New Zealand Informatics Competition 2021
Round 1 Solutions

March 7, 2021

Overview

Questions

The solutions to these questions 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?

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

Read about how the server marking works:

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

Ever wondered why your submission scored zero?

`Why did I score zero? - some common mistakes`

See our list of other useful resources here:

`www.nzoi.org.nz/nzic/resources`

Tips for next time

Remember, this is a contest. The only thing we care about is that your code runs. It doesn't need to be pretty or have comments. There is also no need to worry about invalid input. Input will always be as described in the problem statement. For example, the code below is not necessary.

```python
# Not needed
def error_handling(prompt):
   while True:
       try:
           N = int(input(prompt))
           if N < 0 or N > 100:
               print('That was not a valid integer!')
           else:
               return N
       except ValueError:
           print('Not a valid integer')
...
```

There are a few other things students can do to improve their performance in contests.

Practice getting input

A number of students tripped up on processing input with multiple integers on a single line. A neat trick for processing this sort of input in Python is to use the `str.split()` method and the `map()` function. The `split()` method will break up a string at space characters, returning a list of the words. The `map()` function can be used to apply `int()` to each string in this list, converting them to integers. For example, suppose we have the following line of input:

```
1 4 2 7
```

We can turn this into a list of integers with the Python statement

```python
my_ints = list(map(int, input().split()))
```

Notice that we used `list()`. This is because `map()` returns us a special generator object, not a list. However, generator objects are easily converted to lists.

We suggest having a go at some of the NZIC Practice Problems.

Move on to the next question

If you are spending too much time on a question, move on. There could be easy subtasks waiting in the next question. Often, you will think of a solution to your problem while working on another question. It also helps to come back to a question with a fresh pair of eyes.

Take time to read the questions

Don't underestimate the value of taking time to read and understand the question. You will waste exponentially more time powering off thinking you have the solution only to discover you missed something obvious.

In the real world, it is very rare to be provided with a problem in a simplistic form. Part of the challenge with these contests is reading and understanding the question, then figuring out what algorithm will be needed. Before submitting, check to make sure you have done everything the question asks you to do.

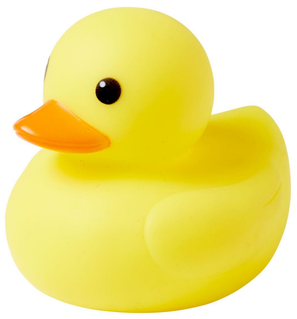
Test your code first!

It is much more time efficient to test your solutions on your own computer first. It can take a while to wait for the server to run your code. It is far more efficient to test it yourself first, submit it, then start reading the next question while you wait for your previous solution to be marked by the server.

While testing, remember to add some edge case tests. For example, if a question specifies "$1 \leq N \leq 10$" then try out an input where $N = 1$. Think of other tricky inputs that might break your code. Look at the feedback the server is giving you.

Use a rubber duck

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



Emma's Socks

https://train.nzoi.org.nz/problems/1213

There will always be two of every sock colour, except for the missing colour, which will only have one. We count the number of each colour of socks, and print out the colour that has a count of 1.

Python Solution

```python
count_red, count_blue, count_purple, count_pink = (0, 0, 0, 0)
for i in range (0,7,1):
    colour_in = input()
    if colour_in == 'Red':
        count_red +=1
    elif colour_in == 'Blue':
        count_blue +=1
    elif colour_in == 'Purple':
        count_purple +=1
    else:
        count_pink +=1

if count_red != 2:
    print('Red')
elif count_blue != 2:
    print('Blue')
elif count_purple != 2:
    print('Purple')
else:
    print('Pink')
```

Or a one-line solution for fun.

```python
[[print(s)if 2-x.count(s) else 0 for s in x]for x in[[input()for _ in
↪  range(7)]]]
```

C++ Solution

```cpp
#include <iostream>
#include <string>
#include <map>
using namespace std;

typedef map<string, int> msi;

int main () {
    msi names = {{"Red",0},{"Blue",0},{"Purple",0},{"Pink",0}};
    for (int i = 0; i < 7; i++) {
        string colour;
        cin >> colour;
        names[colour] += 1;
    }
    for (const auto& [k, v] : names) {
        if (v == 1) {
            cout << k << endl;
            return 0;
        }
    }
    return 1;  // error
}
```

This solution uses structured bindings which are only available since C++17.

Mind Reader

https://train.nzoi.org.nz/problems/1218

First, let's try to figure out how many cards are larger than a certain card within the same suit. One way of doing this is the put all the card numbers, in ascending order, into a list. Then, we can use a function to find the *index* of any card number in the list (such as `index()` in Python or `find()` in C++). This index will be equal to the number of cards *smaller* than that number within the same suit. We know there are 13 cards per suit, so the number of cards *greater* is $13 - index - 1 = 12 - index$.
We can then apply the same approach for the suits. Each suit contains 13 cards, for each suit greater than our card's suit, there are an additional 13 greater cards. Thus, the total number of cards greater is: 13 * *suits greater* + *numbers greater*.

Python Solution

```python
cards=['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K','A']
suit=["spades", "clubs", "diamonds", "hearts"]
user_card=input()
user_suit=input()

my_i=cards.index(user_card)
my_s=suit.index(user_suit)

above_card=12-my_i
above_suit=3-my_s
total=above_card+13*above_suit
print(total)
```

Python 1-Line Solution

```python
print(12-"2 3 4 5 6 7 8 9 10 J Q K A".split().index(input())+(3-"spades
↪   clubs diamonds hearts".split().index(input()))*13)
```

C++ Solution

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
int main () {
    vector<string> n =
        {"2","3","4","5","6","7","8","9","10","J","Q","K","A"};
    vector<string> s = {"spades", "clubs", "diamonds", "hearts"};
    vector<string> sam(2);
    cin >> sam[0] >> sam[1];
    int p1 = search(n.begin(),n.end(),sam.end()-1,sam.end())-n.begin();
    int p2 = search(s.begin(),s.end(),sam.end()-1,sam.end())-s.begin();
    cout << (12 - p1) + (3 - p2)*13 << endl;
}
```

Counting 3s

https://train.nzoi.org.nz/problems/1207

Subtask 1

It holds that $1 \leq N \leq 20$. In this range, there are only two numbers that each contain one 3, these are 3 and 13. Therefore, the answer is 0 for $1 \leq N < 3$, 1 for $3 \leq N < 13$, and 2 for $13 \leq N \leq 20$. We can do this with conditional statements.

Subtask 1 Python Solution

```python
N = int(input())
if N < 3:
    print(0)
elif N < 13:
    print(1)
else:
    print(2)
```

Subtask 1 C++ Solution

```cpp
#include <iostream>
using namespace std;

int main(){
    int N;
    cin >> N;
    if (N < 3) cout << "0\n";
    else if (N < 13) cout << "1\n";
    else cout << "2\n";
}
```

Subtask 2

It holds that $1 \leq N \leq 20\,000$. Since N is still small enough, we can methodically count the number of threes in each integer for all integers up to N.

With this solution, we must loop N times. Each time, we must check each digit in the number to see if it is a 3. The number of digits in some number k is $\log_{10} k$. Hence, we can calculate how many total iterations T we need. For example, there are about 10 numbers with one digit, 100 numbers with two digits, etc. This gives

$$T = \sum_{d=1}^{log_{10}N} \left(d \times 10^d\right) + Nlog_{10}N.$$

So, for $N = 20\,000$ we get

$$T = 1 \times 10^1 + 2 \times 10^2 + 3 \times 10^3 + 2 \times 4 \times 10^4 = 83\,210.$$

A computer can easily do this in under a second. In general, the complexity of this solution is $O(N \log N)$ since the $Nlog_{10}N$ factor dominates for large N.

Subtask 2 Python Solution

```python
N = int(input())
total = 0
for i in range(1, N+1):
    for digit in str(i):
        if digit == "3":
            total += 1
print(total)
```

Subtask 2 C++ Solution

```cpp
#include<iostream>
using namespace std;

int main ()
{
    long N; cin >> N;
    long total = 0;
    for (long i = 1; i <= N; i++) {
        long num = i;
        while (num) {
            if (num % 10 == 3) total += 1;
            num /= 10;
        }
    }
    cout << total << endl;
}
```

Subtask 3 and 4

Here, N can be up to 2 000 000 000 000 000 (two quadrillion). Our Subtask 2 solution would take more than one million years to compute this on the average computer processor, putting it slightly over the one second limit for this problem. We need a new approach. Time to call in our friend mathematics!

First off, you may notice that the answer when N is a multiple of 10 seems to follow a pattern. For $N = 10$, we get 1 three. For $N = 100$, we get 20 threes. For $N = 1000$, we get 300 threes. This pattern continues as we multiply N by 10 with the sequence of answers being $4\,000, 50\,000, 600\,000$, and so on. Specifically, the answer for powers of ten is given by $z \times 10^{z-1}$ where z is the number of zeros in N. At first glance, this pattern might not seem very helpful for values of N that are not multiples of 10, but we can work around this by breaking up N into numbers that are simple multiples of multiples of 10. For example, if $N = 1023$ then the total number of threes up to 1023 is the same as the sum of the total number of threes up to each of 1000, 20, and 3.

With this technique, the only thing we have left to work out is how many extra threes we need to add for numbers with a leading digit that is higher than 1. For example, there are double the number of threes for $N = 20$ than there are for $N = 10$. The same rule holds for numbers with more zeros. For example, 1000 has half the number of threes that 2000 does. This is because for every number of the form $0xxx$ (where x is any digit) there is a corresponding number of the form $1xxx$. Hence, the sum of threes in all numbers fitting the form $0xxx$ is equal to the sum of threes in all numbers fitting $1xxx$. The rule is then to calculate the number of threes in the highest power of 10 that is less than or equal to our number and multiply this by the leading digit of our number. For example, the number of threes up to $N = 1000$ is 300, so the number of threes up to $N = 2000$ is $300 \times 2 = 600$ (where 2 is the leading digit).

For cases where the leading digit is less than 3, we have the full solution. However, if the leading digit is 3. For example, if $N = 322$, then we need to calculate the number of threes we could count if $N = 300$, $N = 20$, and $N = 2$, but then also add an extra 22 threes to our answer which are contributed for all numbers from 301 up to 322 (due to the leading 3). We also need to add 1 to the total to account for the three in 300. For numbers with a leading digit greater than 3, the number of extra threes contributed by the leading 3 is a power of 10. Specifically, we need to account for all the numbers which are of the form $3xxx$.. where x is any lower place value digit. Hence, we add to the total the highest power of 10 that is less than or equal to our simpler number. We could equivalently account for this contribution by increasing the leading digit by one for numbers with a leading digit greater than 3.

In summary

Find the number of digits 3 in all numbers up to and including N.

1. Break the number N into the sum of simpler numbers that only contain a leading digit, with all other digits being zero. For example, $N = 1023$ becomes the sum of the counts for $N = 1000$, $N = 20$, and $N = 3$.
2. For each of these simpler numbers
   (a) Calculate the number of 3s up to the highest power of 10 that is less than or equal to N. For example, the highest power of 10 for $N = 80\,000$ is $10\,000$. The number of 3s up to $10\,000$ is then easily calculated to be $4\,000$ by following the pattern we observed earlier. Let's call this number S_j, where j is the index of the highest place value. This gives us the equation $S_j = j \times 10^{j-1}$. In our example, $j = 4$ since there are 5 digits in the number and we index starting from 0. Hence, $S_4 = 4 \times 10^{4-1} = 4000$.

   (b) Multiply S_j by the digit of highest place value. From the previous example of $N = 80\,000$ we would multiply by 8, giving $32\,000$.

   (c) If the digit of highest place value is equal to three, add the number represented by all digits of lower place value index than j and add one to this number.

   (d) If the digit of highest place value is greater than three, add 10^j threes. Hence, in our previous example, we add 10^4 to $32\,000$. This gives a final total of $42\,000$ digits 3 for numbers up to $N = 80\,000$.

With this approach, our solution requires a main loop which iterates on the number of digits in N instead of having to iterate more than $N \times \log_{10} N$ times as we did in the Subtask 2 solution. This means that instead of far more than 2 quadrillion iterations in the worst case, only 15 iterations are required. The number of digits in N is $\log_{10} N$. Hence, the complexity of this solution is $O(\log N)$. This is much more manageable for large N than the $O(N \log N)$ solution from Subtask 2.

Full Python Solution

```python
N = input()
ans = 0
# Start where i=0 indexes the digit with highest place value
for i in range(len(N)):
    d = int(N[i])  # Digit of highest place value
    j = len(N) - i - 1  # Place value index (inverse of i)
    sj = j * (10**j)//10  # Num threes up to highest pow
    ans += d * sj
    if d == 3:
        # Add number read from lower place value digits
        # Add one to account for the '300...' case.
        # Concat "0" so that an empty sub string does not error
        ans += int("0" + N[i+1:]) + 1
    elif d > 3:
        # Add extra threes according to place value index
        ans += 10**j

print(ans)
```

Or, a one-line solution for fun. Note that this solution takes a slightly different approach to the calculation but it does get the same result.

```python
print([sum([[(p+1 if x>3 else p)*10**(len(N)-i-1)+(int("0"+N[i+1:])+1 if
↪   x==3 else 0)for x,p in[map(int,[N[i],"0"+N[:i]])]][0]for i in
↪   range(len(N))])for N in[input()]][0])
```

Full C++ Solution

```cpp
#include <iostream>
#include <string>
#include <cmath>
using namespace std;

int main()
{
    string N; cin >> N;
    long long value, j, total = 0, len = N.length();
    for (int i = 0; i < len; i++) {
        value = N[i] - '0';
        j = len - i - 1;
        if (j > 0) total += value * j * pow(10, j-1);
        if (value == 3) {
            total += 1;
            if (i + 1 < len) total += stoll(N.substr(i + 1));
        } else if (value > 3) {
            total += pow(10, j);
        }
    }
    cout << total << endl;
}
```

Shocking calculation

https://train.nzoi.org.nz/problems/1195

This problem asks you to use addition and multiplication at most once to add up to a target. This is complicated by the fact that there is a different cost to using each digit or operation.

Subtask 1

The first thing we need to do is to figure out for a given number how much shock it would cost. You would first break up that number into its corresponding digits and add up the shock cost of its digits. Let's call $Shock(x)$ = shock value of typing in the number x.
For Subtask 1, all shock values are 1, so we only need to minimise the number of button presses. Using a multiplication or addition will always result in more button presses - can you see why? This means it's always optimal to simply enter the number and press the '=' button. The answer is just the number of digits in the target value plus 1 for the '=' button.

Python Subtask 1 Solution

```python
N = int(input())
shocks = 0
while N > 0:
    N //= 10
    shocks += 1
print(shocks + 1)
```

And a simple 1-liner:

```python
print(len([input(), input(), input()][0])+1)
```

C++ Subtask 1 Solution

```
#include <iostream>
#include <string>
#include <cmath>

using namespace std;

int N;
int shocks = 0;

int main()
{
    cin >> N;

    for (; N > 0; N/=10) {
        shocks++;
    }

    cout << shocks+1;
}
```

Subtask 2

One way we can obtain the lowest shock to reach a target is to try all possible combination of values which multiply and add to the target.
We can loop through every element A in the range $[0, target]$, where we will sum A and $(target - A)$ to become the target. This will try out all of the possible additions we can do to make up the target.
Additionally, A could be made up of a multiplication for two numbers. We can try to multiply two numbers to A - lets loop through every element B in $[0, A]$. If A is divisible by B, we can try multiplying $B * (A/B) = A$. If the best shock value from this method is better than the shock value of A, then we use that value instead of A.
Using these two operations, we would acquire the optimal shock value of the target. We have to be careful to also run the multiplication method by itself on $target$, as the best solution could be to do a single multiplication without addition. The running time of this algorithm is $\mathcal{O}(N^2)$.

Python Subtask 2 Solution

```python
N = int(input())
V = list(map(int,input().split()))
O = list(map(int,input().split()))

def cost(n):
    return sum(map(lambda x: V[int(x)], str(n)))

best = cost(N) + O[2]

for x in range(1, N + 1):
    cur = cost(x)
    y = 1
    while y*y <= x:
        if x % y == 0:
            cur = min(cur, cost(y) + cost(x // y) + O[1]) # Multiply y
            ↪  and (x/y)
        y += 1

    if x < N:
        # Add x and (N-x). (N-x) might itself be made up of a
        ↪  multiplication, as calculated previously
        cur += cost(N - x) + O[0]

    best = min(best, cur + O[2])

print(best)
```

Subtask 3 (Full solution)

An additional observation that is used to optimise the solution is that when looping through B, it is unnecessary to loop through all of $[0, A]$. Simply looping from 0 to $\sqrt{A}$ is sufficient as this will contain all pairs of numbers that multiply to A. This optimization brings our running time down to $\mathcal{O}(N * \sqrt{N})$, which is sufficient for the full solution.
But we can do better! Instead of trying to find numbers that multiply to A, we can instead try to use A to multiply to *target*. We can try both multiplying A with a number to get *target*, and multiplying A with a number and then adding another number. For the latter choice, we can iterate through all numbers B such that A * B ¡ *target*. For each of those iterations, we calculate the cost of doing $A * B + (target - A * B)$.

Can you see how this approach is faster? *Hint:* $1+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{n} \in \mathcal{O}(log(n))$

Python Full Solution

```
N = int(input())
V = list(map(int,input().split()))
O = list(map(int,input().split()))

def cost(n):
    return sum(map(lambda x: V[int(x)], str(n)))

best = cost(N) + O[2]

for x in range(1, N + 1):
    cur = cost(x)
    y = 1
    while y*y <= x:
        if x % y == 0:
            cur = min(cur, cost(y) + cost(x // y) + O[1]) # Multiply y
            ↪  and (x/y)
        y += 1

    if x < N:
        # Add x and (N-x). (N-x) might itself be made up of a
        ↪  multiplication, as calculated previously
        cur += cost(N - x) + O[0]

    best = min(best, cur + O[2])

print(best)
```

C++ Full Solution

```
#include <cstdio>
#include <map>
#include <cstring>

using namespace std;

int shock[22];
map<char, int> ctoi;

int calcshock(int val) {
        char cval[8];
        int ret = 0;
        sprintf(cval, "%d", val);
        for (int i=0; i<8; i++) {
                if (cval[i] == '\0') break;
                ret += shock[ctoi[cval[i]]];
        }
        return ret;
}

int main() {

        int n;
        scanf("%d", &n);

        for (int i=0; i<10; i++) {
                ctoi['0'+i] = i;
                scanf("%d", shock+i);
        }

        ctoi['+'] = 10;
        scanf("%d", shock + ctoi['+']);
        ctoi['*'] = 11;
        scanf("%d", shock + ctoi['*']);
        ctoi['='] = 12;
        scanf("%d", shock + ctoi['=']);

        int volts = 123456789;
        for (int i=1; i<=n; i++) {
                // init ishock to be shock of typing it in directly
                int ishock = calcshock(i);
                for (int j=1; j*j<=i; j++) {
                        if (i%j == 0) {
                                // try to test if it's better to press
                                ↪  buttons to mulitply to i
                                ishock = min(ishock, calcshock(j) +
                                ↪  calcshock(i/j) + shock[ctoi['*']]);
                        }
                }
                // add shock of typing n-i if n != i
                int cmpshock = ishock + (i!=n ? calcshock(n-i) +
                ↪  shock[ctoi['+']] : 0);
                if (cmpshock + shock[ctoi['=']] <= volts) {
                        volts = min(volts, cmpshock + shock[ctoi['=']]);
                }
                volts = min(volts, cmpshock + shock[ctoi['=']]);

        }

        volts = min(volts, calcshock(n)+shock[ctoi['=']]);
        printf("%d\n", volts);
        return 0;
}
```

Python Faster Solution

```python
price = int(input())
volts = list(map(int,input().split()))
ops = list(map(int,input().split()))
cache = [-1]*100001

def cost(n):
    if cache[n] == -1:
        cache[n] = sum(map(lambda x: volts[int(x)], str(n)))
    return cache[n]

best = cost(price) + ops[2]  # just type the price

for x in range(1, price):
    best = min(best, cost(x) + cost(price - x) + ops[0] + ops[2])  # single addition
    if price % x == 0:
        best = min(best, cost(x) + cost(price // x) + ops[1] + ops[2])  # single multiplication

    for y in range(1, price // x + 1):
        best = min(best, cost(x) + cost(y) + cost(price - x * y) + sum(ops))  # multiply and add

print(best)
```

Hot Chocolate

https://train.nzoi.org.nz/problems/1195

Subtask 1

In this subtask, there is no capacity in the right hot chocolate machine. This means that every student must get their drinks from the left hot chocolate machine. Therefore, there are two cases in which it is impossible for the students to all be satisfied:

- There is not enough capacity in the left machine for everyone to drink
- One or more students have a *strict preference* for the right machine (which they cannot drink out of, as it has no capacity)

If none of those conditions are true, then everyone can happily get their fill from the left machine, and so the total happiness is the sum of everyone's happiness values for the left machine.

Python Subtask 1 Solution

```python
l, r = map(int, input().split())
n = int(input())
total = 0
for i in range(n):
    ci, li, ri = map(int, input().split())
    if li == -1:
        print("Camp is cancelled")
        break
    total += li
    l -= ci
    if l < 0:
        print("Camp is cancelled")
        break
else:
    print(total)
```

Subtask 2

In this subtask, every student has a strong preference. That means that we must assign every student to drink from the machine that they prefer. Similarly to the previous subtask, there are two cases where we cannot satisfy everyone:

- There is not enough capacity in the left machine for every left-preferring student to drink
- There is not enough capacity in the right machine for every right-preferring student to drink

Thus, similarly to the last subtask, we iterate through every student and make them 'drink' from their preferred machine. We stop if a machine cannot supply the current student, or all students have been satisfied.

Python Subtask 2 Solution

```python
l, r = map(int, input().split())
n = int(input())
total = 0
for i in range(n):
    ci, li, ri = map(int, input().split())
    if li == -1:
        total += ri
        r -= ci
    elif ri == -1:
        total += li
        l -= ci
    if l < 0 or r < 0:
        print("Camp is cancelled")
        break
else:
    print(total)
```

Subtask 3

For the general case, we will adopt a *Dynamic Programming* approach. Dynamic Programming, or DP, isn't straight forward to explain, but for your interest you may want to read Chapter 7 of the Competitive Programming Handbook. We will assume you have a reasonable understanding of DP for the rest of the solution to this problem.
You may have noticed that this problem is in many ways similar to the classic Knapsack Problem. You can think of the hot chocolate machines as the knapsacks, the students as the items. For each student, the number of cups they will drink is the item weight and the happiness values are the... well, values. The only differences are:

- There are two knapsacks/coffee machines (and each student gains different values from each)
- **Every student must be given a drink.** In contrast, in the classic knapsack problem, you can choose to take or leave items.

The latter difference will be important in Subtask 4.

Instead of the classic knapsack, where our DP state is `DP[ITEM][CAPACITY]`, we can have a DP state that is `DP[ITEM][LEFT_CAPACITY][RIGHT_CAPACITY]`. For each state, our recurrence relation consists of the optimum of two choices - either we make the student drink from the left or the right machine. We will also have to be careful to check for students with strict preferences (in which case only one of the options could be valid), and that students cannot drink from machines with insufficient remaining capacity.

Python Subtask 3 Solution

```python
L, R = map(int, input().split())
N = int(input())
dp = [[[None for i in range(R+1)] for j in range(L+1)] for k in
↪   range(N)]
students = []
for x in range(N):
    students.append(list(map(int, input().split())))

def solve(idx, left, right):
    if idx == N:
        return 0
    if dp[idx][left][right] is not None:
        return dp[idx][left][right]

    dp[idx][left][right] = -999999999

    if students[idx][1] > 0 and left >= students[idx][0]:
        dp[idx][left][right] = max(dp[idx][left][right], solve(idx + 1,
        ↪  left - students[idx][0], right) + students[idx][1])
    if students[idx][2] > 0 and right >= students[idx][0]:
        dp[idx][left][right] = max(dp[idx][left][right], solve(idx + 1,
        ↪  left, right - students[idx][0]) + students[idx][2])

    return dp[idx][left][right]

print(solve(0, L, R) if solve(0, L, R) > -1 else "Camp is cancelled")
```

Subtask 4 (Full Solution)

Our Subtask 3 solution is on the right track, but our DP formulation just requires too many states, which will exceed both the time and memory limit. But are all of those states really useful? Maybe we could get rid of some of them while still maintaining our general algorithm.
Take, for example, this test case.

```
4 6
2 5 10
3 6 20
```

For that test case, one of our DP states would be `DP[1][0][6]`
This means that we are considering student number 1 (so student number 0 has already been satisfied), and there is 0 and 6 capacity remaining in the left and right machines respectively. But is this state actually useful? For there to be 0 capacity remaining in the left machine, student 0 must have taken 4 cups of hot chocolate. But student 0 only requires 2 cups of chocolate! Therefore, this state is really identical to the state `DP[1][2][6]`. In fact, the only valid DP states for considering student 1's allocation are `DP[1][2][6]` and `DP[1][4][4]`. Those are the cases where student 0 has drunk from the left and right machines respectively.
This leads us to an important observation. If we know the current number of students who have already drank, N, and the remaining capacity in the left machine, L, then the remaining capacity in the right machine is the sum of all N students' required cups minus L. That is, the amount of cups drank from the right machine is equal to the amount of cups students *didn't* drink from the left machine.

This means we can completely discard one dimension from our DP state - we don't need the capacity of the right machine as it is dependent on our other DP states anyway! Our DP state is simply `dp[ITEM][LEFT_CAPACITY]`. To 'drink' from the left machine, we recurse into `dp[i-1][lc - c]` and to 'drink' from the right machine, we recurse into `dp[i-1][lc]`, since the left machine capacity won't be changed if we drink from the right machine. We also calculate the right machine capacity using the method discussed previously in order to make sure there is enough capacity left in that machine.

Python Full Solution

```python
L,R = map(int, input().split())
N = int(input())
students = []
cups = [0]
dp = [[None]*(L+1) for x in range(N+1)]
for x in range(N):
    students.append(list(map(int, input().split())))
    cups.append(cups[-1] + students[-1][0])

def solve(idx, left):
    if idx == N: return 0
    if dp[idx][left] is not None: return dp[idx][left]
    right = L + R - left - cups[idx]
    dp[idx][left] = -99999999999
    if students[idx][1] > 0 and left >= students[idx][0]:
        dp[idx][left] = max(dp[idx][left], solve(idx + 1, left -
         ↪  students[idx][0]) + students[idx][1])
    if students[idx][2] > 0 and right >= students[idx][0]:
        dp[idx][left] = max(dp[idx][left], solve(idx + 1, left) +
         ↪  students[idx][2])
    return dp[idx][left]

print("Camp is cancelled" if solve(0, L) < 0 else solve(0, L))
```

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 $O(N^2)$ 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 $O\left(1\right)$ 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 couple of extra operations every time

N doubles in size.[1] 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 $O(N^2)$. 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, $O(N^2)$ 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 $O(N^2)$ if both these loops run about N times.

[1]More formally, it means there exists some constant c for which the program takes at most c extra operations every time N doubles in size.