign for each b. The score will not be affected by this change because the negative signs cancelled out in the numerator and only appear in absolute values in the denominator. With this change we can again use the algorithm from the previous subtask.

The solution is now the bigger score out of the two groups. The algorithm is correct because we showed that the pair with the highest score is inside a group and we output the maximum score out of any score occuring in a group.

Pomegranates

Task Idea	Johannes Kapfhammer
Task Preparation	Bibin Muttappillil
Description English	Bibin Muttappillil
Description German	Bibin Muttappillil
Description French	N/A
Solution	Bibin Muttappillil
Correction	Bibin Muttappillil and others

Subtask 1: (10 Points)



The solution is $W = 5$, and a path would be $A \to 2 \to 5 \to 7 \to B$. If W could be greater than 5, the truck could not drive on any roads where $w \leq 5$ holds, so you could delete those edges. After the deletion, the shortest path from A to B would be $A \to 2 \to 5 \to 7 \to 8 \to B$, which takes longer than $T = 18$ time, therefore it wouldn't be possible.

Subtask 2: (30 Points)

The first observation required is that the optimal weight is one of the weight limits of an edge. Assume the optimal weight W' would be between the weight limits w_i and w_j, where there is no other weight limit between w_i and w_j and $w_i \leq W' \leq w_j$. If you

instead filled the truck with pomegranates until the weight of w_j you could still use all of the edges from before, since there is no edge between w_j and W', and have higher weight. Henceforth it wouldn't be optimal.

Based on that there are two solutions.

The first solution is to binary search for the answer W. We can check if A and B are connected by edges having a weight limit of at least W using DFS (depth-first search) or BFS (breath-first search) by just looking at the edges that have $w \geq W$. Clearly, if A and B are connected for some W, they are also connected for some smaller W. This montonicity property allows us to apply binary search on W, leading to a solution with a running time of $O(m \log m)$.

Another solution, which is nicer but more difficult to prove, is to look at the minimal spanning tree. Or, rather the *maximal* spanning tree of the graph. In that spanning tree, call it T, there is a unique path P from A to B and we claim that this is the path with the largest minimum weight. Why? Assume there is another path P' that is strictly better, and further assume this path is the best possible. That path must use some edge e that is not in the spanning tree (otherwise it would be the same or worse as P). Say this edge e connects vertices u and v. Let's look at the unique path that connect u and v in the maximal spanning tree T, and more specifically at the edge e' of that path that has the minimum weight. If the weight of e' is larger, we could replace e with the path from u to v in T and get a better solution than P' (contradiction to P' being optimal). If the weight of e' is equal we would get an equal solution (contradiction to P' being strictly better than P). Otherwise, if the weight of e' is smaller, we can remove e' from T and add e to it, leading to a better maximal spanning tree (contradiction to T being maximal). Thus, we have a contradiction, so P was optimal.

To calculate the maximal spanning tree, we can use a modified Kruskal's Algorithm, where we sort the edges from highest weight limit to lowest. Why does this work? Let's take two disjoint subgraphs G_1 and G_2 and look at all the edges that connect G_1 and G_2. For any spanning tree in G_1 and for any in G_2 you want to take the edge with the greatest weight limit, that connects those two spanning trees. The spanning trees in G_1 and G_2 are independent from our choice, so the gready aproach from Kruskal still works. From this argumentation also follows, that we only need to do the algorithm until A and B are connected (since any further edges won't be part of the path from A to B). Also all edges before the connection of A and B were smaller or equal to the edge that connected them, which lies on the only path from A to B. So the answer will just be the weight limit w of the last edge added in Kruskal's Algorithm to connect A and B.

Note: Prim's Algorithm also works.

Let n be the number of vertices and m the number of edges. The vertices, which are not in the component of A and B are irrelevant. Therefore we can assume that $m >= n - 1$ and conclude $n \leq O(m)$. Also: $m \leq O(n^2) \rightarrow O(\log(m)) = O(\log(n^2)) = O(\log(n))$.

Runtime:

- binary search W: $O(m \cdot \log(m))$
- run modified Kruskal: $O(m \cdot \log(m))$

$\rightarrow O(m \cdot \log(m)) = O(m \cdot \log(n))$

Note: With Fibonacci heap you can do Dijkstra, and therefore Prim, in $O(n \cdot \log(n) + m)$ time.

Memory:

- binary search W: $O(m)$
- modified Kruskal: $O(n + m) = O(m)$

$\rightarrow O(m)$

Subtask 3: (20 Points)

It is given that every weight limit of an edge is either 1 or 2. So If the truck weighs $W = 1$ it is guaranteed to work, and $W = 3$ can't work. So we only need to test if W could be 2. If $W = 2$ the truck can't use edges with $w = 1$, but every other edges is available. So we look at the modified graph G' where we delete all edges with $w = 1$, so G' only has edges with $w = 2$. So we need to test whether we can reach the contest place B from the warehouse A in time T. This problem can be solved by using Dijkstra's algorithm (because there are no negative distances) from A. This gives us the shortest distance d from A to B. If $d \leq T$ we can make it with $W = 2$, otherwise we only manage $W = 1$.

Runtime:

- build modified graph: $O(n + m) = O(m)$
- Dijkstra: $O(n + m \cdot \log(m)) = O(m \cdot \log(n)$

$\rightarrow O(m \cdot \log(n))$

Note: With Fibonacci heap you can do Dijkstra in $O(n \cdot \log(n) + m)$ time.

Memory:

- build modified graph: $O(n + m) = O(m)$
- Dijkstra: $O(n + m) = O(m)$

$\rightarrow O(m)$

Subtask 4: (40 Points)

Let's say we want to test if the weight W' works for the truck. We use the same idea as in subtask 3: We delete all edges where $w < W'$, the edges where a truck with weigth W' can't drive. The remaining Graph G' will have edges where $W' \leq w$, so the truck could drive on any of them. Now we do dijkstra on the modified graph G', which will calculate the shortest distance d from A to B. If $d \leq T$ we could fill it with pomegranates up to the point where the weight of the truck is W' and we would manage to make it in time to the contest location.

If you can do it with a weight of W', you can also do it with less weight. If you can't do it with the weight of W', you can't make it with more. As a result of this property, we

can binary search for the maximal W' where the truck can weigh W' and still manage to make it in time T from A to B.

So we binary search over the set of weights, and for a given weight W', we can do the algorithm from above to test if you can do it with this weight.

Runtime:

- binary search over set of weights: $O(\log(m)) = O(\log(n))$
- build modified graph: $O(n + m) = O(m)$
- Dijkstra: $O(n + m \cdot \log(m)) = O(m \cdot \log(n))$

$\rightarrow O(m \cdot \log(n)^2)$

Note: With Fibonacci heap you can do Dijkstra in $O(n \cdot \log(n) + m)$ time.

Memory:

- binary search $O(1)$
- build modified graph: $O(n + m) = O(m)$
- Dijkstra: $O(n + m) = O(m)$

$\rightarrow O(m)$

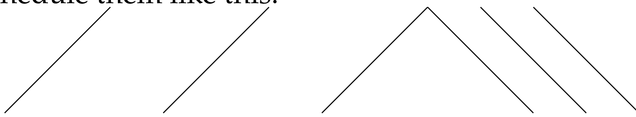
Mine Carts

Task Idea	Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Bibin Muttappillil
Description French	Florian Gatignon
Solution	Johannes Kapfhammer
Correction	Stefanie Zbinden and others

Subtask 1: Busting Stofl's Algorithm (15 Points)

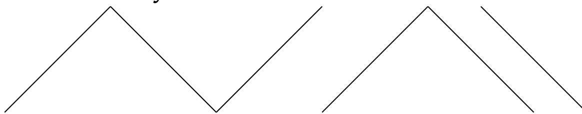
The algorithm runs in $O(n)$ time and uses $O(1)$ memory. The array t does not count for the memory usage, as we've explained in the analysis section.

A possible counterexample: $n = 2$, $d = 2$, $t_0 = 0$, $t_1 = 3$, $t_2 = 6$. The algorithm will schedule them like this:



In this diagram, the time goes from left to right. Station A is represented by points at the bottom, station B is represented by points at the top. The values of the formal description of this task would be $a = [0, 3, 6]$ and $b = [8, 9, 10]$.

A better way to schedule the mine carts are like this:



This corresponds to the values $a = [0, 4, 6]$ and $b = [2, 8, 9]$ and finishes one second earlier.

Subtask 2: Optimal Algorithm (85 Points)

There are two very different algorithms that solve this problem optimally. We start with an idea related to dynamic programming. This ultimatively leads to an algorithm in $O(n)$ running time and $O(\log n)$ space usage. In the end, we look at a binary search/greedy type algorithm which is slightly worse but much easier to explain and runs in $O(n \log n)$ time and $O(1)$ space.

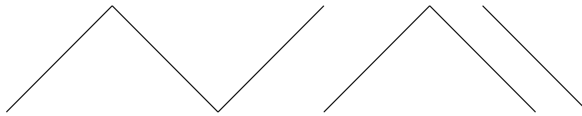
General observations

The first observation, which wasn't made in the task statement, is that without loss of generality we can assume that $a_0 < \ldots, a_{n-1}$ and $b_0 < \ldots, b_{n-1}$. This justifies that in the diagram we didn't write which line corresponds to which ore.

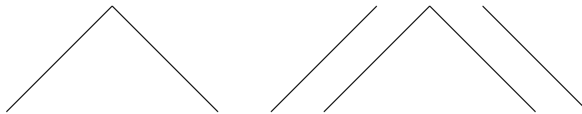
Also, we can assume that there is no b_j that lies between a_i and a_{i+1}, i.e. there is an i such that there is no j with $a_i < b_j < a_{j+1}$. Speaking in terms of the diagram, two lines

next to each other are separated can be put as close as possible.

So whenever an optimal solution looks like this:



We can always transform it into something like this:



DP in $O(n^3)$

Based on those two observations, we can formulate our first DP.

We define the state (i, j) as having sent i mine carts to B (and load off their load) and j of them being back to A.

We can apply dynamic DP programming since an optimal solution for state (i, j) can be built by a solution of either state $(i-k, j)$ or $(i, j-k)$, depending on whether we send our carts to A or to B.

$$\begin{aligned}
\texttt{DP}[i][j] &= \text{earliest time to reach state } (i, j) \\
&= \min\left(\min_{0\le k\le i-j} \text{ToA}(i-k, j, k), \min_{0\le k\le j} \text{ToB}(i, j-k, k)\right) \\
\text{ToA}(i, j, k) &= \text{earliest time to go from state } (i, j) \text{ to state } (i+k, j) \\
&= \max(\texttt{DP}[i][j] + d + k - 1, t[a] + d) \\
\text{ToB}(i, j, k) &= \text{earliest time to go from state } (i, j) \text{ to state } (i, j+k) \\
&= \texttt{DP}[i][j] + d + k - 1
\end{aligned}$$

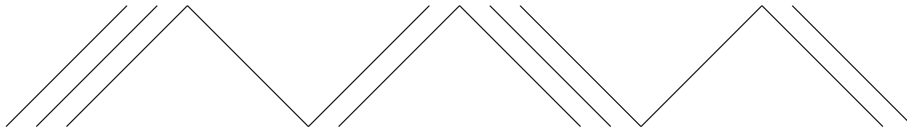
The max in ToA is because we can only go to B once the ore is available.

This dynamic programming approach has $O(n^2)$ states and runs in $O(n^3)$ time. Using some technical tricks it can be optimized to $O(n^2)$ time, but we won't go into details here.

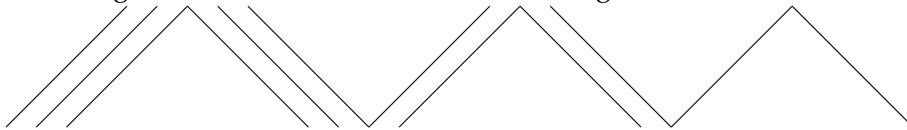
DP in $O(n^2)$

The best way to speed up a DP is to make some observations. We note the following: If, in some optimal solution, we send back one cart (from B to A), we might as well send back all carts that are waiting. Why? Well, they will go back at some point anyways. Returning them earlier does not cost us anything more, we just move back the starting times of the other carts.

As a diagram, if we have something like this:



we might as well transform it into something like this:



Let's call those balanced groups of trips a "block". The second image consists of 3 blocks, the first has size 3, the second size 2 and the third size 1.

The claim from above is that there is a solution that can be described as a sequence of such blocks, and furthermore each prefix of blocks is itself an optimal solution for a prefix of the input.

This allows us to formulate the following DP:

$$\begin{aligned}\texttt{DP}[i] &= \text{earliest time to bring } i \text{ carts to A and back to B.}\\ &= \text{best } k \text{ where the last block has size } i-k\\ &= \min_{0\le k\le i}(\max(\texttt{DP}[k]+i-k, t[i])+i-k+2d)\end{aligned}$$

The last trip to B will start at $\max(\texttt{DP}[k]+i-k, t[i])$, because either we are limited by the time when it will become available or we have to wait until the first $\texttt{DP}[k]$ carts are finished and we've sent our batch in $i-k$ seconds.

DP in $O(n \log n)$

Before we optimize this further, we need some more definitions.

$$\begin{aligned}\mathrm{f}[i][k] &= \texttt{DP}[k]+2i-2k+2d\\ \mathrm{g}[i][k] &= t[i]+i-k+2d\\ \texttt{cost}[i][k] &= \max(\mathrm{f}[i][k], \mathrm{g}[i][k])\\ \texttt{DP}[i] &= \min_{0\le k\le i}\texttt{cost}[i][k]\end{aligned}$$

Let's fix i and look at what happens when we increase k. $\mathrm{g}[i][k]$ is strictly decreasing. On the other hand, $\mathrm{f}[i][k]$ is non-decreasing, as $\texttt{DP}[k]-2k$ is non-decreasing by induction on the following inequality: $\texttt{DP}[i] \le \texttt{DP}[i+1]-2$. Proof: Given any solution for the first $i+1$ mine carts, we can just remove the two trips of the last mine cart and move the other return trips forwards. This gives us a solution for the first i mine carts that is at least 2 seconds better (but there may be an even better solution).

Therefore, for fixed i, $\texttt{cost}[i][k]$ is first decreasing (because g dominates) and then non-decreasing (because f dominates).

Sounds like a job for binary search! And in fact, this gives us a solution in $O(n \log n)$.[1]

[1] We can binary search on the slope: If $\texttt{cost}[i][k] \le \texttt{cost}[i][k+1]$ we go to the left, otherwise to the right.

DP in $\mathcal{O}(n)$

We can do something even smarter than binary search. We know the minimum is right at the point where the maximum changes from g to f.

So let's define k_i as that point:

$$k_i = \max\{k \in \{0, \dots, i\} \mid \mathrm{f}(i,k) \leq \mathrm{g}(i,k)\}$$

We now show that $\mathtt{DP}[i] = \mathtt{cost}[i][k_i]$. We look at the two cases where k is smaller or larger than k_i. For $k < k_i$, we have:

$$\mathtt{cost}[i][k] = g[i][k] > g[i][k_i] = \mathtt{cost}[i][k_i]$$

For $k > k_i$, we have:

$$\mathtt{cost}[i][k] = f[i][k] \geq f[i][k_i+1] \geq g[i][k_i+1]+1 = g[i][k_i] = \mathtt{cost}[i][k_i]$$

This is essentially the same argument as we gave for the binary search, but now more formal and showing that the right answer is in fact always a value of g.

As a last step, we want to show that $k_0 \leq k_1 \leq \dots k_{n-1}$. We show that from $\mathrm{f}[i][j] \leq \mathrm{g}[i][j]$ follows $\mathrm{f}[i+1][j] \leq \mathrm{g}[i+1][j]$, and therefore $k_{i+1} \geq k_i$ by definition of k_{i+1}:

$$\mathrm{g}[i][j]-\mathrm{f}[i][j] = t[i]-\mathtt{DP}[j]-(i-j) \leq t[i+1]-\mathtt{DP}[j-1]-(i-j-1) = \mathrm{g}[i+1][j]-\mathrm{f}[i+1][j]$$

Thus from $\mathrm{g}[i][j]-\mathrm{f}[i][j] \geq 0$ follows $\mathrm{g}[i+1][j]-\mathrm{f}[i+1][j] \geq 0$ which is what we wanted.

This can be translated into a very short algorithm: In our loop of the DP, we have an integer k. We increase that until $\mathrm{f}(i,k+1) > \mathrm{g}(i,k+1)$. Then we take the $\mathtt{cost}[i][k]$ as the optimum.

This algorithm runs in $\mathcal{O}(n)$ time and uses $\mathcal{O}(n)$ memory.

Binary Search/Greedy in $\mathcal{O}(n \log n)$

There's also a greedy algorithm that checks whether we can deliver all ores in time T in $\mathcal{O}(n)$. The algorithm works backwards (not forwards) and is based on the observation of blocks we've seen from before.

If the last ore gets mined at time $t[n-1]$ and have to be finished at time T, we know:

- There is an optimal solution that sends the last cart to B at time $t[n-1]$. Why not later? Because if this time is blocked, there is some mine cart returning to A at time $\leq t[n-1]$. Take the latest of those returning earlier than $t[n-1]$. We can make that mine cart return after the last mine cart has arrived at B and shift everything after it forward by one second (this must be valid because otherwise the time $t[n-1]$ would not be blocked), resulting in an equally good solution.

In general, being non-increasing and then non-decreasing is not enough for binary search as if we don't know in which direction we should go if the function is flat. But in this case the left side is strictly increasing, so equality means that we should go left.

Algorithm: Linear Solution for Mine Carts

```
input : n, d and the array t[0], ..., t[n − 1]
output: The minimal T

1 k = 0
2 DP[0] = 0
3 for i = 0 to n − 1 do
4 |   def cost(x) = max(DP[x] + i − x, t[i]) + i − x + 2 · d
5 |   while k + 1 ≤ i and cost(k + 1) < cost(k) do
6 |   |   k = k + 1
7 |   DP[i + 1] = cost(k)
8 return DP[n]
```

- The last cart will arrive at B in time $t[n-1]+d$. Between then and $T-d$ we can send mine carts back to A, so we have up to $t[n-1]-2\cdot d+1$ spots available.
- Therefore without loss of generality the last block will have size $k := T-2\cdot d+1$ and starts at time $t[i]-k+1$. If $k \leq 0$, there is no way to send the last cart back and thus there is no solution.

This way of selecting the blocks is optimal by the argumentation from the previous section.

Thus we have reduced the problem to checking whether there is a solution for some prefix of the input, and we can do so iteratively.

Algorithm: Greedy check whether there is a solution with time T

```
input : n, d, the array t[0], ..., t[n − 1] and a deadline T
output: true if it's possible or false otherwise

1 i = n − 1
2 while i ≥ 0 do
3 |   k = T − t[i] − 2 · d + 1        // how large can we make the block?
4 |   if k ≤ 0 then
5 |   |   return false                // can't deliver ore i in time
6 |   T = t[i] − k + 1    // the block starts k − 1 seconds before we send ore i
7 |   i = i − k           // we finished the last k ores
8 return true
```

Why can we do binary search on T? Well, if it's possible in time T, it's also possible in time $T+1$. If it's impossible in time T, it's also impossible in time $T-1$. So we're able to binary search on T.

A lower bound on T is $t[n-1]+2\cdot d$ (this is the earliest time the last cart can arrive) and an upper bound is $t[n-1]+2\cdot d+n$ (achieved by a greedy algorithm that waits

until all carts have been sent). Thus the range of the binary search has size n, thus the total running time is $O(n \log n)$, while the space usage is $O(1)$.

This is slightly worse than the $O(n)$ time/$O(n)$ space algorithm shown before.

Light Show

Task Idea	Johannes Kapfhammer
Task Preparation	Florian Gatignon
Description English	Florian Gatignon
Description German	Bibin Muttappillil
Description French	Florian Gatignon
Solution	Florian Gatignon
Correction	Florian Gatignon

Subtask 1: Solve the example (8 points)

You could start the sequence by assigning 0 to all vertices. Then the following sequence of switches will then be a correct solution:

$$B\text{–}A\text{–}C\text{–}D\text{–}C\text{–}F\text{–}C\text{–}D\text{–}C\text{–}A\text{–}B\text{–}E\text{–}B\text{–}A\text{–}C\text{–}D\text{–}C$$

Subtask 2: Combine garlands (24 points)

Since the distinct connected components are totally independent, you just have to find an efficient way of combining each possible state of each component with each possible state of every other component. Since there is no edge between connected components, this can't create any issue with the conditions which are fixed by the labels of the edges.

An intuitive way to do this with two components is as follows: let $a_0, \ldots, a_n$ be the switches for the first component and $b_0, \ldots, b_m$ for the second one. Then we first do all the switches from a_0 to a_n (while storing them in a doubly linked list), and then we switch b_0. Then we start again the switches from the first component, but in reverse order (taking them from the stored list), and add b_1 afterwards. We continue until we get to b_n and go through the switches of the first component a last time[1]. It is clear that this ordering is correct: there are no inconsistencies with the labels (as explained above) and it is complete, since we repeat all of the first component's switches each time we switch a vertex from the second one.

It is easy to use the same idea to manage more than two components by storing the direction for every component and going through them to know which one to advance each time. If you add a third component, you have to go through all the states of the two first ones before making its first switch, so you just use the same approach on another level and continue recursively as you add all connected components. But in that case, our worst case for *next* would be $\Theta(c)$, where c is the number of components. It is possible to do this in $O(1)$ in the worst case in a few steps.

[1] This ordering is called the "boustrophedon product" by Donald Knuth. Boustrophedon originally refers to a system of writing that alternates directions that was notably used in Archaic Greece. The name comes from the way a beef goes through a field to plough it, which can be a practical image to understand how we proceed here.

A first idea is to store which lists are "active". A list is active if and only if we are not at its last element. An easy way to check this in $O(1)$ is to advance the corresponding pointer and to revert it right afterward. Intuitively, we can store these states as a number with c bits. Let the state of the first component that we treat be represented by the rightmost bit. Then what *next* should do is just advance the rightmost active component and make all the components to its right active again. One can use the bit-tricks `x&(x-1)` to find the rightmost active component and `x|(x-1)` to set all of the trailing zeroes to one. If we get to the end of the switches of the component we advanced, we make it passive by setting its bit to 0 and we reverse the direction in which we traverse it for later.

This is not truly $O(1)$ yet, because we've used c-bits numbers. But there is another way to store this number, which is enabled by the fact that we only need two very specific operations: we use a stack of pairs of ints storing ranges of ones.

An example of how to do so follows (*push* adds trailing ones to the number and *pop* removes the rightmost one, turns all bits to its right to ones, and returns it):

```
def push(k):
  stack.push((0,k))
def pop():
  l, r = stack.pop()
  order[l] = not order[l] # change direction for l
  if r-l == 1: return l
  stack.push((l+1, r))
  return l
```

Combining this system with the rest of the solution guarantees a runtime of $O(n)$ for *start* and $O(1)$ for *next*.

Subtask 3: Individual garlands (68 points)

We first look at the solution for trees of which you're given the root r and with edges labeled such that the parent is always '$\geq$' its child. It will be useful for the full solution.

In that case, the solution is very similar to that of the second subtask. We always start with only zeroes. We can take a recursive approach for the switches. It is trivial that the sequence of switches for a single vertex v is just v itself. If we have a vertex v and we know the switches for all of its children, then then the solution is to first switch v and then use the same algorithm as in the second subtask to combine the solutions of the children.

Here, again, we need to use some optimization tricks to get to $O(1)$ for *next*. By just reusing the algorithm from the previous subtask, we get $O(d)$ with d being the depth of the tree.

The idea is to do an implicit depth-first search. We store for each vertex which is its closest active ancestor. We also store which is the currently active vertex at all times (the equivalent of the rightmost 1-bit in our algorithm for the second subtask). If we're done with the switches of a vertex, we go to its closest active ancestor using the stored pointer

and go down one level again. We go down only one level and going up is precomputed. To determine the next child that we need to push, we use the code from the previous subtask and this enables us to run *next* in $O(1)$ in the worst case.

Now we go to the problem for arbitrary trees. For any vertex v, if v has value 0, then all vertices $\leq v$ must have value 0, and if v has value 1, then all vertices $\geq v$ must have value 1 as well. Clearly, we have to treat both groups seperately: we begin with v as 0, do all the combinations of the vertices $\geq v$, then switch v to 1 and do all the combinations of the vertices $\leq v$. Then there is just one problem left: while we can do the same ordering as usual for the $\leq$ vertices, we have to end the combinations of the $\geq$ vertices with all of these vertices having value 1 in order to be authorized to switch v on.

That means we have to find an initial state that allows to do that. To do that in $O(n)$ is quite complicated. We first compute the parity of number of times we have to switch each vertex in total. This is easily computable by using this formula (modulo 2):

```
def pathlength(v):
  return (product((pathlength(w)+1) for w >= v) + 1 +
    product((pathlength(w)+1) for w <= v) + 1) - 1
```

Then we can use this and DFS to compute the initial state. Our DFS function takes two arguments: the current vertex and a boolean flag. We start by calling it for v with the flag as false. Then we recurse into the children of v that are $\geq$, which is easy. Then we also recurse to the children that are $\leq$. Here we need to know the product k of the pathlengths for each left sibling of the studied vertex. If k is even, then we set the vertex's initial state to 0, because we will iterate $k + 1$ times through its states until it reaches a final value of 1, and recurse the DFS into its children. Else, we initialize it with 1 and switch the flag on for the further recursion. When the flag is on, we do the same things but inverting $\leq$ and $\geq$. This allows us to treat that subtree as though its root was initialized with 0 instead of 1, so that the recursion can continue. With that done, we can get the initial state in $O(n)$ (we're doing DFS on a tree with n vertices) and we just need to iterate through the states as before.