Phase 1 Remastered - Part 2

Progress Tracker

Leaderboard

Quiz Leaderboard

7.0 Two Pointers to All The Mistakes I Made

Expected Duration to Complete: 1.5 Week

Table of Contents

Video Class

Divisors

The Naive Approach

The simplest way to find all divisors is to loop from 1 to n, checking for each number if it divides n.

Collapse With Style Copy code

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

vector<int> find_divisors_naive(int n) {
    vector<int> divisors;
    for (int i = 1; i <= n; ++i) {
        if (n % i == 0) {
            divisors.push_back(i);
        }
    }
    return divisors;
}
```

The Sqrt Approach

Instead of checking all the way up to n, we can just check up to $\sqrt{n}$. For every number i that divides n, n/i will also be a divisor.

Here's how we can do it:

Collapse With Style Copy code

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

vector<int> find_divisors_optimized(int n) {
    vector<int> divisors;
    for (int i = 1; i * i <= n; ++i) {
        if (n % i == 0) {
            divisors.push_back(i);
            if (i != n / i) {
                divisors.push_back(n / i);
            }
        }
    }
    return divisors;
}
```

Any divisor d of n must satisfy $d \times \frac{n}{d} = n$. Now, d and $\frac{n}{d}$ can't both be greater than $\sqrt{n}$ because then their product would be greater than n. Therefore, at least one of them must be less than or equal to $\sqrt{n}$.

For example, for $n = 100$, using the optimized approach, we get divisors $1, 100, 2, 50, 4, 25, 5, 20, 10$ in just 10 iterations.

Primes

Primes are numbers that have exactly two divisors: 1 and itself. For example, $2, 3, 5, 7, 11, 13, 17, 19, 23, 29$ are the first 10 primes.

We can check if a number n is prime by checking if it has exactly two divisors. We can do this by using the optimized approach to find divisors. So time complexity is $O(\sqrt{n})$.

Every number n can be uniquely written as a product of primes. For example, $100 = 2 \times 2 \times 5 \times 5$. This is called the prime factorization of n.

We will learn more about prime factorizations in Phase 2.

Finding Number of Divisors From 1 to n

Using the optimized $O(\sqrt{n})$ approach, we can find the number of divisors of all numbers from 1 to n in $O(n\sqrt{n})$ time. But we can do better!

We can find the number of divisors of all numbers from 1 to n in $O(n \log n)$ time using the following code:

Collapse With Style Copy code

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

const int N = 1e6 + 9;
int d[N];

int main() {
    for (int i = 1; i < N; ++i) {
        for (int j = i; j < N; j += i) {
            d[j]++;
        }
    }
}
```

The time complexity is $O(n \log n)$ because the inner loop runs $\frac{n}{i}$ times for each i. So the total number of iterations is $\sum_{i=1}^{n} \frac{n}{i} = n \sum_{i=1}^{n} \frac{1}{i} \approx n \log n$. We learned about this approximation for Harmonic Series in Class 3.

Exercise: Find the sum of divisors of all numbers from 1 to n in $O(n \log n)$ time.

Primes under n

Using the same code as above, we can find the number of divisors of all numbers from 1 to n in $O(n \log n)$ time. The primes under n are the numbers that have exactly 2 divisors. So we can find all primes under n in $O(n \log n)$ time.

Collapse With Style Copy code

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

const int N = 1e7 + 9;
int d[N];
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  for (int i = 1; i < N; i++) {
    for (int j = i; j < N; j += i) {
```

```
      d[j]++;
    }
  }
  vector<int> primes;
  for (int i = 1; i < N; i++) {
    if (d[i] == 2) {
      primes.push_back(i);
    }
  }
  cout << primes.size() << '\n';
  return 0;
}
```

We will learn a similar technique called Sieve of Eratosthenes in Phase 2.

Difference Array

Tutorial: link

A difference array is an array d of length n such that $d[i]$ is the difference between $a[i]$ and $a[i-1]$ for all i from 1 to n. For example, if $a = [1, 3, 5, 2, 4]$, then $d = [1, 2, 2, -3, 2]$.

Finding the difference array is easy. Also, given the difference array, we can find the original array in $O(n)$ time easily and uniquely because the original array is just the prefix sum of the difference array.

Collapse With Style Copy code

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

const int N = 1e6 + 9;
int a[N], d[N];

int main() {
    int n = 5;
    a[1] = 1, a[2] = 3, a[3] = 5, a[4] = 2, a[5] = 4;
    // Finding the difference array from the original array
    for (int i = 1; i <= n; ++i) {
        d[i] = a[i] - a[i - 1];
    }
    for (int i = 1; i <= n; ++i) {
        cout << d[i] << " ";
    }
    cout << endl;

    // Finding the original array from the difference array
    for (int i = 1; i <= n; ++i) {
```

```
        a[i] = a[i - 1] + d[i];
    }
    for (int i = 1; i <= n; ++i) {
        cout << a[i] << " ";
    }
    cout << endl;
}
```

It is useful when modifying the original array is hard but modifying the difference array is easy.

Basic Problem

Statement: Given an array a of length n and q queries of the form l, r, x, for each query, add x to all elements of a from l to r inclusive. Print the final array after all queries.

Constraints: $1 \leq n, q \leq 10^6, 1 \leq l \leq r \leq n, -10^9 \leq x \leq 10^9, -10^9 \leq a[i] \leq 10^9$

Solution:

If we do the queries naively, the time complexity will be $O(nq)$ which is too slow. We can do it in $O(n + q)$ time using difference array.

Consider the difference array d of length n. For each query, the update will magically change only two elements of d: $d[l]$ and $d[r + 1]$. So we can do all queries in $O(q)$ time. And as we saw above, we can find the original array from the difference array in $O(n)$ time.

Collapse With Style

Copy code

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

const int N = 1e6 + 9;
long long a[N], d[N];

int main() {
    int n, q; cin >> n >> q;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }

    for (int i = 1; i <= n; ++i) {
        d[i] = a[i] - a[i - 1];
    }

    while (q--) {
        int l, r, x; cin >> l >> r >> x;
        d[l] += x;
        d[r + 1] -= x;
    }
```

```
    for (int i = 1; i <= n; ++i) {
        a[i] = a[i - 1] + d[i];
    }

    for (int i = 1; i <= n; ++i) {
        cout << a[i] << " ";
    }
    cout << endl;
}
```

Moderate Problem

Statement: We have n lists numbered from 1 to n. Initially, all lists are empty. We have q queries of the form l, r, x. For each query, we have to append x to all lists from l to r inclusive. After all queries, print the number of unique elements in each list.

Constraints: $1 \leq n, q \leq 10^6, 1 \leq l \leq r \leq n, 1 \leq x \leq 10^9$

Solution:

Instead of inserting x to all lists from l to r one by one, we can just insert x to list l inferring that we wanna add it to all lists from l to n. But we have to remove x from all lists $r + 1$ to n because we don't wanna add it to those lists. So we can just remember that we wanna remove it starting from $r + 1$. Check the code below for more clarity.

Collapse With Style Copy code

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

const int N = 1e6 + 9;
vector<int> add[N], remove[N];

int main() {
    int n, q; cin >> n >> q;
    while (q--) {
        int l, r, x; cin >> l >> r >> x;
        add[l].push_back(x);
        remove[r + 1].push_back(x);
    }

    map<int> mp;
    for (int i = 1; i <= n; ++i) {
        for (int x : add[i]) {
            mp[x]++;
        }
```

```
        for (int x : remove[i]) {
            mp[x]--;
            if (mp[x] == 0) {
                mp.erase(x);
            }
        }
        cout << mp.size() << endl;
    }
}
```

Moderate Problem

Problem: link

Solution:

Collapse With Style

Copy code

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

const int N = 1e5 + 9;

int ops[N], x[N], l[N], r[N];
long long a[N], d[N];

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

    int n, m, k; cin >> n >> m >> k;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }
    for (int i = 1; i <= m; ++i) {
        cin >> l[i] >> r[i] >> x[i];
    }
    for (int i = 1; i <= k; ++i) {
        int l, r; cin >> l >> r;
        ops[l]++;
        ops[r + 1]--;
    }
    for (int i = 1; i <= m; ++i) {
        ops[i] += ops[i - 1];
    }

    for (int i = 1; i <= n; i++) {
```

```
        d[i] = a[i] - a[i - 1];
    }

    for (int i = 1; i <= m; ++i) {
        d[l[i]] += 1LL * ops[i] * x[i];
        d[r[i] + 1] -= 1LL * ops[i] * x[i];
    }

    for (int i = 1; i <= n; ++i) {
        a[i] = a[i - 1] + d[i];
    }

    for (int i = 1; i <= n; ++i) {
        cout << a[i] << " ";
```

2D Prefix Sum

Tutorial: must check

Problem: You are given a $n \times m$ matrix. You have to answer q queries of the form x_1, y_1, x_2, y_2. For each query, you have to find the sum of all elements in the submatrix with top-left corner (x_1, y_1) and bottom-right corner (x_2, y_2).

Constraints: $1 \leq n, m \leq 10^3$, $1 \leq q \leq 10^6$, $1 \leq x_1 \leq x_2 \leq n$, $1 \leq y_1 \leq y_2 \leq m$, $-10^9 \leq a[i][j] \leq 10^9$

Solution:

Let $pref[i][j]$ be the sum of all elements in the submatrix with top-left corner $(1, 1)$ and bottom-right corner (i, j). That is $pref[i][j] = \sum_{x=1}^{i} \sum_{y=1}^{j} a[x][y]$.

We can compute $pref[i][j]$ for all i and j in $O(nm)$ time using the following formula: $pref[i][j] = pref[i-1][j] + pref[i][j-1] - pref[i-1][j-1] + a[i][j]$. You can find an interactive visualization here.

Now, for each query, we can find the answer in $O(1)$ time using the following formula: $ans = pref[x_2][y_2] - pref[x_1-1][y_2] - pref[x_2][y_1-1] + pref[x_1-1][y_1-1]$.

Collapse With Style Copy code

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

const int N = 1e3 + 9;
long long pref[N][N];
int a[N][N];

int main() {
    int n, m; cin >> n >> m;
    for (int i = 1; i <= n; ++i) {
```

```
        for (int j = 1; j <= m; ++j) {
            cin >> a[i][j];
        }
    }

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            pref[i][j] = pref[i - 1][j] + pref[i][j - 1] - pref[i - 1][j - 1] + a[i][j];
        }
    }

    int q; cin >> q;
    while (q--) {
        int x1, y1, x2, y2; cin >> x1 >> y1 >> x2 >> y2;
        // long long ans = 0;
        // for (int i = x1; i <= x2; i++) {
        //   for (int j = y1; j <= y2; j++) {
        //     ans += a[i][j];
        //   }
        // }
        long long ans = pref[x2][y2] - pref[x1 - 1][y2] - pref[x2][y1 - 1] + pref[x1 - 1][y
        cout << ans << endl;
    }
}
```

Exercise: Solve this

2D Static Range Update

Statement: You are given a $n \times m$ matrix. You have to perform q updates of the form x_1, y_1, x_2, y_2, x. For each query, you have to add x to all elements in the submatrix with top-left corner (x_1, y_1) and bottom-right corner (x_2, y_2). After all queries, print the final matrix.

Constraints: $1 \leq n, m \leq 10^3$, $1 \leq q \leq 10^6$, $1 \leq x_1 \leq x_2 \leq n$, $1 \leq y_1 \leq y_2 \leq m$, $-10^9 \leq a[i][j], x \leq 10^9$

Solution:

We can do similar stuff like 1D difference array list append problem.

Collapse With Style Copy code

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

const int N = 1e3 + 9;
long long d[N][N], a[N][N];
```

```
int main() {
    int n, m, q; cin >> n >> m >> q;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            cin >> a[i][j];
        }
    }
    while (q--) {
        int x1, y1, x2, y2, x; cin >> x1 >> y1 >> x2 >> y2 >> x;
        d[x1][y1] += x;
        d[x2 + 1][y1] -= x;
        d[x1][y2 + 1] -= x;
        d[x2 + 1][y2 + 1] += x;
    }

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            d[i][j] += d[i - 1][j] + d[i][j - 1] - d[i - 1][j - 1];
        }
    }

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            a[i][j] += d[i][j];
        }
    }

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            cout << a[i][j] << " ";
        }
        cout << endl;
    }
}
```

Two Pointers

Tutorial: step 1, step 2

Warm Up Problem

Problem: link

We actually solved this problem in the last class while learning about merge sort. And we used two pointers there!

Collapse With Style

Copy code

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

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

    vector<int> ans;
    int i = 0, j = 0;
    while (i < n and j < m) {
        if (a[i] < b[j]) {
            ans.push_back(a[i]);
            i++;
        } else {
            ans.push_back(b[j]);
            j++;
        }
    }
    while (i < n) {
        ans.push_back(a[i]);
        i++;
    }
    while (j < m) {
        ans.push_back(b[j]);
        j++;
    }

    for (int x : ans) {
        cout << x << " ";
    }
    cout << endl;
}
```

Main Idea

One Condition:

If $[l, r]$ is good, then $[l + 1, r]$ and $[l, r - 1]$ is also good.

Problem: Largest Subarray with sum <= s

Collapse With Style Copy code

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

const int N = 1e5 + 9;
int a[N];

int main() {
    int n; long long s; cin >> n >> s;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }

    int r = 1; long long sum = 0; int ans = 0;
    for (int l = 1; l <= n; l++) {
        while (r <= n and sum + a[r] <= s) { // move right pointer as long as the sum is <=
            sum += a[r];
            r++;
        }
        ans = max(ans, r - l); // for this l, [l, r - 1] is the largest subarray with sum <
        sum -= a[l]; // remove a[l] from the sum
    }

    cout << ans << endl;
}
```

Time complexity is $O(n)$ because the left pointer moves at most n times and the right pointer moves at most n times. This is what we call **amortized analysis**. That is, even though it looks like the right pointer moves n times for **each** l making the time complexity $O(n^2)$, it actually doesn't. It moves a total of n times for **all** l. So the time complexity is $O(n)$.

Also, we could solve the problem using binary search. But that would make the time complexity $O(n \log n)$. So two pointers technique is better if we can use it. But binary search is more general.

Similar Problem: Number of Subarrays with sum <= s

Collapse With Style Copy code

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

const int N = 1e5 + 9;
int a[N];
```

```
int main() {
    int n; long long s; cin >> n >> s;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }

    int r = 1; long long sum = 0; long long ans = 0;
    for (int l = 1; l <= n; l++) {
        while (r <= n and sum + a[r] <= s) { // move right pointer as long as the sum is <=
            sum += a[r];
            r++;
        }
        ans += r - l; // for this l, [l, r - 1] is the largest subarray with sum <= s
        sum -= a[l]; // remove a[l] from the sum
    }

    cout << ans << endl;
}
```

Similar Problem: Number of Subarrays with sum >= s

Collapse With Style

Copy code

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

const int N = 1e5 + 9;
int a[N];

int main() {
    int n; long long s; cin >> n >> s;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }

    int r = 1; long long sum = 0; long long ans = 0;
    for (int l = 1; l <= n; l++) {
        while (r <= n and sum + a[r] < s) { // move right pointer until the sum is >= s
            sum += a[r];
            r++;
        }
        ans += n - r + 1; // for this l, [l, r - 1] is the largest subarray with sum < s, s
        sum -= a[l]; // remove a[l] from the sum
    }
```

```
    cout << ans << endl;
```

Important Problem: Segments with Small Spread

Collapse With Style — Copy code

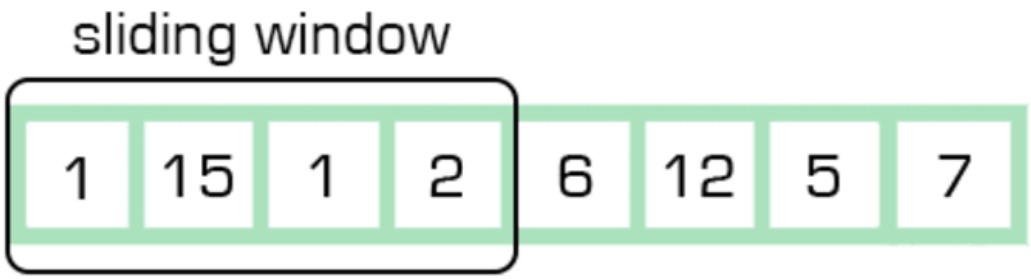
```
#include<bits/stdc++.h>
using namespace std;

using ll = long long;
const int N = 1e5 + 9;
ll a[N];
struct DS {
  multiset<ll> se;
  DS() {}
  void insert(ll x) {
    se.insert(x);
  }
  void erase(ll x) {
    se.erase(se.find(x));
  }
  ll get_max() {
    return *(--se.end());
  }
  ll get_min() {
    return *se.begin();
  }
  ll get() {
    return get_max() - get_min();
  }
  void print() {
    for (auto x: se) {
      cout << x << ' ';
    }
    cout << '\n';
  }
}ds;
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n; ll k; cin >> n >> k;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  ll ans = 0;
  int r = 1;
```

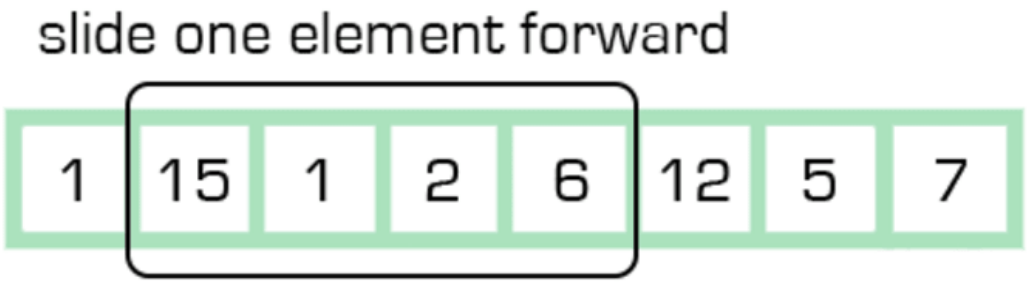
```
  for (int l = 1; l <= n; l++) {
    while (r <= n) {
      ds.insert(a[r]);
      if (ds.get() > k) {
        ds.erase(a[r]);
        break;
      }
      r++;
    }
    ds.print();
    // r - 1 is the maximum index i such that the max(a[l...i]) - min(a[l...i]) <= k
    ans += r - l;
    ds.erase(a[l]);
  }
  cout << ans << '\n';
  return 0;
}
```

Sliding Window Technique

Tutorial: link

A sliding window is a subarray of length k that slides from left to right. For example, if $a = [1, 2, 3, 4, 5, 6]$ and $k = 3$, then the sliding windows are $[1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]$.





Problem: Maximum Sum Subarray of Length K

You are given an array a of length n and an integer k. You have to find the maximum sum of any subarray of length k.

Constraints: $1 \leq n \leq 10^6, 1 \leq k \leq n, -10^9 \leq a[i] \leq 10^9$

Solution:

We can use prefix sum to solve this problem. But that will take $O(n)$ extra space. We can do better using sliding window technique.

Collapse With Style Copy code

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

const int N = 1e6 + 9;
int a[N];

int main() {
    int n, k; cin >> n >> k;
    for (int i = 1; i <= n; ++i) {
        cin >> a[i];
    }

    int sum = 0, ans = 0;
    for (int i = 1; i <= n; ++i) {
        sum += a[i];
        if (i >= k) {
            ans = max(ans, sum);
            sum -= a[i - k + 1];
        }
    }

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

Time complexity: $O(n)$

Extra space: $O(1)$

Problem: Maximum of All Subarrays of Length K

You are given an array a of length n and an integer k. You have to find the maximum of all subarrays of length k.

Constraints: $1 \leq n \leq 10^6, 1 \leq k \leq n, -10^9 \leq a[i] \leq 10^9$

Solution:

Just like the previous problem, we can slide a window of length k from left to right and find the maximum of each window.

Collapse With Style Copy code

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

const int N = 1e5 + 9;
int a[N];
struct DS {
  multiset<int> se;
  DS() {}
  void insert(int x) {
    se.insert(x);
  }
  void erase(int x) {
    se.erase(se.find(x));
  }
  int get_max() {
    return *(--se.end());
  }
}ds;
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int n, k;
  cin >> n >> k;
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (int i = 1; i <= n; i++) {
    ds.insert(a[i]);
    if (i >= k) {
      cout << ds.get_max() << ' ';
      ds.erase(a[i - k + 1]);
    }
  }
  return 0;
}
```

Exercise: Number of Unique Elements in All Subarrays of Length K

You are given an array a of length n and an integer k. You have to find the number of unique elements in all subarrays of length k.

Multiplying Two Different Data Types

When two different types of variables are multiplied in C++, the type of the result depends on the rules of C++ type promotion and conversion. The general rule is that the "smaller" type gets promoted to the "larger" type

before the operation, and the result is of the "larger" type.

For example, if you multiply an integer and a double, the integer will be promoted to a double before the multiplication, and the result will be of type double.

Let's consider some specific examples:

- `int * double`: The result is double. The int is promoted to a double before multiplication.
- `float * double`: The result is double. The float is promoted to a double before multiplication.
- `short * int`: The result is int. The short is promoted to an int before multiplication.
- `long long * int`: The result is long long. The int is promoted to a long long before multiplication.
- `int + bool`: The result is int. The bool is promoted to an int before addition.

These are the standard C++ conversions. However, note that if the range of the "larger" type is insufficient to hold the result, overflow may occur, which leads to undefined behavior.

Common Mistakes in Competitive Programming and How to Avoid Them

---

Tutorial: my blog, must check

More Mistakes:

- Correct equations:
  - If $a \cdot b \leq n$, then $a \leq \lfloor \frac{n}{b} \rfloor$. This is correct.
  - If $a \cdot b < n$, then $a < \lfloor \frac{n}{b} \rfloor$. This is wrong. For example, $2 \cdot 3 < 7$, but $2 < \lfloor \frac{7}{3} \rfloor$ is false.
  - If $a \cdot b \geq n$, then $a \geq \lceil \frac{n}{b} \rceil$. This is correct.
  - If $a \cdot b > n$, then $a > \lceil \frac{n}{b} \rceil$. This is wrong. For example, $2 \cdot 3 > 5$, but $2 > \lceil \frac{5}{3} \rceil$ is false.
- `s.substr(i, j)` returns a string of length j starting from index i. It doesn't return a string of length $j - i$ starting from index i and ending before index j. For example, if `s = "abcdef"`, then `s.substr(1, 3)` returns `"bcd"`, not `"bc"`.

Constant Optimization

Constant Factors are the hidden constants in the time or space complexity of an algorithm. For example, if the number of operations in an algorithm is $5n$, then 5 is the constant factor but its big-O is $O(n)$. And this is slower than an algorithm with $2n$ operations even though both have the same big-O complexity of $O(n)$.

Constant Optimization refers to the practice of reducing the hidden constants in the time or space complexity of an algorithm. While big-O notation gives us an idea of how an algorithm scales, the hidden constants can actually affect the algorithm's actual runtime. Thus, optimizing these constants is vital for performance, particularly in competitive programming where every millisecond counts.

So basically in problems with tight time constraints, an algorithm with a good big-O complexity may still time out if the hidden constants are large. It happens rarely but it does happen. So it's good to know some tricks to reduce the hidden constants.

- **Use `'\n'` instead of `endl`**

  `endl` flushes the output buffer every time it is called and it is slower. So it's better to use `'\n'` instead of `endl`.

- **Efficient I/O**

  Using `cin` and `cout` is slow. So it's better to use `ios_base::sync_with_stdio(0)` and `cin.tie(0)` to make `cin` and `cout` faster.

- **Efficient Data Structures**

  Choose data structures that have lower constant factors. For instance, using arrays or vectors instead of deque is faster because the constant factor of arrays and vectors is lower than that of deque.

- **Bit Manipulation**

  Bit manipulation is faster than arithmetic operations. For example, `x << 1` is faster than `x * 2`. Also, `x & 1` is faster than `x % 2`. But normally it doesn't make much difference.

- **Modulo**

  Modulo is slower than other arithmetic operations. For example, `(a + b) % m;` is slower than `a += b; if (a >= m) a -= m;`. But normally it doesn't make much difference.

- **Avoid Recursion**

  Recursion is slower than iteration. So it's better to avoid recursion if possible.

- **Avoid Floating Point**

  Floating point / Double operations are slower than integer operations. So it's better to avoid floating point if possible.

- **Using `const`**

  Using `const` makes the program faster. For example, if you have to take modulo of a number `mod` many times, it's better to declare `const int mod = 1e9 + 7;` instead of `int mod = 1e9 + 7;`.

- **Big Data Types**

  Using big data types like `long long` is slower than using small data types like `int`. So it's better to use small data types if possible.

- **Big Arrays**

  Accessing a random index of a big array is slower than accessing a random index of a small array. So it's better to use small arrays if possible. So if $N < 10^6$, then it's better to use an array of size 10^6 than an array of size 10^7 unnecessarily.

Note that these optimizations are not always necessary or even useful or may not be necessary at all for beginners. But it's good to know them.

Try to run the following code and see the difference:

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 = 6e8; // for small n it doesn't make much difference
  int mod = 1e9 + 7;
  // const int mod = 1e9 + 7; // try this
  int ans = 0;
  auto st = clock();
  for (int i = 1; i <= n; i++) {
    ans += i;
    ans %= mod;
    // if (ans >= mod) ans -= mod; // try this
  }
  cout << ans << '\n';
  cout << fixed << setprecision(10) << "Time: " << 1.0 * (clock() - st) / CLOCKS_PER_SEC <<
  return 0;
}
```

Implementation Tricks

Tutorial: link

To make your life easier, you should know some implementation tricks. These are not necessary but they are useful.

- Use `#define` and `auto` to make your code shorter and easier to read. We have already learned about them. For example:

  Collapse With Style — Copy code

  ```
  map<string, int> mp;
  mp["hello"] = 1;
  mp["world"] = 2;
  for (auto [s, x] : mp) {
      cout << s << " " << x << endl;
  }
  ```

- Using `1LL`: For typecasting we normally use `(long long)`. But we can also multiply by `1LL` to get the same result. For example, `1LL * a * b` is equivalent to `(long long) a * b`. Similar thing can be done for other data types. For example, `1.0 * a * b` is equivalent to `(double) a * b`.
- Modularize your code. Use functions to make your code easier to read.
- Using `array<type, size>` can sometimes be useful. For example in our scenario 1.5 in Class 5, we have seen the following problem:

Scenario 1.5: You are given $n \leq 10^5$ 5D points of the form (x, y, z, u, v). Sort the points first based on their z coordinate. In case of ties then based on their y coordinate, and still in case of ties then based on their u coordinate. And still in case of ties then based on their x coordinate. And finally still in case of ties then based on their v coordinate. How will you do it?

Solution:

Collapse With Style Copy code

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

// O(nlogn)
int32_t main() {
  int n;
  cin >> n;
  vector<tuple<int, int, int, int, int>> v;
  for (int i = 0; i < n; i++) {
    int x, y, z, u, v;
    cin >> x >> y >> z >> u >> v;
    v.push_back({z, y, u, x, v});
  }
  sort(v.begin(), v.end());
  for (auto x: v) {
    cout << get<3>(x) << ' ' << get<1>(x) << ' ' << get<2>(x) << ' ' << get<0>(x) << '
  }
  return 0;
}
```

So we used `tuple` here. But we can also use `array` here. It's easier to use.

Collapse With Style Copy code

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

// O(nlogn)
int32_t main() {
  int n;
  cin >> n;
  vector<array<int, 5>> v;
  for (int i = 0; i < n; i++) {
    int x, y, z, u, v;
    cin >> x >> y >> z >> u >> v;
    v.push_back({z, y, u, x, v});
  }
```

```
  sort(v.begin(), v.end());
  for (auto x: v) {
    cout << x[3] << ' ' << x[1] << ' ' << x[2] << ' ' << x[0] << ' ' << x[4] << '\n';
  }
  return 0;
}
```

- Using lambda functions is sometimes useful, as it can be used as anonymous inlining that acts as comparators for various STL functions like sort(). Fand also function-like behavior and capture of local variables to perform repetitive tasks, without having to explicitly pass as many arguments.
- Using `memset` and `fill` can be used to initialize arrays with a specific value.
- A simple way of eliminating all duplicates from a vector is to sort it and then use `unique` function. For example:

```
Collapse With Style                                                    Copy code
vector<int> v = {1, 2, 3, 4, 1, 2, 3, 4};
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

- Sorting in descending order:

```
Collapse With Style                                                    Copy code
  // normal way
  sort(v.begin(), v.end());
  reverse(v.begin(), v.end());

  // easier way
  sort(v.begin(), v.end(), greater<int>());

  // easiest way
  sort(v.rbegin(), v.rend());
```

- Min or Max of multiple variables:

```
Collapse With Style                                                    Copy code
  int a = 1, b = 2, c = 3, d = 4;
  // normal way
  int mn = min(a, min(b, min(c, d)));
  int mx = max(a, max(b, max(c, d)));

  // easiest way
```

```
int mn = min({a, b, c, d});
int mx = max({a, b, c, d});
```

- Min or Max of a vector:

Collapse With Style Copy code

```
vector<int> v = {1, 2, 3, 4};

int mn = *min_element(v.begin(), v.end());
int mx = *max_element(v.begin(), v.end());
```

- Builtin Functions:

Collapse With Style Copy code

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

int main() {
    int a = 10, b = 15;
    // gcd of two numbers
    cout << __gcd(a, b) << '\n'; // 5

    int x = 20; // x = 00000000 00000000 00000000 00010100
    long long y = 30; // y = 00000000 00000000 00000000 00000000 00000000 00000000 00
    // number of set bits
    cout << __builtin_popcount(x) << '\n'; // 2
    cout << __builtin_popcountll(y) << '\n'; // 4

    // number of leading zeros
    cout << __builtin_clz(x) << '\n'; // 27 (clz -> count leading zeros)
    cout << __builtin_clzll(y) << '\n'; // 59

    // number of trailing zeros
    cout << __builtin_ctz(x) << '\n'; // 2 (ctz -> count trailing zeros)
    cout << __builtin_ctzll(y) << '\n'; // 1

    // index of the highest set bit (or most significant bit, MSB)
    cout << 31 - __builtin_clz(x) << '\n'; // 4
    cout << 63 - __builtin_clzll(y) << '\n'; // 4
    cout << __lg(x) << '\n'; // 4
    cout << __lg(y) << '\n'; // 4

    // index of the lowest set bit (or least significant bit, LSB)
    cout << __builtin_ffs(x) - 1 << '\n'; // 2 (ffs -> find first set, it returns 1-b
```

```
    cout << __builtin_ffsll(y) - 1 << '\n'; // 1
    cout << __builtin_ctz(x) << '\n'; // 2 (equal to index of the lowest set bit)

    return 0;
}
```

- Easier debugging: `#define trace(x) cerr << #x << ": " << x << " " << endl` and then use `trace(x)` to print the value of `x` along with its name.
- Use sublime snippets or shortcuts to make your life easier.

Necessity and Sufficiency

A necessary condition is a condition that must be present for an event to occur. For example, if a number is even, then it is necessary that the number is divisible by 2. So if A is necessary for B, then B cannot happen without A.

A sufficient condition is a condition that, if present, guarantees that the event will occur. For example, if a number is divisible by 2, then it is a sufficient condition that the number is even. So if A is sufficient for B, then the presence of A ensures that B will occur.

Examples:

- A number being divisible by 4 is sufficient (but not necessary) for it to be even.
- A number being divisible by 2 is both sufficient and necessary for it to be even.
- A number (greater than 2) being odd is necessary for it to be prime. But it is not sufficient. For example, 9 is greater than 2 and odd but it is not prime.
- **Problem:** Given an array of n non-negative integers. In one operation you can select two distinct indices i and j and decrease the value of both $a[i]$ and $a[j]$ by 1. You have to make all the elements of the array equal to 0. Find if it is possible to do so.
    - **Necessary Conditions:**
        - The sum of all the elements of the array must be even. Because in each operation, the sum of the array decreases by 2. So if the sum of the array is odd, then it is not possible to make all the elements of the array equal to 0. By the way, this is also called an **invariant**. That is, the sum of the array modulo 2 is an invariant. An invariant is a quantity or a property that doesn't change throughout the operations.
        - The maximum element cannot be greater than the sum of all the other elements. Because in each operation, the maximum element decreases by at most 1. So if the maximum element is greater than the sum of all the other elements, then it is not possible to make it 0.
    - **Sufficient Condition:** The above two conditions are sufficient for the answer to be "Yes". Because in each operation, you can select the maximum two elements and decrease them by 1. This strategy always works.
- **Problem:** link. Check the editorial for the solution to understand the concept better.
- Lots of problems in competitive programming can be solved using this. First try to find some necessary conditions and then try to prove that they are sufficient.

Self Deception

Tutorial: must check

Key Takeaways:

1. **Self-Deception**: Many people think they are practicing effectively when, in reality, they are going through the motions. This self-deception is hard to detect but can significantly slow down your progress. "Your practice is not as good as you think it is" or even "You aren't as good as you think you are".
2. **Importance of Motivation**: Genuine interest in problem-solving is vital for effective practice.
3. **Misuse of Editorials**: Many rely too much on problem editorials, using them always as an aid instead of a learning tool. Solve problems with the intent of never reading the editorial, this will force you to think deeply.
4. **Biased Selection of Problems**: People often pick problems that they find easier, thereby avoiding areas where they most need improvement. Always try to solve problems that you find difficult but not too difficult.
5. **Goodhart's Law in Practice**: Measuring your progress by the number of problems solved can be misleading. The goal should be effective learning, not merely racking up numbers.

The Forcing Fallacy

Tutorial: must check

Key Takeaways:

1. Many beginners have developed this "technique" for solving problems. They look at the constraints or use some other, usually misguided, intuition to tell whether it is Binary Search, greedy, or something else. They then try to "force" the chosen technique on it.
2. What you should do: Deeply understand the problem, familiarize yourself with the situation, make observations, reductions and rephrase the problem. Actual computation is often the easiest part.
3. In most problems, you are presented with some structure. In a harder problem, you can't directly calculate the thing you are asked to do. Instead, what you should do is try to understand this structure. I mean really understand it. Not just understand the words in the statement, understand what's going on inside. Gather observations. Then you will be able to solve the problem.

Self Learning

It is really important to learn new things by yourself. You can't learn everything in the class or from a course. So you have to learn new things by yourself.

Self-learning is a critical competency for personal and professional development, and it's especially vital in fields like software engineering and competitive programming.

Steps for Effective Self-Learning

Step 1: Identify Your Learning Goals

- **Specific**: Clearly identify what you want to learn.
- **Measurable**: Make sure that your progress can be measured.
- **Achievable**: Your goal should be realistic.

Example: If you want to learn Segment Tree, then your goal could be to learn the theory and solve x amount of problems on Segment Tree within 2 weeks.

Step 2: Gather Resources

- **Books**: Useful for deep knowledge.
- **Online Courses**: Ideal for structured learning.
- **Online Blogs**: Great for specific topis.
- **Youtube Videos**: Great for visual learning.
- **Mentors & Peers**: You friends and seniors can help you a lot.

Step 3: Plan Your Learning Schedule

- **Timelines**: Create a timeline for your learning.
- **Chunking**: Divide the topic into smaller, manageable pieces.
- **Prioritize**: Focus on the most impactful subjects first.

Example: If you want to learn Segment Tree, then you can divide it into 3 parts: theory, implementation, and problems. Then you can spend 2 days on theory, 2 days on implementation, and 10 days on solving problems. (This is just an example. You can divide it in any way you want.)

Step 4: Practice and Feedback

- **Hands-on Practice**: Solving problems is the best way to learn than just reading theory.
- **Feedback Loop**: Continuously evaluate your progress and adjust your strategy accordingly.

Step 5: Review and Revise

- **Repetition**: Feel free to review past materials to solidify your understanding.
- **Teaching**: Explaining the concept to someone else is a powerful revision tool.
- **Refinement**: As you learn, you'll discover more effective methods and resources. Keep refining your approach.

Tools for Self-Learning

1. **Note-taking apps**: Use apps like Notion to take notes and plan your learning.
2. **Calendar apps**: Use apps like Google Calendar to plan your learning schedule.

Tips for Staying Motivated

- **Sense of Purpose**: Remind yourself why you're learning this topic. Get a clear sense of purpose.
- **Community**: Join online communities or social media groups that focus on the topic you're learning.

- **Incentives**: Reward yourself for achieving small milestones. For example, you can reward yourself with a movie or a pizza after solving 10 problems on Segment Tree!
- **Enjoyment**: Make sure that you're enjoying the learning process. If you're not enjoying it, then you're doing something wrong. So try to find out what you're doing wrong and fix it.