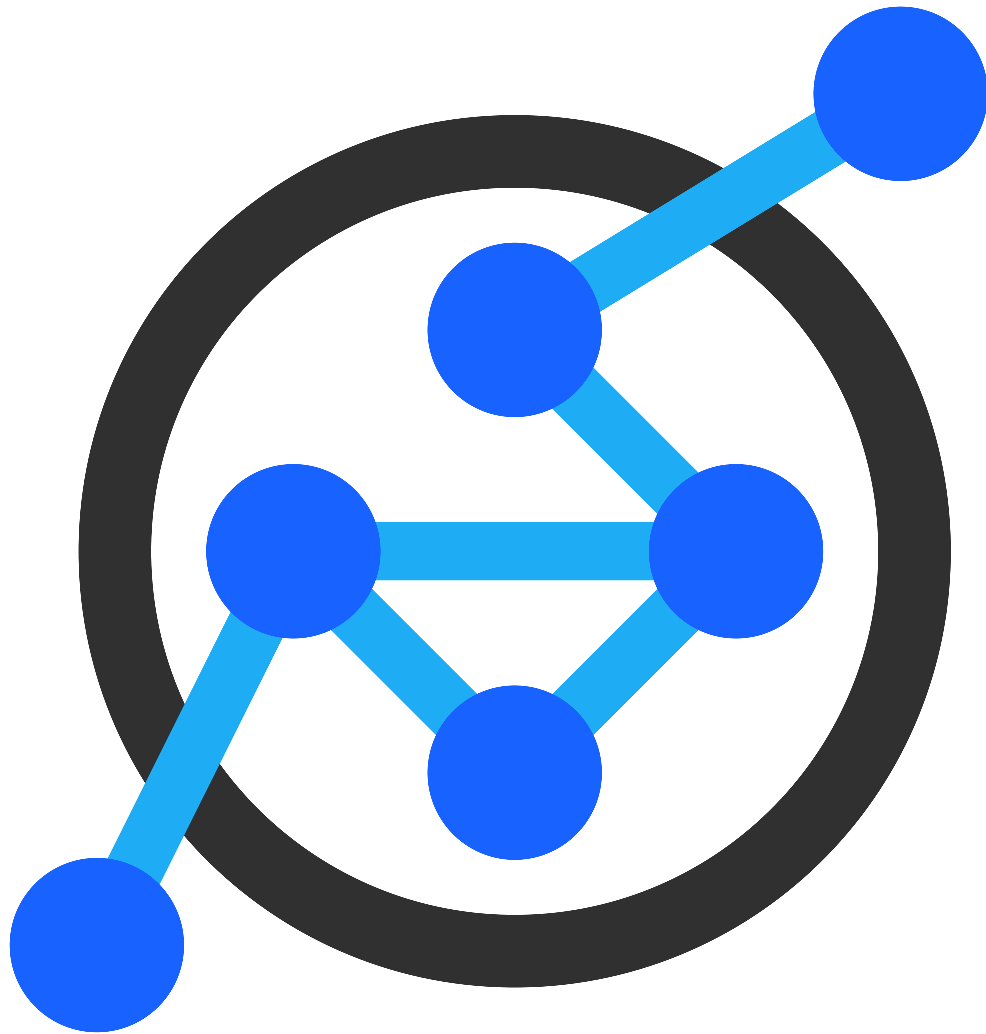
SOI 2P 2023

Solution Booklet



Swiss Olympiad in Informatics

10–14 March 2023

Waterflow

Task Idea	Charlotte Knierim
Task Preparation	Charlotte Knierim
Description English	Charlotte Knierim
Description German	Charlotte Knierim
Description French	Benjamin Faltin
Solution	Charlotte Knierim

We are given a sequence of tap and drain opening and closing times and want to calculate the maximum amount of water that is in Binnas pool at any point in time. Throughout this booklet we will refer to $t = max_i\{e_i\}$ as the time Binna is in the house.

Subtask 1: $c \geq 0$

If there are no drains, then we only have water flowing into the pool. Clearly, in this case the maximum is attained when the last tab is closed. We can view the tabs independently and just calculate for every tap. The water flowing in through tap i will be $(e_i - s_i) \cdot c_i$.

```
//score: 20

#include<bits/stdc++.h>

#define int long long

using namespace std;


signed main(){
	int m;
	cin >> m;

	int sum = 0;

	for(int i=0;i<m;++i) {
		int s,e,c;
		cin >> s >> e >> c;

		sum += (e-s)*c;
	}

	cout << sum << '\n';

}
```

Subtask 2: $m \leq 1000$ and $t \leq 10^4$

In this case we can find a solution that runs in $O(m \cdot t)$. For this, we can just have an array of all minutes and when we read in a tap or drain we go trough all minutes this tap/drain is active and add c_i to the change in this minute. Afterwards we do a summation of the changes while keeping track of the maximum.

```
//score: 40

#include<bits/stdc++.h>
#define int long long

using namespace std;

```

```
8 signed main(){
9         int m;
10        cin >> m;
11
12        vector<vector<int>> flows;
13        int max_t = 0;
14        for(int i = 0; i< m; ++ i) {
15                int s, t, f;
16                cin >> s >> t >> f;
17                flows.push_back(vector<int>({s,t,f}));
18                max_t = max(t,max_t);
19        }
20
21        vector<int> changes(max_t,0);
22
23        for(int i = 0; i<m;++i) {
24                int s = flows[i][0];
25                int t = flows[i][1];
26                int f = flows[i][2];
27
28                for(int j = s; j<t;++j) {
29                        changes[j] += f;
30                }
31        }
32
33        int max_v = 0;
34        int v = 0;
35
36        for(int i = 0 ; i<max_t; ++i) {
37                //cout << changes[i] <<'\n';
38                v += changes[i];
39                v = max(v,0LL);
40                max_v = max(max_v,v);
41        }
42
43        cout << max_v << '\n';
44 }
```

Subtask 3: $t \leq 10^6$

In this subtask we will see a solution that is linear in t. Similarly to the solution of subtasks 2 we keep track of the change in every minute, this time we are a bit more clever, noting the change of the change when reading the input.

```
1 //score: 60
2
3 #include<bits/stdc++.h>
4 #define int long long
5
6 using namespace std;
7
8 signed main(){
9         int m;
10        cin >> m;
11
12        vector<vector<int>> flows;
13        int max_t = 0;
14        for(int i = 0; i< m; ++ i) {
15                int s, t, f;
16                cin >> s >> t >> f;
17                flows.push_back(vector<int>({s,t,f}));
18                max_t = max(t,max_t);
19        }
20
21        vector<int> change(max_t+1,0);
22
```

```
23          for(int i = 0; i<m;++i) {
24                  int s = flows[i][0];
25                  int t = flows[i][1];
26                  int f = flows[i][2];
27
28                  change[s] += f;
29                  change[t] -= f;
30
31          }
32
33          vector<int> changes(max_t+1,0);
34          changes[0] = change[0];
35          for(int i = 1;i<max_t;++i){
36                  changes[i] = changes[i-1]+change[i];
37          }
38
39          int max_v = 0;
40          int v = 0;
41
42          for(int i = 0 ; i<max_t; ++i) {
43                  //cout << changes[i] <<'\n';
44                  v += changes[i];
45                  v = max(v,0LL);
46                  max_v = max(max_v,v);
47          }
48
49          cout << max_v << '\n';
50  }
```

Subtask 4: no further restrictions

We can solve this problem using a scanline. Creating events for the changes and sorting them results in a runningtime of $O(m \log m)$.

```
1  //score: 100
2  #include <bits/stdc++.h>
3  using namespace std;
4  typedef long long ll;
5
6  int main() {
7      cin.tie(0); ios::sync_with_stdio(NULL);
8      int m; cin >> m;
9      vector<pair<int,int>> a(2*m);
10     for (int i=0;i<m;i++) {
11         int s,e,c; cin >> s >> e >> c;
12         a[2*i] = {s,c}; a[2*i+1] = {e,-c};
13     }
14     sort(a.begin(),a.end());
15     ll l = 0, mx = 0, vol = 0, f = 0;
16     for (auto [t,s]: a) {
17         vol = max(vol+f*(t-l),0LL); mx = max(vol,mx);
18         f += s; l = t;
19     }
20     cout << mx;
21 }
```

Embroideries

Task Idea	Andrew Stankevich
Task Preparation	Daniel Rutschmann, Charlotte Knierim
Description English	Priska Steinebrunner, Charlotte Knierim
Description German	Charlotte Knierim
Description French	Benjamin Faltin
Solution	Timon Gehr

Subtask 1: $1 \leq n \leq 16$

There are at most $2^n \leq 2^{16} = 65'536$ subsequences. We can enumerate all of them by using the binary representation of the numbers 0 to $2^n - 1$ as a mask for indices that should occur in a subsequence. For each of the subsequences, we can check whether it is balanced and keep track of all balanced subsequences we can find in this way in a set. In the end, we simply print the size of the set.

```
#include <bits/stdc++.h> // score: 25
using namespace std;

bool is_balanced(const string &subsequence){
    int count=0;
    bool result=true;
    for(char c : subsequence){
        count += (c=='(')-(c==')');
        result &= 0<=count;
    }
    result &= count==0;
    return result;
}

int main(){
    int n;
    cin>>n;
    string pattern;
    cin>>pattern;
    if(n<=16){
        set<string> seen;
        // loop through all masks of subsequences
        for(int mask=0; mask<(1<<n); mask++){
            // extract subsequence
            string subsequence;
            for(int i=0; i<n; i++){
                if(mask & (1<<i)){
                    char c = pattern[i];
                    subsequence.push_back(c);
                }
            }
            // record all distinct balanced ones
            if(is_balanced(subsequence)){
                seen.insert(subsequence);
            }
        }
        cout<<seen.size()<<'\n';
    }
}
```

This solution runs in $O(2^n \cdot n \log n)$, which is enough to score all points for this subtask.

We can alternatively use a `vector<bool>` by encoding bracket sequences as integers and combine the production and consumption of subsequences in a way that avoids memory allocations as follows:

```cpp
#include <bits/stdc++.h> // score: 25
using namespace std;

int main(){
    int n;
    cin>>n;
    string pattern;
    cin>>pattern;
    if(n<=16){
        int num_subseq=1<<n;
        int num_nodes=(1<<(n+1))-1;
        vector<bool> seen(num_nodes);
        for(int mask=0;mask<num_subseq;mask++){
            bool is_balanced=true;
            int sequence=0, count=0;
            for(int i=0; i<n; i++){
                if(mask & (1<<i)){
                    char c=pattern[i];
                    count +=(c=='(')-(c==')');
                    is_balanced &= 0<=count;
                    if(c=='(') sequence = 2*sequence+1;
                    else sequence = 2*sequence+2;
                }
            }
            is_balanced &= count==0;
            seen[sequence] = is_balanced;
        }
        int result=0;
        for(bool s : seen) result+=s;
        cout<<result<<'\n';
    }
}
```

This also gets rid of the log factor, yielding a running time of $O(2^n \cdot n)$.

Subtask 2: $1 \le n \le 36$

In this subtask, there are up to $2^{36} = 68'719'476'736$ subsequences. We can no longer hope to enumerate all of them.

However, note that we can rephrase the task slightly using simple double counting: Instead of computing the number of subsequences that are balanced, we can compute the number of balanced strings of parentheses that occur as subsequences.

We can compute the number of balanced strings up to length 36 exactly, as $\sum_{k=0}^{18} C_k = 656'043'857$, where C_k is the k-th *Catalan number*, given by $C_k = \frac{1}{k+1}\binom{2k}{k}$. Unfortunately, this is still too large to explicitly enumerate.

However, we can do the following trick: We will count the number of balanced strings of parentheses of length $2 \cdot k$ that occur as subsequences of the pattern, for each k between 0 and 18. Then, the result will just be the sum of the results for each k.

How to compute the result for a specific k? Consider all 2^k bracket sequences of length k. For each sequence s, we compute the following information:

- Can it be the beginning of a balanced bracket sequence? If so, compute:
  - The length p_s of the shortest prefix of the pattern such that the sequence s appears as a subsequence of that prefix. (We ignore the sequence if there is no such prefix.)
  - The number of brackets b_s that still need to be closed in a string z that is appended to s to form a balanced sequence of brackets.
- Can it be the end of a balanced bracket sequence? If so, compute:

- The length q_z of the shortest suffix of the pattern such that the sequence z appears as a subsequence of that suffix. (We ignore the sequence if there is no such suffix.)
- The number of brackets d_s that have to be opened in a string s prepended to z to form a balanced sequence of brackets.

Then, the final result is given by $\sum_s \sum_z [p_s + q_z \leq n][b_s = d_z]$. (I.e., for each pair of sequences (s, z), each sequence of length k, their concatenation should be counted as a balanced sequence that occurs in the pattern iff s occurs as a subsequence in a prefix that does not overlap with a suffix within which z occurs as a subsequence, and the number of opened brackets in s matches the number of closed brackets in z.)

We can compute this sum quickly enough by summarizing sequences with the same values for (b_s, p_s) and (d_s, q_s) respectively. (The final sum could be further optimized using prefix sums, but this is not really necessary in this case as n is so small.)

The following program implements this solution. We precompute a reversed version (including reversed parentheses) of the pattern such that we can reuse most of the code related to balance and prefix length computations.

```
1  #include <bits/stdc++.h> // score: 50
2  using namespace std;
3
4  #define sz(v) (int)(v).size()
5
6  int get_balance(const string &subsequence){
7      int count=0;
8      for(char c : subsequence){
9          count += (c=='(')-(c==')');
10         if(count < 0){
11             return -1;
12         }
13     }
14     return count;
15 }
16
17 int extend_prefix(const string &pattern, int shortest_prefix, char next){
18     for(;;){
19         if(shortest_prefix>=sz(pattern))
20             return sz(pattern)+1;
21         if(pattern[shortest_prefix]==next)
22             return shortest_prefix+1;
23         ++shortest_prefix;
24     }
25 }
26
27 int shortest_prefix(string &pattern, string &subsequence){
28     int result=0;
29     for(char c : subsequence){
30         result = extend_prefix(pattern, result, c);
31     }
32     return result;
33 }
34
35 int main(){
36     int n;
37     cin>>n;
38     string pattern;
39     cin>>pattern;
40     string pattern_reverse=pattern;
41     reverse(pattern_reverse.begin(), pattern_reverse.end()); // reverse string
42     for(char &c:pattern_reverse) c ^= '('^')'; // reverse parentheses
43     if(n<=36){
44         int result=0;
45         for(int k=0;k<=n/2;k++){
46             auto prefixes = vector<vector<int>>(k+1, vector<int>(n+1, 0));
```

```
            auto suffixes = vector<vector<int>>(k+1, vector<int>(n+1, 0));
            string s(k, '\0');
            for(int sequence=0; sequence<(1<<k); sequence++){
                for(int i=0;i<k;i++){
                    s[i] = ((sequence & (1 << i)) ? ')' : '(');
                }
                int balance = get_balance(s);
                if(0<=balance && balance<=k){
                    int prefix = shortest_prefix(pattern, s);
                    if(prefix<=n){
                        ++prefixes[balance][prefix];
                    }
                    int suffix = shortest_prefix(pattern_reverse, s);
                    if(suffix<=n){
                        ++suffixes[balance][suffix];
                    }
                }
            }
            for(int balance=0; balance<=k; balance++){
                for(int p=0; p<=n; p++){
                    for(int pp=0; p+pp<=n; pp++){
                        result += prefixes[balance][p]*suffixes[balance][pp];
                    }
                }
            }
        }
        cout<<result<<'\n';
    }
}
```

This solution runs in time $O(2^{n/2} \cdot n^2)$. Note that above we have shown that the result is always smaller than $10^9 + 7$, so we did not have to bother with taking remainders.

Another way to write the `shortest_prefix` function that may appear a little more natural is to have an outer loop that goes over the pattern instead:

```
int shortest_prefix(string &pattern, string &subsequence){
        int n=sz(pattern), m=sz(subsequence);
        if(m==0) return 0;
        for(int i=0, j=0; i<n; i++){
                if(pattern[i]==subsequence[j]){
                        if(++j==m){
                                return i+1;
                        }
                }
        }
        return sz(pattern)+1;
}
```

However, it will be more convenient for further solutions to have the outer loop over the subsequence.

Subtask 3: $1 \leq n \leq 200$

To do even better, let's first consider again the solution approach where we enumerate all balanced sequences of brackets and check whether they occur in the pattern. The following code implements this solution:

```
#include <bits/stdc++.h> // score: 25
using namespace std;

#define sz(v) (int)(v).size()

int n;

vector<string> balanced;
```

```
9  void enumerate(string &cur,int balance){
10     if(balance<0 || balance>n-sz(cur)){
11         return;
12     }
13     if(balance==0){
14         balanced.push_back(cur);
15     }
16     if(sz(cur)<n){
17         cur.push_back('(');
18         enumerate(cur,balance+1);
19         cur.back()=')';
20         enumerate(cur,balance-1);
21         cur.pop_back();
22     }
23 }
24
25 string pattern;
26
27 int extend_prefix(int shortest_prefix, char next){
28     for(;;){
29         if(shortest_prefix>=n)
30             return n+1;
31         if(pattern[shortest_prefix]==next)
32             return shortest_prefix+1;
33         ++shortest_prefix;
34     }
35 }
36
37 int is_subsequence(string &candidate){
38     int shortest_prefix=0;
39     for(char c : candidate){
40         shortest_prefix = extend_prefix(shortest_prefix, c);
41     }
42     return shortest_prefix<=n;
43 }
44
45 int main(){
46     cin>>n>>pattern;
47
48     string cur="";
49     enumerate(cur,0);
50
51     int r=0;
52     for(string &candidate: balanced){
53         r += is_subsequence(candidate);
54     }
55     cout<<r<<'\n';
56 }
```

As we have noted before, the number of balanced bracket sequences grows exponentially fast, and in fact this solution only scores points for the first subtask.

However, it turns out that we can optimize this solution to get a vastly better asymptotic running time.

First, we will inline the subsequence check into the enumeration, such that we only enumerate sequences that are actually subsequences. This also allows an improvement to the pruning check that aborts if the balance gets too high.

```
1 #include <bits/stdc++.h> // score: 50
2 using namespace std;
3
4 #define sz(v) (int)(v).size()
5
6 int n;
7 string pattern;
8
9 int extend_prefix(int shortest_prefix,char next){
```

```
    for(;;){
        if(shortest_prefix>=n)
            return n+1;
        if(pattern[shortest_prefix]==next)
            return shortest_prefix+1;
        ++shortest_prefix;
    }
}

int result=0;
void enumerate(string &cur,int balance,int shortest_prefix){
    if(balance<0 || balance>n-shortest_prefix || shortest_prefix>n){
        return;
    }
    if(balance==0){
        result+=1;
    }
    if(sz(cur)<n){
        cur.push_back('(');
        enumerate(cur,balance+1,extend_prefix(shortest_prefix,'('));
        cur.back()=')';
        enumerate(cur,balance-1,extend_prefix(shortest_prefix,')'));
        cur.pop_back();
    }
}

int main(){
    cin>>n>>pattern;

    string cur="";
    enumerate(cur,0,0);

    cout<<result<<'\n';
}
```

This is actually enough of an optimization to solve the second subtask again: even though it is not feasible to enumerate all subsequences of the pattern of length $n \leq 36$ nor all balanced sequences up to length $n \leq 36$ individually, it is actually feasible to enumerate all subsequences that are balanced.

Now we can make the following observations:

- The contents of the string `cur` are never actually used, therefore we don't need to compute it at all. It suffices to keep track of its length.
- Instead of updating the result as a side effect of the `enumerate` procedure, we can compute the total change to the result internally to the function and return it.
- The function `extend_prefix` can be written recursively instead of with a for loop.

If we implement those changes, we are left with the following solution:

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

#define sz(v) (int)(v).size()

int n;
string pattern;

int extend_prefix(int shortest_prefix,char next){
    if(shortest_prefix>=n)
        return n+1;
    if(pattern[shortest_prefix]==next)
        return shortest_prefix+1;
    return extend_prefix(shortest_prefix+1,next);
}

```

```
17 int solve(int length,int balance,int shortest_prefix){
18     if(balance<0 || balance>n-shortest_prefix || shortest_prefix>n){
19         return 0;
20     }
21     int result=0;
22     if(balance==0){
23         result+=1;
24     }
25     if(length<n){
26         result += solve(length+1,balance+1,extend_prefix(shortest_prefix,'('));
27         result += solve(length+1,balance-1,extend_prefix(shortest_prefix,')'));
28     }
29     return result;
30 }
31
32 int main(){
33     cin>>n>>pattern;
34     cout<<solve(0,0,0)<<'\n';
35 }
```

This solution still runs in exponential time, but we can now actually easily optimize it, because both functions can only be called with a very limited number of different arguments. Therefore, many of the function calls will actually yield the same result. We can simply memoize the results to get a dynamic programming solution that runs in polynomial time. Note that now we also need to take into account results that are larger than $10^9 + 7$.

```
1 #include <bits/stdc++.h> // score: 75
2 using namespace std;
3
4 #define sz(v) (int)(v).size()
5 const int MOD=(int)1e9+7;
6
7 void add(int &a, int b){
8     a += b;
9     if(a>=MOD) a-=MOD;
10 }
11
12 int n;
13 string pattern;
14
15 vector<array<int,2>> memo_ep;
16 int extend_prefix(int shortest_prefix,char next){
17     if(shortest_prefix>=n)
18         return n+1;
19     int &mem=memo_ep[shortest_prefix][next==')'];
20     if(mem!=-1) return mem;
21     if(pattern[shortest_prefix]==next)
22         return shortest_prefix+1;
23     return mem=extend_prefix(shortest_prefix+1,next);
24 }
25
26 vector<vector<vector<int>>> memo;
27 int solve(int length,int balance,int shortest_prefix){
28     if(balance<0 || balance>n-shortest_prefix || shortest_prefix>n){
29         return 0;
30     }
31     int &mem=memo[length][balance][shortest_prefix];
32     if(mem!=-1) return mem;
33     int result=0;
34     if(balance==0){
35         add(result, 1);
36     }
37     if(length<n){
38         add(result, solve(length+1,balance+1,extend_prefix(shortest_prefix,'(')));
39         add(result, solve(length+1,balance-1,extend_prefix(shortest_prefix,')')));
40     }
41     return mem=result;
```

```
42 }
43
44 int main(){
45     cin>>n>>pattern;
46     memo_ep.assign(n, {-1, -1});
47     memo.assign(n+1, vector<vector<int>>(n+1, vector<int>(n+1,-1)));
48     cout<<solve(0,0,0)<<'\n';
49 }
```

This solution runs in time and space $O(n^3)$.

Subtask 4: $1 \leq n \leq 3'000$

Note that the parameter `length` is only used to check whether the length of the current subsequence is still below n. This check is not really necessary anymore because the enumeration terminates once the current bracket sequence no longer appears as a subsequence of the pattern. It is therefore easy to eliminate the `length` parameter. There are various other ways to write a solution that will pass subtask 3 but not subtask 4 as a result of too many factors of n in the running time .

If we get rid of `length`, the resulting solution scores 100 points:

```
1  #include <bits/stdc++.h> // score: 100
2  using namespace std;
3
4  #define sz(v) (int)(v).size()
5  const int MOD=(int)1e9+7;
6
7  void add(int &a, int b){
8      a += b;
9      if(a>=MOD) a-=MOD;
10 }
11
12 int n;
13 string pattern;
14
15 vector<array<int,2>> memo_ep;
16 int extend_prefix(int shortest_prefix,char next){
17     if(shortest_prefix>=n)
18         return n+1;
19     int &mem=memo_ep[shortest_prefix][next==')'];
20     if(mem!=-1) return mem;
21     if(pattern[shortest_prefix]==next)
22         return shortest_prefix+1;
23     return mem=extend_prefix(shortest_prefix+1,next);
24 }
25
26 vector<vector<int>> memo;
27 int solve(int balance,int shortest_prefix){
28     if(balance<0 || balance>n-shortest_prefix || shortest_prefix>n){
29         return 0;
30     }
31     int &mem=memo[balance][shortest_prefix];
32     if(mem!=-1) return mem;
33     int result=0;
34     if(balance==0){
35         add(result, 1);
36     }
37     add(result, solve(balance+1,extend_prefix(shortest_prefix,'(')));
38     add(result, solve(balance-1,extend_prefix(shortest_prefix,')')));
39     return mem=result;
40 }
41
42 int main(){
43     cin>>n>>pattern;
```

```
44      memo_ep.assign(n, {-1, -1});
45      memo.assign(n+1, vector<int>(n+1,-1));
46      cout<<solve(0,0)<<'\n';
47  }
```

Mega jump

Task Idea	Charlotte Knierim, Michał Stawarz, Johannes Kapfhammer
Task Preparation	Jan Schär
Description English	Jan Schär
Description German	Jan Schär
Description French	Benjamin Faltin
Solution	Jan Schär

Subtask 1: Only walking

In this subtask, a mega jump is the same as walking diagonally, which means we can ignore mega jumps. In each step, we can move by at most one in either direction, so we need at least $\max(x, y)$ steps. And we *can* reach the goal in $\max(x, y)$ steps, by first walking diagonally for $\min(x, y)$ steps, and then walking the remaining steps either horizontally or vertically, depending on which coordinate is larger.

```
// score: 10
#include <bits/stdc++.h>
using namespace std;

int main () {
  int k, l, q;
  cin >> k >> l >> q;

  for (int qi = 0; qi < q; qi++) {
    int x, y;
    cin >> x >> y;
    cout << max(x, y) << "\n";
  }
}
```

This solution runs in $O(1)$ time.

Subtask 2: Jumping horizontally

Here, mega jumps only change the horizontal position. That means we need to walk at least y steps to reach the correct vertical position. To make the most out of these unavoidable steps, we can walk them diagonally. If $y > x$, then we can already reach the target with this, so the answer is y. Otherwise, we will have a remaining horizontal distance of $x - y$ which we can cover with a combination of steps and mega jumps. A single mega jump replaces k steps, so we should make $\lfloor \frac{x-y}{k} \rfloor$ mega jumps at least. In this case, the total number of steps we make (horizontal and diagonal) is $x - \lfloor \frac{x-y}{k} \rfloor \cdot k$ (this is the remaining horizontal distance after making $\lfloor \frac{x-y}{k} \rfloor$ mega jumps).

It might be better to make one additional mega jump, because then we might reduce the number horizontal steps. But making more than one additional mega jump cannot help, because even with one we are already past the point which can be reached by walking only diagonally. We can then use the solution from subtask 1 for calculating the number of steps. We need at least y steps, but if we jump far enough that the horizontal distance $x - (\lfloor \frac{x-y}{k} \rfloor + 1) \cdot k$ is bigger than y, then that is the number of steps. Rather than trying to find out whether we should make an additional mega jump, we can just calculate both and take the minimum.

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

```
4
5  int solve (int k, int l, int x, int y) {
6    if (y > x) return y;
7    int min_jumps = (x - y) / k;
8    return min(
9      min_jumps + x - min_jumps * k,
10     (min_jumps + 1) + max(y, (min_jumps + 1) * k - x)
11   );
12 }
13
14 int main () {
15   int k, l, q;
16   cin >> k >> l >> q;
17
18   for (int qi = 0; qi < q; qi++) {
19     int x, y;
20     cin >> x >> y;
21     cout << solve(k, l, x, y) << "\n";
22   }
23 }
```

This solution runs in $O(1)$ time.

Subtask 3: Small distance

In this subtask, the distance is small, so we can use an inefficient solution which just tries out all possibilities. We can make several simple observations: The order of moves does not matter, so we only need to know how many we make of each possible move. There are four different mega jumps, but a (k, l) and a $(-k, -l)$ jump cancel each other out. So rather than saying we have a times (k, l) and b times $(-k, -l)$, we could just say we have $a - b$ times (k, l). The same argument holds for $(k, -l)$ and $(-k, l)$. That means we only need to consider two types of mega jumps. Once the mega jumps are fixed, we can then use the solution from subtask 1 to calculate the number of steps.

To summarize, our strategy is to try out all combinations of numbers of (k, l) and $(k, -l)$ jumps in some range, and for each compute the number of steps. Then we take the minimum of all these possibilities. The question that remains is how big the range should be that we try out. A simple upper bound is $\max(x, y)$ – we know that we need this many steps when not using mega jumps, so we certainly don't want to make more mega jumps than this.

```
1  // score: 20
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  int solve (int k, int l, int x, int y) {
6    int d = numeric_limits<int>::max();
7    int limit = max(x, y);
8    for (int s1 = -limit; s1 <= limit; s1++) {
9      for (int s2 = -limit; s2 <= limit; s2++) {
10       d = min(
11         d,
12         abs(s1) + abs(s2) +
13         max(abs(x - k * (s1 + s2)), abs(y - l * (s1 - s2)))
14       );
15     }
16   }
17   return d;
18 }
19
20 int main () {
21   int k, l, q;
22   cin >> k >> l >> q;
23
24   for (int qi = 0; qi < q; qi++) {
```

```
25    int x, y;
26    cin >> x >> y;
27    cout << solve(k, l, x, y) << "\n";
28  }
29 }
```

This solution runs in $O(\max(x, y)^2)$ time.

Subtask 4: General solution

Now, we need an efficient solution which works in all cases. We can approach this by repeatedly converting the problem into a simpler one, until we can solve it directly.

The points that are reachable with only mega jumps form a kind of grid where half the points are missing, and this is inconvenient to work with. But consider what happens when we make exactly two mega jumps: we can reach the points $(2k, 0)$, $(0, 2l)$, and $(2k, 2l)$. (Here we ignore jumps with negative coordinates, because they don't help when we want to reach a point with non-negative coordinates.) This is very similar to walking. We can see that with an even number of mega jumps, the reachable points form a nice grid with $2k$ horizontal and $2l$ vertical spacing. And to reach a point $(a \cdot 2k, b \cdot 2l)$, we need $2\max(a, b)$ moves. When we have an odd number of mega jumps instead, then the grid is simply shifted by (k, l). If we already have a solution for the even case, we can call it with the target shifted by $(-k, -l)$ and then add one extra move. We can just try both even and odd, and take the minimum.

Now, we have a simpler problem: We can walk steps as before, which take 1 move, and we can make jumps of $(k, 0)$, $(0, l)$, and (k, l), which take 2 moves. (Here, k and l are doubled compared to before, because otherwise we would have a factor 2 everywhere we use them.) We want to minimize walking, because jumping uses fewer moves. To do that, we first compute the minimum number of steps which are required to reach the target no matter what. Then, we calculate how many jumps are required, given this number of steps. However, it is possible that we need one jump less if we make one step more, and since jumps take 2 moves, we save a move in total. Again, we can just try with or without an extra step and take the minimum. Walking more than one additional step won't help, because we would need to walk at least $\min(k, l)$ more steps to further decrease the number of jumps.

To compute the minimum number of steps: In the x-direction, if $k = 0$, then we can't jump in that direction and need to walk x steps. Otherwise, we jump to the closest multiple of k. $x \bmod k$ is the number of remaining steps after $\lfloor \frac{x}{k} \rfloor$ jumps, and if we make one more jump, we walk $k - (x \bmod k)$ steps in the other direction. Because we can walk diagonally, the overall minimum number of steps is the maximum of the two dimensions.

To compute the number of jumps, given the number of steps: We can subtract the number of steps from x and y (clamping to zero if negative), this is the closest we can get by walking the given number of steps diagonally. Then, we calculate how many jumps we need in each direction to reach at least that far.

```
1  // score: 100
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  int solve_jumps (int k, int l, int x, int y, int steps) {
6    int jumps = max(
7      k == 0 ? 0 : (max(x - steps, 0L) + k - 1) / k,
8      l == 0 ? 0 : (max(y - steps, 0L) + l - 1) / l
9    );
10   return 2 * jumps + steps;
11 }
12
13 int solve_rect (int k, int l, int x, int y) {
14   x = abs(x);
```

```
15   y = abs(y);
16   int min_steps = max(
17     k == 0 ? x : min(x % k, k - x % k),
18     l == 0 ? y : min(y % l, l - y % l)
19   );
20   return min(
21     solve_jumps(k, l, x, y, min_steps),
22     solve_jumps(k, l, x, y, min_steps + 1)
23   );
24 }
25
26 int solve (int k, int l, int x, int y) {
27   return min(
28     solve_rect(k * 2, l * 2, x, y),
29     1 + solve_rect(k * 2, l * 2, x - k, y - l)
30   );
31 }
32
33 int main () {
34   int k, l, q;
35   cin >> k >> l >> q;
36
37   for (int qi = 0; qi < q; qi++) {
38     int x, y;
39     cin >> x >> y;
40     int d = solve(k, l, x, y);
41     cout << d << "\n";
42   }
43 }
```

This solution runs in $O(1)$ time.

A note on contest strategy: Many participants made lots of attempts for the full solution, but did not succeed in the end. In this situation, rather than blindly trying various changes, you should stop and try to solve the easier subtask 3 first, so that you have at least partial points. Another advantage is that you can then compare the output between your working sub3 solution and your broken full solution on your own samples, which can help in understanding why it is broken.

Frenemies

Task Idea	Benjamin Faltin
Task Preparation	Benjamin Faltin
Description English	Benjamin Faltin
Description German	Charlotte Knierim
Description French	Benjamin Faltin
Solution	Benjamin Faltin

Subtask 1:

A key observation in this subtask is that if two mice dislike each other they will not have any other enemies or friends. If it was the case one mouse would have multiple enemies contradicting the constraints. Thus, we need to keep track of who has friends, who has enemies and with whom. To calculate the size of the biggest friend group possible we just need to know the number X of pairs of enemies. The answer will then be $N - X$ as we can have a group with everyone but one from each of those pairs. This can be implemented as shown below.

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

int main() {
    cin.tie(0); ios_base::sync_with_stdio(NULL);
    int n,q; cin >> n >> q;
    set<int> has_friends,has_enemies;
    set<pair<int,int>> enemies;
    while (q--) {
        int s,a,b; cin >> s >> a >> b;
        if (a<b) swap(a,b);
        if (s) {
            if (enemies.count({a,b})) cout << "YES\n";
            else if (has_enemies.count(a) || has_enemies.count(b) ||
            has_friends.count(a) || has_friends.count(b)) cout << "NO\n";
            else {
                cout << "YES\n";
                enemies.insert({a,b});
                has_enemies.insert(a);
                has_enemies.insert(b);
            }
        } else {
            if (has_enemies.count(a) || has_enemies.count(b)) cout << "NO\n";
            else {
                cout << "YES\n";
                has_friends.insert(a);
                has_friends.insert(b);
            }
        }
    }
    cout << n-enemies.size();
}
```

Subtask 2: $N \leq 1000$ and $Q \leq 2000$

To process the Q queries we can build a graph the following way and check for bipartiteness. If two mice dislike each other, add an edge between them. If they do like each other, we add an edge between the first mouse and a virtual node as well as another edge between this virtual node and the second mouse.
The size of the biggest friend group is equal to the sum of every component's maximum colour

(not counting virtual nodes).

```
#include <bits/stdc++.h> //score: 60
using namespace std;
vector<vector<int>> adj;
int n, on;

bool bipartite(int& tot) {
    vector<int> side(n,-1);
    bool is_bipartite = true;
    queue<int> q;
    for (int i=0;i<n;i++) {
        if (side[i] == -1) {
            vector<int> cur = {i};
            q.push(i);
            side[i] = 0;
            while (!q.empty()) {
                int v = q.front(); q.pop();
                for (int u : adj[v]) {
                    if (side[u] == -1) {
                        side[u] = side[v] ^ 1;
                        q.push(u);
                        cur.push_back(u);
                    } else is_bipartite &= side[u] != side[v];
                }
            }
            int c1=0,c2=0;
            for (auto u: cur) {
                if (u >= on) continue;
                if (side[u]) c1++;
                else c2++;
            }
            tot += max(c1,c2);
        }
    }
    return is_bipartite;
}

int main() {
    cin.tie(0); ios_base::sync_with_stdio(NULL);
    int q,tmp=0; cin >> n >> q;
    on = n; adj.resize(n);
    while (q--) {
        int e,i,j; cin >> e >> i >> j;
        if (e) {
            adj[i].push_back(j);
            adj[j].push_back(i);
        } else {
            adj.push_back({i,j});
            adj[i].push_back(n);
            adj[j].push_back(n);
            n++;
        }
        if (bipartite(tmp)) {
            cout << "YES\n";
        } else {
            cout << "NO\n";
            adj[i].pop_back();
            adj[j].pop_back();
            if (!e) {
                adj.pop_back();
                n--;
            }
        }
    }
    int cnt=0; bipartite(cnt);
    cout << cnt << "\n";
}
```

Subtask 3: $N \leq 10^5$ and $Q \leq 3.5 \cdot 10^5$

The question now becomes how to efficiently check for online bipartiteness. To do this, we will use a union-find data structure with $2N$ nodes. For every mouse i ($0 \leq i \leq N-1$), node i represents the colour of mouse i and node $i+N$ its opposite colour. Now if mouse i and mouse j dislike each other, we merge the nodes i with $j+N$ and $i+N$ with j. If they like each other, we merge i with j and $j+N$ with $i+N$. If the merging connects i and $i+N$ we know it will make the graph non-bipartite. Finally, we can find the size of the biggest friend group in the same way as subtask 2.

```
#include <bits/stdc++.h> //score: 100
using namespace std;
vector<int> p,cnt;

int find_set(int v) {
    if (v == p[v]) return v;
    return p[v] = find_set(p[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    p[b] = a;
    if (a != b) cnt[a] += cnt[b];
}

int main() {
    cin.tie(0); ios_base::sync_with_stdio(NULL);
    int n,q; cin >> n >> q;
    p.resize(2*n); cnt.resize(2*n);
    for (int i=0;i<n;i++) {
        p[i] = i; p[i+n] = i+n;
        cnt[i] = 1;
    }

    while (q--) {
        int e,i,j; cin >> e >> i >> j;
        if (e) {
            if (find_set(i) == find_set(j)) cout << "NO\n";
            else {
                cout << "YES\n";
                union_sets(i,j+n);
                union_sets(i+n,j);
            }
        } else {
            if (find_set(i) == find_set(j+n)) cout << "NO\n";
            else {
                cout << "YES\n";
                union_sets(i,j);
                union_sets(i+n,j+n);
            }
        }
    }
    int mx = 0;
    for (int i=0;i<n;i++) {
        if (find_set(i) == i) mx += max(cnt[i],cnt[i+n]);
    }
    cout << mx << "\n";
}
```