Summer of Science
2025

CS02
Advanced Algorithms for CP

Endterm Report

Prepared by: Saksham Khandelwal - 24B0965
Instructor: Nirav Bhattad
Date: July 21, 2025

Contents

Preface

Throughout this project, my focus was on deepening my understanding of algorithms commonly used in competitive programming (CP). My approach was more theory-oriented, with an emphasis on understanding the concepts and some implementations.

Most of the problems I practiced were basic in nature, such as the Leetcode Top 150 and some other sources or the Codeforces contest. However, I did not maintain a detailed record of them. As such, this report does not include a comprehensive list of the problems solved during the project.

This report is structured as follows. First, a full list of topics covered is presented, followed by dedicated sections for each advanced algorithm, where I explain the logic and applications.

The Explained Algorithms section provides concise yet comprehensive explanations of selected algorithms, along with example problems to illustrate their application.

1 Planned vs. Actual Coverage

Original Plan

This was my week-by-week plan made at the beginning of the project:

- **Week 1:** Basics – STL containers, greedy algorithms, sorting, binary search, recursion, backtracking, and simple dynamic programming.
- **Week 2:** Graph algorithms – BFS, DFS, Topological Sort, Dijkstra, Bellman-Ford, Floyd-Warshall, Kruskal, Union-Find, Kosaraju, LCA.
- **Week 3:** Catch-up week – reinforce topics and complete pending ones.
- **Week 4:** Light algorithms – Two pointers, Sliding window, Mo's Algorithm, and basic string algorithms.
- **Week 5:** Fenwick Tree and Segment Tree.
- **Week 6:** Fast Fourier Transform (FFT) and Number Theoretic Transform (NTT).
- **Week 7:** Suffix Arrays, LCP Arrays, Ukkonen's Algorithm.
- **Week 8:** One advanced algorithm (planned: Berlekamp–Massey).

What Was Actually Covered

Most of the plan was followed well, especially in the first few weeks. Here is a summary of what was completed:

- **Basics (Week 1):** Covered STL containers and algorithms, sorting (selection, insertion, merge, quick, counting), binary search, recursion, backtracking, greedy techniques, and introductory dynamic programming.
- **Graphs (Week 2):** Studied BFS, DFS, topological sort, Dijkstra, Bellman-Ford, Floyd-Warshall.
- **Catch-up (Week 3):** Prim's algorithm, Kosaraju's algorithm, Union-Find (DSU), Kruskal's algorithm,
- **Light Algorithms (Week 4):** Lowest Common Ancestor in Binary Tree, Two Pointers, Sliding Window
- **Trees (Week 5):** Implemented and understood Fenwick Trees and Segment Trees thoroughly.
- **Transform Algorithms (Week 6):** Learned Fast Fourier Transform (FFT) in detail. Studied Number Theoretic Transform (NTT) only in theory due to complexity.
- **Skipped (Week 7):** I planned to study suffix arrays, LCP arrays, and Ukkonen's algorithm, but ended up skipping this week due to other academic load. Instead I proceeded with the string matching algorithms (KMP and Rabin-Karp).
- **Advanced Algorithm (Week 8):** Studied Berlekamp–Massey thoroughly along with matrix exponentiation.

2 Topics Covered

Basic Algorithms and Techniques

- STL Containers and Sorting Algorithms
- Binary Search
- Recursion and Backtracking
- Greedy Algorithms
- Basic Dynamic Programming
- Two Pointers and Sliding Window

Graph Algorithms

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Topological Sort (Kahn's algorithm)
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall Algorithm
- Prim's Algorithm
- Kruskal's Algorithm
- Union-Find (Disjoint Set Union - DSU)
- Kosaraju's Algorithm (SCC)
- Lowest Common Ancestor (Binary Lifting)

String Algorithms

- Knuth-Morris-Pratt (KMP)
- Rabin-Karp

Advanced Data Structures

- Fenwick Tree (Binary Indexed Tree)
- Segment Tree

Advanced Techniques

- Fast Fourier Transform (FFT)
- Number Theoretic Transform (NTT) — Theory Only
- Matrix Exponentiation
- Berlekamp–Massey Algorithm

3 Explained Algorithms

3.1 Dijkstra's Algorithm

Dijkstra's Algorithm is a fundamental technique used to find the shortest path from a source node to all other nodes in a graph with nonnegative edge weights. It works greedily by always expanding the node with the smallest known tentative distance and updating the distances to its neighbors.

Core Idea

The algorithm maintains a distance array where `dist[i]` holds the current known shortest distance from the source to node i. Initially, all distances are set to infinity, except the source which is zero.

A Min priority queue is used to efficiently fetch the node with the smallest distance at each step. For each node taken from the queue, its adjacent nodes are checked: if the path through the current node gives a shorter distance to a neighbor, the neighbor's distance is updated and reinserted into the priority queue.

Why It Works

The key invariant of Dijkstra's algorithm is that once a node is processed, its shortest path is guaranteed. This works only because all edge weights are nonnegative, so once you find a path, there cannot be a cheaper path through unprocessed nodes.

If the edge weights were negative, a shorter path could still be discovered later, which is why the Dijkstra's algorithm does not work in that case.

Complexity

- Time: $O((N + E) \log N)$ using a binary heap
- Space: $O(N)$ for the distance array

Problems

- **CSES - Shortest Routes I**
  Given a directed weighted graph, compute the shortest distance from node 1 to all other nodes. This problem is the direct implementation of the algorithm.
- **Codeforces 2000G - Call During the Journey**
  This is a clever reverse Dijkstra problem. Instead of finding earliest arrival times, we compute the latest departure times from each node such that arrival at node n is still at or before t_0. Bus availability is time constrained, so each edge is considered based on whether it fits the allowed time window otherwise walking and waiting are always an option.

3.2 DSU + Kruskal's Algorithm

Kruskal's algorithm is a greedy method to find the Minimum Spanning Tree (MST) of a connected and weighted graph. It operates by sorting the edges by weight and adding them one by one to the MST as long as they don't form a cycle. To detect cycles efficiently, we use the Disjoint Set Union data structure, which keeps track of connected components.

Core Idea

The algorithm works in the following steps:

- Sort all edges of the graph in non decreasing order of their weights.
- Initialize DSU so that each node is its own parent.
- Iterate over the sorted edges. For each edge (u, v):
  - If u and v belong to different components add the edge to the MST and merge the components using union operation.
  - If u and v are already in the same component, skip the edge.

Why It Works

Kruskal's algorithm is based on the greedy principle. At each step, it chooses the smallest edge that connects two different components, ensuring that no cycles are formed.

The DSU structure supports two main operations:

- `find(u)`: Returns the root of the component containing node u. Path compression is used for efficiency.
- `union(u, v)`: Merges the components containing u and v. Union by rank or size helps keep the tree flat.

Because Kruskal's algorithm always picks the next smallest edge that doesn't form a cycle, and DSU guarantees efficient cycle detection, the algorithm is optimal for MST construction.

Complexity

- Time: $\mathcal{O}(E \log E)$
- Space: $\mathcal{O}(N)$ for the DSU data structures.

Problems

- **CSES - Road Reparation**
  Given an undirected weighted graph, find the minimum cost to connect all nodes. This is a direct implementation of Kruskal's algorithm using DSU to detect and avoid cycles while building the MST.

- **Codeforces - Minimum spanning tree for each edge**
  You are given a connected graph and asked, for each edge, what would be the weight of the MST if that edge were forced to be included.
  This requires building an MST once with Kruskal, and then for every edge not in the MST, checking the maximum edge in the path between its two endpoints. Removing that and adding the fixed edge gives the answer.

3.3 String Matching Algorithms: KMP and Rabin-Karp

String matching algorithms are designed to find occurrences of a pattern string P within a text string T. Two classic approaches are:

- **KMP (Knuth-Morris-Pratt)**: A Linear time algorithm based on prefix function.

- **Rabin-Karp**: Algorithm using rolling hash to detect matches.

Core Idea

KMP Algorithm:

- Compute a `prefix function` $\pi[i]$ for the pattern P, where $\pi[i]$ stores the length of the longest proper prefix which is also a suffix for $P[0 \dots i]$.

- While matching, if a mismatch occurs at position i, we shift the pattern by $\pi[i-1]$ instead of starting from scratch, thus avoiding redundant comparisons.

- This makes the algorithm linear in total time.

Rabin-Karp Algorithm:

- Hash the pattern P to a number using polynomial rolling hash.

- Slide a window of length $|P|$ over T and compute hashes of all substrings of length $|P|$ using a rolling hash.

- Compare hashes. If they match, verify character-by-character to avoid false positives due to hash collisions.

Why It Works

KMP: Works due to efficient reuse of the work already done using the prefix-suffix table. It ensures no character in T is compared more than once. The prefix function stores information about how the pattern overlaps with itself.

Rabin-Karp: Works by converting strings to numeric hash values such that equality of hashes (generally) implies equality of substrings. The rolling hash lets us compute substring hashes in constant time.

Complexity

- **KMP**
  - Time: $\mathcal{O}(N + M)$
  - Space: $\mathcal{O}(M)$
- **Rabin-Karp**
  - Time: Average $\mathcal{O}(N + M)$, Worst-case $\mathcal{O}(N \cdot M)$ (when all collisions)
  - Space: $\mathcal{O}(1)$

Code

```cpp
vector<int> computePrefixFunction(const string &pattern) {
    int m = pattern.size();
    vector<int> lps(m, 0); // Longest Prefix Suffix
    int len = 0;
    for (int i = 1; i < m; ++i) {
        while (len > 0 && pattern[i] != pattern[len])
            len = lps[len - 1];
        if (pattern[i] == pattern[len]) len++;
        lps[i] = len;
    }
    return lps;
}

vector<int> KMP(const string &text, const string &pattern) {
    vector<int> result;
    int n = text.size(), m = pattern.size();
    vector<int> lps = computePrefixFunction(pattern);
    int i = 0, j = 0;
    while (i < n) {
        if (text[i] == pattern[j]) {i++; j++;}
        if (j == m) {
            result.push_back(i - j);
            j = lps[j - 1];
        } else if (i < n && text[i] != pattern[j]) {
            if (j != 0) j = lps[j - 1];
            else i++;
        }
    }
    return result;
}
```

Listing 1: Knuth-Morris-Pratt

```
const int d = 256;        // ASCII Characters
const int q = 1009;        // A prime number for mod

vector<int> RabinKarp(const string &text, const string &pattern) {
    vector<int> result;
    int n = text.size(), m = pattern.size();
    if (m > n) return result;

    int h = 1;
    for (int i = 0; i < m - 1; i++)
        h = (h * d) % q; // For rolling hash change

    int p = 0, t = 0;
    for (int i = 0; i < m; i++) {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }

    for (int i = 0; i <= n - m; i++) {
        if (p == t) {
            if (text.substr(i, m) == pattern)
                result.push_back(i);
        }
        if (i < n - m) {
            t = (d * (t - text[i] * h) + text[i + m]) % q;
            if (t < 0) t += q;
        }
    }
    return result;
}
```

Listing 2: Rabin-Karp

Problems

- **CSES - String Matching**
  Count the number of occurrences of pattern P in text T.
  This is a classic direct use of the KMP algorithm for full pattern matching.

- **Codeforces - Compress Words**
  Given a list of words, concatenate them into the shortest string such that each word is a suffix of the current string or starts after the current suffix ends.
  This uses rolling hash (Rabin-Karp) to compute maximum overlap between consecutive words.

3.4 Fenwick Tree

A Fenwick Tree is a data structure that supports efficient range sum queries and point updates on an array. It is widely used in range query problems where updates and queries need to be processed in logarithmic time.

Fenwick Trees are typically used for:

- Point update and range sum queries.
- Inversion counting.
- Frequency table manipulation.

Core Idea

The Fenwick Tree maintains an implicit tree structure using binary representation of indices:

- Each index i in the tree array stores the sum of elements in a specific range of the original array, determined by the lowest set bit of i.
- To query the range sum from 1 to i, we traverse backward: $i, i - (i\& - i), \ldots$
- To update a single element at index i, we traverse forward: $i, i + (i\& - i), \ldots$

This structure allows both operations to run in $\mathcal{O}(\log N)$ time.

Why It Works

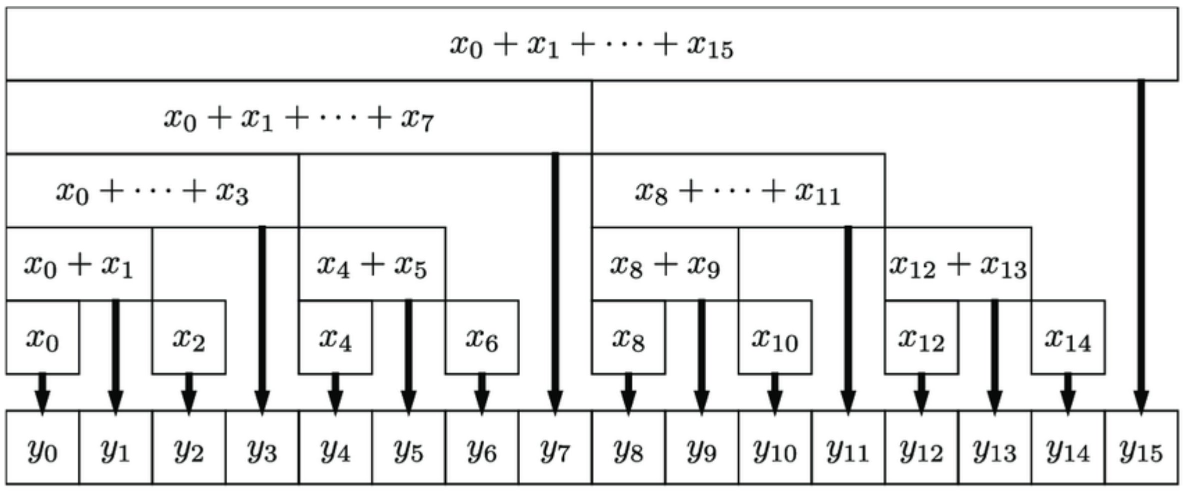
The Fenwick Tree relies on the binary representation of indices:

- Each index i covers a segment of the array of length 2^k (where k is the position of the lowest set bit).
- When querying the sum up to i, we combine segments whose union gives the full range $[1, i]$.
- For updating, we propagate the change to all segments that include index i by moving to parent indices.

It is compact, easy to implement, and faster than Segment Trees when only point updates and range queries are needed.

Complexity

- Time: $\mathcal{O}(\log N)$ per operation (both update and query)
- Space: $\mathcal{O}(N)$


$x_0 + x_1 + \cdots + x_{15}$
$x_0 + x_1 + \cdots + x_7$
$x_0 + \cdots + x_3$
$x_8 + \cdots + x_{11}$
$x_0 + x_1$
$x_4 + x_5$
$x_8 + x_9$
$x_{12} + x_{13}$
x_0
x_2
x_4
x_6
x_8
x_{10}
x_{12}
x_{14}
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12} y_{13} y_{14} y_{15}

Fenwick tree array for size n = 15

Problems

- **CSES - Static Range Sum Queries**
  Given an array of n numbers and q queries, each asking for the sum of elements from l to r.
  Use a Fenwick Tree to build prefix sums and answer each query in $\mathcal{O}(\log N)$.

- **Codeforces - Berland Fair**
  You are given a list of item prices and total money. You can buy items in rounds. Find how many total items can be bought.
  This problem is a good application of Fenwick Trees to maintain range sums and support fast range sum queries and removal of items as their prices are spent.

3.5 Segment Tree

A Segment Tree is a tree based data structure that allows efficient querying and updating of segments. It supports a wide variety of range based queries in logarithmic time.

Segment Trees are used in:

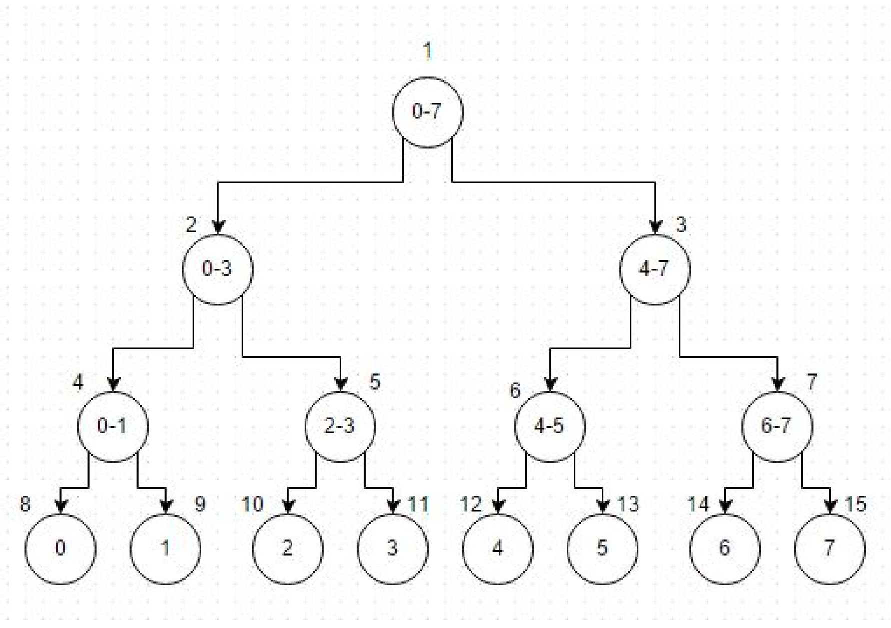
- Range queries (sum/min/max/GCD/etc.)
- Point and range updates
- Lazy propagation for range modification

Core Idea

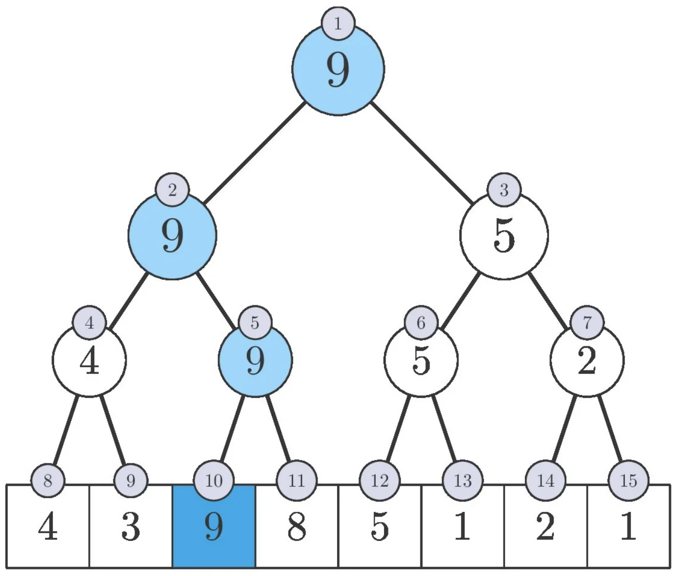
- Segment Tree is a binary tree where each node represents a segment of the array.
- The root represents the range $[0, n-1]$, and each node splits its range into two halves.

- For a query over range $[l, r]$, we traverse the tree and combine information from overlapping segments.
- For updates, we update a leaf node and propagate the change back up to the root.

For n elements, we build a tree with $\sim 4n$ nodes (due to odd numbers else $2n$ for 2^k range), allowing updates and queries in $\mathcal{O}(\log n)$ time.



Segment Tree Structure for array of size 8



Segment Tree example for Range Max

Why It Works

Segment Trees are structured to divide and conquer: each node summarizes information about a subsegment of the array.

The recursive splitting of the array allows:

- Efficient merging of results for queries from left and right subtrees.
- Efficient propagation of updates to affected segments.

Segment Trees maintain logarithmic height, which ensures all operations are logarithmic in the worst case. Lazy propagation extends this to support range updates efficiently.

Complexity

- Time:
  - Build: $\mathcal{O}(n)$
  - Query: $\mathcal{O}(\log n)$
- Space: $\mathcal{O}(4n)$

Problems

- **CSES - Range Sum Queries II**
  Given an array, support point updates and sum queries over any range $[l, r]$.
- **Codeforces - The Child and Sequence**
  Support 3 operations on an array: range sum query, point update, and range modulo update. We can use segment trees by maintaining max values and skipping subtrees where all values are less than the modulus for range modulo updates.

Code

```
class SegmentTree {
private:
    vector<int> tree, lazy, A;
    int n;

    void build(int node, int start, int end) {
        if (start == end) {
            tree[node] = A[start];
        } else {
            int mid = (start + end) / 2;
            build(2 * node, start, mid);
            build(2 * node + 1, mid + 1, end);
            tree[node] = tree[2 * node] + tree[2 * node + 1];
        }
    }
```

```
void updateRange(int node, int start, int end, int l, int r, int
    val) {
     if (lazy[node] != 0) {
         tree[node] += (end - start + 1) * lazy[node];
         if (start != end) {
             lazy[node * 2] += lazy[node];
             lazy[node * 2 + 1] += lazy[node];
         }
         lazy[node] = 0;
     }

     if (start > end || start > r || end < l)
         return;

     if (start >= l && end <= r) {
         tree[node] += (end - start + 1) * val;
         if (start != end) {
             lazy[node * 2] += val;
             lazy[node * 2 + 1] += val;
         }
         return;
     }

     int mid = (start + end) / 2;
     updateRange(2 * node, start, mid, l, r, val);
     updateRange(2 * node + 1, mid + 1, end, l, r, val);
     tree[node] = tree[2 * node] + tree[2 * node + 1];
}

int queryRange(int node, int start, int end, int l, int r) {
     if (start > end || start > r || end < l)
         return 0;

     if (lazy[node] != 0) {
         tree[node] += (end - start + 1) * lazy[node];
         if (start != end) {
             lazy[node * 2] += lazy[node];
             lazy[node * 2 + 1] += lazy[node];
         }
         lazy[node] = 0;
     }

     if (start >= l && end <= r)
         return tree[node];

     int mid = (start + end) / 2;
     int p1 = queryRange(2 * node, start, mid, l, r);
     int p2 = queryRange(2 * node + 1, mid + 1, end, l, r);
     return p1 + p2;
}
```

Listing 3: Segment Tree Implementation

```
public:
    SegmentTree(const vector<int>& input) {
        A = input;
        n = A.size();
        tree.resize(4 * n);
        lazy.resize(4 * n, 0);
        build(1, 0, n - 1);
    }

    void update(int l, int r, int val) {
        updateRange(1, 0, n - 1, l, r, val);
    }

    int query(int l, int r) {
        return queryRange(1, 0, n - 1, l, r);
    }
};
```

3.6 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT) and its inverse in $\mathcal{O}(n \log n)$ time. In competitive programming, it is mainly used for fast polynomial multiplication, which can be used in problems like convolutions and combinatorics.

Given two polynomials $A(x)$ and $B(x)$ of degree n, FFT computes the product $C(x) = A(x) \cdot B(x)$ in efficient manner.

Core Idea

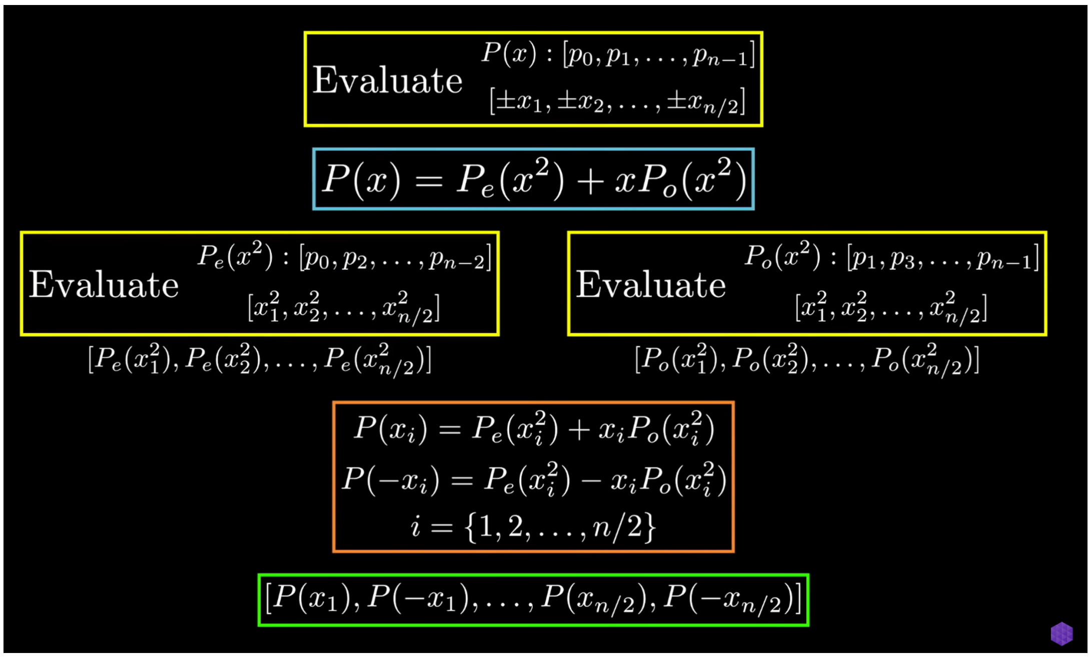
FFT is based on the idea of evaluating a polynomial at roots of unity, multiplying pointwise, and then interpolating back. The key steps are:

- Represent the polynomials as vectors of coefficients.
- Perform DFT on both vectors using a divide and conquer approach, evaluating the polynomials at n complex roots of unity.
- Multiply the results pointwise.
- Apply inverse FFT to get the coefficients of the result polynomial.

The recursion in FFT splits the problem into even and odd indexed parts, halving the size at each level, yielding an $\mathcal{O}(n \log n)$ runtime.

Complexity

- Time: $\mathcal{O}(n \log n)$
- Space: $\mathcal{O}(n)$



Evaluate $P(x) : [p_0, p_1, \ldots, p_{n-1}]$ $[\pm x_1, \pm x_2, \ldots, \pm x_{n/2}]$

$$\boxed{P(x) = P_e(x^2) + xP_o(x^2)}$$

Evaluate $P_e(x^2) : [p_0, p_2, \ldots, p_{n-2}]$ $[x_1^2, x_2^2, \ldots, x_{n/2}^2]$

$[P_e(x_1^2), P_e(x_2^2), \ldots, P_e(x_{n/2}^2)]$

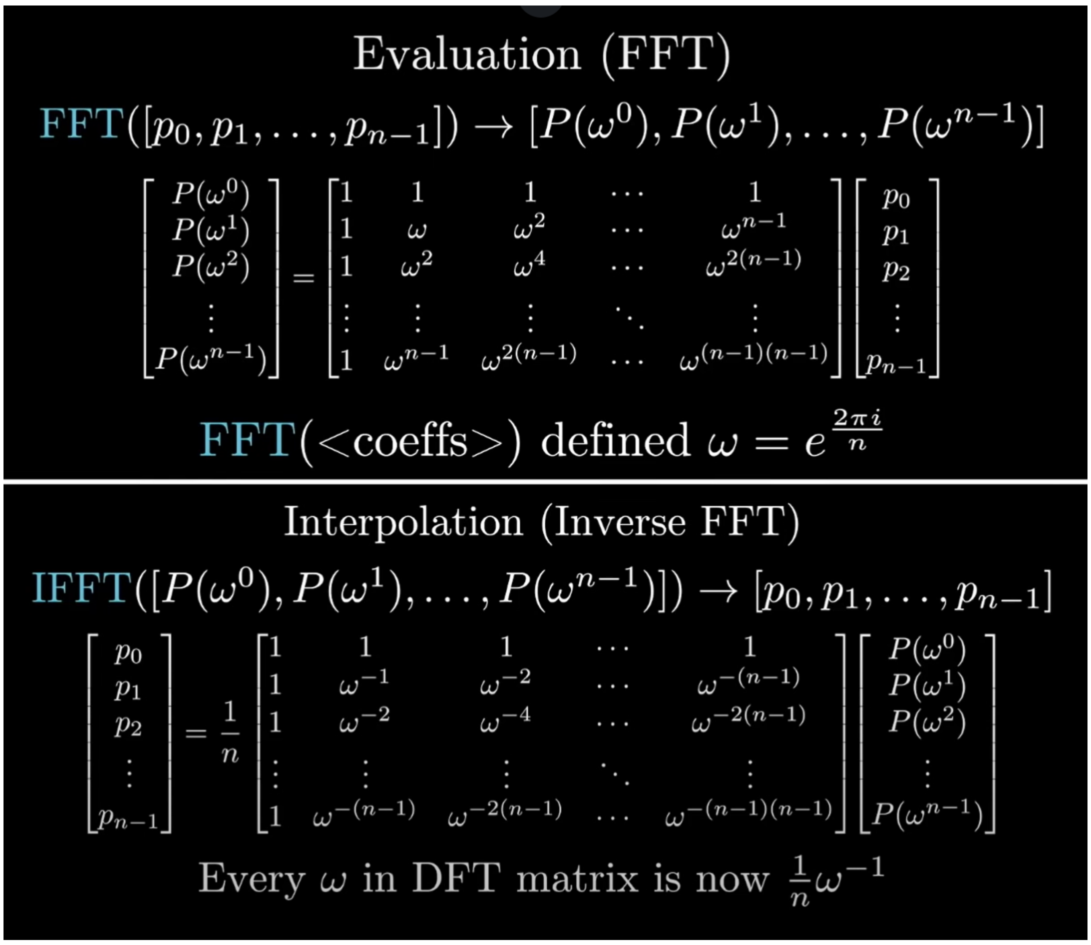
Evaluate $P_o(x^2) : [p_1, p_3, \ldots, p_{n-1}]$ $[x_1^2, x_2^2, \ldots, x_{n/2}^2]$

$[P_o(x_1^2), P_o(x_2^2), \ldots, P_o(x_{n/2}^2)]$

$$P(x_i) = P_e(x_i^2) + x_iP_o(x_i^2)$$
$$P(-x_i) = P_e(x_i^2) - x_iP_o(x_i^2)$$
$$i = \{1, 2, \ldots, n/2\}$$

$$[P(x_1), P(-x_1), \ldots, P(x_{n/2}), P(-x_{n/2})]$$

Figure 3.1: Fast Fourier Transform Overall Logic



Evaluation (FFT)

$$\text{FFT}([p_0, p_1, \ldots, p_{n-1}]) \to [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$$

$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

$\text{FFT}(<\text{coeffs}>)$ defined $\omega = e^{\frac{2\pi i}{n}}$

Interpolation (Inverse FFT)

$$\text{IFFT}([P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]) \to [p_0, p_1, \ldots, p_{n-1}]$$

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix}$$

Every ω in DFT matrix is now $\frac{1}{n}\omega^{-1}$

Figure 3.2: Matrix Form of FFT Steps

Code

```
typedef complex<double> cd;
const double PI = acos(-1);

void fft(vector<cd> &a, bool invert)
{
    int n = a.size();
    if (n == 1)
        return;

    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++)
    {
        a0[i] = a[i * 2];
        a1[i] = a[i * 2 + 1];
    }

    fft(a0, invert);
    fft(a1, invert);

    for (int i = 0; 2 * i < n; i++)
    {
        double angle = 2 * PI * i / n * (invert ? -1 : 1);
        cd w = polar(1.0, angle);
        a[i] = a0[i] + w * a1[i];
        a[i + n / 2] = a0[i] - w * a1[i];
        if (invert)
        {
            a[i] /= 2;
            a[i + n / 2] /= 2;
        }
    }
}

vector<int> multiply(const vector<int> &a, const vector<int> &b)
{
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size()) n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);

    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];

    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

Listing 4: Fast Fourier Transform

Problems

- **POLYMUL - Polynomial Multiplication**
  Direct Implementation of the FFT, given two polynomials we need to return the polynomial created after their multiplication.

3.7 Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is the modular integer analogue of the Fast Fourier Transform (FFT). Like FFT, it is used to multiply polynomials in $\mathcal{O}(n \log n)$ time but over a finite fields, which avoids precision issues.

Core Idea

NTT uses similar ideas as FFT but replaces complex roots of unity with primitive roots modulo a prime number p.

Steps:

- Choose a prime p such that $p = c \cdot 2^k + 1$, where k is the maximum size of the transform.

- Find a primitive root g modulo p (primitive root is the number for the given field such that its powers mod p can give all numbers from 1 to p-1), such that $g^{(p-1)/n}$ gives an n-th root of unity modulo p.

- Perform the same divide and conquer structure as in FFT, but using modular arithmetic.

- Use the inverse NTT to convert from point value form back to coefficient form, using modular inverse.

Why It Works

NTT replaces complex arithmetic with modular arithmetic:

- Roots are chosen to exist in the finite field $\mathbf{Z}_p$.

- Polynomial multiplication works the same as in FFT, by evaluating, multiplying pointwise, and interpolating.

- It guarantees exact results and avoids floating-point rounding errors that are common in FFT.

The condition that $p = c \cdot 2^k + 1$ ensures that 2^k-th roots of unity exist modulo p, which is necessary for recursive halving in the transform.

Common primes used:

- $998244353 = 119 \cdot 2^{23} + 1$, primitive root $g = 3$
- $7340033 = 7 \cdot 2^{20} + 1$, primitive root $g = 3$

Complexity

- Time: $\mathcal{O}(n \log n)$
- Space: $\mathcal{O}(n)$

Note

No specific problem was practiced for the Number Theoretic Transform. Instead, I focused on understanding the underlying theory through online resources, including explanatory articles and visualization-based video tutorials.

3.8 Matrix Exponentiation

Matrix Exponentiation is a powerful technique to solve k order linear recurrence relations efficiently in $\mathcal{O}(k^3 \log n)$ time. Instead of computing terms of the recurrence iteratively (which takes $\mathcal{O}(k^3 n)$ time), we can represent the recurrence as a matrix transformation and raise it to a high power using binary exponentiation.

It is especially useful when n is large (e.g., $n = 10^{18}$) and the recurrence relation is linear with constant coefficients.

Core Idea

A linear recurrence like

$$F(n) = a_1 F(n-1) + a_2 F(n-2) + \cdots + a_k F(n-k)$$

can be rewritten in matrix form:

$$\begin{bmatrix} F(n) \\ F(n-1) \\ \vdots \\ F(n-k+1) \end{bmatrix} = A \cdot \begin{bmatrix} F(n-1) \\ F(n-2) \\ \vdots \\ F(n-k) \end{bmatrix}$$

Now, to compute $F(n)$, we raise this matrix to the power $n - k$ and multiply it with the initial vector $[F(k), F(k-1), \ldots, F(1)]^T$.

Where A is a $k \times k$ transformation matrix of the form:

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_{k-1} & a_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

Why It Works

- Exponentiating the matrix simulates multiple steps in a single operation.
- Binary exponentiation allows this to be done in $\mathcal{O}(k^3 \log n)$ time.

Complexity

- Time: $\mathcal{O}(k^3 \log n)$
- Space: $\mathcal{O}(k^2)$ for matrix storage.

Code

```
vector<vector<ll>> multiply(vector<vector<ll>>& A, vector<vector<ll>>&
   B) {
    int k = A.size();
    vector<vector<ll>> C(k, vector<ll>(k));
    for (int i = 0; i < k; ++i)
        for (int j = 0; j < k; ++j)
            for (int x = 0; x < k; ++x)
                C[i][j] = (C[i][j] + A[i][x] * B[x][j]) % MOD;
    return C;
}

vector<vector<ll>> power(vector<vector<ll>> A, ll n) {
    int k = A.size();
    vector<vector<ll>> res(k, vector<ll>(k));
    for (int i = 0; i < k; ++i) res[i][i] = 1;
    while (n > 0) {
        if (n & 1) res = multiply(res, A);
        A = multiply(A, A);
        n >>= 1;
    }
    return res;
}
```

Listing 5: Matrix Exponentiation for square matrix of any order

Problems

- **Codeforces - Fibonacci**
  Compute n-th Fibonacci number modulo $10^9 + 7$. Requires efficient solution via matrix exponentiation.

- **Codeforces - Knight Paths**
  You're given a grid and asked to find how many ways a knight can move from one cell to another in exactly k steps.
  At first, this looks like a standard dynamic programming or BFS problem, but the clever trick is to turn it into a matrix exponentiation problem. You create a transition matrix where each cell represents a position on the grid, and a 1 in position (i, j) means you can move from cell i to cell j with a knight move.
  You need redefine matrix multiplication so that addition becomes normal addition, and multiplication becomes counting the number of paths.

3.9 Berlekamp–Massey Algorithm

The Berlekamp–Massey algorithm finds the shortest linear recurrence relation for a given sequence. Given a sequence $S = [s_0, s_1, \ldots, s_n]$, the algorithm returns the minimal linear recurrence:

$$s_k = c_1 s_{k-1} + c_2 s_{k-2} + \cdots + c_d s_{k-d}$$

It's especially powerful when combined with matrix exponentiation to compute terms like $s_{10^{18}}$ in efficient time.

Core Idea

The algorithm iteratively builds the shortest recurrence $C(x)$.

Let $C(x)$ be the current guess for the recurrence. On encountering a discrepancy, the algorithm updates $C(x)$ by subtracting a scaled version of a past valid recurrence $B(x)$.

The main update logic is:

$$C(x) \leftarrow C(x) - \frac{d}{b} x^{k-m} B(x)$$

Where: - d is the discrepancy at step k - b is the last discrepancy where $B(x)$ was updated - m is the index at which $B(x)$ was last updated

Why It Works

- Based on the assumption that a minimal recurrence exists.

- Updates always preserve correctness for all previous terms.

- Each update increases the degree of the recurrence only when necessary.

Code

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

struct Matrix
{
    vector<vector<ll>> mat;
    int size;

    Matrix(int n) : size(n)
    {
        mat.assign(n, vector<ll>(n));
    }

    Matrix operator*(const Matrix &rhs) const
    {
        Matrix res(size);
        for (int i = 0; i < size; ++i)
            for (int k = 0; k < size; ++k)
                for (int j = 0; j < size; ++j)
                    res.mat[i][j] = (res.mat[i][j] + mat[i][k] * rhs.
                        mat[k][j]) % MOD;
        return res;
    }

    Matrix pow(ll exp) const
    {
        Matrix res(size);
        for (int i = 0; i < size; ++i)
            res.mat[i][i] = 1;
        Matrix base = *this;
        while (exp)
        {
            if (exp & 1)
                res = res * base;
            base = base * base;
            exp >>= 1;
        }
        return res;
    }
};

// Modular inverse using Fermat's little theorem (MOD must be prime)
ll modinv(ll a, ll m = MOD)
{
    ll res = 1, e = m - 2;
    while (e)
    {
        if (e & 1)
            res = res * a % m;
        a = a * a % m;
        e >>= 1;
    }
    return res;
}
```

Listing 6: Matrix struct definition

```
vector<ll> berlekampMassey(const vector<ll> &s)
{
    int n = s.size();
    vector<ll> C = {1}, B = {1};
    ll b = 1;
    int L = 0, m = 1;

    for (int i = 0; i < n; ++i)
    {
        ll d = 0;
        for (int j = 0; j < (int)C.size(); ++j)
        {
            d = (d + C[j] * s[i - j]) % MOD;
        }

        if (d == 0)
        {
            ++m;
            continue;
        }

        vector<ll> T = C;
        ll coef = d * modinv(b) % MOD;
        if (C.size() < B.size() + m)
            C.resize(B.size() + m);
        for (int j = 0; j < (int)B.size(); ++j)
        {
            C[j + m] = (C[j + m] - coef * B[j] % MOD + MOD) % MOD;
        }

        if (2 * L <= i)
        {
            B = T;
            L = i + 1 - L;
            b = d;
            m = 1;
        }
        else ++m;
    }

    C.erase(C.begin());
    for (auto &x : C)
        x = (MOD - x) % MOD;
    return C;
}

Matrix buildMatrix(const vector<ll> &rec)
{
    int k = rec.size();
    Matrix M(k);
    for (int i = 0; i < k - 1; ++i) M.mat[i + 1][i] = 1;
    for (int j = 0; j < k; ++j) M.mat[0][j] = rec[j];
    return M;
}
```

Listing 7: Berlekamp Massey Algorithm

```
ll computeKthTerm(const vector<ll> &initial, const vector<ll> &rec, ll
    k)
{
    int d = rec.size();
    if (k < (int)initial.size())
        return initial[k];

    Matrix M = buildMatrix(rec);
    Matrix Mk = M.pow(k - d + 1);

    ll result = 0;
    for (int i = 0; i < d; ++i)
    {
        result = (result + Mk.mat[0][i] * initial[d - 1 - i]) % MOD;
    }
    return result;
}
```

Listing 8: Nth term Computation

Complexity

- Time: $\mathcal{O}(n^2)$ — where n is the length of the sequence.

Problems

- **Library Checker - Find Linear Recurrence**
  Given a sequence, compute the shortest linear recurrence relation. This is a direct application of Berlekamp–Massey.

- **Codeforces 622F - The Sum of the k-th Powers**
  In this problem, we are asked to compute the sum $1^k + 2^k + \cdots + n^k$ modulo $10^9 + 7$. Instead of using the Faulhaber's formula that uses Bernoulli fractions here, we can take advantage of the fact that the sequence of k-th power sums is a polynomial of degree $k + 1$, and use the Berlekamp–Massey algorithm to compute a linear recurrence for it. We first precompute the values of the prefix sum sequence up to $2k + 5$ terms, and then apply BM to derive the recurrence. Once the recurrence is known, we can compute the n-th term efficiently using matrix exponentiation.

4 Conclusion

I explored many important algorithms used in competitive programming in this time, starting from basic techniques to more advanced mathematical methods for solving problems efficiently.

What I Learned:

- Improved my understanding of key algorithm strategies.
- Practiced using these algorithms on a variety of problems.
- Got familiar with the math behind advanced techniques like FFT, NTT, etc.

What I Plan to Do Next:

- Keep solving and grinding problems regularly on platforms like LeetCode and give contests on Codeforces to get better and faster.
- Study and practice more advanced algorithms.

Overall, this project helped me build a strong foundation in competitive programming.