

Thinking, Understanding, Reasoning in Algorithms

Includes Data Structures
and Solutions to Problems

T.U.R.A.

By Taksh Kothari and Apurva Bhat

Mentored by Vinit Ajgaonkar

Contents

Preface	6
1. Introductory Problems	7
1.1. Concepts	7
1.1.1. Basic C++ Syntax	7
1.1.1.1. Question	7
1.1.1.2. Solution	7
1.1.1.3. Data Types	8
1.1.1.4. Variables	9
1.1.1.5. Input/Output	9
1.1.1.6. Conditional Statements	9
1.1.1.7. Loops	9
1.1.1.8. Classes/Structs	9
1.1.1.9. Arrays/Vectors	9
1.1.1.10. Functions	9
1.1.1.11. Summary	9
1.1.2. Input Output Optimization	10
1.1.2.1. <code>ios_base::sync_with_stdio(false)</code>	10
1.1.2.2. <code>endl</code> vs <code>"\n"</code>	10
1.1.3. Space & Time Complexity	10
1.1.4. Pointers	11
1.1.5. Vectors in Depth	12
1.1.6. Recursion	13
1.1.7. Sorting	13
1.1.7.1. Binary Search	14
1.1.7.2. Lower Bound and Upper Bound	16
1.1.7.3. <code>lower_bound()</code> & <code>upper_bound()</code> with Custom Sorting	17
1.1.8. Permutations	18
1.1.8.1. <code>next_permutation()</code>	19
1.1.9. Backtracking	20
1.1.10. Queue	22
1.2. Solutions to CSES Questions	25
1.2.1. Weird Algorithm	25
1.2.2. Missing Number	26
1.2.3. Repetitions	27
1.2.4. Increasing Array	29
1.2.5. Permutations	31
1.2.6. Number Spiral	32
1.2.7. Two Knights	34
1.2.8. Two Sets	36
1.2.9. Bit Strings	38
1.2.10. Trailing Zeros	39
1.2.11. Coin Piles	40
1.2.12. Palindrome Reorder	41
1.2.13. Gray Code	43
1.2.14. Tower of Hanoi	46
1.2.15. Creating Strings	48

1.2.16.	Apple Division	49
1.2.17.	Chessboard and Queens	51
1.2.18.	Raab Game I	53
1.2.19.	Mex Grid Construction	55
1.2.20.	Knight Moves Grid	57
1.2.21.	Grid Coloring I	60
1.2.22.	Digit Queries	62
1.2.23.	String Reorder	64
1.2.24.	Grid Path Description	67
2.	Sorting and Searching	70
2.1.	Concepts	70
2.1.1.	Bit Operations	70
2.1.1.1.	AND (&)	70
2.1.1.2.	OR ()	70
2.1.1.3.	XOR (^)	70
2.1.1.4.	NOT (~)	70
2.1.1.5.	Left Shift (<<) and Right Shift (>>)	71
2.1.1.6.	Lowest Set Bit (LSB)	71
2.1.2.	Bitmask	71
2.1.3.	Prefix Sum	72
2.1.4.	Binary Indexed Tree	74
2.1.4.1.	Fenwick Trees as Indexed Sets	77
2.1.4.2.	Index Compression	78
2.1.5.	Linked List	81
2.1.5.1.	<code>std::list</code>	82
2.1.6.	Greedy Algorithms	84
2.1.7.	Sets	85
2.1.7.1.	<code>lower_bound</code> and <code>upper_bound</code>	86
2.1.7.2.	<code>multiset</code>	86
2.1.7.3.	<code>unordered_set</code>	86
2.1.7.4.	<code>unordered_multiset</code>	86
2.1.8.	Maps	87
2.1.8.1.	Important Note on Accessing Keys	88
2.1.8.2.	<code>lower_bound</code> and <code>upper_bound</code>	88
2.1.8.3.	<code>multimap</code>	88
2.1.8.4.	<code>unordered_map</code>	88
2.1.8.5.	<code>unordered_multimap</code>	89
2.1.9.	Stacks	89
2.1.9.1.	Common Use Cases	90
2.1.9.2.	Size and Empty Check	91
2.1.9.3.	Stack vs Vector	91
2.1.9.4.	Example: Balanced Parentheses	91
2.1.9.5.	Important Notes	92
2.1.10.	Lambda expressions	92
2.1.11.	Sorting with a custom sorting order.	93
2.2.	Solutions to CSES Questions	94
2.2.1.	Distinct Numbers	94
2.2.2.	Apartments	95

2.2.3.	Ferris Wheel	97
2.2.4.	Concert Tickets	99
2.2.5.	Restaurant Customers	101
2.2.6.	Movie Festival	102
2.2.7.	Sum of Two Values	104
2.2.8.	Maximum Subarray Sum	106
2.2.9.	Stick Lengths	107
2.2.10.	Missing Coin Sum	109
2.2.11.	Collecting Numbers	111
2.2.12.	Collecting Numbers II	112
2.2.13.	Playlist	115
2.2.14.	Towers	117
2.2.15.	Traffic Lights	118
2.2.16.	Distinct Values Subarrays	119
2.2.17.	Distinct Values Subsequences	121
2.2.18.	Josephus Problem I	123
2.2.19.	Josephus Problem II	124
2.2.20.	Nested Ranges Check	126
2.2.21.	Nested Ranges Count	128
2.2.22.	Room Allocation	132
2.2.23.	Factory Machines	134
2.2.24.	Tasks and Deadlines	136
2.2.25.	Reading Books	138
2.2.26.	Sum of Three Values	140
2.2.27.	Sum of Four Values	142
2.2.28.	Nearest Smaller Values	144
2.2.29.	Subarray Sums I	145
2.2.30.	Subarray Sums II	147
2.2.31.	Subarray Divisibility	149
2.2.32.	Distinct Values Subarrays II	151
2.2.33.	Array Division	153
2.2.34.	Movie Festival II	155
2.2.35.	Maximum Subarray Sum II	157
3.	Dynamic Programming (WIP)	159
4.	Graph Algorithms (WIP)	159
5.	Range Queries (WIP)	159

Preface

Among the many resources available to learners, the CSES Problem Set is a remarkably well-curated collection, grouping problems by the underlying algorithms and ideas rather than by difficulty alone.

This book builds directly on that approach. The algorithms are organized into sections that mirror the headings of the CSES Problem Set, and each section contains clear explanations that complement the problems sourced from CSES.

Whether you are a beginner exploring competitive programming for the first time or an experienced coder refining your approach, the content is structured to be both accessible and enriching. Instead of relying on heavy notation or unnecessary complexity, explanations focus on clarity and intuition.

Additionally this is a work in progress. It has an active [GitHub repository](#) and contributions from the community are warmly welcomed. With collective effort, this book can become more comprehensive, refined, and enjoyable for readers around the world.

We acknowledge the use of AI tools in the creation of this book and for the cover. All material presented is our own work and has been carefully reviewed to ensure accuracy, clarity, and originality.

We are grateful to our mentor Mr. Vinit Ajgaonkar who started us in our competitive programming journey. His inputs were valuable in the creation of this book.

Lastly, the algorithms in this book can be implemented in any programming language. However, we use C++ because the International Olympiad in Informatics has switched to using only C++ (since 2021), and C++ is the most commonly used language for contests such as the International Collegiate Programming Contest.

Happy coding, and welcome to the journey!

1. Introductory Problems

1.1. Concepts

1.1.1. Basic C++ Syntax

This section will go over the basic c++ syntax with a simple question and the implementation to solve the question. The code will cover the main elements required to write basic c++ programs.

1.1.1.1. Question

Write a C++ program that processes student performance data. First, accept the total number of students from the user. Subsequently, collect each student's name along with their corresponding marks.

The program should then analyze this data to identify and display the name or names of all students who achieved the highest percentage among their peers. In cases where multiple students share the top score, all their names should be printed.

1.1.1.2. Solution

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  double calcPercent(int numerator, int denominator){
6      return numerator * 100.0 / denominator;
7      // An example of a single line comment
8      /* An example of a multiline comment
9          numerator / denominator * 100.0 will first do integer division.
10         That's why we multiply by 100.0 first and then divide by the
11         denominator. */
12 }
13 struct Student{
14     string name;
15     pair<int, int> marks;
16     double percent;
17     Student(); // this is a default constructor
18
19     // this is a parameterized constructor
20     Student(string name, pair<int, int> marks) {
21         this->name = name;
22         this->marks = marks;
23         percent = calcPercent(marks.first, marks.second);
24     }
25 };
26
27 // Program execution begins from here
```

C++

```

28 int main() {
29     int n;
30     cin >> n;
31
32     // to create an array of n Students the default constructor was necessary
33     Student arr[n];
34     double maxPercentage = 0.0;
35
36     for(int i = 0; i < n; i++){
37         string name;
38         pair<int, int> marks;
39         cin >> name >> marks.first >> marks.second;
40
41         // calling parameterized constructor
42         arr[i] = Student(name, marks);
43         maxPercentage = max(arr[i].percent, maxPercentage);
44     }
45
46     // a vector is a resizable array with some useful functions
47     // memory is automatically allocated in a vector
48     vector<Student> best;
49
50     for(int i = 0; i < n; i++)
51         if(arr[i].percent == maxPercentage) {
52             // push_back() adds the student to the end of the vector
53             best.push_back(arr[i]);
54         }
55
56     cout << "Names, Marks and Percentages of top scorers!" << endl;
57     for(int i = 0; i < best.size(); i++) {
58         cout << "Name: " << best[i].name;
59         cout << ", Marks: " << best[i].marks.first << "/" <<
            best[i].marks.second;
60         cout << ", Percentage: " << best[i].percent << endl;
61     }
62
63     return 0; // end code
64 }

```

While this isn't the only way to solve the question, the code should cover the most basic C++ syntax.

1.1.1.3. Data Types

This code contained the following data types:

1. int - Integer
2. double - floating point

3. `string` - Text

4. `pair<int,int>` - A `pair` is a data type that can be a combination of 2 other data types, and each individual part can be accessed with `.first` and `.second`. In this case it was 2 ints, but it could be a pair of `int` and `string` and much more.

1.1.1.4. Variables

Variables are fundamental building blocks in programming that serve as named storage locations in a computer's memory where you can store data that your program needs to work with. Variables are strongly typed in C++, which means you must specify their data type and then their name.

1.1.1.5. Input/Output

Input and output are done with `cin` and `cout` and angle brackets: `>>` for input and `<<` for output.

1.1.1.6. Conditional Statements

Conditional statements are represented with `if`. The part inside the `if` block runs if the condition is true. You can also use `else`, which triggers if the `if` block above is `false`, and create if-else ladders with `else if`, which triggers if the above `if` and `else if` blocks were `false`.

1.1.1.7. Loops

A loop in the example is a `for` loop, which has 3 parts. The first part initializes a variable. The second part is the condition to determine if the loop should continue, and the third part is what happens at the end of the loop block, which is usually to update the variable initialized in the first part.

1.1.1.8. Classes/Structs

In this program we made a struct because they're easier to use than a class. They work in nearly the same way, though, and the only difference is that members in a struct are public by default, while members in a class are private by default.

1.1.1.9. Arrays/Vectors

An array is a list of many of the same data type. In this program we made an array of `Students`, which is our own data type. We also made a vector, which, unlike an array, has a dynamic size.

1.1.1.10. Functions

A function is something that accepts parameters and returns a value. This includes our `calcPercent` function and the 2 constructors used to make `Student`.

1.1.1.11. Summary

These basic concepts are meant to serve as a revision or as an introduction to C++ syntax for those of you recently switching to C++. This is just the tip of the iceberg, and there is much more to learn about C++ as you go along.¹

¹More about C++ syntax can be learned [here](#).

1.1.2. Input Output Optimization

1.1.2.1. `ios_base::sync_with_stdio(false)`

By default, C++ streams (`cin`, `cout`) are synchronized with C's `stdio` (`scanf`, `printf`) to allow interleaved use. This synchronization adds overhead. Calling `ios_base::sync_with_stdio(false)` disables this synchronization, making C++ streams significantly faster—often 2-3x faster for large inputs. However, once disabled, you cannot mix C++ streams with C-style I/O in the same program. This is particularly useful in competitive programming where performance matters and you only use `cin/cout`.

`std::cin` is also tied to `std::cout`, which means that whenever input is accepted, the output is first flushed (displayed) before accepting input, which is useful for interactive programs but not needed for competitive programming. We usually untie these with `cin.tie(nullptr)`.

```
1 ios_base::sync_with_stdio(false);
2 cin.tie(nullptr);
```

C++

1.1.2.2. `endl` vs `"\n"`

The `endl` manipulator does two things: inserts a newline character and flushes the output buffer. Flushing is an expensive operation that forces immediate write to the output device. In contrast, `"\n"` only inserts a newline without flushing. For programs with many output lines, using `"\n"` instead of `endl` can dramatically improve performance since the buffer is flushed automatically when full or when the program ends. Use `endl` only when you need immediate output (like interactive programs) or while debugger for when you need to know exactly when the code breaks; otherwise, use `"\n"`.

```
1 // Slow
2 cout << "Result: " << x << endl;
3 // Fast
4 cout << "Result: " << x << "\n";
```

C++

1.1.3. Space & Time Complexity

Space complexity is simply a measure of how much memory a program requires and time complexity is simply a measure of how much longer it takes a program to run as the input size grows larger. We represent it by using something called Big-O Notation. For instance, say we have a program that has a time complexity of $O(n)$; this means that the function is linear, i.e., if you double the input size, the program will take twice as long. A program with time complexity $O(n^2)$ will take 4 times as long for twice the input size.

Similarly, a program with a space complexity of $O(n)$ will take twice the memory for double the input size. E.g a program that uses an array to store the n values in the input. A program with a space complexity of $O(n^2)$ will take 4 times as much memory for double the input.

When solving a problem, always analyze the time complexity of your algorithm. To verify your solution will run efficiently, substitute the maximum input size into your time

complexity formula. The resulting number of operations should be less than 10^8 , as this is approximately how many operations a typical computer can execute per second. Also check your space complexity and memory limits. While the constraints on space complexity are more lenient, some problems will require you to think about how to optimize memory.

The time complexity for the CSES Questions which you will see throughout the book have a time limit of 1.00s and a memory limit of 512MB.

1.1.4. Pointers

Unlike in other higher-level programming languages which you may be familiar with, C++ allows you to have full control over how to allocate memory. This is achieved by using pointers.

A pointer is a variable that stores a memory location instead of the value. Here's an example of code which uses pointers, and we'll explain what it does:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      int a = 5; //made an int variable with value 5.
6      int *b = new int(7); //made an integer pointer with value 7 at that memory
        location.
7      int &c = a; //made an int variable which refers to the same memory location
        as a.
8      int *d = &a; //made an int pointer which points to the same memory location
        as a.
9
10     cout << a << " " << *b << " " << c << " " << *d << endl;
11     c = 9; //also changes a and d
12     cout << a << endl; //9
13     *d = 15; // also changes a and c
14     cout << a << endl; //15
15
16     delete b; //every time you use "new" you must always "delete" the pointer to
        prevent memory leaks
17     return 0;
18 }
```

While we have written comments, we'll still go deeper to explain the most important lines:

- `int a = 5` creates a variable `a` which has a value of 5.
- `int *b = new int(7)` makes a pointer `b`, which at its memory location has the value 7.
- `int &c = a` makes a variable `c` which has the same value that `a` has. This means that modifying one of them will modify the other. They are the same value with 2 different names.

- `int *d = &a` makes a pointer `d` which stores the memory location of `a`. This also makes `d` the same as `a` and `c`; however, `d` is a memory location which at the location has the same value as `a` and `c`.
- `cout << a << " " << *b << " " << c << " " << *d << endl` outputs `a`, `*b` (which is the value at memory location `b`), `c`, and `*d` (which is the value at memory location `d`).
- `c = 9` changes the values of `a` and the value at memory location `d` to 9.
- `*d = 15` changes the value at memory location `d` to 15, which also changes `a` and `c`.
- `delete b` is the most important line. **Every time you use the keyword `new`, you must use `delete` to free up the memory.** Otherwise, that memory will remain allocated to nobody after your program has ended. This is called a memory leak, and the only way to free up such “leaked” memory is by restarting your computer.

To summarize the new syntax of pointers:

1. `int *x` creates a pointer which stores a memory location.
2. `*x` **dereferences** the pointer, allowing you to see the value.
3. `int &x` allows you to pass another variable by reference, i.e., both variables share the same memory location.
4. `&x` gives the memory location of the variable `x`.

1.1.5. Vectors in Depth

We’re going to go into vectors in a little more depth. As stated before, vectors are almost the same as arrays except they are dynamic, meaning elements can be added and removed, but only at the end. This is done by the `push_back()` and `pop_back()` functions.

The way vectors make this efficient time-wise without wasting a lot of memory is by allocating some memory x in a row. When you `push_back()` an element such that it now exceeds x , it moves the entire allocated memory to a new location and allocates memory worth $2x$.

This means that the time complexity of inserting elements into a vector is close, but not quite, $O(1)$. This is called amortized $O(1)$ because it looks at the average instead of each single operation, and because vector resizes occur infrequently.

Note that vectors’ constant factors are bigger than arrays, which means for questions where every little efficiency matters to solve the question, if you don’t need a vector, don’t use one. However, in every other case, it’s much safer and more convenient to use vectors instead of arrays. The main reasons being that:

1. It’s easier to initialize all values in a vector:

```
1 vector<int> v(5, -1); //Initializes vector of size 5 filled with -1
```

C++

2. When passing an array to a function, it **always** passes by reference. Passing by reference simply means that the function can make changes to the original array. Sometimes, however, we wish to pass by value, meaning that a new copy is made. With vectors, we have such freedom to choose.

More technical details about vectors can be found [here](#).

1.1.6. Recursion

Recursion is the concept of calling some function inside of itself. Say we want to compute the factorial of a number n . We can do:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int fact(int n){
5      if(n == 1) //base case
6          return 1;
7      return n * fact(n - 1); //recursion
8  }
9
10 int main(){
11     int n;
12     cin >> n;
13     cout << fact(n) << endl;
14
15     return 0;
16 }
```

C++

Every recursive algorithm has 2 main things:

1. A base case: some failure point at which you must return a known value. In this code it was $n = 1$, and we returned 1 for that base case. You can always have multiple base cases if necessary.
2. Recursion: this is the part where you call the original function on a smaller problem than the original. In this case we call `fact(n-1)` and then multiply it by n to get `fact(n)`.

Fun fact: It's proven that any recursive function can be written with a loop! Loops are more efficient than recursion, so if it is easier to write a loop, you should. However, some programs are too hard to convert to loops, so you should stick to recursion.

1.1.7. Sorting

To sort a data structure like an array or vector, C++ has its own sort function for this:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      int arr[] = {3,4,6,2,5,1};
6      vector<int> v = {6,2,4,5,1,3};
7
8      sort(arr, arr+6); //Sorts the array {1,2,3,4,5,6}
9      sort(v.begin(), v.end()); //Sorts the vector {1,2,3,4,5,6}
10 }
```

C++

```

11     return 0;
12 }

```

As you can see, the sort function accepts 2 pointers, the start position of the sort and one position after the end of where you want the elements sorted. `arr` is a pointer to the start of the array. You can add a number to this pointer to jump ahead that many places.

`arr + 6` is one position past the end of the array because we want to sort the entire array in this case, although you don't always have to. `v.begin()` is a pointer to the start of the vector and `v.end()` points one place after the last element of the vector. You can also add a value to `v.begin()` to jump to other positions in the vector to sort only a part of it.

The time complexity of `std::sort` is $O(n \log n)$.

1.1.7.1. Binary Search

Let's say you want to find a certain number in a list of numbers to see if it exists. Normally, the way you would do this is by iterating over each element in the array and checking if it matches the element you're looking for. The time complexity of this is $O(n)$. However, if we were to first sort the array, we can find a number in $O(\log n)$!

You may not have realized it, but you have probably already used binary search in your life at least once! Whenever you use a dictionary, you don't search word by word to see if it matches the word you are looking for, you instead apply something similar to binary search.

Say you're looking for the word "computers". You open to the middle of the dictionary. You'll probably be in the m-n section, which is too far ahead, so you jump back halfway. You repeat this until you get to the c section. However, you may be in the ca section, which is now behind, so you jump forward halfway until you reach "computers". While you may not be doing exactly this, we can use this method to find things really quickly.

The main steps are as follows:

1. Start in the middle
2. If you are equal to the target, you have found the value! Otherwise, go to step 3.
3. If you are less than the target, eliminate the left half and then jump to the middle of the right half, then go back to step 2. If not, go to step 4.
4. If you are more than the target, eliminate the right half and jump to the middle of the left half, then go to step 2.

Let's see the algorithm in action on an example. Say we have the following sorted array:

`{1, 4, 4, 5, 6, 6, 7, 9, 13, 15, 16, 18, 21, 30}`

And let the target number we are looking for be 18. Let there be the variables `left` = 0, `right` = 13, and `middle` = $\frac{\text{left} + \text{right}}{2} = \frac{0 + 13}{2} = 6$, which is the average of `left` and `right`.

`{1, 4, 4, 5, 6, 6, 7, 9, 13, 15, 16, 18, 21, 30}`

Now we can compare the value of `middle` with our target, 18. As you can see, `middle` < 18. This tells us that our target value lies to the right of `middle`. We can now update `middle` by first making `left` = `middle` + 1 = 6 + 1 = 7, then make `middle` = $\frac{\text{left} + \text{right}}{2} = \frac{7 + 13}{2} = 10$.

{1, 4, 4, 5, 6, 6, 7, 9, 13, 15, 16, 18, 21, 30}

Once again we are too low, so we set $\text{left} = \text{middle} + 1 = 10 + 1 = 11$, and then $\text{middle} = \frac{\text{left} + \text{right}}{2} = \frac{11+13}{2} = 12$.

{1, 4, 4, 5, 6, 6, 7, 9, 13, 15, 16, 18, 21, 30}

This time we're too high, so now we set $\text{right} = \text{middle} - 1 = 12 - 1 = 11$, and then $\text{middle} = \frac{\text{left} + \text{right}}{2} = \frac{11+11}{2} = 11$.

{1, 4, 4, 5, 6, 6, 7, 9, 13, 15, 16, 18, 21, 30}

Now middle is equal to 18, our target! And it only took us 4 steps. If we had iterated normally, it would've taken 12.

Here is the implementation of this, where the user will supply us with a sorted list of numbers and a target value for us to find. We output whether the value exists and then its position in the list:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      ios_base::sync_with_stdio(0);
6      cin.tie(0);
7      cout.tie(0);
8
9      int n, t;
10     cin >> n >> t;
11     vector<int> v(n);
12     for(int i = 0; i < n; i++)
13         cin >> v[i];
14
15     int l = 0, r = n - 1, m;
16     while(l <= r){
17         m = (l + r)/2;
18         if(v[m] == t){//if the number at m meets the target
19             cout << "YES" << endl;
20             cout << m << endl;
21             return 0;
22         }
23         else if(v[m] < t)
24             l = m + 1;//eliminate the left (lesser) half
25         else if(v[m] > t)
26             r = m - 1;//eliminate the right (greater) half
27     }
28     cout << "NO" << endl;
29     return 0;
30 }
```

C++

1.1.7.2. Lower Bound and Upper Bound

Usually, whenever we do binary search, we rarely ever want to know if a value is actually there or not. Rather, we'd like to know two things:

1. The first number in the list greater than or equal to the number. This is called finding the **lower bound**.
2. The first number in the list **strictly** greater than the number. This is called finding the **upper bound**.

To be able to compute the **lower bound** and **upper bound** of some number t , we only need to modify the while loop of our earlier binary search algorithm:

Lower Bound:

```
1  int l = 0, r = n - 1, m, lb = -1;
2  while(l <= r){
3      m = (l + r)/2;
4      if(v[m] < t)
5          l = m + 1;
6      else if(v[m] >= t){// >= instead of > because it's lower bound.
7          lb = m;//we set the lower bound to the middle
8          r = m - 1;// and then eliminate the right half.
9      }
10 }
11
12 cout << lb << endl;
```

C++

Upper Bound:

```
1  int l = 0, r = n - 1, m, ub = -1;
2  while(l <= r){
3      m = (l + r)/2;
4      if(v[m] <= t)//changed < to <= so the equal condition isn't missed.
5          l = m + 1;
6      else if(v[m] > t){
7          ub = m;//we set the upper bound to the middle
8          r = m - 1;//and then eliminate the right half.
9      }
10 }
11
12 cout << ub << endl;
```

C++

You can try the algorithm for lower bound and upper bound on an array with a target value and see how this works.

Now, luckily for you, C++ comes with its own upper bound and lower bound functions! Here are their use cases:

Note that `upper_bound()` and `lower_bound()` only work properly on sorted lists in ascending order. They will output the wrong value otherwise.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      ios_base::sync_with_stdio(0);
6      cin.tie(0);
7      cout.tie(0);
8
9      int n, t;
10     cin >> n >> t;
11     vector<int> v(n);
12     for(int i = 0; i < n; i++)
13         cin >> v[i];
14
15     vector<int>::iterator lb = lower_bound(v.begin(), v.end(), t); //
    lower_bound() returns an iterator to the position of the lower bound of t.
16     vector<int>::iterator ub = upper_bound(v.begin(), v.end(), t); //
    upper_bound() returns an iterator to the position of the upper bound of t.
17
18     cout << lb - v.begin() << endl; //difference between lb and v.begin() tells
    you the index of the lower bound.
19     cout << ub - v.begin() << endl; //difference between ub and v.begin() tells
    you the index of the upper bound.
20     return 0;
21 }
```

As you can see from the code above:

- `lower_bound(v.begin(), v.end(), t)` returns an iterator to the lower bound of `t`.
- `upper_bound(v.begin(), v.end(), t)` returns an iterator to the upper bound of `t`.

To get the index, we simply do `lb - v.begin()` and `ub - v.begin()` because that takes the difference in memory location.

You now might wonder how to get the largest element less than or the largest element less than or equal to a target. This can be achieved by subtracting 1 from the lower bound and upper bound, respectively.

1.1.7.3. `lower_bound()` & `upper_bound()` with Custom Sorting

Sometimes your vector may not be sorted in ascending order. Sometimes it might be descending, or it could have some custom ordering. In these cases, it's important to understand what `lower_bound()` and `upper_bound()` are actually doing.

`lower_bound(first, last, val, comp())` returns an iterator of the first value where `comp(*it, val)` is false.

`upper_bound(first, last, val, comp())` returns an iterator of the first value where `comp(val, *it)` is true.

By default, the `comp()` function is `operator<()`. However, this can be changed to `greater<int>()`, which returns true if the first number is greater than the second number, which is needed for it to work properly on a descending list. Note, however, that `upper_bound()` and `lower_bound()` may not actually give the mathematical definition of lower bound and upper bound if you use them on a descending list. Apply a correction factor as needed.

1.1.8. Permutations

Let's say you are given a string, and you wish to list out all possible permutations of the string. For instance, "abcde".

You could probably write out all $5! = 120$ possibilities on your own, but what rule could you use to make a computer do it? Try listing the permutations yourself and see if you come up with something before reading onwards.

Alright, here's the method:

Let's first list out the permutations of a string of length 3, "abc":

- "abc"
- "acb"
- "bac"
- "bca"
- "cab"
- "cba"

As you can see, we went through all permutations starting from the string sorted in ascending order and then to descending order.

To then go from one permutation to the next, there are 3 steps:

1. Scan from right to left. The first position where you find the current element less than the next one (`str[i] < str[i+1]`) is the pivot.
2. Swap the element at the pivot with its upper bound to the right of it (See Section 1.1.7.2).
3. Reverse all elements after the pivot.

Try this out with your letter sequence, and you'll see that this is probably what you do intuitively without realizing it.

Here's the code for that algorithm:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  bool permute(string &str){
5      for(int i = str.length() - 2; i >= 0; i--){
6          if(str[i] < str[i + 1]){//pivot finding
```

C++

```

7
8     //finds the lower bound of element at pivot.
9     int lb_idx = lower_bound(str.begin() + (i + 1), str.end(), str[i],
10    greater<int>()) - 1 - str.begin();
11
12    //swaps the lb and the element at the pivot
13    swap(str[i], str[lb_idx]);
14
15    //reverses the elements after the pivot
16    reverse(str.begin() + (i + 1), str.end());
17
18    //successfully produced the next permutation
19    return true;
20 }
21
22 return false; // failed to produce the next permutation. This happens when
23 the string is in descending order because that's the last permutation.
24 }
25 int main(){
26
27     string str = "abcd";
28
29     do {
30         cout << str << endl;
31     } while(permute(str));
32
33     return 0;
34 }

```

1.1.8.1. next_permutation()

Fortunately for you, C++ already has a function that generates the next permutation!

Here's the same code we wrote above but using `next_permutation()`:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      string str = "abcd";
6
7      do {
8          cout << str << endl;

```

C++

```

9     } while(next_permutation(str.begin(), str.end()));
10
11     return 0;
12 }

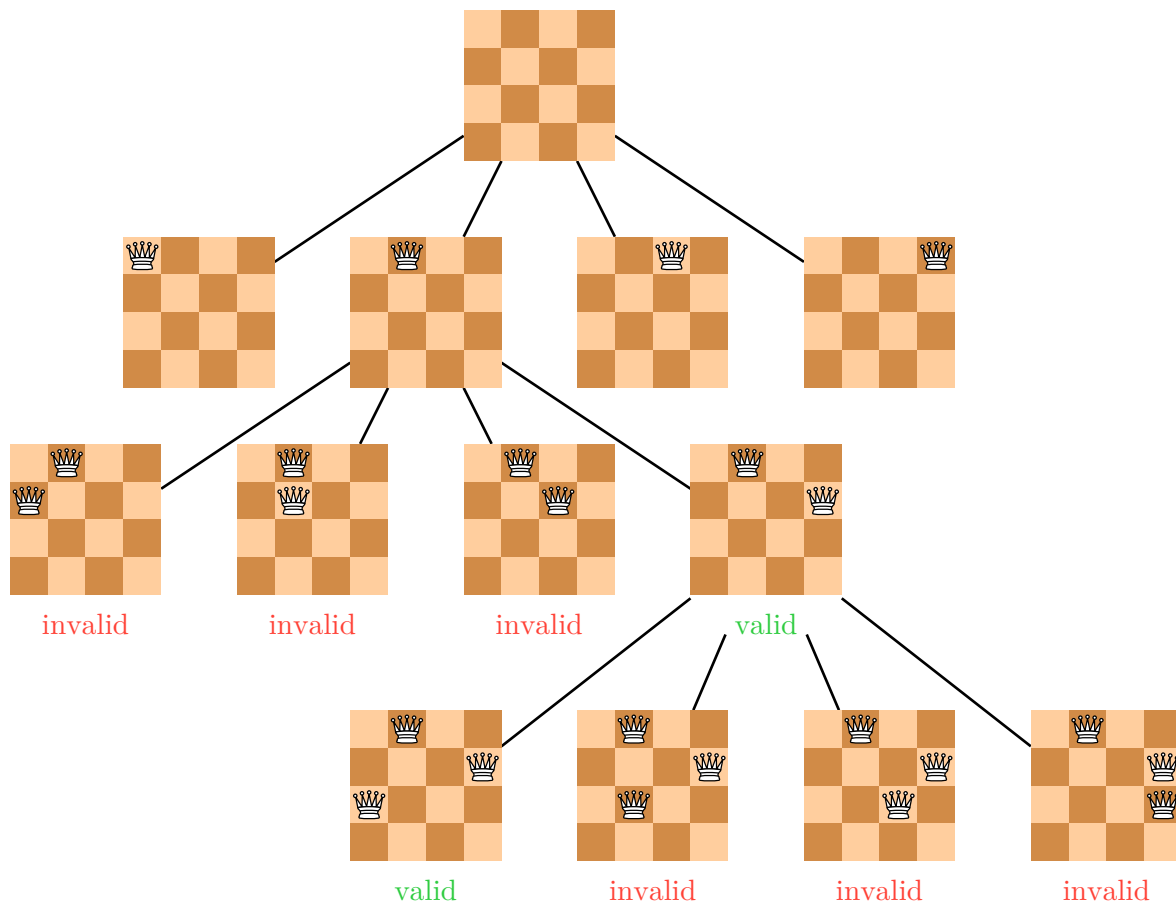
```

1.1.9. Backtracking

A backtracking algorithm is one where you recursively go through all possibilities and then backtrack at invalid solutions. Let's use an example to explain this better.

Say we want to know, for an $n \times n$ chess board, how many ways are there to place n queens such that no two queens attack each other.²

This problem can be solved by using backtracking. We can start by placing the first queen in all positions on the first row. For each of those positions, we see which positions we can place a queen in the second row, and so on. Let's look at some partial solutions when $n = 4$.



As you can see, we start with an empty board, then we place a queen in all positions on the first row. Then we place the next queen on the next row in a valid position and go from there. We only go further if the existing position is valid and prune off any invalid position saving a lot of time as invalid positions don't branch off to take more invalid positions. This is the essence of backtracking.

²If you don't know anything about chess, two queens attack each other if they both lie on the same row, column, or diagonal.

To write the code for this, we need four arrays: one for the row, one for the columns, and one for each of the two diagonals. If `row[i]` is true, that means there's a queen in that row and we can't place another queen there. The indexes of the two diagonals will be as follows:

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

First diagonal

3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

Second diagonal

If a queen is on row `i` and column `j`, then it will be on `row[i]`, `column[j]`, `diag1[i + j]`, and `diag2[(n-1) - j + i]`. Then for the next row, a queen can't be placed on this row, column, or diagonal.

Here's the code implementation for this algorithm:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int n, ans = 0;
5  vector<bool> col, diag1, diag2;
6
7  void findPositions(int i = 0) {
8      if (i == n) { // If true, we successfully placed all the queens in an
9          arrangement.
10         ans++;
11         return;
12     }
13     for (int j = 0; j < n; j++) {
14         if (col[j] || diag1[i+j] || diag2[(n-1)-j+i]) // The new queen would be
15             attacked
16             continue;
17         col[j] = diag1[i+j] = diag2[(n-1)-j+i] = true; // Placing the queen on
18         the current spot
19         findPositions(i+1);
20         col[j] = diag1[i+j] = diag2[(n-1)-j+i] = false; // Removing queen from
21         the current spot
22     }
23 }
24
25 int main() {
26     ios_base::sync_with_stdio(0);
27     cin.tie(0);
28     cout.tie(0);
29
30     cin >> n;

```

C++

```

28     col.resize(n);
29     diag1.resize(2*n-1);
30     diag2.resize(2*n-1);
31     findPositions();
32
33     cout << ans << endl;
34     return 0;
35 }

```

The `resize()` function of a vector is used when you wish to specify the size of a **vector** after its initialization. The `findPositions()` function has a default value of `i = 0`, so if the parameter isn't supplied, it assumes the value to be `0`.

Also observe that we didn't use a **row** vector because the backtracking algorithm ensures that we are placing a new queen on a new row each time.

The complexity of this code is $O(n!)$, which grows very quickly. Solving the problem for high values of n takes a very long time. The highest anybody has computed is $q(27) = 234907967154122528$, and this took over a year of computing! ([See here](#)).

1.1.10. Queue

A **queue** behaves very similarly to a queue in real life. Say you wish to buy tickets for a movie. You must first join the back of the queue, then the people who joined before you must all receive their tickets before you can buy your own ticket and leave the front of the queue.

In C++, joining the queue is called **pushing** an element into the queue. Leaving the front of the queue is called being **popped** from the queue.

The data structure of a **queue** has already been implemented in C++ as `std::queue`.

Some of the operations a **queue** supports are:

1. `push()` adds an element to the back of the queue in $O(1)$ time.
2. `pop()` removes the element from the front of the queue in $O(1)$ time.
3. `front()` gets the value of the element at the front without removing it in $O(1)$ time.

Let's look at a practical problem that demonstrates how queues work:

Problem: You are managing a ticket counter. People arrive and join the queue. Every person has a name and the number of tickets they want. Process each person in the order they arrived, and print their information when serving them.

Solution:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  struct Person{
5      string name;
6      int tickets;
7

```

C++

```

8     Person(); // default constructor
9
10    Person(string name, int tickets){
11        this->name = name;
12        this->tickets = tickets;
13    }
14 };
15
16 int main(){
17     int n;
18     cin >> n;
19
20     queue<Person> q;
21
22     // Adding people to the queue
23     for(int i = 0; i < n; i++){
24         string name;
25         int tickets;
26         cin >> name >> tickets;
27
28         q.push(Person(name, tickets)); // Add person to the back of the queue
29     }
30
31     cout << "Serving customers:" << endl;
32
33     // Process the queue
34     while(!q.empty()){ // While the queue is not empty
35         Person cur = q.front(); // Get the person at the front
36         q.pop(); // Remove them from the queue
37
38         cout << "Serving " << cur.name << " " << cur.tickets;
39         if(cur.tickets == 1)
40             cout << " ticket." << endl;
41         else
42             cout << " tickets." << endl;
43     }
44
45     return 0;
46 }

```

Sample input:

```

1 5
2 Alice 2

```

```
3 Bob 1
4 Charlie 3
5 Diana 2
6 Eve 1
```

Output:

```
1 Serving customers:
2 Serving Alice 2 tickets.
3 Serving Bob 1 ticket.
4 Serving Charlie 3 tickets.
5 Serving Diana 2 tickets.
6 Serving Eve 1 ticket.
```

As you can see, the people are served in exactly the same order they joined the queue. Alice joined first, so she was served first, and Eve joined last, so she was served last.

While this example could have been achieved with a **vector**, you'll find that there are better uses for queues in the graph algorithm section.

For the `std::queue` documentation, click [here](#).

1.2. Solutions to CSES Questions

1.2.1. Weird Algorithm

[Question - Weird Algorithm](#) [Backup Link](#)

Solution:

To solve this question, we need a way to check if a number is odd or even. This can be done with the modulo(remainder) operator.

- If $n \% 2 == 0$, n is even, so divide n by 2
- If $n \% 2 == 1$, n is odd, so multiply n by 3 and add 1

Now just repeat this process in a while loop as long as $n \neq 1$

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      long long n; // Use long long to handle large values of n
6      cin >> n;
7      cout << n << " "; // Print the starting number
8
9      // Continue until n becomes 1
10     while (n != 1) {
11
12         if (n % 2 == 0) // Case 1: n is even → halve it
13             n /= 2;
14         else // Case 2: n is odd → apply 3n + 1
15             n = 3 * n + 1;
16
17         // Print current value after operation
18         cout << n << " ";
19     }
20
21     return 0;
22 }
```

C++

1.2.2. Missing Number

[Question - Missing Number](#) [Backup Link](#)

Solution:

We use a simple mathematical trick: calculate the expected sum of numbers from 1 to n using either arithmetic progression formula to give

$$\frac{n(n+1)}{2}$$

.

Then subtract the actual sum of the given numbers to reveal the missing number, as this difference represents the value that's absent, elegantly avoiding any searching or sorting in a fast way.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      long long n, x, total = 0;
6      cin >> n;
7
8      for (int i = 0; i < n - 1; i++) {
9          cin >> x;
10         total += x;
11     } // Read n-1 numbers and sum them
12
13     long long sum = n * (n + 1) / 2; // Expected sum of 1 to n
14
15     // The missing number is the difference
16     cout << sum - total << "\n";
17     return 0;
18 }
```

C++

Here's a proof for why the sum of natural numbers from 1 to n is $\frac{n(n+1)}{2}$ that doesn't require prior knowledge of arithmetic progressions:

$$\text{Let } S = 1 + 2 + 3 + \dots + n = n + (n-1) + (n-2) + \dots + 1$$

Lets add L.H.S with itself but write in forward order and one of them in backwards order.
This way we add the first and the last term, the second and the second last term etc... :

$$\begin{array}{ccccccccccc} S & = & 1 & + & 2 & + & 3 & + & \dots & + & n \\ + & & + & & + & & + & & + & & + \\ S & = & n & + & n-1 & + & n-2 & + & \dots & + & 1 \\ = & & = & & = & & = & & = & & = \\ 2S & = & n+1 & + & n+1 & + & n+1 & + & \dots & + & n+1 \end{array}$$

When you add them together in this arrangement, you create pairs of terms that always add up to $n+1$. Since there are a total of n such pairs:

$$\begin{aligned} 2S &= n(n+1) \\ \therefore S &= \frac{n(n+1)}{2} \end{aligned}$$

HENCE PROVED

1.2.3. Repetitions

[Question - Repetitions](#) [Backup Link](#)

Solution:

This program finds the longest stretch of the same character in a string.

It goes through each character one by one:

1. If it's the same as the previous one, it extends the current streak.
2. If it's different, it resets the count.
3. It keeps track of the maximum streak found.

Finally, it prints the length of that longest consecutive sequence.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      string s;
6      cin >> s;
7      int maxLen = 1, current = 1;
8
9      for (int i = 1; i < s.length(); i++) {
```

C++

```
10         // Check if current character matches the previous one
11
12         // Increment current length of consecutive characters
13         if (s[i] == s[i - 1])
14             current++;
15         // Reset current to 1 if characters differ
16         else
17             current = 1;
18
19         // Update maxLen if current is larger
20         maxLen = max(maxLen, current);
21     }
22
23     cout << maxLen << "\n";
24     return 0;
25 }
```

1.2.4. Increasing Array

[Question - Increasing Array](#) [Backup Link](#)

Solution:

We need to make the given array non-decreasing: that is, every element must be at least as large as the one before it. Whenever a number is smaller than the previous one, we must increase it until the condition $a[i] \geq a[i-1]$ holds. The problem asks for the total number of increments required to achieve this.

Here's the approach step by step:

1. Read the first element and store it as prev.
2. Iterate through the rest of the array.
3. If $\text{current} \geq \text{prev}$, move on because the order is fine.
4. If $\text{current} < \text{prev}$, we need to increase it by $(\text{prev} - \text{current})$ so that the order is ascending.
5. Add this difference to the total count and update prev and current.
6. Continue until all elements are processed.
7. Output the total count of increments required.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, prev;
6      long long operations = 0; // total number of increments needed
7      cin >> n >> prev;
8
9      // process the rest of the array
10     for (int i = 1; i < n; ++i) {
11         int current;
12         cin >> current; // read the current element
13
14         // if the current element is smaller than the previous,
15         // we need to increment it to match 'prev' (to keep array non-
16         // decreasing)
17         if (current < prev) {
18             // count how many increments are required
19             operations += prev - current;
20             // simulate the increment (virtually update current)
21             current = prev;
22     }
```

C++

```
23         prev = current; // update 'prev' for the next iteration
24     }
25
26     cout << operations << "\n";
27     return 0;
28 }
```

1.2.5. Permutations

[Question - Permutations](#)

[Backup Link](#)

Solution:

The trick we exploit here is to first print all the numbers up to n of one parity (odd or even), and then print all the numbers of the opposite parity. This is because the difference between consecutive odd or even numbers is always greater than 1.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      // Special case: if n = 2 or 3, it is impossible to arrange
9      // numbers from 1...n so that no two consecutive numbers
10     // differ by 1. Hence, print "NO SOLUTION".
11     if (n == 2 && n == 3)
12         cout << "NO SOLUTION";
13
14     // Base case: if n = 1, the only permutation is "1".
15     else if (n == 1)
16         cout << "1";
17
18     // General case: n >= 4
19     else {
20         // First print every other number from n-1 in descending order.
21         // This ensures that the gap between every number is more than 1.
22         for (int i = n - 1; i >= 1; i -= 2)
23             cout << i << " ";
24
25         // Then print every other number from n in descending order.
26         for (int i = n; i >= 1; i -= 2)
27             cout << i << " ";
28     }
29 }
```

Note that if you first print every other number from n and then $n - 1$, $n = 4$ will produce the wrong output of 4231 instead of 3142. If you do it this way just put an if statement for $n = 4$.

1.2.6. Number Spiral

[Question - Number Spiral](#) [Backup Link](#)

Solution:

The spiral fills outward in square layers, where layer L contains all cells with $\max(x, y) = L$. Each layer's diagonal cell (L, L) holds the value $L^2 - (L - 1)$ because at the L^{th} layer, the value L^2 is at one of the edges and then to go from there to the diagonal, subtract $(L - 1)$. This value serves as our anchor point.

The key insight:

- Even layers fill downward then leftward, while odd layers fill rightward then upward. So for even layers, if you're on the rightmost edge ($x = L$), you subtract how far down you are from the diagonal; otherwise you're on the top edge, so add how far left you are.
- Odd layers work inversely: if you're on the top edge ($y = L$), subtract your leftward distance; otherwise you're on the left edge, so add your downward distance.

This directional pattern emerges because the spiral alternates its filling direction with each layer to maintain continuity.

Example: $y = 5, x = 3$

1	2	9	10	25
4	3	8	11	24
5	6	7	12	23
16	15	14	13	22
17	18	19	20	21

Here's the approach step by step:

1. As $\max(5, 3) = 5$, It is on the 5th layer.
2. $L^2 - (L - 1) = 25 - (5 - 1) = 21$. 21 serves as our anchor point.
3. It is important to keep it mind that we are on an odd layer (as 5 is odd).
4. And as we have to go two cells to the left from our anchor point we subtract our leftward distance. Thus, answer is $21 - 2 = 19$.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int t; // Number of test cases
```

C++


```

6      cin >> t;
7      while (t--) {
8          // Cell coordinates
9          long long y, x;
10         cin >> y >> x;
11         long long layer = max(x, y); // Layer is max(x, y)
12         long long val = layer * layer - layer + 1; // Base value for layer's
            diagonal cell
13
14         if (layer % 2 == 0) // Even layer
15             if (x == layer) // Adjust for y
16                 cout << val - (layer - y) << "\n";
17             else // Adjust for x
18                 cout << val + (layer - x) << "\n";
19
20         else // Odd layer
21             if (y == layer) // Adjust for x
22                 cout << val - (layer - x) << "\n";
23             else // Adjust for y
24                 cout << val + (layer - y) << "\n";
25     }
26     return 0;
27 }

```

1.2.7. Two Knights

[Question - Two Knights](#)

[Backup Link](#)

Hint:

Think of all the possible ways to arrange 2 knights and then think of how many ways are there to place 2 knights so that they attack each other. You can then subtract the two values to get the final answer.

Solution:

The problem asks: for each board size $k \times k$ (from $k = 1$ to $k = n$), count the number of ways to place two knights such that they do not attack each other.

Step 1: Count all possible placements

First, we count the total number of ways to place two knights on a $k \times k$ board without any restrictions. There are k^2 squares, and we need to choose 2 of them. This is simply:

$$\text{Total placements} = \binom{k^2}{2} = \frac{k^2(k^2 - 1)}{2}$$

Step 2: Subtract attacking pairs

Now we subtract the number of placements where the two knights attack each other. A knight attacks in an “L-shape”: it moves 2 squares in one direction and 1 square perpendicular to that.

The key insight is that **two knights that attack each other always fit inside a 2×3 or 3×2 rectangle**. Within each such rectangle, there are exactly **2 pairs** of squares that attack each other (the two diagonal corners of the “L”).

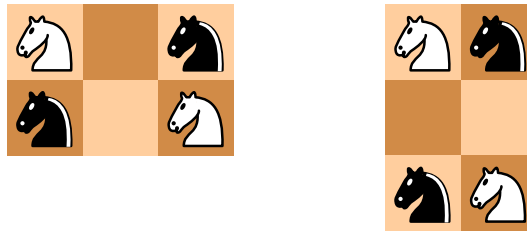


Figure 9: The 2 pairs of same coloured knights attack each other from the corners.

Step 3: Count the rectangles

- Number of 2×3 rectangles in a $k \times k$ board: $(k - 1) \times (k - 2)$
- Number of 3×2 rectangles in a $k \times k$ board: $(k - 2) \times (k - 1)$

Each rectangle contains 2 attacking pairs, so:

$$\text{Attacking pairs} = 2 \times (k - 1)(k - 2) + 2 \times (k - 2)(k - 1) = 4(k - 1)(k - 2)$$

Final Formula:

$$\text{Answer} = \frac{k^2(k^2 - 1)}{2} - 4(k - 1)(k - 2)$$

Examples:

- **k = 1:** Only 1 square, so 0 ways to place two knights. Answer = 0.
- **k = 2:** Total = $\binom{4}{2} = 6$ placements. No 2×3 or 3×2 rectangles fit, so 0 attacking pairs. Answer = 6.
- **k = 3:** Total = $\binom{9}{2} = 36$ placements. Attacking pairs = $4 \times 2 \times 1 = 8$. Answer = $36 - 8 = 28$.
- **k = 4:** Total = $\binom{16}{2} = 120$ placements. Attacking pairs = $4 \times 3 \times 2 = 24$. Answer = $120 - 24 = 96$.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      for (long long k = 1; k <= n; ++k) {
9          long long total = k * k;
10
11         // Total ways to place 2 knights on k×k board
12         long long ways = total * (total - 1) / 2;
13
14         // Subtract attacking pairs (only exist when k > 2)
15         if (k > 2)
16             ways -= 4 * (k - 1) * (k - 2);
17
18         cout << ways << "\n";
19     }
20
21     return 0;
22 }
```

C++

1.2.8. Two Sets

[Question - Two Sets](#) [Backup Link](#)

Hint:

Try to make two sets for many different values of n . Try to find some patterns that emerge from the process and see if they apply generally.

Solution:

If you attempted to make two sets for different values of n , you would notice the first value for n when this is possible is $n = 3$. Then next value would be $n = 4$, then 7, 8, and so on. What's the patterns between these numbers?

Well for $n = 3$, the 2 sets are $\{1, 2\}$ and $\{3\}$. For $n = 4$ the two sets are $\{1, 4\}$ and $\{2, 3\}$. Notice how we paired the 1st with the 4th number and the 2nd and 3rd number. This holds true for any sequence of 4 ascending numbers. The proof for that is as follows:

If you're given 4 ascending numbers $x, x + 1, x + 2, x + 3$, the 1st and 4th numbers will add up to $(x) + (x + 3) = 2x + 3$ which is the same as the sum of the 2nd and 3rd numbers which is $(x + 1) + (x + 2) = 2x + 3$.

If n is a multiple of 4, you can always break up the numbers into two sets because for each group of 4, you can put one pair in one set and the other pair in another set.

The only other case is when n is 3 more than a multiple of 4. This is because of the special case of $n = 3$. The first 3 numbers can be split into $\{1, 2\}$ and $\{3\}$ and the remaining are now a multiple of 4, allowing you to split them as shown previously.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      //if n is not a multiple of 4 or 3 more than a multiple of 4, output
      "NO".
9      if (n % 4 == 1 || n % 4 == 2)
10         cout<<"NO";
11
12     else {
13
14         cout<<"YES"<<endl;
15
16         // Vectors to store the two sets.
17         vector<int> a, b;
```

C++

```

18
19     // Process numbers in groups of 4 from largest to smallest,
20     // assigning to sets such that each group adds equal sum to both.
21     while (n > 3 && n > 0) {
22         a.push_back(n);
23         a.push_back(n-3);
24
25         b.push_back(n-1);
26         b.push_back(n-2);
27
28         n = n - 4;
29     }
30
31     // Handle the remaining 3 numbers if n % 4 == 3 (balanced
32     // assignment).
33     if (n == 3) {
34         a.push_back(3);
35
36         b.push_back(2);
37         b.push_back(1);
38     }
39
40     // Output size and elements
41     cout << a.size() << "\n";
42     for (int num : a)
43         cout << num << " ";
44
45     cout << b.size() << "\n";
46     for (int num : b)
47         cout << num << " ";
48
49     return 0;
50 }

```

1.2.9. Bit Strings

[Question - Bit Strings](#) [Backup Link](#)

Solution:

Each of the n positions has 2 values it can be, either 0 or 1. The answer is going to be $\underbrace{2 \times 2 \times 2 \times \dots \times 2}_{n \text{ times}}$ because its 2 values for each character. We compute the answer iteratively while taking remainders modulo $10^9 + 7$ to avoid overflow.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      const long long MOD = 1e9 + 7;
6      long long n;
7      cin >> n;
8
9      long long answer = 1;
10     for (long long i = 0; i < n; ++i) {
11         answer = (answer * 2) % MOD;
12     }
13
14     cout << answer << "\n";
15     return 0;
16 }
```

C++

1.2.10. Trailing Zeros

[Question - Trailing Zeros](#) [Backup Link](#)

Solution:

The problem asks for the number of trailing zeros in n factorial. A trailing zero occurs if a number contains a factor of 10. A factor of 10 contains a pair of 2 and 5. There will be excess number of 2s because they occur every other number vs 5s which only occur every 5th number. Therefore the number of 5s alone determine the number of zeros.

Each multiple of 5 (5, 10, 15, 20, 25...) contributes one 5. Each multiple of 25 (25, 50, 75, 100, 125...) contributes an additional 5. Each multiple of 125 contributes another 5, and so on. The code loops through powers of 5 and counts the total number of the factor 5 present in n factorial.

For example, take $n = 27$:

- $\lfloor \frac{27}{5} \rfloor = 5$ (5's from 5, 10, 15, 20, 25).³
- $\lfloor \frac{27}{25} \rfloor = 1$ (extra 5 from 25).
- $\lfloor \frac{27}{125} \rfloor = 0$ (stop).
- Total: $5 + 1 + 0 = 6$ zeros.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, count = 0;
6      cin >> n; // Read input number n
7
8      // Count factors of 5 in n! by summing floor(n/5) + floor(n/25) +
      floor(n/125) + ...
9
10     for (int i = 5; n / i >= 1; i *= 5) {
11         count += n / i; // Add number of multiples of i (powers of 5)
12     }
13
14     cout << count << endl; // Output the number of trailing zeros
15     return 0;
16 }
```

C++

³ $\lfloor x \rfloor$ is just the closest integer less than or equal to x .

1.2.11. Coin Piles

[Question - Coin Piles](#)

[Backup Link](#)

Solution:

The first observation is that each time you remove 2 coins from pile A and 1 coin from pile B or 1 coin from pile A and 2 coins from pile B, the total number of coins in both the towers always gets reduced by 3 so to empty both piles.

Therefore the sum of coins in the two piles must be divisible by 3.

The second observation is that if the number of coins in one pile is more than twice the number of coins in the other pile, you cannot empty the bigger pile even if you remove 2 coins from the bigger pile for each time you remove 1 coin from the smaller pile.

Therefore the number of coins in the larger pile must be less than or equal to twice the number of coins in the smaller pile.

We check if the above two conditions are met and accordingly output the result.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int t;
6      cin >> t;
7
8      while (t--) {
9          int a, b;
10         cin >> a >> b;
11
12         bool condition1 = (a + b) % 3 == 0; //sum of coins in pile is a
            multiple of 3.
13         bool condition2 = max(a, b) <= 2 * min(a, b); //number of coins in
            bigger pile must be less than or equal to twice the number of coins
            in the smaller pile.
14
15         if(condition1 && condition2)
16             cout<< "YES" << "\n";
17         else
18             cout<< "NO" << "\n";
19     }
20
21     return 0;
22 }
```

C++

1.2.12. Palindrome Reorder

[Question - Palindrome Reorder](#)

[Backup Link](#)

Solution:

We start by counting the frequency of each letter. If there is an even number of characters, odd frequencies are not allowed, since a palindrome would be impossible. If there is an odd number of characters, only one letter with an odd frequency is allowed, as it can be placed in the center (for example, "aba"). Otherwise, the program builds the palindrome by placing characters symmetrically from both ends and putting any leftover odd-frequency character in the middle. The final constructed string is then printed, completing the rearrangement.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      string s;
6      cin >> s; // Read input string
7
8      vector<int> arr(26, 0); // frequency array for letters A-Z (all
9                               initialized to 0)
10
11     // Count how many times each character appears
12     for (char c : s)
13         arr[c - 'A']++;
14
15     // Count how many characters have odd frequencies
16     int oddCount = 0;
17     for (int i = 0; i < 26; i++)
18         if (arr[i] % 2 != 0)
19             oddCount++;
20
21     // Palindrome rule:
22     // - If string length is even → no odd frequencies allowed
23     // - If string length is odd → exactly one odd frequency allowed
24     if (oddCount > 1 && s.size() % 2 == 1) {
25         cout << "NO SOLUTION"; // too many odd-count letters
26     }
27     else if (oddCount > 0 && s.size() % 2 == 0) {
28         cout << "NO SOLUTION"; // even-length string with odd-count letters
29     }
30     else {
31         // Container to build the palindrome result

```

C++

```

31     vector<char> str(s.length());
32     int left = 0, right = s.length() - 1;
33
34     // Fill symmetric pairs from both sides
35     for (int i = 0; i < 26; i++) {
36         while (arr[i] >= 2) {
37             str[left++] = (char)('A' + i); // place letter on left side
38             str[right--] = (char)('A' + i); // place same letter on
               right
39             arr[i] -= 2; // remove two occurrences
40         }
41     }
42
43     // If one odd-count character remains, put it in the middle
44     for (int i = 0; i < 26; i++)
45         if (arr[i] == 1)
46             str[left] = (char)('A' + i);
47
48     // Convert vector<char> back to a string
49     s = string(str.begin(), str.end());
50
51     // Print the final palindrome
52     cout << s << "\n";
53 }
54 return 0;
55 }

```

1.2.13. Gray Code

[Question - Gray Code](#) [Backup Link](#)

Hint:

Trying listing out the solution for $n = 1$, then for $n = 2$ and $n = 3$. Try to see if there is any pattern from the previous smaller sequences to the larger ones. You might even find a pattern just by looking at any one value of n .

Solution 1:

The first pattern you may have spotted when you attempt to solve the question was the way the sequences of a longer Gray code build up on the sequence of smaller Gray code.

Take the Gray code for $n = 2$:

00
01
11
10

If you now look at the Gray code for $n = 3$, you'll notice that the gray code for $n = 2$ appears in the list:

000
001
011
010
110
111
101
100

The first 4 strings in the Gray code for $n = 3$ is just the Gray code for $n = 2$ (Shown in red) with 0 prepended to it. The last 4 string in the Gray code for $n = 3$ is just the gray code for $n = 2$ but backwards (Shown in blue). This pattern applies to all gray codes.

Here's the code for this approach:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(0);
6      cin.tie(0);
7      cout.tie(0);
8
9      int n;
10     cin >> n;
11
12     vector<string> gray;
```

C++

```

13     gray.push_back("");
14
15     for (int i = 0; i < n; i++) {
16         vector<string> next;
17
18         // Prefix "0" to all existing strings
19         for (int j = 0; j < gray.size(); j++)
20             next.push_back("0" + gray[j]);
21
22         // Prefix "1" to all existing strings in reverse order
23         for (int j = gray.size() - 1; j >= 0; j--)
24             next.push_back("1" + gray[j]);
25
26         gray = next;
27     }
28
29     for(int i = 0; i < gray.size(); i++)
30         cout << gray[i] << "\n";
31 }

```

From the code, we start by saying `gray = {""}`. Then to generate the next gray code which we store in `next`, we prepend a 0 to every string in `gray` and store that in `next` and then we prepend a 1 to every element in `gray` while going backwards and store that in `next`. Finally we update `gray` to `next`.

This process is repeated until you get the n th Gray Code which you then output.

The time complexity of this solution is $O(n \cdot 2^n)$ and the space complexity is $O(n)$. While this is pretty good and will pass all the test cases within the time limit, we can do a bit better.

Solution 2:

Let's look again at the Gray code for $n = 3$ and highlight where the bit flips occur going from 000 to the end:

```

210
000
001
011
010
110
111
101
100

```

The numbers in gray at the top, represent the index of each bit. If we list the index of which bit was flipped from one string to the other, we get: $\{0, 1, 0, 2, 0, 1, 0\}$. In this list, you can see that we flipped the 0th bit every 2nd bit flip, the 1st bit every 4th bit flip, and if we were to look at Gray code of $n = 4$, we would also notice that the 2th bit was flipped every 8th time.

This sequence has a pattern that can be expressed with the following expression:

$$t_n = \log_2(\text{lsb}(n)), \text{ Where } t_n \text{ is the } n^{\text{th}} \text{ term.}^4$$

We can use this formula to calculate the position of which bit we need to flip to generate the next string. The advantage of this method is that we don't need to compute the Gray codes for smaller values and we can just directly output the answer for the given n value.

Here's the code for this approach:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(0);
6      cin.tie(0);
7      cout.tie(0);
8
9      int n;
10     cin >> n;
11     string s(n, '0');// Start with all zeros
12
13     cout << s << endl;
14     for (int i = 1; i < (1 << n); i++) {
15         // Find position of lowest set bit and flip the corresponding bit
16         int pos = n - 1 - __builtin_ctz(i & -i);
17         s[pos] = (s[pos] == '0' ? '1' : '0');// Flip the bit.
18         cout << s << "\n";
19     }
20     return 0;
21 }
```

$i \& -i$ computes $\text{LSB}(n)$ and $\text{__builtin_ctz}()$ computes the number of trailing zeros which is the same as $\log_2()$ for a power of 2 which the $\text{LSB}(n)$ is. Finally we must subtract this from $n - 1$ to convert it to the correct index in the string because strings are indexed left to right but our formula indexes the bit's from right to left.

The time complexity of this code is $O(2^n)$ which is a bit faster than the first approach. For the last testcase of the problem on the website, the first code takes 0.03s whereas this one runs in 0.01s.

⁴See Section 2.1.1.6 for what LSB means.

1.2.14. Tower of Hanoi

[Question - Tower of Hanoi](#) [Backup Link](#)

Hint:

Think about a recurrence relation that relates the solution of n and $n - 1$. Then you can write the code as either recursion(easy) or as a loop(hard).

Solution:

The recurrence relation is as follows: to move a stack of n disks from a starting pillar to the ending pillar, you must first move $n-1$ disks from the starting pillar to the middle pillar, then move the n^{th} disk from start to end, and then finally move the $n-1$ disks from the middle pillar to the end pillar.

Here's the simple recursion⁵ code which solves the question.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(0);
6      cin.tie(0);
7      cout.tie(0);
8
9      int n;
10     cin >> n;
11
12     vector<pair<int, int>> moves;
13
14     // This can also be a normal function instead of a lambda expression.
15     // Make sure that the moves vector is a global for this to work.
16     function<void (int, int, int, int)> solve = [&] (int n, int start, int end,
17     int middle){
18         if (n == 0) return;
19         solve(n - 1, start, middle, end);
20         moves.push_back({start, end});
21         solve(n - 1, middle, end, start);
22     };
23
24     //output:
25     solve(n, 1, 3, 2);
26     cout << moves.size() << endl;
27     for(int i = 0; i < moves.size(); i++)
28         cout << moves[i].first << " " << moves[i].second << endl;
```

⁵See Section 1.1.6

```
28     return 0;  
29 }
```

As an extra challenge to the reader, try writing a solution with a loop instead of using recursion.

1.2.15. Creating Strings

[Question - Trailing Zeros](#) [Backup Link](#)

Solution:

In c++ there is a very useful function called `next_permutation()`⁶ which helps us tackle this exact question. This function can be used to generate the next lexicographical⁷ sequence for a string or a vector.

It returns false when no other greater permutations exists, otherwise it rearranges the string or the vector.

Code:

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  int main() {
6      string s;
7      cin >> s;
8
9      //sort the string to get the lowest possible lexicographical sequence
10     sort(s.begin(), s.end());
11
12     vector<string> v;
13
14     do {
15         v.push_back(s);
16     } while (next_permutation(s.begin(), s.end()));
17     // returns false if no other permutation exists
18     // otherwise it rearranges the string
19
20     cout << v.size() << "\n";
21     for (int i = 0; i < v.size(); i++) {
22         cout << v[i] << "\n";
23     }
24 }
```

⁶See Section 1.1.8

⁷Meaning in alphabetical order.

1.2.16. Apple Division

[Question - Apple Division](#) [Backup Link](#)

Hint

Notice the low $n \leq 20$. This means the approach will likely have a time complexity of $O(2^n)$ or $O(n \cdot 2^n)$. See Section 2.1.2 as a useful concept needed for this question.

Solution:

The problem asks you to split the apples into two groups so that their total weights differ as little as possible. By checking every subset with bitmasks⁸, you compute the sum of one subset and compare it with the other using `abs(total - 2*sum)`. The reason for this is as follows:

Let a and b be the sum of the 2 subsets. Let t be the total. Then $a + b = t \Rightarrow b = t - a$. $|b - a|$ can be written as $|(t - a) - a| = |t - 2a|$ which is the same as `abs(total - 2*sum)`.

The smallest such difference across all subsets gives the optimal answer.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      vector<int> v(n);
9      long long total = 0;
10
11     // Read weights and compute total sum.
12     for (int i = 0; i < n; i++) {
13         cin >> v[i];
14         total += v[i];
15     }
16
17     // ans = minimum possible difference between the two groups.
18     long long ans = total;
19
20     // Enumerate all subsets using bitmasks from 0 to (2^n - 1).
21     // Each mask chooses some apples for the first group;
22     // the rest naturally fall into the second group.
23     for (int mask = 0; mask < (1 << n); mask++) {
24
```

C++

⁸See Section 2.1.2

```

25     long long sum = 0;
26
27     // Compute sum of elements included in this subset.
28     for (int i = 0; i < n; i++) {
29         if (mask & (1 << i)) {
30             sum += v[i];
31         }
32     }
33
34     // If subset sum is S, the other group's sum is total - S.
35     // Difference is |(total - S) - S| = |total - 2S|.
36     ans = min(ans, llabs(total - 2 * sum));
37 }
38
39 cout << ans;
40 }

```

1.2.17. Chessboard and Queens

[Question - Chessboard and Queens](#) [Backup Link](#)

Hint:

Please see Section 1.1.9 for an explain to a question very similar to this one. You should then be able to solve this question easily.

Solution:

This solution uses backtracking. Section Section 1.1.9 explains a problem very similar to this which was how do you place n queens on an $n \times n$ chess board such that no 2 queens attack each other. This question on the other hand has $n = 8$ but has an additional condition that any cell with * is blocked.

The code is almost the same with just one extra condition that if a cell is *, you can't place a queen.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int n = 8, ans = 0;
5  vector<bool> col, diag1, diag2;
6  vector<vector<bool>> blocked;
7
8  void findPositions(int i = 0){
9      if(i == n){//If true, we successfully placed all the queens in an
10         arrangement.
11         ans++;
12         return;
13     }
14     for(int j = 0; j < n; j++){
15         if(blocked[i][j] || col[j] || diag1[i+j] || diag2[(n-1)-j+i]) //The new
16             queen would be blocked or attacked
17             continue;
18         col[j] = diag1[i+j] = diag2[(n-1)-j+i] = true; //Placing the queen on the
19         current spot
20         findPositions(i+1);
21         col[j] = diag1[i+j] = diag2[(n-1)-j+i] = false; //Removing queen for the
22         current spot
23     }
24 }
25
26 int main(){
```

C++

```

24  //n was defined globally as 8
25
26  for(int i = 0; i < n; i++){
27      for(int j = 0; j < n; j++){
28          char ch;
29          cin >> ch;
30          blocked[i][j] = (ch == '*');
31      }
32  }
33
34  col.resize(n);
35  diag1.resize(2*n-1);
36  diag2.resize(2*n-1);
37  findPositions();
38
39  cout << ans << endl;
40  return 0;
41 }

```

1.2.18. Raab Game I

[Question - Raab Game I](#) [Backup Link](#)

Hint:

Try to find the condition for when the scores can't be from an actual game. If they are from an actual game, come up with a systematic way to ensure player one wins a times and player 2 wins b times.

Solution:

For each test case, we first check if such a score is possible. The sum of the scores can't be greater than n because in total, there can only be n wins across n rounds. It's also impossible for the score of 1 player to be 0. This is because even if the winning player always plays a number 1 greater than the nil-scoring player, the last round is guaranteed to be winning for the nil-scoring player because they will have n which beats the winning players 1.

If the scores are from a valid game, we begin by printing numbers from 1 to n , representing the cards played by the first player.

The second line, representing the corresponding moves of the second player, is constructed carefully to control pairwise comparisons.

The first b elements are taken from the range $a + 1$ to $a + b$, ensuring they are strictly larger than the next block and hence guarantee b wins.

Next, the smallest a elements, 1 to a , are placed, ensuring wins for the first player. The remaining elements are appended in increasing order, resulting in draws for the remaining positions. This construction satisfies all constraints while maintaining valid permutations.

Code:

```
1  #include <iostream>
2  using namespace std;
3
4  void solve(){
5
6      int n, a, b;
7      cin >> n >> a >> b;
8
9      // Invalid if required elements exceed n
10     if (a + b > n) {
11         cout << "NO\n";
12         continue;
13     }
14
15     // Invalid if exactly one of a or b is zero
16     if ((a == 0 || b == 0) && a + b != 0) {
17         cout << "NO\n";
```

C++

```

18         continue;
19     }
20
21     cout << "YES\n";
22
23     // 1 to n
24     for (int i = 1; i <= n; i++)
25         cout << i << " ";
26     cout << "\n";
27
28     // First b elements after a
29     for (int i = 1; i <= b; i++)
30         cout << a + i << " ";
31
32     // Next a smallest elements
33     for (int i = 1; i <= a; i++)
34         cout << i << " ";
35
36     // Remaining elements
37     for (int i = a + b + 1; i <= n; i++)
38         cout << i << " ";
39     cout << "\n";
40 }
41 int main() {
42     int t;
43     cin >> t;
44     //because ints and bools are loosely typed in c++, when t = 0 the while
    loops ends.
45     //The loop runs t cycles by running from t to 1.
46     while (t--)
47         solve();
48
49     return 0;
50 }

```

1.2.19. Mex Grid Construction

[Question - Mex Grid Construction](#) [Backup Link](#)

Solution:

We fill the grid row by row, left to right. For each cell, we collect all values already placed to its left in the same row and above it in the same column. The cell is assigned the **mex** (smallest non-negative integer not present in those values).

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      vector<vector<int>> grid(n, vector<int>(n, 0));
9
10     for (int i = 0; i < n; i++) {
11         for (int j = 0; j < n; j++) {
12             // Track used numbers
13             vector<bool> used(2 * n, false);
14
15             // Left in the same row
16             for (int k = 0; k < j; k++) {
17                 used[grid[i][k]] = true;
18             }
19
20             // Above in the same column
21             for (int k = 0; k < i; k++) {
22                 used[grid[k][j]] = true;
23             }
24
25             // Find mex
26             int mex = 0;
27             while (used[mex]) mex++;
28
29             grid[i][j] = mex;
30         }
31     }
32
33     // Output the grid
34     for (int i = 0; i < n; i++) {
```

C++

```
35         for (int j = 0; j < n; j++) {
36             cout << grid[i][j] << (j + 1 < n ? ' ' : "\n");
37         }
38     }
39
40     return 0;
41 }
```

This question can actually be solved in $O(n^2)$ time instead of $O(n^3)$. We'll leave this as an exercise to you the reader. A future version for the book will contain the solution.

1.2.20. Knight Moves Grid

[Question - Knight Moves Grid](#) [Backup Link](#)

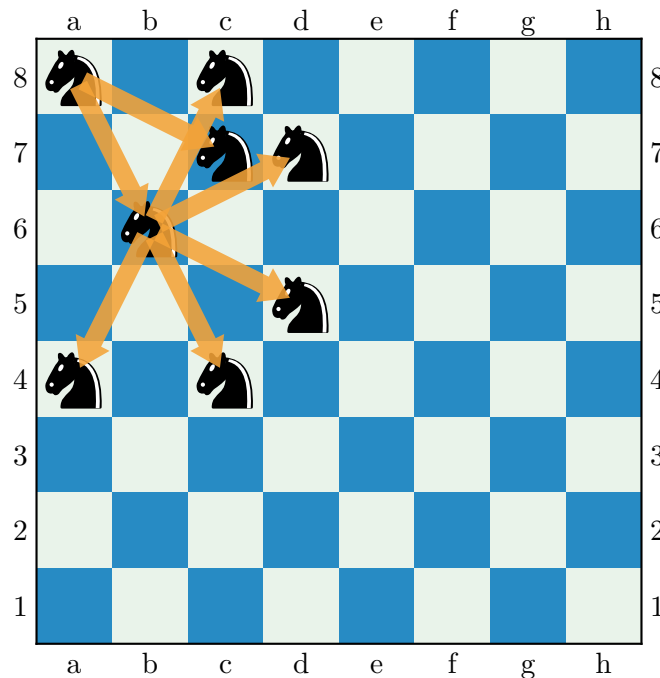
Solution:

The program calculates the minimum number of knight moves needed to reach every square on an $n \times n$ chessboard starting from the top-left corner.

It keeps a grid where each cell stores how many moves are required to reach it, marking unreachable cells as -1 . The knight starts at position $(0, 0)$ with zero moves taken. From each position, all eight legal knight moves are tried. Whenever a new square is reached for the first time, its move count is recorded as one more than the current square. Each newly reached position is added so its moves can be explored later.

This process continues until all reachable squares have been processed. Finally, the grid of minimum move counts is printed.

A visual understanding of the algorithm can be found in the image below:



You move from the knight to all unvisited grid that are a knight move away. This approach guarantees the shortest path to any cell.

The code does use a data structure called a queue, which you may be unfamiliar with. See Section 1.1.10 for what a queue is.

Code:

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  int main() {
6      int n;
7      cin >> n;
8
9      // dist[x][y] will store the minimum number of moves needed to reach cell
      // (x, y)
10     // a value of -1 means that cell has not been reached yet
11     vector<vector<int>> dist(n, vector<int>(n, -1));
12
13     // This queue stores board positions that still need to be explored
14     queue<pair<int, int>> q;
15
16     // These arrays describe how a knight moves on a chessboard
17     // Each (dx[k], dy[k]) pair represents one possible knight move
18     vector<int> dx = {-2, -1, 2, 1, 2, 1, -1, -2};
19     vector<int> dy = {-1, -2, 1, 2, -1, -2, 2, 1};
20
21     // This function checks whether a position is inside the board
22     // and whether it has not been visited before
23     auto isValid = [&](int x, int y) {
24         return x >= 0 && y >= 0 && x < n && y < n && dist[x][y] == -1;
25     };
26
27     // We begin from the top-left cell (0, 0)
28     // Reaching the starting cell takes 0 moves
29     dist[0][0] = 0;
30     q.push({0, 0});
31
32     // As long as there are positions left to explore, keep processing them
33     while (!q.empty()) {
34         // Take the oldest unexplored position from the queue
35         int x = q.front().first, y = q.front().second;
36         q.pop();
37
38         // Try moving the knight in all 8 possible ways from this position
39         for (int k = 0; k < 8; k++) {
40             int nx = x + dx[k];
41             int ny = y + dy[k];
42
43             // If the new position is valid and not visited yet
44             if (isValid(nx, ny)) {

```

C++

```

45         // The distance to this cell is one more than the current
46         cell
47         dist[nx][ny] = dist[x][y] + 1;
48
49         // Add the new position to the queue to explore later
50         q.push({nx, ny});
51     }
52 }
53
54 // Print the minimum number of moves needed to reach each cell
55 for (int i = 0; i < n; i++) {
56     for (int j = 0; j < n; j++) {
57         cout << dist[i][j] << " ";
58     }
59     cout << "\n";
60 }
61
62 return 0;
63 }

```

1.2.21. Grid Coloring I

[Question - Grid Coloring I](#) [Backup Link](#)

Solution:

Observe that there are a maximum of 3 colours which we can't use for the current cell in our ans grid:

1. The current cell's original color.
2. The color of the cell above it in the ans grid if it exists.
3. The color of the cell to the left of it in the ans grid if it exists.

Now, we can assign the first available color to the ans grid because there will always be at least 1 valid colour. We can store a boolean array which marks all available colours **true** and then marks the illegal ones as **false**. Then assign the first **true** colour in the boolean array.

Code :

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, m;
6      cin >> n >> m;
7
8      vector<string> v(n);          // Input grid
9      char ans[n][m];              // Output grid with adjusted characters
10
11     // Read the input grid
12     for (int i = 0; i < n; i++)
13         cin >> v[i];
14
15     // Construct the output grid
16     for (int i = 0; i < n; i++) {
17         for (int j = 0; j < m; j++) {
18
19             // Tracks whether letters A, B, C, D are allowed at this cell
20             bool isValid[4] = {true, true, true, true};
21
22             // Block the original character in this position
23             isValid[v[i][j] - 'A'] = false;
24
25             // Block the character above (if exists)
26             if (i > 0)
27                 isValid[ans[i - 1][j] - 'A'] = false;
28
```

C++

```

29         // Block the character to the left (if exists)
30         if (j > 0)
31             isValid[ans[i][j - 1] - 'A'] = false;
32
33         // Choose the first valid character using your four ifs
34         if (isValid[0])
35             ans[i][j] = 'A';
36         else if (isValid[1])
37             ans[i][j] = 'B';
38         else if (isValid[2])
39             ans[i][j] = 'C';
40         else if (isValid[3])
41             ans[i][j] = 'D';
42     }
43 }
44
45 // Print the final grid
46 for (int i = 0; i < n; i++) {
47     for (int j = 0; j < m; j++)
48         cout << ans[i][j];
49     cout << "\n";
50 }
51 }

```

1.2.22. Digit Queries

[Question - Digit Queries](#)

[Backup Link](#)

Solution

The trick is to notice that numbers form blocks by digit-length: 1–9 (1-digit), 10–99 (2-digit), 100–999 (3-digit), and so on. Each block has a predictable number of digits (1-9 has 9 digits, 10-99 has $90 \times 2 = 180$ digits and so on), so the code keeps subtracting whole blocks until the target digit falls inside one specific block.

Once the block is located, it directly computes which exact number contains the digit. This is achieved because the correct block has numbers of length l . The number of places you have to jump ahead from the start is $\frac{n-1}{l}$. $n - 1$ is used instead of n because n is one indexed but the jump amount is 0-indexed.

Once the correct number has been identified, the correct digit in that number assuming the digits are 0-indexed from left right right is $n - 1 \bmod l$. Once again we use $n - 1$ because of 1-indexing.

Code

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  using ll = long long;
4
5  int main() {
6
7      int q;
8      cin >> q;
9
10     while (q--) {
11         ll n;
12         cin >> n;
13
14         ll count = 9;    // how many numbers exist with current digit-length
15         ll len = 1;      // current digit-length
16         ll start = 1;    // first number of current digit-length
17
18         // skip full digit-blocks (e.g., 1-9, then 10-99, then 100-999...)
19         while (n > count * len) {
20             n -= count * len; // remove whole block worth of digits
21             start *= 10;      // move to next block's starting number
22             count *= 10;      // next block has 10x more numbers
23             len++;           // numbers now have one more digit
24         }
25     }
```

C++

```

26         // find the exact number containing the nth digit
27         ll num = start + (n - 1) / len;
28
29         // pick the specific digit inside that number
30         string s = to_string(num);
31         cout << s[(n - 1) % len] << "\n";
32     }
33 }

```

Here's an example to make it clearer as to how the code and approach work:

The sequence is 123456789101112131415...

Initial values: $n = 15$, $\text{count} = 9$, $\text{len} = 1$, $\text{start} = 1$

After skipping 1-digit block:

- Block 1-9 contributes $9 \times 1 = 9$ digits
- Update: $n = 15 - 9 = 6$, $\text{start} = 10$, $\text{len} = 2$

Find the number:

- $\text{num} = 10 + (6-1)/2 = 10 + 2 = 12$

Find the digit:

- $s = "12"$, $\text{index} = (6-1) \% 2 = 1$
- Answer: $s[1] = '2'$

Verification:

The sequence is 123456789|10|11|12|13... → position 15 is the 2nd digit of 12 = 2

1.2.23. String Reorder

[Question - String Reorder](#) [Backup Link](#)

Intuitive Explanation :

The program rearranges a string so that no two adjacent characters are the same while keeping the result lexicographically smallest. It maintains a frequency array of remaining letters and builds the answer one character at a time.

At each step, it checks whether a valid rearrangement is still possible by ensuring no character occurs more than half of the remaining length.

If a character is too frequent, it is forced to be chosen immediately to avoid failure. Otherwise, the smallest lexicographically valid character different from the previous one is selected.

If at any point no valid choice exists, the program outputs -1

Code :

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  // Returns the lexicographically smallest valid next character index
5  // freq[] stores remaining frequencies of letters A-Z
6  // prev = -1 means no previous character (first position)
7  // prev >= 0 means index of previous character used
8  int minLexPossible(int freq[], int prev) {
9      int maxLetter = 0, sum = 0;          // maxLetter = highest frequency, sum
      = total remaining letters
10     int minLetter = -1, maxIndex = 0; // minLetter = smallest valid choice,
      maxIndex = most frequent letter
11
12     // Find the smallest lexicographically valid letter different from prev
13     for (int i = 0; i < 26; i++) {
14         if (freq[i] > 0 && i != prev) {
15             minLetter = i;
16             break;
17         }
18     }
19
20     // Compute total remaining letters and the letter with maximum frequency
21     for (int i = 0; i < 26; i++) {
22         sum += freq[i];
23         if (freq[i] > maxLetter) {
24             maxLetter = freq[i];
25             maxIndex = i;
```

C++


```

26     }
27 }
28
29 // If any letter appears too often, rearrangement is impossible
30 if (maxLetter * 2 > sum + 1) return -1;
31
32 // If the most frequent letter must be placed now to avoid failure, force
  it
33 if (maxLetter * 2 > sum) return maxIndex;
34
35 // Otherwise, choose the smallest lexicographically valid letter
36 return minLetter;
37 }
38
39 int main() {
40     string s, ans = ""; // s = input string, ans = constructed result
41     cin >> s;
42
43     int freq[26] = {0}; // Frequency array for letters A-Z
44     for (char c : s) freq[c - 'A']++;
45
46     // Choose the first character (no previous restriction)
47     int idx = minLexPossible(freq, -1);
48     if (idx == -1) {
49         cout << "-1"; // Impossible to form valid string
50         return 0;
51     }
52
53     ans += char(idx + 'A'); // Append chosen character
54     freq[idx]--;           // Decrease its frequency
55
56     // Build the rest of the string character by character
57     for (int i = 1; i < s.size(); i++) {
58         // Previous character index is ans[i - 1] - 'A'
59         idx = minLexPossible(freq, ans[i - 1] - 'A');
60         if (idx == -1) {
61             cout << "-1"; // No valid continuation
62             return 0;
63         }
64         ans += char(idx + 'A'); // Append next character
65         freq[idx]--;           // Update frequency
66     }
67
68     cout << ans; // Output the lexicographically smallest valid arrangement
69 }

```


1.2.24. Grid Path Description

[Question - Grid Path Description](#) [Backup Link](#)

Hint:

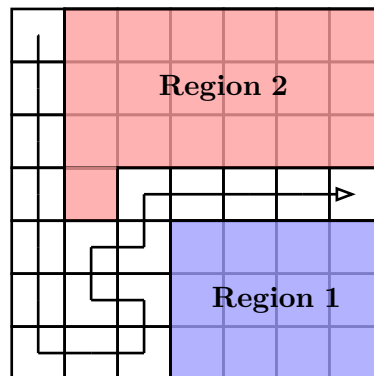
Try using backtracking⁹ but optimizing it for cases when you know it's impossible to reach the bottom-left while covering all other squares.

Solution:

You start at the top-left cell and must end at the bottom-left cell without visiting any cell twice. Each move must follow the given string, where ? allows any direction and other characters force a specific move. The code tries all allowed moves step by step while marking visited cells and backtracking when either all cells are reached or it's impossible to continue the path.

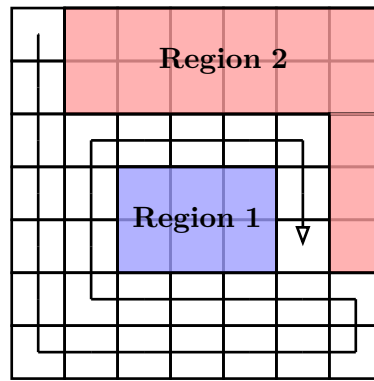
While you could implement this easily, this will be too slow. Hence, you need to optimize it to catch impossible routes as soon as possible.

1. If a path reaches the bottom-left cell early (i.e. before covering other cells), backtrack right here because all future paths can't end up at the bottom-left.
2. If a path gets stopped by a wall and has the option to move left or right along the wall, then the grid will get split into 2 regions, and you can't cover all cells in both regions. Hence you must backtrack
3. The extension to the previous point is that if a path get's stopped by cells previously visited and have to option to move left or right along the visited cells, the grid will again get split into 2 regions. Hence you must backtrack.



2 regions formed by getting stopped at a wall

⁹see Section 1.1.9



2 regions form by getting stopped by previously visited cells

Code:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  string path;
5  bool visited[7][7];          // Marks cells already used in the current path
6  int ans = 0;
7
8  const int dx[] = {1, -1, 0, 0}; // Change in row for each move
9  const int dy[] = {0, 0, 1, -1}; // Change in column for each move
10 const char dir[] = {'D', 'U', 'R', 'L'}; // Corresponding move letters
11
12 bool inside(int x, int y) {
13     return x >= 0 && x < 7 && y >= 0 && y < 7; // Checks grid boundaries
14 }
15
16 bool is_blocked(int x, int y) {
17     if (!inside(x, y) || visited[x][y]) return true; // Outside grid or
18     // already used
19     return false;
20 }
21
22 void gridPaths(int x, int y, int step) {
23     // If we reached the target cell
24     if (x == 6 && y == 0) {
25         if (step == 48) ans++; // Count only if all moves were used
26         return;
27     }
28     //Backtrack if the current path has split the grid into 2 regions.
29     //x+1 is right, x-1 is left, y+1 is up, y-1 is down.
30     if ((is_blocked(x + 1, y) && is_blocked(x - 1, y) &&
31         !is_blocked(x, y + 1) && !is_blocked(x, y - 1)) ||

```

C++

```

32         (!is_blocked(x + 1, y) && !is_blocked(x - 1, y) &&
33         is_blocked(x, y + 1) && is_blocked(x, y - 1)))
34         return;
35
36     visited[x][y] = true; // Mark this cell as visited
37
38     for (int d = 0; d < 4; ++d) {
39         int nx = x + dx[d];
40         int ny = y + dy[d];
41
42         // Enforce the given path character if it is not '?'
43         if (path[step] != '?' && path[step] != dir[d]) continue;
44
45         // Move only to valid and unused cells
46         if (!is_blocked(nx, ny))
47             gridPaths(nx, ny, step + 1);
48     }
49
50     visited[x][y] = false; // Undo the move before trying other
    possibilities
51 }
52
53 int main() {
54     cin >> path;
55     gridPaths(0, 0, 0); // Start from top-left with zero moves
    taken
56     cout << ans << endl;
57     return 0;
58 }

```


2.1.1.5. Left Shift (<<) and Right Shift (>>)

Left shifting is moving all the bits some number of places to the left. Each left shift is just multiplying the number by 2. So `cout << (3 << 4);` would be `000011` \rightarrow `000110` \rightarrow `001100` \rightarrow `011000` \rightarrow `110000`, which is $3 \times 2^4 = 3 \times 16 = 48$. Right shifting works in the exact opposite manner. Each right shift gives you the floor of the number divided by 2 ($\lfloor \frac{n}{2} \rfloor$). So `cout << (57 >> 3);` is `111001` \rightarrow `011100` \rightarrow `001110` \rightarrow `000111` = 7.

2.1.1.6. Lowest Set Bit (LSB)

The lowest set bit is the value of the rightmost bit of a binary number that is set to 1. This bit contributes the least to the number. For example, the number 20 in binary is 10100. The rightmost bit that is 1 is in the third position from the right (0-indexed from the right). The value it represents is $100_2 = 2^2 = 4$, which means $\text{LSB}(20) = 4$.

If you want to find the $\text{LSB}(n)$, all you have to do is $n \& -n$.

Why does this work? For that, you must first break up $-n$ into $\sim n + 1$ because that's taking the 2's complement. When you take the 1's complement of n ($\sim n$), the rightmost 1 becomes the rightmost 0. All bits to the right of this 0 are 1's.

If you now add 1 to get the 2's complement, all the ones up to the rightmost 0 become 0's and the rightmost 0 becomes a 1. So now when you take the **and** of **n** and **-n**, the bits just before the rightmost one have all been flipped, so **&** will make them all 0's. Only the rightmost bit is 1 in both **n** and **-n**, so it will be preserved. This will give you a new number in binary which is the **LSB(n)**.

Here's how it looks on 20:

$$\begin{array}{r} 20 = 0000000000000000000000000100 \\ \& - 20 = 1111111111111111111111111101 \\ \hline 0000000000000000000000000100 \end{array}$$

2.1.2. Bitmask

Bitmasking is the technique of using the binary representation of numbers to represent subsets of a set. Let's look at a problem which can be solved using bitmasks.

Say you're given an array and want to return all possible subsets of that array. Here's the code on how to do that, and then we'll go through it:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n = 3;
6      vector<int> v = {5, 4, 7};
7
8      // mask takes all values from 0 to 2^n - 1
9      for (int mask = 0; mask < (1 << n); mask++) {
10         cout << "{ ";
11
12         for (int i = 0; i < n; i++) {
```

C++

```

13     // checking if the ith bit of the mask is 1
14     if (mask & (1 << i))
15         cout << v[i] << " ";
16     }
17
18     cout << "}" << endl;
19 }
20
21 return 0;
22 }

```

In the code, the variable `mask` goes through all subsets, where each subset is numbered from 0 to $2^n - 1$. In this case $n = 3$, so `mask` goes from 0 to 7. Then for each value of `mask`, you output all the elements `v[i]` where the `i`th bit (from right to left) is true. This will generate the following output:

```

1 { }
2 { 5 }
3 { 4 }
4 { 5 4 }
5 { 7 }
6 { 5 7 }
7 { 4 7 }
8 { 5 4 7 }

```

2.1.3. Prefix Sum

Let's say you're asked this question: You're given an array of numbers, and then you're given some queries. Each query will give you a range. Your goal is to output the sum of all numbers in that range. For example, let's say you have the following array:

$$[5, -6, 4, 3, 12, 6, -7, -3]$$

And now you're told to find the sum of elements from index 4-7, index 2-5, index 1-3. The answers to that would be:

$$\begin{aligned}
 12 + 6 + -7 + -3 &= 8 \\
 4 + 3 + 12 + 6 &= 25 \\
 -6 + 4 + 3 &= 1
 \end{aligned}$$

Note that indices are 0-indexed.

You could solve this question by simply iterating through all elements in each range and then adding them up. However, each of these operations is amortized $O(n)$. If there are q queries, your complexity would be $O(nq)$. If n and q 's limits are 2×10^5 , $O(nq)$ would be too slow.

The much faster way would be to compute a **prefix sum** array. This means that every element `pref[i]` stores the sum of all elements from `v[0]` to `v[i]`. Now let's say you want to know the sum from index `a` to `b`. You only have to compute `pref[b] - pref[a-1]` to get the answer. Using our example, the prefix sum array would be:

original array [5, -6, 4, 3, 12, 6, -7, -3]
prefix sum array [5, -1, 3, 6, 18, 24, 17, 14]

Now the answers to the 3 queries are:

$$\begin{aligned}14 - 6 &= 8 \\24 - (-1) &= 25 \\6 - 5 &= 1\end{aligned}$$

You get the correct answer by only having to subtract 2 numbers rather than having to add an entire array.

Here's the code for the implementation of prefix sum:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, q;
6      cin >> n >> q;
7
8      vector<int> v(n);
9      for (int i = 0; i < n; i++)
10         cin >> v[i];
11
12     vector<int> pref(n);
13     pref[0] = v[0];
14
15     for (int i = 1; i < n; i++)
16         pref[i] = v[i] + pref[i-1];
17
18     for (int i = 0; i < q; i++) {
19         int a, b;
20         cin >> a >> b;
21
22         if (a != 0)
23             cout << (pref[b] - pref[a-1]) << endl;
24         else
25             cout << pref[b] << endl;
26     }
27
28     return 0;
29 }
```

C++

Sample input:

```
8 3
5 -6 4 3 12 6 -7 -3
```

```

4 7
2 5
1 3

```

Output:

```

8
25
1

```

The space complexity is $O(n)$ and the prefix sum construction runs in $O(n)$ time. Each query operation runs in $O(1)$ time.

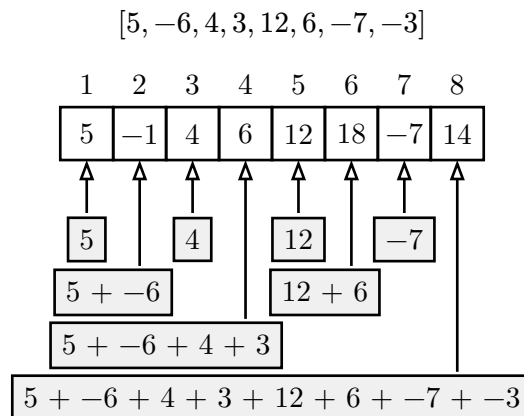
2.1.4. Binary Indexed Tree

Let's say for the question in the previous section, we not only want the ability to find the sum in a given range, but we also want to update an element in the array. This means that we need to be able to both change values in the array and output the sum in any given range quickly.

Our earlier approach of maintaining a prefix sum fails because, even though we can output the sum of elements in a range in $O(1)$, if we change even a single value, the time it takes to regenerate the whole array is amortized $O(n)$. For the constraints of $n \leq 2 \times 10^5, q \leq 2 \times 10^5$, this is too slow.

There is a data structure that can help us do updates and sums in $O(\log(n))$. This is called a **binary indexed tree (BIT)** or a **Fenwick tree**. In a Fenwick tree, each element is 1-indexed. The i th value in the Fenwick tree stores the sum of all elements in the original array from $i - \text{LSSB}(i) + 1$ to i (see Section 2.1.1.6 for the meaning of LSSB).

To understand this better, let's look at the array from our previous example and the Fenwick tree that's constructed from the array.



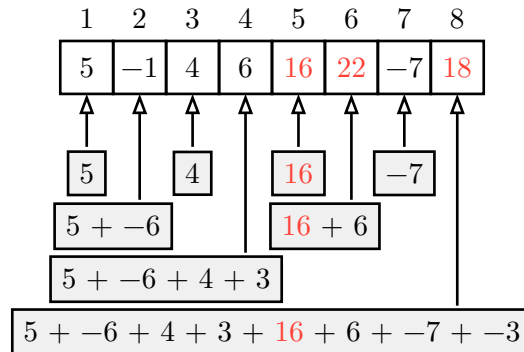
Note that the values in a Fenwick tree are 1-indexed, so there will be an empty element at `fenw[0]`.

Hopefully the image makes it clearer how data is stored. The reason for storing data like this is that if you want to add a value to one element in the original array, you only need to update $O(\log n)$ values in the Fenwick tree. After doing this, you can find the sum in $O(\log n)$. Say we wish to add 4 to the 5th index (1-indexed); we only need to update the 5th,

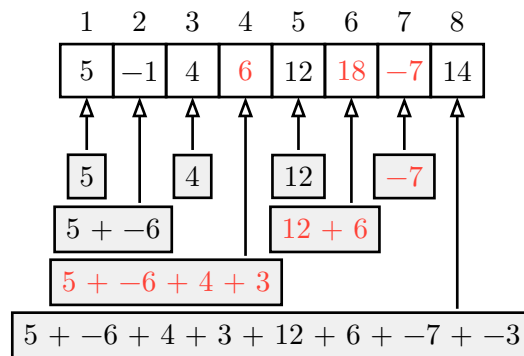
6th, 8th index. If you now want to compute the prefix sum of the array from index 7, you only need to add the values in the 16th index.

Here's a diagram to illustrate the changes:

Add 4 to the 5th index:



Prefix sum at the 7th index:



If you can calculate the prefix sum at some index i in $O(\log n)$, you can then do $\text{sum}(b) - \text{sum}(a - 1)$ to find the sum of numbers in the subarray from a to b .

Here's the code for the Fenwick tree implementation:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int n;
5  vector<int> fenw;
6
7  void add(int x, int k) {
8      for (; x <= n; x += x & -x) // x & -x is the LSSB(x)
9          fenw[x] += k;
10 }
11
12 int sum(int x) {
13     int ans = 0;
14     for (; x > 0; x -= x & -x) // x & -x is the LSSB(x)
15         ans += fenw[x];
```

C++

```

16     return ans;
17 }
18
19 int sum(int a, int b) {
20     return sum(b) - sum(a - 1);
21 }
22
23 int main() {
24     int q;
25     cin >> n >> q;
26     fenw.resize(n + 1, 0);
27     for (int i = 1; i <= n; i++) {
28         int x;
29         cin >> x;
30         add(i, x);
31     }
32
33     for (int i = 0; i < q; i++) {
34         int t;
35         cin >> t;
36         if (t == 1) { // addition queries
37             int x, k;
38             cin >> x >> k;
39             add(x, k);
40         }
41         else if (t == 2) { // range sum queries
42             int a, b;
43             cin >> a >> b;
44             int ans = sum(a, b);
45             cout << sum(a, b) << endl;
46         }
47     }
48     return 0;
49 }

```

Sample input:

```

8 6
5 -6 4 3 12 6 -7 -3
2 4 7
1 5 4
2 4 7
2 1 3
1 3 -8
2 2 7

```

Output:

14
18
3
8

2.1.4.1. Fenwick Trees as Indexed Sets

A Fenwick tree can also be used as an indexed set. In Section 2.1.7, a set was explained to be a data structure that lets you insert, find, and erase elements in $O(\log n)$ time. It is also sorted and contains unique elements. However, it's not possible to simply access the 2nd, 5th, or 12th value in a set unless you iterate all the way from the beginning to that position. That makes accessing elements at a specific index $O(n)$.

If you also want to be able to access elements at specific indexes in a set, you can use a Fenwick tree as a frequency table. This means that at `fenw[x]`, you store the number of times element `x` occurs in your original data. Of course, `fenw[x]` will actually store the sum of frequencies from `x - LSSB(x) + 1` to `x`, but you get the point. This makes it sorted by default because it describes how many times 1 appears, followed by how many times 2 appears, and so on.

Now if you want to add an element `a` to the set, you simply do `add(a, 1)` in the Fenwick tree to increase its frequency by 1. If you want to remove an element `b`, you do `add(b, -1)` to decrease its frequency by 1. If you want to find the index of the last occurrence of an element `c`, do `sum(c)`. If you want to find the index of the first occurrence of `c`, do `sum(c - 1) + 1`. And finally, the main difference is the ability to find what element is at position `i`. This requires a new function.

Here's the code for the search function:

```
1  int search(int idx){
2      int ans = 0;
3
4      for(int k = floor(log2(n)); k >= 0; k--){ // go through the powers of 2
5          if(ans + (1 << k) <= n && fenw[ans + (1 << k)] < idx){ // this element is
              before idx
6              ans += 1 << k; // update the answer
7              idx -= fenw[ans]; // account for all indices up to fenw[ans]
8          }
9      }
10
11     return ans + 1; // ans was the value that was before idx, so one value
              ahead is at idx
12 }
```

C++

In the code of the search function, you start with `k = floor(log2(n))` where 2^k is the largest power of 2 less than or equal to n . Then for each value, you check to see if its index (`fenw[ans + (1 << k)]`) is less than `idx`. If it is, you add 2^k to the answer and then subtract `idx` by the indexes covered (`fenw[ans]`).

Since `ans` stores the number that is definitely before the index, `ans + 1` tells you what number is exactly at `idx`. The reason why you can't find the index directly is because the Fenwick tree frequency table can have multiple of the same numbers, so you can't guarantee that you will find what value is at your exact `idx`.

Finally, there is one more thing required to make a Fenwick tree useful as an indexed set. A normal set's size is based on the amount of input n , which makes its space complexity $O(n)$. However, a Fenwick tree is built on the frequency table of the data, which makes it have a space complexity of $O(a)$ where a is the largest input. Usually $n \leq 2 \times 10^5$, however a can be as large as 10^9 ! This would require around **30 gigabytes** of data, which is way past the memory limits of a question. The solution to this problem is to do **index compression**. Because the amount of data is going to be small, we simply remap all the large numbers down to smaller numbers.

For example, say you have the following array:

[3, 10, 4, 5, 2, 2]

If we were to store this array as an indexed set, it would require the storage of $10 + 1$ (because 1-indexed) ints of storage. Notice that there are only 5 unique numbers in this entire vector (2, 3, 4, 5, 10). If we were to reassign these numbers to just (1, 2, 3, 4, 5), our indexed set would only take $5 + 1$ (because 1-indexed) ints of memory. This technique is called **index compression**.

2.1.4.2. Index Compression

To perform index compression, you need to sort the original vector of values stored in a different vector. Let's call this other vector `comp`. Then remove all the duplicate elements from `comp`. Then to compress the indices, find at what index values from the original vector appear in `comp`. This can be done efficiently with `lower_bound()` because `comp` is sorted. For the above example, `comp` would look like:

0	1	2	3	4
2	3	4	5	10

Because of `comp`, 2 will get mapped to $0 + 1 = 1$ (because 1-indexing for the Fenwick tree), 3 gets mapped to $1 + 1 = 2$, and so on.

Here's the code for the implementation of index compression:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      int n;
6      cin >> n;
7      vector<int> v(n);
8      for(int i = 0; i < n; i++)
9          cin >> v[i];
10
11     vector<int> comp = v;
```

C++

```

12  sort(comp.begin(), comp.end());
13  comp.erase(unique(comp.begin(), comp.end()), comp.end());
14
15  for(int i = 0; i < n; i++)
16      v[i] = lower_bound(comp.begin(), comp.end(), v[i]) - comp.begin() + 1;
17      // lower_bound - comp.begin() gives you the index and +1 makes it 1-
        indexed
18
19  return 0;
20 }

```

Sample Input:

```
1 3 10 4 5 2 2
```

Output:

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

The code uses the `std::unique()` function, which in a sorted vector, moves all duplicate elements to the end and returns a pointer at the start of the duplicate elements. We then use this pointer and erase all the duplicate elements till the end to generate `comp` correctly.

To compress all the values in `v`, we get an iterator to their position in `comp` using `lower_bound()` and then subtract it with `comp.begin()` to get its position (0-indexed). We then add 1 to get the compressed value such that it is 1-indexed.

Here's a code that fully summarizes the process of an indexed set:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int n;
5  vector<int> fenw;
6
7  void add(int x, int k){
8      for(; x <= n; x += x & -x) // x & -x is the LSSB(x)
9          fenw[x] += k;
10 }
11
12 int sum(int x){
13     int ans = 0;
14     for(; x > 0; x -= x & -x) // x & -x is the LSSB(x)
15         ans += fenw[x];
16     return ans;

```

C++

```

17 }
18
19 int sum(int a, int b){
20     return sum(b) - sum(a - 1);
21 }
22
23 int search(int idx){
24     int ans = 0;
25
26     for(int k = floor(log2(n)); k >= 0; k--){ // go through the powers of 2
27         if(ans + (1 << k) <= n && fenw[ans + (1 << k)] < idx){ // this element is
            before idx
28             ans += 1 << k; // update the answer
29             idx -= fenw[ans]; // account for all indices up to fenw[ans]
30         }
31     }
32
33     return ans + 1; // ans was the value before idx, so one value ahead is at
        idx
34 }
35
36 int main(){
37     int n;
38     cin >> n;
39     vector<int> v(n);
40     for(int i = 0; i < n; i++){
41         cin >> v[i];
42     }
43     vector<int> comp = v;
44     sort(comp.begin(), comp.end());
45     comp.erase(unique(comp.begin(), comp.end()), comp.end());
46
47     for(int i = 0; i < n; i++){
48         v[i] = lower_bound(comp.begin(), comp.end(), v[i]) - comp.begin() + 1;
49     }
50     fenw.resize(comp.size() + 1);
51     n = comp.size();
52
53     for(int i = 0; i < v.size(); i++){
54         add(v[i], 1);
55     }
56     // Now the fenw vector is fully ready to behave as an indexed set
57
58     return 0;
59 }

```


2.1.5. Linked List

A linked list is a data structure where every element in the list has a value and a pointer to the next element. This makes removing elements at a given position $O(1)$ because you only have to make the element before the erased one point to the element after the erased one.

The same is true for inserting an element at a given position.

Linked lists can either be linear or circular. In a linear linked list, the last element points to `nullptr`. In a circular linked list, the last element points back to the first element.

Here's the implementation of a linear linked list:

```
1  struct Node{
2      int val; // the value in the current element
3      Node* nxt; // a pointer to the next element
4      Node(const int& val = 0, Node* nxt = nullptr){ // constructor
5          this->val = val;
6          this->nxt = nxt;
7      }
8  };
9
10 struct List{
11     Node* head;
12
13     List(const int& size = 0, const int& val = 0){ // creating the linked list.
14         // The list will have "size" elements filled with val.
15         head = new Node();
16         Node* cur = head;
17         for(int i = 1; i <= size; i++){
18             cur->val = val;
19             cur = cur->nxt = new Node();
20         }
21
22         void insert(Node* pos, const int& val){ // inserts a new node after pos
23             Node* nxt = pos->nxt;
24             pos->nxt = new Node(val, nxt);
25         }
26
27         void erase(Node* pos){ // Erases the next node after pos
28             if(pos->nxt == nullptr)
29                 return;
30             Node* nxt = pos->nxt;
31             pos->nxt = nxt->nxt;
32             delete(nxt);
33         }
34     };
}
```

C++

Of course, this is a very poor implementation with not much memory safety, leading to memory leaks. Fortunately, C++ has its own implementation of a linked list.

2.1.5.1. `std::list`

Here's a code example of how the C++ implementation of a linked list is used:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      int n;
6      cin >> n;
7
8      list<int> l(n, 0);
9      for(list<int>::iterator it = l.begin(); it != l.end(); it++){
10         int x;
11         cin >> x;
12         *it = x;
13     }
14
15     for(list<int>::iterator it = l.begin(); it != l.end(); it++) // loop to
        erase every odd element (1-indexed)
16         it = l.erase(it);
17
18     for(list<int>::iterator it = l.begin(); it != l.end(); it++)
19         l.insert(it, *it - 1); // before each element, insert the value of that
        element - 1
20
21     for(list<int>::iterator it = l.begin(); it != l.end(); it++)
22         cout << *it << " ";
23
24     return 0;
25 }
```

Sample input:

```
1 6
2 4 -6 3 8 7 -2
```

Output:

```
1 -5 -6 9 8 -1 -2
```

As you can see from the code, if you want to store a value, you simply update `*it`. If you want to insert a value before the current iterator, use `l.insert(it, val)`. Lastly, if you want

to erase the current iterator, use `l.erase(it)`. `erase()` also returns the position to the next iterator so that you don't invalidate your current iterator.

For the `std::list` documentation, click [here](#).

2.1.6. Greedy Algorithms

A greedy algorithm is a type of algorithm where the solution for a smaller subpart of the question also applies to the whole question. A greedy algorithm never goes back and corrects its previous decision. Let's take a look at a question that can be solved with a greedy algorithm:

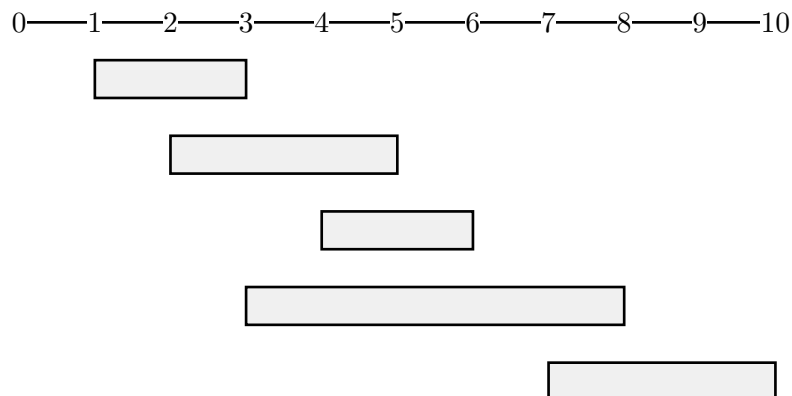
Question: You are given a list of events. Every event has a start time and an end time. You can only attend one event at a time. Your goal is to pick events in such a way so that you can attend the maximum number of events.

The algorithm to solve this question is to pick an event which ends the earliest, and then pick the next non-overlapping event that ends the earliest, and so on. This always ensures that you can pick the largest number of events.

The reason for this is to think of the opposite. If you were to pick an event that ends later, in the best case you can still pick the same number of new events. However, in the worst case you will overlap some events that you could have picked. Let's look at the following example:

start	end
1	3
2	5
4	6
3	8
7	10

Here's the visualization of all the events:



These events are currently sorted in ascending order of their end times. Let's say instead of following the strategy by picking the first event $\{1, 3\}$, you were to pick the second event $\{2, 5\}$, which has a later end time. The only new event you can also attend is $\{7, 10\}$. If you were to pick $\{1, 3\}$, you can pick $\{7, 10\}$, but you can also pick $\{4, 6\}$. This is why it's always better to pick events which end earlier—because you have nothing to lose and everything to gain.

There are many other questions where you can use a greedy approach, and you'll understand how to use them by solving such questions¹⁰.

¹⁰See Section 2.2.24

2.1.7. Sets

A **set** in a data structure in **c++**, which has the following properties:

1. A new element can be added to a **set** in $O(\log n)$ time.
2. An element can be found in $O(\log n)$ time.
3. An element can be removed in $O(\log n)$ time.
4. All elements are sorted in ascending order.
5. All elements in a **set** are unique

Here's a quick problem, whose solution will explain how to use sets:-

Accept numbers from a user. Then check if a number exists in the list, if it does, print **YES** followed by removing that number from the list, otherwise print **NO**. At the end print the new list in ascending order.

Solution:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5
6      int n, q;
7      cin >> n >> q;
8      set<int> s;
9
10     for(int i = 0; i < n; i++){
11         int x;
12         cin >> x;
13         s.insert(x); // Inserts a value into the set.
14     }
15
16     for(int i = 0; i < q; i++){
17         int x;
18         cin >> x;
19         if(s.find(x) != s.end()){ // s.find(x) returns the position of x in the
            set.
20             cout << "YES" << endl;
21             s.erase(x); // s.erase(x) removes x from the set
22         }
23         else
24             cout << "NO" << endl;
25     }
26
27     for(set<int>::iterator it = s.begin(); it != s.end(); it++){
28         cout << *it << " ";
29     }
30     return 0;
```

In the solution, we can see that

- `s.insert(x)` inserts `x` into the `set s`. This will ensure that `s` will remain sorted by inserting it into the correct place.
- `s.find(x)` returns the position of `x` in `s`. If `x` doesn't exist, it will return `s.end()` which is a pointer at one place past the position of the last element in the set.
- `s.erase(x)` removes `x` from `s`.

Finally we end up printing all values that are currently in `s`. However, you may notice that instead of the traditional loop with a variable `i` that increase, we're using a `set<int>::iterator`. An iterator is simply a pointer that is used to go over a data structure that is not traditionally indexed. You can very much use the same syntax with vectors too, but it's not necessary.

2.1.7.1. lower_bound and upper_bound

Unlike for vectors, if you try to use the `lower_bound()` and `upper_bound()` functions, it won't execute binary search and will instead search through them in linear time. The reason for this is that set iterators are not random access, i.e. you can't just say `it + 5` and get the element 5 places ahead of `it`. Instead, you must run a loop to do `it++` 5 times. Fortunately, `set`'s have their own implementation of `lower_bound()` and `upper_bound()`. If you have a `set<int> s`, then `s.lower_bound(t)` will return an iterator to the lower bound of `t` and `s.upper_bound(t)` will return an iterator to the upper bound of `t`.

2.1.7.2. multiset

A `multiset` is exactly like a `set` except that it can store multiple of the same elements, whereas a `set` does not store duplicates. The syntax for using a `multiset` is identical to a `set`, just write `multiset` instead of `set`

2.1.7.3. unordered_set

An `unordered_set` works a bit different than a `set`. It supports the following operations

1. A new element can be added to a `unordered_set` in $O(1)^*$ time.
2. An element can be found in $O(1)^*$ time.
3. An element can be removed in $O(1)^*$ time.
4. The order of elements are random.
5. All elements in a `unordered_set` are unique

Notice how it almost identical to a `set` other than the fact that it faster with the downside of no sorted order. This looks as if it would be useful to use an `unordered_set` instead of a `set` if you just want to check if elements exists or not due to their $O(1)$ vs the much slower $O(\log n)$. However, this $O(1)$ is not guaranteed and for large test cases that you may expect during questions, it usually ends being the much worse $O(n)$ which will lead to a Time Limit Exceeded (TLE). This is why you should always use a `set` over an `unordered_set` even if you don't care about the sorting order.

2.1.7.4. unordered_multiset

Again, it's the same as an `unordered_set` except that it can store multiple of the same element. This also has $O(1)$ operations with the caveat that its worse case is $O(n)$. So you should use `multiset` over `unordered_multiset`.

2.1.8. Maps

A map is a data structure in C++ which has the following properties:

1. A new key-value pair can be added to a map in $O(\log n)$ time.
2. A value can be accessed using its key in $O(\log n)$ time.
3. A key-value pair can be removed in $O(\log n)$ time.
4. All keys are sorted in ascending order.
5. All keys in a map are unique.

Here's a quick problem whose solution will explain how to use maps:

Accept student names and their scores from a user. Then accept queries for student names and print their scores if they exist; otherwise, print **NOT FOUND**. After processing all queries, print all students and their scores in alphabetical order by name.

Solution:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5
6      int n, q;
7      cin >> n >> q;
8      map<string, int> m;
9
10     for(int i = 0; i < n; i++){
11         string name;
12         int score;
13         cin >> name >> score;
14         m[name] = score; // Inserts or updates the key-value pair.
15     }
16
17     for(int i = 0; i < q; i++){
18         string name;
19         cin >> name;
20         if(m.find(name) != m.end()){ // m.find(name) returns the position of the
            key in the map.
21             cout << m[name] << endl;
22         }
23         else
24             cout << "NOT FOUND" << endl;
25     }
26
27     for(map<string, int>::iterator it = m.begin(); it != m.end(); it++){
28         cout << it->first << " " << it->second << endl;
29     }
30     return 0;
```

C++

In the solution, we can see that:

- `m[key] = value` inserts or updates the key-value pair in the map `m`. This will ensure that `m` remains sorted by key by inserting it into the correct place.
- `m.find(key)` returns the position of `key` in `m`. If `key` doesn't exist, it will return `m.end()`, which is a pointer at one place past the position of the last element in the map.
- `m.erase(key)` removes the key-value pair with the specified `key` from `m`.

Finally, we end up printing all key-value pairs that are currently in `m`. However, you may notice that instead of the traditional loop with a variable `i` that increases, we're using a `map<string, int>::iterator`. An iterator is simply a pointer that is used to go over a data structure that is not traditionally indexed. When iterating through a map, each iterator points to a pair, where `it->first` is the key and `it->second` is the value.

2.1.8.1. Important Note on Accessing Keys

When you use `m[key]`, if the key doesn't exist in the map, it will automatically create an entry with that key and a default value (0 for integers, empty string for strings, etc.). This can sometimes lead to unintended behavior.

If you want to check if a key exists without creating it, always use `m.find(key) != m.end()` instead of just accessing `m[key]`.

2.1.8.2. lower_bound and upper_bound

Just like sets, maps have their own implementation of `lower_bound()` and `upper_bound()` that work in $O(\log n)$ time.

If you have a `map<int, int> m`, then:

- `m.lower_bound(t)` will return an iterator to the first key-value pair whose key is not less than `t`
- `m.upper_bound(t)` will return an iterator to the first key-value pair whose key is greater than `t`

2.1.8.3. multimap

A `multimap` is exactly like a map except that it can store multiple values for the same key, whereas a `map` does not allow duplicate keys.

The syntax for using a `multimap` is similar to a `map`—just write `multimap` instead of `map`. However, note that you cannot use the `[]` operator with `multimap` since a key can have multiple values. Instead, you must use `insert()` and `find()`.

2.1.8.4. unordered_map

An `unordered_map` works a bit differently than a map. It supports the following operations:

- A new key-value pair can be added to an `unordered_map` in $O(1)$ time.
- A value can be accessed using its key in $O(1)$ time.
- A key-value pair can be removed in $O(1)$ time.

- The order of keys is random.
- All keys in an `unordered_map` are unique.

Notice how it's almost identical to a map other than the fact that it's faster, with the downside of no sorted order.

This looks as if it would be useful to use an `unordered_map` instead of a `map` if you just want to store and retrieve values by keys, due to their $O(1)$ versus the much slower $O(\log n)$.

However, this $O(1)$ is not guaranteed, and for large test cases that you may expect during questions, it usually ends up being the much worse $O(n)$, which will lead to a Time Limit Exceeded (TLE).

This is why you should always use a `map` over an `unordered_map`, even if you don't care about the sorting order.

2.1.8.5. `unordered_multimap`

Again, it's the same as an `unordered_map` except that it can store multiple values for the same key.

This also has $O(1)$ operations with the caveat that its worst case is $O(n)$. So you should use `multimap` over `unordered_multimap`.

2.1.9. Stacks

A stack is a data structure in C++ which has the following properties:

1. A new element can be added to the top of a stack in $O(1)$ time.
2. The top element can be accessed in $O(1)$ time.
3. The top element can be removed in $O(1)$ time.
4. Elements follow the Last In First Out (LIFO) principle.
5. You can only access the most recently added element.

Here's a quick problem whose solution will explain how to use stacks:

Accept numbers from a user and push them onto a stack. Then process queries where you either add a new number to the stack or remove the top number and print it. At the end, print all remaining elements in the stack from top to bottom.

Solution:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5
6      int n, q;
7      cin >> n >> q;
8      stack<int> st;
9
10     for(int i = 0; i < n; i++){
```

C++

```

11     int x;
12     cin >> x;
13     st.push(x); //Pushes a value onto the top of the stack.
14 }
15
16 for(int i = 0; i < q; i++){
17     int type;
18     cin >> type;
19     if(type == 1){
20         int x;
21         cin >> x;
22         st.push(x); //Adds x to the top of the stack
23     }
24     else if(type == 2 && !st.empty()){
25         cout << st.top() << endl; //st.top() returns the top element
26         st.pop(); //st.pop() removes the top element
27     }
28 }
29
30 while(!st.empty()){
31     cout << st.top() << " ";
32     st.pop();
33 }
34 return 0;
35 }

```

In the solution, we can see that:

- `st.push(x)` pushes `x` onto the top of the stack `st`. This operation adds the element to the very top.
- `st.top()` returns the element at the top of `st`. This does not remove the element, just accesses it.
- `st.pop()` removes the top element from `st`. Note that `pop()` does not return anything—you must use `top()` first if you want the value.
- `st.empty()` returns `true` if the stack is empty, `false` otherwise.

Finally, we end up printing all values that are currently in `st` from top to bottom. Notice that we must keep calling `pop()` to remove elements as we go, since we can only access the top element at any given time.

2.1.9.1. Common Use Cases

Stacks are particularly useful for:

- **Reversing elements:** Since stacks follow LIFO, pushing elements and then popping them gives you the reverse order.
- **Balanced parentheses:** Push opening brackets onto the stack and pop when you encounter closing brackets.

- **Function call tracking:** The system uses a call stack to keep track of function calls and returns.
- **Depth-First Search (DFS):** Stacks can be used to implement DFS traversal in graphs and trees.

2.1.9.2. Size and Empty Check

Before accessing or removing elements from a stack, it's important to check if it's empty:

```
1 if(!st.empty()){
2     int x = st.top();
3     st.pop();
4 }
```

C++

You can also get the size of the stack using `st.size()`, which returns the number of elements currently in the stack in $O(1)$ time.

2.1.9.3. Stack vs Vector

You might wonder why you'd use a stack when you could just use a vector and always access/modify the last element using `back()` and `pop_back()`.

The answer is that stacks provide a cleaner interface when you only need LIFO behavior. They prevent accidental access to middle elements and make your code's intent clearer. However, if you need to access elements other than the top, you should use a vector or deque instead.

2.1.9.4. Example: Balanced Parentheses

Here's a classic problem that demonstrates the power of stacks:

```
1 bool isBalanced(string s){
2     stack<char> st;
3
4     for(char c : s){
5         if(c == '(' || c == '{' || c == '['){
6             st.push(c);
7         }
8         else{
9             if(st.empty()) return false;
10
11             char top = st.top();
12             st.pop();
13
14             if(c == ')' && top != '(') return false;
15             if(c == '}' && top != '{') return false;
16             if(c == ']' && top != '[') return false;
17         }
18     }
```

C++

```

18     }
19
20     return st.empty();
21 }

```

This function checks if parentheses, braces, and brackets are properly balanced in a string by using a stack to match opening and closing symbols.

2.1.9.5. Important Notes

- Unlike vectors, stacks do not support iteration. You cannot use a loop to go through all elements without removing them.
- Always check if a stack is empty before calling `top()` or `pop()` to avoid runtime errors.

2.1.10. Lambda expressions

Lambda expressions are a way to write functions in line without having to write them separately. For example:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5
6      int n;
7      cin >> n;
8
9      function<int (int)> fact = [&] (int num){ // defining the lambda expression
10         if(num == 1)
11             return 1;
12         return num * fact(num-1);
13     };
14
15     cout << fact(n) << endl;
16
17     return 0;
18 }

```

As you can see we've defined a function within the main function. The first part `function<int (int)>` says that you're making a function with return type `int` and one `int` parameter. Then after the equal to the `[&]` part allows you to access variables in the scope of the outer function by reference. `[=]` would allow you to access them by value and `[]` wouldn't allow any access. Then you write the actual contents of the function inside the braces.

Lambda expressions are also useful to just make temporary functions without having to make it into a variable. You'll see this used properly in the next section.

2.1.11. Sorting with a custom sorting order.

Say you wish to sort a `vector` in descending order, or you have something more complicated in mind. Well the `sort()` function has an extra parameter to supply your own sorting order.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5
6      int arr[] = {3,4,6,2,5,1};
7      vector<int> v = {6,2,4,5,1,3};
8
9      sort(arr, arr+6, greater<int>()); //Sorts the array {6,5,4,3,2,1}
10     sort(v.begin(), v.end(), greater<int>()); //Sorts the vector {6,5,4,3,2,1}
11
12     return 0;
13 }
```

The `greater<int>()` function returns true when for the 2 inputs `a` and `b`, `a > b`. Using `sort` this way ensure that all elements make your comparator function true.

Let's say you want to sort a `vector<pair<int,int>>` such that the second element is sorted in ascending order and only if they are equal are the first elements sorted in descending order. Here's how you could go about it, this will also demonstrate how to use lambda expressions:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5
6      int n;
7      cin >> n;
8      vector<pair<int,int>> v;
9      for(int i = 0; i < n; i++)
10         cin >> v[i];
11
12     sort(v.begin(), v.end(), [](const pair<int,int> &a, const pair<int,int> &b)
13     {
14         return a.second < b.second || a.second == b.second && a.first > b.first;
15     });
16     return 0;
17 }
```

2.2. Solutions to CSES Questions

2.2.1. Distinct Numbers

[Question - Distinct Numbers](#)

[Backup Link](#)

Solution:

Accept all the numbers and insert them into a set¹¹. Then report the size of the set. This works due to the fact that a set only stores unique elements and removes duplicates.

Code:

```
1
2  #include <bits/stdc++.h>
3  using namespace std;
4
5  int main(){
6
7      int n;
8      cin >> n;
9
10     set<int> s;
11
12     for(int i = 0; i < n; i++){
13         int x;
14         cin >> x;
15         s.insert(x); // Accepting and inserting the values into the set.
16     }
17
18     cout << s.size() << endl; // Outputs the number of unique elements.
19
20     return 0;
21 }
```

C++

¹¹See Section 2.1.7

2.2.2. Apartments

[Question - Apartments](#) [Backup Link](#)

Solution:

We can sort both applicants and apartments, then use a two pointer approach to match each applicant with the smallest available apartment whose size differs by at most k .

The two pointer approach is when you either move both pointers, or one of the pointers based on a condition.

In this case if an apartment is too small for the current applicant, we move to the next apartment which is larger.

If an apartment is too large for the current applicant, we move to the applicant who wants a large apartment.

Lastly if an apartment is of the right size, we increase the answer by 1 and go to the next apartment and next applicant.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, m, k;
6      cin >> n >> m >> k;
7
8      vector<int> applicants(n), apartments(m);
9
10     // Read applicant preferences
11     for (int i = 0; i < n; i++)
12         cin >> applicants[i];
13
14     // Read apartment sizes
15     for (int i = 0; i < m; i++)
16         cin >> apartments[i];
17
18     // Sort both arrays
19     sort(applicants.begin(), applicants.end());
20     sort(apartments.begin(), apartments.end());
21
22     int count = 0;
23     int i = 0, j = 0;
24
25     // Two-pointer approach to match applicants to apartments
26     while (i < n && j < m) {
```

C++

```

27         // Check if current apartment fits current applicant's preference
28         if (abs(applicants[i] - apartments[j]) <= k) {
29             count++; //increase the answer
30             i++; //move to next applicant
31             j++; //move to next largest apartment
32         }
33         // If apartment is too small, try the next larger apartment
34         else if (applicants[i] - apartments[j] > k)
35             j++;
36         // If apartment is too big, try the next applicant who wants a bigger
37         // apartment
38         else
39             i++;
40     }
41     cout << count << endl;
42     return 0;
43 }

```


2.2.3. Ferris Wheel

[Question - Ferris Wheel](#) [Backup Link](#)

Solution:

The algorithm sorts all weights, then uses two pointer, one at the lightest and one at the heaviest person, to form pairs without exceeding the limit. If they can share a gondola, both of them ride it; otherwise, the heavier one goes alone. This greedy pairing minimizes the total number of gondolas.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, x;
6      cin >> n >> x;
7
8      vector<int> weights(n);
9      for (int i = 0; i < n; i++) {
10         cin >> weights[i];
11     }
12
13     // Sort the weights
14     sort(weights.begin(), weights.end());
15
16     int gondolas = 0;
17     int left = 0, right = n - 1;
18
19     while (left <= right) {
20         // If heaviest and lightest can share a gondola
21         if (weights[left] + weights[right] <= x) {
22             left++; // go to the next lightest
23             right--; // go to the next heaviest
24         }
25         // Otherwise, heaviest gets their own gondola
26         else {
27             right--;
28         }
29         gondolas++;
30     }
31 }
```

C++

```
32     cout << gondolas << endl;  
33  
34     return 0;  
35 }
```

2.2.4. Concert Tickets

[Question - Concert Tickets](#) [Backup Link](#)

Solution:

Store all ticket prices in a `multiset`¹² to keep them sorted and allow duplicates. Each customer gives an offer, and you use `upper_bound()`¹³ to find the first price strictly greater than that offer, then step one step back to get the best affordable ticket. If such a ticket exists, print it and remove it; otherwise print `-1`. This algorithm neatly handles each request without iterating through the whole list.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, m, input;
6      cin >> n >> m;
7
8      // Multiset to store ticket prices in sorted order
9      multiset<int> prices;
10
11     // Store the m offers from customers
12     vector<int> offers(m);
13
14     // Insert the n ticket prices into the multiset
15     for (int i = 0; i < n; i++) {
16         cin >> input;
17         prices.insert(input);
18     }
19
20     // Read all offers
21     for (int i = 0; i < m; i++)
22         cin >> offers[i];
23
24     // For each offer, try to find the best possible ticket
25     for (int i = 0; i < m; i++) {
26
27         // Find the first price strictly greater than the offer
28         auto it = prices.upper_bound(offers[i]); // auto gets set to
29         multiset<int>::iterator
```

¹²See Section 2.1.7.2

¹³See Section 1.1.7.2

```

30         // If upper_bound points to begin(), no ticket <= offer exists
31         if (it == prices.begin())
32             cout << "-1" << endl;
33         else {
34             // Move iterator to the largest price <= offer
35             --it;
36
37             // Output that price
38             cout << *it << endl;
39
40             // Remove that ticket so it can't be reused
41             prices.erase(it);
42         }
43     }
44
45     return 0;
46 }

```

2.2.5. Restaurant Customers

[Question - Restaurant Customers](#) [Backup Link](#)

Solution:

The algorithm sorts all arrival and departure times, then uses two pointers to simulate guests entering and leaving. Each arrival increases the current count, and each departure decreases it. The maximum value reached during this sweep gives the peak number of guests present simultaneously.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5
6      int n;
7      cin >> n;
8      vector<int> arrivals(n), departures(n);
9      for (int i = 0; i < n; ++i) {
10         cin >> arrivals[i] >> departures[i];
11     }
12
13     // Sort arrival and departure times
14     sort(arrivals.begin(), arrivals.end());
15     sort(departures.begin(), departures.end());
16
17     int i = 0, j = 0, curr = 0, ans = 0;
18     // Sweep through both arrays to find maximum overlap
19     while (i < n && j < n) {
20         if (arrivals[i] < departures[j]) {
21             curr++;      // new guest arrives
22             ans = max(ans, curr);
23             i++;
24         } else {
25             curr--;      // a guest departs
26             j++;
27         }
28     }
29
30     cout << ans << "\n"; // maximum guests present at once
31     return 0;
32 }
```

C++

2.2.6. Movie Festival

[Question - Movie Festival](#)

[Backup Link](#)

Solution:

We store each movie as a pair of (end_time, start_time) and sort by end_time so we can always consider the earliest finishing movie first. The greedy approach works because picking the movie that ends earliest leaves maximum time for future movies.

We iterate through all movies, watching one only if it starts after the previous one ends. Each time we do, we increment our count and update the latest end time, ensuring the optimal number of movies are chosen.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      // Store each movie as a pair (end_time, start_time)
9      vector<pair<int, int>> movies(n);
10     for (int i = 0; i < n; ++i) {
11         int start, end;
12         cin >> start >> end;
13         movies[i] = {end, start};
14     }
15
16     // Sort movies by their ending time (greedy choice)
17     sort(movies.begin(), movies.end());
18
19     int maxMovies = 0;
20     int currentEnd = 0; // The end time of the last watched movie
21
22     // Iterate through all movies
23     for (pair<int, int> [end, start] : movies) {
24         // If the current movie starts after or exactly when the previous one
        ended
25         if (start >= currentEnd) {
26             maxMovies++; // Watch this movie
27             currentEnd = end; // Update the end time to this movie's end
28         }
29     }
```

```
30
31     cout << maxMovies << endl; // Output the result
32     return 0;
33 }
```

2.2.7. Sum of Two Values

[Question - Sum of Two Values](#) [Backup Link](#)

Solution:

The algorithm sorts all numbers, then uses two pointers, one starting at the smallest and one at the largest value, to find a pair that sums to the target. If the sum is too small, the left pointer moves right to a larger number; if too large, the right pointer moves left to a smaller number.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, target;
6      cin >> n >> target;
7
8      // Store each number along with its original index
9      vector<pair<int, int>> nums(n);
10     for (int i = 0; i < n; i++) {
11         // number value
12         cin >> nums[i].first;
13
14         // original position (1-based index)
15         nums[i].second = i + 1;
16     }
17
18     // Sort numbers by value to apply two-pointer technique
19     sort(nums.begin(), nums.end());
20
21     int left = 0, right = n - 1;
22     while (left < right) {
23         int sum = nums[left].first + nums[right].first;
24
25         // If target sum found, print their original indices
26         if (sum == target) {
27             cout << nums[left].second << " " << nums[right].second;
28             return 0;
29         }
30         // Move pointers based on comparison with target
31         else if (sum < target)
32             left++; // make the sum larger
33         else
```



```
34         right--;// make the sum smaller
35     }
36
37     // If no valid pair found
38     cout << "IMPOSSIBLE";
39     return 0;
40 }
```

2.2.8. Maximum Subarray Sum

[Question - Maximum Subarray Sum](#) [Backup Link](#)

Solution:

The algorithm finds the maximum possible sum of a continuous sequence in an array. It begins by assuming the first element is the best sum. Then, as it moves through the array, it decides whether to add the current element to the current sum or start the sum fresh from the current number. At each step, it updates the overall best sum found so far, ensuring the final answer is the largest contiguous total.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      vector<long long> arr(n);
9      for (int i = 0; i < n; i++) {
10         cin >> arr[i];
11     }
12
13     // max_current = maximum subarray sum ending at the current index
14     // max_global = overall maximum subarray sum found so far
15     long long max_current = arr[0];
16     long long max_global = arr[0];
17
18     // Iterate through the array
19     for (int i = 1; i < n; i++) {
20         // Either start a new subarray at arr[i] or extend the existing one
21         max_current = max(arr[i], max_current + arr[i]);
22
23         // Update the global maximum if needed
24         max_global = max(max_global, max_current);
25     }
26
27     // Output the maximum subarray sum
28     cout << max_global << endl;
29     return 0;
30 }
```

C++

2.2.9. Stick Lengths

[Question - Stick Lengths](#) [Backup Link](#)

Hint:

Think about what value minimizes the sum of absolute distances from all points. Consider a number line with all stick lengths plotted on it. If you need to pick one point that minimizes the total distance to all other points, what mathematical property does that point have?

Solution:

The key insight is that the median minimizes the sum of absolute deviations. When we need to make all sticks equal length, we're essentially finding a target value that minimizes the total cost, where cost is the absolute difference between each stick's current length and the target.

Why the median? Consider what happens when we move the target value slightly:

If we move the target to the right by 1 (increase it), all sticks shorter than the target need to be lengthened by 1, while sticks longer than the target need to be shortened by 1.

Example: Consider sticks with lengths [2, 3, 5, 8, 9] (already sorted). The median is 5.

- At median (target = 5): Cost = $|2 - 5| + |3 - 5| + |5 - 5| + |8 - 5| + |9 - 5|$
 $= 3 + 2 + 0 + 3 + 4 = \mathbf{12}$
- Move right (target = 6): Cost = $|2 - 6| + |3 - 6| + |5 - 6| + |8 - 6| + |9 - 6| = 4 + 3 + 1 + 2 + 3 = \mathbf{13}$
 - ▶ The 3 sticks below the median (2, 3, 5) each cost +1 more = +3 total
 - ▶ The 2 sticks above the median (8, 9) each cost -1 less = -2 total
 - ▶ Net change: +3 - 2 = +1 (cost increased!)
- Move left (target = 4): Cost = $|2 - 4| + |3 - 4| + |5 - 4| + |8 - 4| + |9 - 4|$
 $= 2 + 1 + 1 + 4 + 5 = \mathbf{13}$
 - ▶ The 2 sticks below the median (2, 3) each cost -1 less = -2 total
 - ▶ The 3 sticks above the median (5, 8, 9) each cost +1 more = +3 total
 - ▶ Net change: -2 + 3 = +1 (cost increased!)

Notice that moving away from the median in either direction increases the total cost because there are more sticks on one side that get penalized than sticks on the other side that benefit.

The algorithm works as follows:

- Sort the array to easily access the median
- Choose the middle element (for odd n) or either middle element (for even n) as the target
- Calculate the sum of absolute differences between each stick and the target

For even-length arrays, any value between the two middle elements works equally well, but using one of the middle elements is simplest.

Code:

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

C++

```

3
4  int main() {
5      int n;
6      cin >> n;
7
8      vector<int> v(n);
9      for (int i = 0; i < n; i++)
10         cin >> v[i];
11
12     // Sort the array in ascending order
13     sort(v.begin(), v.end());
14
15     // Choose the middle element (median) as the central value
16     int c = v[v.size() / 2];
17
18     long long ans = 0;
19     // Calculate the total distance of all elements from the median
20     for (int i = 0; i < n; i++)
21         ans += abs(v[i] - c);
22
23     // Output the minimum total distance
24     cout << ans;
25 }

```

2.2.10. Missing Coin Sum

[Question - Missing Coin Sum](#)

[Backup Link](#)

Hint:

Solution:

By sorting the coins in non-decreasing order, we can process them greedily. The greedy approach is as follows:

Initialize a variable `sumSoFar` to 0, representing the maximum sum we can create with the coins processed so far.

For each coin value `currCoin` : If `currCoin` is greater than `sumSoFar + 1`, it means we cannot create the sum `sumSoFar + 1` (since all remaining coins are too large). Thus, `sumSoFar + 1` is the answer. Otherwise, add `currCoin` to `sumSoFar`, as we can now create all sums up to `sumSoFar + currCoin` by including or excluding `currCoin` in subsets.

If we process all coins without finding a gap, the smallest sum we cannot create is `sumSoFar + 1`.

This works because we can create all sums from 0 to `sumSoFar`, and if the next coin `currCoin` is at most `sumSoFar + 1`, we can extend the range of creatable sums to `sumSoFar + currCoin`.

A gap occurs when a coin is too large to fill the next sum (`sumSoFar + 1`), making that sum impossible to form.

Code :

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5
6      int n;
7      cin >> n;
8
9      vector<long long> coins(n);
10     for (int i = 0; i < n; i++) cin >> coins[i];
11     sort(coins.begin(), coins.end());
12
13     long long sumSoFar = 0;
14     // We can currently form all sums from 1 to sumSoFar.
15
16     for (long long currCoin : coins) {
17         // If the next needed sum is sumSoFar + 1 and currCoin is bigger,
18         // we cannot fill that gap.
19         if (currCoin > sumSoFar + 1) {
```

C++

```
20         cout << sumSoFar + 1 << "\n";
21         return 0;
22     }
23
24     // Otherwise, currCoin helps us extend reachable sums.
25     sumSoFar += currCoin;
26 }
27
28 // If all coins processed and no gap found, next unreachable sum is
29 // sumSoFar + 1.
29 cout << sumSoFar + 1 << "\n";
30 return 0;
31 }
```

2.2.11. Collecting Numbers

[Question - Collecting Numbers](#)

[Backup Link](#)

Solution:

The program stores the index of each number in the order it appears. It then scans numbers from 1 to n and checks whether a number appears before its predecessor. Whenever this happens, a new round is required. The final count represents the total number of rounds needed.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5
6      int n;
7      cin >> n;
8
9      vector<int> position(n + 1);
10     for (int i = 0; i < n; i++) {
11         int value;
12         cin >> value;
13         position[value] = i;
14     }
15
16     int rounds = 1;
17     for (int i = 2; i <= n; i++) {
18         if (position[i] < position[i - 1]) { //if the larger number
19             rounds++;
20         }
21     }
22
23     cout << rounds;
```

C++

2.2.12. Collecting Numbers II

[Question - Collecting Numbers II](#) [Backup Link](#)

Solution

This problem works with a permutation of numbers from 1 to n and asks how many rounds are needed to collect the numbers in increasing order. A new round is required whenever the position of a number x appears after the position of $x+1$ in the array. Initially, we scan the array and count how many such “breaks” exist to compute the number of rounds.

For each query, two positions in the array are swapped. A full recount after every swap would be too slow, so the key idea is to only update the parts of the array that are affected. Swapping two values only changes the order relations involving those values and their immediate neighbors ($x-1$, x , $x+1$). Before performing the swap, we subtract any existing breaks caused by these values. After the swap, we recompute and add back the new breaks.

By maintaining an array that stores the current position of each value, each check can be done in constant time. This allows every query to be processed efficiently, keeping the total complexity fast even for large inputs.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int n, m;
5  vector<int> arr;    // Stores the permutation
6  vector<int> pos;    // pos[x] = index where value x is currently located
7
8  // Checks whether x and x+1 form a "break"
9  // A break means x appears after x+1 in the array
10 bool isBreak(int x) {
11     if (x < 1 || x >= n) return false;    // Out of valid range
12     return pos[x] > pos[x + 1];
13 }
14
15 int main() {
16
17     cin >> n >> m;
18
19     arr.resize(n);
20     pos.resize(n + 2);
21
22     // Read the permutation and record positions of each value
23     for (int i = 0; i < n; i++) {
```

C++


```

24     cin >> arr[i];
25     pos[arr[i]] = i;
26 }
27
28 // Initially, at least one round is needed
29 int rounds = 1;
30
31 // Count how many times x comes after x+1
32 // Each such case increases the number of rounds
33 for (int x = 1; x < n; x++) {
34     if (isBreak(x)) {
35         rounds++;
36     }
37 }
38
39 // Process each swap query
40 while (m--) {
41     int a, b;
42     cin >> a >> b;
43     a--;
44     b--; // Convert to 0-based indexing
45
46     int u = arr[a];
47     int v = arr[b];
48
49     // Only these values can affect the number of breaks
50     // because other relative orders remain unchanged
51     set<int> affected = {
52         u - 1, u, u + 1,
53         v - 1, v, v + 1
54     };
55
56     // Remove old breaks before swapping
57     for (int x : affected) {
58         if (isBreak(x)) {
59             rounds--;
60         }
61     }
62
63     // Perform the swap in the array
64     swap(arr[a], arr[b]);
65
66     // Update positions of the swapped values
67     swap(pos[u], pos[v]);
68

```

```
69         // Add new breaks after swapping
70         for (int x : affected) {
71             if (isBreak(x)) {
72                 rounds++;
73             }
74         }
75
76         // Output the current number of rounds
77         cout << rounds << "\n";
78     }
79
80     return 0;
81 }
```

2.2.13. Playlist

[Question - Playlist](#)

[Backup Link](#)

Solution:

The trick is to slide a window across the array while keeping all its elements distinct. A set tracks which songs are currently inside the window: if the next song is already present, we shrink the window from the left until the duplicate disappears. Otherwise we extend the window to include it. As the window grows and shrinks, we keep updating the maximum length, which becomes the length of the longest playlist with all unique songs.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      // Read the array
9      vector<int> v(n);
10     for (int i = 0; i < n; i++) {
11         cin >> v[i];
12     }
13
14     // Sliding window to find the longest subarray with all distinct
    elements.
15     set<int> window;    // stores current distinct elements inside the
    window
16     int left = 0;       // left pointer of the window
17     int right = 0;      // right pointer of the window
18     int bestLen = 0;    // maximum size found
19
20     // Expand the window using 'right'
21     while (right < n) {
22         // If v[right] already exists, shrink the window from the left
23         // until v[right] can be inserted without duplicates.
24         if (window.count(v[right])) {
25             window.erase(v[left]);
26             left++;
27         }
28         else {
29             // Element is unique in the window – include it
```

C++

```
30         window.insert(v[right]);
31
32         // Update max length
33         bestLen = max(bestLen, right - left + 1);
34
35         // Move right pointer forward
36         right++;
37     }
38 }
39
40 cout << bestLen;
41 return 0;
42 }
```

2.2.14. Towers

[Question - Towers](#) [Backup Link](#)

Solution:

The idea is to maintain the top blocks of all towers in a **multiset**. For each new block, place it on the leftmost tower whose top is strictly greater; if no such tower exists, you start a new one. This greedy strategy works because always using the smallest possible valid tower keeps future placements flexible. The number of towers equals the size of the **multiset**.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  using ll = long long;
4
5  int main() {
6
7      int n;
8      cin >> n;
9
10     multiset<int> tops; // stores the current top element of each tower
11
12     for (int i = 0; i < n; i++) {
13         int x;
14         cin >> x;
15
16         // Find first tower whose top > x (we can place x on that tower)
17         auto it = tops.upper_bound(x); // multiset<int>::iterator is the type
            of auto
18
19         if (it != tops.end()) {
20             // Reuse this tower: remove old top and replace with x
21             tops.erase(it);
22         }
23         // Start a new tower or update reused one with top = x
24         tops.insert(x);
25     }
26
27     // Number of towers equals the number of distinct tops
28     cout << tops.size() << "\n";
29
30     return 0;
31 }
```

C++

2.2.15. Traffic Lights

[Question - Traffic Lights](#) [Backup Link](#)

Solution:

The program simulates cutting a stick of length `a` at `b` given positions. It uses a `set` to store all the positions of the traffic lights and a `multiset` to track road segment lengths without a traffic light. After each cut that a traffic light makes, it removes the old segment and adds two new ones. Finally, it prints the length of the largest segment remaining after each cut.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int a, b, x;
6      cin >> a >> b;
7
8      set<int> trafficLights; // trafficLights stores the positions of the
                             // traffic lights
9      multiset<int> lengths; // lengths stores all the road segment lengths
                             // without a traffic light
10     trafficLights.insert(a); // rightmost boundary
11     trafficLights.insert(0); // leftmost boundary
12     lengths.insert(a); // initially one segment of length 'a'
13
14     for (int i = 0; i < b; i++) {
15         cin >> x;
16
17         // Insert the new cut position and find its neighbors
18         auto mid = trafficLights.insert(x); // auto is set<int>::iterator
19         auto first = prev(mid); // auto is multiset<int>::iterator
20         auto last = next(mid); // auto is multiset<int>::iterator
21
22         // Remove the old segment and add the two new smaller segments
23         lengths.erase(lengths.find(*last - *first));
24         lengths.insert(*last - *mid);
25         lengths.insert(*mid - *first);
26
27         // Output the largest segment length after each cut
28         cout << *lengths.rbegin() << " "; // rbegin() is a reverse iterator at
                             // the last element.
29     }
30 }
```

2.2.16. Distinct Values Subarrays

[Question - Distinct Values Subarrays](#) [Backup Link](#)

Solution:

This code uses a sliding window to count subarrays with all distinct elements. The right pointer expands the window, while a frequency map¹⁴ tracks duplicates. If a duplicate appears, the left pointer shrinks the window until all elements are unique again. At each position, the number of valid subarrays ending there is added to the answer.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      vector<int> a(n);
9      for (int i = 0; i < n; i++) {
10         cin >> a[i];
11     }
12
13     // Frequency map for elements in the current window
14     map<int, int> freq;
15
16     int left = 0;           // Left pointer of the sliding window
17     long long answer = 0;   // Total number of valid subarrays
18
19     for (int right = 0; right < n; right++) {
20         freq[a[right]]++; // Add current element to the window
21
22         // Shrink window until all elements are distinct
23         while (freq[a[right]] > 1) {
24             freq[a[left]]--;
25             left++;
26         }
27
28         // Number of distinct subarrays ending at 'right'
29         answer += (right - left + 1);
30     }
```

¹⁴See Section 2.1.8

```

31
32     cout << answer << "\n";
33 }

```

Alternate Solution:

You can use a `set` instead of a `map`. This works by storing all elements in the current window in the `set`, when a duplicate of the element at the right pointer is found, you move the left pointer of the window and keep removing elements from the `set` until the duplicate element is removed, then add the element at the right pointer to the window.

Code:

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main(){
5      //Accepting Input:
6
7      int n;
8      cin >> n;
9
10     vector<int> v(n);
11     for(int i = 0; i < n; i++)
12         cin >> v[i];
13
14     //Computing Answer:
15     long long ans = 0;
16     set<int> s; //stores the elements in the current window
17     for(int l = 0, r = 0; r < n; r++){
18         //The while removes keeps removing elements from the window until the
19         //duplicate of v[r] is removed.
20         while(s.find(v[r]) != s.end()){
21             s.erase(v[l]);
22             l++;
23         }
24         s.insert(v[r]); //and the current element to the window
25         ans += r-l+1; //number of subarrays ending at r
26     }
27
28     cout << ans << "\n";
29     return 0;
30 }

```

C++

2.2.17. Distinct Values Subsequences

[Question - Distinct Values Subsequences](#) [Backup Link](#)

Solution:

For each distinct value with `occ` occurrences, we have `(occ + 1)` choices: exclude it (0 copies) or choose 1 of the `occ` identical copies to include. Multiplying choices for all distinct numbers gives total possible combinations including the empty subsequence. Subtract 1 to remove the empty subsequence case, leaving the count of all distinct-value subsequences.

Example:

For the array `{1, 3, 5, 2, 9, 3, 2}`

The frequency table stores:

Key	Value
1	1
2	2
3	2
5	1
9	1

The number of distinct value subsequences is:

$$\begin{aligned} &= (1 + 1) \cdot (2 + 1) \cdot (2 + 1) \cdot (1 + 1) \cdot (1 + 1) \\ &= 2 \cdot 3 \cdot 3 \cdot 2 \cdot 2 \\ &= 72 \end{aligned}$$

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  using ll = long long;
5  const int MOD = 1e9 + 7;
6
7  int main() {
8
9      int n;
10     cin >> n;
11
12     // Count frequency of each distinct value
13     map<ll, ll> freq;
14     for (int i = 0; i < n; i++) {
15         int x;
```

C++

```

16         cin >> x;
17         freq[x]++;
18     }
19
20     // For each distinct value, we can include 0, 1, 2, ..., or occ copies
21     // This gives (occ + 1) choices per value; multiply all choices together
22     ll ans = 1;
23     for (auto [num, occ] : freq) {
24         ans = (ans * (occ + 1)) % MOD;
25     }
26
27     // Subtract 1 to exclude the empty subsequence
28     // The reason for adding MOD before taking the modulo is because in c++,
29     // a number less than 0 can give a negative remainder which is bad for
30     // future calculation. Hence you make sure it's possible by adding MOD.
31     ans = (ans - 1 + MOD) % MOD;
32     cout << ans << "\n";
33 }

```

2.2.18. Josephus Problem I

[Question - Josephus Problem I](#) [Backup Link](#)

Solution:

We store all people in a linked list¹⁵, for efficient deletions while moving forward. An iterator walks through the list, skipping one person each time. When the iterator reaches the end, it wraps back to the beginning. Each erased element is printed in order.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      list<int> circle;
9      for (int i = 1; i <= n; i++)
10         circle.push_back(i);
11
12     auto it = circle.begin();//auto is list<int>::iterator
13
14     while (!circle.empty()) {
15         // move to the next person (skip one)
16         it++;
17         if (it == circle.end())//circle back
18             it = circle.begin();
19
20         cout << *it << " ";
21
22         // erase returns iterator to next element
23         it = circle.erase(it);
24
25         if (it == circle.end() && !circle.empty())//circle back
26             it = circle.begin();
27     }
28
29     return 0;
30 }
```

¹⁵See Section 2.1.5

2.2.19. Josephus Problem II

[Question - Josephus Problem II](#) [Backup Link](#)

Hint:

If you've thought about the question for a while, you would have realised that you need the ability to do the following operations efficiently (i.e $O(\log n)$):

1. Jump from a certain number to the next number k places ahead.
2. Delete the element at the current index.

Section 2.1.4 explains the data structure known as a Fenwick tree which gives you the ability to do these operations.

Solution:

We can use the Fenwick Tree (Binary Indexed Tree)¹⁶ as an indexed set to efficiently jump by any amount k and remove elements.

Set every element from 1 to n in the fenwick tree to 1 to indicate they are all in the list 1 time. Start at index 0, then jump k places mod n so it's circular, use the `search()` function to find what number is at that index + 1 (Fenwick Trees are 1 indexed) and remove it by subtracting its frequency by 1.

The time complexity is $O(n \log n)$: n removals, each taking $O(\log n)$ for searching and removing.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int n;
5  vector<int> fenw;
6
7  void add(int x, int k){
8      for(; x <= n ; x += x & -x) // x & -x is the LSSB(x)
9          fenw[x] += k;
10 }
11
12 int search(int idx){
13     int ans = 0;
14
15     for(int k = floor(log2(n)); k >= 0; k--){ //go through the powers of 2.
16         if(ans + (1 << k) <= n && fenw[ans + (1 << k)] < idx){ //this element is
            before the idx.
17             ans += 1 << k; //update the answer.
        }
```

C++

¹⁶See Section 2.1.4

```

18     idx -= fenw[ans]; //account for all indices upto fenw[ans].
19 }
20 }
21
22     return ans + 1; //ans was the value that was before idx, so one value ahead
    of that is at idx.
23 }
24
25 int main(){
26
27     int k;
28     cin >> n >> k;
29     fenw.resize(n + 1); //allocating memory to the fenwick tree;
30
31     for (int i = 1; i <= n; i++)
32         add(i, 1); //increase the frequency of all numbers by 1
33
34     for(int rem = n, idx = 0; rem > 0; rem--){ //rem is the people remaining in
    the circle
35         // Move k steps forward in circular manner
36         idx = (idx + k) % rem;
37
38         // Find the number at index idx + 1 (+1 because a fenwick tree 1 one
    indexed)
39         int pos = search(idx + 1);
40         cout << pos << " ";
41
42         // Mark this number as removed by decreasing frequency to 0
43         add(pos, -1);
44     }
45
46     return 0;
47 }

```

2.2.20. Nested Ranges Check

[Question - Nested Ranges Check](#) [Backup Link](#)

Hint:

Try sorting the intervals in ascending order of left index and descending order of right index. What algorithm could you come up with where you only have to iterate once through the list to get the answer? Think about why this sorting method was mentioned and what advantages it has.

Solution:

For detecting if a range is “contained” by another range, iterate in the forward direction (left to right) tracking the maximum right endpoint of a range seen so far. If the current range’s right endpoint is less than or equal to the maximum, it means this range is contained by a previous one because the previous ranges will also have a left endpoint less than the current.

For detecting if a range “contains” another range, iterate backwards (right to left) tracking the minimum right endpoint. If the current range’s right endpoint is greater than or equal to the minimum, it contains at least one subsequent range because the current ranges left endpoint will be lesser than the subsequent one.

The time complexity is $O(n \log n)$.

For a deeper explanation to the problem, see the solution to the next question.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  struct Range{
5      int l, r, idx;
6
7      bool operator<(const Range& ran){//Operator overloading so we can get the
        correct sorting order.
8          return l < ran.l || l == ran.l && r > ran.r;
9      }
10 };
11
12 int main() {
13
14     int n;
15     cin >> n;
16     vector<Range> ranges(n);
17     for (int i = 0; i < n; i++) {
18         cin >> ranges[i].l >> ranges[i].r;
19         ranges[i].idx = i;
```

C++

```

20     }
21
22     sort(ranges.begin(), ranges.end());
23     vector<int> contains(n, 0), contained(n, 0);
24
25     int maxRight = INT_MIN;
26     for (int i = 0; i < n; i++) {
27         if (ranges[i].r <= maxRight)
28             contained[ranges[i].idx] = 1;
29         maxRight = max(maxRight, ranges[i].r);
30     }
31     int minRight = INT_MAX;
32     for (int i = n - 1; i >= 0; i--) {
33         if (ranges[i].r >= minRight)
34             contains[ranges[i].idx] = 1;
35         minRight = min(minRight, ranges[i].r);
36     }
37     for (int x : contains)
38         cout << x << " ";
39     cout << "\n";
40
41     for (int x : contained)
42         cout << x << " ";
43     cout << "\n";
44 }

```

2.2.21. Nested Ranges Count

[Question - Nested Ranges Count](#) [Backup Link](#)

Hint:

Try sorting the intervals in ascending order of left index and descending order of right index. What algorithm could you come up with where you only have to iterate once through the list to get the answer? Think about why this sorting method was mentioned and what advantages it has.

Solution:

The problem asks us to find two values for each range: the number of other ranges it contains, and the number of other ranges that contain it. A naive solution comparing every pair would be too slow ($O(n^2)$). We'll need a better approach.

Say we were to sort every interval in ascending order of their left index and in descending order of their right index. This seems pretty random, however if you think about the first range in the sorted list, you can guarantee that any range that comes after it will be inside it simply by checking if its right index is less than the right index of the first range.

This applies to all ranges in the sorted list i.e. for any range i , any subsequent range j will be contained by i if the right index of range j is less than i .

Likewise, the converse also applies, for any range i , the **count** of ranges that i is **contained** by is all ranges j , where $j < i$, such that right index of j is more than right index of i .

Let's look at an example to understand this better:

Say we're given the following ranges, for now they're going to be sorted in the order we want

$$\{\{1, 10\}, \{2, 8\}, \{2, 6\}, \{5, 7\}\}$$

Now the first range cannot be contained by anyone, because so the answer for range 0 is 0.

Range 1 is contained by Range 0 because $10 > 8$, so the answer for range 1 is 1.

Range 2 is contained by both range 0 and 1 because $10 > 6$ and $8 > 6$. Therefore the answer for range 2 is 2.

Lastly range 3 is contained by ranges 0 and 1 but **not** by range 2 because $10 > 7$, $8 > 7$, but $6 < 7$. Therefore the answer for range 3 is 2.

The output of the second line for the question hence would be 0, 1, 2, 2

This however only ends up solving half the question, we still need to find out for every range i , how many ranges it contains.

For this we can use the same sorting order, just iterate in reverse. The reasoning is the same as before: for any range i , the **count** of ranges that i **contains** is the number of ranges j , where $j > i$ such that right index of j is less than the right index of i .

Using the ranges given previously, range 3 can contain no other ranges, therefore its answer is 0.

Range 2 does **not** contain Range 3 because $7 > 6$. Therefore its answer is 0.

Range 1 contains both range 2 and range 3 because $6 < 8$ and $7 < 8$. Therefore its answer is 2.

Lastly range 0 contains range 1, range 2, and range 3, because $8 < 10$, $6 < 10$, and $7 < 10$. Therefore its answer is 3.

While the approach seems logical, how can we efficiently find out how many ranges have a right end point less than the current range or greater than the current range. For that, we can use a Fenwick tree as an indexed set. You can add the right index of all ranges either succeeding or preceding and then find the how many right indexes are more or less than the current range.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  vector<int> fen;
5
6  struct Range{
7      int l, r, idx;
8
9      bool operator<(const Range& ran){//operator overloading for the custom
      sorting.
10         return l < ran.l || l == ran.l && r > ran.r;
11     }
12 };
13
14 void add(int x, int k){
15     for(; x < fen.size(); x += x & -x)
16         fen[x] += k;
17 }
18
19 int sum(int x){
20     int ans = 0;
21     for(; x > 0; x -= x & -x)
22         ans += fen[x];
23     return ans;
24 }
25
26 int main(){
27     ios_base::sync_with_stdio(0);
28     cin.tie(0);
29     cout.tie(0);
30     //freopen("NestedRangesCount.in", "r", stdin);
31     //freopen("NestedRangesCount.out", "w", stdout);
32     int n;
33     cin >> n;
```

C++

```

34
35     vector<Range> v(n);
36     vector<int> comp;
37
38     for(int i = 0; i < v.size(); i++){
39         cin >> v[i].l >> v[i].r;
40         v[i].idx = i;
41         comp.push_back(v[i].r);
42     }
43
44     //Index compression for the right endpoints:
45     sort(comp.begin(), comp.end());
46     comp.erase(unique(comp.begin(), comp.end()), comp.end());
47
48     for(int i = 0; i < v.size(); i++)
49         v[i].r = lower_bound(comp.begin(), comp.end(), v[i].r) - comp.begin() +
50         1;
51
52     sort(v.begin(), v.end());
53     fen.resize(comp.size() + 1);
54
55     vector<int> c(n), d(n); // c[i] is the number of ranges v[i] contains, d[i]
56     is the number of ranges that v[i] is contained by.
57
58     for(int i = v.size()-1; i >= 0; i--){
59         // for every range, add the number of ranges with r < v[i].r. l > v[i].r
60         // because of the sorting order.
61         // if l = v[i].l , then the smaller ranges will get added first to
62         // correctly find c[i] because of the sorting order.
63         // sum(v[i].r) - 1 gives that number.
64         add(v[i].r, 1);
65         c[v[i].idx] = sum(v[i].r) - 1;
66     }
67
68     //Resetting the Fenwick tree.
69     fen.clear();
70     fen.resize(comp.size() + 1);
71
72     for(int i = 0; i < v.size(); i++){
73         // for every range, add the number of ranges with r > v[i].r. l < v[i].l
74         // because of the sorting order.
75         // if l = v[i].l , then the larger ranges will get added first to
76         // correctly find d[i] because of the sorting order.
77         // sum(fen.size() - 1) - sum(v[i].r-1) - 1 gives that number.
78         add(v[i].r, 1);
79         d[v[i].idx] = sum(fen.size()-1) - sum(v[i].r-1) - 1;

```

```
74     }
75
76     for(int i = 0; i < c.size(); i++)
77         cout << c[i] << " ";
78     cout << endl;
79
80     for(int i = 0; i < d.size(); i++)
81         cout << d[i] << " ";
82     cout << endl;
83
84     return 0;
85 }
```

2.2.22. Room Allocation

[Question - Room Allocation](#)

[Backup Link](#)

Solution:

We can sort the customers by arrival time. For each customer, we check if any previously used room has become free (i.e., its last guest departed before the current guest arrives). We use a multiset to efficiently track rooms by their end times: if a room's end time is less than the current customer's arrival time, we reuse it; otherwise, we allocate a new room. This greedy choice is optimal because assigning an available room to the earliest arriving customer never leads to a worse solution than leaving it empty.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      // Store each customer's booking: {start_time, end_time, original_index}
9      vector<tuple<int, int, int>> bookings(n);
10
11     for (int i = 0; i < n; i++) {
12         int start, end;
13         cin >> start >> end;
14         bookings[i] = {start, end, i};
15     }
16
17     // Sort bookings by start time
18     sort(bookings.begin(), bookings.end());
19
20     // Track available rooms: {end_time, room_number}
21     // Sorted by end_time to find rooms that become free earliest
22     multiset<pair<int, int>> availableRooms;
23
24     vector<int> assignedRoom(n);
25     int totalRooms = 0;
26
27     for (const auto& [start, end, originalIndex] : bookings) {
28         int roomNumber;
29
30         // Find a room that's free before this booking starts
31         // upper_bound finds first room with end_time > start
```

C++

```

32     // We want the room with end_time <= start, so we go one back
33     auto it = availableRooms.upper_bound({start, INT_MAX}); //auto is
multiset<pair<int,int>>::iterator
34
35     if (it == availableRooms.begin()) {
36         // No available room found - need a new room
37         roomNumber = ++totalRooms;
38     } else {
39         // Reuse an existing room that's now free
40         --it;
41         roomNumber = it->second;
42         availableRooms.erase(it);
43     }
44
45     // Mark this room as occupied until 'end' time
46     availableRooms.insert({end, roomNumber});
47     assignedRoom[originalIndex] = roomNumber;
48 }
49
50 // Output results
51 cout << totalRooms << "\n";
52 for (int room : assignedRoom) {
53     cout << room << " ";
54 }
55 cout << "\n";
56
57 return 0;
58 }

```

2.2.23. Factory Machines

[Question - Factory Machines](#)

[Backup Link](#)

Hint:

Observe that the amount of items the machines in total produce increase consistently with time(i.e monotonically). You can also in $O(n)$ compute the amount of items the machines can make in a given amount of time. Think about how you could use this to find the minimum time to make t items.

Solution:

The key idea is to use binary-search on answers. This means you assume the answer(minimum time to make t items) is some time mid . You then compute how many items can be made in time mid by summing up $mid/v[i]$ for each machine. If the amount of products you made is greater than or equal to t , you try a smaller time by halving the range towards smaller values. If you made less than t products, you try a larger time by halving the range towards larger values. This guarantees we find the earliest moment when production meets the target.

Code :

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  using ll = long long;
4
5  int main() {
6
7      ll n, t;
8      cin >> n >> t;
9
10     // v[i] = time taken by machine i to produce ONE item
11     vector<ll> v(n);
12     for (int i = 0; i < n; i++) {
13         cin >> v[i];
14     }
15
16     // Binary search on time.
17     // low = minimum possible time
18     // high = a very large upper bound (1e18 works for all constraints)
19     ll low = 1, high = 1e18, ans = -1;
20
21     while (low <= high) {
22         ll mid = (low + high) / 2;
23     }
```

C++

```

24         // Count how many items all machines can produce in 'mid' time
25         ll total = 0;
26         for (int i = 0; i < n; i++) {
27             total += mid / v[i];
28
29             // If already enough, stop early (avoid overflow + speedup)
30             if (total >= t) break;
31         }
32
33         // If we can produce at least t items in 'mid' time,
34         // try to find an even smaller valid time by halving the range
35         if (total >= t) {
36             ans = mid;
37             high = mid - 1;
38         }
39         else {
40             // Need more time, so halve the range towards larger values.
41             low = mid + 1;
42         }
43     }
44
45     cout << ans << "\n";
46     return 0;
47 }

```

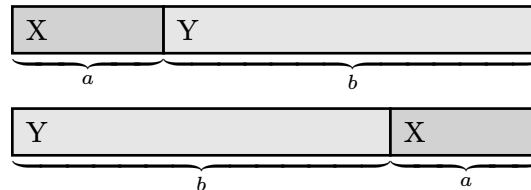
2.2.24. Tasks and Deadlines

[Question - Tasks and Deadlines](#)

[Backup Link](#)

Solution:

The greedy approach is to always prioritize tasks with the shortest durations first, because choosing a long task early delays all subsequent tasks and reduces their rewards by a larger amount. Say we have 2 tasks X and Y arranged in the following 2 arrangements:



If you compare the change in score from the first ordering to the second ordering: in the second ordering, Y gets completed a units of time earlier so the score of Y increases by a . However, X gets completed b units of time later, so the score of X decreases by b . The net change in score is $a - b < 0$ which is lesser than it was in the first ordering. Hence you must sort the tasks by duration to ensure the maximum score.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  using ll = long long;
4
5  int main() {
6      int n;
7      cin >> n;
8
9      // jobs[i] = {duration, deadline}
10     vector<pair<int, int>> jobs(n);
11     for (int i = 0; i < n; i++) {
12         cin >> jobs[i].first >> jobs[i].second;
13     }
14
15     // Sort by duration (or by first element), ensures earliest finishing
    // attempts first
16     sort(jobs.begin(), jobs.end());
17
18     ll time_elapsed = 0; // running sum of durations
19     ll total_reward = 0; // accumulated reward
20
21     for (int i = 0; i < n; i++) {
22         time_elapsed += jobs[i].first; // finish this job at
            this time
```

C++


```
23         total_reward += jobs[i].second - time_elapsed; // reward = deadline
                - completion time
24     }
25
26     cout << total_reward << "\n";
27     return 0;
28 }
```

2.2.25. Reading Books

[Question - Reading Books](#)

[Backup Link](#)

Solution:

If one book takes more than half the total time, one child will be forced to wait while the other finishes that long book. Otherwise, they can optimally interleave their reading with no idle time. Thus, the answer is $\max(\text{total_time}, 2 * \text{longest_book})$.

The rigorous reason for why it's possible to optimally interleave their reading with no idle time is as follows:

Let b_1, b_2, \dots, b_n be the n books in order of smallest to largest.

Let person one read the books in order of largest to smallest : $b_n, b_{n-1}, b_{n-2}, \dots, b_1$

Let person two read the books in the same order except the read the longest book b_n at the end : $b_{n-1}, b_{n-2}, \dots, b_1, b_n$

Person one will never catchup to the same book as person two because they are always going to be reading a book that is at least as long and the book person two is reading. The only time a conflict arises is when person 2 goes to read b_n .

If $b_n \leq \text{sum} - b_n$, then person one would've moved on before person two comes around to reading b_n ; otherwise person 2 will have to wait for person one to finish. The inequality can be rearranged as $2 \cdot b_n \leq \text{sum}$ which when true results in an answer of sum , else the answer is $2 \cdot b_n$.

Code :

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;
7
8      vector<long long> books(n);
9      long long total = 0, max_book = 0;
10
11     // Read book times, track:
12     // 1) total time of all books
13     // 2) the longest single book
14     for (int i = 0; i < n; i++) {
15         cin >> books[i];
16         total += books[i];
17         max_book = max(max_book, books[i]);
18     }
```

C++

```
19
20 // Minimum total time is governed by:
21 // - total (one person reading sequentially)
22 // - OR twice the largest book (two-person parallel reading constraint)
23 // The answer is the max of these two.
24 cout << max(total, 2 * max_book) << "\n";
25
26 return 0;
27 }
```

2.2.26. Sum of Three Values

[Question - Sum of Three Values](#) [Backup Link](#)

Solution:

This question is an extension of Sum of Two Values¹⁷. We can pick one number $v[i]$ and then seeing if there are any remainder 2 values $v[l]$, $v[r]$ in the range $i+1$ to $n-1$ such that $v[i] + v[l] + v[r] = \text{target}$. The remainder 2 values are found by using the same 2 pointer approach as shown in the previous question.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, target;
6      cin >> n >> target;
7
8      // Store {value, original_index}
9      vector<pair<int, int>> v(n);
10
11     for (int i = 0; i < n; i++) {
12         cin >> v[i].first;
13         v[i].second = i + 1; // keep original 1-based index
14     }
15
16     // Sort by value so we can use the two-pointer trick
17     sort(v.begin(), v.end());
18
19     // Fix one element v[i], then find two others using two pointers
20     for (int i = 0; i < n - 2; i++) {
21         int l = i + 1; // left pointer
22         int r = n - 1; // right pointer
23
24         while (l < r) {
25             int sum = v[i].first + v[l].first + v[r].first;
26
27             if (sum == target) {
28                 // Output original positions (not sorted ones)
29                 cout << v[i].second << " " << v[l].second << " " <<
30                     v[r].second;
31                 return 0;
32             }
33         }
34     }
```

¹⁷See Section 2.2.7

```

32         else if (sum < target) {
33             l++;          // need a larger sum -> move left pointer right
34         }
35         else {
36             r--;          // need a smaller sum -> move right pointer left
37         }
38     }
39 }
40
41 // If no triple found
42 cout << "IMPOSSIBLE";
43 return 0;
44 }

```

2.2.27. Sum of Four Values

[Question - Sum of Four Values](#) [Backup Link](#)

Solution:

This question is an extension of Sum of Three Values¹⁸. We can pick two numbers $v[i]$, $v[j]$ then seeing if there are any remainder 2 values $v[l]$, $v[r]$ in the range $i+1$ to $n-1$ such that $v[i] + v[l] + v[r] = \text{target}$. The remainder 2 values are found by using the same 2 pointer approach as shown in the previous question.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, target;
6      cin >> n >> target;
7
8      // Store (value, original_index)
9      vector<pair<int, int>> nums(n);
10
11     for (int i = 0; i < n; i++) {
12         cin >> nums[i].first;
13         nums[i].second = i + 1; // 1-based indexing as required by CSES
14     }
15
16     // Sort by value to enable two-pointer scanning later
17     sort(nums.begin(), nums.end());
18
19     // Fix first two values using indices i and j
20     for (int i = 0; i < n - 3; i++) {
21         for (int j = i + 1; j < n - 2; j++) {
22
23             int left = j + 1;
24             int right = n - 1;
25
26             // Two-pointer search for remaining pair
27             while (left < right) {
28                 long long sum = nums[i].first
29                     + nums[j].first
30                     + nums[left].first
31                     + nums[right].first;
```

¹⁸See Section 2.2.7

```

32
33         if (sum == target) {
34             // Output original positions
35             cout << nums[i].second << " "
36             << nums[j].second << " "
37             << nums[left].second << " "
38             << nums[right].second;
39             return 0;
40         }
41
42         // Adjust pointers based on sum size
43         if (sum < target) {
44             left++;
45         } else {
46             right--;
47         }
48     }
49 }
50 }
51
52 cout << "IMPOSSIBLE";
53 return 0;
54 }

```

2.2.28. Nearest Smaller Values

[Question - Nearest Smaller Values](#) [Backup Link](#)

Solution:

We can use a stack¹⁹ of pairs (value, index) that stores the previously seen values that are less than the current value x in ascending order. This is achieved by first popping all elements greater than x and then either outputting the index at the top of the stack (`s.top().second`) or if the stack is empty, outputting zero. Finally push x into the stack.

The index at the top of the stack is guaranteed to be the closest value less than x because any other values that are less than x which occurred earlier than the answer were pushed in first and hence will be lower in the stack than the answer.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ll n, a;
6      cin >> n;
7      stack<pair<int,int>> s; // Stores pairs of (value, index) in sorted order
                             // by value
8
9      for (int idx = 1; idx <= n; idx++) {
10         cin >> x;
11
12         while(!s.empty() && s.top().first >= x) //remove all elements >= x
13             s.pop();
14
15         if(s.empty()) //if there are no elements < x
16             cout << 0 << " ";
17         else //output the element with the idx closest to x i.e the top of the
            stack
18             cout << s.top().second << " ";
19
20         s.push({x,idx});
21     }
22
23     return 0;
24 }
```

¹⁹See Section 2.1.9

2.2.29. Subarray Sums I

[Question - Subarray Sums I](#) [Backup Link](#)

Solution:

Because all numbers are positive adding or removing an element to the current subarray will increase or decrease the sum respectively. This lets us use a sliding window to keep track of the current sum of the subarray.

For every element r , we first increase the size of the current subarray from $l, r-1$ to l, r and increase the sum by $v[r]$. If this sum is greater than the target x , we shrink the window from the left and remove those elements from the sum until the sum is less than or equal to x . If the sum is equal to x we increase our answer by 1.

Code :

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  using ll = long long;
4
5  int main() {
6      ll n, x;
7      cin >> n >> x;
8
9      vector<int> v(n);
10     for (int i = 0; i < n; i++)
11         cin >> v[i];
12
13     ll sum = 0; // current window sum
14     ll ans = 0; // number of subarrays equal to x
15     ll l = 0; // left pointer of sliding window
16
17     for (int r = 0; r < n; r++) {
18         sum += v[r]; // expand window to the right
19
20         // shrink window from the left while sum > x
21         while (sum > x) {
22             sum -= v[l];
23             l++;
24         }
25
26         // if current window sum = x, increase the answer by 1
27         if (sum == x) ans++;
28     }
29 }
```

C++

```
30     cout << ans << "\n";  
31 }
```

2.2.30. Subarray Sums II

[Question - Subarray Sums II](#) [Backup Link](#)

Solution:

We iterate through the array while maintaining a prefix sum. A map stores how many times each prefix sum has appeared so far. At each position, if a previous prefix sum equals $\text{currentSum} - \text{targetSum}$, a subarray with sum targetSum exists (The subarray starts 1 place after the previous prefix sum). We add the frequency of how often that previous prefix sum value occurred to the answer and then increase the frequency of the value of the current prefix sum.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  using ll = long long;
5
6  int main() {
7      ll n, targetSum;
8      cin >> n >> targetSum;
9
10     vector<ll> array(n);
11     for (int i = 0; i < n; i++) {
12         cin >> array[i];
13     }
14
15     // prefixSumCount[s] = how many times prefix sum with value s has
16     // occurred
17     map<ll, ll> prefixSumCount;
18     prefixSumCount[0] = 1; // empty prefix
19
20     ll currentSum = 0;
21     ll subarrayCount = 0;
22
23     for (int i = 0; i < n; i++) {
24         currentSum += array[i];
25
26         // count subarrays ending here with sum = targetSum
27         subarrayCount += prefixSumCount[currentSum - targetSum];
28
29         // increase the frequency of the value of the current prefix sum
```

C++

```
29         prefixSumCount[currentSum]++;  
30     }  
31  
32     cout << subarrayCount << "\n";  
33     return 0;  
34 }
```

2.2.31. Subarray Divisibility

[Question - Subarray Divisibility](#) [Backup Link](#)

Hint:

If you solved the question above (Subarray Sums II), try using the same technique of storing frequencies, but figure out what you should be storing the frequency of so that you can figure out a subarrays divisibility.

Solution:

We use prefix sums modulo n to count subarrays whose sum is divisible by n . Each element is first reduced modulo n to keep values small.

As we iterate, we maintain the current prefix sum modulo n . A map stores the frequency of each modulo value of the prefix sums. If the same modulo appears again, there exists a subarray that has its sum divisible by n (If the current remainder is n and some previous remainder is also n , then their difference is 0 which means it's divisible). We add the frequency of the current modulo to the answer and increase the frequency of that modulo in the map.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      long long n;
6      cin >> n;
7
8      vector<long long> arr(n);
9      for (int i = 0; i < n; i++) {
10         cin >> arr[i];
11         arr[i] %= n;           // keep values within modulo n
12     }
13
14     unordered_map<long long, long long> frequency;
15     frequency[0] = 1;         // empty prefix sum
16
17     long long prefixSum = 0;
18     long long result = 0;
19
20     for (int i = 0; i < n; i++) {
21         prefixSum = (prefixSum + arr[i]) % n;
22         if (prefixSum < 0) prefixSum += n;
23     }
```

C++

```
24         result += frequency[prefixSum];
25         frequency[prefixSum]++;
26     }
27
28     cout << result << "\n";
29     return 0;
30 }
```

2.2.32. Distinct Values Subarrays II

[Question - Distinct Values Subarrays II](#) [Backup Link](#)

Hint

If you've solved Distinct Values Subarrays²⁰, try using the same method but modifying it for the needs of this question.

Solution:

We can use a sliding window while storing the frequencies of every element in a `map`. When you expand the right end of the window, check to see if the number of distinct elements has gone up. If that number exceeds `k`, shrink the window from the left until the distinct elements are at most `k`. Increase the answer by the length of the window because that's how many subarrays have at most `k` distinct values ending at the right endpoint.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      long long n, k;
6      cin >> n >> k;
7
8      vector<long long> v(n);
9      for (int i = 0; i < n; i++) {
10         cin >> v[i];
11     }
12
13     map<long long, int> freq; // frequency of each element in current window
14     long long left = 0, ans = 0, distinct = 0;
15
16     for (int right = 0; right < n; right++) {
17         // add right element to window
18         if (++freq[v[right]] == 1) { // If it's a new element, increase the
19             // number of distinct elements.
20             distinct++;
21         }
22
23         // shrink window until we have at most k distinct elements
24         while (distinct > k) {
25             freq[v[left]]--;
26             if (freq[v[left]] == 0) { // if an element is no longer present in
27                 // the window, decrease the number of distinct elements
28                 distinct--;
29             }
30             left++;
31         }
32         ans += (right - left + 1);
33     }
34     cout << ans << endl;
35     return 0;
36 }
```

²⁰See Section 2.2.16

```
26         distinct--;  
27     }  
28     left++;  
29 }  
30  
31     // all subarrays ending at right with start in [left, right] are  
    valid  
32     ans += (right - left + 1);  
33 }  
34  
35 cout << ans << "\n";  
36 }
```


2.2.33. Array Division

[Question - Array Division](#) [Backup Link](#)

Hint:

If you were given a test sum s , you could find out how many subarrays you would need such that the sum of each subarray would be at most s . Could you use this information to find the largest s such that you only divide the array k times?

Solution:

We can use binary search on answers to solve this question. We assume the answer lies in the range `left` to `right`. We then guess the sum `mid = (l + r)/2`, we then check if we can split the array into k subarrays such that the sum of each subarray is at most `mid`. If it's possible, you can then attempt a lower sum in half the range `left` to `mid-1`. If it's not possible, you can then attempt a higher sum in half the range `mid+1` to `right`. Repeat till you find the smallest sum.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3  using ll = long long;
4
5  int main() {
6      int n, k;
7      cin >> n >> k;
8      vector<int> a(n);
9
10     ll left = 0, right = 0;
11     // 'left' = minimum possible max sum (largest single element)
12     // 'right' = maximum possible max sum (sum of all elements)
13
14     for (int i = 0; i < n; i++) {
15         cin >> a[i];
16         left = max(left, (ll)a[i]);
17         right += a[i];
18     }
19
20     ll ans = right;    // Best answer found via binary search
21
22     while (left <= right) {
23         ll mid = (left + right) / 2;
24
25         // Try to split into <= k subarrays where no subarray sum exceeds
           'mid'
```

C++

```

26     int subarrays = 1;
27     ll current_sum = 0;
28     bool possible = true;
29
30     for (int i = 0; i < n; i++) {
31         // If adding this element exceeds 'mid', start a new subarray
32         if (current_sum + a[i] > mid) {
33             subarrays++;
34             current_sum = a[i];
35
36             // Too many subarrays means that mid is too small
37             if (subarrays > k) {
38                 possible = false;
39                 break;
40             }
41         }
42         else //Add the current element to this subarray's sum
43             current_sum += a[i];
44     }
45     //you could split the array into k subarray with their sum being at
46     //most mid
47     //so try values that are smaller than mid to get a better answer
48     if (possible) {
49         ans = mid;
50         right = mid - 1;
51     }
52     else //you couldn't split the array into k subarray with their sum
53         //being at most mid
54         left = mid + 1; // so try values that are larger than mid.
55 }
56
57 cout << ans << "\n";
58 return 0;
59 }

```

2.2.34. Movie Festival II

[Question - Movie Festival II](#) [Backup Link](#)

Solution:

The movies are sorted by ending time so earlier-finishing movies are considered first. A `multiset` stores when each of the `k` watchers becomes free. For each movie, we find the watcher who become free as close as possible to when the current movie starts (i.e. the time when the watcher becomes free is the largest value less than or equal to the start time of the movie). If such a watcher exists, we assign the movie and update when the time when they become free. This greedy process maximizes the total number of movies watched.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, k;
6      cin >> n >> k;
7
8      // movies[i] = {end_time, start_time}
9      vector<pair<int, int>> movies(n);
10     for (int i = 0; i < n; i++) {
11         cin >> movies[i].second >> movies[i].first;
12     }
13
14     // Sort movies by ending time (classic interval scheduling)
15     sort(movies.begin(), movies.end());
16
17     // Each element represents when a watcher becomes free
18     multiset<int> freeAt;
19     for (int i = 0; i < k; i++) {
20         freeAt.insert(0);
21     }
22
23     int watched = 0;
24
25     for (auto [endTime, startTime] : movies) {
26         // Find the watcher who is free as close to the startTime (i.e
27         // largest values less than or equal to startTime)
28         auto it = freeAt.upper_bound(startTime); //auto is
29         multiset<int>::iterator
```

C++

```

30         // No watcher available
31         continue;
32     }
33
34     // Assign this movie to the latest possible free watcher
35     freeAt.erase(--it); //upperbound - 1 is the same as largest value less
                           than or equal to startTime
36     freeAt.insert(endTime); //updating when the watcher gets free
37     watched++;
38 }
39
40 cout << watched << "\n";
41 return 0;
42 }

```

2.2.35. Maximum Subarray Sum II

[Question - Maximum Subarray Sum II](#) [Backup Link](#)

Solution:

We can first build a prefix sum array so any subarray sum can be computed in $O(1)$. A multiset stores prefix sums such that if you subtract the prefix sums in the multiset from the prefix sum at the current index `right`, you get the sums of subarrays with lengths `minLen` to `maxLen` ending at `right`. As `right` moves forward, new valid prefixes are added to the set. Prefixes that would make the subarray longer than `b` ending at `right` are removed. The smallest prefix in the `multiset(valid.begin())` gives the maximum possible subarray ending at `right`. The answer is updated by comparing all such valid subarrays.

Code:

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int n, minLen, maxLen;
6      cin >> n >> minLen >> maxLen;
7
8      vector<long long> values(n);
9      for (int i = 0; i < n; i++) {
10         cin >> values[i];
11     }
12
13     // Prefix sums: prefix[i] = sum of first i elements
14     vector<long long> prefix(n + 1, 0);
15     for (int i = 0; i < n; i++) {
16         prefix[i + 1] = prefix[i] + values[i];
17     }
18
19     multiset<long long> valid;
20     long long bestSum = LLONG_MIN;
21
22     for (int right = minLen; right <= n; right++) {
23         // Add the prefix sum corresponding to a subarray of length >= minLen
24         valid.insert(prefix[right - minLen]);
25
26         // Remove the prefix sum that would make subarray length > maxLen
27         if (right > maxLen) {
28             valid.erase(valid.find(prefix[right - maxLen - 1]));
29         }
30     }
```

```
31         // Best subarray ending at 'right'
32         bestSum = max(bestSum, prefix[right] - *valid.begin());
33     }
34
35     cout << bestSum << "\n";
36     return 0;
37 }
```

3. Dynamic Programming (WIP)

4. Graph Algorithms (WIP)

5. Range Queries (WIP)