SOI 2P 2019

Solution Booklet

Swiss Olympiad in Informatics

15–18 March 2019

Swiss Olympiad in Informatics

Round 2P, 2019

Stollen cut

Task Idea	Johannes Kapfhammer
Task Preparation	Daniel Rutschmann
Description English	Simon Meinhard
Description German	Joël Mathys
Description French	Yunshu Ouyang
Solution	Simon Meinhard

Subtask 1: $k = 1$

20 points

We can try to cut the stollen at every possible place and check what deliciousness difference we get. To compute the deliciousness of both portions, loop over both subarrays. Looping over both portions takes $O(n)$ time, and there are $O(n)$ possible ways to cut the stollen. Thus this solution takes $O(n) \cdot O(n) = O(n^2)$ which is sufficiently fast for $n \leq 2000$. The correctness follows from the fact that we examine every possible cut.

40 points

The previous solution will be too slow for $n \leq 10^6$. We consider two ways to improve the previous algorithm.

- We reduce the time it takes to check the deliciousness difference for every possible cut. We will see two different ways to achieve this.
- We reduce the number of cuts we need to check.

First approach

For the first approach, say that we have the deliciousness values $d_1, d_2, \cdots, d_k, d_{k+1}, \cdots, d_n$. Furthermore assume that we already cut the stollen after the k-th part and that the deliciousness of the portions are a (i.e. $\sum_{i=1}^{k} d_i = a$) and b (i.e. $\sum_{i=k+1}^{n} d_i = b$). What can we say if we cut the stollen after the $k+1$-th part? Clearly, we must add part $k+1$ to the left portion, and remove part $k+1$ from the right portion. The new values are thus $a + d_{k+1}$ and $b - d_{k+1}$. This transition can be computed in constant time. How can we design an algorithm based on this observation? Once we precomputed the values when we cut the stollen just before the first part (the first mouse gets nothing), we can compute all deliciousness differences using the transition described above. As we move the cut $O(n)$ times to the right, this takes $O(n) \cdot O(1) = O(n)$ time. The only thing that we still need to think of is to compute the values for the leftmost cut. This can be easily done in linear time by looping over the array once. The total running time is therefore $O(n) + O(n) = O(n)$. The correctness follows because we examine every possible cut.

Alternatively, we can use a technique known as prefix sums. The idea is that we are given values $a_1, a_2, \cdots, a_n$ and we receive some queries (l, r) where we have to compute

$a_l + a_{l+1} + \cdots + a_r$. We can easily answer every query in linear time by simply looping over the subarray. The key idea to make this faster is to precompute some data before the first query, the so called prefix sum array $s_0, s_1, \cdots, s_n$ where $s_0 = 0, s_1 = a_1, s_2 = a_1 + a_2, \cdots$, and $s_n = a_1 + a_2 + \cdots + a_n$. Here, s_i contains the sums of all values of array a up to index i. Now, the result for a query (l, r) becomes $s_r - r_{l-1}$. This is optimal as it takes constant time. The prefix sum array can be computed in linear time by noticing that $s_i = s_{i-1} + a_i$ for $1 \leq i \leq n$ and $s_0 = 0$. Using this technique, we can simply precompute the prefix sum array for the deliciousness values in $O(n)$ time, loop over all $O(n)$ possible cuts and compute the deliciousness difference in constant time using the prefix sum technique, and finally return the minimum absolute difference. The algorithm takes $O(n) + O(n) \cdot O(1) = O(n)$ time. The correctness follows as we examine every possible cut.

It is possible to generalize the prefix sum technique to multiple dimension. Can you figure out how[1]?

Second approach

We first observe that when we move the cut to the right, the deliciousness of the left portion increases while the deliciousness of the right portion decreases. The differences $a_1, a_2, \cdots, a_n$ of the left minus the right portion thus increase. As we need to minimize the absolute difference of deliciousness, it suffices to find the minimum (in terms of absolute value) of two values, the minimum non negative difference x^+ and the minimum (in terms of absolute value) negative difference x^- (if it exists, think of a stollen that has total deleciousness 0). As the values $a_1, a_2, \cdots, a_n$ are increasing, we can use binary search here to find x^+. Binary search takes $O(\log n)$ iterations. For every iteration, we need to compute the difference a_i. We can simply do this by looping over the array in linear time. To find x^-, we can use binary search as well or observe that when the differences are distinct, x^- and x^+ are next to each other[2]. Thus the algorithm takes $O(\log n) \cdot O(n) = O(n \log n)$. The correctness follows from the binary search algorithm.

To compute the difference a_i, we could have also used the prefix sum technique, which would have reduced the complexity to $O(n) + O(\log n) \cdot O(1) = O(n)$. However your algorithm needs at least $O(n)$ time to compute the prefix sum array, so you can simply loop over all possible cuts as described in the first approach and end up with the same time complexity and an easier implementation.

Subtask 2: $k = 2$

10 points

We proceed in a very similar way as for the first subtask. We loop over all $O(n^2)$ possible cuts and compute the three deliciousness values in linear time. The time complexity becomes $O(n^2) \cdot O(n) = O(n^3)$. The correctness follows from the fact that we examine every possible cut.

[1] The query time is not constant anymore and depends on the dimension.

[2] How can you deal with the more general case where the differences are not necessarely distinct?

30 points

We use a prefix sum array as described previously to compute the deliciousness values of portions in constant time. Using the same algorithm structure as for the previous subtask, the total running time is $O(n) + O(n^2) \cdot O(n) = O(n^2)$. The correctness follows from the fact that we examine every possible cut.

60 points

To solve this exercise, first consider a situation where you already determined the left portion, but you still have to split the right side of the stollen to get the middle and right portion. Let us denote the deliciousness of left, middle and right portion by p_1, p_2 and p_3 respectively. Two cases now arise.

- $p_2 \geq p_3$. Let us assume that if you remove the last portion of the middle part and include it in the right portion, we still have that the deliciousness of the middle part is at least as large as the deliciousness of the right part. If $p_1 \geq p_2$, then we get a better way to distribute the stollen between the mouses, as the maximum deliciousness stays the same (p_1) and the minimum deliciousness increases (p_3 plus the extra part). By a similar reasoning, it is easy to see that all other cases ($p_1 \leq p_3$ and $p_2 \geq p_1 \geq p_3$) we get a better stollen distribution as well. This implies that we only need to consider one distribution for the case where $p_2 \geq p_3$.
- $p_2 < p_3$. One can reason about this case in a very similar way, and conclude that while p_2 plus the first part of the right portion is smaller than p_3 minus the first part of the right portion, it is benificial to add the first part of the right portion to the middle portion.

Using this observation, we can use our binary search approach from the subtasks where $k = 1$. We loop over all possible $O(n)$ left portions, and then use binary search to find the best solution for both possible cases ($p_2 \geq p_3$ and $p_2 < p_3$). Using prefix sums as described above to accelerate binary search, we get $O(n) + O(n) \cdot \big(O(\log n) \cdot O(1)\big) = O(n \log n)$ for the time complexity, where the first term comes from the prefix sum array computation.

Brick tower

Task Idea	Joël Mathys
Task Preparation	Jan Schär
Description English	Joël Mathys
Description German	Jan Schär
Description French	Simon Meinhard
Solution	Jan Schär

Subtask 1: 20 Points ($x \leq 10^5$, guaranteed possible)

To solve the first test group, we can iterate through all tower widths $b < x$ starting at 2, and in an inner loop test all leftmost tower heights k from 1 to x. To test whether we have found the right parameters b and k, we can calculate the number of bricks used $kb + \frac{b(b-1)}{2}$ and compare with x. If they are equal, we output b.

This gives us the smallest possible b because we iterate from small to large b.

Running time is $O(x^2)$. Because it is guaranteed possible in this test group, and we stop as soon as we find the correct b, we can improve this bound to $O(x^{3/2})$.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main () {
  int x; cin >> x;
  for (int b = 2; b < x; b++) {
    for (int k = 1; k < x; k++) {
      int sum = k * b + b * (b - 1) / 2;
      if (sum == x) {
        cout << b << "\n";
        return 0;
      }
    }
  }
}
```

Subtask 2: 20 Points ($x \leq 10^{14}$, x odd)

This test group is very easy. If $x = 1$, then the answer is "`IMPOSSIBLE`", as we clearly can't make a tower wider than 1 with just 1 brick. For all other inputs, the answer is 2; the height of the tower is $\frac{x-1}{2}$ on the left and $\frac{x+1}{2}$ on the right.

We can combine this with the previous solution for 40 points.

Subtask 3: 60 Points ($x \leq 10^7$)

We can easily adjust the first solution to run in $O(x)$. Instead of testing every possible k, we can just try to calculate k. We can reformulate $x = kb + \frac{b(b-1)}{2}$ as $k = \frac{x - \frac{b(b-1)}{2}}{b}$. Because

k must be an integer, this just means testing if $x - \frac{b(b-1)}{2}$ is divisible by b.

```
#include <bits/stdc++.h>
using namespace std;
#define int int64_t
signed main () {
  int x; cin >> x;
  for (int b = 2; b < x; b++) {
      if ((x - b * (b - 1) / 2) % b == 0) {
        cout << b << "\n";
        return 0;
    }
  }
  cout << "IMPOSSIBLE\n";
}
```

Subtask 4: 80 Points ($x \leq 10^{14}$)

We can optimize the previous solution further if we observe that k can't be negative, so we only have to check the bs where $\frac{b(b-1)}{2} \leq x$.

This solution runs in $O(\sqrt{x})$.

```
#include <bits/stdc++.h>
using namespace std;
#define int int64_t
signed main () {
  int x; cin >> x;
  for (int b = 2; b < x && b * (b - 1) / 2 <= x; b++) {
      if ((x - b * (b - 1) / 2) % b == 0) {
        cout << b << "\n";
        return 0;
    }
  }
  cout << "IMPOSSIBLE\n";
}
```

Subtask 5: 100 Points ($x < 2^{63}$)

For the last test group, we need to do some more mathematical analysis of the problem. We want to solve $x = kb + \frac{b(b-1)}{2}$ with x known for the smallest possible b, where k and b are integers, $k > 0$ and $b > 1$.

We can rewrite the previous equation as $2x = b(2k + b - 1)$. From this we see that b is a factor of $2x$. We can also see that if b is even, then the other factor, $2k + b - 1$, is odd, and so b must in that case contain *all* the prime factors 2 in $2x$.

If x is a power of two, then we would have $2k + b - 1 = 1$, but this contradicts with $k > 0$ and $b > 1$, so the answer is "`IMPOSSIBLE`".

In all other cases, we can write x as $x = 2^a \cdot p \cdot r$, where p is the smallest odd prime factor of x and r is odd. Then we claim that the answer will be $b = \min(2^{a+1}, p)$.

Clearly, this satisfies $b > 1$ and b is an integer, and because of what we saw above there can't be a smaller solution. Now it only remains to prove that this solution satisfies the other conditions.

First, consider the case where $b = 2^{a+1}$. We have $prb = 2x = b(2k + b - 1)$ which leads to $k = \frac{pr-b+1}{2}$. We know that $pr \geq b$ (otherwise we would be in the other case), so k is positive. And because pr is odd and b even, k must be an integer.

Now for the case where $b = p$. Here we have $2^{a+1}rb = 2x = b(2k + b - 1)$ leading to $k = \frac{2^{a+1}r-b+1}{2}$, where again $2^{a+1}r \geq b$, so $k > 0$, and as b is odd k is an integer.

To use this result for writing a more efficient solution, we can first calculate 2^{a+1} and pr, and answer "`IMPOSSIBLE`" if $pr = 1$ (because then x is a power of two). Otherwise, we search for the smallest prime factor of pr. We only have to test up to $\sqrt{pr}$, because if there exists a factor greater than that but smaller than pr, then there also exists a prime factor smaller than $\sqrt{pr}$. We also only have to check numbers smaller than 2^{a+1}. pr could be a prime number itself, so we also have to consider it as a potential solution.

The running time of this solution is $O(x^{1/3})$. If i is the number of iterations, then we have $i < 2^{a+1}$ and $i < \sqrt{pr}$. Because $2x = 2^{a+1} \cdot \sqrt{pr} \cdot \sqrt{pr}$, i is maximized if both 2^{a+1} and $\sqrt{pr}$ are $(2x)^{1/3}$.

```cpp
#include <bits/stdc++.h>
using namespace std;
#define int int64_t
signed main () {
  int x;
  cin >> x;
  int pa = 2;
  while ((x & 1) == 0) { // test if x is even
    pa = pa << 1;
    x = x >> 1;
  }
  if (x == 1) {
    cout << "IMPOSSIBLE\n";
    return 0;
  }
  int b = min(pa, x);
  for (int i = 3; i < b && i * i <= x; i++) {
    if (x % i == 0) b = i;
  }
  cout << b << "\n";
}
```

Unique Coding

Task Idea	Johannes Kapfhammer
Task Preparation	Martin Raszyk
Description English	Martin Raszyk
Description German	Martin Raszyk
Description French	Simon Meinhard
Solution	Martin Raszyk

Subtask 1: 50 Points

To solve the first subtask, we can generate all K^N words of length N over the alphabet with K letters and check if their encodings using the given binary code match the encoded message from the input. Since the length of the encoded message equals M, the overall running time of this solution is $O(K^N \times M)$.

Subtask 2: 100 Points

To solve the second subtask, we use dynamic programming over prefixes of the encoded message and numbers of letters they should decode to. More formally, we compute the value $\texttt{DP}[p][l]$ for all $p = 1, \ldots, M$ and $l = 1, \ldots, N$ which is defined as follows: let enc denote the encoded message; then $\texttt{DP}[p][l] = 0$ iff $\texttt{enc}[1..p]$ cannot be decoded to l letters, $\texttt{DP}[p][l] = 1$ iff $\texttt{enc}[1..p]$ can be uniquely decoded to l letters, and $\texttt{DP}[p][l] = 2$ iff $\texttt{enc}[1..p]$ can be decoded to l letters in multiple ways.

We compute the values $\texttt{DP}[p][l]$ for increasing l. For $l = 1$, we can easily compute $\texttt{DP}[p][1]$ by checking if the prefix $\texttt{enc}[1..p]$ equals the encoding of a letter. In particular, we have $\texttt{DP}[p][1] \in \{0, 1\}$. For $l > 1$, we check for each letter s whose encoding is shorter than p (i.e., $|s| < p$) whether it occurs as a suffix of $\texttt{enc}[1..p]$, and then use the values $\texttt{DP}[p - |s|][l - 1]$, which we have computed already, to compute $\texttt{DP}[p][l]$. To reconstruct the decoded message, we also record the unique last letter s whenever $\texttt{DP}[p][l] = 1$.

If $\texttt{DP}[M][N] = 0$, we output "`not decodable`". If $\texttt{DP}[M][N] = 2$, we output "`not uniquely decodable`". Finally, if $\texttt{DP}[M][N] = 1$, we need to output the unique decoded message. We obtain it backwards as follows: we can read its last letter s from $back[M][N] = 1$, then we have $\texttt{DP}[M - |s|][N - 1] = 1$ and we can recursively compute the unique decoding of $\texttt{enc}[1..(M - |s|)]$ (i.e., the encoded message without the last letter s) to $N - 1$ letters. This way, we compute a letter from the decoded message in $O(1)$ and the whole messsage can be reconstructed in time $O(N)$.

Computing a single value $\texttt{DP}[p][l]$ can be performed in time $O(KM)$ (there are K letters to check and the encoding of each of them has length at most M). In total, there are $O(MN)$ values $\texttt{DP}[p][l]$ to compute which yields the overall running time in $O(NM^2K)$.

The overall space complexity of this solution is in $O(M(N + K))$.

Subtask 3: Even better solution

For an even better solution (which was not needed), we need to quickly find the letters whose encodings equal to a suffix of `enc`$[1..p]$, for all p. For a fixed letter s, we can find all p such that s is a suffix of `enc`$[1..p]$ using KMP in time $O(M)$. In total, this requires extra time and space both in $O(KM)$, but allows to compute a single value `DP`$[p][l]$ in time $O(K)$. Altogether, this brings the overall running time down to $O(NMK)$ and does not affect the asymptotic space complexity.

Skiing Resort

Task Idea	Monika Steinova
Task Preparation	Monika Steinova
Description English	Monika Steinova
Description German	Stefanie Zbinden
Description French	Florian Gatignon
Solution	Monika Steinova

We model the problem as a pair of directed graphs G, H, each on vertices 1 through n. Two vertices a,b are connected in G by a directed edge of length c if there is a cable car going from a to b that takes c minutes. Similarly, we add all ski-slopes from a to b lasting c minutes as directed edges in the graph H going from a to b with length c.

Let $d(p, q)$ denote the shortest directed path from p to q in G. Similarly, let $D(p, q)$ be the longest directed path from p to q in H. We are looking into a pair of vertices (p, q) such that the ratio

$$\frac{D(q, p)}{d(p, q)}$$

is maximised.

One possible approach is to check all ordered pairs (p, q), calculate the respective distances and find the optimal one. For the problem of finding the shortest path, we can use a variety of algorithms, e.g. Dijkstra, Bellman-Ford or Floyd-Warshall.

The problem of finding the longest path between two vertices p and q is, on general graphs, NP-hard, i.e., there is no efficient (polynomial) algorithm exists unless P = NP (which is mostly considered to be false). Luckily, our graph is rather special. Note that the ski slopes only go from a junction of higher elevation to a junction of a lower one. As such, there can be no directed cycles in the graph of ski slopes. This is called a directed acyclic graph (DAG). Directed acyclic graphs have many useful properties, and countless problems hard on general graphs are efficiently solvable on DAGs. Longest path is one of these problems. We outline three approaches in the order of decreasing time complexity.

Subtask 1: 25 + 25 Points ($n \leq 500$, $k, m \leq 500$)

One strategy to find the longest path is to use Floyd-Warshall's algorithm with a slight modification: initially all pairs of vertices not connected by an edge have a distance $-\infty$. In the innermost loop, max is used instead of min.

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    d[i][j] = inf
    D[i][j] = -inf
  }
}
```

```
for ((a,b,c) in G.edges) {
  d[a][b] = c
}

for ((a,b,c) in H.edges) {
  D[a][b] = c
}


for (int k = 0; k < n; ++k) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      d[i][j] = min(d[i][j], d[i][k] + d[k][j])
      D[i][j] = max(D[i][j], D[i][k] + D[k][j])
    }
  }
}
```

We use Floyd-Warshall for finding all $d(p, q)$ and the max-version of Floyd-Warshall for finding all $D(p, q)$ in $O(n^3)$. Then we find the optimal pair of junctions in $O(n^2)$.

Make sure to represent the ratio of skiing to riding cable car as a fraction (a pair of two integers), and not a floating point number, which may be imprecise. To check which of the two fractions is larger without using floating point arithmetic, we use the fact that

$$\frac{a}{b} \leq \frac{c}{d} \iff ad \leq bc\,.$$

Subtask 2: 25 Points ($n \leq 1000$, $k, m \leq 2000$)

In the previous section we used Floyd-Warshall that has cubic time complexity, which becomes too slow in the third subtask. The issue with Floyd-Warshall is that it is efficient only for dense graphs. For sparse graphs, more efficient methods exist, such as Dijkstra's algorithm. To adapt the Dijkstra's algorithm to finding the maximum distance, we can use a similar trick as in previous section, i.e., using maximum instead of minimum to update distances. Note that to select the vertex for processing, the one with the **smallest** distance from the source among the unprocessed ones is selected, as in the original Dijkstra's algorithm.

In a single run of Dijkstra's algorithm or the max-Dijkstra, we find all $d(p, q)$, $D(p, q)$, respectively, for a fixed p. We simply run the algorithm n times to calculate all $d(p, q)$ and $D(p, q)$ and proceed as before.

The total time complexity is $O(n(m + k) \log n)$.

Subtask 3: 25 Points ($n \leq 2000$, $k, m \leq 4000$)

We can also get rid of the logarithmic factor, again thanks to the fact our graph is a DAG. All DAGs have a property of admitting a *topological sort*. A topological sort is a relabelling of vertices to 1 through n, such that all edges are of format (a, b) where $a < b$. A DAG may have more than one topological sort, but finding one of them is a relatively simple problem, solved as follows:

```
for (int i = 1; i <= n; ++i) {
  select any vertex v with zero incoming edges
  labelling[v] = i
  remove vertex v and all outgoing edges from the graph
}
```

If we maintain a set or a queue of vertices with indegree 0, the above can be performed in $O(|E|)$, where E is the set of edges.

A topological sort can be leveraged to solve various problems on a DAG using dynamic programming, and shortest path is one of them. Let $d[i]$ be the distance from a fixed p to vertex i, initially $d[i] = \infty$ and $d[p] = 0$. We process the vertices in topological order, and upon processing vertex j, we update the shortest path to be

$$d[k] = \min(d[k], d[j] + c(j, k)),$$

where $c(j, k)$ is the length of the edge from j to k (if present).

This correctness of the algorithm can be proved by induction. It is trivially true for the source vertex. Assume that all vertices $i < k$ have already been processed and $d[i]$ is the shortest distance from source to i. As a result, $d[k]$ is the shortest distance from source to k.

Similar to what we did before, substituting ∞ with $-\infty$ and $\min(\cdot, \cdot)$ with $\max(\cdot, \cdot)$ gives an algorithm for the longest distance. This pair of dynamic programming solutions is run n times, once for each starting vertex. As the complexity of each run is $O(m + k)$, the overall complexity is $O(n(m + k))$.