ACM@UCI Handbook

ICPC Templates

Ryan Yoshida

Contents

Part I
Introduction

1 About

1.1 Purpose

The ACM@UCI Handbook is a printable resource intended for use in contests such as ICPC. It provides data structure templates and code snippets for quick and reliable implementation. Priority is placed on data structures and algorithms which meet one or several of the following criteria: commonly seen in contests, difficult/tedious to remember/implement, and/or too niche to find in other competitive programming handbooks.

This handbook is not intended to be used as a teaching resource. Explanations of underlying theory and concepts for the provided data structures and algorithms will be provided only insofar as it is expedient to their implementation in contest. To make the best use of this handbook you should first have a strong understanding the purpose and theory behind every data structure and algorithm.

1.2 Organization

Part II

Data Structures

2 Graphs

2.1 Link-Cut Tree

2.2 Splay Tree

2.3 Union Find

3 Range Queries

3.1 Segment Tree

3.1.1 Point Update

3.1.2 Lazy Propagation

3.2 Square Root Decomposition

3.3 Sparse Table

Sparse tables are data structures which answer range queries in $O(1)$. Its construction takes $O(N \log N)$ and it requires $O(N \log N)$ space. It only works if the operation is idempotent (i.e. for operation $*$: $a * a = a$). Examples of idempotent functions include *min*, *gcd*.

The provided C++ implementation takes an IdempotentFunc as a template type. Pass a type with an implemented functor for this.

```
#include <vector>

template <typename T, typename IdempotentFunc>
class SparseTable {
private:
    using F = IdempotentFunc;

    int N,K;
    std::vector<std::vector<T>> st;
    std::vector<int> log;

public:
    SparseTable(const std::vector<T> &arr)
    : N{static_cast<int>(arr.size())}, K{0} {
        while ((1<<K) < N) { ++K; }

        // initialize sparse table
        st.assign(N, std::vector<T>(K+1));
        for (int i = 0; i < N; ++i) { st[i][0] = arr[i]; }

        // fill sparse table
        for (int j = 1; j <= K; ++j)
            for (int i = 0; i + (1<<j) <= N; ++i)
                st[i][j] = F{}(st[i][j-1], st[i+(1<<(j-1))][j-1]);

        // Initialize log mappings
        log.assign(N+1, 0);
        for (int i = 2; i <= N; ++i) log[i] = log[i/2] + 1;
    }
    // Applies IdempotentFunc on all elements in range [l,r)
    // where l is inclusive and r is exclusive
    T query(int l, int r) {
        int j = log[r-l];
        return F{}(st[l][j], st[r-(1<<j)][j]);
    }
};
```

3.4 Wavelet Tree

4 String Processing

4.1 Aho Corasick

Aho Corasick creats a finite automata on a prefix trie (4.4) to match occurences of a set of strings in a search string. Equivalently, it can be seen as an implementation of Knuth Morris Pratt (10.1) for a set of strings rather than one pattern string. It works the same as KMP by creating pointers to the longest proper suffix that is a prefix for each index of each string in the set. Construction of the Aho Corasic Trie is $O(M)$ where M is the sum of character counts of all pattern strings. Iteration through the search string is worst case $O(NK)$ where N is the length of the search string and K is the number of pattern strings. In most cases however, it is safe to assume that iteration through the search string is $O(N)$.

Implementation: The Aho Corasick Trie is encapsulated in a class. Due to the nature of the Aho Corasick automata construction algorithm, an `Aho` object must be constructed with a vector of pattern strings `dict`. To traverse the Aho Corasick Trie, a helper class `Aho::AhoAutomata` is provided. The `AhoAutomata` represents a state (node) in the Aho Corasick Trie. An iterator is provided for the `AhoAutomata` to iterate through the *ids* of all pattern strings in `dict` that are terminated at that node (the *id* of a pattern string is its index in `dict`). Note that the `Node` struct stores the child pointers as an `std::array` of length 256. This can be adjusted based on the problem.

```
#include <algorithm>

#include <vector>
#include <array>
#include <queue>
#include <string>

class Aho {
private:
    struct Node {
        Node *suffix;   // longest prefix that is suffix
        Node *output;   // longest word that is a suffix
        std::vector<int> wordids;   // ids of all words terminated at node
                                    // vector to handle duplicates
        std::array<Node *, 256> c;  // links to children of node

        Node()
        : suffix{nullptr}, output{nullptr}, wordids{std::vector<int>()} {
            std::fill(std::begin(c), std::end(c), nullptr);
        }
    };

private:
    Node *root;

public:
    class AhoAutomata {
    private:
        Aho::Node *root, *curr;
    public:
        AhoAutomata(Aho::Node *root) : root{root}, curr{root} { }

        // Advances to the next character, follows suffix link on failure
        void next(char c) {
            Aho::Node *tmp = curr;
            while (tmp->c[c] == nullptr && tmp != root) { tmp = tmp->suffix; }
            if (tmp->c[c] != nullptr) { curr = tmp->c[c]; }
            else { curr = root; }
        }

        class Iterator {
        private:
            Node *outputnode;
            int index;
        public:
            Iterator(Node *outputnode) : outputnode{outputnode}, index{0} {
                if (this->outputnode != nullptr
                        && this->outputnode->wordids.empty()) {
                    this->outputnode = outputnode->output;
                }
            }

            int operator*() { return outputnode->wordids[index]; }
            Iterator &operator++() {
                if (outputnode != nullptr) {
                    if (++index >= (int) outputnode->wordids.size()) {
                        outputnode = outputnode->output;
                        index = 0;
                    }
                }
                return *this;
            }

            bool operator==(const Iterator &other) {
                return outputnode == other.outputnode && index == other.index;
            }
            bool operator!=(const Iterator &other) {
                return outputnode != other.outputnode || index != other.index;
            }
        };

        AhoAutomata::Iterator begin() { return AhoAutomata::Iterator(curr); }
        AhoAutomata::Iterator end() { return AhoAutomata::Iterator(nullptr); }
    };
```

```
private:
    // Perform BFS to construct suffix and output links
    void computeLinks() {
        root->suffix = root;    // suffix link of root is itself

        std::queue<Node *> q;
        for (Node *next : root->c) if (next != nullptr) {
            next->suffix = root;    // edge case
            q.push(next);
        }

        while (!q.empty()) {
            Node *curr = q.front(); q.pop();
            for (int c = 0; c < 256; ++c) if (curr->c[c] != nullptr) {
                Node *suffixofcurr = curr->suffix;
                while (suffixofcurr->c[c] == nullptr && suffixofcurr != root)
                    suffixofcurr = suffixofcurr->suffix;

                if (suffixofcurr->c[c] != nullptr)
                    curr->c[c]->suffix = suffixofcurr->c[c];
                else
                    curr->c[c]->suffix = root;

                q.push(curr->c[c]);
            }
            if (!(curr->suffix->wordids.empty()))
                curr->output = curr->suffix;
            else
                curr->output = curr->suffix->output;
        }
    }

public:
    Aho(const std::vector<std::string> &dict) : root{new Node()} {
        for (int i = 0; i < static_cast<int>(dict.size()); ++i) {
            Node *curr = root;
            for (const char &c : dict[i]) {
                if (curr->c[c] == nullptr) { curr->c[c] = new Node(); }
                curr = curr->c[c];
            }
            curr->wordids.push_back(i);
        }

        computeLinks();
    }

    ~Aho() {
        std::queue<Node *> q;
        q.push(root);
        while (!q.empty()) {
            Node *curr = q.front(); q.pop();
            for (int c = 0; c < 256; ++c) if (curr->c[c] != nullptr) {
                q.push(curr->c[c]);
            }
            delete curr;
        }
    }

    AhoAutomata getAutomata() { return AhoAutomata(root); }

};
```

Example Usage: An example of how to use the Aho class is provided below.

```
// s is the search string, p is the pattern string set
void aho_example(const std::string &s, const std::vector<std::string> &p) {
    Aho aho(p);
    Aho::AhoAutomata automata = aho.getAutomata();

    for (int i = 0; i < (int) s.length(); ++i) {
        automata.next(s[i]);
        for (const int &id : automata) {
            std::cout << p[id] << "␣terminates␣at␣index␣" << i << std::endl;
        }
    }

    return 0;
}
```

4.2 EerTreE

EerTreE or Palindromic Tree is a data structuring for finding and tracking palindromes in a string. It is similar to the Aho Corasick tree as it uses suffix links to the longest proper suffix that is a palindrome. This implementation provides an iterator that, at each character, iterates through the lengths of all palindromes ending at that character. Construction of the EerTreE is $O(N)$.

```
#include <array>
#include <string>

class EerTreE {
private:
    struct Node {
        Node *suf;
        std::array<Node *, 256> lab;
        int len;
        Node(int len) : suf{nullptr}, len{len} {
            std::fill(std::begin(lab), std::end(lab), nullptr);
        }
    };

    void deconstruct(Node *curr) {
        for (Node *child : curr->lab) if (child != nullptr)
            deconstruct(child);
        delete curr;
    }
```

```
public:
    Node *root1, *root2, *current;
    int i, cnt; // cnt is number of unique palindromes
    std::string s;

    EerTreE() : i{0}, cnt{0}, s{""} {
        root1 = new Node(-1);
        root1->suf = root1;
        root2 = new Node(0);
        root2->suf = root1;
        current = root2;
    }

    ~EerTreE() {
        deconstruct(root1);
        deconstruct(root2);
    }

    void append(char c) {
        s += c;
        Node *cur = current;
        while (i - 1 - cur->len < 0 || s[i-1-cur->len] != c)
            cur = cur->suf;

        if (cur->lab[c] == nullptr) {
            ++cnt;
            Node *tmp = new Node(cur->len + 2);
            cur->lab[c] = tmp;

            if (tmp->len == 1) {
                tmp->suf = root2;
                current = tmp;
            } else {
                cur = cur->suf;
                while (i - 1 - cur->len < 0 || s[i-1-cur->len] != c)
                    cur = cur->suf;
                tmp->suf = cur->lab[c];
                current = tmp;
            }
        } else {
            current = cur->lab[c];
        }

        ++i;
    }

    class Iterator {
    private:
        Node *cur;
    public:
        Iterator(Node *curnode) : cur{curnode} { }
        int operator*() { return cur->len; }
        Iterator &operator++() { cur = cur->suf; return *this; }
        bool operator==(const Iterator &other) { return cur == other.cur; }
        bool operator!=(const Iterator &other) { return cur != other.cur; }
    };

    EerTreE::Iterator begin() { return Iterator(current); }
    EerTreE::Iterator end() { return Iterator(root1); }
};
```

4.3 Suffix Array

A suffix array sorts the suffixes of a string in lexicographical order. The below provided implementation constructs a suffix array in $O(N \log N)$. Also included in this implementation is the construction of the longest common prefix array which gives the longest common prefix between suffixes adjacent in the suffix array. The `LCP` array is constructed in $O(N)$.

```
#include <ostream>
#include <algorithm>
#include <vector>
#include <numeric>

template <typename T=char, T term='$'>
class SuffixArray {
public:
    class LCP;

private:
    size_t size;
    std::vector<T> str;
    std::vector<int> sa;
    std::vector<int> ra;

public:
    template <typename InputIt>
    SuffixArray(InputIt first, InputIt last) : size{0} {
        for (InputIt cur = first; cur != last; ++cur, ++size);
        ++size; // determine size
        str.assign(size, T{}); sa.assign(size, 0); ra.assign(size, 0);
        std::copy(first, last, std::begin(str)); str[size-1] = term;
        std::iota(std::begin(sa), std::end(sa), 0);
        std::transform(std::begin(str), std::end(str), std::begin(ra),
                [](const T &v) ->int { return static_cast<int>(v); });

        std::stable_sort(std::begin(sa), std::end(sa),
                [&](const int &a, const int &b)->bool {
                    return ra[a] < ra[b];
                });
        rerank(0); // allows the handling of negative values
        for (int k = 0; (1<<k) <= static_cast<int>(size); ++k) {
            for (int i = 0; i < static_cast<int>(size); ++i) {
                sa[i] = (sa[i] - (1<<k) + ((int) size)) % ((int) size);
            }
            countSort();
            rerank(1<<k);

            if (ra[sa[size-1]] == static_cast<int>(size)-1) break;
        }
    }

private:
    void countSort() {
        std::vector<int> cnt(size, 0);
        for (const int rank : ra) { ++cnt[rank]; }
        int sum = 0; for (int &c : cnt) { int tmp = c; c = sum; sum += tmp; }

        std::vector<int> sa_(size, 0);
        for (const int cur : sa) { sa_[cnt[ra[cur]]++] = cur; }
        sa = sa_;
    }

    void rerank(int k) {
        std::vector<int> ra_(size, 0);
        int r = ra_[sa[0]] = 0;
        for (int i = 1; i < static_cast<int>(size); ++i) {
            int newL=ra[sa[i]],     newR=ra[(sa[i]+k)%size];
            int preL=ra[sa[i-1]],   preR=ra[(sa[i-1]+k)%size];
            if (newL != preL || newR != preR) ++r;
            ra_[sa[i]] = r;
        }
        ra = ra_;
    }

public:
    size_t getSize() const { return size; }
    int getRank(int suffix) const { return ra[suffix]; }
    int operator[] (const int &i) const { return sa[i]; }

    SuffixArray<T,term>::LCP getLCP() const {
        return SuffixArray<T,term>::LCP(*this);
    }

    class LCP {
    private:
        size_t size;
        std::vector<int> lcp;
    public:
        LCP(const SuffixArray &suf_arr)
        : size{std::max(size_t{},suf_arr.getSize())},
                lcp{std::vector<int>(size, 0)} {
            int curMatch = 0;
            for (int i = 0; i < static_cast<int>(suf_arr.getSize())-1; ++i) {
                int curSuffixRank = suf_arr.getRank(i);
                if (curSuffixRank == 0) {
                    curMatch = lcp[curSuffixRank] = 0; continue;
                }
                int suffixAbove = suf_arr[curSuffixRank - 1];

                while (suf_arr.str[i+curMatch] ==
                        suf_arr.str[suffixAbove+curMatch])
                    ++curMatch;
                lcp[curSuffixRank] = curMatch;
                curMatch = std::max(0, curMatch-1);
            }
        }

        size_t getSize() const { return size; }
        int operator[](const int &i) const { return lcp[i]; }
    };

};
```

4.4 Trie

A Trie (short for reTRIEval tree), or Prefix Tree, stores a dictionary of strings in a manner that makes lookup by their prefix very efficient. Construction is $O(N)$ where N is the sum of character counts of all dictionary strings. Lookup of a word or prefix is $O(M)$ where M is the length of the query string.

```
#include <array>
#include <string>
class Trie {
private:
    class Node {
    public:
        bool isWord;
        std::array<Node *, 256> c;
        Node() : isWord{false} {
            std::fill(std::begin(c), std::end(c), nullptr);
        }
    };

    Node *root;

    void deconstruct(Node *curr) {
        for (Node *child : curr->c) if (child != nullptr) deconstruct(child);
        delete curr;
    }
public:
    Trie() : root{new Node()} { }

    ~Trie() {
        deconstruct(root);
    }

    void insert(const std::string &word) {
        Node *curr = root;
        for (char c : word) {
            curr = curr->c[c] == nullptr ?
                (curr->c[c] = new Node()) : curr->c[c];
        }
        curr->isWord = true;
    }
    bool search(const std::string &word) const {
        Node *curr = root;
        for (char c : word)
            if (curr->c[c] == nullptr) return false;
            else curr = curr->c[c];
        return curr->isWord;
    }

    bool startsWith(string prefix) const {
        Node *curr = root;
        for (char c : prefix)
            if (curr->c[c] == nullptr) return false;
            else curr = curr->c[c];
        return true;
    }
};
```

Part III
Algorithms

5 Dynamic Programming

5.1 Sum over Subsets

Given an array `A`, sum over subsets fills array `S` such that `S[mask]` is the sum of all values at indices `s` where $s \subseteq mask$ in `A`. This algorithm is $O(N2^N)$

```
// Initialize S array
for (int i = 0; i < (1<<N); ++i)
    S[i] = A[i];

for (int i = 0; i < N; ++i) for (int mask = 0; mask < (1<<N); ++mask)
    if (mask & (1<<i))
        S[mask] += S[mask^(1<<i)];
```

```
from copy import copy

S = copy(A)
for i in range(N):
    for mask in range(1<<N):
        if mask & (1<<i):
            S[mask] += S[mask^(1<<i)]
```

6 Geometry

6.1 Definitions

This section will begin with some geometric object definitions. For clarity, all references made to these class definitions will use the `geo` namespace. Note that some problems benefit from alternative definitions than that given here.

6.1.1 2D Definitions

```
template <typename T> struct point2d {
    T x,y;
    point2d(T x=T{}, T y=T{}) : x{x}, y{y} { }
};
```

```
template <typename T> struct line2d {
    T a,b,c; // ax + by + c = 0
    line2d(T a, T b, T c) : a{a}, b{b}, c{c} { }

    // This constructor only works for T=double
    line2d(const geo::point2d<T> &a, const geo::point2d<T> &b) {
        if (std::abs(a.x - b.x) < 0.000001) { // be careful with floating
            points
            // This is a vertical line
            this->a = T{1}; this->b = T{0}; this->c = T{-a.x};
        } else {
            this->a = - (a.y - b.y) / (a.x - b.x);
            this->b = T{1};
            this->c = - (this->a * a.x) - a.y;
        }
    }
};
```

6.1.2 3D Definitions

```
template <typename T> struct point3d {
    T x,y,z;
    point3d(T x=T{}, T y=T{}, T z=T{}) : x{x}, y{y}, z{z} { }
};
```

6.2 Closest Points

6.3 Convex Hull

```
#include <cstdio>
#include <iostream>
#include <vector>
#include <algorithm>


namespace ConvexHull {

        template <typename T>
        struct Point {
                T x,y;

                bool operator<(const Point<T>& other) const {
                        return (x == other.x) ? y < other.y : x < other.x;
                }

                bool operator==(const Point<T>& other) const {
                        return x == other.x && y == other.y;
                }
        };

        template <typename T>
        struct Vector {
                T x,y;
        };

        template <typename T>
        T isccw(Vector<T>& a, Vector<T>& b) {
                return a.x*b.y - a.y*b.x;
        }

        template <typename T>
        void buildHull(std::vector<Point<T>>& hull, std::vector<Point<T>>&
            points, bool ccw) {
                for (Point<T>& p : points) {
                        while (hull.size() >= 2) {
                                Point<T>& a = hull[hull.size() - 2];
                                Point<T>& b = hull[hull.size() - 1];
                                Point<T>& c = p;
                                Vector<T> v1 = Vector<T>{b.x-a.x, b.y-a.y};
                                Vector<T> v2 = Vector<T>{c.x-b.x, c.y-b.y};
                                if (ccw && isccw<T>(v1, v2) > 0) {
                                        break;
                                } else if (!ccw && isccw<T>(v1, v2) < 0) {
                                        break;
                                }

                                hull.pop_back();
                        }

                        while (!hull.empty() && p == hull.back()) {
                                hull.pop_back();
                        }

                        hull.push_back(p);
                }
        }

        template <typename T>
        void buildConvexHull(std::vector<Point<T>>& points, std::vector<Point<
           T>>& convex_hull) {
                std::sort(points.begin(), points.end());
                std::vector<Point<T>> lower_hull, upper_hull;
                buildHull<T>(lower_hull, points, true);
                buildHull<T>(upper_hull, points, false);

                convex_hull.assign(lower_hull.size() + std::max(0, (int)
                   upper_hull.size()-2), Point<T>{0,0});
                int j = 0;
                for (int i = 0; i < lower_hull.size(); ++i,++j) {
                        convex_hull[j] = lower_hull[i];
                }
                for (int i = upper_hull.size()-2; i > 0; --i,++j) {
                        convex_hull[j] = upper_hull[i];
                }
        }
}

int main() {

        int N;
        while (scanf("%d", &N), N) {
                std::vector<ConvexHull::Point<int>> points(N);
                std::vector<ConvexHull::Point<int>> convex_hull;
                for (int i = 0; i < N; ++i) {
                        int x, y; scanf("%d %d", &x, &y);
                        points[i] = ConvexHull::Point<int>{x, y};
                }

                ConvexHull::buildConvexHull<int>(points, convex_hull);

                printf("%d\n", convex_hull.size());
                for (const ConvexHull::Point<int>& p : convex_hull) {
                        printf("%d %d\n", p.x, p.y);
                }
        }

        return 0;
}
```

7 Graphs

7.1 Bellman-Ford's Algorithm: Single Source Shortest Path

```
#include <vector>

// ========== FOR THE USER TO DEFINE ========== //
typedef int WeightType;                                 // by default all edge
    weight computations will use integers
const WeightType INF= 1e9;                // by default a path of infinite cost
   will be 1e9
const WeightType NEG_INF = -1e9;          // by default a negative weight of
   infinite value is -1e9
// ============================================ //



// GRAPH ABSTRACTION =========================
struct Edge {
        int v;  // the destination vertice
        WeightType w;   // the weight of the edge
};

typedef std::vector<Edge> EdgeList;
typedef std::vector<EdgeList> AdjList;  // Type for adjacency lists.
                                                                              //

                                                                                 If

                                                                                 you

                                                                                 have

                                                                                 graph

                                                                                 g

                                                                                 represe

                                                                                 as

                                                                                 an

                                                                                 adjacen

                                                                                 list
                                                                                 ,
g
[
u
]

=

list

of

all

edges

with

source

of

u

typedef std::vector<WeightType> Costs;  // the return type of the bellmanFord
   () function, defines SSSP between source and all nodes
// END GRAPH ABSTRACTION =====================


// From a single source returns a Costs vector containing the cost of the
   shortest path to all nodes. If unreachable will be INF.
// If reachable through negative cycle will be NEG_INF.
Costs bellmanFord(AdjList& adj, int source) {

        Costs dists(adj.size(), INF);
        dists[source] = 0;      // SSSP to source is 0
        for (int i = 0; i < adj.size()-1; ++i) {
                for (int u = 0; u < adj.size(); ++u) {
                        for (int j = 0; j < adj[u].size(); ++j) {
                                Edge& e = adj[u][j];
                                if (dists[u] != INF)
                                        dists[e.v] = std::min(dists[e.v],
                                           dists[u] + e.w);
                        }
                }
        }

        // Check for Negative Cycles
        for (int i = 0; i < adj.size(); ++i) {  // This outer loop is required
            to determine which nodes have a -INF path
                                                                                                    //
                for (int u = 0; u < adj.size(); ++u) {
                        for (int j = 0; j < adj[u].size(); ++j) {
                                Edge& e = adj[u][j];
                                if (dists[u] != INF && dists[e.v] > dists[u] +
                                        e.w) {
                                            dists[e.v] = NEG_INF;
                                   }
                          }
                 }
        }

        return dists;
}
```

7.2 Dijkstra's Algorithm: Single Source Shortest Path

7.3 Floyd-Warshall's Algorithm: All Pairs Shortest Path

Floyd-Warshall's Algorithm computes the shortest path between every pair of nodes in a graph. Acting on an adjacency matrix `adj` where `adj[u][v]` is the weight of an edge from `u` to `v`, it fills the matrix so that `adj[u][v]` contains the cost of the shortest path between vertices `u` and `v`. This is done in $O(N^3)$.

For implementation ensure that the adjacency matrix is filled in properly. Be sure to consider the case of self-loops: should they be initialized to `0`? Be sure to consider double edges (more than one edge between two vertices): should you take the minimum? Are there negative edge weights? This may result in negative cycles. A negative cycle will result in `adj[u][u] < 0` after Floyd Warshall's is complete.

```
// Let adj be a NxN matrix where adj[u][v] is initially the
//  weight of edge u -> v
//  adj[u][v] will be INFTY if no edge exists
for (int k = 0; k < N; ++k) {
    for (int u = 0; u < N; ++u) {
        for (int v = 0; v < N; ++v) {
            if (adj[u][k] != INFTY && adj[k][v] != INFTY) {
                adj[u][v] = std::min(adj[u][v], adj[u][k]+adj[k][v]);
            }
        }
    }
}
// At this point adj[u][v] will be the cost of the shortest
//  path from u to v
//  If no path exists it will be INFTY
```

```
# Let adj be a NxN matrix where adj[u][v] is initially the
#  weight of edge u -> v
#  adj[u][v] will be INFTY if no edge exists
for k in range(N):
    for u in range(N):
        for v in range(N):
            if adj[u][k] != INFTY and adj[k][v] != INFTY:
                adj[u][v] = min(adj[u][v], adj[u][k]+adj[k][v])
# At this point adj[u][v] will be the cost of the shortest
#  path from u to v
#  If no path exists it will be INFTY
```

7.4 Minimum Spanning Tree

7.4.1 Kruskall's Algorithm

7.4.2 Prim's Algorithm

7.5 Network Flow

7.5.1 Edmonds Karp's Algorithm

7.5.2 Dinic's Algorithms

```
// As of right now this solves https://open.kattis.com/problems/maxflow
#include <bits/stdc++.h>

typedef long long LL;

struct Edge {
        int u, v, v_index, residual_index;
        LL c;
        LL init;
};

typedef std::vector<Edge> EdgeList;
typedef std::vector<EdgeList> AdjList;
typedef std::vector<int> LevelGraph;
typedef std::vector<int> Pruner;

const LL INF = std::numeric_limits<LL>::max();

int N, M, S, T;
AdjList adj;
LevelGraph levels;
Pruner next;            // use to prune the DFS, holds the next edge to search
    in DFS
LL maxFlow;

void augment(Edge& e, int flow) {
        adj[e.u][e.v_index].c -= flow;
        adj[e.v][e.residual_index].c += flow;
}

LL dfs(int u, LL flow) {
        if (u == T) return flow;

        int numEdges = adj[u].size();
        for (; next[u] < numEdges; ++next[u]) {
                Edge& e = adj[u][next[u]];

                if (e.c > 0 && levels[u]+1 == levels[e.v]) {    // there is
                    flow and goes to next level
                        LL bottleNeck = dfs(e.v, std::min(flow, e.c));
                        if (bottleNeck > 0) {
                                // augment the edge
                                augment(e, bottleNeck);
                                return bottleNeck;
                        }
                }
        }

        return 0;
}


// Perform a BFS to build level graph and determine whether blocking flow
   reached
bool bfs() {
        levels.assign(N, -1);

        std::queue<int> q;
        q.push(S);      // start at the source
        levels[S] = 0;
        while (!q.empty()) {
                int u = q.front(); q.pop();

                for (const Edge& e : adj[u]) {
                        if (e.c > 0 && levels[e.v] == -1) {
                                levels[e.v] = levels[u] + 1;
                                q.push(e.v);
                        }
                }
        }

        return levels[T] != -1;
}


void dinics() {

        maxFlow = 0;
        while (bfs()) { // loop while there is not a blocking flow
                next.assign(N, 0);

                // dfs from source to sink until no more flow
                for (LL f = dfs(S, INF); f != 0; f = dfs(S, INF)) {
                        maxFlow += f;
                }
        }
}

int main() {

        scanf("%d␣%d␣%d␣%d", &N, &M, &S, &T);
        adj.assign(N, EdgeList());
        for (int i = 0; i < M; ++i) {
                int u, v; LL c; scanf("%d␣%d␣%lld", &u, &v, &c);
                if (u == v) continue;

                int v_index = adj[u].size();
                int residual_index = adj[v].size();
                adj[u].push_back(Edge{u,v,v_index, residual_index,c,c});
                adj[v].push_back(Edge{v,u,residual_index,v_index,0,0});
        }

        dinics();

        EdgeList ans;
        for (int i = 0; i < N; ++i) {
                for (Edge e : adj[i]) {
                        if (e.init - e.c > 0) {
                                ans.push_back(e);
                        }
                }
        }
        int edgeCount = ans.size();

        printf("%d␣%lld␣%d\n", N, maxFlow, edgeCount);
        for (Edge& e : ans) {
                printf("%d␣%d␣%lld\n", e.u, e.v, e.init - e.c);
        }

        return 0;
}
```

7.5.3 Hopcroft Karp's Algorithm

7.5.4 Min-Cost Max Flow

7.6 Tarjan's Algorithm

7.6.1 Bridges & Articulation Points

7.6.2 Strongly Connected Components

7.7 Trees

7.7.1 Binary Lift

7.7.2 Lowest Common Ancestor

8 Math

8.1 Arithmetic Sum

Computes an arithmetic sum constant time. The function was intentionally designed to have a similar function signature to Python's `range` function. Thus, `arithmetic_sum(start,end,step)` will yield the same result as `sum(range(start,end,step))` in Python, but it will do so in $O(1)$. For this function to work, $start < end$ and $step > 0$.

Important: `start` is inclusive, `end` is exclusive.

```
int arithmetic_sum(int start int end, int step=1) {
    int n = 1+(end-start-1)/step;
    if (n%2) return (start + ((n-1)>>1)*step)*n;
    else return (2*start + (n-1)*step)*(n>>1);
}
```

```
def arithmetic_sum(start: int, end: int, step: int=1):
    n = 1+(end-start-1)//step;
    if n%2:
        return (start + ((n-1)>>1)*step)*n
    else:
        return (2*start + (n-1)*step)*(n>>1)
```

8.2 Chinese Remainder Theorem

8.3 Division Over Modulo

Computes $\frac{a}{b}$ mod mod by multiplying `a` to the inverse mod of `b`. Uses the power over modulo function described in 8.9. This function only works if `b` and `mod` are relatively coprime (i.e. $gcd(b, mod) = 1$). Can be used to compute inverse modulo by setting `a=1`. The time complexity of this algorithm is $O(\log_2 mod)$.

```
int divide_modulo(int a, int b, int mod) {
    return (a * (pow(b, mod-2, mod) % mod)) % mod;
}
```

```
def divide_modulo(a: int, b: int, mod: int):
    return (a * (pow(b, mod-2, mod) % mod)) % mod
```

8.4 Fast Fourier Transform

8.5 Geometric Sum

8.5.1 Finite Bounds

Returns $\sum_{k=start}^{end-1} a^k = \frac{a^{start}-a^{end}}{1-a}$.

```
double geometric_sum(int start, int end, double a) {
    return (std::pow(a,start) - std::pow(a,end))/(1.0-a);
}
```

```
def geometric_sum(start: int, end: int, a: float):
    return (pow(a,start) - pow(a,end))/(1-a)
```

8.5.2 Finite Start, Infinite End

Returns $\sum_{k=start}^{\infty} a^k = \frac{a^{start}}{1-a}$.

```
double geometric_sum(int start, double a) {
    return (std::pow(a,start))/(1.0-a);
}
```

```
def geometric_sum(start: int, a: float):
    return pow(a, start)/(1-a)
```

8.5.3 Infinite Scaled Sum

Returns $\sum_{k=0}^{\infty} ka^k = \frac{a}{(1-a)^2}$

```
double geometric_sum(double a) {
    return a/((1-a)*(1-a));
}
```

```
def geometric_sum(a: float):
    return a/((1-a)*(1-a))
```

8.6 Greatest Common Divisor

The algorithm for finding greatest common divisor is Euclid's algorithm. The time complexity of this algorithm is $O(\log_{10} max(a,b))$.

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

```
def gcd(a: int, b: int):
    return a if b == 0 else gcd(b, a%b)
```

8.7 Lowest Common Multiple

The algorithm for finding lowest common multiple uses Euclid's algorithm for greatest common divisor (8.6). The time complexity of this algorithm is $O(\log_{10} max(a,b))$.

```
int lcm(int a, int b) { return a * (b / gcd(a,b)); }
```

```
def lcm(a: int, b: int):
    return a * (b // gcd(a,b))
```

8.8 Modulo

A safe modulo for use with negative numbers. Use for languages like C++ when `a` can be negative.

```
int modulo(int a, int b) {
    const int res = a%b;
    return res >= 0 ? res : res + b;
}
```

8.9 Power Over Modulo

Returns $b^k \mod mod$. This implementation cannot handle $b < 0$ in C++. To handle this case, use the safe modulo described in section 8.8. Note that no Python implementation is provided because it is already implemented in the standard library function `pow(a,b[,mod])`. The time complexity of this algorithm is $O(\log_2(k))$.

```
int pow(int b, int k, int mod) {
    if (k == 0) return 1 % mod;
    else if (k == 1) return b % mod;
    else {
        int res = pow(b, k/2, mod);
        return (((res*res)%mod) * ((k%2 ? b : 1)%mod))%mod;
    }
}
```

8.10 Sum of Squares

Returns $\sum_{k=0}^{end} k^2 = \frac{n(n+1)(2n+1)}{6}$

```
int sum_of_squares(int end) { return (end*(end+1)*(2*end+1))/6; }
```

```
def sum_of_squares(end: int):
    return (end*(end+1)*(end+2))//6
```

8.11 Sum of Cubes

Returns $\sum_{k=0}^{end} k^3 = \frac{n^2(n+1)^2}{4}$

```
int sum_of_cubes(int end) { return (end*end*(end+1)*(end+1))/4; }
```

```
def sum_of_cubes(end: int):
    return (end*end*(end+1)*(end+1))//4
```

9 Miscellaneous

9.1 G++ Compiler Args

```
g++ $(CFLAGS) -o $(TARGET) $(SRC) $(LDFLAGS)

# CFLAGS: c++ version (e.g. -std=c++11)
# TARGET: target executable (e.g. a.out)
# SRC: source files (e.g. main.cpp)
# LDFLAGS: additional compiler flags
#   - useful flags incude:
#       > -g (gdb)
#       > -Wall (catch all warnings)
#       > -fsanitize=address (display more information on segfault)
#       > -Werror (make all warnings into errors)
#       > -pedantic (issue warnings demanded by strict ISO C++ rules)
#       > -O2 (optimize code, longer compile time)
```

9.2 GNU Policy Based Data Structures

```
// 5 lines
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
template <class T> using ordered_set = tree<T, null_type, less<T>, rb_tree_tag
   , tree_order_statistics_node_update>;
template <class K, class V> using ordered_map = tree<K, V, less<K>,
   rb_tree_tag, tree_order_statistics_node_update>;
```

```
// Initialize with:
//      ordered_set<T> os;
//
// Functionality
//      os.find_by_order()
//      os.order_of_key()
```

9.3 Gray Code

```
int g(int n) { return n ^ (n>>1); }
```

9.3.1 Inverse Gray Code

```
int rev_g(int g) {
    int n = 0;
    for (; g; g >>= 1) { n ^= g; }
    return n;
}
```

9.4 Large Primes

```
492366587, 1000000007, 2147483647, 63018038201, 99999199999, 1746860020068409
```

9.5 Prime Numbers Up To 5000

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113
,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197
,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281
,283,293,307,311,313,317,331,337,347,349,353,359,367,373,379
,383,389,397,401,409,419,421,431,433,439,443,449,457,461,463
,467,479,487,491,499,503,509,521,523,541,547,557,563,569,571
,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659
,661,673,677,683,691,701,709,719,727,733,739,743,751,757,761
,769,773,787,797,809,811,821,823,827,829,839,853,857,859,863
,877,881,883,887,907,911,919,929,937,941,947,953,967,971,977
,983,991,997,1009,1013,1019,1021,1031,1033,1039,1049,1051,1061,1063,1069
,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,1163,1171,1181,1187
,1193,1201,1213,1217,1223,1229,1231,1237,1249,1259,1277,1279,1283,1289,1291
,1297,1301,1303,1307,1319,1321,1327,1361,1367,1373,1381,1399,1409,1423,1427
,1429,1433,1439,1447,1451,1453,1459,1471,1481,1483,1487,1489,1493,1499,1511
,1523,1531,1543,1549,1553,1559,1567,1571,1579,1583,1597,1601,1607,1609,1613
,1619,1621,1627,1637,1657,1663,1667,1669,1693,1697,1699,1709,1721,1723,1733
,1741,1747,1753,1759,1777,1783,1787,1789,1801,1811,1823,1831,1847,1861,1867
,1871,1873,1877,1879,1889,1901,1907,1913,1931,1933,1949,1951,1973,1979,1987
,1993,1997,1999,2003,2011,2017,2027,2029,2039,2053,2063,2069,2081,2083,2087
,2089,2099,2111,2113,2129,2131,2137,2141,2143,2153,2161,2179,2203,2207,2213
,2221,2237,2239,2243,2251,2267,2269,2273,2281,2287,2293,2297,2309,2311,2333
,2339,2341,2347,2351,2357,2371,2377,2381,2383,2389,2393,2399,2411,2417,2423
,2437,2441,2447,2459,2467,2473,2477,2503,2521,2531,2539,2543,2549,2551,2557
,2579,2591,2593,2609,2617,2621,2633,2647,2657,2659,2663,2671,2677,2683,2687
,2689,2693,2699,2707,2711,2713,2719,2729,2731,2741,2749,2753,2767,2777,2789
,2791,2797,2801,2803,2819,2833,2837,2843,2851,2857,2861,2879,2887,2897,2903
```

```
,2909,2917,2927,2939,2953,2957,2963,2969,2971,2999,3001,3011,3019,3023,3037
,3041,3049,3061,3067,3079,3083,3089,3109,3119,3121,3137,3163,3167,3169,3181
,3187,3191,3203,3209,3217,3221,3229,3251,3253,3257,3259,3271,3299,3301,3307
,3313,3319,3323,3329,3331,3343,3347,3359,3361,3371,3373,3389,3391,3407,3413
,3433,3449,3457,3461,3463,3467,3469,3491,3499,3511,3517,3527,3529,3533,3539
,3541,3547,3557,3559,3571,3581,3583,3593,3607,3613,3617,3623,3631,3637,3643
,3659,3671,3673,3677,3691,3697,3701,3709,3719,3727,3733,3739,3761,3767,3769
,3779,3793,3797,3803,3821,3823,3833,3847,3851,3853,3863,3877,3881,3889,3907
,3911,3917,3919,3923,3929,3931,3943,3947,3967,3989,4001,4003,4007,4013,4019
,4021,4027,4049,4051,4057,4073,4079,4091,4093,4099,4111,4127,4129,4133,4139
,4153,4157,4159,4177,4201,4211,4217,4219,4229,4231,4241,4243,4253,4259,4261
,4271,4273,4283,4289,4297,4327,4337,4339,4349,4357,4363,4373,4391,4397,4409
,4421,4423,4441,4447,4451,4457,4463,4481,4483,4493,4507,4513,4517,4519,4523
,4547,4549,4561,4567,4583,4591,4597,4603,4621,4637,4639,4643,4649,4651,4657
,4663,4673,4679,4691,4703,4721,4723,4729,4733,4751,4759,4783,4787,4789,4793
,4799,4801,4813,4817,4831,4861,4871,4877,4889,4903,4909,4919,4931,4933,4937
,4943,4951,4957,4967,4969,4973,4987,4993,4999
```

9.6 Subset Enumeration

Given a bitmask `m`, the following iterates through all submasks of `m`.

```
for (int s=m; s; s=(s-1)&m) {
    // ...
}
```

```
s = m
while s:
    # ...
    s = (s-1) & m
```

9.7 VimRC

```
set tabstop=4 softtabstop=4
set shiftwidth=4
set expandtab
set smartindent
set number
set relativenumber
set linebreak
set noswapfile
```

10 Strings

10.1 Knuth Morris Pratt: String Matching

10.2 Levinshtein Distance: Edit Distance