Module 1 – Getting Started

The aim of this module is not to start with the techniques involved in Problem Solving but to provide an overall view of 'what to expect' and 'what not to' from this subject.

Some of you might find it difficult to digest all the terms mentioned in this module, but I encourage you to read these chapters once and get yourself familiarized with the jargon that you will see again in upcoming chapters with great detail. For beginners, this module (or perhaps many other concepts in this book) is not a one-time-read material. It is quite possible that you may require multiple reads to understand something. It is also possible that you may not get the concepts cleared on some day, but as you progress in the subjects, maybe after few days or weeks, you will eventually understand them. So, don't get disheartened if the first read didn't make much sense.

You will find things difficult; you might have to search some concepts online, you might have to read things over and over again, but eventually you will get what you wanted. That's how – Learning is done!

And if ever a thought of quitting comes to your mind, remember 'The Newton's First Law of Motion'. It simply states – a body continues to be in a state of rest or motion, due to inertia, unless an external imbalanced force is applied to it. So, to set something in motion, we need some initial effort. But once you have forced yourself harder and you get started in some field, you just keep getting better and better without stopping!

All the best! Wish you a wonderful journey!

1. INTRODUCTION TO PROBLEM SOLVING

Right from the title of this book to this page, you must have seen the words 'Problem Solving' a lot many times. So, you know that we are interested in solving problems. *But what kind of problems?* We aren't interested in some political or environmental problems, because we are interested in the problems which are solvable! Just kidding! Our main interest is in understanding – *how to solve algorithmic problems.* We will discuss a lot about algorithms throughout this book. For now, you can consider them as a class of problems which follow a fixed set of instructions to produce some desired output based on the input given to it.

There is one term – *Competitive Programming,* which is synonymous to Problem Solving with only difference being, that there are time constraints in the former. Most of the times I would be using the terms 'Problem Solving' and 'Competitive Programming' almost interchangeably. But more or less, the core directive of Competitive Programming (CP) could be stated as: *"Given some solved algorithmic problems, solve them as efficiently and quickly as possible!"* Let's try to understand each of the terms one by one.

- The term *'solved algorithmic problem'* implies that in CP, we are dealing with solved problems and not the research-based ones where the solutions are still unknown. Some people (at least the problem author) have definitely solved these problems before.
- To *'solve them'* implies that we must push our knowledge to a certain required level so that we can produce a working code that can solve these problems too. This means we have to write a code that produces the required output on the hidden input/test data within the stipulated time limit.
- In CP, obvious brute-force algorithms don't work most of the time. Hence, we aim for algorithms that are quite *'efficient'* and can produce the desired output under a certain time limit, like 1 second or 5 seconds.
- The need to solve the problem *'as quickly as possible'* is where the competitive element lies — speed is a very natural goal in human behaviour.

But the biggest question which you must ask is – *Why should we learn the art of problem solving? Or rather, why should we even try CP?*

Probably I won't be able to do justice to this question by listing down a few advantages of CP as an answer. Hence, I have provided a lot many stories which you will encounter again and again, throughout this book. All these stories are real which have been collected from various authorized resources. The purpose of telling you these stories is to make you realize that the algorithm design and analysis is not just theory, but a tool that must be used at times to solve some real-world problems.

But for time being, I am giving you some possible advantages of CP from the perspective of software engineers.

Advantages of CP:

- Programmers who are well-versed & habitual in solving CP problems can be considered the ones with nice mental-ability and it is assumed that they use their skills and write efficient code, even when they get to write big pieces of software.
- By hiding the actual test data from the problem statement, CP encourages the problem solvers to exercise their mental strength to think of all possible corner cases of the problem and test their programs with those cases. This is typical in real life where software engineers have to test their software a lot to make sure that the software meets the requirements set by clients.
- Some programming competitions are done in a team setting to encourage teamwork as software engineers usually do not work alone in real life.
- It is fun to do. One learns a lot about programming while practicing tough problems.

With that said, there are high chances that people who are good at CP can turn into good software programmers as well. However, this in general is not true. Many people have created big software without any knowledge of Data Structures or Algorithms initially. However, such cases are very rare and it's better to learn CP because of its demand and knowledge it provides. Almost every MNC demands CP tests as a prerequisite for software jobs these days. In my opinion, it's better to utilize your time by learning CP, just for fun, rather than counting over its advantages.

1.1 Know where you stand

Before you jump into CP, I would suggest you to assess yourself and find where you stand now. For that, following is one such question which is similar to the questions that are usually asked in CP contests. Read the problem and try to visualize the solution that the problem is expecting us to find.

Problem Statement: Let (x, y) be the coordinates of a student's house on a 2D plane. There are 2N students and we want to pair them into N groups. Let d_i be the distance between the houses of 2 students who are in i^{th} group. We have to form N groups (each containing 2 students), such that,

$$cost = \sum_{i=1}^{N} d_i$$

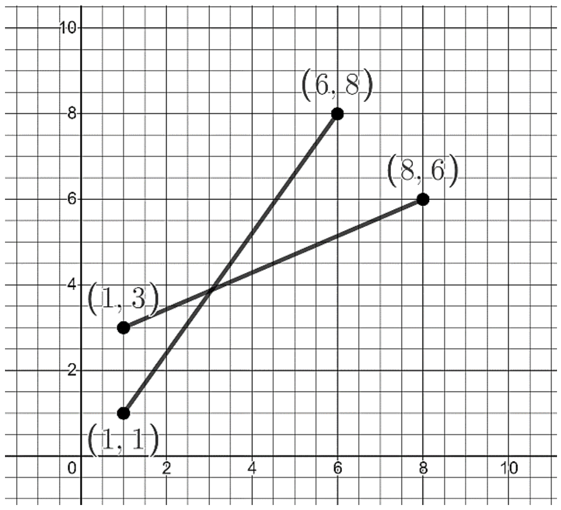
is minimized. Print the minimum cost, rounded to 2 decimal places.

Constraints:
$1 \le N \le 8$; $0 \le x, y \le 1000$; Time limit: 1 second

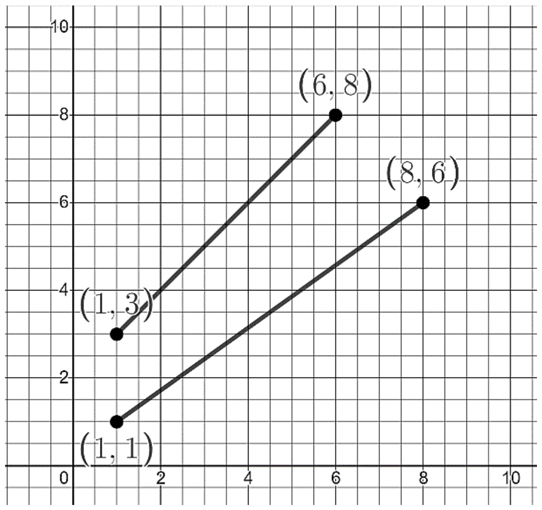
Sample Input:
N = 2 and the coordinates of the 2N = 4 houses are {1, 1}, {8, 6}, {6, 8}, and {1, 3}.

Sample Output:
Cost = 4.83

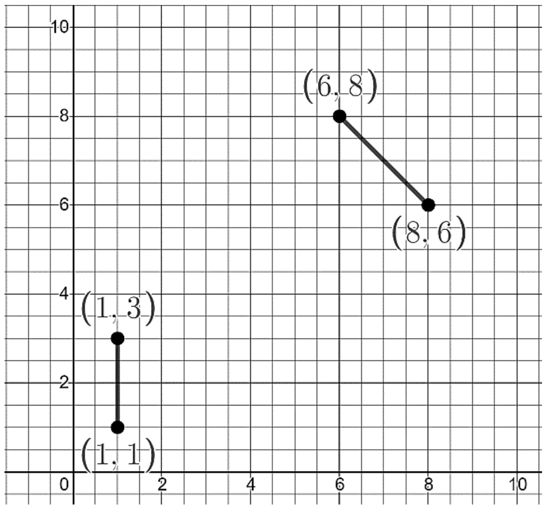
Explanation:
There are 3 ways in which we can form 2 groups of these 4 students. In each way, the cost is the sum of the lengths of the segments connecting 2 points (i.e. forming a group of 2).



In this case (i),
Cost = 8.60 + 7.61 = 16.21



In this case (ii),
Cost = 7.07 + 8.60 = 15.67



In this case (iii),
Cost = 2.00 + 2.83 = 4.83, which is the most optimal result.

Can you solve this problem for the given constraints? If yes, how much time would you likely require to write a working code? Think and try not to flip this image immediately!

Now ask yourself: Which of the following best describes you? As said, it is okay if you are unclear about some terminology now. You will eventually learn all the stuff anyway.

- *Level 1: One who is unclear*
  You read the problem and got confused. You tried to code something, but all your attempts failed to find the general solution. If you thought of a greedy logic –repeatedly pair the two remaining students with the shortest separating distances, then your logic is wrong. (*We will see – 'Why?' later in this book.*) Finally, you thought to do naïve complete search (brute-force) by trying all possible pairings of points, but you realize that your code is exceeding the time-limit of 1 second.

- *Level 2: One who gives up*
  You read the problem and realize that you have seen such problem before. Even if not, you are able to decipher that this is a question which could be solved by using *Dynamic Programming.* You know, that you haven't learnt how to solve such DP questions; hence you skip the problem and reads another one.

- *Level 3: One who is slow*
  You read the problem and realize this is a hard problem *(by 'hard', we meant NP hard).* The problem is famously known as – "minimum weight perfect matching on a small general weighted graph." However, since the input is small, the problem could be solved using DP with bitmasking. The DP state is a bitmask that describes a matching status. Matching two unmatched students, say *i* and *j,* will turn on two bits *i* and *j* in the bitmask. Once you realize this, you start writing down the top-down DP logic. You try your code on test cases, debug it more and keep on doing this until your code runs perfectly fine. After 3 to 4 hours, you get a perfect working code for this problem.

- *Level 4: One who is fast*
  You did all the steps mentioned for Level 3 programmer within 30-45 minutes.

- *Level 5: One who is elite!*
  You did all the steps mentioned for Level 3 programmer within 15-20 minutes. And this is because, you know this famous problem and the logic already. You didn't design the algorithm and just coded it very fast.

By the end of this book, you can expect to reach at least Level 3 and after a little bit of practice, Level 4 is not that far. Even reaching Level 4 is considered as a great accomplishment for those who are focusing to shift towards developing softwares or pursue Machine Learning & Data Science later.

Note: Being well-versed in CP is not the end goal, but only a means to an end. The true end goal is to produce intelligent problem-solving minds which are much readier to produce better software and to face harder research problems in the future.

1.2 Tips to Get Started

We start this book by giving you several general tips.

a) *Type Faster*

Although this tip may not be much relevant to problem solving skills, but people have benefitted themselves with this a lot. For the students who are trying to achieve better ranks in CP contests, should know that the *ith* and *ith+1* rank differs only by a minute or so, which is just the matter of typing speed. Not only about the ranks, but often there are situations when we couldn't find the efficient code and we have to resort to brute-force approach. And many people fail to do so because they couldn't write the code at the last minute.
Hence, typing speed of about 50-60 words per minute is desired.

b) *Quickly Identify Problem Types*

In the initial stages, you may classify the problems to the strategy which is used to solve them. For example, a problem can be classified as a DP problem, or a Greedy problem, or some geometric problem. However, such classification is good only in initial stages. This is because some techniques like sorting and searching are not considered as separate problem-solving techniques as most of the times they are used as sub-routines to solve some bigger problem. There is no separate class of problems for 'Data Structures'. It is assumed that all data structures are known as a pre-requisite. A problem can belong to multiple techniques. For e.g. *Floyd Warshall algorithm* is both – a DP and a graph algorithm. Similarly, *Kruskal's Algorithm* is both – a greedy and a graph algorithm. Harder problems require multiple techniques to get solved. So, the main goal is not to associate/label problems with the techniques required to solve them.

Once you feel that you have got better in problem solving, you should start classifying problems into following 3 types:

- *Problems of this type I have solved before. Hence, I am confident in re-solving this problem.*

- *Problems that I have seen before, but I couldn't solve it at that time. Maybe I will have to think harder now, or refer some of my previously learnt theory to solve it.*

- *I haven't seen any problem like this before. I will have to look at hints or solutions for solving it. I should try similar problems now to test whether I learnt and could apply the concept on similar questions or not.*

As you turn into a veteran problem solver, you should be able to classify most of the questions and their types into first two classes.

c) *Do Algorithmic Analysis*

Entire *Chapter 2 – Algorithmic Analysis*, is focused on this tip. Refer that chapter for more details.

d) *Master Programming Languages*

It doesn't matter which programming languages you choose. There are online platforms which accept solutions to the problems, in a form of some code, for more than 30 programming languages! Codechef being one such example. So, the only condition is – whichever programming language you pick, you should be master of that.

Personally, I am a fan of C/C++ and Python. These are the two languages in which I will be providing you all the codes in this book. C/C++ is a lot faster and almost 75% of programmers on CP platforms use these languages. Python, despite being slow, is very famous due to the ease and clarity of thoughts it provides, even when the algorithm is written in the form of code.

In this book, I would assume that you know the programming languages, at least at an intermediate level. You are familiar with STL, templates, dynamic memory allocation, pointers, etc. in C++11 or any higher standard release of C++. Coding in Python is easy, but coding efficiently in Python is a lot tough. You need to know a lot about iterators, generators, comprehensions, lambdas, references, function calls, etc. if you really wanna code something better and faster in Python.

e) *Have your own CP set-up ready*

Every CP programmer has a set-up of his own. This set-up includes proper selection of IDE, some shorthand terminal settings, template starter file, command line utilities, etc. Some programmers have code snippets of the most frequently used algorithms ready; so that they could just copy-paste it and edit it a little whenever required. In short, whichever tools enhance your performance, come under your CP set-up.

However, my suggestion would be, not to rely heavily on the setup. This is because the contest platforms vary a lot and many times you won't even have the auto-completion feature or sometimes even syntax highlighting would be missing. In ideal situation, even if you are asked to write a code on notepad/gedit, you should be able to do it. Most companies use Google Docs while interviewing online or provide a white-board to write the code if the interview is on-site. The only reason of using setup is – writing code on notepad/gedit is a lot time consuming and we wish to invest our time in learning things faster, rather than wasting time just on typing.

I have shared my CP set-up in *Appendix A.*

f) *Do some team-work*

This is only valid if you are planning to participate in some team-based competition. In such cases, you should already have enough practice of coding on blank paper and designing hard corner cases for questions.

Coding on blank paper is useful when your teammate is using the computer. So that, when it is your time to use the computer, you should be able to type the code as soon as possible, rather than wasting time thinking in front of the computer.

If you are done with your work, design hard corner cases for your teammate who is writing logic on the computer. Hope his/her code passes these cases too!

1.3 Anatomy of a Problem

You can skip this section if you have solved some questions already on competitive programming platforms like CodeChef, Codeforces, HackerEarth, HackerRank, SPOJ, etc.

A problem is usually described using following components:

a) *Problem Description:*

Usually, the easier problems are written to deceive contestants and made to appear difficult, for example by adding '*extra information'* to create a diversion. Contestants should be able to filter out these unimportant details and focus on the essential ones. However, harder problems are usually written as succinctly as possible—they are already difficult enough without additional embellishment.

So, if you find a problem with a lot of unnecessary story or talk in it, there are very high chances that the problem is very easy and demands a very simple logic/insight from you. At these situations, you should not think a lot deep and rather hunt for the problem's weakness.

b) *Input and Output Description:*

In this section, you will be given details on how the input is formatted and on how you should format your output. This part is usually written in a formal manner.

c) *Constraints:*

A good problem should have clear input constraints as the same problem might be solvable with different algorithms for different input constraints.

d) *Sample Input and Output:*

Problem authors usually only provide trivial test cases to contestants. The sample input/output is intended for contestants to check their basic understanding of the problem and to verify if their code can parse the given input using the given input format and produce the correct output using the given output format. Do not submit your code to the judge if it does not even pass the given sample input/output.

e) *Explanation (optional):*

This section includes the optional explanation of how the sample output is obtained from the sample input. In some cases, the problem authors may drop hints or add footnotes to further describe the problem.

1.4 Warm Up Problems

There is no better way to begin the journey in problem solving than to solve a few problems. In this section, we would be solving particularly 'super-easy' and 'easy' problems from CodeChef. We will be using Codechef as our prime programming platform for practicing problems. *(That was quite a lot of alliteration!)* To solve these problems, you need to have the basic knowledge of looping, conditionals and simple-math implementations in C++ or Python.

You are encouraged to practice on this platform by 'Submitting' the code yourself, as you go through this book. Here is how we will mention our problems:

➢ **Problem No. TAG – Problem Name**

To access the same problem on the platform, you have to search for the problem using the link, as mentioned in the following format:

https://www.codechef.com/problems/<TAG>

E.g. A problem with TAG as ADASCOOL will be available at the link, https://www.codechef.com/problems/ADASCOOL

If you ever get stuck and you need to look at the solution for some problem, people of Codechef community write detailed explanations for you, in the form of *EDITORIALS*. On the problem page, most of the times – you will find an *editorial link* which will take you to the solution.

Note: In this book, I will just provide the synopsis of the problem description, some noteworthy points in the explanation and my code for that problem. You mandatorily have to visit the problem's webpage for reading the entire problem in detail – as described in 'Anatomy of a Problem' section. PLEASE, DO THINK OVER THE PROBLEMS A LITTLE, before you proceed to see the solution.

PROBLEM SET – 1

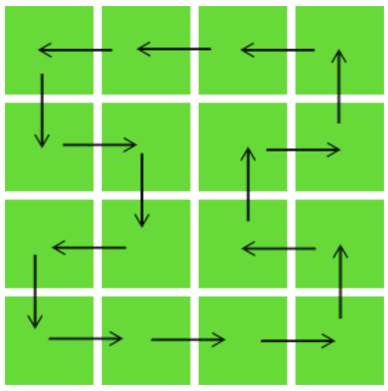
Problem 1.1 ADASCOOL – Ada School

Problem Synopsis:

You are given an N x M matrix where every element can be swapped by its direct neighbour (up, down, left and right). Is it possible to swap them such that no element is at its original place?

Explanation:

The image showing the movement of the students is very deceptive.



There are three key observations, which one can make here:

<u>Observation 1:</u>

The problem says, *the element can be swapped by its direct neighbour viz. up, down, left or right; such that no element is at its original place.*

To achieve the end goal, we don't need to swap the students in the complex way as explained. Instead, we can think of a much simpler swapping scheme.

Imagine that there are 6 columns. We aren't focusing on rows now. There can be any number of rows. In this case, we can swap entire column 1 with column 2 by swapping two elements, belonging to these two rows, at every row. Similarly, we can swap entire column 3 with column 4 and column 5 with column 6. Notice that no element is at its original position after such shuffling. *So, if the number of columns is even – we are done!*

But what if the number of columns is odd? Do we stop? No! We now try to do same thing with rows. Check if the number of rows is even. If yes, then we are done again. We need to swap row 1 with row 2, row 3 with row 4 and so on. *So, if the number of rows is even, we are done again!*

This clearly means that there is no issue if both of them are even. You can swap the adjacent columns or adjacent rows. The problem comes when both of them are odd. After noting few examples on paper, you can realize that – it is not possible to shuffle all the students if the number of rows, as well as columns, are odd. *There will be one and only one element, who will not be able to swap its position in this case.*

Observation 2:

With observation 1, we have already reached the solution. This observation is just to demonstrate – *how a complex looking problem has such a trivial logic!*

The logic of the solution for this problem is just this – *Take two numbers, n and m, and check if at least one of them is even or not. If yes, print 'YES'. Or else, both are odd, just print 'NO'.* Just look at the way, how this logic has been framed to form such a question!

Observation 3:

Even though, the problem is over – I would like you to think over Observation 2 once again. We just saw, how such a trivial logic was used to form such a question. *Can you use this same logic and think of some another use-case or a scenario or a problem, in which you can fit this logic?*

Here is something similar which you can observe on the chess board. The below discussion/observation also serves as *the intuitive proof of why the solution can't exist when both, n and m, are odd.*

Label any cell white. Its neighbours must be black. If N and M are both odd, then there is a total of odd number of cells; which means that the number of white cells is NOT equal to the number of black cells. Ultimately, the operation asks us to place a white cell to its neighbouring black one, and vice versa. But if number of white and black cells are not equal, we can see that there will always be 1 student left out in every configuration. This completes the proof for non-existence of answer for odd M and odd N.

Solution:

C++ code:

```
int main() {
    int t; cin >> t;
    while (t--) {
        int n, m; cin >> n >> m;
        if (n%2 == 0 || m%2 == 0)
            cout << "YES" << endl;
        else
            cout << "NO" << endl;
    }
    return 0;
}
```

Python code:

After this problem, I will be providing only 'def main():' section, as checking 'if __name__ == "__main__":' is constant for all python codes.

```
def main():
    t = int(input())
    for _ in range(t):
        n, m = map(int, input().split())
        if n%2 == 0 or m%2 == 0:
            print("YES")
        else:
            print("NO")

if __name__ == "__main__":
    main()
```

Problem 1.2 CHCHCL – Chef and Chocolate

Problem Synopsis:

Two players are breaking a chocolate bar of N x M size into pieces. A player's turn consists of choosing one piece and breaking it in two. A player who is unable to break the chocolate further loses. Which player wins the game?

Explanation:

This is a very easy problem. We know that the chocolate can be broken into N x M pieces at max. Once we have N x M chocolate pieces of size 1 x 1, the next player loses. So, the outcome of the game depends on who makes the N*Mth move. If N*M is odd, the first player to make the move loses; otherwise, if N*M is even, the second player loses.

Try it with some sample cases to convince yourself.

So, the logic is – *just check if a number is even or odd!*

Solution:

C++ Code:

```
int main() {
    int t; cin >> t;
    while (t--) {
        int n, m; cin >> n >> m;
        if ((1ll*n*m)%2 == 0)
            cout << "Yes" << endl;
        else
            cout << "No" << endl;
    }
    return 0;
}
```

*We have used '1ll' in multiplication of n*m, to typecast the product to 'long long' type to avoid overflows. However, there are no such overflow issues in Python.*

Python Code:

```
def main():
    t = int(input())
    for _ in range(t):
        n, m = map(int, input().split())
        if (n*m)%2 == 0:
            print("Yes")
        else:
            print("No")
```

Problem 1.3 COKE – Chef Drinks Coke

Problem Synopsis:

Given temperatures and prices of N coke cans. You want to drink the cheapest coke which quenches your thirst. You know a coke shall quench your thirst if it's temperature if within the range [L, R]. All these cokes lie at distance M and the normal temperature is K.

For every second, if the current temperature of coke is x, it changes as follows.

- If x > K, x reduces by 1.
- If x < K, x increases by 1.
- If x == K, x remains same.

Explanation:

It can be seen that we can consider each coke separately. For each coke, we need to determine whether after M seconds, whether its temperature is within the acceptable range, and for all cokes, we can take the one with minimum cost. If no coke satisfies, the answer is -1.

Now, we need to determine - what shall be the temperature of coke after M seconds. If Initial temperature is more than K, then every second the temperature reduces until it reaches K. Once it reaches K, it remains K. So, the final temperature can be written as $max(K, C_i - M)$.

In other case, temperature increases every second till it reaches K. So, the final temperature can be written as $min(K, C_i + M)$ (as we want to stop as soon as it reaches temperature K).

Hence, we can check for each code whether it is within the range [L, R] and take minimum cost.

Solution:

C++ Code:

```
int main() {
    int t; cin >> t;
    while (t--) {
        int n, m, k, l, r; cin >> n >> m >> k >> l >> r;
        int minimum = 1e9; int c[n], p[n];
        for (int i = 0; i < n; i++) {
            cin >> c[i] >> p[i];
            if (abs(c[i]-k) <= m)
                c[i] = k;
            else if (c[i] > k)
                c[i] -= m;
            else
                c[i] += m;

            if (l <= c[i] && c[i] <= r)
                minimum = min(minimum, p[i]);
        }
        if (minimum == (int)1e9)
            minimum = -1;

        cout << minimum << endl;
    }
    return 0;
}
```

Python Code:

```python
def main():
    for _ in range(int(input())):
        n, m, k, l, r = map(int, input().split())
        least_price = []
        for i in range(n):
            c, p = map(int,input().split())
            time = 0
            while time < m:
                if c > k + 1:
                    c -= 1
                    time += 1
                elif c < k - 1:
                    c += 1
                    time += 1
                else:
                    c = k
                    time += 1
            if l <= c and c <= r:
                least_price.append(p)
        if len(least_price) == 0:
            print(-1)
        else:
            print(min(least_price))
```

Problem 1.4 COPS - Cops and the Thief Devu

Problem Synopsis:

There are 100 houses in a lane, numbered from 1 to 100. N cops are positioned in some houses. A cop's running speed is h houses per second and he can run for at max t seconds. Find number of houses where a thief can hide such that he won't be caught by any of the cops.

Explanation:

For a particular house, we want to find out whether this house can be checked by some cop or not. We know that a cop can cover a maximum of $h \times t$ inter-house distances in t seconds. So, if the distance between the thief's hiding house and cop's house is less than or equal to $h \times t$, then the cop can catch the thief. We just need to check whether the current house can be reached by any of the cops or not. If yes, then it is not safe otherwise it is safe.

So, we can describe the solution succinctly as follows:

```
ans = 0
for each house from 1 to 100:
    safe = true
    for each cop houses from 1 to N:
       if (the cop can reach the house)
          safe = false
    if (safe) ans += 1
```

Clearly the above implementation of the problem will take $O(100.N)$ time. (*Will discuss about complexity analysis in the next chapter.*)

A faster solution is as follows:

Let us say thief is currently at house pp and we want to check whether he will be safe in this house or not. If we can find the nearest cops in both directions of the lane from current house, then we just need to check whether these nearest cops in either direction can reach the house p in time or not. We will describe a method for finding nearest cop in forward direction faster than $O(N)$ time. Backward direction can be handled similarly.

Let us can create a sorted array of houses of cops. We want to find the first element in the array having value ≥ p. This can be done by using binary search over the array. Time complexity of this will be $O(N)$ per search operation in array.

You can also find the same thing using $lower\_bound$ in set in C++. Set maintains a balanced binary search tree underneath it, which takes $O(logN)$ time for each $lower\_bound$ query.

So, this solution runs in $O(100 * logN)$ time.

Can we make it even faster?

We have to find $nextCopHouse/prevCopHouse$ information for each house faster. Let us see how can find $nextCopHouse$ information faster. Let us make a boolean array $isCop$ of size 100 where $isCop[i]$ denotes that there is a cop in i^{th} house or not.

Now, we go from house number 100 to 1 and update the $nextCopHouse$ information by maintaining the position of latest house having cop in it.

```
latestHouseHavingCop = -1;
for house p from 100 to 1:
    if (there is a cop in the house):
        latestHouseHavingCop = p;
    nextCopHouse[p] = latestHouseHavingCop;
```

Time complexity of this solution is $O(N + 100)$.

Solution:

C++ Code:

```cpp
int main() {
    int t; cin >> t;
    while (t--) {
        int m, x, y; cin >> m >> x >> y;
        long long count = 0;
        bool a[101] = {false};
        x = x*y;
        for (int i = 0; i < m; i++) {
            cin >> y;
            int l = 1 > y-x ? 1 : y-x;
            int r = 100 < y+x ? 100 : y+x;
            for (int j = l; j < r; j++)
                a[i] = true;
        }
        for (int k = 1; k <= 100; k++)
            if (!a[k])
                count++;
        cout << count << endl;
    }
    return 0;
}
```

Python Code:

```python
def main():
    for _ in range(int(input())):
        cops_houses, houses_per_min, search_mins = map(int, input().split())
        houses_searched = houses_per_min*search_mins
        list_of_cop_houses = list(map(int, input().split()))
        list_of_cop_houses.sort()
        lower_range = 0; higher_range = 0; escape_houses = 0

        for cop_house in list_of_cop_houses:
            lower_range = cop_house - houses_searched - higher_range
            if lower_range > 0:
                escape_houses = escape_houses + lower_range - 1
            higher_range = cop_house + houses_searched
        if higher_range < 100:
            escape_houses = escape_houses + 100 - higher_range
        print(escape_houses)
```

Problem 1.5 CUTBOARD – Cut the board

Problem Synopsis:

Given N x M chessboard, make cuts on some of the edges such that the board does not get split into pieces. How many such cuts can you make at maximum?

Explanation:

Note that we can make cuts between every two-consecutive row of cells. There will be $N - 1$. Such rows to be cut. Within a row we can make $M - 1$ cuts. This way of cutting will make the board look like an extended E - shape structure.

So, number of cuts is $(N - 1) \times (M - 1)$.

Solution:

C++ Code:

```
// You just have to output (m-1)*(n-1).
```

Python Code:

```
# You just have to output (m-1)*(n-1).
```

Problem 1.6 DSTAPLS – Distribute Apples

Problem Synopsis:

Given N apples and K boxes, and you have 2 candidates who fill the boxes as following:

- Every minute, first candidate puts 1 apple in each of the K boxes.
- Every minute, the second candidate puts K apple in the box with least number of apples

We need to tell if the final configuration of apples filled by both candidates will be same or not.

Explanation:

Configuration can be equal only at times t = K, 2K, 3K... and so on. Make sure that you have enough apples so that all apples are exhausted at these favourable values of t. Realize that since first candidate puts 1 apple in each box every minute, while the second candidate puts K apples at once in the box he chooses. The configuration can only be same at a time t such that t is a multiple of K. This puts a restriction that N must be divisible by K^2, so that all the apples are exhausted only at such favourable t.

I will give 2 points of view or perspectives for the proof. Each point of view is based on deriving the proof by looking at actions of the candidate.

The first candidate puts 1 apple in each of the boxes, while second candidate puts K apples in a box per minute. This also implies that, number of apples the second candidate has put in some box is always a multiple of K (as he puts K apples every time!). Now, the final configuration can only be same if the number of apples put in by first candidate is also a multiple of K. This can occur only if time t is a multiple of K, as the first candidate puts just 1 apple per minute. Since t must be a multiple of K as proved just now, let $t = i \times K$ where i is some integer. Now, since the number of boxes is also K, the second candidate has filled each of the K boxes with $K \times i$ apples. Also, the first candidate puts 1 apple every minute, and hence has also filled all boxes with $K \times i$ apples. Hence the final configuration is same at $t = K \times i$ for all integral values of i.

The alternate perspective of proof his from point of view of second candidate. In first minute, the first candidate has filled all boxes with 1 apple. Now, the final configuration can only be same iff the second candidate also fills all the boxes - else it's obvious that final configuration is different. This is possible only if t is a multiple of K. Now, we know that final configurations are same at $t = K \times i$ where i is some integer. This imposes a very nice restriction on N.

The restriction is that - N must be such that all N apples are exhausted at time $t = K \times i$. We also know that each minute K apples are being used by each candidate! This means the total number of apples N, such that all apples are used at $t = K \times i$ is $N = i \times K^2$. This means that N must be divisible by K^2.

Writing a solution in Python or Java is comfortable. For C++, where support for numbers > 2^{64} is not there by default, we will use below trick. Now, since $K \leq 10^{18}$, hence squaring K will cause overflow. The usual procedure to then check if N is divisible by K^2 is to first check if $N \,\%\, K == 0$ and then check that $(N/K) \,\%\, K == 0$. Since it's given that N is always a multiple of K, we just need to check if $(N/K) \,\%\, K = = 0$. If it's true, then the configurations are same - else they are not!

Solution:

C++ Code:

```
int main() {
    int t; cin >> t;
    while(t--) {
        long long n, k; cin >> n >> k;
        if ((n/k)%k == 0)
            cout << "NO" << endl;
        else
            cout << "YES" << endl;
    }
    return 0;
}
```

Python Code:

```
# Translate the C++ code. It't just one if-check.
```

Problem 1.7 LAPIN - Lapindromes

Problem Synopsis:

Given a string S, if we split it in the middle (if S has an odd number of characters, disregard the middle character), then if the frequency of each character is the same in both halves, S is called a “Lapindrome”. Given the string S, test if it is a lapindrome or not.

Explanation:

Maintain frequencies for the left half and the right half for each character. After computing the frequency of each half, then check if the frequencies of all the characters match. If so, output “YES”, else output “NO”.

The one thing which you should learn from this code is, how by subtracting ‘a’ from a character, we are mapping the characters from their ASCII value to a range of [1, 26].

Solution:

C++ Code:

```
bool isLapin(string s) {
    int n = s.length(), freqLeft[26] = {0}, freqRight[26] = {0};
    for (int i = 0; i < n/2; i++)
        freqLeft[s[i]-'a']++;

    for (int i = (n+1)/2; i < n; i++)
        freqRight[s[i]-'a']++;
    bool flag = true;
    for (int c = 0; c < 26; c++) {
        if (freqLeft != freqRight)
            flag = false;
    }
    return flag;
}
```

Python Code:

```
def main():
    t = int(input())
    for _ in range(t):
        string = input()
        length = len(string)
        mid = length//2
        if length%2 == 0:
            one = string[:mid]
            two = string[mid:]
        else:
            one = string[:mid+1]
            two = string[mid:]
        for i in one:
            if one.count(i) != two.count(i):
                print("NO")
                break
        else:
            print("YES")
```

Problem 1.8 LIFTME – Lift Requests

Problem Synopsis:

Given are $1 \le Q \le 10^6$ requests. Initially the lift is at floor 0. In each request a lift first moves to floor F_i from its last location, and then moves to floor D_i. Finally, the lift can be at any floor. Find total number of floors traversed. $F_i, D_i \le N \le 10^6$.

Explanation:

Let's set $curr = 0$, initially. Let the total number of floors travelled be $totalCount$. Now, when we first go to floor F_i, number of floors thar we are travelling are $|F_i - curr|$. Next, we go to D_i. Then, the number of floors travelled = $|D_i - F_i|$. We just need to add this to $totalCount$ and set $curr = D_i$ and we ready to process the next request.

Solution:

C++ Code:

```
int main(){
    int t; cin >> t;
    while(t--){
        int n, q; cin >> n >> q;
        long long sum = 0;
        long long curr = 0;
        while(q--){
            int a,b; cin >> a >> b;
            sum += abs(a-curr) + abs(b-a);
            curr = b;
        }
        cout << sum << endl;
    }
    return 0;
}
```

Python Code:

```
# Translate the C++ code. It's very simple.
```

PROBLEMS FOR PRACTICE:

- AVG – Average Number
- CANDY123 – Bear and Candies
- CARVANS - Caravans
- CFMM – Making a Meal
- CHFMOT18 – Chef and Demonetisation
- PCJ18B – Chef and Board Games
- THREEFR – Three Friends
- TRACE – Trace of a Matrix