```
}
```

Now subtask 3 is complete - and we have finished the problem! The complete solution in C++ and Python is available below.

If you have any further questions - on this problem or other coding-related questions - feel free to email me at ats391@gmail.com and ask. Hope you enjoyed the problem!

C++ Solution

```cpp
#include <iostream>
#include <cstring>

using namespace std;
int main ()
{
    int alpha, beta;
    alpha = 0;
    beta = 90;

    int angle;
    string request;
    cin >> request >> angle;
    while (request != "END")
    {
        if (request == "BOOST")
        {
            beta += angle;
        } else if (request == "TRAVEL")
        {
            beta += angle;
            alpha += angle;
        }
        cin >> request >> angle;
    }
    beta %= 360;
    alpha %= 360;

    int missile;
    cin >> missile;
    cout << alpha << " " << beta << endl;
    if ((alpha < missile && beta > missile) ||
        (alpha == missile || beta == missile) ||
        (missile < beta && alpha > beta) ||
        (missile > alpha && alpha > beta))
    {
        printf("Missile located.\n");
    }
    else
    {
        printf("Missile unlocatable.\n");
    }

    return 0;
}
```

Python 3 Solution

```
action, angle = input().split()
alpha, beta = 0, 90

while action != "END":
    angle = int(angle)

    if action == "TRAVEL":
        alpha += angle
        beta += angle
    elif action == "BOOST":
        beta += angle

    action, angle = input().split()

alpha %= 360
beta %= 360

missile = int(input())

print(alpha, beta)
if alpha <= missile <= beta or missile <= beta <= alpha or beta <= alpha <=
missile:
    print("Missile located.")
else:
    print("Missile unlocatable.")
```

Twilight Sparkle's Magic Spells

Problem Statement

Full problem at: https://train.nzoi.org.nz/problems/836

Twilight Sparkle is researching some magic spells. A spell is a non-empty finite sequence of **A**pples, **B**ananas, or **C**iwifruit.

Spells can be safe or unsafe. An unsafe spell contains one or more of the following patterns:

Pattern	Name
AAAAA	Appleboom
ABBA	Stockholm Syndrome
ABCABC	Double Rainbow

Help Twilight determine whether a spell is safe or unsafe.

Input

The first line of input will contain a single integer, $0 < N \leq 100$, representing the length of the spell. The next line will contain a string of N letters, each either A, B, or C. This is the spell itself.

Output

Output SAFE if the spell is safe; otherwise UNSAFE.

Explanations

- BAAAAAB is unsafe, as it contains an Appleboom.
- AAAACA is safe. While it has five Apples, the Ciwifruit in between means that it can't form an Appleboom.
- ABBABBA is unsafe, as it contains two (overlapping) Stockholm Syndromes.
- ABBAAAAA is also unsafe, as it contains both a Stockholm and an Appleboom. They don't cancel out!
- CBACBA is safe. It has the same fruit as a Double Rainbow, but the fruit are in the wrong order.

Analysis

The first step to understand any problem is to break it down into smaller problems. Rather than think about finding all three patterns: what would we do if it was just one pattern we were looking for?

The easy way

Most languages have some inbuilt way of finding substrings. A substring is just a string of characters which is contained within another string. In Python, this function is called `find()`. In this case, `find()` returns the index at which a substring is found or -1 if it is not found. If one of the patterns is contained in a spell, we know the spell must be UNSAFE. Therefore, we want to print UNSAFE if `find()` does not return -1.

Suppose that `find()` does return -1. In this case, we don't know if the spell is SAFE or UNSAFE because we haven't checked the other patterns yet. We can use a loop to repeat the above step once for each pattern. If we check all patterns and discover none of them present in the spell, then the spell must be SAFE.

This method is implemented as the Python 3 Solution below.

The hard way

Maybe you can't remember what the `find()` equivalent is for your language, or you just want an extra challenge? In this case, we don't have the luxury of a way to easily determine if a pattern is contained in a spell. This might make the problem seem a lot harder, but we can use the same "break it down" approach as before.

A neat trick you can use when writing code is to pretend you do have a magical `find()` function which does what you want. In this case, we would solve this problem the easy way (as above). Then, all we need to do is create our own `find()` function! Let's define what this function should do… It must return some number if a pattern is found or -1 if it is not found.

Breaking it down even more, we can think of our pattern and spell each as a list of characters. Then, using a loop, we can determine if the pattern is present in the spell. Specifically,

1. Loop through each character s in spell.
2. Check if this character is the first character in our pattern.
    a. If it is, then move to the next character in spell and check if this is the second letter, and so on.
        i. If we run out of character in pattern, then we have found our pattern, so return some number.

   b. If it is not the first character in our pattern, then move to the next character in spell and repeat from step 2.

3. If we reach the end of spell without detecting a pattern, return -1.

Your find function might look something like this…

```
def find(string, pattern):

    i = 0
    for s in string:
        if s == pattern[i]:
            if i == len(pattern) - 1:
                return i

            i += 1

    return -1
```

Python 3 Solution

```
N = int(input())
spell = input()

patterns = ["AAAAA", "ABBA", "ABCABC"]

is_safe = True
for pat in patterns:
    if spell.find(pat) != -1:
        is_safe = False
        break

print("SAFE" if is_safe else "UNSAFE")
```

If we want to use our own `find()` function, then replace the line

```
    if spell.find(pat) != -1:
```

with

```
    if find(spell, pat) != -1:
```

Python 3 One Line Solution Just for Fun :)

```
print("SAFE" if input() and all(map(lambda a: not (a[0].find(a[1]) + 1),
    zip([input()] * 3, ["AAAAA", "ABBA", "ABCABC"]))) else "UNSAFE")
```

Java Solution

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(system.in);
        boolean isUnsafe = false;

        int N = scan.nextInt();
        String spell = scan.next();

        for (int i = 0; i < N; i++) {
            if ( (i + 4 <= N && spell.substring(i, i + 4).equals("ABBA")) ||
                 (i + 5 <= N && spell.substring(i, i + 5).equals("AAAAA")) ||
                 (i + 6 <= N && spell.substring(i, i + 6).equals("ABCABC")) )
            {
                isUnsafe = true;
                break;
            }
        }

        if (isUnsafe) {
            System.out.println("UNSAFE");
        } else {
            System.out.println("SAFE");
        }
    }
}
```

This was a student's solution.

Myrtle Rust

Problem Statement

Full problem at: https://train.nzoi.org.nz/problems/847

Myrtle Rust, a wind-borne fungus which effects native NZ plants, has made its way into a nursery in the upper North. Every morning, trucks carrying plants (which might be infected) leave their starting nursery and make it to a corresponding destination nursery at the end of the day. If the truck contains an infected plant, all plants in the destination nursery will become infected overnight. This means all the trucks which leave form this nursery the next day will carry infected plants!

Given a description of all nurseries and how plants are moved between them, write a program to calculate how many days it will take for North Island nurseries to become completely infected with Myrtle Rust.

Input

The first line will contain three numbers separated by a single space. The first, N, is the number of North Island nurseries, the second, Z, is the number corresponding to the nursery which became infected first, and the last, T, is the number of trucks in the nursery network. Note that nurseries are numbered from 0 up to $N-1$.

What follows are T lines, one for each truck. Each line will contain two space-separated integers. The first, s, is the number corresponding to the nursery in which the truck starts at, and the second, d, is the truck's destination nursery. Every nursery will be connected to the network via a truck.

Output

- Print a single number representing how many days it takes for the North Island to become infected with Myrtle Rust.
- If the North Island never becomes fully infected with Myrtle Rust then print -1.

Subtasks

- (60%) The truck's delivery routes form a single chain of nurseries. E.g. Truck 0 goes from nursery 1 to nursery 2 and truck 1 goes from nursery 2 to nursery 3
  to form a continuous open-ended chain.
- (40%) The full problem.

Analysis

When a question talks about "connections" or a "network", it strongly indicates the presence of a graph problem. However, the first subtask does not require any knowledge of graphs to produce a solution.

What's a graph?

Graphs are a way of representing connections between things. They are made of **nodes** (the items or values) and **edges** (the arrows between items and values). If you have not heard these terms before, check out this introduction to graphs: https://youtu.be/82zlRaRUsaY?t=46s

A common mistake

Many people failed the case where $N = 1$. If there is only one nursery, and it is infected to begin with, then the correct answer is that it takes zero days for all nurseries to become infected; not one day. Many students missed the full solution because of this single case :(.

Because this simple case was only checked for the first 60% subtask, a full solution which failed this case was able to get 40%.

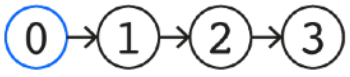
Tip: Think of some simple edge cases and test them on your program before submitting.

Subtask 1

For this subtask, we know the nurseries are connected in a single chain. In other words, each nursery can only have one or two delivering/departing trucks. Let's look at the first example case given in the question…

```
4 0 3
0 1
1 2
2 3
```

Drawing pictures can be super helpful. We can draw this as:



where 0, the blue circle, is the first infected nursery and the arrows show truck deliveries between nurseries. So how many days does it take for all nurseries to become infected? Let's break it down…

1. Initially, on day 0, nursery 0 is infected.

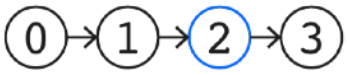
2. After the first day, a truck traveling to nursery 1 carries infected plants, causing 1 to be infected.
3. After the second day, a truck traveling from nursery 1 to nursery 2 infects nursery 2.
4. Finally, on the third day, a truck going to nursery 3 causes it to become infected.
5. Since nursery 3 was the last nursery to be infected, it took three days to infect the whole network.

So, one solution might be to start at the initially infected nursery and follow the arrows until we reach the end. The number of days is then the number of arrows we followed. However, there is one case we are missing.

Let's consider the third example case given in the question…

```
4 2 3
0 1
1 2
2 3
```

which can be drawn as



This is very similar to the first example, except that the initially infected nursery is 2. If we apply our strategy "follow the arrows" we will end up at nursery 3. However, not all the nurseries are infected. In fact, no matter how much time passes, the whole network will never be infected. Therefore, we must print -1 to indicate this.

Overall, for the first subtask, we require three steps

1. Follow the arrows from the initially infected nursery and count the number of arrows followed as the number of days.
2. When there are no more arrows to follow, check if we have infected all the nurseries in our network.
3. If all nurseries are infected then print the days. Otherwise, print -1.

Note: Take a look at the below example…

We need to be careful when the infected nursery has two trucks leaving from it. In such a case, we must follow both arrows and then choose the longest chain as the number of days. It would take 2 days in this case, not 1.

A solution for subtask 1, in Python 3, might look like this:

```
N, Z, T = map(int, input().split())

# Read in connections between nurseries into a list of lists
connections = [[] for _ in range(N)]
for i in range(T):
    s, d = map(int, input().split())
    connections[s].append(d)

# Start at Z and follow the first arrow while keeping track of length
cur = Z
length_1 = 0
while len(connections[cur]) != 0:
    cur = connections[cur][0]
    length_1 += 1

# Check for a second arrow. If it exists, follow it and keep track of length
length_2 = 0
if len(connections[Z]) == 2:
    length_2 = 1
    cur = connections[Z][1]
    while len(connections[cur]) != 0:
        cur = connections[cur][0]
        length_2 += 1

# If the total length from following both chains is the same as the total
# number of nurseries then print the largest of the two lengths
if length_1 + length_2 != T:
    print(-1)
else:
    print(max(length_1, length_2))
```

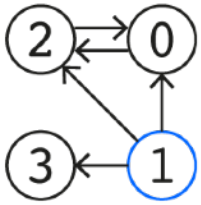
Notice how we store the connections between nurseries. This means we can get a list of the destinations for all trucks leaving from some nursery A with the statement connections[A].

Subtask 2

For the remaining part of the problem, the number of connections per nursery is unrestricted. The second example from the question was

```
4 1 5
1 3
1 2
0 2
1 0
2 0
```

Which can be drawn as



In this case, the network takes one day to become infected.

Breadth-First Search

We can use the Breadth-First Search (BFS) algorithm to traverse our graph and keep track of the distance from our starting nursery. BFS is a common technique for graph problems and there are heaps of resources online about it. Check it out.

Python 3 Solution

```python
# Read in connections between nurseries
connections = [[] for _ in range(N)]
for i in range(T):
    s, d = map(int, input().split())
    connections[s].append(d)

# Keep track of which nurseries have been infected
infected = [False for _ in range(N)]

# BFS
# Keep a list of nurseries to visit next. Start at the initially infected
# nursery which is 0 distance from itself.
todo = [(Z, 0)]
max_dist = 0
while len(todo) != 0:
    cur, dist = todo.pop(0)
    if infected[cur]:
        continue

    infected[cur] = True
    max_dist = max(max_dist, dist)

    for n in connections[cur]:
        # Increase the distance from the starting nursery by one
        todo.append((n, dist + 1))

if False in infected:
    print(-1)
else:
    print(max_dist)
```

Bad Dragon

Problem Statement

Full Problem at: https://train.nzoi.org.nz/problems/840

You are the knight in shining armour ready to rescue your princess, but she is guarded by the big bad dragon who is very scary. So scary that you do not dare to face the dragon directly. Luckily, the wizard in your party can cast a spell to make the big bad dragon go to sleep for a certain amount of time. The big bad dragon is not scary when it's sleeping.

Given this knowledge, you have decided to plan and make the best use of time that the dragon is asleep and maximize the amount of hit point damage you can deal to the dragon before it wakes up.

The wizard can sleep the dragon for T seconds. You have 100 mana initially and S skills at your disposal. Each skill costs m_i mana and takes t_i seconds to cast and deals h_i hit points of damage. Each skill can only be used once. In addition, depending on your mood of the day, you either regenerate 1 or 0 mana per second. However, your mana can **never** exceed the maximum capacity of 100.

The damage of a skill is dealt at the end of its cast duration and the mana is used at the start of a cast. For example, suppose you regenerate 1 mana per second and have a skill with 10 seconds of cast time. If you cast your skill as soon as the wizard sleeps the dragon, you will deal hit point damage at exactly 10 seconds after the wizard has cast their spell. You will also regenerate up to 10 mana in this time, but you must have had sufficient mana to have cast the skill initially.

You want to know, after T seconds, what is the maximum amount of hit points you can deal to the dragon.

Input

- The first line contains two integers $1 <= T <= 100$, $1 <= S <= 100$, $R = 0 \text{ or } 1$, which are respectively: the numbers of seconds your wizard can sleep the dragon, the number of skills you have available, and the mana regenerated per second depending on your mood.
- The next S lines each contain three integers $0 <= m_i <= 100$, $1 <= t_i <= T$, and $1 <= h_i <= 1000$ which describes the mana cost, seconds to cast, and the amount of hit point damage skill i does.

Output

Please output one integer on one line with its value being the maximum amount of hit points you can deal to the dragon after T seconds.

Keep in mind this question takes a while to be tested on the server. Don't waste time waiting for it to complete. Move to the next question.

Subtasks

- Subtask 1 (25 points): Each skill costs exactly 33 mana and there is no regeneration.
- Subtask 2 (25 points): You have access to 20 and only 20 skills and there is no regeneration.
- Subtask 3 (25 points): your skills cost 0 mana.
- Subtask 4 (50 points): No additional limits.

Note: You always start with 100 mana and as you are not the most energetic knight in the kingdom you do not have to be casting a skill for the entire duration that the dragon is asleep for. Given you optimise the number of hit points dealt you're free to take a break for as long as you want.

Analysis

Essentially the task is to utilise a set of skills that can fit within the time and mana limits that maximizes the hit point value.

Mana can either regenerate at 1 per second or no regeneration. Luckily 60% of the points do not depend on this and for subtasks 1, 2, and 3 we can disregard regeneration.

A couple of observations to start with

- There’s no need to order the use of skills, using skill A before or after skill B does not change the hit point outcome, mana cost, or time cost.
- Many of the subtasks have unique restraints enabling you to solve those via unique methods and gain partial scores (this is extremely important to maximize your score should you not arrive at a full solution).
- There’s essentially two choices for each skill: use it or don’t use it.

Let's look at the subtasks

1. For the first subtask: you have 100 mana that does not regenerate and all skills cost 33 mana. From this we know we can only use a maximum of 3 skills.

    a. Assuming we will use 3 skills, then we can simply try all combinations of 3 skills and record down the highest hit point value, this results in $O((\text{number} - \text{of} - \text{skills})^3)$ time complexity.

    b. But, what if we only had time to use 2 skills or even just 1 skill? To resolve this you could have special case and tried again to try all combinations of 2/1 skill(s). However, a simple trick is to add 2 skills with mana cost 0, time cost 0, and hit point damage of 0. E.g. If we only use 2 skills then the 3rd skill can be one of the newly added "blank" skills.

```
# Brute force solution assuming 33 mana for each case

T, S, R = map(int, input().split())

if R == 1:
    # We don't know how to do this so quit now
    exit()

skills = []
for i in range(S):
    # Get skills as a tuple of the form (mana, time, damage)
    skills.append(tuple(map(int, input().split())))

# Add two empty skills so that having less than 3 is an option
skills.extend([(0, 0, 0)] * 2)
S += 2  # Account for extra skills

def damage(T, skills, a, b, c):
    """Given a time T and a list of skills, choose skills a, b, c and
    return the total damage gained. If the skills form an invalid
    combination (i.e. take too much time) then reutrn 0."""

    # Sum up all mana, time, and damage for the combination a, b, c
    mana = sum(skills[i][0] for i in (a, b, c))
    time = sum(skills[i][1] for i in (a, b, c))
    dam = sum(skills[i][2] for i in (a, b, c))

    if time <= T:
        return dam

    return 0

best_damage = 0
for a in range(S):
    for b in range(S):
        for c in range(S):
```

```
            # We can't use the same skill more than once
            if a == b or b == c or a == c:
                continue

            # Check if this combination is better
            best_damage = max(best_damage, damage(T, skills, a, b, c))

print(best_damage)
```

2. For the second subtask: you always have 20 skills.

    a. Often, it is useful too look out for small limits (less than 25) for problems or sub-problems that have binary choices. In this case, the choice is: do we use a skill or do we not use a skill. Binary means two and there are only two things to choose from.

    b. Given this observation, why don't we try every combination of skills?

    c. An analysis on the complexity of doing so gives us a time complexity of $O(2^{\text{number}-\text{of}-\text{skills}}) \sim = 1\,\text{million}$. This should be just enough to run in 4 seconds for Python. In contrast, an efficient, optimised, simple brute force in C or C++ can handle up to 100 million operations per second.

3. For the 3rd subtask, we remove the dimension of mana from the problem and it reverts to a classic knapsack problem.

    a. DP, or Dynamic Programming, isn't straight forward to explain, but for your interest you may want to read:

    https://en.wikipedia.org/wiki/Knapsack_problem

    b. Essentially the "weight" is the time taken, and the "value" is hit point damage.

    c. For an full explanation of the DP solution for this problem, please refer to the subtask 4 solution.

    Recursive knap-sacks implementations were unfortunately too slow in Python. The below student submission is a good example of a recursive knap-sack. Some people find recursive solutions easier to understand than iterative solutions. Comments have been added to their solution.

```
# A very nice student knap-sack solution using recursion. This solution
# received 40% but could have gotten 80% if implemented using a table
T, S, R = list(map(int, input().split()))

skills = []
for _ in range(S):
```

```
    skills.append(tuple(map(int, input().split())))


# Recursive knap-sack
def knap(mana, time, n):
    # Base case
    if n == -1:
        return 0

    if skills[n][0] > mana or skills[n][1] > time:
        # We don't have enough time or mana so don't take the n'th skill
        return knap(mana, time, n - 1)
    else:
        # Pick the best option between choosing the n'th skill and not
        # choosing it
        choose = skills[n][2] + \
            knap(mana-skills[n][0], time-skills[n][1], n - 1)
        dont_choose = knap(mana, time, n - 1)
        return max(choose, dont_choose)

# Start with 100 mana
print(knap(100, T, S - 1))
```

4. For the last subtask, we have the classical 2 dimensional knapsack with a twist - that one of the "weights" can regenerate.

    a. To set up our state, let's define an array dps[TIME][MANA] = highest amount of damage that can be done in TIME seconds and having MANA remaining.

    b. First, let's ignore mana regen: how would these states transition from one to another? We go back to our observation about the usage of skills: we can either use or not use a skill.

    So, given any dps[TIME][MANA], if we use a skill s_i which has a time t_i, some mana m_i, and damage d_i, then move to another state dps[TIME + t_i][MANA - m_i] and see if dps[TIME][MANA] + d_i will become the new highest damage for that state (After using the skill: New time = TIME + time used for skill, New mana = MANA - mana used for skill).

    In pseudo code:

```
dps[TIME+t_i][MANA-m_i] = max(dps[TIME][MANA] + d_i, dps[TIME+t_i][MANA-m_i])
```

    c. Now if we want to know what's the highest damage dealable at time T, we just want to know what is dps[T][any], as we can have any mana remaining at the end.

How can we get this value? We can do this either recursively or iteratively. One observation to note is on iterative solutions, you do not actually have to record whether or not you've used a skill or not should you start from the end and move towards the front. For example, consider a skill that takes 2 seconds and T = 5. At 5 sec we consider moving from 3 sec -> 5 sec, then we look at 4 sec (2->4), 3 sec (1->3), 2 sec (0->2) - as you can see there are no chance that you can use a skill twice.

d. How do we incorporate mana regeneration? We need to ensure we have enough MANA to use a skill when moving from dps[TIME][MANA] -> dps[TIME + t_i][MANA - m_i]. Thus, we can add the time it takes the skill to be casted onto the final mana outcome.

   I.e. Instead of dps[TIME][MANA] -> dps[TIME + t_i][MANA - m_i], now we move from dps[TIME][MANA] -> dps[TIME + t_i][MANA - $m_i + t_i$]. This observation makes up the last 20% of the problem.

Use subtasks as a hint

Sometimes the solution is not just in the task but also the limitations. It is helpful to have a good understand of algorithm complexity for these subtasks as each limitation gives you a hint to a solution.

- Being only able to use 3 skills meant you could have 3 nested loops going through a maximum of 100 skills.
- Having only 20 skills meant you could check to use or not use each one.
- When skills cost 0 mana, we can just pretend that the mana limit does not exist and simplify our problem.

What can lead you to think Dynamic Programing (DP)? Ask yourself:

- Can we simplify the state into a table (dps[TIME][MANA]) that fits into memory?
- Is the transition between states, i.e. dps[TIME][MANA] -> dps[TIME + t_i][MANA - $m_i + t_i$], straight forward?
    - One tip is if you can multiply all the state bounds and transition range together to fit within a ballpark complexity then you could give this a thought (in this case, TIME x MANA is the state and there's only 2 transition choices).

Overall, if DP is a relatively new concept then it can be quite difficult to grasp, but with more exposure and experience to the field they become much easier to spot and disseminate. So many common problems

can boil down to a simple DP solution, so it’s worth investing your time in getting to know if a bit better.

C++ Solution

```cpp
#include <cstdio>
#include <algorithm> // for max function

using namespace std;

const int MAX_TIME = 123, MAX_MANA = 123;

int t, s, r;

// this stores the highest amount of damage that can be dealt in
// dps[time_in_seconds][mana_remaining]
int dps[MAX_TIME][MAX_MANA];

int main() {
    scanf("%d%d%d", &t, &s, &r);

    for (int i = 0; i < s; i++) {
        int mana, time, dmg;

        scanf("%d%d%d", &mana, &time, &dmg);

        // by iterating from back to front we ensure a skill is only used
        // once
        for (int j = t - time; j >= 0; j--) {
            for (int k = 100; k >= 0; k--) {

                // ensure we currently have enough mana to cast this skill
                if ((k - mana) >= 0) {
                    // checks if using skill i from dps[j-time][k-mana]
                    // results in higher damage than what we have so far
                    dps[j + time][min(100, k - mana + r*time)] =
                        max(dps[j + time][min(100, k - mana + r*time)],
                                                dps[j][k] + dmg);
                    // r*time is the amount of mana regenerate, we must make
                    // sure we do not generate more than the maximum amount
                    // of mana
                }
            }
        }
        for (int j = 1; j < t; j++) {
            for (int k = 100; k >= 0; k--) {
                // do nothing this second
                dps[j][min(100, k + r)] =
                    max(dps[j][min(100, k + r)], dps[j - 1][k]);
            }
        }
    }
    for (int i = 100; i > 0; i--) {
        dps[t][i - 1] = max(dps[t][i - 1], dps[t][i]);
    }
    int ans = dps[t][0];
```

```c
    // 100 mana is cap mana
    printf("%d\n", ans);
    return 0;
}
```

Python 3 Solution

```python
MAX_TIME, MAX_MANA = 123, 123

t, s, r = map(int, input().split())

# Create a list of lists (look up list comprehensions if the notation is
# confusing) which stores the highest amount of damage that can be dealt in
# dps[time_in_seconds][maina_remaining]
dps = [[0 for _ in range(MAX_TIME)] for _ in range(MAX_MANA)]

for i in range(s):

    # Get the next skill
    mana, time, dmg = map(int, input().split())

    # By iterating from back to front we ensure a skill is only used once
    for j in range(t - time, -1, -1):
        for k in range(100, -1, -1):

            # Ensure we currently have enough mana to cast this skill
            if k - mana >= 0:
                # Checks if using skill i from dps[j - time][k - mana]
                # results in higher damage than what we have so far
                dps[j + time][min(100, k - mana + r*time)] = \
                    max(dps[j + time][min(100, k - mana + r*time)], \
                                                dps[j][k] + dmg)
                # r * time is the amount of mana regenerate, we must make
                # sure we do not generate more than the maximum amount of
                # mana


    for j in range(1, t):
        for k in range(100, -1, -1):
            # Do nothing this second
            dps[j][min(100, k + r)] = \
                max(dps[j][min(100, k + r)], dps[j - 1][k])

for i in range(100, 0, -1):
    dps[t][i - 1] = max(dps[t][i - 1], dps[t][i])

# 100 mana is cap mana
print(dps[t][0])
```

Mindy's Challenge

Problem Statement

Full problem at: https://train.nzoi.org.nz/problems/846

Mindy has a collection of wonders, each meticulously labelled with a unique identification number starting from 0. However, not only has Mindy given two items the same ID by mistake, little gremlins have changed some IDs to match numbers which have already been assigned a wonder in the collection. What a mess! Luckily, Mindy knows that she has N items in her collection where the maximum assigned ID number is $N-2$. She has asked for your help to track down just one of the duplicate IDs.

Given a list of ID numbers, develop an algorithm to find one of the duplicate IDs given the constraints below. It doesn't matter which duplicate you find if there are multiple to choose from.

1. Use a template* given below to read the ID list/array into memory. This part has no associated points and is not considered as part of your algorithm.
2. Use $O(1)$ extra space**. Explained more below.
3. Do not change the input array referred to as ids. Consider it as read-only.
4. You are limited to $O(N)$ accesses of the list/array. Therefore, your algorithm should run in $O(N)$ time.

Warning: this problem is not the same as the Mindy one in Round 2.

**If your language is not available, you may write your own template which does the equivalent.*

***Beware that you cannot use data structures such as sets or dictionaries as these allocate extra memory behind the scenes. Some sorting algorithms do this as well. However, you should not require any of these things to solve this problem.*

Input

The integer N on a single line followed by N lines each containing one ID.

Output

One of the duplicate ID numbers on a single line. There will always be at least one duplicate.

Analysis

An explanation of O notation is provided with the question above. If it still doesn't make sense, try this link: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

Essentially, we need to find a duplicate number in a list. What makes this problem hard is not the problem itself, but rather the constraints on the solution. Some students may have recognised this problem from the previous round. However, there are some differences. The first major difference is that numerous duplicates are allowed in a list. The second major difference is with the solution constraints. Specifically, the solution must:

1. Use $O(1)$ extra space;
2. Not change the input list (i.e. the list is read-only); and
3. Be limited to $O(N)$ accesses of the array.

The first constraint instantly rules out many obvious solutions. For example, you might think to go through each item in the list checking if you have already seen that item or not. However, to remember all the previously seen numbers requires storage in a list/array, a set, or some other data structure. All of these storage methods require extra memory space which is proportional to the number of items in our list. We are only allowed to use a fixed amount of storage for our algorithm.

Many people tripped up on the second constraint. A common attempt involved using the sign of each number to indicate if it had already been seen or not. However, changing the values of items in the input list is not allowed here. This constraint differs from the question in the previous round.

Finally, your solution must complete using only $O(N)$ accesses of the list. This puts nested loops out of the question.

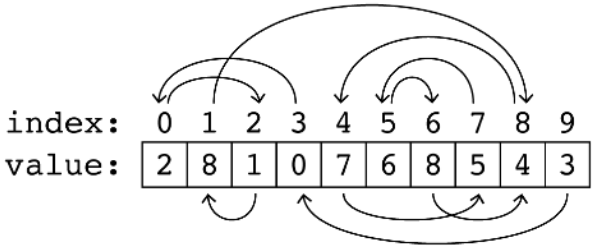
One possible solution is shown below, in C++ and Python 3, along with a detailed explanation later in this document.

Solution Explanation

Looking at the code below it might seem a bit magical. That's because there are a few key observations that someone must make to arrive at this solution.

Take the first sample case with the list `[2, 8, 1, 0, 7, 6, 8, 5, 4, 3]` for example. We can draw this where the next item in this list is found

by going to the index position corresponding to the current value e.g. `next_item = ids[current_item]`. It would look like this…



Where the index is the position of each item in our list and the value is the number at that position in our list. Drawing it using arrows like this is called a graph, made of **nodes** (the values) and **edges** (the arrows). If you have not heard these terms before, check out this introduction to graphs: https://youtu.be/82zlRaRUsaY?t=46s.

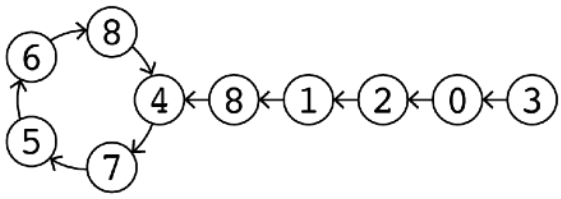
There are some interesting properties we can notice about a graph made from the list of IDs.

1. **There is always at least one duplicate node.** This was given to you in the question. However, there is an important connection to our graph visualisation as we will see later.

   For interest sake, it wouldn't have mattered if the question had told you this or not. We can prove this is the case… As stated in the problem, the list only contains values between 0 and $N-2$. This means every item will point to a valid index in the list. Also, since we have an item at every position from 0 up to $N-1$, then not all the items can be assigned a unique value. We have at least one more item than we have possible values. i.e. there is at least one duplicate.

2. **There is always at least one loop**. Since every item in the list contains a value between 0 and $N-2$, then the next value in our chain will always be at some valid index in the list. Therefore, we are guaranteed to be stuck in a loop at some point.

3. **One duplicate is in a loop, and one is outside a loop.** We can visualise this better by redrawing the graph from above without the list structure and index labels.



The circular bit on the left we'll call the 'loop' and the long bit on the right we'll call the 'tail'. Look at the two `(8)` nodes. Both of these point to the `(4)` node. But how do we know this will be the case for every possible list? And if it is the case, how do we find both the loop and the tail parts of the graph? The key lies in the final observation…

4. **The last node cannot be part of a loop.** This is perhaps the most important observation. We know that every node from index 0 up to $N-1$ has a value between 0 and $N-2$. Notice that there is one less available value than we have indices. In fact, the index which is missing from the available values is the last one, a.k.a. $N-1$. This means that no item in the list can have the value $N-1$ and, therefore, no node can point (have an edge) to the last item in the list. This means the last item can never be part of a loop.

This is good news! From observation 2, we know that from any node we must eventually end up in a loop. Therefore, by starting at the last node, we have a way to find both a tail and a loop which are connected (by observation 4). Then, by observation 3, we know that for a tail to join a loop, two nodes must point to the same index i.e. have the same value. Therefore, all we need to do is design an algorithm which finds both these nodes. In other words, we need to find where the tail joins the loop. Let's do it!..

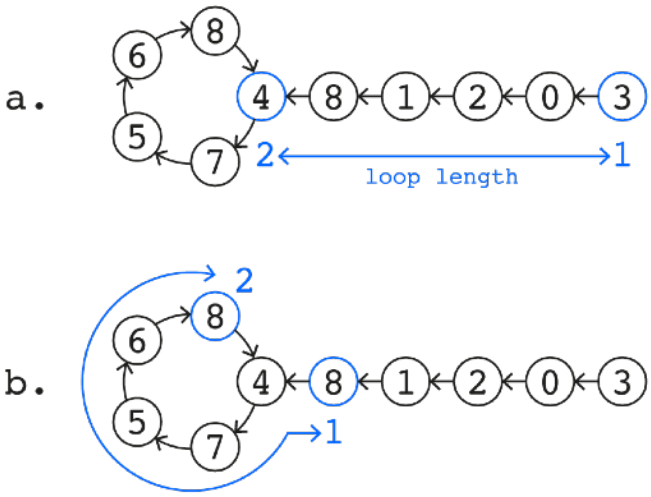
1. **Find the loop.** We could use a loop detection algorithm such as Floyds Loop Detection algorithm. However, we know that our loop can never be longer than N, so if we just moved along our graph N times, starting from the last item, then we must end up in a loop.

2. **Calculate the loop length.** We will need this information for the next few steps. To do this, keep moving around the loop, counting each transition between nodes, until you end up where you started.

3. **Set up two pointers on the tail.** The second pointer must be one loop length ahead of the end of the tail. Why? So that when we step through the graph, the second pointer will come around the loop to meet the first pointer at the node where the tail intersects with the loop. This is explained further in the next step…

4. **Move both pointers until they meet at the intersection.** This process is visualised below where `1` is the first pointer and `2` is

the second pointer and the distance between them is the loop length we calculated (see a.). As the pointers both move together to the left, the stay loop length apart (see b.). Eventually, they will arrive at the same value which is our duplicate! In this case, it is 8.



As proven above, this method works for all cases including multiple duplicates. The code shown below implements this algorithm in Python 3 and C++. We didn't need to make any changes to the values in the list and no extra storage space was required. Also, because we are only using single loops which depend on N, we use $O(N)$ accesses of the list (explained more below). Therefore, all of our requirements are met. Yay!

Space and Access Analysis

Additional memory storage is required for the first, second, and length variable values. These storage requirements don't change as N increases so we are using $O(1)$ extra space.

Reading from the top of the code below… Step (1.) uses N access to move the first pointer. Step (2.) can't use more than N accesses to find the loop length since a loop cannot be longer than the list. This means step (3.) also requires a maximum of N since the loop depends on length. Finally, moving first and second around the loop as part of step (4.) requires about $2N$ accesses, since we access the list twice for each iteration of the while loop. Overall, the maximum number of accesses comes to around $5N$. Then, ignoring constants, we see the number of accesses are proportional to N and the number of accesses is thus $O(N)$.

Python 3 Solution

```python
N = int(input())
ids = []  # A list of Mindy's IDs
for i in range(N):
    ids.append(int(input()))

# Create a pointer to the last list item
first = N - 1

# (1.) Move the first pointer N times to ensure it gets trapped
# into a loop
for _ in range(N):
    first = ids[first]

# (2.) Calculate the length of the loop we are trapped in using a
# second pointer
length = 1
second = ids[first]
while second != first:
    length += 1
    second = ids[second]

# Now that we know the length of a loop, reset both pointers to
# the end of the list again
first = second = N - 1

# (3.) Move the second pointer 'length' ahead of the first
for _ in range(length):
    second = ids[second]

# (4.) Move both pointers together until they meet up. The place
# where they meet up will be the same as the place where the first
# pointer enters the loop
while first != second:
    first = ids[first]
    second = ids[second]

# Both first and second end up in the same place. However, the
# first pointer entered the loop from outside whereas the second
# pointer was already trapped in a loop. When the while above
# exits, we know that first and second have the same value.
# However, first and second came from different places.
# Therefore, we can conclude that first and second are duplicates
print(first)
```

C++ Solution

```cpp
#include <iostream>
int ids[1000];  // An array of Mindy's IDs

int main()
{
    int N;
    std::cin >> N;
    for (int i = 0; i < N; i++) {
        std::cin >> ids[i];
    }

    // Use pointers 'first' and 'second' to traverse the graph
    // starting at the last node
    int first = N - 1, second = 0;

    // Follow the chain N times to guarantee ending up in a loop
    for (int i = 0; i < N; i++) first = ids[first];

    // Determine loop length
    int length = 1;
    second = ids[first];
    while (second != first) {
        length++;
        second = ids[second];
    }

    // Restart from end node
    first = second = N - 1;

    // Move the second pointer to 'length' ahead of the first
    for (int i = 0; i < length; i++) second = ids[second];

    // Walk around until the second pointer meets with the first
    // at the entry to the loop
    while (first != second) {
        first = ids[first];
        second = ids[second];
    }

    // Both nodes contain the same ID, so this is one solution
    std::cout << first << std::endl;

    // O(N) time and accesses. O(1) extra space
}
```