April 29, 2023

New Zealand Informatics Competition 2023

Round One

Editorial by Bruce C, Jonathon S, Anatol C, Phoebe Z, Zalan V

Contents

Introduction

The solutions to this round of NZIC problems are discussed in detail below. In a few questions we may refer to the Big O complexity of a solution, e.g. $O(N)$. There is an explanation of Big O complexity at the end of this document.

Resources

Ever wondered what the error messages mean?

https://www.nzoi.org.nz/nzic/resources/understanding-judge-feedback.pdf

Read about how the server marking works:

https://www.nzoi.org.nz/nzic/resources/how-judging-works-python3.pdf

Ever wondered why your submission scored zero?

https://www.nzoi.org.nz/nzic/resources/why-did-i-score-zero.pdf

See our list of other useful resources here:

https://www.nzoi.org.nz/nzic/resources

Inventory Management

Problem authored by Jonathan Khoo
https://train.nzoi.org.nz/problems/1314

Subtask 1

The problem asks us to find an optimal subset of knives such the sum of the d values of those knives is maximised. Since $K = 1$, we can only choose 1 knife, so it follows that we should choose the knife with the maximum d value. This can be done by iterating through the list of d values and finding the maximum one, or simply using the `max()` function in Python.

Python Solution

```
N, K = list(map(int, input().split()))
d = list(map(int, input().split()))
print(max(d))
```

Subtask 2

$N, K \leq 1000$, so a brute force solution with time complexity $O(NK)$ will pass. Firstly, we can make the observation that the optimal subset of knives will always contain the K knives with the largest d values, because if we ever have a d value that is not one of the K largest d values, we can swap it for one of the K largest d values. Therefore, it suffices to maintain a list of currently available d values and perform K iterations, where in each iteration we find maximum d value in the current list, add it to the total sum and delete it from the list. Our answer is the total sum after K iterations. Since there are K iterations and each iteration runs in linear time on the list of d values, the total time complexity is $O(NK)$.

Python Solution

```
N, K = list(map(int, input().split()))
d = list(map(int, input().split()))

res = 0
for i in range(K):
    cur = max(d)
    res += cur
    d.remove(cur)

print(res)
```

Subtask 3

$N, K \leq 100000$ here, so the solution to subtask 2 will not pass within the time limit. However, we can use the same observation from subtask 2 to reach the full solution. Instead of finding the maximum d for K iterations, we can sort the list in non-increasing order and calculate the sum of the first K elements of the sorted list to achieve the same answer. This can be accomplished with Python's built-in `sort()` function which runs in $O(N \log N)$. Since we only need to go through the first K elements after the list is sorted, the total time complexity is $O(N \log N)$ which will obtain the full score.

Python Solution

```
N, K = list(map(int, input().split()))
d = list(map(int, input().split()))

res = 0
d.sort(reverse=True)
```

```
for i in range(K):
    res += d[i]

print(res)
```

Repdigits

Problem authored by Jonathan Khoo
https://train.nzoi.org.nz/problems/1308

Note that the model subtask solutions shown here may only solve the one specific subtask. In order to obtain points for multiple subtasks you may need to integrate solutions for several subtasks into one submission on the NZIC contest website.

Subtask 1

For this subtask we have $N \leq 45$. This bound may seem strange at first but 45 is actually equal to the sum of all the integers from one through to nine (which are all repdigits) and in fact, all the integers from 1 to 45 can be represented as a sum of distinct numbers between one and nine. From this, there are a few ways to go about programming a solution.

One way is to simply always take the largest number between one and nine that is still valid until everything you have selected adds up to the target number. This will always work and you can easily verify this for all values up to 45 by simply running your program on all those numbers.

Python Solution

```
N = int(input())

current_total = 0
values = []

while current_total != N:
  for i in range(9, 0, -1):
    if i not in values and current_total + i <= N:
      values.append(i)
      current_total += i

print(len(values))

for val in values:
  print(val)
```

Extra Challenge

As an extra challenge, you may want to try find why this shorter and faster python code does effectively the same thing as the one above.

```
N = int(input())

values = []

for i in range(9, 0, -1):
  if i <= N:
    values.append(i)
    N -= i

print(len(values))

for val in values:
  print(val)
```

Subtask 2

We can generalise the solution in the previous subtask to work for numbers which need larger repdigits in their decomposition. If instead of just taking the largest number between 1 and 9 at each step we take the largest repdigit, then we always arrive at a valid solution. A proof is provided at the end of this section.

Since in this subtask $N \leq 10^5$, to find the largest repdigit we can use, we can do this naively in $O(N)$ time.

The overall complexity of the solution below is $O(N)$.

Python Solution

The code for this solution can be found at the end of this document.

Extra Challenge (Advanced)

If you know Dynamic Programming, try to solve this subtask while minimizing the number of repdigits used in your solution (you may find the solutions for the next subtasks helpful).

Subtask 3

In this subtask, N can be represented as a sum of exactly 2 distinct repdigits. Therefore, if we had a list of all the repdigits $\leq N$, we can just check every repdigit and see if subtracting it from N yields another (different) repdigit in which case we have found a solution. This works because there are actually very few repdigits we would ever need to check (in fact there are exactly 81).

Since $N \leq 10^9$, to generate this list, we cannot just go through every number up to N and check if it is a repdigit as this would be $O(N)$. Instead we need a smarter way to generate repdigits. One way (of many) is presented below.

Generating Repdigits

Note that all the repdigits with x digits (where x can be any natural number) are a multiple of the number made of x ones. For example, 4444 is a repdigit and is a multiple of 1111.

So, to generate all repdigits, we can generate all the numbers made of ones ($1, 11, 111, \ldots$) and multiply them by all numbers between one and nine to get all repdigits with the same number of digits as them.

Conveniently, we can get from one of these "one numbers" to the next by the formula $10 \times x + 1$. For example, to get the one after 1111, we do $10 \times 1111 + 1 = 11111$. Generating all repdigits up to 10^9 can be done with the following code (be careful that naively implementing this in C++ may result in integer overflow):

```
repdigits = []
x = 1

while x <= 1000000000:
  for i in range(1, 10):
    repdigits.append(x * i)
  x = 10 * x + 1
```

Python Solution

The code for this solution can be found at the end of this document.

Subtask 4

To solve the full problem, we can combine the ideas in the solutions for Subtask 2 and 3. We will use the same sort of greedy algorithm as in Subtask 2 except that instead of finding each next repdigit in

O(N), we can simply pregenerate a list of all the repdigits similar to Subtask 3. This runs in $O(\log N)$ overall (given in proof of correctness below).

Python Solution
The code for this solution can be found at the end of this document.

C++ Solution
The code for this solution can be found at the end of this document.

Proof of Correctness & Time Complexity

In informatics, it is often not necessary to mathematically prove correctness as long as you can convince yourself (and the testcases) that your solution works. However, we will do so here for your understanding.

To prove the correctness of the greedy algorithm in Subtasks 2 & 4, it suffices to show that for any repdigit A, there is another repdigit B such that $A < B \leq 2A$.

This is quite easy to demonstrate. If we have a repdigit A made of any digit D that isn't a nine, the next repdigit B is the one with digits $D + 1$. The quotient of these two repdigits is simply $\frac{D+1}{D}$ which for any natural number D is at most 2 and so $A < B \leq 2A$.

If A consists of k nines, the next repdigit B is the one made of $k + 1$ ones. The double of A will at least start with the numbers 18 and will have the same number of digits as B. Thus it is clear that $A < B \leq 2A$.

Since this is true, then if for a certain N we have found the largest repdigit C, we are guaranteed that $2C > N$ which means that the number we get from subtracting C, is strictly less than C.

$$\begin{aligned} 2C &> N \\ 2C - C &> N - C \\ C &> N - C \end{aligned}$$

This means that our algorithm will never need to subtract C a second time (since it can't) and so the solution will never contain the same repdigit twice. Therefore, any solution it generates will be valid.

On top of that, for any natural number N, there will always be a repdigit $\leq N$ because the number one is a repdigit. As such, our algorithm can find a solution for any number N.

For the time complexity, the first step in our algorithm is finding all repdigits $\leq N$, for any number of digits there are exactly nine repdigits so the total number of repdigits is proportional to the number of digits in N. Therefore, it takes $O(\log N)$ time to find the repdigits.

Next, at every step of our greedy algorithm we at least half our value of N and so we will do $O(\log N)$ steps each of which take $O(1)$ resulting in overall $O(\log N)$ time complexity for this step.

Both steps take $O(\log N)$ so overall complexity is $O(\log N)$.

Emma's Switches

Problem authored by Bruce Chen
https://train.nzoi.org.nz/problems/1311

Note that the model subtask solutions shown here only solve the one specific subtask. In order to obtain points for multiple subtasks you may need to integrate solutions for several subtasks into one submission on the NZIC contest website.

Subtask 1

N and D are small, so a brute-force approach which simply simulates the actions of Emma should pass within the time and memory constraints. This approach has a time complexity of $O(N \times D)$.

Python Solution

```
N, D = map(int, input().split())

switches = [False] * N

for j in range(D):
  k, p = map(int, input().split())
  for i in range(k-1, N, p):
    switches[i] = not switches[i]

for s in switches:
  print("ON" if s else "OFF")
```

Subtask 2

N and D can now be large (up to 50,000), so the brute-force approach no longer works. In the worst case, $N \times D = 2.5$ billion operations which is far too many for the 1 second time limit.

An important observation to make is that as long as each switch is toggled the same number of times in total, their final states will be the same. This means that the ordering of the days does not matter. For example, the sample input of:

```
5 2
1 2
2 3
```

gives the exact same output as

```
5 2
2 3
1 2
```

The second thing to notice is that given $k_i = 1$ and $1 \leq p_i \leq 50$ in this subtask, there are only 50 distinct ways in which Emma can toggle the switches each day. Therefore, we can rearrange and group the days by their p_i values to reduce the number of operations.

To do so, we keep a tally for how many times each of the 50 distinct p_i values appear across all days. Then, for each distinct p_i value, we will sweep across all light switches and add the tally of p_i to every p_i-th switch. We complete one last iteration through all the light switches; if the light switch has been toggled an odd number of times, output `ON`. Otherwise, output `OFF`. This has a time complexity of $O\left(p_{\max} \times \frac{N}{p_{\text{avg}}}\right)$. If we assume that in the worst case $p_{\text{avg}} \approx 1$ then the complexity becomes $O(p_{\max} N)$, which is still sufficient.

Python Solution

```
N, D = map(int, input().split())
s = [0] * N
ps = [0] * 51
for x in range(D):
    k, p = map(int, input().split())
    ps[p] += 1

for p in range(1, 51):
    for x in range(0, N, p):
        s[x] += ps[p]

for x in s:
    print("ON" if x % 2 == 1 else "OFF")
```

Subtask 3

The same observations from subtask 2 will help us here. Given $1 \leq k_i \leq N$, there are only N distinct ways for Emma to toggle the light switches each day, so we can instead group the days by their distinct k_i values. For each distinct k_i value, we tally up the number of days with that k_i value.

We then iterate across all light switches, incrementing the number of toggles by the number of days with that k_i value as we go across. Same as before, output `ON` if the total number of toggles for a switch is odd.

This has a time complexity of $O(N)$.

Python Solution

```
N, D = map(int, input().split())
s = [0] * N
for x in range(D):
    k, p = map(int, input().split())
    s[k-1] += 1

toggles = 0
for x in range(0, N):
    toggles += s[x]
    print("ON" if toggles % 2 == 1 else "OFF")
```

Subtask 4

We can combine the approaches of subtask 2 and subtask 3 together to obtain the full solution. We can try to group the days by their p_i value as we did in subtask 2, but this doesn't immediately work. As k_i is no longer constrained to 1, days with the same separator (p_i) value but different k_i value can be out of "phase" with one another. Instead, we can group the days by both its p_i value and also its "phase" value (defined as $k_i \bmod p_i$).

Again, there are only 50 distinct p values, and for each there are p possible "phases", so the total number of groups would be:

$$\sum p = 1 + 2 + \ldots + 50 = 50\left(\frac{1+50}{2}\right) = 1275$$

For each group, we need to sweep across the light switches in a similar fashion to the solution of subtask 3, incrementing a counter with any new days in that group along the way. At first this might seem like $O(\text{number of groups} \times n) = O(1275n)$, but it is important to remember that all groups with a given p separator value only needs to iterate through $\frac{N}{p}$ switches. Therefore, across the p groups which

have the same separator value but different phase values the number of elements accessed totals to $p \times \frac{N}{p} = N$. As there are 50 distinct separator values, the complexity of this solution is overall $O(50n)$.

Python Solution

The code for this solution can be found at the end of this document.

C++ Solution

The code for this solution can be found at the end of this document.

Checking your understanding

How would you modify the solution if the problem was extended so that for each day you're given an additional integer m_i, the maximum number of light switches that Emma toggles in a day before stopping?

Up for more of a challenge?

Try out the problem Emma's Switches 2. Make sure you fully understand the complexities of all subtask solutions of the original problem first.

Sunsprint

Problem authored by Joseph Grace
https://train.nzoi.org.nz/problems/1307

Subtasks 1 & 2

Since $T = 0$, you will accumulate the same amount of exposure for every second spent in the sunlight. Thus, the problem becomes finding the path from rest stop 0 to rest stop $N - 1$ which minimises the amount of time spent in unshaded paths. Some readers will note that we can use Dijkstra's algorithm to solve this, however we will explore a different method which we will extend to solve the full problem.

We are guaranteed that the graph is a DAG (Directed Acyclic Graph) as each of the paths are one way and it is not possible to visit the same rest stop twice. Thus we can use a technique called DP (Dynamic Programming) on a DAG. We visit all nodes using a depth-first traversal, and calculate the minimum amount of exposure that we need to get from a given rest stop to rest stop $N - 1$. What we make sure is that we only calculate the minimum exposure for each rest stop once and traverse each edge exactly once. This makes the complexity of this solution $O(N + M)$.

Python Solution

The code for this solution can be found at the end of this document.

Subtask 4

In this subtask, we are guaranteed that the paths form a line, and therefore there is only one unique sequence of paths from rest stop 0 to rest stop $N - 1$. This simplifies our decision making at each rest stop. Instead of also considering which path to take next, we only need to consider whether to cross the next path now, or wait until the intensity lessens. We can solve this with another application of dynamic programming. If we are at rest stop v at the t-th second and it takes d seconds to travel to the next rest stop, we can either travel to the next rest stop now, which would accumulate sunlight intensities $s_t + s_{t+1} + ... + s_{t+d-1}$, or wait until the $t + 1$-th second. To minimise sun exposure, we simply take the minimum of these two choices. In other words, if we let $dp_{v,t}$ be the minimum sun exposure required to reach rest stop $N - 1$ starting at rest stop v at the t-th second, then $dp_{v,t} = \min\left(dp_{v,t+1}, dp_{v+1,t+d} + \sum_{k=t}^{t+d-1} s_k\right)$. As there are only N rest stops and T starting times for each rest stop that give distinct values, we can store the previously computed $dp_{v,t}$ values. In total, there are $N \times T$ dp values we need to calculate, giving our algorithm an $O(NT)$ complexity.

C++ Solution

The code for this solution can be found at the end of this document.

Subtasks 3 & 5

For subtasks 3 and 5, we are no longer guaranteed that there is one unique sequence of paths from rest stop 0 to rest stop $N - 1$, so we will also need to consider which path to take next. However we can extend our solution for subtask 4 for a general graph using similar ideas. Whereas previously we only considered whether to traverse the next path now or wait until the next second, now we must also consider which path to take if we choose to move to the next rest stop. Just like above, we will choose the minimum out of all these options to build our optimal solution. Our DP recurrence is now:

$$dp_{v,t} = \min\left\{dp_{v,t+1}, \min_{i \in adj_v}\left(dp_{b_i,t+d_i} + \sum_{k=t}^{t+d_i-1} s_k\right)\right\}$$

Where adj_v is the set of paths going out from rest stop v, b_i is the endpoint of the i-th path and d_i is the length of the i-th path.

Note that if $t \geq T$, since the brightness of the sun stops changing, it is no longer ever necessary to stop at on of the shaded spots. On top of that, it does not matter what the value of t actually is, since now that $t \geq T$, we will always get the same sequence of sunlight no matter the actual time. We can write this as:

$$dp_{v,t} = dp_{v,T} \text{ if } t \geq T$$

And, because we no longer stop at shaded spots and s_i is constant, we get the simpler dp recurrence:

$$dp_{v,T} = \min_{i \in adj_v} \left(dp_{b_i,T} + d_i I \right)$$

As in each $dp_{v,t}$ we need to consider each path directly connected to v, as well as summing the sun intensity for each path length, our time complexity comes to $O((N+M)TD)$, where D is the maximum of all d values. This is sufficient to pass subtask 3 and 5 as d is at most 3 in both subtasks for all paths.

C++ Solution

The code for this solution can be found at the end of this document.

Subtask 6

For subtask 6, d is no longer constrained and can be up to 500 so our solution from above will be too slow. To speed this up, we can use a precomputated prefix sum to quickly calculated the sum of sunlight intensities accumulated for each path in our DP. A prefix sum of an array stores the sum of its elements from 1 to k at index k. In other words, for an array a, its prefix will store a_1 at index 1, $a_1 + a_2$ at index 2, $a_1 + a_2 + a_3$ at index 3 and so on. This structure allows us to calculate a subarray sum in constant time after an $O(N)$ precomputation; to calculate for $\sum_{i=L}^{R} a_i$, we can simply take $\text{prefix}[R] - \text{prefix}[L-1]$. This reduces our time complexity down to $O((N+M)T)$, which is sufficient to solve the full problem.

C++ Solution

The code for this solution can be found at the end of this document.

Additional Example Solutions

Repdigits Subtask 2 Python Solution

```
N = int(input())

def is_repdigit(n):
  s = str(n)

  for i in range(1, len(s)):
    if s[i] != s[0]:
      return False

  return True

def find_repdigit(n):
  while not is_repdigit(n):
    n -= 1
  return n

values = []

while N > 0:
  next_val = find_repdigit(N)
  values.append(next_val)
  N -= next_val

print(len(values))

for val in values:
  print(val)
```

Repdigits Subtask 3 Python Solution

```
import sys

N = int(input())

def is_repdigit(n):
  s = str(n)

  for i in range(1, len(s)):
    if s[i] != s[0]:
      return False

  return True

x = 1
while x <= N:
  for i in range(1, 10):
    repdigit = x * i

    if repdigit <= N:
      res = N - repdigit
      if res != repdigit and is_repdigit(res):
        print(2)
        print(repdigit)
        print(res)
```

```
        sys.exit(0)
  x = 10 * x + 1
```

Repdigits Subtask 4 Python Solution

```
N = int(input())

repdigits = []
x = 1

while x <= N:
  for i in range(1, 10):
    repdigits.append(x * i)
  x = 10 * x + 1

values = []

for x in reversed(repdigits):
  if x <= N:
    values.append(x)
    N -= x

print(len(values))

for val in values:
  print(val)
```

Repdigits Subtask 4 C++ Solution

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  int N;
  cin >> N;

  vector<int> repdigits;
  long long x = 1;
  while (x <= N) {
    for (int i = 1; i <= 9; i++) {
      repdigits.push_back(x * i);
    }

    x = 10 * x + 1;
  }

  vector<int> values;

  for (int i = repdigits.size() - 1; i >= 0 && N > 0; i--) {
    if (repdigits[i] <= N) {
      values.push_back(repdigits[i]);
      N -= repdigits[i];
    }
  }

  cout << values.size() << endl;
```

```
  for (int x: values) {
    cout << x << endl;
  }
}
```

Emma's Switches Subtask 4 Python Solution

```
N, D = map(int, input().split())

switches = [0] * N
queries = [[0] * N for i in range(51)]

for i in range(D):
  k, p = map(int, input().split())
  queries[p][k-1] += 1

for p in range(1, 51):
  for s in range(p):
    count = 0
    for i in range(s, N, p):
      count += queries[p][i]
      switches[i] += count

for s in switches:
  print("ON" if s % 2 == 1 else "OFF")
```

Emma's Switches Subtask 4 C++ Solution

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

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

  int N, D;
  cin >> N >> D;
  vector<int> switches(N, 0);
  vector<vector<int>> queries(51, vector<int>(N, 0));
  for (int i = 0; i < D; i++) {
    int k, p;
    cin >> k >> p;
    queries[p][k-1]++;
  }

  for (int p = 1; p <= 50; ++p) {
    for (int s = 0; s < p; ++s) {
      int count = 0;
      for (int i = s; i < N; i += p) {
        count += queries[p][i];
        switches[i] += count;
      }
    }
  }

  for (int k : switches)
    cout << (k % 2 == 1 ? "ON\n" : "OFF\n");
}
```

Sunsprint Subtask 1 & 2 Python Solution

```python
import sys
sys.setrecursionlimit(5000)

I, T = map(int, input().split())
input()
N, M = map(int, input().split())
adjacency = [[] for i in range(N)]
for i in range(M):
    line = input().split()
    a, b, d = map(int, line[:3])
    if line[3] == "S":
        d = 0
    adjacency[a].append((b, d))

distances = [-1 for i in range(N)]

def dfs(node, parent):
    min_dist = 10**9
    if node == N - 1:
        min_dist = 0
    for child, dist in adjacency[node]:
        if child == parent:
            continue
        if distances[child] == -1:
            dfs(child, node)
        min_dist = min(min_dist, distances[child] + dist)
    distances[node] = min_dist

dfs(0, -1)
print(distances[0] * I)
```

Sunsprint Subtask 4 C++ Solution

```cpp
#include <bits/stdc++.h>

using namespace std;

int I, T, N, M;
int s[5000], d[5000], dp[2000][5001];
bool covered[5000];

int getCost(int path, int start) {
    if (covered[path]) return 0;
    int total = 0;
    for (int i = start; i < start + d[path]; i++) {
        if (i >= T) total += I;
        else total += s[i];
    }
    return total;
}

int solve(int node, int t) {
    if (t >= T) t = T;
    if (node == N - 1) return 0;
    if (dp[node][t] != -1) return dp[node][t];
```

```
    int res = getCost(node, t) + solve(node + 1, t + d[node]);
    if (t < T) res = min(res, solve(node, t + 1));

    return dp[node][t] = res;
}

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

    cin >> I >> T;

    for (int i = 0; i < T; i++) {
        cin >> s[i];
    }

    cin >> N >> M;

    for (int i = 0; i < M; i++) {
        int a, b;
        char c;
        cin >> a >> b;
        cin >> d[a] >> c;
        covered[a] = c == 'S';
    }

    memset(dp, -1, sizeof(dp));

    cout << solve(0, 0) << endl;
}
```

Sunsprint Subtask 3 & 5 C++ Solution

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

using namespace std;

int I, T, N, M;
int s[5010], d[5010], b[5010], dp[2010][5010];
bool covered[5010];
vector<int> adj[2010];

int getCost(int path, int start) {
    if (covered[path]) return 0;
    int total = 0;
    for (int i = start; i < start + d[path]; i++) {
        if (i >= T) total += I;
        else total += s[i];
    }
    return total;
}

int solve(int node, int t) {
    if (t >= T) t = T;
    if (node == N - 1) return 0;
    if (dp[node][t] != -1) return dp[node][t];
```

```
    int res = INT_MAX;

    for (int edge: adj[node]) {
        res = min(res, getCost(edge, t) + solve(b[edge], t + d[edge]));
    }

    if (t < T) {
        res = min(res, solve(node, t + 1));
    }

    return dp[node][t] = res;
}

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

    cin >> I >> T;

    for (int i = 0; i < T; i++) {
        cin >> s[i];
    }

    cin >> N >> M;

    for (int i = 0; i < M; i++) {
        int a;
        char c;
        cin >> a >> b[i] >> d[i] >> c;

        adj[a].push_back(i);
        covered[i] = c == 'S';
    }

    memset(dp, -1, sizeof(dp));

    cout << solve(0, 0) << endl;
}
```

Sunsprint Subtask 6 C++ Solution

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

int I,T,N,M;
int intensities[5555];

struct Edge
{
    int dest;
    int length;
    bool shaded;
};
vector<vector<Edge>> adj;

int dp[10010][5010];
```

```
int rsq[5555];

int cost(int time, Edge& edge)
{
    return edge.shaded ? 0 : rsq[time + edge.length] - rsq[time];
}

int solve(int node, int time)
{
    time = min(time, T);
    auto& result = dp[node][time];
    if (result != -1)
        return result;
    if (node == N-1)
        return result = 0;

    result = time >= T ? INT_MAX / 2 : solve(node, time+1);
    for (Edge& edge : adj[node])
        result = min(result, solve(edge.dest, time + edge.length) + cost(time, edge));
    return result;
}

int main()
{
    cin >> I >> T;
    for (int i = 0; i < T; i++)
        cin >> intensities[i];
    for (int i = T; i < T+500; i++)
        intensities[i] = I;
    rsq[0] = 0;
    for (int i = 0; i < T+500; i++)
        rsq[i+1] = rsq[i] + intensities[i];

    cin >> N >> M;
    adj.resize(N);
    for (int i = 0; i < M; i++)
    {
        int a,b,d;
        char c;
        cin >> a >> b >> d >> c;
        adj[a].push_back({b, d, c == 'S'});
    }

    memset(dp, -1, sizeof(dp));
    cout << solve(0, 0);
}
```

Big O Complexity

Computer scientists like to compare programs using something called Big O notation. This works by choosing a parameter, usually one of the inputs, and seeing what happens as this parameter increases in value. For example, let's say we have a list N items long. We often call the measured parameter N. For example, a list of length N.

In contests, problems are often designed with time or memory constraints to make you think of a more efficient algorithm. You can estimate this based on the problem's constraints. It's often reasonable to assume a computer can perform around 100 million (100,000,000) operations per second. For example, if the problem specifies a time limit of 1 second and an input of N as large as 100,000, then you know that an $O\left(N^2\right)$ algorithm might be too slow for large N since $100{,}000^2 = 10{,}000{,}000{,}000$, or 10 billion operations.

Time Complexity

The time taken by a program can be estimated by the number of processor operations. For example, an addition $a + b$ or a comparison $a < b$ is one operation.

$O(1)$ time means that the number of operations a computer performs does not increase as N increases (i.e. does not depend on N). For example, say you have a program containing a list of N items and want to access the item at the i-th index. Usually, the computer will simply access the corresponding location in memory. There might be a few calculations to work out which location in memory the entry i corresponds to, but these will take the same amount of computation regardless of N. Note that time complexity does not account for constant factors. For example, if we doubled the number of calculations used to get each item in the list, the time complexity is still $O(1)$ because it is the same for all list lengths. You can't get a better algorithmic complexity than constant time.

$O(\log N)$ time suggests the program takes a constant number of extra operations every time N doubles in size. For example, finding a number in a sorted list using binary search might take 3 operations when $N = 8$, but it will only take one extra operation if we double N to 16. As far as efficiency goes, this is pretty good, since N generally has to get very, very large before a computer starts to struggle.

$O(N)$ time means you have an algorithm where the number of operations is directly proportional to N. For example, a maximum finding algorithm `max()` will need to compare against every item in a list of length N to confirm you have indeed found the maximum. Usually, if you have one loop that iterates N times your algorithm is $O(N)$.

$O\left(N^2\right)$ time means the number of operations is proportional to N^2 . For example, suppose you had an algorithm which compared every item in a list against every other item to find similar items. For a list of N items, each item has to check against the remaining $N - 1$ items. In total, $N(N - 1)$ checks are done. This expands to $N^2 - N$. For Big O, we always take the most significant term as the dominating factor, which gives $O\left(N^2\right)$. This is generally not great for large values of N, which can take a very long time to compute. As a general rule of thumb in contests, $O\left(N^2\right)$ algorithms are only useful for input sizes of $N \lesssim 10{,}000$. Usually, if you have a nested loop in your program (loop inside a loop) then your solution is $O\left(N^2\right)$ if both these loops run about N times.