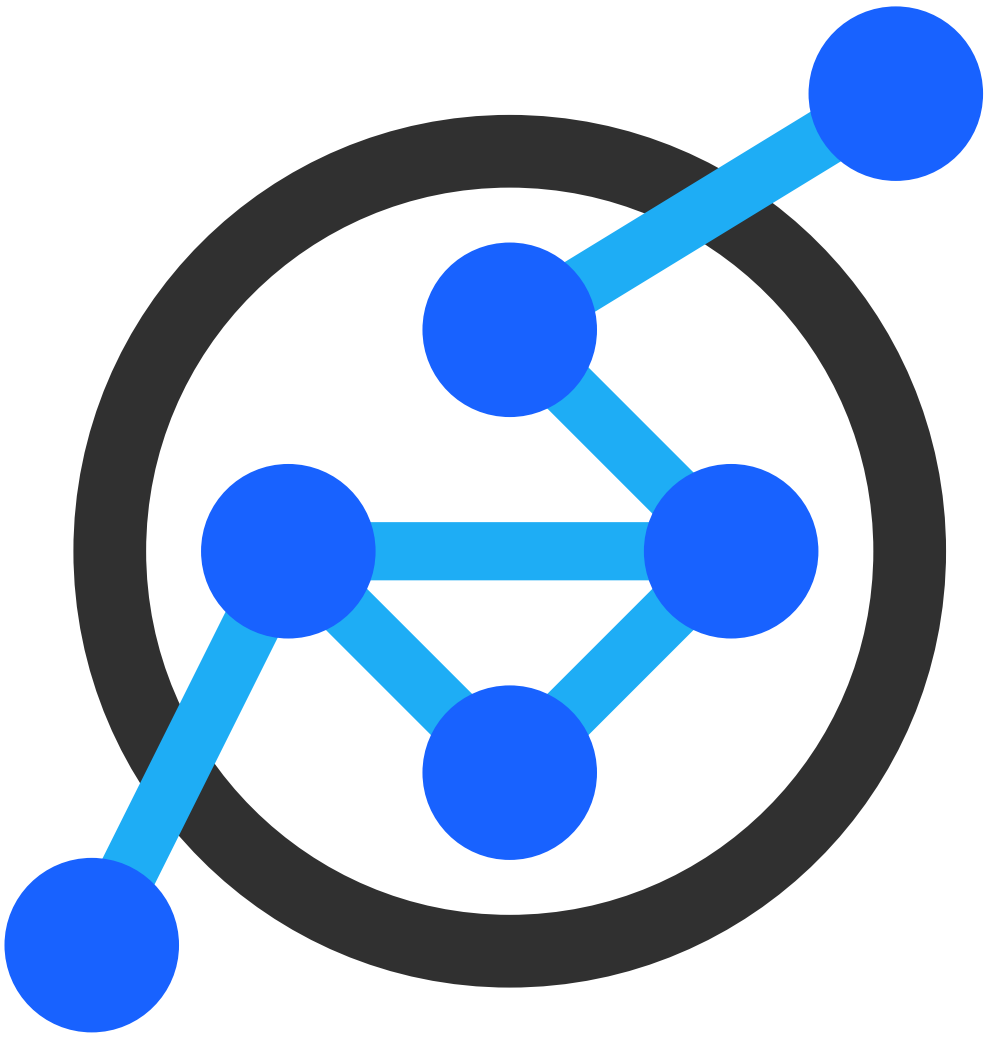
SOI 2P 2022

Solution Booklet



Swiss Olympiad in Informatics

10–14 March 2022

Candles

Task Idea	Johannes Kapfhammer
Task Preparation	Florian Gatignon
Description English	Florian Gatignon
Description German	Joël Huber
Description French	Florian Gatignon
Solution	Florian Gatignon

You are given N points with integer coordinates $x_0, \ldots, x_{N-1}$ on an axis, an integer K and a starting coordinate s. Your goal is to find the shortest way to start at s and visit K of those N points. To achieve this, you may need to go in one direction and then in the other direction, for example if some points are too far away or if there aren't K points on either side of s. You only had to output the length of the shortest way to do this.

Note, first, that it is always optimal to use a sequence of K points at coordinates $k_0 \leq \cdots \leq k_{K-1}$ such that among the other $N-K$ points, none has a coordinate strictly between k_0 and k_{K-1}. The reason is simple: otherwise, we could cut the total length of the path by using any such point instead of one of the points at k_0 or k_{K-1}. That means that we only have to look at $N-K+1$ candidate sets of K points, but to find those easily, we have to sort the coordinates $x_0, \cdots, x_{N-1}$, which takes $O(N \log N)$. This is the first step of our algorithm.

Assume, then, that the coordinates are now sorted, so $x_0 \leq \cdots \leq x_{N-1}$. The candidate sets of points are $\{x_0, \cdots, x_{K-1}\}, \{x_1, \cdots, x_K\}, \cdots, \{x_{N-K}, \cdots, x_{N-1}\}$. The key to making the algorithm fast enough to get a full score is to compute the distance for each of these sets in $O(1)$. Checking all possible sets would then take a total of $O(N)$, so the total runtime would still be $O(N \log N)$.

To compute the total distance for the set $\{x_i, \cdots, x_{i+K-1}\}$ in $O(1)$, it is useful to observe that this distance only depends on s, x_i and x_{i+K-1}. From s, we must travel to the closest of x_i and x_{i+K-1}. Then, we need to traverse the whole set. If $x_i < s < x_{i+K-1}$, we will travel twice between s and one extremity, but since it is the closest extremity from s, this distance is optimal for the given set (the distance travelled twice is the smallest possible, and we have to traverse the whole set anyway). Therefore, the optimal distance for that set is $\min(|x_i - s|, |x_{i+K-1} - s|) + x_{i+K-1} - x_i$. The best of those values for $0 \leq i <= N-K$ is the full solution.

The full algorithm is then as follows:

1. Read the input ($O(N)$).
2. Sort the coordinates ($O(N \log N)$.
3. Set the best answer yet to ∞ ($O(1)$).
4. For every $0 \leq i \leq N-K$, compute the answer for the set $\{x_i, \cdots, x_{i+K-1}\}$. If this is smaller than the current best answer yet, store it as the best answer yet (N times $O(1)$, so $O(N)$).
5. Print the best answer yet, which is the best answer overall ($O(1)$).

In total, the runtime is $O(N \log N)$. We use $O(N)$ storage for the input and $O(1)$ additional memory. A C++ implementation of this algorithm can be found on the next page.

```cpp
#include <soi>
signed main() {
    // getting the input //
    int N = read_int(), K = read_int(), s = read_int();
    vector<int> x(N);
    for(int i = 0; i < N; i++)
        x[i] = read_int();
    sort(x.begin(), x.end()); // O(N*log(N))
    int sol = 1e15; // ~ infinity
    // solving the task //
    for(int i = 0; i+K<=N; i++) { // go through all candidate K-length segments
        // x[i+K-1]-x[i]: total length of the segment
        // min(abs(s-x[i]),abs(s-x[i+K-1])): shortest way to an extremity of the segment
        int candidate = min(abs(s-x[i]),abs(s-x[i+K-1]))+x[i+K-1]-x[i];
        sol = min(sol, candidate);
    }
    // printing the result //
    print(sol);
}
```

Order of Departure

Task Idea	Johannes Kapfhammer, Timon Gehr
Task Preparation	Timon Gehr
Description English	Timon Gehr
Description German	Timon Gehr
Description French	Mathieu Zufferey
Solution	Timon Gehr

Note that we can model the friendships as a graph, where vertices are mice and there is an edge between two mice if and only if they are friends. We have to order the vertices of the graph such that if we direct edges from vertices that are earlier in the order to vertices that are later in the order, each vertex has a specified in-degree (corresponding to the number of received cheese balls).

Subtask 1: Subtask 1 (30 points)

In subtask 1, $M = N \cdot (N-1)2$, which means that our graph is complete. This means the number of received cheese balls is precisely the number of mice that were already at the party before each mouse. Therefore, this number is also the (1-based) index of the unique order in which the mice could have arrived. We can therefore simply print the mice in this order:

```cpp
#include <bits/stdc++.h> // score: 30
using namespace std;

int main(){
        int n;
        cin>>n;
        vector<int> solution(n,-1);
        for(int i=0;i<n;i++){
                int cheese;
                cin>>cheese;
                solution[cheese-1]=i;
        }
        int m;
        cin>>m;
        for(int i=0;i<n;i++){
                cout<<solution[i]<<" \n"[i+1==n];
        }
}
```

This solution runs in $O(N)$ time.

Subtask 2: Subtask 2

In subtask 2, not all mice are necessarily friends, therefore we cannot directly read off the result anymore. However, we can make the following observations:

- There is always at least one mouse that received one cheese ball (the first mouse to arrive gets exactly one cheese ball, from Mouse Stofl).
- Whenever a mouse receives exactly one cheese ball, we can make it leave first. This is because it has to depart before any of its friends (otherwise it should have gotten a cheese ball from such a friend) and it does not influence the number of cheese balls obtained by any other mouse.
- Conversely, a mouse that obtained exactly one cheese ball must have given a cheese ball to all its friends (except Mouse Stofl).
- If we consider an instance where the first mouse to depart never attended the party at all, all its friends in the original instance obtain one fewer cheese ball.

A possible algorithm therefore works as follows:

It builds a valid departure order one element at a time. If the position of a mouse in the departure order has already been determined, we will say that that mouse *departed*. The other mice *are still partying*. You can imagine that Mouse Stofl sends them home one by one as he executes the algorithm. Mouse Stofl does not count as still partying, because he is busy sending people home and it simplifies exposition.

We will also imagine that we have in mind some specific possible departure order to which we can extend the current one. In the beginning, we can imagine we have an arbitrary possible order, as it is guaranteed to exist by the task statement. We may have to change it later to another possible solution to accommodate the actual decisions made by the algorithm.

We maintain the number of cheese balls that each mouse still at the party obtained from mice who have already departed.

In each step, the algorithm selects a mouse that should depart next. There is at least one mouse that has received no cheese ball from mice that are still partying (the first mouse in our imagined valid departure order). The algorithm picks any one of them and send it home. Possibly, there are multiple such mice and the algorithm did not pick the first mouse in our imagined valid departure order. In this case, we can change the imagined valid departure order by dragging this mouse to the first spot as well, maintaining the relative order of all other mice. This does not change the required number of cheese balls for any mouse: Some mice currently remaining at the party were scheduled to depart before our mouse. They cannot be friends with our mouse as it did not receive any cheese balls from them. The remaining mice maintain their relative order to all other mice. Therefore, the changed imagined departure order is still valid.

Whenever a mouse departs, we loop through all of its friends and reduce the number of cheese balls received from mice that are still at the party by one (as one of their friends just departed). (Technically, it's sufficient to only loop over friends that are still partying, but values for other mice do not matter anymore, so it's okay if they become negative.)

In the end, we have sent all mice home and their order of departure therefore matches our imagined valid departure order. Therefore, our solution is also valid.

We can implement this algorithm naively as follows:

```cpp
#include <bits/stdc++.h> // score: 50
using namespace std;

int main(){
        int n;
        cin>>n;
        vector<int> cheese(n);
        for(int i=0;i<n;i++){
                cin>>cheese[i];
                --cheese[i]; // ignore Stofl's cheese
        }
        int m;
        cin>>m;
        vector<vector<int>> g(n);
        for(int j=0;j<m;j++){
                int a,b;
                cin>>a>>b;
                g[a].push_back(b);
                g[b].push_back(a);
        }
        for(int i=0;i<n;i++){
                int current=-1;
                for(int k=0;k<n;k++){ // find mouse that got no cheese
                        if(cheese[k]==0)
                                current=k;
                }
                assert(current!=-1); // solution has to exist
                for(int next:g[current]) // reduce friend's cheese
                        --cheese[next];
                cheese[current]=-1; // no longer consider mouse
                cout<<current<<" \n"[i+1==n]; // send it home
```

```
32          }
33  }
```

This solution runs in time $O(N^2)$.

Subtask 3: Subtask 3

We can implement the same algorithm more cleverly: instead of checking all mice in each step, we maintain a queue with all mice that could depart now. Every time we reduce the number of cheese balls received from mice that are still partying, we can check if it hits zero. If so, we add it to the queue. Then in each step we can just pick a mouse that is ready to leave from the queue.

```
#include <bits/stdc++.h> // score: 70
using namespace std;

int main(){
        int n;
        cin>>n;
        vector<int> cheese(n);
        queue<int> ready;
        for(int i=0;i<n;i++){
                cin>>cheese[i];
                if(--cheese[i]==0){ // find mice that could leave first
                        ready.push(i);
                }
        }
        int m;
        cin>>m;
        vector<vector<int>> g(n);
        for(int j=0;j<m;j++){
                int a,b;
                cin>>a>>b;
                g[a].push_back(b);
                g[b].push_back(a);
        }
        for(int i=0;i<n;i++){
                assert(!ready.empty()); // solution has to exist
                int current=ready.front(); // find mouse that got no cheese
                ready.pop();
                for(int next:g[current]){ // reduce friend's cheese
                        if(--cheese[next]==0) // note new departure candidates
                                ready.push(next);
                }
                cheese[current]=-1; // no longer consider mouse
                cout<<current<<" \n"[i+1==n]; // send it home
        }
        assert(ready.empty());
}
```

This solution runs in time $O(N + M)$, because each friendship is only considered three times: once when it is read and once each when the involved mice depart.

Subtask 4: Subtask 4

Recall the only mice that can depart first at any given moment are those who have not received any cheese balls from mice who are still partying. To lexicographically minimize our solution, we first and foremost have to minimize the first element of the solution and then ensure the remainder of the solution is again lexicographically minimal among all possible ways to extend our solution. Therefore, it suffices to, among the mice that may depart, pick the one with the smallest index in each time step:

```cpp
#include <bits/stdc++.h> // score: 65
using namespace std;

int main(){
        int n;
        cin>>n;
        vector<int> cheese(n);
        for(int i=0;i<n;i++){
                cin>>cheese[i];
                --cheese[i]; // ignore Mouse Stofl's cheese
        }
        int m;
        cin>>m;
        vector<vector<int>> g(n);
        for(int j=0;j<m;j++){
                int a,b;
                cin>>a>>b;
                g[a].push_back(b);
                g[b].push_back(a);
        }
        for(int i=0;i<n;i++){
                int current=-1;
                for(int k=0;k<n;k++){ // find mouse that got no cheese
                        if(cheese[k]==0){
                                current=k;
                                break; // pick the one with smallest index
                        }
                }
                for(int next:g[current]) // reduce friend's cheese
                        --cheese[next];
                cheese[current]=-1; // no longer consider mouse
                cout<<current<<" \n"[i+1==n]; // send it home
        }
}
```

This solution runs in time $O(N^2)$.

Subtask 5: Subtask 5

We can use basically the same optimization we used to improve the solution to subtask 2 to be fast enough for subtask 3: We maintain a queue of all mice who could currently depart. From those, we have to always pick the one with lowest index, so we will in fact use a priority queue that is ordered by mouse index:

```cpp
#include <bits/stdc++.h> // score: 100
using namespace std;

int main(){
        int n;
        cin>>n;
        vector<int> cheese(n);
    // prioritize mice with small index:
        priority_queue<int,vector<int>,greater<int>> ready;
        for(int i=0;i<n;i++){
                cin>>cheese[i];
                if(--cheese[i]==0){ // find mice that could leave first
                        ready.push(i);
                }
        }
        int m;
        cin>>m;
        vector<vector<int>> g(n);
        for(int j=0;j<m;j++){
                int a,b;
                cin>>a>>b;
                g[a].push_back(b);
```

```
23                  g[b].push_back(a);
24          }
25          for(int i=0;i<n;i++){
26                  assert(!ready.empty()); // solution has to exist
27                  int current=ready.top(); // find smallest mouse that got no cheese
28                  ready.pop();
29                  for(int next:g[current]){ // reduce friend's cheese
30                          if(--cheese[next]==0) // note new departure candidates
31                                  ready.push(next);
32                  }
33                  cheese[current]=-1; // no longer consider mouse
34                  cout<<current<<" \n"[i+1==n]; // send it home
35          }
36          assert(ready.empty());
37  }
```

This solution runs in time $(N \log N + M)$, as we have to execute N push and pop operations on the priority queue.

Timber!

Task Idea	Joël Huber
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Timon Gehr
Description French	Florian Gatignon
Solution	Johannes Kapfhammer

This task may look tricky on first glance because in order to build a solution, you need to know the constraints (which is the answer: d), but you can't know that before having a solution.

Subtask 1: Binary Search

Luckily for us, we have *monotonicity*: if there is a solution for d, there is also a solution for $d-1$ (the solution for d is still valid for $d-1$). Conversly, if there is no solution for d there is also no solution for $d+1$ (strengthening the constraint does not help us finding a solution).

This means we can use binary search find the answer.

```cpp
#include <bits/stdc++.h>
#define int int64_t
using namespace std;

int best(vector<int> const& v, int d) {
  // compute best solution for fixed d
}

signed main() {
  cin.exceptions(ios::badbit | ios::eofbit | ios::failbit);
  ios::sync_with_stdio(false);
  size_t n, s; cin >> n >> s;
  vector<int> v;
  copy_n(istream_iterator<int>(cin), n, back_inserter(v));

  int l=0, r=n+1;
  while (r-l > 1) {
    int m = (l+r)/2;
    if (best(v, m) >= s)
      l = m;
    else
      r = m;
  }
  if (l == 0)
    cout << "Impossible\n";
  else if (l == n)
    cout << "Infinity\n";
  else
    cout << l << '\n';
}
```

It is important to handle the edge cases properly:

- If the sum of all values is smaller than s then the output should be "Impossible".
- And if the largest value is larger or equal to s then the answer is "Infinity".

The binary search above does that automatically. We have $l = 0$ as indicator for impossible (note that `best` is never called with 0, because m can only be 0 for $l = 0$ and $r = 1$, but then the binary search is done). Similarly, we set the right bound to $n + 1$ to indicate infinity. Also note here that `best` is never called with $n + 1$.

So what remains is coming up with strategies to find the best solution given d.

As so often with binary search, we change the problem statement slightly: we don't want to know *whether* it is possible for some d, but rather "what is the maximal amount of wood for some d" – and then compare that answer with s.

Subtask 2: Taking every d-th element (40 Points)

If the input is increasing, we can just take every d-th element, starting with the biggest one.

```
int best(vector<int> const& v, int d) {
    int ans = 0;
    for (int i=(int)v.size() - 1; i >= 0; i -= d)
        ans += v[i];
    return ans;
}
```

Subtask 3: DP (100 Points)

Fo the full solution you can do a DP approach:

$$\begin{aligned}\texttt{DP}[i] &= \text{max amount of wood we can take from } h_0, h_1, \ldots, h_i \\ &= \max \begin{cases} \texttt{DP}[i-1] & \text{if } i > 0 \quad \text{(ignore } h_i \text{ and take from 0 to } i-1\text{)} \\ \texttt{DP}[i-d] + h_i & \text{if } i \geq d \quad \text{(take } h_i \text{ and the best solution from } d \text{ trees before)} \\ h_i & \text{if } i < d \quad \text{(only take } h_i\text{, can't take anything else)} \end{cases}\end{aligned}$$

To avoid the case distinction or out of bound checks in the code one can make the DP table $n + d$ long.

```
int best(vector<int> const& v, int d) {
    vector<int> dp(n+d);
    for (int i=0; i<n; ++i)
        dp[i+d] = max(dp[i+d-1], dp[i] + v[i]);
    return dp.back();
}
```

The `best` function runs in $O(n)$ time and space, therefore the full solution with binary search around it runs in $O(n \log n)$ time and $O(n)$ space.

RPG Gameplay

Task Idea	Jan Schär, Petr Mitrichev
Task Preparation	Joël Huber
Description English	Joël Huber
Description German	Joël Huber
Description French	Florian Gatignon
Solution	Joël Huber

General Observations - Part 1

Let m be the maximal number of monsters Stofl can hit. Then the solution is going to be $n+1-m$. Thus it suffices to calculate the maximal number of monsters that Stofl can hit.

Observation 1: It is always optimal to hit a monster if an arm is available.

Proof: Suppose there is a an optimal solution that doesn't hit a monster even if an arm is available and consider the first occasion this happens, let it be monster i. Without loss of generality, the left arm is available. Then consider the next time the left arm is used and let it be monster j. Consider the solution where instead of monster j, we attack monster i. This new solution will hit the same number of monsters and thus is optimal as well. We can repeat this argument or use induction to do it formally to see that there is an optimal solution which always attacks when the arm is available.

Subtask 1, $r \longrightarrow \infty$

This subtask can be solved greedily: Consider only the left arm and forget the right arm for the moment. Then by observation 1, we can just greedily always attack the next possible monster. Then we have one "joker"-attack with the right arm, as we can attack with it exactly once. But note that the right arm can only hit an additional monster if the left arm didn't hit all of them.

```
from sys import stdin

n, l, r = map(int, stdin.readline().split())
pos = map(int, stdin.readline().split())

cr = 0
res = 0
for nx in pos:
    if cr <= nx:
        res += 1
        cr = nx + l

if res != n:
    res += 1

print(n + 1 - res)
```

Subtask 3, $n \leq 16$

Let M be the set of monsters and S be a subset of the monsters. Then we can check how many monsters we can defeat if the left arm only hits monsters in S and the right arm only monsters in $M \setminus S$. By applying the greedy strategy from earlier to both subsets, we can find the highest number of monsters we can beat respecting the partition. So we can just loop through all subsets S of M, checking every partition. This runs in $O(n2^n)$.

```python
def greedy(arr, d):
    cr = 0
    cres = 0
    for nx in arr:
        if cr <= nx:
            cres += 1
            cr = nx + d
    return cres

best = 0
for i in range(1<<n):
    a = []
    b = []
    for j in range(n):
        if i & (1<<j) > 0:
            a.append(pos[j])
        else:
            b.append(pos[j])
    best = max(best, greedy(a, l) + greedy(b, r))
print(n + 1 - best)
```

Subtask 4, $n \leq 3 \cdot 10^3$

Let's first precompute two arrays: $\text{arr}_l[i]$ is the next monster that can be attacked with the left arm if we attack monster i with it. Define $\text{arr}_r[i]$ analogously. arr_l and arr_r can be calculated using two pointers:

```python
def next_pos(n, pos, df):
    l, r = 0, 0
    arr = []
    for l in range(n + 1):
        while r < n and pos[r] - pos[l] < df:
            r += 1
        arr.append(r)
    return arr

arr_l = next_pos(n, pos, l)
arr_r = next_pos(n, pos, r)
```

Consider the following 2-dimensional dp: $\text{dp}[i][j]$ is the maximal number of monsters that can be defeated until the end if the next monster attacked by the left arm is the i-th monster or later, and the next monster attacked by the right arm is the j-th monster or later. Note that it might be a bit tricky to make sure that i and j don't attack the same monster: E. g. if you calculate $\text{dp}[i][j]$, it might depend on the state $\text{dp}[j][k]$ and then it can happen that it kills monster j both when calculating $\text{dp}[i][j]$ and when calculating $\text{dp}[j][k]$. However, if we do the transitions nicely, we can fix this:

```python
for i in range(n, -1, -1):
    for j in range(n, -1, -1):
        if (i < n):
            dp[i][j] = max(dp[i][j], dp[i + 1][j])
        if (j < n):
            dp[i][j] = max(dp[i][j], dp[i][j + 1])
        if (i < j):
            dp[i][j] = max(dp[i][j], dp[arr_l[i]][j] + 1)
        if (i > j):
            dp[i][j] = max(dp[i][j], dp[i][arr_r[j]] + 1)
```

Note that we always move the arm that is further behind. This dp takes $O(n^2)$ time and $O(n^2)$ memory. For subtask 5, this results in either RE or MLE, as it exceeds the memory limit. (And even if it passed the memory limit, it would probably get TLE as getting that much memory is slow). In the next section, we will now prove why this dp never attacks a monster twice.

General Observations - Part 2

Lemma 1: When walking through the states (i, j), if we only attack with the arm that is currently further behind, then we will never hit a monster twice.

Proof: Suppose monster with index i gets hit both by the left and the right arm. Without loss of generality, when going through the different states, we hit it with the left arm first. Look at the state (i, j) when we decide to hit monster i with the left arm. There are two cases: If $i < j$, then we can never hit the monster with index i with the right arm, as when we go continue going through the states, the position of the right arm will never decrease. If $i > j$, this is then a contradiction to the fact that we always attack with the arm that's further behind.

Note that this lemma doesn't allow us to attack a monster if $i = j$ holds (as no arm is further behind). However, if we want to attack this monster with let's say the left arm, then we can walk through the states like this: $(i, i) \to (i, i + 1) \to (\text{arr}_l[i], i + 1)$ to attack monster i.

General Observations - Part 3

Note that from observation 1, we can already build a semi-greedy strategy: Let s_l and s_r be the next monster that the left respectively the right arm can attack. Consider the following semi-greedy:

---

if $s_l < s_r$ **then**
  Attack s_l with the left arm
else if $s_l > s_r$ **then**
  Attack s_r with the right arm
else if $s_l = s_r$ **then**
  There are two things we could do:
  - Move s_r forward by 1 monster (and continue with the greedy)
  - Move s_l forward by 1 monster (and continue with the greedy)
  Try both options and take the one that kills more monsters.

By Lemma 1, we will never hit any monster twice when using this strategy.

Subtask 2, $l = r$

Consider the semi-greedy from earlier. Consider the third case ($s_l = s_r$). As in this case $l = r$, the two options we are trying out are the exactly the same, as the left and the right arm have the same cooldown. So we can just hit the monster with any arm and a simple greedy works again:

```cpp
#include <iostream>
#include <vector>

using namespace std;

signed main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int n;
    int l, r;
    cin >> n >> l >> r;

    int s_l = -1, s_r = -1;
    int res = n + 1;

    for (int i = 0; i < n; i++)
    {
        int x;
```

```
21          cin >> x;
22          if (s_l <= x) res--, s_l = x + l;
23          else if (s_r <= x) res--, s_r = x + r;
24      }
25      cout << res << "\n";
26  }
```

Note that this code also solves subtask 1, as it always uses the left arm when it's available.

Subtask 5, $n \leq 2 \cdot 10^4$

Consider the following dp: dp[i] is the maximal number of monsters that can be defeated if we only consider the monsters starting from i to the end. Note that we have both arms available to attack monster i. Then We can calculate $dp[i]$ by testing both possibilities of attacking it (left or right arm), and then apply the greedy algorithm as long as $s_l \neq s_r$. As soon as $s_l = s_r$ we can just take the value of dp[s_l].

```
pos.push_back((int)(2e9+100));

vector<int> arr_l = next_pos(n, pos, l);
vector<int> arr_r = next_pos(n, pos, r);

vector<int> dp(n + 1, 0);

auto check = [&](int s_l, int s_r)
{
    int res = 0;

    while (s_l != s_r)
    {
        res++;
        if (s_l < s_r) s_l = arr_l[s_l];
        else s_r = arr_r[s_r];
    }

    return res + dp[f];
};

for (int i = n - 1; i >= 0; i--) dp[i] = max(check(i, i + 1), check(i + 1, i));
cout << n + 1 - dp[0] << "\n";
```

This uses $O(n^2)$ time and $O(n)$ memory, which is good enough for $n \leq 2 \cdot 10^4$.

Note that sometimes not all dp states need to be calculated. In fact, if you only calculate the dp states we need (e. g. by recursion or by using a stack or in some different way), this will actually run in $O(n)$ for subtask 2. Can you figure out why?

Subtask 6, $n \leq 10^6$, Full solution

Let's fix one arm and look at it's trace while "pulling the other arm with it". Without loss of generality, $l \leq r$.

Consider the semi-greedy from earlier and consider a state $(s_l, s_r) = (i, j)$ in this semi-greedy. As we will always move the arm that is further behind, $t_j \leq t_i + r$ always holds. Now let's look at a new dp defined by dp[i][j] being the maximal number of monsters that can be slain if the left arm ends at position i and the right arm ends at position j. Note that this dp moves forwards, we are considering the monsters between 0 and i, and not as before the mosnters between i and the end. By the observation of the semi-greedy, it suffices to look at states where $t_j \leq t_i + r$ holds. Now suppose that we're considering a state in the dp where $j < i$. Then we can greedily move j forwards. Thus the only states we need to consider in the dp are states (i, j) such that $t_i \leq t_j \leq t_i + r$.

Lemma 2: Let x_i be the smallest index such that $t_i \leq t_{x_i} \leq t_i + r$ holds and such that $\text{dp}[i][x_i] = \max_{t_i \leq t_j \leq t_i + r} (\text{dp}[i][j])$. Then if there is an optimal solution passing through some state

(i, j), then there is also a solution passing through (i, x_i).

Proof: Consider an optimal solution going through (i, j). If now $j = x_i$, we're done. If $j > x_i$, then consider the following solution: Take the configuration for state (i, x_i) until the state and continue with the configuration for (i, j) afterwards. This solution can't be worse than the other optimal solution by definition of x_i, thus there is an optimal solution going through (i, x_i). In the remaining case $j < x_i$. Consider the following solution: Take the configuration for (i, x_i) until x_i. Then take the configuration for (i, j) except for the first monster we hit with the right arm. Then by definition of x_i, we now that the configuration of (i, x_i) will hit at least one monster more than the configuration of (i, j), but we lose one monster later on. Thus our solution hits the same number of monsters than the initial optimal solution. So we only need to prove that we can actually do this. Indeed, the next monster that we are going to hit appears at time $t_k \geq t_j + r \geq t_i + r \geq t_{x_i}$, which proves that this solution is valid. Thus there is indeed an optimal solution going through (i, x_i).

From Lemma 2, we learn that we only need to consider states of the form (i, x_i). Thus let $\text{opt}[i] = (\text{dp}[i][x_i], x_i)$. When we are processing $\text{opt}[i]$, either i is behind or i is at the same location than j. Thus we can always move with the left arm (i) and then "drag" the other arm along, which means we greedily move it along. Either we attack with the left arm, which means updating $\text{arr}_l[i]$, or we can not attack the monster (which means updating $i + 1$). Note that we only need to do the second step if $\text{opt}[i][0] = i$. And note again that we need to take care not to attack a monster twice by checking whether $\text{opt}[i][0] = i$ holds when we attack with the left arm. Whenever we update some future-state however, we need to greedily attack with the right arm. Since $l \leq r$ holds, we only need some amount in $O(1)$ steps until the right arm is back in the interval $[t_{i'}, t_{i'} + r]$, where i' is the state we moved to. Thus the running time of this algorithm is in $O(n)$. In code, it looks like this:

```cpp
if (l > r) swap(l, r); // Note: If you forget this, then it will be slow

vector<int> arr_l = getnextarr(n, pos, l);
vector<int> arr_r = getnextarr(n, pos, r);
vector<pair<int,int>> dp(n + 1, make_pair(-1e9, n + 1));
dp[0] = make_pair(0, 0);

auto update = [&](int ind, int v, int ot)
{
    if (v > dp[ind].first) dp[ind].first = v, dp[ind].second = ot;
    if (v == dp[ind].first) dp[ind].second = min(dp[ind].second, ot);
};

for (int curr = 0; curr < n; curr++)
{
    int nx = arr_l[curr];
    int s = dp[curr].second;
    // Don't forget this! Otherwise we will hit the monster with intex "curr" twice
    if (s == curr) s++;
    int res = dp[curr].first + 1;
    while (s < nx) { res++; s = arr_r[s]; }
    update(nx, res, s);

    // Note that we only need to do what follows if if dp[curr].second == curr.
    // However, the code is shorter without the if-statement, and it doesn't make
    // anything slow or incorrect
    nx = curr + 1;
    s = dp[curr].second;
    res = dp[curr].first;
    while (s < nx) { res++; s = arr_r[s]; }
    update(nx, res, s);
}

cout << n + 1 - dp[n].first << "\n";
```