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

This is a book meant for competitive programming. We wrote this book because we felt that other resources while good, lacked the ability to explain more complex topics well. Editorial written to questions that we used to practice were also not written well for the most complex problem. Sometimes even if an editorial is written well, we'd first spend hours trying to solve the question before looking up the solution and then realise we needed some well known concept. To solve this frustration and give you, the reader, the ability to solve as many questions on your own. We first go through all the concepts required to solve a bunch of questions and then provide hints and solutions to the questions.

We're using the CSES Problem Set as our question bank and you can go and create an account there and start solving. Depending on how much programming and `c++` you know, you can first skim through the concepts required for the section of the CSES Problem Set that you're working on and make sure you know them well enough. If you do get stuck despite knowing the concepts, there are hints to give you a little help and the full solution, well written and easy to understand there for you.

This book does expect some basic knowledge about programming in at least 1 programming language even if that language isn't `c++`. If you are completely new to programmer, we have linked a resource in the first section where you can learn the basics.

We hope this will help you become a better competitive programmer.

Taksh Kothari and Apurva Bhat.

1. Introductory Problems

1.1. Concepts

1.1.1. Basic `c++` syntax

1.1.1.1. Question

Accept the number of students from user. Accept their names and marks. Print the Name(s) of students who scored the highest percentage. We deliberately use `c++` features useful for programming contests.

Solution:

```cpp

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

double calcPercent(int numerator, int denominator){
  return numerator * 100.0 / denominator;
  // An example of a single line comment
  /* An example of a multiline comment
     numerator / denominator * 100.0 will first do integer division.
     That's why we multiply by 100.0 first and then divide by the denominator. */
}

struct Student{
  string name;
  pair<int, int> marks;
  double percent;
  Student(); // this is a default constructor

  // this is a parameterized constructor
  Student(string name, pair<int, int> marks) {
    this->name = name;
    this->marks = marks;
    percent = calcPercent(marks.first, marks.second);
  }
};

// Program execution begins from here
int main() {
  int n;
  cin >> n;

  // to create an array of n Students the default constructor was neccesary
  Student arr[n];
```

```
  double maxPercentage = 0.0;

  for(int i = 0; i < n; i++){
    string name;
    pair<int, int> marks;
    cin >> name >> marks.first >> marks.second;

    // calling parameterized constructor
    arr[i] = Student(name, marks);
    maxPercentage = max(arr[i].percent, maxPercentage);
  }

  // a vector is a resizeable array with some useful functions
  // memory is automatically allocated in a vector
  vector<Student> best;

  for(int i = 0; i < n; i++)
    if(arr[i].percent == maxPercentage) {
      // push_back() adds the student to the end of the vector
      best.push_back(arr[i]);
    }

  cout << "Names, Marks and Percentages of top scorers!" << endl;
  for(int i = 0; i < best.size(); i++) {
    cout << "Name: " << best[i].name;
    cout << ", Marks: " << best[i].marks.first << "/" << best[i].marks.second;
    cout << ", Percentage: " << best[i].percent << endl;
  }

  return 0; // end code
}
```

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

1.1.1.2. Data Types

This code contained the data types `int`(Integer which is a non decimal number) , `double` (Decimal Number), `string` (Text) and `pair<int,int>`. A `pair` is a datatype 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 `int`'s but it could be a pair of `int` and `string` and much more.

1.1.1.3. Variables

Variables are strongly typed in `c++` which means you must specify their datatype and then their name.

1.1.1.4. Input/Output

Input and output is does with `cin` and `cout` and angle brackets `>>` for input and `<<` for output.

1.1.1.5. Conditional Statements

Conditions 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`.

1.1.1.6. 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 3 part is what happens at the end of the loop block which is usually to update the variable initialized in the first part.

1.1.1.7. Classes/Structs

In this program we made a `struct` because their easier to use than a `class`. They work in nearly the same way though and the only difference really is that members in a `struct` are `public` by default but members in a `class` are `private` by default.

1.1.1.8. Arrays/Vectors

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

1.1.1.9. 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`.

More about `c++` syntax can be learned here.

1.1.2. Time Complexity

Time Complexity is simply a measure of how much longer it takes a program to run as the input size grows larger. We represent by using something called Big-O Notation. For instance, say we have a program that is $O(n)$, 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.

Whenever you are solving a question, always calculate the time complexity of your algorithm. When you plug in the maximum input sizes into your time complexity, the amount of time it should take should be less than 10^8 because that's usually how many operations occur in one second.

1.1.3. Pointers

Unlike in other higher level programs 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 a code which uses pointers and we'll explain what it does:-

```cpp

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

int main(){
  int a = 5; //made an int variable with value 5.
  int *b = new int(7);//made an integer pointer with value 7 at that memory location.
  int &c = a;//made an int variable which refers to the same memory location as a.
  int *d = &a;//made an int pointer which points to the same memory location as a.

  cout << a << " " << *b << " " << c << " " <<  *d << endl;
  c = 9;//also changes a and d
  cout << a << endl;//9
  *d = 15;// also changes a and c
  cout << a << endl;//15

  delete b;//every time you use "new" you must always "delete" the pointer to prevent memory leaks
  return 0;
}
```

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 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 my 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 allowed 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`.

1.1.4. 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 the 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 wastes 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 reason being that:-

1. It's easier to initialize all values in a vector
   ```
   vector<int> v(5,-1)//Initializes vector of size 5 filled with -1
   ```
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.

1.1.5. 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:

```cpp

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

int fact(int n){
  if(n == 1) //base case
    return 1;
  return n * fact(n - 1);//recursion
}

int main(){
  int n;
  cin >> n;
  cout << fact(n) << endl;

  return 0;
}
```

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 recursion 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.

1.1.6. Sorting

To sort a data structure like an array of vector, `c++` has it's own sort function for this:

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

int main(){

  int arr[] = {3,4,6,2,5,1};
  vector<int> v = {6,2,4,5,1,3};

  sort(arr, arr+6);//Sorts the array {1,2,3,4,5,6}
  sort(v.begin(), v.end());//Sorts the vector {1,2,3,4,5,6}

  return 0;
}
```

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. 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! When ever 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 find 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 half way. 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 by half way forward 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. Starting in the middle
2. If you are equal to the target you have found the value! Otherwise, go to 3.
3. If you are less than the target, eliminate the left and then jump to the middle of the right half then go back up to 2. If not, go to 4
4. If you are more than the target, eliminate the right half and jump to the middle of the left, then go to 2.

Let's see the algorithm in action:-

Let's say we have the following sorted array:

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

And let that the target number we are looking for be 18. Let the be the variables $\text{left} = 0$, $\text{right} = 13$, and $\text{middle} = \frac{\text{left} + \text{right}}{2} = \frac{0+13}{2} = 6$ 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, $\text{middle} < 18$. This tells us that our target value lies to the right of middle. We can now update middle by first making $\text{left} = \text{middle} + 1 = 6 + 1 = 7$, then make $\text{middle} = \frac{\text{left} + \text{right}}{2} = \frac{7+13}{2} = 10$.

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

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

```C++

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

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

  int n, t;
  cin >> n;
  vector<int> v(n);
  for(int i = 0; i < n; i++)
    cin >> v[i];

  int l = 0, r = n - 1, m ;
  while(l <= r){
    m = (l + r)/2;
    if(v[m] == t){//if the number at m meets the target
      cout << "YES" << endl;
      cout << m << endl;
      return 0;
    }
    else if(v[m] < t)
      l = m + 1;//eliminate the left(lesser) half
    else if(v[m] > t)
      r = m - 1;//eliminate the right(greater) half
  }
  cout << "NO" << endl;
  return 0;
}
```

1.1.8. 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 2 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:

```cpp
int l = 0, r = n - 1, m, lb = -1;
while(l <= r){
  m = (l + r)/2;
  if(v[m] < t)
    l = m + 1;
  else if(v[m] >= t){// >= instead of > because it lower bound.
    lb = m;//we set the lower bound to the middle
    r = m - 1;// and then eliminate the right half.
  }
}

cout << lb << endl;
```

Upper Bound:

```cpp
int l = 0, r = n - 1, m, ub = -1;
while(l <= r){
  m = (l + r)/2;
  if(v[m] <= t)//< to <= the equal condition isn't missed.
    l = m + 1;
  else if(v[m] > t){
    ub = m;//we set the upper bound to the middle
    r = m - 1;//and then eliminate the right half.
  }
}

cout << ub << endl;
```

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

Now lucky for you, `c++` comes with it's own upper bound and lower bound functions! Here's their use cases:

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

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

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

  int n, t;
  cin >> n;
  vector<int> v(n);
  for(int i = 0; i < n; i++)
    cin >> v[i];

  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.
  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.

  cout << lb - v.begin() << endl;//difference between lb and v.begin() tell you the index of the lower bound.
  cout << ub - v.begin() << endl;//difference between ub and v.begin() tell you the index of the upper bound.
  return 0;
}
```

As you can see from the code above:-

- `lower(v.begin(),v.end(),t)` returns an `iterator` to the lower bound of `t`.
- `upper(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 do you get the largest element lesser than or the largest element less than or equal to. This can be achieved by subtracting 1 to the lower bound and upper bound respectively.

1.1.8.1. `lower_bound()` & `upper_bound()` with custom sorting.

Sometimes your `vector` may not be sorted in ascending order. Sometimes it might be descending, sometimes it could be 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 more 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 it on a descending list. Apply a correction factor as needed.

1.1.9. 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 do to make a computer do it? Try listing the permutations yourself and see if you come up with sometime before reading onwards.

Alright, here's the method:

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

1. `"abc"`
2. `"acb"`
3. `"bac"`
4. `"bca"`
5. `"cab"`
6. `"cba"`

As you can see, we went through all permutations starting from the string sorted in ascending order and then in 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]`). This position is the pivot.
2. Swap the element at the pivot with its upper bound to the left of it (See Section 1.1.8)
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:

C++

```cpp

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


int main(){

  string str = "abcd";

  do {
    cout << str << endl;
  } while(next_permutation(str.begin(),str.end());

  return 0;
}
```

1.1.9.1. `next_permutation()`

Fortunately for you, `c++` already has a function for you that generates the next permutation! Here's the same code we wrote above but using `next_permutation()`:

C++

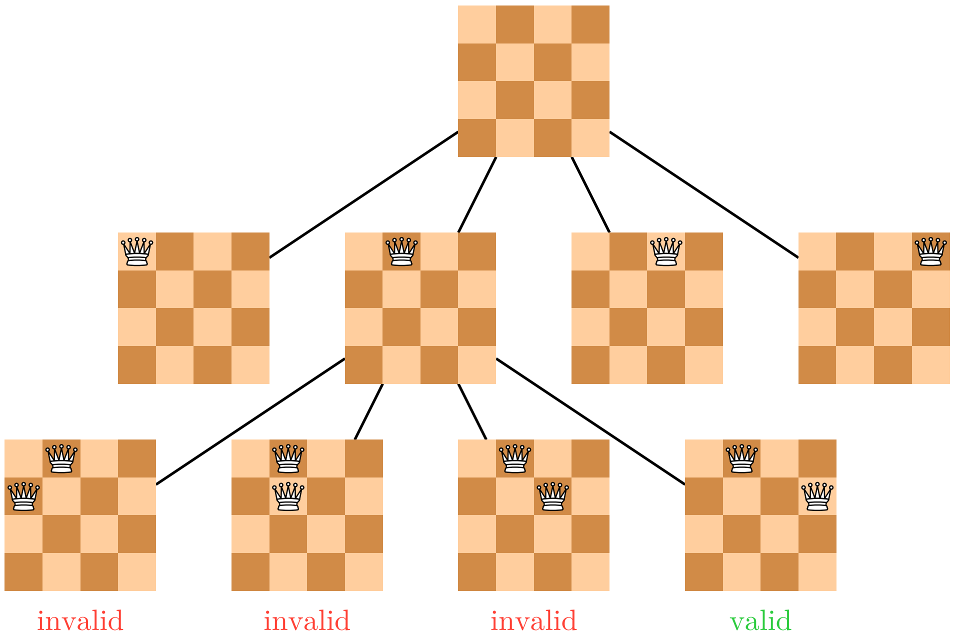
```cpp
#include <bits/stdc++.h>
using namespace std;

bool permute(string &str){
  for(int i = str.length() - 2; i >= 0; i--){
    if(str[i] < str[i + 1]){//pivot finding

      int ub_idx = lower_bound(str.begin() + (i + 1), str.end(), str[i], greater<int>()) - 1 - str.begin();//finds the upper bound of element at pivot.
      swap(str[i], str[ub_idx]);//swaps the ub and the element at the pivot
      reverse(str.begin() + (i + 1), str.end());//reverses the elements after the pivot

      return true;//successfully produced the next permutation

    }
  }

  return false;//failed to produce the next permutation. This happens when the string is in descending order because that the last permutation.
}

int main(){

  string str = "abcd";

  do {
    cout << str << endl;
  } while(permute(str));

  return 0;
}
```

1.1.10. Backtracking

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

Say we want to know for a $n \times n$ chess board, how may ways are there to place n queens, such that two queens never attack each other.[1]

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 position, see which positions you can place a queen in the second row and so on. Let's looks 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 then go from there.

To write the code for this. We need 4 arrays, one for every row, columns, and both diagonals. If `row[i]` is true, that means there's a queen in that row and we can't place a queen there. The indexes of the two diagonals will be a 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 on `diag[(n-1) - j + i]`. Then for the next row, a queen can't be placed on this row, columns, and diagonal.

Here's the code of the implementation for this algorithm:

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

C++

```cpp

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

int n, ans = 0;
vector<bool> col, diag1, diag2;

void findPositions(int i = 0){
  if(i == n){//If true, we successfully placed all the queens in an arrangement.
    ans++;
    return;
  }

  for(int j = 0; j < n; j++){
    if(col[j] || diag1[i+j] || diag2[(n-1)-j+i]) //The new queen would be attacked
      continue;
    col[j] = diag1[i+j] = diag2[(n-1)-j+i] = true;//Placing the queen on the current spot
    findPositions(i+1);
    col[j] = diag1[i+j] = diag2[(n-1)-j+i] = false;//Removing queen for the current spot
  }
}

int main(){
  //n was defined globally

  cin >> n;
  col.resize(n);
  diag1.resize(2*n-1);
  diag2.resize(2*n-1);
  findPositions();

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

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 observer that we didn't use a `row vector`, because the backtracking algorithm ensures that we are placing the new queen on a new row each time.

The complexity of this code is $O(n!)$ which grows very quickly. Solving the question 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).

1.1.11. 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 and then you can buy your own ticket and then 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` is:

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:

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

struct Person{
  string name;
  int tickets;

  Person();// default constructor

  Person(string name, int tickets){
    this->name = name;
    this->tickets = tickets;
  }
};

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

  queue<Person> q;

  // Adding people to the queue
  for(int i = 0; i < n; i++){
    string name;
    int tickets;
```

```cpp
    cin >> name >> tickets;

    q.push(Person(name, tickets));// Add person to the back of the queue
  }

  cout << "Serving customers:" << endl;

  // Process the queue
  while(!q.empty()){// While the queue is not empty
    Person cur = q.front();// Get the person at the front
    q.pop();// Remove them from the queue

    cout << "Serving " << cur.name << " " << cur.tickets;
    if(cur.tickets = 1)
      cout << " ticket." << endl;
    else
      cout << " tickets." << endl;
  }

  return 0;
}
```

Sample input:

```
5
Alice 2
Bob 1
Charlie 3
Diana 2
Eve 1
```

Output:

```
Serving customers:
Serving Alice 2 tickets.
Serving Bob 1 ticket.
Serving Charlie 3 tickets.
Serving Diana 2 tickets.
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've been achieved with a `vector`, you'll find that there are better uses for queue in the graph algorithm section.

For the `std::queue` documentation, click here.

1.2. CSES Practice 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 != 1`

Code :

C++

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

int main() {
    long long n; // Use long long to handle large values of n
    cin >> n;
    cout << n << " "; // Print the starting number

    // Continue until n becomes 1
    while (n != 1) {

      if (n % 2 == 0) // Case 1: n is even → halve it
        n /= 2;
      else // Case 2: n is odd → apply 3n + 1
        n = 3 * n + 1;

      // Print current value after operation
      cout << n << " ";
    }

    return 0;
}
```

1.2.2. Missing Number

Question - Missing Number   Backup Link

Explanation :

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 :

C++

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

int main() {
    long long n, x, total = 0;
    cin >> n;

    for (int i = 0; i < n - 1; i++) {
        cin >> x;
        total += x;
    } // Read n-1 numbers and sum them

    long long sum = n * (n + 1) / 2;  // Expected sum of 1 to n

    // The missing number is the difference
    cout << sum - total << "\n";
    return 0;
}
```

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:

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

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... :

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

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:

$$2S = n(n + 1)$$

$$\therefore S = \frac{n(n + 1)}{2}$$

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 :

C++

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

int main() {
    string s;
    cin >> s;
    int maxLen = 1, current = 1;

```

```
    for (int i = 1; i < s.length(); i++) {
        // Check if current character matches the previous one

        // Increment current length of consecutive characters
        if (s[i] == s[i - 1])
          current++;
        // Reset current to 1 if characters differ
        else
          current = 1;

        // Update maxLen if current is larger
        maxLen = max(maxLen, current);
    }

    cout << maxLen << "\n";
    return 0;
}
```

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 current $\geq$ prev, move on because the order is fine.
4. If current $<$ prev, we need to increase it by (prev $-$ 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 :

C++

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

int main() {
    int n, prev;
    long long operations = 0; // total number of increments needed
    cin >> n >> prev;

    // process the rest of the array
    for (int i = 1; i < n; ++i) {
        int current;
        cin >> current;  // read the current element

        // if the current element is smaller than the previous,
        // we need to increment it to match 'prev' (to keep array non-decreasing)
        if (current < prev) {
            // count how many increments are required
            operations += prev - current;
            // simulate the increment (virtually update current)
            current = prev;
        }
```

```

        prev = current; // update 'prev' for the next iteration
    }

    cout << operations << "\n";
    return 0;
}
```

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 :

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

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

    // Special case: if n = 2 or 3, it is impossible to arrange
    // numbers from 1...n so that no two consecutive numbers
    // differ by 1. Hence, print "NO SOLUTION".
    if (n = 2 && n = 3)
        cout << "NO SOLUTION";

    // Base case: if n = 1, the only permutation is "1".
    else if (n == 1)
        cout << "1";

    // General case: n >= 4
    else {
        // First print every other number from n-1 in descending order.
        // This ensures that the gap between every number is more than 1.
        for (int i = n - 1; i >= 1; i -= 2)
            cout << i << " ";

        // Then print every other number from n in descending order.
        for (int i = n; i >= 1; i -= 2)
            cout << i << " ";
    }
}
```

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 $21 - 2 = 21$.

Code:

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

int main() {
```

```
    int t; // Number of test cases
    cin >> t;
    while (t--) {
        // Cell coordinates
        long long y, x;
        cin >> y >> x;
        long long layer = max(x, y); // Layer is max(x, y)
        long long val = layer * layer - layer + 1; // Base value for layer's diagonal cell

        if (layer % 2 == 0) // Even layer
            if (x == layer) // Adjust for y
                cout << val - (layer - y) << "\n";
            else // Adjust for x
                cout << val + (layer - x) << "\n";

        else // Odd layer
            if (y == layer) // Adjust for x
                cout << val - (layer - x) << "\n";
            else // Adjust for y
                cout << val + (layer - y) << "\n";
    }
    return 0;
}
```

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").

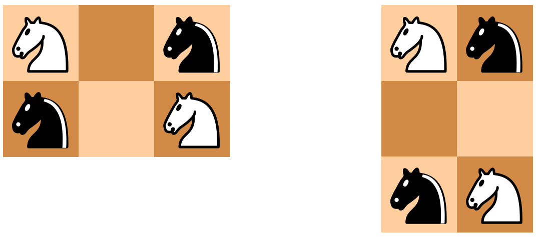


Figure 5: The 2 pairs of same coloured knights attack each other from the corners.

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:

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

Code:

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

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

  for (long long k = 1; k <= n; ++k) {
    long long total = k * k;

    // Total ways to place 2 knights on k×k board
    long long ways = total * (total - 1) / 2;

    // Subtract attacking pairs (only exist when k > 2)
    if (k > 2)
      ways -= 4 * (k - 1) * (k - 2);

    cout << ways << "\n";
  }

  return 0;
}
```

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 1^{st} with the 4^{th} number and the 2^{nd} and 3^{rd} 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 1^{st} and 4^{th} numbers will add up to $(x) + (x + 3) = 2x + 3$ which is the same as the sum of the 2^{nd} and 3^{rd} 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 :

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

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

    //if n is not a multiple of 4 or 3 more than a multiple of 4, output "NO".
    if (n % 4 == 1 || n % 4 == 2)
      cout<<"NO";

    else {

        cout<<"YES"<<endl;

        // Vectors to store the two sets.
        vector<int> a, b;
```

```
        // Process numbers in groups of 4 from largest to smallest,
        // assigning to sets such that each group adds equal sum to both.
        while (n > 3 && n > 0) {
                a.push_back(n);
                a.push_back(n-3);

                b.push_back(n-1);
                b.push_back(n-2);

                n = n - 4;
        }

        // Handle the remaining 3 numbers if n % 4 == 3 (balanced assignment).
        if (n == 3) {
            a.push_back(3);

            b.push_back(2);
            b.push_back(1);
        }

        // Output size and elements
        cout << a.size() << "\n";
        for (int num : a)
            cout << num << " ";

        cout << b.size() << "\n";
        for (int num : b)
            cout << num << " ";

        return 0;
    }
}
```

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 :

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

int main() {
    const long long MOD = 1e9 + 7;
    long long n;
    cin >> n;

    long long answer = 1;
    for (long long i = 0; i < n; ++i) {
        answer = (answer * 2) % MOD;
    }

    cout << answer << "\n";
    return 0;
}
```

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).[2]
- $\lfloor \frac{27}{25} \rfloor = 1$ (extra 5 from 25).
- $\lfloor \frac{27}{125} \rfloor = 0$ (stop).
- Total: $5 + 1 + 0 = 6$ zeros.

Code:

C++

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

int main() {
    int n, count = 0;
    cin >> n; // Read input number n

    // Count factors of 5 in n! by summing floor(n/5) + floor(n/25) + floor(n/125) + ...

    for (int i = 5; n / i >= 1; i *= 5) {
        count += n / i; // Add number of multiples of i (powers of 5)
    }

    cout << count << endl; // Output the number of trailing zeros
    return 0;
}
```

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

1.2.11. Coin Piles

Question - Coin Piles   Backup Link Is Broken will fix later.

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 :

C++

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

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

    while (t--) {
        int a, b;
        cin >> a >> b;

        bool condition1 = (a + b) % 3 == 0;//sum of coins in pile is a multiple of 3.
        bool condition2 = max(a, b) <= 2 * min(a, b);//number of coins in bigger mile must be less than or equal to twice the number of coins in the smaller pile.

        if(condition1 && condition2)
          cout<< "YES" << "\n";
        else
          cout<< "NO" << "\n";
    }

    return 0;
}
```

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:

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

int main() {
    string s;
    cin >> s; // Read input string

    vector<int> arr(26, 0); // frequency array for letters A–Z (all initialized to 0)

    // Count how many times each character appears
    for (char c : s)
        arr[c - 'A']++;

    // Count how many characters have odd frequencies
    int oddCount = 0;
    for (int i = 0; i < 26; i++)
        if (arr[i] % 2 != 0)
            oddCount++;

    // Palindrome rule:
    // - If string length is even → no odd frequencies allowed
    // - If string length is odd  → exactly one odd frequency allowed
    if (oddCount > 1 && s.size() % 2 == 1) {
        cout << "NO SOLUTION"; // too many odd-count letters
    }
    else if (oddCount > 0 && s.size() % 2 == 0) {
        cout << "NO SOLUTION"; // even-length string with odd-count letters
    }
    else {
        // Container to build the palindrome result
```

```cpp
        vector<char> str(s.length());
        int left = 0, right = s.length() - 1;

        // Fill symmetric pairs from both sides
        for (int i = 0; i < 26; i++) {
            while (arr[i] >= 2) {
                str[left++] = (char)('A' + i);   // place letter on left side
                str[right--] = (char)('A' + i);  // place same letter on
                right
                arr[i] -= 2; // remove two occurrences
            }
        }

        // If one odd-count character remains, put it in the middle
        for (int i = 0; i < 26; i++)
            if (arr[i] == 1)
                str[left] = (char)('A' + i);

        // Convert vector<char> back to a string
        s = string(str.begin(), str.end());

        // Print the final palindrome
        cout << s << "\n";
    }
    return 0;
}
```

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:

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

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

    int n;
    cin >> n;

    vector<string> gray;
```

```
    gray.push_back("");

    for (int i = 0; i < n; i++) {
        vector<string> next;

        // Prefix "0" to all existing strings
        for (int j = 0; j < gray.size(); j++)
            next.push_back("0" + gray[j]);

        // Prefix "1" to all existing strings in reverse order
        for (int j = gray.size() - 1; j >= 0; j--)
            next.push_back("1" + gray[j]);

        gray = next;
    }

    for(int i = 0; i < gray.size(); i++)
        cout << gray[i] << "\n";
}
```

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 sore that in `next`. Finally we update `gray` to next.

This process is repeated until you get the nth 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 on string to the other, we get: $\{0, 1, 0, 2, 0, 1, 0\}$. In this list, you can see that we flipped the 0^{th} bit every 2^{nd} bit flip, the 1^{st} bit every 4^{th} bit flip, and if we were to look at Gray code of $n = 4$, we would also notice that the 2^{th} bit was flipped every 8^{th} time.

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

$$t_n = \log_2(\text{lsb}(n)), \text{Where } t_n \text{ is the } n^{\text{th}} \text{ term.}$$[3]

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:

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

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

    int n;
    cin >> n;
    string s(n, '0');// Start with all zeros

    cout << s << endl;
    for (int i = 1; i < (1 << n); i++) {
        // Find position of lowest set bit and flip the corresponding bit
        int pos = n - 1 - __builtin_ctz(i & -i);
        s[pos] = (s[pos] == '0' ? '1' : '0');// Flip the bit.
        cout << s << "\n";
    }
    return 0;
}
```

`i & -i` computes $\text{LSB}(n)$ and `__builtin_ctz()` computes the number of trailing zeros which is the same as `log2()` 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.

[3] See Section 2.1.1.7 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 recursion[4] code which solves the question.

C++

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

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

  int n;
  cin >> n;

  vector<pair<int, int>> moves;

  // This can also be a normal function instead of a lambda expression.
  // Make sure that the moves vector is a global for this to work.
  function<void (int, int, int, int)> solve = [&] (int n, int start, int end, int middle){
    if (n == 0) return;
    solve(n - 1, start, middle, end);
    moves.push_back({start, end});
    solve(n - 1, middle, end, start);
  };

  //output:
  solve(n, 1, 3, 2);
  cout << moves.size() << endl;
  for(int i = 0; i < moves.size(); i++)
    cout << moves[i].first << " " << moves[i].second << endl;
```

[4] See Section 1.1.5

```
    return 0;
}
```

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()`[5] which helps us tackle this exact question. This function can be used to generate the next lexicographical[6] 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 :

C++

```cpp

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

int main() {
    string s;
    cin >> s;

    //sort the string to get the lowest possible lexiographical sequence
    sort(s.begin(), s.end());

    vector<string> v;

    do {
        v.push_back(s);
    } while (next_permutation(s.begin(), s.end()));
    // returns false if no other permutation exists
    // otherwise it rearranges the string

    cout << v.size() << "\n";
    for (int i = 0; i < v.size(); i++) {
        cout << v[i] << "\n";
    }
}
```

[5]See Section 1.1.9

[6]Meaning 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 bitmasks[7], 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:

C++

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

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

    vector<int> v(n);
    long long total = 0;

    // Read weights and compute total sum.
    for (int i = 0; i < n; i++) {
        cin >> v[i];
        total += v[i];
    }

    // ans = minimum possible difference between the two groups.
    long long ans = total;

    // Enumerate all subsets using bitmasks from 0 to (2^n - 1).
    // Each mask chooses some apples for the first group;
    // the rest naturally fall into the second group.
    for (int mask = 0; mask < (1 << n); mask++) {

```

[7] See Section 2.1.2

```
        long long sum = 0;

        // Compute sum of elements included in this subset.
        for (int i = 0; i < n; i++) {
            if (mask & (1 << i)) {
                sum += v[i];
            }
        }

        // If subset sum is S, the other group's sum is total - S.
        // Difference is |(total - S) - S| = |total - 2S|.
        ans = min(ans, llabs(total - 2 * sum));
    }

    cout << ans;
}
```

1.2.17. Chessboard and Queens

Question - Chessboard and Queens   Backup Link

Hint:

Please see Section 1.1.10 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 1.1.10 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 :

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

int n = 8, ans = 0;
vector<bool> col, diag1, diag2;
vector<vector<bool>> blocked;

void findPositions(int i = 0){
  if(i == n){//If true, we successfully placed all the queens in an arrangement.
    ans++;
    return;
  }

  for(int j = 0; j < n; j++){
    if(blocked[i][j] || col[j] || diag1[i+j] || diag2[(n-1)-j+i]) //The new queen would be blocked or attacked
      continue;
    col[j] = diag1[i+j] = diag2[(n-1)-j+i] = true;//Placing the queen on the current spot
    findPositions(i+1);
    col[j] = diag1[i+j] = diag2[(n-1)-j+i] = false;//Removing queen for the current spot
  }
}

int main(){
```

```
  //n was defined globally as 8

  for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
      char ch;
      cin >> ch;
      blocked[i][j] = (ch == '*');
    }
  }

  col.resize(n);
  diag1.resize(2*n-1);
  diag2.resize(2*n-1);
  findPositions();

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

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 :

C++

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

void solve(){

    int n, a, b;
    cin >> n >> a >> b;

    // Invalid if required elements exceed n
    if (a + b > n) {
        cout << "NO\n";
        continue;
    }

    // Invalid if exactly one of a or b is zero
    if ((a == 0 || b == 0) && a + b != 0) {
        cout << "NO\n";
```

```
        continue;
    }

    cout << "YES\n";

    // 1 to n
    for (int i = 1; i <= n; i++)
        cout << i << " ";
    cout << "\n";

    // First b elements after a
    for (int i = 1; i <= b; i++)
        cout << a + i << " ";

    // Next a smallest elements
    for (int i = 1; i <= a; i++)
        cout << i << " ";

    // Remaining elements
    for (int i = a + b + 1; i <= n; i++)
        cout << i << " ";
    cout << "\n";
}
int main() {
    int t;
    cin >> t;
  //because ints and bools are loosely typed in c++, when t = 0 the while loops ends.
  //The loop runs t cycles by running from t to 1.
    while (t--)
      solve();

    return 0;
}
```

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:

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

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

    vector<vector<int>> grid(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // Track used numbers
            vector<bool> used(2 * n, false);

            // Left in the same row
            for (int k = 0; k < j; k++) {
                used[grid[i][k]] = true;
            }

            // Above in the same column
            for (int k = 0; k < i; k++) {
                used[grid[k][j]] = true;
            }

            // Find mex
            int mex = 0;
            while (used[mex]) mex++;

            grid[i][j] = mex;
        }
    }

    // Output the grid
    for (int i = 0; i < n; i++) {
```

```
        for (int j = 0; j < n; j++) {
            cout << grid[i][j] << (j + 1 < n ? ' ' : "\n");
        }
    }

    return 0;
}
```

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

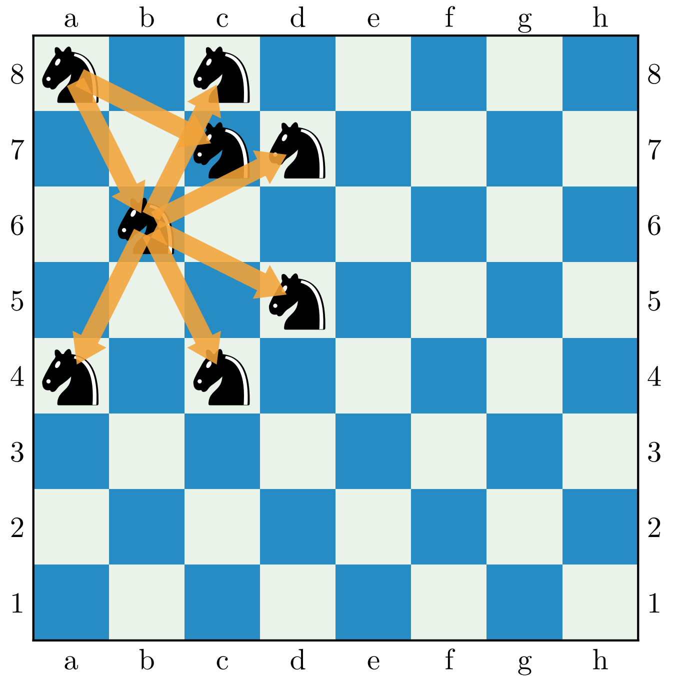
Solution:

The program calculates the minimum number of knight moves needed to reach every square on an `n x 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 1.1.11 for what a queue is.

Code:

C++

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

using namespace std;

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

    // dist[x][y] will store the minimum number of moves needed to reach cell (x, y)
    // a value of -1 means that cell has not been reached yet
    vector<vector<int>> dist(n, vector<int>(n, -1));

    // This queue stores board positions that still need to be explored
    queue<pair<int, int>> q;

    // These arrays describe how a knight moves on a chessboard
    // Each (dx[k], dy[k]) pair represents one possible knight move
    vector<int> dx = {-2, -1, 2, 1, 2, 1, -1, -2};
    vector<int> dy = {-1, -2, 1, 2, -1, -2, 2, 1};

    // This function checks whether a position is inside the board
    // and whether it has not been visited before
    auto isValid = [&](int x, int y) {
        return x >= 0 && y >= 0 && x < n && y < n && dist[x][y] == -1;
    };

    // We begin from the top-left cell (0, 0)
    // Reaching the starting cell takes 0 moves
    dist[0][0] = 0;
    q.push({0, 0});

    // As long as there are positions left to explore, keep processing them
    while (!q.empty()) {
        // Take the oldest unexplored position from the queue
        int x = q.front().first, y = q.front().second;
        q.pop();

        // Try moving the knight in all 8 possible ways from this position
        for (int k = 0; k < 8; k++) {
            int nx = x + dx[k];
            int ny = y + dy[k];

            // If the new position is valid and not visited yet
            if (isValid(nx, ny)) {
```

```
                // The distance to this cell is one more than the current cell
                dist[nx][ny] = dist[x][y] + 1;

                // Add the new position to the queue to explore later
                q.push({nx, ny});
            }
        }
    }

    // Print the minimum number of moves needed to reach each cell
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << dist[i][j] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

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 :

C++

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

int main() {
    int n, m;
    cin >> n >> m;

    vector<string> v(n);        // Input grid
    char ans[n][m];             // Output grid with adjusted characters

    // Read the input grid
    for (int i = 0; i < n; i++)
        cin >> v[i];

    // Construct the output grid
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {

            // Tracks whether letters A, B, C, D are allowed at this cell
            bool isValid[4] = {true, true, true, true};

            // Block the original character in this position
            isValid[v[i][j] - 'A'] = false;

            // Block the character above (if exists)
            if (i > 0)
                isValid[ans[i - 1][j] - 'A'] = false;

```

```
            // Block the character to the left (if exists)
            if (j > 0)
                isValid[ans[i][j - 1] - 'A'] = false;

            // Choose the first valid character using your four ifs
            if (isValid[0])
                ans[i][j] = 'A';
            else if (isValid[1])
                ans[i][j] = 'B';
            else if (isValid[2])
                ans[i][j] = 'C';
            else if (isValid[3])
                ans[i][j] = 'D';
        }
    }

    // Print the final grid
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << ans[i][j];
        cout << "\n";
    }
}
```

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

```C++
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {

    int q;
    cin >> q;

    while (q--) {
        ll n;
        cin >> n;

        ll count = 9;     // how many numbers exist with current digit-length
        ll len = 1;       // current digit-length
        ll start = 1;     // first number of current digit-length

        // skip full digit-blocks (e.g., 1–9, then 10–99, then 100–999...)
        while (n > count * len) {
            n -= count * len;  // remove whole block worth of digits
            start *= 10;       // move to next block's starting number
            count *= 10;       // next block has 10× more numbers
            len++;             // numbers now have one more digit
        }

```

```cpp
        // find the exact number containing the nth digit
        ll num = start + (n - 1) / len;

        // pick the specific digit inside that number
        string s = to_string(num);
        cout << s[(n - 1) % len] << "\n";
    }
}
```

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

The sequence is 123456789101112131415...

Initial values: `n = 15`, `count = 9`, `len = 1`, `start = 1`

After skipping 1-digit block:
- Block 1-9 contributes $9 \times 1 = 9$ digits
- Update: `n = 15 - 9 = 6`, `start = 10`, `len = 2`

Find the number:
- `num = 10 + (6-1)/2 = 10 + 2 = 12`

Find the digit:
- `s = "12"`, `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 :

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

// Returns the lexicographically smallest valid next character index
// freq[] stores remaining frequencies of letters A–Z
// prev = -1 means no previous character (first position)
// prev >= 0 means index of previous character used
int minLexPossible(int freq[], int prev) {
    int maxLetter = 0, sum = 0;        // maxLetter = highest frequency, sum = total remaining letters
    int minLetter = -1, maxIndex = 0; // minLetter = smallest valid choice, maxIndex = most frequent letter

    // Find the smallest lexicographically valid letter different from prev
    for (int i = 0; i < 26; i++) {
        if (freq[i] > 0 && i != prev) {
            minLetter = i;
            break;
        }
    }

    // Compute total remaining letters and the letter with maximum frequency
    for (int i = 0; i < 26; i++) {
        sum += freq[i];
        if (freq[i] > maxLetter) {
            maxLetter = freq[i];
            maxIndex = i;
```

```
        }
    }

    // If any letter appears too often, rearrangement is impossible
    if (maxLetter * 2 > sum + 1) return -1;

    // If the most frequent letter must be placed now to avoid failure, force it
    if (maxLetter * 2 > sum) return maxIndex;

    // Otherwise, choose the smallest lexicographically valid letter
    return minLetter;
}

int main() {
    string s, ans = "";   // s = input string, ans = constructed result
    cin >> s;

    int freq[26] = {0};  // Frequency array for letters A–Z
    for (char c : s) freq[c - 'A']++;

    // Choose the first character (no previous restriction)
    int idx = minLexPossible(freq, -1);
    if (idx == -1) {
        cout << "-1";    // Impossible to form valid string
        return 0;
    }

    ans += char(idx + 'A'); // Append chosen character
    freq[idx]--;            // Decrease its frequency

    // Build the rest of the string character by character
    for (int i = 1; i < s.size(); i++) {
        // Previous character index is ans[i - 1] - 'A'
        idx = minLexPossible(freq, ans[i - 1] - 'A');
        if (idx == -1) {
            cout << "-1"; // No valid continuation
            return 0;
        }
        ans += char(idx + 'A'); // Append next character
        freq[idx]--;            // Update frequency
    }

    cout << ans; // Output the lexicographically smallest valid arrangement
}
```

1.2.24. Grid Path Description

Question - Grid Path Description   Backup Link

Intuitive Explanation :

The program counts how many valid ways exist to move through a `7×7` grid using exactly 48 moves. 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.

Paths that get trapped or split the grid into unreachable regions are stopped early. Every complete path that reaches the destination at exactly 48 steps is counted.

Code :

```cpp
using namespace std;

string path;
bool visited[7][7];          // Marks cells already used in the current path
int ans = 0;

const int dx[] = {1, -1, 0, 0};   // Change in row for each move
const int dy[] = {0, 0, 1, -1};   // Change in column for each move
const char dir[] = {'D', 'U', 'R', 'L'};  // Corresponding move letters

bool inside(int x, int y) {
    return x >= 0 && x < 7 && y >= 0 && y < 7;   // Checks grid boundaries
}

bool is_blocked(int x, int y) {
    if (!inside(x, y) || visited[x][y]) return true; // Outside grid or already used
    return false;
}

void dfs(int x, int y, int step) {
    // If we reached the target cell
    if (x == 6 && y == 0) {
        if (step == 48) ans++;  // Count only if all moves were used
        return;
    }

    // Stop early if movement is forced into a dead split
    if ((is_blocked(x + 1, y) && is_blocked(x - 1, y) &&
         !is_blocked(x, y + 1) && !is_blocked(x, y - 1)) ||
```

```
        (!is_blocked(x + 1, y) && !is_blocked(x - 1, y) &&
         is_blocked(x, y + 1) && is_blocked(x, y - 1)))
        return;

    visited[x][y] = true;   // Mark this cell as used

    for (int d = 0; d < 4; ++d) {
        int nx = x + dx[d];
        int ny = y + dy[d];

        // Enforce the given path character if it is not '?'
        if (path[step] != '?' && path[step] != dir[d]) continue;

        // Move only to valid and unused cells
        if (inside(nx, ny) && !visited[nx][ny])
            dfs(nx, ny, step + 1);
    }

    visited[x][y] = false;  // Undo the move before trying other possibilities
}

int main() {
    cin >> path;
    dfs(0, 0, 0);           // Start from top-left with zero moves taken
    cout << ans << endl;
    return 0;
}
```

2. Sorting and Searching

2.1. Concepts

2.1.1. Bit Operations

In `c++`, you can perform binary operations on individual bits. This may sound confusing, so let's look at some examples.

2.1.1.1. Negative Numbers

In a computer, all numbers are stored in binary. For an `int`, the computer allocated 32 bits. The number 5 stored in an `int` actually looks like:

$$00000000000000000000000000000101$$

Now for an `unsigned int`, the largest number you can store would be $2^{32} - 1 = 4,294,967,295$, which in binary would be:

$$11111111111111111111111111111111$$

However regular `int` need the capability to store negative numbers. If we start from 0 and then subtract 1, we roll back and reach -1 which would be:

$$11111111111111111111111111111111$$

Going back one more place give's -2:

$$11111111111111111111111111111110$$

And so on. until $-2^{31} = -2147483648$:

$$10000000000000000000000000000000$$

if you subtract one from this you go to $2^{31} - 1 = 2147483647$

$$01111111111111111111111111111111$$

The way to convert from binary to decimal using this signed format is the realised the rightmost bit represents -2^{31} instead of positive 2^{31}. So to write a negative number in binary, you can set the rightmost bit to 1 and then find what number to add to -2^{31} to get $-n$. This would be $2^{31} - n$. Finding this is a pain however, so there's a much better way:

Say we want to find out what is -9 in binary. First we must take the **$1'$s complement** of positive 9. This simply means we flip every bit ($0'$s become $1'$s and $1'$s become $0'$s):

$$\begin{aligned} 9 = {} & 00000000000000000000000000001001 \\ & 11111111111111111111111111110110 = -10 \end{aligned}$$

If you positive number was n, this will give you $-n - 1$. To get $-n$, you must add 1. This is called taking the **$2'$s complement**. This would give you:

$$-9 = 11111111111111111111111111110111$$

2.1.1.2. AND (`&`)

Let's say I have the numbers 5 and 7. The question is what would be the output to `cout << (5 & 7);`?

First we write 5 and 6 in binary, which become 0101 and 0110. Then we perform the `and` operation on each bit to get a new number in binary:

$$\begin{array}{r} 0101 \\ \texttt{\&}\ 0110 \\ \hline 0100 \end{array}$$

Finally convert 0100 back to decimal, which is 4

So the code `cout << (5 & 6);` would output `4`

2.1.1.3. OR (|)

Now we want to find out the output of `cout << (5 | 6) << endl`. We now perform the `or` operation on each bit

$$\begin{array}{r} 0101 \\ \texttt{|}\ 0110 \\ \hline 0111 \end{array}$$

Which is 3 in decimal.

2.1.1.4. XOR (^)

Now we want to find out the output of `cout << (5 ^ 6) << endl`. We now perform the `xor` operation on each bit

$$\begin{array}{r} 0101 \\ \texttt{\^{}}\ 0110 \\ \hline 0011 \end{array}$$

Which is 7 in decimal.

2.1.1.5. NOT(~)

The `not(~)` operator flips all the bits of a number. In the earlier examples, we we're only showing 4 bits because the numbers were small. However, the `int` type has a total of 32 bits. So if you want to find the output of `cout << (~5) << endl` The answer would be:

$$\begin{array}{r} \sim 00000000000000000000000000000101 \\ \hline 11111111111111111111111111111010 \end{array}$$

Which is ~~`4294967290`~~ `-6`. This is because ~ generates the $1'$s complement which for some positive n will give you $-n-1$.

2.1.1.6. 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 $000011 \to 000110 \to 001100 \to 011000 \to 110000$ 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 $111001 \to 011001 \to 001100 \to 000110 = 6$.

2.1.1.7. 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 right most bit is in the second position from the left (0-indexed). The value it represents is $100 = 2^2 = 4$ which means the $\mathrm{LSB}(20) = 4$.

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

Why does this work? For that, you must first break up `-n` into `~(n+1)` because that's taking the $2'$s complement. When you take the $1'$s complement of n (`~n`) the rightmost 1 becomes the rightmost 0. All bit 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 become 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 $\mathrm{LSB}(n)$.

Here's how it looks on 20:

$$
\begin{aligned}
20 &= 00000000000000000000000000010100 \\
\& -20 &= \underline{11111111111111111111111111101100} \\
&\ 00000000000000000000000000000100
\end{aligned}
$$

2.1.2. Bitmask

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

Say your 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 the code:

```cpp

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

int main(){

  int n = 3;
  vector<int> v = {5, 4, 7};

  for(int mask = 0; mask < (1 << n); mask++){//mask takes all values from 0 - 2^n-1.

    cout << "{ ";

    for(int i = 0; i < n; i++){
      if(mask & (1 << i))//checking if the ith of the mask is 1
        cout << v[i] << " ";
    }

    cout << "}" << endl;
  }

  return 0;
}
```

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.

```
{ }
{ 5 }
{ 4 }
{ 5 4 }
{ 7 }
{ 5 7 }
{ 4 7 }
{ 5 4 7 }
```

2.1.3. Prefix sum

Let's say your asked this question. You're given an array of numbers, and then your 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 index 4-7, index 2-5, index 1-3. The answers to that would be:

$$12 + 6 + -7 + -3 = 8$$
$$4 + 3 + 12 + 6 = 25$$
$$-6 + 4 + 3 = 1$$

Note that indices are 0-indexed.

You could solve this questions 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 must faster way would be to compute a **prefix sum** array. This means that every element `pref[i]` sore 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:

$$\text{original array } \{5, -6, 4, 3, 12, 6, -7, -3\}$$
$$\text{prefix sum array } \{5, -1, 3, 6, 18, 24, 17, 14\}$$

Now the answers to the 3 queries are:

$$14 - 6 = 8$$
$$24 - (-1) = 25$$
$$6 - 5 = 1$$

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:

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

int main(){

  int n, q;
  cin >> n >> q;

  vector<int> v(n);
  for(int i = 0; i < n; i++)
    cin >> v[i];

```

```
  vector<int> pref(n);
  pref[0] = v[0];

  for(int i = 1; i < n; i++)
    pref[i] = v[i] + pref[i-1];

  for(int i = 0; i < q; i++){
    int a, b;
    cin >> a >> b;

    if(a != 0)
      cout << (pref[b] - pref[a-1]) << endl;
  }

  return 0;
}
```

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 both update and query operations run in $O(\log n)$ time.

2.1.4. Binary Indexed Tree

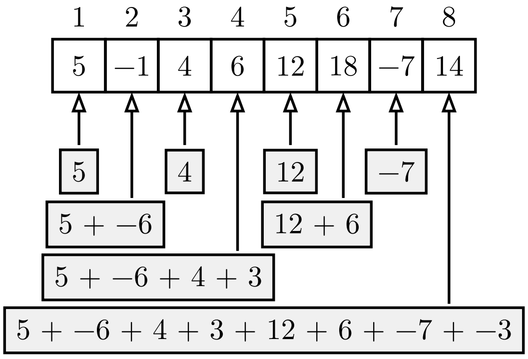
Let's say for the question in the previous section, we not only want the ability to find the sum in a given range, we also want to update an element in the array. This means that we need to be able to both change values in the array and output the sum in any given range quickly.

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 even change a single value, the time it takes to generate the whole array is amortized $O(n)$. For the constraints of $n \leq 2 * 10^5, q \leq 2 * 10^5$, this is too slow.

There is a data structure which can help us do updates and sums in $O(n \log(n))$. This is called a **binary indexed tree(BIT)** or a **Fenwick tree**. In a Fenwick tree, each element is 1-indexed. The ith 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 2.1.1.7 for meaning of LSSB).

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

$$\{5, -6, 4, 3, 12, 6, -7, -3\}$$



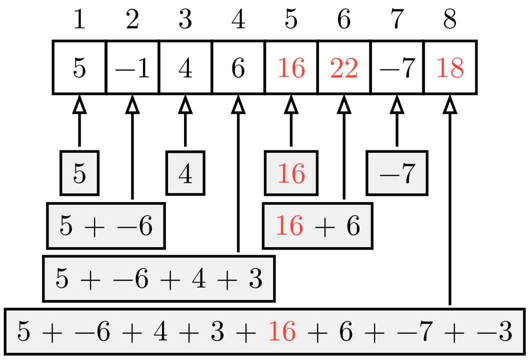
Note that the values in a Fenwick tree are one-indexed, so there we be an empty element at `fenw[0]`.

Hopefully the image made it clearer on how data is stored. The reason for storing data like this is because if you want to add a value to 1 element in the original array, you only need to update $O(\log n)$ values in the Fenwick tree. And after doing this, you can find the sum in $O(\log n)$. Say we wish to add 4 to the 5th index (one-indexed), we only need to update the 5th, 6th, 8th

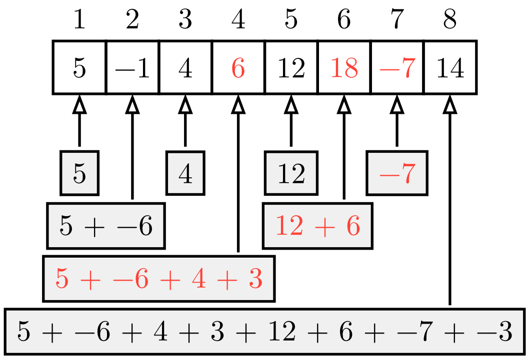
index. If you now want to compute the prefix sum of the array from index 7, you only need to add the values in the 7th, 6th, 4th index.

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 `sum(b) - sum(a-1)` to find the sum of numbers in the subarray `a` to `b`.

Here's the code for the Fenwick tree implementation:

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

int n;
vector<int> fenw;

void add(int x, int k){
  for(; x <= n ; x += x & -x)// x & -x is the LSSB(x)
    fenw[x] += k;
}

int sum(int x){
  int ans = 0;
  for(; x > 0; x -= x & -x)// x & -x is the LSSB(x)
    ans += fenw[x];
  return ans;
}

int sum(int a, int b){
  return sum(b) - sum(a - 1);
}
```

```cpp

int main(){
  int q;
  cin >> n >> q;
  fenw.resize(n + 1, 0);
  for(int i = 1; i <= n; i++){
    int x;
    cin >> x;
    add(i, x);
  }

  for(int i = 0; i < q; i++){
    int t;
    cin >> t;
    if(t == 1){// addition queries
      int x, k;
      cin >> x >> k;
      add(x, k);
    }
    else if(t == 2){// range sum queries
      int a, b;
      cin >> a >> b;
      int ans = sum(a, b);
      cout << sum(a, b) << endl;
    }
  }
  return 0;
}
```

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 Tree's 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 was also sorted and contained unique elements. However, it's not possible to simply access the 2nd, or 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 `fewn[x]`, you store the of times element `x` occurs in your original data. Of course `fewn[x]` will actually store the sum of frequencies from `x - LSSB(x) + 1` to `x` but you get the point. This makes is 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 of the search function:

```cpp
int search(int idx){
  int ans = 0;

  for(int k = floor(log2(n)); k >= 0; k--){//go through the powers of 2.
    if(1 << k <= n && fenw[ans + (1 << k)] < idx){//this element is before the idx.
      ans += 1 << k;//update the answer.
      idx -= fenw[ans];//account for all indices upto fenw[ans].
    }
  }

  return ans + 1;//ans was the value that was before idx, so one value ahead of that is at idx.
}
```

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 it's index (`fenw[ans + 1 << k]`) is less than the `idx`. If it is, you add 2^k to the answer and then subtract `idx` by the indexes covered (`fenw[ans]`).

Since `ans` store the number that is definitely before 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 there at your exact `idx`.

Finally there is one more thing required to make a Fenwick tree useful as an indexed set. A normal sets size is based on the amount of input n which makes its space complexity $O(n)$.

However, a Fenwick tree is build 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 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 how that are are only 5 unique numbers in this entire vector $(2, 3, 4, 5, 10)$. If we were to resign 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:

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

int main(){
  int n;
  cin >> n;
  vector<int> v;
  for(int i = 0; i < n; i++)
    cin >> v[i];

  vector<int> comp = v;
  sort(comp.begin(), comp.end());
  comp.erase(unique(comp.begin(),comp.end()), comp.end());
  for(int i = 0; i < n; i++)
    v[i] = lower_bound(comp.begin(), comp.end(), v[i]) - comp.begin() + 1;// lower_bound - comp.begin() gives you the index and +1 makes sure it's one indexed.
  return 0;
}
```

Sample Input:

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

Output:

```
2 5 2 3 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. When 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 it's position 0-indexed. We then add 1 to get the compressed value such that it is 1-indexed.

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

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

int n;
vector<int> fenw;

void add(int x, int k){
  for(; x <= n ; x += x & -x)// x & -x is the LSSB(x)
    fenw[x] += k;
}

int sum(int x){
  int ans = 0;
  for(; x > 0; x -= x & -x)// x & -x is the LSSB(x)
    ans += fenw[x];
  return ans;
}

int sum(int a, int b){
  return sum(b) - sum(a - 1);
}

int search(int idx){
  int ans = 0;

  for(int k = floor(log2(n)); k >= 0; k--){//go through the powers of 2.
    if(1 << k <= n && fenw[ans + (1 << k)] < idx){//this element is before the idx.
      ans += 1 << k;//update the answer.
      idx -= fenw[ans];//account for all indices upto fenw[ans].
    }
  }

  return ans + 1;//ans was the value that was before idx, so one value ahead of that is at idx.

int main(){
  int n;
  cin >> n;
  vector<int> v;
```

```
  for(int i = 0; i < n; i++)
    cin >> v[i];

  vector<int> comp = v;
  sort(comp.begin(), comp.end());
  comp.erase(unique(comp.begin(),comp.end()), comp.end());
  for(int i = 0; i < n; i++)
    v[i] = lower_bound(comp.begin(), comp.end(), v[i]) - comp.begin() + 1;

  for(int i = 0; i < n; i++)
    add(v[i], 1);

  //Now the fen vector is fully ready to behave as an indexed set.

  return 0;
}
```

2.1.5. Linked List

A linked list is a data structure, where ever element in a list 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 in a given position.

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

Here's the implementation of a linear linked list:

```cpp

struct Node{
  int val;// the value in the current element
  Node* nxt;//a pointer to the next element
  Node(const int& val = 0, Node* nxt = nullptr){//constructor
    this->val = val;
    this->nxt = nxt;
  }
};

struct List{
  Node* head;

  List(const int& size = 0, const int& val = 0){//creaing the linked list. The list will have "size" elements filled with val.
    head = new Node();
    Node* cur = head;
    for(int i = 1; i <= size; i++){
      cur->val = val;
      cur = cur->nxt = new Node();
    }
  }

  void insert(Node* pos, const int& val){//inserts a new node after poss
  Node* nxt = pos->nxt;
    pos->nxt = new Node(val,nxt);
  }

  void erase(Node* pos){//Erases the next node after pos
    if(pos->nxt == nullptr)
      return;
    Node* nxt = pos->nxt;
    pos->nxt = nxt->nxt;
    delete(nxt);
  }
```

```
}
```

Of course this Is a very poor implementation with not much memory safety leading to memory leaks. Fortunately `c++` has it's own implementation of a linked list.

2.1.5.1. `std::list`

Here's a cod examples on how the `c++` implementation of a linked list is used:

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

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

  list<int> l(n, 0);
  for(list<int>::iterator it = l.begin(); it != l.end(); it++){
    int x;
    cin >> x;
    *it = x;
  }

  for(list<int>::iterator it = l.begin(); it != l.end(); it++)// loop to erase every odd element(1-indexed).
    it = l.erase(it);

  for(list<int>::iterator it = l.begin(); it != l.end(); it++)
    l.insert(it, *it-1);//before each element insert the value of that element-1.

  for(list<int>::iterator it = l.begin(); it != l.end(); it++)
    cout << *it << " ";

  return 0;
}
```

Sample input:

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

Output:

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

As you can see from the code, if you want to store a value you simple update `*it`. If you want to insert a value before the current iterator, do `l.insert(it, val)`. Lastly if you want to erase the current iterator, do `l.erase(it)`. `erase()` also return 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 it's previous decision. Let's take a look at a question that can be solve with a greedy algorithm:

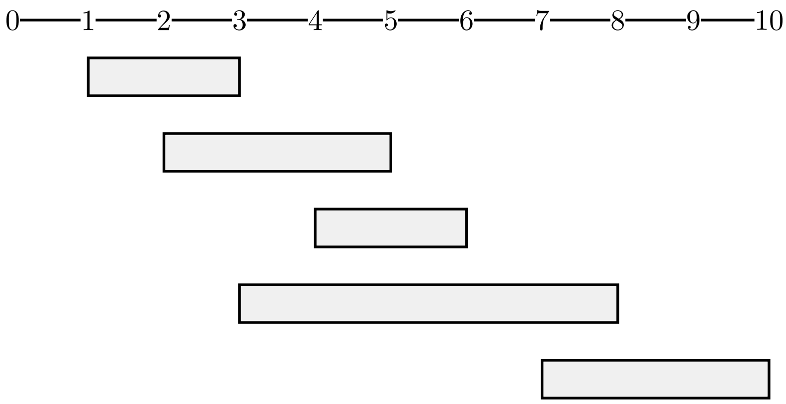
Question: You are given a list of events. Every event has a start time and an end time. You can only attend one event at a time. Your goal is to pick events in such a way so that you can attend the maximum number of events.

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 the ends later, at best case you can still pick the same number of new events. However at worst 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 start 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 can also pick $\{4, 6\}$. This is why it's always better to pick event's which end earlier because you have nothing to loose 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 questions.

(Add tag to Tasks and deadlines when documents are merged)

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:

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

int main(){

  int n, q;
  cin >> n >> q;
  set<int> s;

  for(int i = 0; i < n; i++){
    int x;
    cin >> x;
    s.insert(x);//Inserts a value into the set.
  }

  for(int i = 0; i < q; i++){
    int x;
    cin >> x;
    if(s.find(x) != s.end()){//s.find(x) returns the position of x in the set.
      cout << "YES" << endl;
      s.erase(x);//s.erase(x) removes x from the set
    }
    else
      cout << "NO" << endl;
  }

  for(set<int>::iterator it = s.begin(); it != s.end(); it++){
    cout << *it << " ";
  }
  return 0;
```

```
31 }
```

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 the 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 then 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. Lambda expressions

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

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

int main(){

  int n;
  cin >> n;

  function<int (int)> fact = [&] (int num){ // defining the lambda expression
    if(num == 1)
      return 1;
    return num * fact(num-1);
  };

  cout << fact(n) << endl;

  return 0;
}
```

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.

2.1.9. Sorting with a custom sorting order.

Say you with 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.

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

int main(){

  int arr[] = {3,4,6,2,5,1};
  vector<int> v = {6,2,4,5,1,3};

  sort(arr, arr+6, greater<int>());//Sorts the array {6,5,4,3,2,1}
  sort(v.begin(), v.end(), greater<int>());//Sorts the vector {6,5,4,3,2,1}

  return 0;
}
```

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:

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

int main(){

  int n;
  cin >> n;
  vector<pair<int,int>> v;
  for(int i = 0; i < n; i++)
    cin >> v[i];

  sort(v.begin(), v.end(), [](const pair<int,int> &a, const pair<int,int> &b){
    return a.second < b.second || a.second == b.second && a.first > b.first)
  });

  return 0;
}
```

2.2. CSES Practice Questions

2.2.1. Distinct Numbers

Question

Question - Distinct Numbers   Backup Link

Explanation

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

More about sets can be found here.

Solution

```cpp

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

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

  int n;
  cin >> n;
  set<int> s;
  for(int i = 0; i < n; i++){
    int x;
    cin >> x;
    s.insert(x);
  }
  cout << s.size() << endl;
  return 0;
}
```

2.2.2. Apartments

Question - Apartments   Backup Link

Explanation :

The algorithm sorts both applicants and apartments, then uses a two pointer approach to match each applicant with the smallest available apartment whose size differs by at most `k`. If an apartment is too small, move to the next apartment; if it's too large, move to the next applicant. This greedy method ensures the maximum number of matches efficiently.

Code :

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

int main() {
    int n, m, k;
    cin >> n >> m >> k;

    vector<int> applicants(n), apartments(m);

    // Read applicant preferences
    for (int i = 0; i < n; i++)
        cin >> applicants[i];

    // Read apartment sizes
    for (int i = 0; i < m; i++)
        cin >> apartments[i];

    // Sort both arrays
    sort(applicants.begin(), applicants.end());
    sort(apartments.begin(), apartments.end());

    int count = 0;
    int i = 0, j = 0;

    // Two-pointer approach to match applicants to apartments
    while (i < n && j < m) {
        // Check if current apartment fits current applicant's preference
        if (abs(applicants[i] - apartments[j]) <= k) {
            count++;
            i++;
            j++;
        }
```

```
        // If apartment is too small, try next apartment
        else if (applicants[i] - apartments[j] > k) {
            j++;
        }
        // If apartment is too big, try next applicant
        else {
            i++;
        }
    }

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

2.2.3. Ferris Wheel

Question - Ferris Wheel   Backup Link

Explanation :

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 are removed; otherwise, the heavier one goes alone. This greedy pairing minimizes the total number of gondolas.

Code :

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

int main() {
    int n, x;
    cin >> n >> x;

    vector<int> weights(n);
    for (int i = 0; i < n; i++) {
        cin >> weights[i];
    }

    // Sort the weights
    sort(weights.begin(), weights.end());

    int gondolas = 0;
    int left = 0, right = n - 1;

    while (left <= right) {
        // If heaviest and lightest can share a gondola
        if (weights[left] + weights[right] <= x) {
            left++;
            right--;
        }
        // Otherwise, heaviest gets their own gondola
        else {
            right--;
        }
        gondolas++;
    }

    cout << gondolas << endl;
```

```

    return 0;
}

```

2.2.4. Concert Tickets

Question - Concert Tickets   Backup Link

Intuitive Explanation :

Store all ticket prices in a multiset to keep them sorted and allow duplicates. Each customer gives an offer, and you use `upper_bound` 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 :

C++

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

int main() {
    int n, m, input;
    cin >> n >> m;

    // Multiset to store ticket prices in sorted order
    multiset<int> prices;

    // Store the m offers from customers
    vector<int> offers(m);

    // Insert the n ticket prices into the multiset
    for (int i = 0; i < n; i++) {
        cin >> input;
        prices.insert(input);
    }

    // Read all offers
    for (int i = 0; i < m; i++)
        cin >> offers[i];

    // For each offer, try to find the best possible ticket
    for (int i = 0; i < m; i++) {

        // Find the first price strictly greater than the offer
        auto it = prices.upper_bound(offers[i]);

        // If upper_bound points to begin(), no ticket <= offer exists
        if (it == prices.begin()) {
```

```
            cout << "-1" << endl;
        }
        else {
            // Move iterator to the largest price <= offer
            --it;

            // Output that price
            cout << *it << endl;

            // Remove that ticket so it can't be reused
            prices.erase(it);
        }
    }
}
```

2.2.5. Restaurant Customers

Question - Restaurant Customers   Backup Link

Explanation :

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 :

C++

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

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

    int n;
    cin >> n;
    vector<int> arrivals(n), departures(n);
    for (int i = 0; i < n; ++i) {
        cin >> arrivals[i] >> departures[i];
    }

    // Sort arrival and departure times
    sort(arrivals.begin(), arrivals.end());
    sort(departures.begin(), departures.end());

    int i = 0, j = 0, curr = 0, ans = 0;
    // Sweep through both arrays to find maximum overlap
    while (i < n && j < n) {
        if (arrivals[i] < departures[j]) {
            curr++;      // new guest arrives
            ans = max(ans, curr);
            i++;
        } else {
            curr--;      // a guest departs
            j++;
        }
    }

    cout << ans << "\n"; // maximum guests present at once
```

```
    return 0;
}

```

2.2.6. Movie Festival

Question - Movie Festival   Backup Link

Intuitive Explanation :

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 :

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

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

    // Store each movie as a pair (end_time, start_time)
    vector<pair<int, int>> movies(n);
    for (int i = 0; i < n; ++i) {
        int start, end;
        cin >> start >> end;
        movies[i] = {end, start};
    }

    // Sort movies by their ending time (greedy choice)
    sort(movies.begin(), movies.end());

    int maxMovies = 0;
    int currentEnd = 0;  // The end time of the last watched movie

    // Iterate through all movies
    for (auto [end, start] : movies) {
        // If the current movie starts after or exactly when the previous one ended
        if (start >= currentEnd) {
            maxMovies++;       // Watch this movie
            currentEnd = end;  // Update the end time to this movie's end
        }
    }

```

```
    cout << maxMovies << endl; // Output the result
    return 0;
}

```

2.2.7. Sum of Two Values

Question - Sum of Two Values   Backup Link

Explanation :

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; if too large, the right pointer moves left. This efficiently finds the correct pair in linear time after sorting.

Code :

C++

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

int main() {
    int n, target;
    cin >> n >> target;

    // Store each number along with its original index
    vector<pair<int, int>> nums(n);
    for (int i = 0; i < n; i++) {
        // number value
        cin >> nums[i].first;

        // original position (1-based index)
        nums[i].second = i + 1;
    }

    // Sort numbers by value to apply two-pointer technique
    sort(nums.begin(), nums.end());

    int left = 0, right = n - 1;
    while (left < right) {
        int sum = nums[left].first + nums[right].first;

        // If target sum found, print their original indices
        if (sum == target) {
            cout << nums[left].second << " " << nums[right].second;
            return 0;
        }
        // Move pointers based on comparison with target
        else if (sum < target) left++;
        else right--;
```

```
    }

    // If no valid pair found
    cout << "IMPOSSIBLE";
    return 0;
}

```

2.2.8. Maximum Subarray

Question - Maximum Subarray Sum   Backup Link

Intuitive Explanation :

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 keep adding to the current streak or start 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 :

C++

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

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

    vector<long long> arr(n);
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // max_current = maximum subarray sum ending at the current index
    // max_global = overall maximum subarray sum found so far
    long long max_current = arr[0];
    long long max_global = arr[0];

    // Iterate through the array
    for (int i = 1; i < n; i++) {
        // Either start a new subarray at arr[i] or extend the existing one
        max_current = max(arr[i], max_current + arr[i]);

        // Update the global maximum if needed
        max_global = max(max_global, max_current);
    }

    // Output the maximum subarray sum
    cout << max_global << endl;
    return 0;
}

```

```
32
```

2.2.9. Stick Lengths

Question - Stick Lengths   Backup Link

Intuitive Explanation :

The program minimizes the total adjustment cost to make all sticks equal in length. It sorts the stick lengths and picks the median as the target length since the median minimizes the sum of absolute differences. Unlike the mean, which minimizes squared differences, the median ensures minimal total movement for all sticks.

That might sound technical and complicated, so here's an easier way to picture it:

Intuitively, the median balances the values — half the sticks are shorter and half are longer — so adjusting everything toward it requires the least total effort. If you chose the mean, extreme stick lengths would pull the target toward them, increasing the total distance everyone else has to adjust, whereas the median stays steady and fair, unaffected by outliers.

Code :

C++

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

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

    vector<int> v(n);
    for (int i = 0; i < n; i++)
        cin >> v[i];

    // Sort the array in ascending order
    sort(v.begin(), v.end());

    // Choose the middle element (median) as the central value
    int c = v[v.size() / 2];

    long long ans = 0;
    // Calculate the total distance of all elements from the median
    for (int i = 0; i < n; i++)
        ans += abs(v[i] - c);

    // Output the minimum total distance
    cout << ans;
}
```

2.2.10. Missing Coin Sum

Question - Missing Coin Sum   Backup Link

Intuitive Explanation :

Sorting the Coins: By sorting the coins in non-decreasing order, we can process them greedily.

Greedy Approach: 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 current_max + 1.

Why This Works: If we can create all sums from 0 to `sumSoFar`, and 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 :

C++

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    vector<long long> coins(n);
    for (int i = 0; i < n; i++) cin >> coins[i];
    sort(coins.begin(), coins.end());

    long long sumSoFar = 0;
    // We can currently form all sums from 1 to sumSoFar.

    for (long long currCoin : coins) {
        // If the next needed sum is sumSoFar + 1 and currCoin is bigger,
        // we cannot fill that gap.
        if (currCoin > sumSoFar + 1) {
```

```
            cout << sumSoFar + 1 << "\n";
            return 0;
        }

        // Otherwise, currCoin helps us extend reachable sums.
        sumSoFar += currCoin;
    }

    // If all coins processed and no gap found, next unreachable sum is
    sumSoFar + 1.
    cout << sumSoFar + 1 << "\n";
    return 0;
}
```

2.2.11. Collecting Numbers

Question - Collecting Numbers   Backup Link

Explanation :

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 :

C++

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

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

    vector<int> position(n + 1);
    for (int i = 0; i < n; i++) {
        int value;
        cin >> value;
        position[value] = i;
    }

    int rounds = 1;
    for (int i = 2; i <= n; i++) {
        if (position[i] < position[i - 1]) {
            rounds++;
        }
    }

    cout << rounds;
}
```

2.2.12. Collecting Numbers II

Question - Collecting Numbers II   Backup Link

Explanation :

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 :

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

int n, m;
vector<int> arr;   // Stores the permutation
vector<int> pos;   // pos[x] = index where value x is currently located

// Checks whether x and x+1 form a "break"
// A break means x appears after x+1 in the array
bool isBreak(int x) {
    if (x < 1 || x >= n) return false;   // Out of valid range
    return pos[x] > pos[x + 1];
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    cin >> n >> m;

    arr.resize(n);
    pos.resize(n + 2);

```

```
    // Read the permutation and record positions of each value
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
        pos[arr[i]] = i;
    }

    // Initially, at least one round is needed
    int rounds = 1;

    // Count how many times x comes after x+1
    // Each such case increases the number of rounds
    for (int x = 1; x < n; x++) {
        if (isBreak(x)) {
            rounds++;
        }
    }

    // Process each swap query
    while (m--) {
        int a, b;
        cin >> a >> b;
        a--;
        b--;   // Convert to 0-based indexing

        int u = arr[a];
        int v = arr[b];

        // Only these values can affect the number of breaks
        // because other relative orders remain unchanged
        set<int> affected = {
            u - 1, u, u + 1,
            v - 1, v, v + 1
        };

        // Remove old breaks before swapping
        for (int x : affected) {
            if (isBreak(x)) {
                rounds--;
            }
        }

        // Perform the swap in the array
        swap(arr[a], arr[b]);

        // Update positions of the swapped values
```

```
        swap(pos[u], pos[v]);

        // Add new breaks after swapping
        for (int x : affected) {
            if (isBreak(x)) {
                rounds++;
            }
        }

        // Output the current number of rounds
        cout << rounds << "\n";
    }

    return 0;
}
```

2.2.13. Playlist

Question - Playlist   Backup Link

Explanation :

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 :

C++

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

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

    // Read the array
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    // Sliding window to find the longest subarray with all distinct elements.
    set<int> window;     // stores current distinct elements inside the window
    int left = 0;        // left pointer of the window
    int right = 0;       // right pointer of the window
    int bestLen = 0;     // maximum size found

    // Expand the window using 'right'
    while (right < n) {
        // If v[right] already exists, shrink the window from the left
        // until v[right] can be inserted without duplicates.
        if (window.count(v[right])) {
            window.erase(v[left]);
            left++;
        }
        else {
            // Element is unique in the window — include it
```

```
            window.insert(v[right]);

            // Update max length
            bestLen = max(bestLen, right - left + 1);

            // Move right pointer forward
            right++;
        }
    }

    cout << bestLen;
    return 0;
}
```

2.2.14. Towers

Question - Towers   Backup Link

Explanation :

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 :

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    multiset<int> tops; // stores the current top element of each tower

    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;

        // Find first tower whose top > x (we can place x on that tower)
        auto it = tops.upper_bound(x);

        if (it != tops.end()) {
            // Reuse this tower: remove old top and replace with x
            tops.erase(it);
        }
        // Start a new tower or update reused one with top = x
        tops.insert(x);
    }

    // Number of towers equals the number of distinct tops
    cout << tops.size() << "\n";

    return 0;
```

```
}
```

2.2.15. Traffic Lights

Question - Traffic Lights   Backup Link

Intuitive Explanation :

The program simulates cutting a stick of length `a` at `b` given positions. It uses a multiset ms to store all cut points and another multiset lens to track segment lengths. After each cut, it removes the old segment and adds two new ones. Finally, it prints the length of the largest segment remaining after each cut.

Code :

C++

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

int main() {
    int a, b, x;
    cin >> a >> b;

    multiset<int> ms, lens; // ms stores all cut positions, lens stores all segment lengths
    ms.insert(a); // rightmost boundary
    ms.insert(0); // leftmost boundary
    lens.insert(a); // initially one segment of length 'a'

    for (int i = 0; i < b; i++) {
        cin >> x;

        // Insert the new cut position and find its neighbors
        auto mid = ms.insert(x);
        auto first = prev(mid);
        auto last = next(mid);

        // Remove the old segment and add the two new smaller segments
        lens.erase(lens.find(*last - *first));
        lens.insert(*last - *mid);
        lens.insert(*mid - *first);

        // Output the largest segment length after each cut
        cout << *lens.rbegin() << " ";
    }
}

```

2.2.16. Distinct Values Subarrays

Question - Distinct Values Subarrays   Backup Link

Explanation :

This code uses a sliding window to count subarrays with all distinct elements. The right pointer expands the window, while a frequency map 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 :

C++

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

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

    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    // Frequency map for elements in the current window
    map<int, int> freq;

    int left = 0;           // Left pointer of the sliding window
    long long answer = 0;   // Total number of valid subarrays

    for (int right = 0; right < n; right++) {
        freq[a[right]]++;   // Add current element to the window

        // Shrink window until all elements are distinct
        while (freq[a[right]] > 1) {
            freq[a[left]]--;
            left++;
        }

        // Number of distinct subarrays ending at 'right'
        answer += (right - left + 1);
    }

    cout << answer;
```

```
}
```

2.2.17. Distinct Values Subsequences

Question - Distinct Values Subsequences   Backup Link

Explanation :

For each distinct value with `occ` occurrences, we have `(occc + 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

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

Code :

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

using ll = long long;
const int MOD = 1e9 + 7;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    // Count frequency of each distinct value
```

```
    map<ll, ll> freq;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        freq[x]++;
    }

    // For each distinct value, we can include 0, 1, 2, ..., or occ copies
    // This gives (occ + 1) choices per value; multiply all choices together
    ll ans = 1;
    for (auto [num, occ] : freq) {
        ans = (ans * (occ + 1)) % MOD;
    }

    // Subtract 1 to exclude the empty subsequence
    ans = (ans - 1 + MOD) % MOD;
    cout << ans << "\n";

    return 0;
}
```

2.2.18. Josephus Problem I

Question - Josephus Problem I   Backup Link

Explanation :

We store all people in a linked list, 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 :

C++

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

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

    list<int> circle;
    for (int i = 1; i <= n; i++)
        circle.push_back(i);

    auto it = circle.begin();

    while (!circle.empty()) {
        // move to the next person (skip one)
        it++;
        if (it == circle.end())
            it = circle.begin();

        cout << *it << " ";

        // erase returns iterator to next element
        it = circle.erase(it);

        if (it == circle.end() && !circle.empty())
            it = circle.begin();
    }

    return 0;
}
```

2.2.19. Josephus Problem II

Question - Josephus Problem II   Backup Link

Explanation :

Solution Overview:

- Uses a Fenwick Tree (Binary Indexed Tree) to efficiently track which positions are still active
- Each position is initially marked as “1” (active), and changed to “0” when removed

Key Operations:

1. update(idx, delta): Marks a position as active (+1) or removed (-1)
2. query(idx): Returns count of active elements from position 1 to idx
3. findKth(k): Uses binary search on the Fenwick Tree to find the k-th active element in O(log n) time

Algorithm Flow:

- Start with all n positions active
- In each iteration, move k steps forward in the circular list of remaining elements
- Use `findKth` to map from the circular index to the actual position
- Output that position and mark it as removed
- Repeat until all elements are removed

Complexity: O(n log n) - n removals, each taking O(log n) for finding and updating

Code :

C++

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

class FenwickTree {
    vector<int> tree;
    int n;
public:
    FenwickTree(int n) : n(n), tree(n + 1, 0) {}

    // Add delta to position idx (used to mark elements as active/inactive)
    void update(int idx, int delta) {
        for (; idx <= n; idx += idx & -idx)
            tree[idx] += delta;
    }

    // Get count of active elements in range [1, idx]
    int query(int idx) {
        int sum = 0;
        for (; idx > 0; idx -= idx & -idx)
```

```
            sum += tree[idx];
        return sum;
    }

    // Binary search to find the k-th active element (1-indexed)
    int findKth(int k) {
        int pos = 0, bit = 1;
        // Find the highest power of 2 <= n
        while (bit <= n) bit <<= 1;
        bit >>= 1;

        // Binary search from highest bit to lowest
        for (; bit > 0; bit >>= 1) {
            if (pos + bit <= n && tree[pos + bit] < k) {
                k -= tree[pos + bit];
                pos += bit;
            }
        }
        return pos + 1;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, k;
    cin >> n >> k;

    // Initialize Fenwick Tree and mark all positions as active
    FenwickTree ft(n);
    for (int i = 1; i <= n; i++)
        ft.update(i, 1);

    int remaining = n;  // Track how many elements are still active
    int idx = 0;        // Current position in the circular arrangement

    while (remaining > 0) {
        // Move k steps forward in circular manner
        idx = (idx + k) % remaining;

        // Find the actual position of the (idx+1)-th active element
        int pos = ft.findKth(idx + 1);
        cout << pos << " ";

```

```
        // Mark this position as removed
        ft.update(pos, -1);
        remaining--;
    }

    cout << "\n";
    return 0;
}
```

2.2.20. Nested Ranges Check

Question - Nested Ranges Check   Backup Link

Solution:

The algorithm uses a clever sorting trick: ranges are stored as ((l, -r), index) so that when sorted, ranges with the same left endpoint have larger right endpoints first. This ordering is crucial for the sweep line approach.

For detecting "contained" ranges, it sweeps left to right tracking the maximum right endpoint seen so far. If the current range's right endpoint is $\leq$ maxRight, it means this range is contained by a previous one (since they have l_current $\geq$ l_previous and r_current $\leq$ r_previous).

For detecting "contains" ranges, it sweeps right to left tracking the minimum right endpoint. If the current range's right endpoint is $\geq$ minRight, it contains at least one subsequent range (the range extends at least as far right as some range with a greater or equal left endpoint).

The time complexity is O(n log n) due to sorting, and space complexity is O(n). The key insight is that the special sorting allows both containment checks to be done in linear time after sorting.

```C++
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<pair<pair<int,int>, int>> ranges(n);
    for (int i = 0; i < n; i++) {
        int l, r;
        cin >> l >> r;
        ranges[i] = {{l, -r}, i};
    }
    sort(ranges.begin(), ranges.end());
    vector<int> contains(n, 0), contained(n, 0);
    int maxRight = INT_MIN;
    for (auto &[p, idx] : ranges) {
        int r = -p.second;
        if (r <= maxRight)
            contained[idx] = 1;
        maxRight = max(maxRight, r);
    }
    int minRight = INT_MAX;
    for (int i = n - 1; i >= 0; i--) {
```

```
        int r = -ranges[i].first.second;
        int idx = ranges[i].second;
        if (r >= minRight)
            contains[idx] = 1;
        minRight = min(minRight, r);
    }
    for (int x : contains) cout << x << " ";
    cout << "\n";
    for (int x : contained) cout << x << " ";
    cout << "\n";
}
```

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 it's 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 it's 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 it's 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.

Looking at the code should make it much clearer:

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

vector<int> fen;

struct Range{
  int l, r, idx;

  bool operator<(const Range& ran){
    return l < ran.l || l == ran.l && r > ran.r;
  }
};

void add(int x, int k){
  for(; x < fen.size(); x += x & -x)
    fen[x] += k;
}

int sum(int x){
  int ans = 0;
  for(; x > 0; x -= x & -x)
    ans += fen[x];
  return ans;
}

int main(){
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout.tie(0);
  //freopen("NestedRangesCount.in","r",stdin);
  //freopen("NestedRangesCount.out","w",stdout);
  int n;
  cin >> n;
```

```
    vector<Range> v(n);
    vector<int> comp;

    for(int i = 0; i < v.size(); i++){
      cin >> v[i].l >> v[i].r;
      v[i].idx = i;
      comp.push_back(v[i].r);
    }

    //Index compression for the right endpoints:
    sort(comp.begin(), comp.end());
    comp.erase(unique(comp.begin(), comp.end()), comp.end());

    for(int i = 0; i < v.size(); i++)
      v[i].r = lower_bound(comp.begin(), comp.end(), v[i].r) - comp.begin() +
      1;

    sort(v.begin(), v.end());
    fen.resize(comp.size() + 1);

    vector<int> c(n), d(n);// c[i] is the number of ranges v[i] contains, d[i]
    is the number of ranges that v[i] is contained by.

    for(int i = v.size()-1; i >= 0; i--){
      // for every range, add the number of ranges with r < v[i].r. l > v[i].r
      because of the sorting order.
      // if l = v[i].l , then the smaller ranges will get added first to
      correctly find c[i] because of the sorting order.
      // sum(v[i].r) - 1 gives that number.
      add(v[i].r,1);
      c[v[i].idx] = sum(v[i].r) - 1;
    }

    //Resetting the Fewnick tree.
    fen.clear();
    fen.resize(comp.size() + 1);

    for(int i = 0; i < v.size(); i++){
      // for every range, add the number of ranges with r > v[i].r. l < v[i].l
      because of the sorting order.
      // if l = v[i].l , then the larger ranges will get added first to
      correctly find d[i] because of the sorting order.
      // sum(fen.size() - 1) - sum(v[i].r-1) - 1 gives that number.
      add(v[i].r,1);
      d[v[i].idx] = sum(fen.size()-1) - sum(v[i].r-1) - 1;
```

```
  }

  for(int i = 0; i < c.size(); i++)
    cout << c[i] << " ";
  cout << endl;

  for(int i = 0; i < d.size(); i++)
    cout << d[i] << " ";
  cout << endl;

  return 0;
}
```

2.2.22. Room Allocation

Question - Room Allocation   Backup Link

Explanation :

We use a greedy algorithm by sorting customers by their 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 suitable free room exists, 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 :

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

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

    // Store each customer's booking: {start_time, end_time, original_index}
    vector<tuple<int, int, int>> bookings(n);

    for (int i = 0; i < n; i++) {
        int start, end;
        cin >> start >> end;
        bookings[i] = {start, end, i};
    }

    // Sort bookings by start time
    sort(bookings.begin(), bookings.end());

    // Track available rooms: {end_time, room_number}
    // Sorted by end_time to find rooms that become free earliest
    multiset<pair<int, int>> availableRooms;

    vector<int> assignedRoom(n);
    int totalRooms = 0;

    for (const auto& [start, end, originalIndex] : bookings) {
        int roomNumber;

        // Find a room that's free before this booking starts
```

```
        // upper_bound finds first room with end_time > start
        // We want the room with end_time <= start, so we go one back
        auto it = availableRooms.upper_bound({start, INT_MAX});

        if (it == availableRooms.begin()) {
            // No available room found - need a new room
            roomNumber = ++totalRooms;
        } else {
            // Reuse an existing room that's now free
            --it;
            roomNumber = it->second;
            availableRooms.erase(it);
        }

        // Mark this room as occupied until 'end' time
        availableRooms.insert({end, roomNumber});
        assignedRoom[originalIndex] = roomNumber;
    }

    // Output results
    cout << totalRooms << "\n";
    for (int room : assignedRoom) {
        cout << room << " ";
    }
    cout << "\n";

    return 0;
}
```

2.2.23. Factory Machines

Question - Factory Machines   Backup Link

Explanation :

The key idea is that the number of items produced increases monotonically with time, so we can binary-search the minimum time needed to make at least `t` items. For any guessed time `mid`, we compute how many items all machines together can produce by summing $\frac{\texttt{mid}}{v[i]}$. If the total is $\geq$ t, we try a smaller time; otherwise, we increase the time. This guarantees we find the earliest moment when production meets the target.

Code :

C++

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {

    ll n, t;
    cin >> n >> t;

    // v[i] = time taken by machine i to produce ONE item
    vector<ll> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    // Binary search on time.
    // low  = minimum possible time
    // high = a very large upper bound (1e18 works for all constraints)
    ll low = 1, high = 1e18, ans = -1;

    while (low <= high) {
        ll mid = (low + high) / 2;

        // Count how many items all machines can produce in 'mid' time
        ll total = 0;
        for (int i = 0; i < n; i++) {
            total += mid / v[i];

            // If already enough, stop early (avoid overflow + speedup)
            if (total >= t) break;
        }
```

```

        // If we can produce at least t items in 'mid' time,
        // try to find an even smaller valid time.
        if (total >= t) {
            ans = mid;
            high = mid - 1;
        }
        else {
            // Not enough items — need more time.
            low = mid + 1;
        }
    }

    cout << ans << "\n";
    return 0;
}
```

2.2.24. Tasks and Deadlines

Question - Tasks and Deadlines   Backup Link

Explanation :

In this problem, we must schedule tasks to maximize total reward, where each task gives a reward only if completed before its deadline.

The intuitive 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 chances of meeting deadlines. By sorting tasks by duration and maintaining a timeline, we ensure we fit the maximum number of tasks in the shortest possible time. Whenever adding a new task would exceed its deadline, we can replace the longest task in our schedule with it if it has a smaller duration. Thus, the algorithm minimizes wasted time and maximizes the number of completed tasks for optimal total reward.

Code :

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

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

    // jobs[i] = {duration, deadline}
    vector<pair<int, int>> jobs(n);
    for (int i = 0; i < n; i++) {
        cin >> jobs[i].first >> jobs[i].second;
    }

    // Sort by duration (or by first element), ensures earliest finishing attempts first
    sort(jobs.begin(), jobs.end());

    ll time_elapsed = 0;   // running sum of durations
    ll total_reward = 0;   // accumulated reward

    for (int i = 0; i < n; i++) {
        time_elapsed += jobs[i].first;              // finish this job at this time
        total_reward += jobs[i].second - time_elapsed;  // reward = deadline - completion time
    }
```

```

    cout << total_reward;
}
```

2.2.25. Reading Books

Question - Reading Books   Backup Link

Explanation :

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 $\max(\text{total\_time}, 2 * \text{longest\_book})$.

Code :

C++

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

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

    vector<long long> books(n);
    long long total = 0, max_book = 0;

    // Read book times, track:
    // 1) total time of all books
    // 2) the longest single book
    for (int i = 0; i < n; i++) {
        cin >> books[i];
        total += books[i];
        max_book = max(max_book, books[i]);
    }

    // Minimum total time is governed by:
    // - total (one person reading sequentially)
    // - OR twice the largest book (two-person parallel reading constraint)
    // The answer is the max of these two.
    cout << max(total, 2 * max_book) << "\n";

    return 0;
}
```

2.2.26. Sum of Three Values

Question - Sum of Three Values   Backup Link

Explanation :

This solution finds three numbers that sum to the target by first sorting the array and then fixing one element at a time. For each fixed element, it uses a two-pointer scan on the remaining range to efficiently search for a complementary pair. Sorting allows the sum to guide pointer movement, reducing the search from cubic to quadratic time. If no valid triple exists, the answer is declared impossible, keeping the logic clean and deterministic.

Code :

C++

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

int main() {
    int n, target;
    cin >> n >> target;

    // Store {value, original_index}
    vector<pair<int, int>> v(n);

    for (int i = 0; i < n; i++) {
        cin >> v[i].first;
        v[i].second = i + 1;  // keep original 1-based index
    }

    // Sort by value so we can use the two-pointer trick
    sort(v.begin(), v.end());

    // Fix one element v[i], then find two others using two pointers
    for (int i = 0; i < n - 2; i++) {
        int l = i + 1;       // left pointer
        int r = n - 1;       // right pointer

        while (l < r) {
            int sum = v[i].first + v[l].first + v[r].first;

            if (sum == target) {
                // Output original positions (not sorted ones)
                cout << v[i].second << " " << v[l].second << " " <<
                v[r].second;
                return 0;
```

```cpp
            }
            else if (sum < target) {
                l++;          // need a larger sum → move left pointer right
            }
            else {
                r--;          // need a smaller sum → move right pointer left
            }
        }
    }

    // If no triple found
    cout << "IMPOSSIBLE";
    return 0;
}
```

2.2.27. Sum of Four Values

Question - Sum of Four Values   Backup Link

Explanation :

If you understood the above three sum algorithm, four sum simply extends this by fixing two elements instead of one. You iterate through all pairs (i, j), then for each pair, use the same two-pointer technique to find the remaining two numbers that complete the target sum.

Code :

C++

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

int main() {
    int n, target;
    cin >> n >> target;

    // Store (value, original_index)
    vector<pair<int, int>> nums(n);

    for (int i = 0; i < n; i++) {
        cin >> nums[i].first;
        nums[i].second = i + 1;   // 1-based indexing as required by CSES
    }

    // Sort by value to enable two-pointer scanning later
    sort(nums.begin(), nums.end());

    // Fix first two values using indices i and j
    for (int i = 0; i < n - 3; i++) {
        for (int j = i + 1; j < n - 2; j++) {

            int left = j + 1;
            int right = n - 1;

            // Two-pointer search for remaining pair
            while (left < right) {
                long long sum = nums[i].first
                + nums[j].first
                + nums[left].first
                + nums[right].first;

                if (sum == target) {
```

```
                    // Output original positions
                    cout << nums[i].second << " "
                    << nums[j].second << " "
                    << nums[left].second << " "
                    << nums[right].second;
                    return 0;
                }

                // Adjust pointers based on sum size
                if (sum < target) {
                    left++;
                } else {
                    right--;
                }
            }
        }
    }

    cout << "IMPOSSIBLE";
    return 0;
}
```

2.2.28. Nearest Smaller Values

Question - Nearest Smaller Values   Backup Link

Intuitive Explanation :

We use a set of pairs (value, index) to maintain a sorted collection of elements seen so far. The lower_bound function efficiently locates the first element not smaller than the current value, allowing quick access to the previous smaller element by moving one step back. After each iteration, larger or equal elements are erased to maintain order and correctness.

Code :

C++

```cpp

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ll n, a;
    cin >> n;
    set<pair<ll,ll>> s;  // Stores pairs of (value, index) in sorted order by value

    for (int i = 0; i < n; i++) {
        cin >> a;

        // Find the first element in the set whose value >= current value 'a'
        auto it = s.lower_bound({a, -1});

        // If there's no smaller value, output 0
        if (it == s.begin()) cout << "0 ";
        else {
            // Otherwise, go one step back to get the last smaller element
            --it;
            cout << it->second + 1 << " ";  // Output its index (1-based)
        }

        // Remove all elements with value >= 'a' (not needed anymore)
        auto it2 = s.lower_bound({a, -1});
        s.erase(it2, s.end());

        // Insert current element (value, index)
        s.insert({a, i});
    }
}
```

```
33
```

2.2.29. Subarray Sums I

Question - Subarray Sums I   Backup Link

Explanation :

All numbers are positive, so we keep a sliding window: expand the right end, and whenever the sum exceeds x we shrink from the left until it fits. Each time the window sum equals x we have one valid subarray.

Code :

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ll n, x;
    cin >> n >> x;

    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    ll curr = 0;   // current window sum
    ll count = 0;  // number of subarrays equal to x
    ll l = 0;      // left pointer of sliding window

    for (int r = 0; r < n; r++) {
        curr += v[r];               // expand window to the right

        // shrink window from the left while sum exceeds x
        while (curr > x) {
            curr -= v[l];
            l++;
        }

        // if current window sum matches x, count it
        if (curr == x) count++;
    }

    cout << count;
}
```

2.2.30. Subarray Sums II

Question - Subarray Sums II   Backup Link

Explanation :

We iterate through the array while maintaining a running prefix sum. A map stores how many times each prefix sum has appeared so far. At each position, if a previous prefix sum equals `currentSum` − `targetSum`, a subarray with the required sum exists. We add its frequency to the answer and then record the current prefix sum. This counts all valid subarrays in linear time.

Code :

C++

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

using ll = long long;

int main() {
    ll n, targetSum;
    cin >> n >> targetSum;

    vector<ll> array(n);
    for (int i = 0; i < n; i++) {
        cin >> array[i];
    }

    // prefixSumCount[s] = how many times prefix sum 's' has occurred
    map<ll, ll> prefixSumCount;
    prefixSumCount[0] = 1;   // empty prefix

    ll currentSum = 0;
    ll subarrayCount = 0;

    for (int i = 0; i < n; i++) {
        currentSum += array[i];

        // count subarrays ending here with sum = targetSum
        subarrayCount += prefixSumCount[currentSum - targetSum];

        // record current prefix sum
        prefixSumCount[currentSum]++;
    }

```

```
    cout << subarrayCount << "\n";
    return 0;
}
```

2.2.31. Subarray Divisibility

Question - Subarray Divisibility   Backup Link

Explanation :

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 how many times each modulo value has appeared so far. If the same modulo appears again, the subarray between them has sum divisible by `n`. We add the frequency of the current modulo to the answer and update the map.

Code :

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

int main() {
    long long n;
    cin >> n;

    vector<long long> arr(n);
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
        arr[i] %= n;                  // keep values within modulo n
    }

    unordered_map<long long, long long> frequency;
    frequency[0] = 1;                // empty prefix sum

    long long prefixSum = 0;
    long long result = 0;

    for (int i = 0; i < n; i++) {
        prefixSum = (prefixSum + arr[i]) % n;
        if (prefixSum < 0) prefixSum += n;

        result += frequency[prefixSum];
        frequency[prefixSum]++;
    }

    cout << result;
    return 0;
}
```

2.2.32. Distinct Values Subarrays II

Question - Subarray Distinct Values   Backup Link

Explanation :

Use a sliding window with a frequency map. Expand the right end; if the number of distinct elements exceeds k, shrink from the left until it doesn't. Every position contributes `window_length` new subarrays ending there, so we add that to the answer.

Code :

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

int main() {
    long long n, k;
    cin >> n >> k;

    vector<long long> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    map<long long, int> freq;          // frequency of each element in current window
    long long left = 0, ans = 0, distinct = 0;

    for (int right = 0; right < n; right++) {
        // add right element to window
        if (++freq[v[right]] == 1) {
            distinct++;
        }

        // shrink window until we have at most k distinct elements
        while (distinct > k) {
            freq[v[left]]--;
            if (freq[v[left]] == 0) {
                distinct--;
            }
            left++;
        }

        // all subarrays ending at right with start in [left, right] are valid
```

```
        ans += (right - left + 1);
    }

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

2.2.33. Array Division

Question - Array Division   Backup Link

Explanation :

This solution minimizes the largest subarray sum when dividing the array into `k` consecutive subarrays using binary search. It establishes bounds where the answer lies between the largest element and the total sum of the array. For each candidate sum (mid), it checks if it's possible to split the array into at most `k` subarrays without any subarray exceeding that sum. If possible, a lower sum is attempted; otherwise, a higher sum is tested. Finally, the smallest feasible maximum subarray sum is output as the answer.

Code :

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, k;
    cin >> n >> k;
    vector<int> a(n);

    ll left = 0, right = 0;
    // 'left' = minimum possible max sum (largest single element)
    // 'right' = maximum possible max sum (sum of all elements)

    for (int i = 0; i < n; i++) {
        cin >> a[i];
        left = max(left, (ll)a[i]);
        right += a[i];
    }

    ll ans = right;   // Best answer found via binary search

    while (left <= right) {
        ll mid = (left + right) / 2;

        // Try to split into <= k subarrays where no subarray sum exceeds 'mid'
        int subarrays = 1;
```

```
        ll current_sum = 0;
        bool possible = true;

        for (int i = 0; i < n; i++) {
            // If adding this element exceeds 'mid', start a new subarray
            if (current_sum + a[i] > mid) {
                subarrays++;
                current_sum = a[i];

                // Too many subarrays ⇒ mid is too small
                if (subarrays > k) {
                    possible = false;
                    break;
                }
            } else {
                current_sum += a[i];
            }
        }

        if (possible) {
            ans = mid;        // mid works ⇒ try to lower the maximum sum
            right = mid - 1;
        } else {
            left = mid + 1;   // mid too small ⇒ increase
        }
    }

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

2.2.34. Sliding Median

Question - Sliding Median   Backup Link

Explanation :

Maintain two multisets: `low` holds the smaller half (including the median) and `high` holds the larger half. After each insert/remove we rebalance so that `low` is never smaller than `high` and differs by at most one element. The median is always the largest element of `low`.

Code :

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

multiset<int> lowSet, highSet;

void rebalance() {
    if (lowSet.size() > highSet.size() + 1) {
        highSet.insert(*lowSet.rbegin());
        auto it = lowSet.end();
        --it;
        lowSet.erase(it);
    } else if (lowSet.size() < highSet.size()) {
        lowSet.insert(*highSet.begin());
        highSet.erase(highSet.begin());
    }
}

void add(int x) {
    if (lowSet.empty() || x <= *lowSet.rbegin()) lowSet.insert(x);
    else highSet.insert(x);
    rebalance();
}

void remove_one(int x) {
    auto it = lowSet.find(x);
    if (it != lowSet.end()) lowSet.erase(it);
    else {
        it = highSet.find(x);
        highSet.erase(it);
    }
    rebalance();
}

```

```cpp
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, k;
    cin >> n >> k;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];

    for (int i = 0; i < k; i++) add(a[i]);
    cout << *lowSet.rbegin();

    for (int i = k; i < n; i++) {
        add(a[i]);
        remove_one(a[i - k]);
        cout << " " << *lowSet.rbegin();
    }
    cout << "\n";
    return 0;
}
```

2.2.35. Sliding Cost

Question - Sliding Cost   Backup Link

Explanation :

As in Sliding Median, keep two multisets split around the median but also track their sums. The total cost is `median * |low| - sumLow + sumHigh - median * |high|`. After each insertion/removal we rebalance and recompute using the maintained sums in O(1).

Code :

C++

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

multiset<int> lowSet, highSet;
ll sumLow = 0, sumHigh = 0;

void rebalance() {
    if (lowSet.size() > highSet.size() + 1) {
        int x = *lowSet.rbegin();
        sumLow -= x;
        lowSet.erase(prev(lowSet.end()));
        highSet.insert(x);
        sumHigh += x;
    } else if (lowSet.size() < highSet.size()) {
        int x = *highSet.begin();
        sumHigh -= x;
        highSet.erase(highSet.begin());
        lowSet.insert(x);
        sumLow += x;
    }
}

void add(int x) {
    if (lowSet.empty() || x <= *lowSet.rbegin()) {
        lowSet.insert(x);
        sumLow += x;
    } else {
        highSet.insert(x);
        sumHigh += x;
    }
    rebalance();
}
```

```

void remove_one(int x) {
    auto it = lowSet.find(x);
    if (it != lowSet.end()) {
        sumLow -= x;
        lowSet.erase(it);
    } else {
        it = highSet.find(x);
        sumHigh -= x;
        highSet.erase(it);
    }
    rebalance();
}

ll cost() {
    int med = *lowSet.rbegin();
    return med * 1LL * lowSet.size() - sumLow + sumHigh - med * 1LL *
    highSet.size();
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, k;
    cin >> n >> k;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];

    for (int i = 0; i < k; i++) add(a[i]);
    cout << cost();
    for (int i = k; i < n; i++) {
        add(a[i]);
        remove_one(a[i - k]);
        cout << " " << cost();
    }
    cout << "\n";
    return 0;
}
```

2.2.36. Movie Festival II

Question - Movie Festival II   Backup Link

Explanation :

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 latest watcher free at or before its start time. If such a watcher exists, we assign the movie and update their free time. This greedy process maximizes the total number of movies watched.

Code :

C++

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

int main() {
    int n, k;
    cin >> n >> k;

    // movies[i] = {end_time, start_time}
    vector<pair<int, int>> movies(n);
    for (int i = 0; i < n; i++) {
        cin >> movies[i].second >> movies[i].first;
    }

    // Sort movies by ending time (classic interval scheduling)
    sort(movies.begin(), movies.end());

    // Each element represents when a watcher becomes free
    multiset<int> freeAt;
    for (int i = 0; i < k; i++) {
        freeAt.insert(0);
    }

    int watched = 0;

    for (auto [endTime, startTime] : movies) {
        // Find a watcher who is free at or before startTime
        auto it = freeAt.upper_bound(startTime);

        if (it == freeAt.begin()) {
            // No watcher available
            continue;
        }
```

```

        // Assign this movie to the latest possible free watcher
        freeAt.erase(--it);
        freeAt.insert(endTime);
        watched++;
    }

    cout << watched;
    return 0;
}
```

2.2.37. Maximum Subarray Sum II

Question - Maximum Subarray Sum II   Backup Link

Explanation :

The code finds the maximum subarray sum whose length lies between `a` and `b`. It first builds a prefix sum array so any subarray sum can be computed in O(1). A multiset stores candidate prefix sums that can serve as valid subarray starts. As the right end moves forward, new valid prefixes are added to the set. Prefixes that would make the subarray longer than b are removed. The smallest prefix in the set gives the maximum possible subarray ending at the current index. The answer is updated by comparing all such valid subarrays efficiently.

Code :

C++

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

int main() {
    int n, minLen, maxLen;
    cin >> n >> minLen >> maxLen;

    vector<long long> values(n);
    for (int i = 0; i < n; i++) {
        cin >> values[i];
    }

    // Prefix sums: prefix[i] = sum of first i elements
    vector<long long> prefix(n + 1, 0);
    for (int i = 0; i < n; i++) {
        prefix[i + 1] = prefix[i] + values[i];
    }

    multiset<long long> candidates;
    long long bestSum = LLONG_MIN;

    for (int right = minLen; right <= n; right++) {
        // Add prefix corresponding to subarrays of length >= minLen
        candidates.insert(prefix[right - minLen]);

        // Remove prefix that would make subarray length > maxLen
        if (right > maxLen) {
            candidates.erase(candidates.find(prefix[right - maxLen - 1]));
        }
```

```cpp

        // Best subarray ending at 'right'
        bestSum = max(bestSum, prefix[right] - *candidates.begin());
    }

    cout << bestSum << "\n";
    return 0;
}
```