International Institute of Information Technology, Hyderabad

Algorithmic Theory

Data Structures, Algorithms and Competitive Programming

College Notes

Maintainers: Animesh Sinha, Gourang Tandon, Arpan Dasgupta, Yogottam Khandelwal, Aman Kumar Singh.

December 10, 2019

Chapter 1

Heaps

1.1 Question Patterns

1.1.1 Generic Types

- Find the kth Minimum or Maximum. This can also be on arrays or trees with online insertion or deletion.
- Priority Queue questions. Typically greedy algorithms best implemented this way. (eg. Dijkstra)

1.1.2 Specific Illustrations

When not to use Heaps *Question: Find the kth-Maximum sum of all subarrays of any given array.* This question demostrates that when we have **k-sorted elements appended to our queue of k-maximum elements** at one go, it's better to use an actual **Sorted array and the Merge Function for Updates**.

Lazy Deletion Heaps cannot search, so heaps cannot delete. A simple solution for this is to maintain another heap of all deleted elements, and if the actual heap and deleted heap have the same top element, keep popping them out. We only care about the top element, so we can be Lazy in the deletion of the elements lower down in the tree.

1.2 Elementary Theory

1.2.1 Basic Operations

Heapification A **O(n)** algorithm exists for Heapification. It uses the standard sift-down procedure, but starts by correcting the lowest layers first. So start at the end of the array and move back.

Insert A **O(log n)** algorithm is used to insert. We push the element at the end of the array and sift-up or sift-down the element till the element is at a valid locale.

Delete For a **O(log n)** deletion algorithm, we must swap the last element with the element to be deleted, sift-up/down the former last element, and pop-out the last element. This is only possible if we have a refernce to the element, search and delete take O(n) time to perform on heaps.

Minimum/Maximum Minimum on a MinHeap takes **O(1)** time, and so does Maximum on a MaxHeap.

1.2.2 Implementation Code

```
#ifndef CODE_BINARY_HEAP_H
#define CODE_BINARY_HEAP_H

#include <cstdint>
#include <functional>
#include <iostream>
#include <vector>

using namespace std;

template <class Type>
class BinaryHeap {
   protected:
    function<bool(Type, Type)> comparator;

    vector<Type> data;
    void siftDown(unsigned long pos) {
        if (pos >= this->data.size())
            return;
        bool lChildExists = this->lChild(pos) < this
           ->data.size(),
             rChildExists = this->rChild(pos) < this
                ->data.size();
        if ((lChildExists &&
             comparator(this->data[this->lChild(pos)
                ], this->data[pos])) ||
            (rChildExists &&
             comparator(this->data[this->rChild(pos)
                ], this->data[pos]))) {
            if (!rChildExists ||
                (lChildExists && comparator(this->
                   data[this->lChild(pos)],
                                            this->
                                               data[
                                               this
                                               ->
                                               rChild
                                               (pos)
                                               ])))
                                               {
```

```
                this->swap(pos, this->lChild(pos));
                siftDown(this->lChild(pos));
            } else {
                this->swap(pos, this->rChild(pos));
                siftDown(this->rChild(pos));
            }
        }
    }
    void siftUp(unsigned long pos) {
        if (pos == 0)
            return;  // This is the Root Element
        if (comparator(this->data[pos], this->data[
           this->parent(pos)])) {
            this->swap(pos, this->parent(pos));
            this->siftUp(this->parent(pos));
        }
    }
    inline unsigned long parent(unsigned long val) {
        return (val - 1) / 2;
    }
    inline unsigned long lChild(unsigned long val) {
        return 2 * val + 1;
    }
    inline unsigned long rChild(unsigned long val) {
        return 2 * val + 2;
    }
    void swap(unsigned long x, unsigned long y) {
        this->data[x] ^= this->data[y];
        this->data[y] ^= this->data[x];
        this->data[x] ^= this->data[y];
    }
public:
    explicit BinaryHeap(const vector<Type> &list,
                        function<Type(Type, Type)>
                           heapComparator = less<>()
                           ) {
        this->comparator = heapComparator;
        for (auto i : list)
            this->data.push_back(i);
        for (long i = this->data.size() - 1; i >= 0;
            i--) {
            this->siftDown((unsigned)i);
        }
    }
    explicit BinaryHeap(function<Type(Type, Type)>
       heapComparator = less<>()) {
        this->comparator = heapComparator;
        this->data = vector<Type>();
    }
    BinaryHeap(const BinaryHeap &heap) {
        this->comparator = heap.comparator;
        for (auto val : heap.data)
            this->data.push_back(val);
    }
```

```
    void insert(int val) {
        this->data.push_back(val);
        this->siftUp(this->data.size() - 1);
    }
    void remove(unsigned long pos) {
        this->swap(this->data.size() - 1, pos);
        this->data.pop_back();
        this->siftUp(pos);
        this->siftDown(pos);
    }
    Type top() {
        return this->data[0];
    }
    vector<Type> sort() {
        vector<Type> result;
        BinaryHeap copy(*this);
        while (!copy.data.empty()) {
            result.push_back(copy.top());
            copy.remove(0);
        }
        return result;
    }
};

#endif  // CODE_BINARY_HEAP_H
```

Chapter 2

Range Queries

2.1 Square Root Decomposition

Break the array down into chunks of $\sqrt{n}$. Store the answer to these chunks. The answer to range queries is the answer in the starting chunk after L, plus in the ending chunk before R, plus the stored results of everything in between. Updates can be done in $O(\sqrt{n})$ steps by just updating the result of the chunk.

2.1.1 Motivating Examples

This cannot be solved by a Segment Tree (easily).

Example 1.1

Question: Given and array, support update and query operation for number of elements less than k in the given range.
Solution: Maintain sorted vector of each block of the square-root decomposition. Search over all blocks and find lowerbound(k), the sum of these counts gives the answer in $O(\sqrt{n}log(n))$. For update, just sort again, using insertion sort you get $O(\sqrt{n})$.

2.2 Mo's Algorithm

This is an algorithm to handle offline queries by sorting them and trying to cleverly avoid recomputing portions that were already solved for in the previous queries. (Direcly usable with associative, commutative, invertible operations).

Algorithm

Break down the array in blocks of size $\sqrt{n}$. Sort queries by starting block, them by ending position. The right pointer keeps moving forward for each block, the

left pointer keeps moving back and forth withing a block, adding elements as it goes back and subtracting as it goes forth. This results in all queries being solved in $O(n\sqrt{n})$ time.

Example 2.1

Question: Given an array, find the number of elements distinct elements in range (l, r). Sort the queries first by the starting block, then by the end position. Maintian a frequency array of all elements currently between right and left pointers. For each of the $O(\sqrt{n})$ blocks, the start pointer moves atmost $O(\sqrt{n})$ times back and forth in the block, adding and deleting elements. For each block the right pointer only goes forward adding in elements, $O(\sqrt{n})$ blocks each taking $O(n)$ time.

Example 2.2

Question: Given an array, find the f(s)*f(s)*s for all distinct s in range (l, r), where f(s) is the frequency of s.
Same Algorithm, and same frequency array as above, just find $f(s)*f(s)*s$ instead of $\delta(f(s))$ as above.

2.3 Segment Trees

2.3.1 Iterative Implementation

```
#ifndef CODE_SEGTREE_H
#define CODE_SEGTREE_H

#include <cmath>
#include <functional>
#include <iostream>
#include <vector>

using namespace std;

template <class Type>
class SegmentTree {
   protected:
    vector<Type> data;
    unsigned long size;
    inline unsigned long parent(unsigned long i) {
        return i >> 1;
    }
    inline unsigned long lChild(unsigned long i) {
        return i << 1;
    }
    inline unsigned long rChild(unsigned long i) {
        return i << 1 | 1;
```

```
    }
    inline unsigned long sibling(unsigned long i) {
        return i ^ 1;
    }
    inline unsigned long element(unsigned long i) {
        return i + size;
    }
    inline bool isRoot(unsigned long i) {
        return i == 1;
    }
    inline bool islChild(unsigned long i) {
        return (i & 1) == 0;
    }
    inline bool isrChild(unsigned long i) {
        return (i & 1) != 0;
    }
    function<Type(Type, Type)> operation;
    Type defaultValue;

public:
    explicit SegmentTree(const vector<Type> &list,
                         function<Type(Type, Type)>
                            segOperation,
                         Type defaultTo) {
        size = (1ul << (long)ceil(log2(list.size()))
            );
        data = vector<Type>(size * 2, defaultTo);
        defaultValue = defaultTo;
        operation = segOperation;
        for (unsigned long i = 0; i < list.size(); i
            ++)
            data[i + size] = list[i];
        for (unsigned long i = size - 1; i > 0; --i)
            data[i] = operation(data[lChild(i)],
                data[rChild(i)]);
    }
    void modify(unsigned long position, Type value)
        {
        data[element(position)] = value;
        for (data[position = element(position)]; !
            isRoot(position);
             position = parent(position)) {
            if (islChild(position))
                data[parent(position)] =
                    operation(data[position], data[
                        sibling(position)]);
            if (isrChild(position))
                data[parent(position)] =
                    operation(data[sibling(position)
                        ], data[position]);
        }
    }
    Type query(unsigned long l, unsigned long r) {
        Type lAccumulator = defaultValue,
            rAccumulator = defaultValue;
        for (l = element(l), r = element(r); l < r;
```

```
                l = parent(l), r = parent(r)) {
            if (isrChild(l)) {
                lAccumulator = operation(
                    lAccumulator, data[l++]);
            }
            if (isrChild(r)) {
                rAccumulator = operation(data[--r],
                    rAccumulator);
            }
        }
        return operation(lAccumulator, rAccumulator)
            ;
    }
};
template <class Type>
class SegmentUpdate {
   protected:
    vector<Type> data;
    unsigned long size;
    inline unsigned long parent(unsigned long i) {
        return i >> 1;
    }
    inline unsigned long lChild(unsigned long i) {
        return i << 1;
    }
    inline unsigned long rChild(unsigned long i) {
        return i << 1 | 1;
    }
    inline unsigned long sibling(unsigned long i) {
        return i ^ 1;
    }
    inline unsigned long element(unsigned long i) {
        return i + size;
    }
    inline bool isRoot(unsigned long i) {
        return i == 1;
    }
    inline bool islChild(unsigned long i) {
        return (i & 1) == 0;
    }
    inline bool isrChild(unsigned long i) {
        return (i & 1) != 0;
    }
    function<Type(Type, Type)> operation;
    Type defaultValue;

   public:
    explicit SegmentUpdate(const vector<Type> &list,
                           function<Type(Type, Type)
                               > segOperation,
                           Type defaultTo) {
        size = (1ul << (long)ceil(log2(list.size()))
            );
        data = vector<Type>(size * 2, defaultTo);
        defaultValue = defaultTo;
        operation = segOperation;
```

```
        for (unsigned long i = 0; i < list.size(); i
            ++)
            data[i + size] = list[i];
        for (unsigned long i = size - 1; i > 0; --i)
            data[i] = operation(data[lChild(i)],
                data[rChild(i)]);
    }
    void modify(unsigned long l, unsigned long r,
        Type value) {
        for (l = element(l), r = element(r); l < r;
             l = parent(l), r = parent(r)) {
            if (isrChild(l)) {
                data[l] = operation(data[l], value);
                l++;
            }
            if (isrChild(r)) {
                --r;
                data[r] = operation(data[r], value);
            }
        }
    }
    Type query(unsigned long position) {
        Type accumulator = defaultValue;
        for (position = element(position);; position
            = parent(position)) {
            accumulator = operation(accumulator,
                data[position]);
            if (isRoot(position))
                break;
        }
        return accumulator;
    }
};

template <typename Type>
class LazySegtree {
    int size;
    vector<Type> tree, lazy;
    Type _default;
    function<Type(Type, Type)> _operation;
    function<Type(Type, Type)> _setter;

    void split(int node) {
        lazy[2 * node] = _setter(lazy[2 * node],
            lazy[node]);
        tree[2 * node] = _setter(tree[2 * node],
            lazy[node]);
        lazy[2 * node + 1] = _setter(lazy[2 * node +
            1], lazy[node]);
        tree[2 * node + 1] = _setter(tree[2 * node +
            1], lazy[node]);
        lazy[node] = _default;
    }
    void merge(int node) {
        tree[node] = _operation(tree[2 * node], tree
            [2 * node + 1]);
```

```
    }
  public:
    LazySegtree(int n, const function<Type(Type,
        Type)> &op,
                const function<Type(Type, Type)> &
                    set, const Type identity) {
        for (size = 1; size < n; size <<= 1)
            ;
        _setter = set, _operation = op, _default =
            identity;
        tree.assign(2 * size, _default);
        lazy.assign(2 * size, _default);
    }
    void modify(int l, int r, Type delta, int node =
        1, int x = 0, int y = -1) {
        if (y == -1)
            y = size;
        if (r <= x || l >= y)
            return;
        if (l <= x && y <= r) {
            lazy[node] = _setter(lazy[node], delta);
            tree[node] = _setter(tree[node], delta);
            return;
        }
        split(node);
        modify(l, r, delta, 2 * node, x, (x + y) /
            2);
        modify(l, r, delta, 2 * node + 1, (x + y) /
            2, y);
        merge(node);
    }
    Type query(int l, int r, int node = 1, int x =
        0, int y = -1) {
        if (y == -1)
            y = size;
        if (r <= x || l >= y)
            return _default;
        if (l <= x && y <= r) {
            return tree[node];
        }
        split(node);
        Type lres = query(l, r, 2 * node, x, (x + y)
            / 2);
        Type rres = query(l, r, 2 * node + 1, (x + y
            ) / 2, y);
        merge(node);
        return _operation(lres, rres);
    }
};
template <typename Type>
class ImplicitSegtree {
    struct Node {
        Type data = 0, lazy = 0;
        Node *l_ptr = nullptr, *r_ptr = nullptr;
```

```
        Node *l_child() {
            if (l_ptr == nullptr) {
                l_ptr = new Node;
                r_ptr = new Node;
            }
            return l_ptr;
        }
        Node *r_child() {
            if (r_ptr == nullptr) {
                l_ptr = new Node;
                r_ptr = new Node;
            }
            return r_ptr;
        }
    };
    int size;
    Node *root;
    Type _default;
    function<Type(Type, Type)> _operation;
    function<Type(Type, Type)> _setter;

    void split(Node *node) {
        node->l_child()->lazy = _setter(node->
            l_child()->lazy, node->lazy);
        node->r_child()->lazy = _setter(node->
            r_child()->lazy, node->lazy);
        node->l_child()->data = _setter(node->
            l_child()->data, node->lazy);
        node->r_child()->data = _setter(node->
            r_child()->data, node->lazy);
        node->lazy = _default;
    }
    void merge(Node *node) {
        node->data = _operation(node->l_child()->
            data, node->r_child()->data);
    }

public:
    ImplicitSegtree(int n, const function<Type(Type,
        Type)> &op,
                    const function<Type(Type, Type)>
                        &set,
                    const Type identity) {
        for (size = 1; size < n; size <<= 1)
            ;
        _setter = set, _operation = op, _default =
            identity;
        root = new Node;
    }

    void modify(int l, int r, Type delta, Node *node
        = nullptr, int x = 0,
                int y = -1) {
        if (node == nullptr)
            node = root, y = size;
        if (r <= x || l >= y)
            return;
```

```
            if (l <= x && y <= r) {
                node->lazy = _setter(node->lazy, delta);
                node->data = _setter(node->data, delta);
                return;
            }
            split(node);
            modify(l, r, delta, node->l_child(), x, (x +
                y) / 2);
            modify(l, r, delta, node->r_child(), (x + y)
                / 2, y);
            merge(node);
        }
        Type query(int l, int r, Node *node = nullptr,
           int x = 0, int y = -1) {
            if (node == nullptr)
                node = root, y = size;
            if (r <= x || l >= y)
                return _default;
            if (l <= x && y <= r) {
                return node->data;
            }
            split(node);
            Type lres = query(l, r, node->l_child(), x,
               (x + y) / 2);
            Type rres = query(l, r, node->r_child(), (x
               + y) / 2, y);
            merge(node);
            return _operation(lres, rres);
        }
    };
    template <typename Type>
    class ImplicitSegupdate {
        struct Node {
            Type data = 0;
            Node *l_ptr = nullptr, *r_ptr = nullptr;
            Node *l_child() {
                if (l_ptr == nullptr) {
                    l_ptr = new Node;
                    r_ptr = new Node;
                }
                return l_ptr;
            }
            Node *r_child() {
                if (r_ptr == nullptr) {
                    l_ptr = new Node;
                    r_ptr = new Node;
                }
                return r_ptr;
            }
        };
        int size;
        Node *root;
        function<Type(Type, Type)> _setter;

       public:
        ImplicitSegupdate(int n, const function<Type(
```

```
        Type, Type)> &set) {
        for (size = 1; size < n; size <<= 1)
            ;
        _setter = set;
        root = new Node;
    }

    void modify(int l, int r, Type delta, Node *node
        = nullptr, int x = 0,
                int y = -1) {
        if (node == nullptr)
            node = root, y = size;
        if (r <= x || l >= y)
            return;
        if (l <= x && y <= r) {
            node->data = _setter(node->data, delta);
            return;
        }
        modify(l, r, delta, node->l_child(), x, (x +
            y) / 2);
        modify(l, r, delta, node->r_child(), (x + y)
            / 2, y);
    }
    Type query(int p, Node *node = nullptr, int x =
       0, int y = -1) {
        if (node == nullptr)
            node = root, y = size;
        if (x == p && y == p + 1) {
            return node->data;
        }
        0 if (x <= p && p < (x + y) / 2) return
            _setter(
            node->data, query(p, node->l_child(), x,
                (x + y) / 2));
        else return _setter(node->data,
                            query(p, node->r_child()
                                , (x + y) / 2, y));
    }
};

#endif  // CODE_SEGTREE_H
```

Chapter 3

Dynamic Programming

The focus of this chapter would be to enlist all the optimizations to a DP possible and types of recurrences solvable using them.

3.1 Matrix Exponentiation

Example 1.1

Number of ways to construct an array starting in X, and ending in Y, with no two adjacent elements are the same.

$$dp[i] = \begin{bmatrix} dp[i][\text{CLASH}] \\ dp[i][\text{CLEAN}] \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ k-1 & k-2 \end{bmatrix} \times dp[i-1]$$

3.2 Bitmasks

3.2.1 Classical Bitmasks

The idea is trivial, we take each bitmask to represent an arbitrary subset of any given set.

3.2.2 Sum over Subsets

We want to sum for each mask, some given function (as an array) for all it's submasks. We can take each mask to start off, and then go down a series of all it's subsets in decreasing order of value of mask, the algorithm for this will be $j = (j-1)\ \&\ mask$, initially $j_0 = mask$.

Time complexity is the following:

$$T(n) = \sum_{k=0}^{n-1} C_k^n 2^k = 3^k \tag{3.1}$$

Example 2.1

ind the number of ways in the given array of 10^6 numbers

Chapter 4

Game Theory

4.1 NIM Games and Sprague-Grundy Theorem

4.2 Take Away Games

4.2.1 Identifying the Losing States

Theorem 2.1

Let H_i denote all the losing states, and $f(x)$ denote the number of stones that can be removed in the next move after x stones in the previous. Then we can find the losing states as follows.

$$H_{k+1} = H_k + H_m, \quad where \;\; m = min\{j : f(H_j) \geq H_k\} \tag{4.1}$$

The idea is that we can remove any $H_j + H_k$ stones, we can think of them as two separate piles. We cannot win on either pile, so the only way to win is when the H_j pile ends, the last move was enough that $f(last\,move) \geq H_k$ so that we can win next move. If this is not possible, then the state is losing.

KEY IDEA: Find the RECURRENCE, make a SOLUTION HYPOTHESIS by monitoring the pattern and prove it BY INDUCTION to get all the losing states.

4.2.2 A few Example Functions

Example: f(x) = x

1 stone is losing, so $H_1 = 1$. And whenever H_k is losing, the $min\{H_j : f(H_j) \geq H_k\} = min\{H_j : H_j \geq H_k\} = H_k$, therefore the losing states are 2^n.

Example: f(x) = 2x - 1

1 stone is losing, so $H_1 = 1$. Our Hypothesis, $H_k = 2H_{k-1}$. And whenever H_k is losing, the $min\{H_j : f(H_j) \geq H_k\} = min\{H_j : 2 * H_j - 1 \geq H_k\} = H_k$, therefore the losing states are 2^n.

Example: f(x) = 2x: Fibonacci NIM

1 stone is losing, so $H_1 = 1$. Our Hypothesis, $H_k = H_{k-1}+H_{k-2}$. And whenever H_k is losing, the $min\{H_j : f(H_j) \geq H_k\} = min\{H_j : 2 * H_j \geq H_k\} = H_{k-1}$, therefore the losing states are the Fibonacci numbers.

4.2.3 Winning Strategy

New Binary number systems

We find that we can express any number as a sum of the values of H_1, H_2, ..., so we construct a binary like number system where a the place value of the i-th digit is H_i and the face value is 0 or 1. Let's call this H-binary. (Note: This expression is unique and complete for powers of 2, and for the Fibonacci numbers - Zeckendorf theorem, as in the above examples).

The greedy strategy

Given any starting state that is not losing, we can write out it's representation in the H-Binary system. Since this will have more than 1 ones in it's representation, we subtract the LEAST SIGNIFICANT BIT.

Now, the opponent cannot remove the next one in the representation, because of the property of number systems that $H_j > f(H_i) \ \ \forall j > i$, due to the way we found losing states H_i. Finally, when our oppnent removes any value from the form .....1000000 (Any value, last set bit 1, and 0s), he will get a 1 in the resulting representation .....0000110.

Now in our move we shall remove the lowest set bit again. This is possible, as the last move must have been greater than or equal to H_j if j is lowest set bit. (Obviously, because when we add back we need to have 1+1 = 0 to get all the numbers back to 0 and 1 at the position that could not be removed). So $f(H_j) >= H_j$, this move is possible, and we can win.

4.2.4 Proof of Victory

On our moves, we reduce the number of ones in the representation by 1. Our opponent, if he removes the one at H_j, he has to insert a 1 and position smaller than j. So he increases or keeps constant the number of ones. Obviously, the last move will be played by us, reducing the number of ones to 0 and finishing the game.

4.2.5 References

Problems

Fibonacci Nim (Direct Implementation) [ICPC Kolkata 2018] `https://www.codechef.com/KOL18ROL/problems/SNOWMAN`

Theory

Contains most of the theory mentioned above: `http://www.cut-the-knot.org/Curriculum/Games/TakeAway.shtml#theory`

4.3 Finding Invarients

Mark out a state and all it's children. Either try MINIMAX TREE, and if the state space is large, try to find invarients, specially MODULO or PARITY.

Example 3.1

Question: Start with {(4 sticks, length 4), (1 stick, length 1)}. In a move, we can break a stick or remove k sticks of length k. Last move wins. Find the winning states & strategy. Any state be (n1, n2, n3, n4). All states with (n1 + n3) % 2 == (n2 + n4) % 3 are winning positions, all others are losing. We can prove than any winning position goes to losing position and vice-versa.

Chapter 5

Mathematical Tools

5.1 Fast Fourier Transforms

5.1.1 Motivation and Purpose

We want to be able to interconvert a polynomial between **SAMPLES representation and COEFFICIENTS representation**.

A n-degree polynomial can be a n-dimentional COEFFICIENTS vector where it is easy to compute the value at any random x, and as a n-dimentional SAMPLES vector that has n (x, y) pairs, making it easy to multiply vectors. If we can convert back and forth in $O(n \cdot log(n))$, and *multiply polynomials* in SAMPLES land in $O(n)$, then we get a speed up over the typical $O(n^2)$ for multipling each coefficient with every other.

5.1.2 Algorithm

The Recurrence

Given a polynomial $P(x)$, to convert it into samples at $X = \{X_1, X_2..., X_n\}$. We can solve it using the recurrence

$$P(X) = P_{even}(x^2) + x \cdot P_{odd}(x^2) \tag{5.1}$$

where P_{even} is the polynomials with only even coefficients, P_{odd} with only odd. Note that the degree of the resulting polynomials P_{even} and P_{odd} is half of the original.

The Complex Numbers

We also want the size of the set X, at which we have to evaluate the polynomial to go down. So we can use complex numbers, for polynomial of degree n, we use **the 2^k-roots of unity** where 2^k is the smallest value power of 2 bigger than n.

The set will keep collapsing to half it's size after each step, as squares of exactly two of these roots is the same.

The Divide and Conquer

In summary, we are performing a Divide and Conquer solve, where each state is ¡P, X¿. We start with our **Original P, and X = 1**. In each divide step, we split the P into two parts, and each part gets one root of X, here +1 and -1. Then that divides in 4, at values +i, -i, +1 and -1. There are log(n) layers with 2^l polynomials of size $n/2^l$, each polynomial is to be evaluated at 1 value.

5.1.3 Some Mathematical Representations

$$\begin{bmatrix} 1 & x_1^2 & x_1^3 & x_1^4 & ... & x_1^n \\ 1 & x_2^2 & x_2^3 & x_2^4 & ... & x_2^n \\ 1 & x_3^2 & x_3^3 & x_3^4 & ... & x_3^n \\ ... & ... & ... & ... & ... & ... \\ 1 & x_n^2 & x_n^3 & x_n^4 & ... & x_n^n \end{bmatrix} \times \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ ... \\ c_n \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ ... \\ s_n \end{bmatrix} \tag{5.2}$$

Here we have the Vandermond Matrix of a set of N values for X times the coefficient vector gives the samples vector. We decided to choose our values in X as such: $X = \{1, \omega_n, \omega_n^2, \omega_n^3, ..., \omega_n^{n-1}\}$.

5.1.4 Inverse Fourier Transform

$$\begin{bmatrix} 1 & 1 & 1 & 1 & ... & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \omega_n^3 & ... & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & ... & \omega_n^{2n-2} \\ ... & ... & ... & ... & ... & ... \\ 1 & \omega_n^{n-1} & \omega_n^{2n-2} & \omega_n^{3n-3} & ... & \omega_n^{(n-1)(n-1)} \end{bmatrix}^{-1} \times \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ ... \\ s_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ ... \\ c_n \end{bmatrix} \tag{5.3}$$

Here is the definition of the operation. We are going from samples to coefficient, so we need multiplication by inverse of the matrix for Fourier. This is easy, because the inverse is just the complex conjugate divided by n.

$$V^{-1} = \bar{V}/n \tag{5.4}$$

So we can use the FFT Algorithm again, since $X = 1, \bar{\omega}^1, \bar{\omega}^2, \bar{\omega}^3, ..., \bar{\omega}^n$. And divide the answer by n. Note that X is still the same set, so no change to FFT is needed.

5.1.5 Number Theoretic Transforms

The number theoretic transform is based on generalizing the n-th primitive root of unity to a "quotient ring" instead of the usual field of complex numbers.

We take a number w that satisfies $w^n \equiv 1(modp)$ going through each of the numbers only and atmost once.

5.1.6 Code Sample

```
#include <iostream>
#include <vector>
#include <cmath>
#include <complex>
using namespace std;

// Performs the Fast Fourier Transforms
vector<complex<double>> fft(const vector<complex<
    double>> &arr, bool inverse = false)
{
  unsigned long n = arr.size();
  if (n == 1)
    return vector<complex<double>>(1, arr[0]);
  vector<complex<double>> unity_roots(n);
  for (int i = 0; i < n; i++)
  {
    double alpha = 2 * M_PI * i / n;
    unity_roots[i] = complex<double>(cos(alpha), !
        inverse ? sin(alpha) : -sin(alpha));
  }
  vector<complex<double>> arr_even(n / 2), arr_odd(n
      / 2);
  for (int i = 0; i < n / 2; i++)
  {
    arr_even[i] = arr[i * 2];
    arr_odd[i] = arr[i * 2 + 1];
  }
  vector<complex<double>> fft_even = fft(arr_even,
      inverse);
  vector<complex<double>> fft_odd = fft(arr_odd,
      inverse);
  vector<complex<double>> fft(n);
  if (inverse)
    for (auto &el : fft_even)
      el *= fft_even.size();
  if (inverse)
    for (auto &el : fft_odd)
      el *= fft_odd.size();
  for (int k = 0; k < n / 2; k++)
  {
    fft[k] = fft_even[k] + unity_roots[k] * fft_odd[
        k];
    fft[k + n / 2] = fft_even[k] - unity_roots[k] *
        fft_odd[k];
  }
  if (inverse)
    for (auto &el : fft)
      el /= fft.size();
  return fft;
}

// Performs the Number Theoretic Transforms
vector<long long> ntt(vector<long long> a, bool
    invert = false)
```

```
{
  const long long mod = 7340033;
  const long long root_forward = 5;
  const long long root_inverse = 4404020;
  const long long root_power = 1 << 20;

  unsigned long n = a.size();

  for (int i = 1, j = 0; i < n; ++i)
  {
    int bit = n >> 1;
    for (; j >= bit; bit >>= 1)
      j -= bit;
    j += bit;
    if (i < j)
      swap(a[i], a[j]);
  }

  for (int len = 2; len <= n; len <<= 1)
  {
    long long w_len = invert ? root_inverse :
       root_forward;
    for (int i = len; i < root_power; i <<= 1)
      w_len = w_len * 1ll * w_len % mod;
    for (int i = 0; i < n; i += len)
    {
      int w = 1;
      for (int j = 0; j < len / 2; ++j)
      {
        int u = a[i + j], v = int(a[i + j + len / 2]
            * 1ll * w % mod);
        a[i + j] = u + v < mod ? u + v : u + v - mod
           ;
        a[i + j + len / 2] = u - v >= 0 ? u - v : u
           - v + mod;
        w = int(w * 1ll * w_len % mod);
      }
    }
  }
  auto mod_inverse = [](long long x, long long
     modulus) {
    long long cumulative = x, result = 1;
    for (long long power = modulus - 2; power > 0;
       power /= 2)
    {
      if (power % 2 == 1)
        result = (result * cumulative) % modulus;
      cumulative = (cumulative * cumulative) %
         modulus;
    }
    return result;
  };

  if (invert)
  {
    int nrev = mod_inverse(n, mod);
    for (int i = 0; i < n; ++i)
      a[i] = int(a[i] * 1ll * nrev % mod);
```

```
    }
    return a;
}
```

5.2 Group Theory

5.2.1 Burnside's Lemma

It states the number of elements in the Orbit of a When a Group G acts on a Set X is the mean of the number of unique elements in the subgroup due to

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g| \tag{5.5}$$

where X^g is the number of elements in set X fixed by the element g,

5.3 Number Theory

5.3.1 Stern-Brocot Tree

It is a **Binary Search Tree** of Fractions such than the path from the root to any number, is an incrementally closer set of approximations (Continued Fraction approximations) to that number.

Notation: we represent the continued fraction as an array.

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ldots + \cfrac{1}{k}}} = [a_0; a_1, ..., a_k]$$

This representation is not unique, since $[a_0; a_1, ..., a_k] = [a_0, a_1, ..., a_k - 1, 1]$ because $\frac{1}{a_k} = \frac{1}{(a_k - 1) + \frac{1}{1}}$

Parent-Child Relations

Parent of $[a_0, a_1, ..., a_k]$ is $[a_0, a_1, ..., a_k - 1]$
Children of $[a_0, a_1, ..., a_k]$ are $[a_0, a_1, ..., a_k - 1, 2]$ and $[a_0, a_1, ..., a_k + 1]$.

5.3.2 GCD and Divisors

List of all divisors

The list of all divisors, when sorted, has pair such that $factor[i] * factor[n - i - 1] == number$.

5.4 Geometry

5.4.1 Graham Scan for Convex Hull

Chapter 6

Graphs and Tree

6.1 Trees

6.1.1 Heavy Light Decomposition

Motivating Problem

Given a Tree, handle the following queries:

- Update(edge, weight): Change the weight of any given edge in the Tree.
- Query(nodeX, nodeY): Find the heaviest eadge between two nodes, x and y.

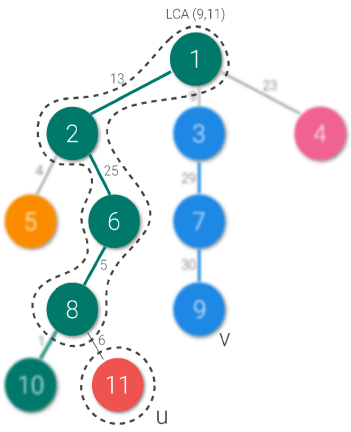

Figure 6.1: Heavy Light decomposition of a Tree

6.2 Basic Algorithms on Graphs

6.2.1 List of algorithms

- Depth First Search
- Breadth First Search
- Shortest Path - Dijkstra's
- Shortest Path - Bellman Ford
- Shortest Path - Floyd Warshall
- Connected Components
- Topological Sort
- Prim's Maximum Spanning Tree

6.2.2 Code

```
#ifndef CODE_GRAPH_H
#define CODE_GRAPH_H

#include <iostream>
#include <numeric>
#include <queue>
#include <stack>
#include <vector>
using namespace std;

typedef long long ll;
typedef vector<long long> vll;
typedef pair<long long, long long> pll;
typedef vector<pair<long long, long long>> vpl;
typedef vector<vector<long long>> mll;
typedef vector<bool> vbl;

class Graph {
   public:
    enum NodeColor { VISITED, VISITING, UNVISITED };
    struct Node {
        int index;
        vpl adjacent;
        NodeColor color = UNVISITED;
    };
    vector<Node> list;
    int n;
    Graph(int n) {
        list.resize(n);
        for (int i = 0; i < n; i++)
            list[i].index = i;
        this->n = n;
    }
    void add_edge(int u, int v, long long w = 1) {
        list[u].adjacent.emplace_back(v, w);
```

```
        list[v].adjacent.emplace_back(u, w);
    }

    pair<vll, vll> dijkstra(vll from) {
        vll dist(n, INT64_MAX), parent(n, INT32_MAX)
            ;
        priority_queue<pll, vpl, greater<>> q;
        for (auto index : from) {
            dist[index] = 0;
            q.emplace(index, 0);
        }
        while (!q.empty()) {
            pll top = q.top();
            q.pop();
            if (top.second > dist[top.first])
                continue;
            for (auto edge : list[top.first].
                adjacent) {
                if (top.second + edge.second < dist[
                    edge.first]) {
                    dist[edge.first] = top.second +
                        edge.second;
                    parent[edge.first] = top.first -
                        1;
                    q.emplace(edge.first, top.second
                        + edge.second);
                }
            }
        }
        return {dist, parent};
    }

    // Returns sorted vector of indices
    vector<int> topological_sort() {
        vector<int> in_degree(list.size(), 0),
            result;
        result.reserve(list.size());
        for (auto node : list)
            for (auto route : node.adjacent)
                in_degree[route.first - 1]++;
        queue<int> process;
        for (int i = 0; i < list.size(); i++) {
            if (in_degree[i] == 0) {
                process.push(i);
                result.push_back(i);
            }
        }
        while (!process.empty()) {
            int processing = process.front();
            process.pop();
            for (auto route : list[processing].
                adjacent) {
                in_degree[route.first - 1]--;
                if (in_degree[route.first - 1] == 0)
                    {
                    process.push(route.first - 1);
```

```
                    result.push_back(route.first -
                        1);
                }
            }
        }
        return result;
    }

    mll components() {
        vbl visited(n);
        mll result(0);
        for (int i = 0; i < n; i++) {
            if (visited[i])
                continue;
            vll component;
            stack<ll> process;
            process.push(list[i].index);
            component.push_back(i);
            visited[i] = true;
            while (!process.empty()) {
                ll processing = process.top();
                process.pop();
                for (pll neighbor : list[processing
                    ].adjacent) {
                    if (!visited[neighbor.first]) {
                        process.push(neighbor.first)
                            ;
                        component.push_back(neighbor
                            .first);
                    }
                }
            }
            result.push_back(component);
        }
        return result;
    }

    pair<vll, vll> bellman_ford(vll from) {
        vll distances(n, INT64_MAX);
        vll parent(n, INT32_MAX);
        // Bellman Ford Algorithm
        for (ll &i : from)
            distances[i] = 0;
        for (int i = 0; i < n - 1; i++) {
            for (int source = 0; source < n - 1;
                source++) {
                if (distances[source] == INT64_MAX)
                    continue;
                for (const auto &edge : list[source
                    ].adjacent) {
                    ll sink = edge.first;
                    if (distances[source] + edge.
                        second < distances[sink]) {
                        distances[sink] = distances[
                            source] + edge.second;
                        parent[sink] = source;
                    }
```

```
                }
            }
        }
        // Checking for negative cycles and putting
            -1 if it exists.
        for (ll source = 0; source < n - 1; source
            ++) {
            for (const auto &edge : list[source].
                adjacent) {
                ll sink = edge.first;
                if (distances[source] + edge.second
                    < distances[sink]) {
                    for (ll i : from)
                        distances[i] = -1;
                    return {distances, parent};
                }
            }
        }
        return {distances, parent};
    }

    vector<vector<long long>> floyd_warshall() {
        vector<vector<long long>> distances(n,
            vector<long long>(n, INT64_MAX));
        for (int i = 0; i < n; i++)
            distances[i][i] = 0;
        for (int i = 0; i < n; i++)
            for (auto route : list[i].adjacent)
                distances[i][route.first] = route.
                    second;
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (distances[i][k] == INT64_MAX
                        ||
                        distances[k][j] == INT64_MAX
                            )
                        continue;
                    distances[i][j] =
                        min(distances[i][j],
                            distances[i][k] +
                            distances[k][j]);
                }
            }
        }
        return distances;
    }

    pair<ll, vll> prims_mst() {
        priority_queue<pll, vpl, greater<>> routes;
        vll costs(n);
        vbl visited(n, false);
        for (int i = 0; i < n; i++) {
            if (!visited[i])
                routes.emplace(INT32_MAX, i);
            while (!routes.empty()) {
```

```
                pll best = routes.top();
                routes.pop();
                if (!visited[best.second])
                    costs[best.second] = best.first;
                visited[best.second] = false;
                for (const auto &path : list[best.
                    second].adjacent)
                    if (!visited[path.second])
                        routes.push(path);
            }
        }
        ll sum = accumulate(costs.begin(), costs.end
            (), 0);
        return {sum, costs};
    }
};

#endif  // CODE_GRAPH_H
```

Chapter 7

Problem Solving Strategies

7.1 General Advice and Basic Patters

- If you need to Minimize a Maximum or Maximize a Minimum, use Binary Search.
    1. https://www.codechef.com/KH19MOS/problems/ANAJOBS
- If you want to use DP / Expectation to find something optimal, use Exchange Argument.
    1. https://www.codechef.com/GWR17ROL/problems/KALADIN

7.2 List of Algorithms and Ideas we know

- 2-SAT