Phase 1 Remastered - Part 2

Progress Tracker

Leaderboard

Quiz Leaderboard

6.0 Sort Your Life Out: A Binary Search Saga

Expected Duration to Complete: 1.5 Week

Table of Contents

Video Class

Phase_1 Remastered Part_2 Class_6

Linear Search

Tutorial: link.

Problem: Given an array a of n integers and a number x, check if x is present in a or not.

Collapse With Style Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n, x; cin >> n >> x;
  vector<int> a(n);
  for (int i = 0; i < n; i++) {
    cin >> a[i];
  }
  // O(n)
  for (int i = 0; i < n; i++) {
    if (a[i] == x) {
      cout << "YES\n";
```

```
      return 0;
    }
  }
  cout << "NO\n";
  return 0;
}
```

Binary Search

Tutorial: link1, link2.

Problem: Given a **sorted** array a of n integers and a number x, check if x is present in a or not.

Algorithm:

- Start with the entire array as the **search interval**.
- Find the middle element and compare it to the target value:
  - If it's equal, the search is successful.
  - If the target is less than the middle element, continue the search on the left half.
  - If the target is greater than the middle element, continue the search on the right half.
- Repeat step 2 with the appropriate half of the array until the target is found or the search interval is empty.

The time complexity of binary search is $O(\log n)$ because each time we divide the array into half.

Solve this problem: link.

Recursive:

Collapse With Style Copy code

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

const int N = 1e5 + 9;
int a[N], x;
// search for x in a[l...r]
bool search(int l, int r) { // O(log n)
  if (l > r) return false;
  int mid = (l + r) / 2;
  if (a[mid] == x) return true;
  else if (x < a[mid]) {
    return search(l, mid - 1);
  }
  else {
    return search(mid + 1, r);
  }
}
```

```
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n, q; cin >> n >> q;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  while (q--) {
    cin >> x;
    bool found = search(1, n);
    if (found) {
      cout << "YES\n";
    }
    else {
      cout << "NO\n";
    }
  }
  return 0;
}
```

Iterative:

Collapse With Style

Copy code

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

const int N = 1e5 + 9;
int a[N], x;
// search for x in a[l...r]
bool search(int l, int r) { // O(log n)
  while (l <= r) { // run while there is at least one element in the search interval
    int mid = (l + r) / 2;
    if (a[mid] == x) {
      return true;
    }
    else if (x < a[mid]) {
      r = mid - 1;
    }
    else{
      l = mid + 1;
    }
  }
  return false;
}
```

```
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n, q; cin >> n >> q;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  while (q--) {
    cin >> x;
    bool found = search(1, n);
    if (found) {
      cout << "YES\n";
    }
    else {
      cout << "NO\n";
    }
  }
  return 0;
}
```

Also, fixing the search interval is really important.

A real life example of binary search would be searching a word in a dictionary.

Imagine you have a physical dictionary and you want to find the definition of a specific word. The words in the dictionary are sorted alphabetically.

1. You open the dictionary to the middle page and look at the first word on that page.
2. You compare the word you're looking for to the word in the middle:
   - If the word matches, you've found the word you were looking for.
   - If the word you're looking for comes before the word in the middle, you continue your search in the first half of the dictionary.
   - If the word you're looking for comes after the word in the middle, you continue your search in the second half of the dictionary.
3. You repeat this process, continually halving the section of the dictionary you're looking through until you find the word or conclude that it's not in the dictionary.

You can visualize binary search here.

Binary Search 0/1

Problem: Given a **sorted** array a of n integers where each element is either 0 or 1. Find the index of the first 1 in the array. If there is no 1 present in the array, output -1.

```
Collapse With Style                                                    Copy code
```

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  int a[n + 1];
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  int l = 1, r = n, ans = -1;
  while (l <= r) { // run while there is at least one element in the search interval
    int mid = (l + r) / 2;
    if (a[mid] == 1) {
      ans = mid;
      r = mid - 1;
    }
    else {
      l = mid + 1;
    }
    // do not set l = mid or r = mid
    // always set mid + 1 or mid - 1
    // what happens if you set l = mid or r = mid while doing while (l <= r) ?
    // try with l = 1 and r = 1 and you will find out
  }
  cout << ans << '\n';
  return 0;
}
```

Binary Search on Monotonic Functions

A function is monotonic if it is either entirely nonincreasing or nondecreasing.

Let's say we want to find the maximum or minimum value for which a certain condition is true and the condition is monotonic, that is, if the condition is true for a certain value, it will be true for all values greater or less than that value. In other words if we want to find the maximum or minimum value of x for which $f(x)$ is true, and $f(x)$ is monotonic, we can use binary search to find the answer.

For example, in the above 0/1 problem, we want to find the index of the first 1 in the array. Let $f(x)$ be the condition that there is a 1 at index x. $f(x)$ is monotonic because if there is a 1 at index x, there will be a 1 at all indices greater than x. So we can use binary search to find the answer.

Problem: Given an array $a_1, a_2, \ldots, a_n (1 \leq a_i \leq 10^9)$ of $1 \leq n \leq 10^5$ integers and a number $1 \leq s \leq 10^{14}$. Find the largest integer x for which $\sum_{i=1}^{n} \lfloor \frac{a_i}{x} \rfloor \geq s$. If there is no such x, output -1.

Let $f(x)$ be the condition that $\sum_{i=1}^{n} \lfloor \frac{a_i}{x} \rfloor \geq s$. $f(x)$ is monotonic because if $f(x)$ is true, then $f(x-1)$ is also true. So we can use binary search to find the answer.

Collapse With Style

Copy code

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

const int N = 1e5 + 9;
int a[N], n;
long long s;
bool f(int x) {
  long long sum = 0;
  for (int i = 1; i <= n; i++) {
    sum += a[i] / x;
  }
  return sum >= s;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cin >> n >> s;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  int l = 1, r = 1e9, ans = -1;
  while (l <= r) {
    int mid = (l + r) / 2;
    if (f(mid)) {
      ans = mid;
      l = mid + 1;
    }
    else {
      r = mid - 1;
    }
  }
  cout << ans << '\n';
  return 0;
}
```

Exercise: Solve this problem: Factory Machines

Binary Search on Answer

Sometimes we will have to find the maximum or minimum of some function but it is hard to find the maximum or minimum of the function directly, and it is easier to check if the value of the function is greater than or less than a certain value. In this case, we can use binary search to find the maximum or minimum value of the function.

Consider the following problem: Maximum Median

Finding the maximum median using at most k operations is very hard, but we can easily check if the median can be $\geq$ some value x. Let $f(x)$ be the condition that the median can be $\geq x$. $f(x)$ is monotonic because if $f(x)$ is true, then $f(x-1)$ is also true. So we can use binary search on the answer to find the maximum median.

Collapse With Style

Copy code

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

const int N = 2e5 + 9;
int a[N], n, k;

bool f(int x) { // check if the median can be >= x
  long long sum = 0;
  for (int i = n / 2 + 1; i <= n; i++) {
    if (a[i] < x) sum += x - a[i];
  }
  return sum <= k;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cin >> n >> k;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  sort(a + 1, a + n + 1);
  int l = 1, r = 2e9, ans = 0;
  while (l <= r) {
    int mid = l + (r - l) / 2; // using (l + r) / 2 can cause overflow here as r can be 2e9
    if (f(mid)) {
      ans = mid;
      l = mid + 1;
    }
    else {
      r = mid - 1;
    }
```

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

Lower Bound, Upper Bound and Binary Search Function

Tutorial:link.

Lower bound of x is the first element in the array that is $\geq x$. Upper bound of x is the first element in the array that is $> x$.

For example, consider the following array: $1, 2, 4, 4, 4, 6, 6, 9$. Lower bound of 4 is 2 and upper bound of 4 is 5.

Vectors in C++ have two functions called lower_bound and upper_bound. lower_bound returns an iterator to the first element in the array that is $\geq x$ and upper_bound returns an iterator to the first element in the array that is $> x$. If there is no such element, then lower_bound returns an iterator to the end of the array and upper_bound returns an iterator to the end of the array. The time complexity of both of these functions is $O(log(n))$.

Also, there is a function called binary_search that returns true if x is present in the array and false otherwise. It works in $O(log(n))$ time.

Collapse With Style Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  vector<int> v({1, 2, 4, 4, 4, 6, 6, 9});
  int x = 4;
  if (binary_search(v.begin(), v.end(), x)) {
    cout << "YES\n";
  }
  auto it = lower_bound(v.begin(), v.end(), x); // first position such that v[i] >= x
  cout << (it - v.begin()) << '\n'; // 2
  it = upper_bound(v.begin(), v.end(), x); // first position such that v[i] > x
  cout << (it - v.begin()) << '\n'; // 5

  int count_of_x = upper_bound(v.begin(), v.end(), x) - lower_bound(v.begin(), v.end(), x)
  cout << count_of_x << '\n'; // 3

  int l = 4, r = 6;
  int count_of_numbers_in_range = upper_bound(v.begin(), v.end(), r) - lower_bound(v.begin
  cout << count_of_numbers_in_range << '\n'; // 5
```

```
  return 0;
}
```

Binary Search on MinMax Functions

Sometimes you will have to minimize some maximum values or maximize some minimum values. In most of this cases, you can use binary search to find the answer.

Consider the following problem: Array Division

We have to divide the array into k subarrays such that the maximum sum of a subarray is minimized. Let $f(x)$ be the condition that the maximum sum of a subarray is $\leq x$. $f(x)$ is monotonic because if $f(x)$ is true, then $f(x+1)$ is also true. So we can use binary search to find the answer.

Collapse With Style Copy code

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

const int N = 2e5 + 9;
int a[N], n, k;

bool f(long long x) { // check if the maximum sum of a subarray is <= x
  for (int i = 1; i <= n; i++) {
    if (a[i] > x) return false;
  }
  long long sum = a[1];
  int cnt = 1;
  for (int i = 2; i <= n; i++) {
    sum += a[i];
    if (sum > x) {
      cnt++;
      sum = a[i];
    }
  }
  return cnt <= k;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cin >> n >> k;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  long long l = 1, r = 2e14, ans = 0;
```

```
  while (l <= r) {
    long long mid = (l + r) / 2;
    if (f(mid)) {
      ans = mid;
      r = mid - 1;
    }
    else {
      l = mid + 1;
    }
  }
  cout << ans << '\n';
  return 0;
}
```

You can do similar things to maximize some minimum values. In those cases, you can define a function to check if the minimum value is $\geq$ some value x and do binary search on that.

In general, if you have to minimize some maximum values or maximize some minimum values, you can use binary search to find the answer.

Binary Search For Kth Smallest/Largest Values

Sometimes you will have to find the kth smallest/largest value of some function. In most of this cases, you can use binary search to find the answer.

Consider the following problem: K-th Not Divisible by n

We have to find the kth smallest value that is not divisible by n. Let $f(x)$ be the condition that the number of values from 1 to x that are not divisible by n is $\geq k$. $f(x)$ is monotonic because if $f(x)$ is true, then $f(x+1)$ is also true. So we can use binary search to find the answer.

Collapse With Style Copy code

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

int n, k;
bool f(int x) { // check if the number of values from to 1 to x that are not divisible by n
  int not_divisible = x - x / n;
  return not_divisible >= k;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int tc;
  cin >> tc;
```

```
  while (tc--) {
    cin >> n >> k;
    int l = 1, r = 2e9, ans = 0;
    while (l <= r) {
      int mid = l + (r - l) / 2;
      if (f(mid)) {
        ans = mid;
        r = mid - 1;
      }
      else {
        l = mid + 1;
      }
    }
    cout << ans << '\n';
  }
  return 0;
}
```

Similar Problem: K-th Sum

Solution:

Collapse With Style

Copy code

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

const int N = 1e5 + 9;
int a[N], b[N], n;
long long k;
// O(logn)
int get_count(int lim) {
  // count how many elements in b which are <= lim
  // b is sorted
  // int l = 1, r = n, ans = 0;
  // while (l <= r) {
  //   int mid = (l + r) / 2;
  //   if (b[mid] <= lim) {
  //     ans = mid;
  //     l = mid + 1;
  //   }
  //   else {
  //     r = mid - 1;
  //   }
  // }
  // return ans;
  return upper_bound(b + 1, b + n + 1, lim) - b - 1;
```

```
}
// O(n logn)
bool f(int x) { // is the answer (kth smallest sum) <= x ?
  long long cnt = 0;
  for (int i = 1; i <= n; i++) {
    // count how many j's are there such that
    // a[i] + b[j] <= x => b[j] <= x - a[i]
    cnt += get_count(x - a[i]);
  }
  return cnt >= k;
}
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cin >> n >> k;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (int i = 1; i <= n; i++) {
    cin >> b[i];
  }
  sort(a + 1, a + n + 1);
  sort(b + 1, b + n + 1);
  // vector<int> v;
  // for (int i = 1; i <= n; i++) {
  //   for (int j = 1; j <= n; j++) {
  //     v.push_back(a[i] + b[j]);
  //   }
  // }
  // sort(v.begin(), v.end());
  // cout << v[k - 1] << '\n';
  int l = 0, r = 2e9, ans = 0;
  while (l <= r) {
    int mid = l + (r - l) / 2;
    if (f(mid)) {
      ans = mid;
      r = mid - 1;
    }
    else {
      l = mid + 1;
    }
  }
  // O(30 * n logn)
  cout << ans << '\n';
  return 0;
```

```
}
```

Binary Search on Doubles

For doing binary search when the search space is in doubles (so the answer is also a double), we will have to do things a bit differently.

Consider the following problem: Equation.

We have to find the minimum value of x such that $x^2 + \sqrt{x} = c$. Let $f(x) = x^2 + \sqrt{x}$. $f(x)$ is monotonic. So we can use binary search to find the answer.

Collapse With Style Copy code

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

double f(double x) {
  return x * x  + sqrt(x);
}
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  double c; cin >> c;
  double l = 0, r = 1e5, ans;
  int iterations = 100; // running binary search for 100 iterations should be enough for th
   // while (r - l > 1e-7) { // another approach: until the l and r are 10^-7 difference ap
  while (iterations--) {
    double mid = (l + r) / 2;
    if (f(mid) >= c) {
      ans = mid;
      r = mid; // not mid +/- 1
    }
    else {
      l = mid; // not mid +/- 1
    }
  }
  cout << fixed << setprecision(10) << ans << '\n';
  return 0;
}
```

Bubble Sort

Tutorial: link.

- Repeat $n-1$ times:
  - For each index i from 1 to $n-1$:
    - If $a[i] > a[i+1]$, swap $a[i]$ and $a[i+1]$.
- So basically, we are moving the largest element to the end of the array in each iteration.
- The name "Bubble Sort" comes from the way the algorithm works. In the process of sorting, the largest unsorted element "bubbles up" to its correct position in the array with each iteration through the array. The visual effect of this process resembles bubbles rising to the surface of a liquid, and that's where the algorithm gets its name.
- Time complexity: $O(n^2)$.
- Extra space: $O(1)$.

Collapse With Style Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  int a[n + 1];
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (int step = 1; step < n; step++) {
    for (int i = 1; i < n; i++) {
      if (a[i] > a[i + 1]) {
        swap(a[i], a[i + 1]);
      }
    }
  }
  for (int i = 1; i <= n; i++) {
    cout << a[i] <<' ';
  }
  return 0;
}
```

Visualize Sorting Algorithms

You can visualize how all sorting algorithms work. Go to this link and enjoy!

Selection Sort

Tutorial: link.

- Repeat $n-1$ times:
  - Find the minimum element in the unsorted part of the array.
  - Swap it with the first element of the unsorted part of the array.
- Time complexity: $O(n^2)$.
- Extra Space: $O(1)$.

Collapse With Style Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  int a[n + 1];
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (int i = 1; i < n; i++) {
    int min_index = i;
    for (int j = i + 1; j <= n; j++) {
      if (a[j] < a[min_index]) {
        min_index = j;
      }
    }
    swap(a[i], a[min_index]);
  }
  for (int i = 1; i <= n; i++) {
    cout << a[i] <<' ';
  }
  return 0;
}
```

Insertion Sort

Tutorial: link.

- For each index i from 2 to n:

- Store $a[i]$ in a variable key.
- Move all the elements from 1 to $i-1$ that are greater than key one position right of their current position.
- Insert key at the correct position in the array.

- Time complexity: $O(n^2)$.
- Extra Space: $O(1)$.

Collapse With Style

Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  int a[n + 1];
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (int i = 2; i <= n; i++) {
    int key = a[i];
    int j = i - 1;
    while (j >= 1 && a[j] > key) {
      a[j + 1] = a[j];
      j--;
    }
    a[j + 1] = key;
  }
  for (int i = 1; i <= n; i++) {
    cout << a[i] <<' ';
  }
  return 0;
}
```

Counting Sort

Tutorial: link.

- Create a new array `cnt` of size `max + 1` where `max` is the maximum element of the array.
- For each element x in the array, increment `cnt[x]`.
- For each element i from 1 to `max` print i `cnt[i]` times.

- Time complexity: $O(n + max)$.
- Extra Space: $O(max)$.

Collapse With Style

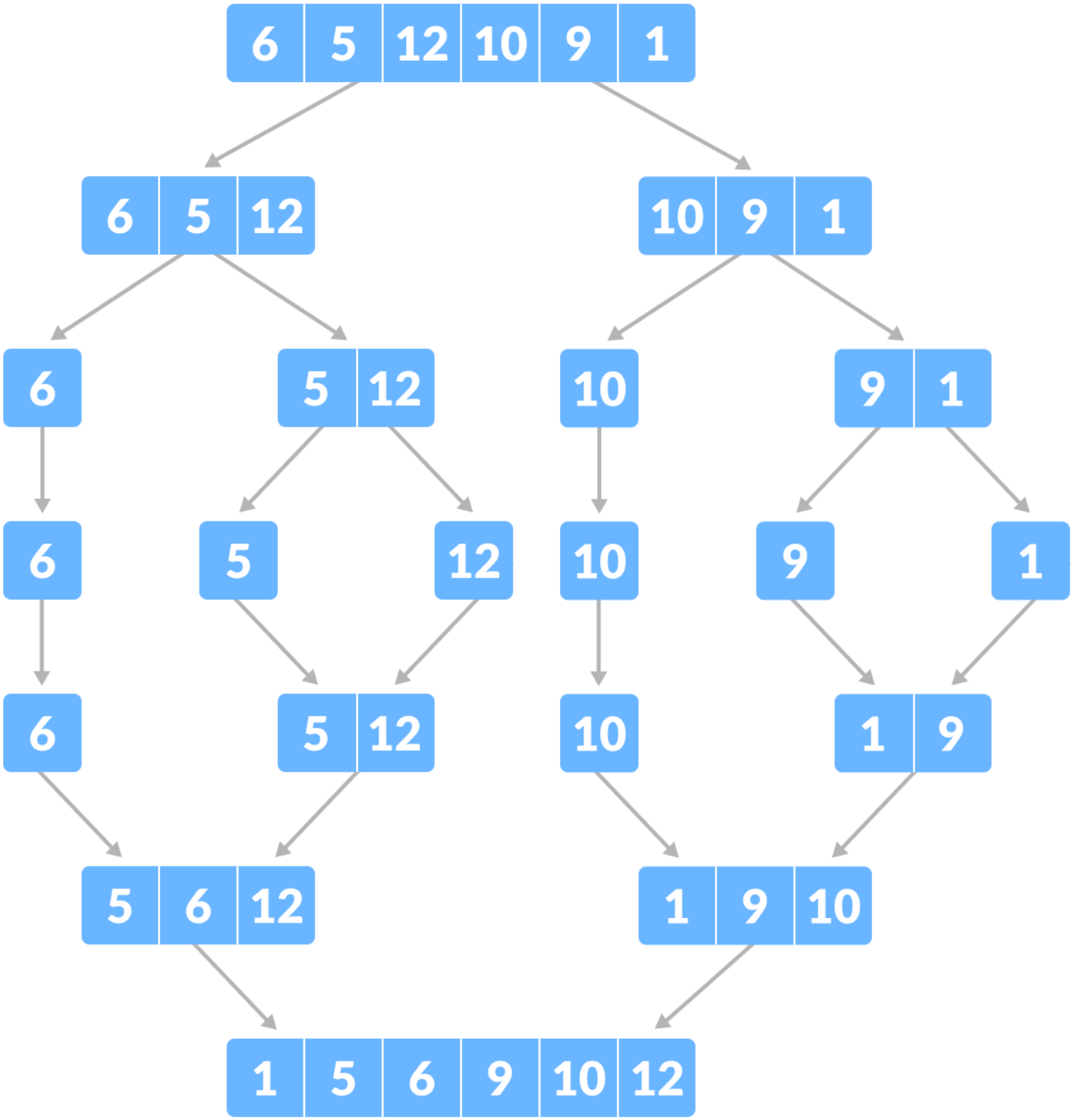
Copy code

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

const int MAX = 1e6; // max element of the array
int cnt[MAX + 1];
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  int a[n + 1];
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (int i = 1; i <= n; i++) {
    cnt[a[i]]++;
  }
  for (int i = 1; i <= MAX; i++) {
    for (int j = 1; j <= cnt[i]; j++) {
      cout << i << ' ';
    }
  }
  return 0;
}
```

Merge Sort

Tutorial: link



- **Divide:** Begin by dividing the original unsorted array into two halves. If the array has an odd number of elements, one half will have one more element than the other.
- **Conquer:** Sort the two subarrays recursively using merge sort.
- **Combine:** Merge the two sorted subarrays to produce the sorted array of n elements.
- Time complexity: $O(n \log n)$.
- Extra Space: $O(n)$.
- Solve this problem using merge sort: Merge sort

Collapse With Style

Copy code

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

// given two sorted arrays l and r, merge them into one sorted array
vector<int> merge(vector<int> &l, vector<int> &r) {
  int n = l.size(), m = r.size();
  vector<int> ans;
  int i = 0, j = 0;
  while (i < n && j < m) {
    if (l[i] < r[j]) ans.push_back(l[i++]);
    else ans.push_back(r[j++]);
  }
  while (i < n) ans.push_back(l[i++]);
  while (j < m) ans.push_back(r[j++]);
  return ans;
}

vector<int> a;
vector<int> merge_sort(int l, int r) {
  if (l == r) return {a[l]};
  int mid = l + r >> 1;
  vector<int> left = merge_sort(l, mid);
  vector<int> right = merge_sort(mid + 1, r);
  return merge(left, right);
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  a.resize(n);
  for (int i = 0; i < n; i++) {
    cin >> a[i];
  }
  vector<int> ans = merge_sort(0, n - 1);
  for (int i = 0; i < n; i++) {
    cout << ans[i] << ' ';
  }
  return 0;
}
```

You can visualize merge sort here.

Divide and Conquer

A typical Divide and Conquer algorithm solves a problem using following three steps.

- **Divide:** Break the given problem into subproblems of same type.
- **Conquer:** Recursively solve these subproblems
- **Combine:** Appropriately combine the answers

Fact: You already know this! Merge sort uses a divide and conquer algorithm.

Another example problem would be finding number of inversions in an array. You will find my solution using a divide and conquer approach in the video tutorial of this class.

Sound of Sorting

You can listen to the sound of sorting algorithms. Go to this link and enjoy! This is just for fun, you don't need to understand anything from this video.

C++ Sort

Of course, you can always use C++ sort function to sort an array. The time complexity of C++ sort is $O(n \log n)$.

Lambda Functions

C++ Lambda functions are a concise way to create anonymous (unnamed) functions right within the body of a function. You can use it when you need a short function that you don't want to define it elsewhere in your code.

The general syntax of a lambda function is:

Collapse With Style Copy code

```
[capture](parameters) -> return_type {
    // code
}
```

- **Capture**: Determines how variables from outside the lambda function can be accessed within it.
- **Parameters**: Specifies the parameters, like in a regular function.
- **Return Type**: (Optional) Specifies the return type. If omitted, it's inferred from the return statements in the lambda.
- **Code**: The code inside the lambda function.

Capture modes enable access to variables from the enclosing scope.

- `[]` : Captures nothing.
- `[=]` : Captures all local variables by value.

- `[&]`: Captures all local variables by reference.
- `[a, &b]`: Captures variable `a` by value and variable `b` by reference.

Collapse With Style Copy code

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

int main() {
    // Example 1: Capture by Value
    int x = 10;
    auto lambda_by_value = [=](int a) -> int {
        // you can't change x in this lambda function
        // x is always 10 even if you change it outside later
        return a + x;
    };
    cout << "Capture by value: " << lambda_by_value(5) << endl; // Output: 15

    // Example 2: Capture by Reference
    auto lambda_by_reference = [&](int a) -> int {
        // you can change x in this lambda function
        // x is the same as the one outside at any moment
        return a + x;
    };

    cout << "Capture by reference: " << lambda_by_reference(5) << endl; // Output: 15
    x = 20;
    cout << "Capture by reference after changing x: " << lambda_by_reference(5) << endl; /

    // Example 3: Mixed Capture
    int a = 10, b = 20;
    auto mixed_capture = [a, &b](int y) -> int {
        return y + a + b;
    };
    cout << "Mixed capture: " << mixed_capture(5) << endl; // Output: 35
    a = 20; b = 30;
    cout << "Mixed capture after changing b: " << mixed_capture(5) << endl; // Output: 45

    // Example 4: Capture Nothing
    auto no_capture = [](int a, int b) -> int {
        // can't access any variable from the outside
        return a * b;
    };
    cout << "Capture nothing: " << no_capture(5, 6) << endl; // Output: 30
```

```
    // Example 5: Auto Return Type (Inferred)
    auto inferred_return = [x](int a) {
        return a * x; // Return type inferred as int
    };
    cout << "Inferred return type: " << inferred_return(5) << endl; // Output: 50

    return 0;
}
```

I use lambda functions while using custom comparators in C++ (see below) or when I need to write a quick function that uses lots of local variables and I don't want to pass these variables as parameters and don't wanna write them as global variables either.

For example:

Collapse With Style Copy code

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

// this code takes a 2D array as input
// flattens it and prints it in one line
int main() {
    int n, m; cin >> n >> m;
    auto get_id = [&](int i, int j) { // get id for cell (i, j)
        return i * m + j;
    };
    int f[n * m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            int x; cin >> x;
            f[get_id(i, j)] = x;
        }
    }
    for (int i = 0; i < n * m; i++) {
        cout << f[i] << ' ';
    }
    return 0;
}
```

You can use it as per your needs.

Custom Comparator

A custom comparator is a function or callable object that defines how two elements should be compared during sorting. It returns a boolean value, determining the order of the elements. Custom comparators provide a flexible

way to define your own sorting rules in C++.

You can define a custom comparator as a function or a lambda expression and pass it as the third argument to `sort`.

Collapse With Style Copy code

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

// Custom comparator as a function
bool cmp(pair<int, int> a, pair<int, int> b) {
  // Sort by first element in descending order
  // If first elements are equal, sort by second element in ascending order
  if (a.first != b.first) return a.first > b.first;
  return a.second < b.second;
}

int main() {
  vector<pair<int, int>> v = {{1, 2}, {2, 1}, {2, 3}, {1, 1}, {3, 2}};
  sort(v.begin(), v.end(), cmp);
  for (auto p : v) {
    cout << p.first << ' ' << p.second << '\n';
  }
  cout << '\n';
  // Custom comparator as a lambda expression
  sort(v.begin(), v.end(), [](pair<int, int> a, pair<int, int> b) { // no capture method i
    // Sort by second element in descending order
    // If second elements are equal, sort by first element in ascending order
    if (a.second != b.second) return a.second > b.second;
    return a.first < b.first;
  });
  for (auto p : v) {
    cout << p.first << ' ' << p.second << '\n';
  }
  return 0;
}
```

Strict Weak Ordering

In C++, the comparator function must follow the strict weak ordering rule. It means that the comparator function must return false for equal elements. If the comparator returns true for equal elements, the sort function will not work properly.

For example, the following comparator function does not follow the strict weak ordering rule:

Collapse With Style Copy code

```
bool cmp(int a, int b) {
  return a <= b;
}
```

This comparator function returns true for equal elements. So, the sort function will not work properly with this comparator. This might give you a wrong answer or even a runtime error.

ALWAYS RETURN FALSE FOR EQUAL ELEMENTS IN A COMPARATOR FUNCTION. link

Problem: The Smallest String Concatenation

Consider the following solution:

Collapse With Style Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  string s[n + 1];
  for (int i = 1; i <= n; i++) {
    cin >> s[i];
  }
  sort (s + 1, s + n + 1, [](string a, string b) {
    return a + b <= b + a;
  });
  for (int i = 1; i <= n; i++) {
    cout << s[i];
  }
  cout << '\n';
  return 0;
}
```

Link to my submission: link

This gives runtime error. Why? Because the comparator function returns true for equal elements. So, the sort function will not work properly with this comparator.

To fix this, we can change the comparator function to:

Collapse With Style Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  string s[n + 1];
  for (int i = 1; i <= n; i++) {
    cin >> s[i];
  }
  sort (s + 1, s + n + 1, [](string a, string b) {
    return a + b < b + a;
  });
  for (int i = 1; i <= n; i++) {
    cout << s[i];
  }
  cout << '\n';
  return 0;
}
```

Link to my submission: link

This gives AC!

Fix It (Again)

In a previous class, we went through this. For problems like doing something on all pairs, all subarrays, all triplets, etc., you can just fix one of the elements and do something on the rest of the elements. This is a very common technique to solve problems. Sometimes you may need to fix two or more elements.

- Problem: Sum of Two Values
  - **Solution:**

```cpp
Collapse With Style                                                    Copy code

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

const int N = 2e5 + 9;
int a[N];

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

```
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    map<int, int> mp;
    for (int i = 1; i <= n; i++) {
        // fix that a[i] is the right element
        // now we need to find x - a[i] in the previous elements
        if (mp.find(x - a[i]) != mp.end()) {
            cout << mp[x - a[i]] << ' ' << i << '\n';
            return 0;
        }
        mp[a[i]] = i;
    }
    cout << "IMPOSSIBLE\n";
    return 0;
}
// O(n log n)
```

- Problem: Subarray Divisibility
  - **Solution:**

Collapse With Style — Copy code

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

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; cin >> n;
  int a[n + 1], pref[n + 1];
  pref[0] = 0;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
    pref[i] = pref[i - 1] + a[i];
    pref[i] %= n;
    pref[i] = (pref[i] + n) % n;
  }
  long long ans = 0;
  map<int, int> mp;
  mp[pref[0]]++;
  for (int i = 1; i <= n; i++) {
    ans += mp[pref[i]];
    mp[pref[i]]++;
  }
  cout << ans << '\n';
```

```
  return 0;
}
// O(n log n)
```

Greedy With Sorting

We have already discussed some easy greedy problems that can be solved using sorting in a previous class. Here are some more problems that can be solved using sorting.

- Problem: Movie Festival
  - **Wrong Greedy:** Always choose the movie that starts first. This is wrong. Consider the following test case:

    ```
    4
    1 10
    2 4
    6 7
    8 9
    ```

    The optimal answer is 3, but the above greedy gives 1.
  - **Correct Greedy:** Always choose the movie that ends first. This is correct because if we choose a movie that ends later, we will always have more options to choose movies later.

Collapse With Style — Copy code

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

const int N = 2e5 + 9;
int a[N], b[N];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i] >> b[i];
    }
    vector<pair<int, int>> v;
    for (int i = 1; i <= n; i++) {
        v.push_back({b[i], a[i]});
    }
    sort(v.begin(), v.end());
    int ans = 0, last = 0;
    for (auto x: v) {
        if (x.second >= last) {
            ans++;
            last = x.first;
        }
```

```
    }
    cout << ans << '\n';
    return 0;
}
// O(n log n)
```

- Problem: Stick Lengths

  In this problem, we need to make all the sticks equal. We can make all the sticks equal to the median stick and it is the optimal answer. Why?

  Solution:

Collapse With Style Copy code

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

const int N = 2e5 + 9;
int a[N];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    sort(a + 1, a + n + 1);
    int median = a[(n + 1) / 2];
    long long ans = 0;
    for (int i = 1; i <= n; i++) {
      ans += abs(a[i] - median);
    }
    cout << ans << '\n';
    return 0;
}
// O(n log n)
```

- Problem: Tasks and Deadlines

  So all deadlines will get added to the total time. So we will have to minimize the sum of the times taken to complete the tasks. So we will sort the tasks in increasing order of the time taken to complete them. Then we will take the tasks in this order and add the time taken to complete them to the total time. So the total time will be minimized.

  Solution:

Collapse With Style Copy code

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

const int N = 2e5 + 9;
int a[N], b[N];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i] >> b[i];
    }
    vector<pair<int, int>> v;
    for (int i = 1; i <= n; i++) {
        v.push_back({a[i], b[i]});
    }
    sort(v.begin(), v.end());
    long long ans = 0, cur = 0;
    for (auto x: v) {
        cur += x.first;
        ans += x.second - cur;
    }
    cout << ans << '\n';
    return 0;
}
// O(n log n)
```

- Problem: Towers

  In this problem, we need to find the minimum number of towers required to store all the boxes. We can use a greedy approach. For each tower, we will try to put the box in the tower that has the smallest height and can store the box. If no tower can store the box, we will create a new tower. We can use a multiset to store the heights of the towers.

  Solution:

Collapse With Style Copy code

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

const int N = 2e5 + 9;
int a[N];

int main() {
    int n;
```

```
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    multiset<int> tower_tops;
    for (int i = 1; i <= n; i++) {
        auto it = tower_tops.upper_bound(a[i]);
        if (it != tower_tops.end()) {
            tower_tops.erase(it);
        }
        tower_tops.insert(a[i]);
    }
    cout << tower_tops.size() << '\n';
    return 0;
}
// O(n log n)
```

- Problem: Stick Divisions

  In this problem, we need to find the minimum cost to divide the sticks into pieces of given lengths. We can use a greedy approach. We will always choose the two smallest sticks and merge them. We can use a priority queue to store the lengths of the sticks.

  Solution:

Collapse With Style Copy code

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

const int N = 2e5 + 9;
int a[N];

int main() {
   ios_base::sync_with_stdio(0);
    cin.tie(0);
    int n, x;
    cin >> x >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    priority_queue<int, vector<int>, greater<int>> q;
    for (int i = 1; i <= n; i++) {
        q.push(a[i]);
    }
    long long ans = 0;
    while (q.size() > 1) {
```

```
        int x = q.top(); q.pop();
        int y = q.top(); q.pop();
        ans += x + y;
        q.push(x + y);
    }
    cout << ans << '\n';
    return 0;
}
// O(n log n)
```

Video Session

Session 6 - Phase 1 Remastered