Team Reference Document, NWERC 2017

Team TUMbling - TU München

November 2017

Contents

C++ Language

Compiling

```
g++ -std=c++11 -Wall -Wextra -fsanitize=undefined -D_GLIBCXX_DEBUG
    program.cpp
./a.out < input.txt > output.txt
```

Timing

```
g++ -std=c++11 program.cpp
time ./a.out < input.txt > output.txt
```

Debugging

```
g++ -std=c++11 -g program.cpp
gdb a.out
    run < input.txt > output.txt
    finish
    quit
```

Input and output

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

int main() {
    ios_base::sync_with_stdio(false);

    int n; string s;
    cin >> n;

    //line break must be read before using getline
    getline(cin, s);
    for (int i = 0; i < n; i++)
        getline(cin, s), cout << s << "\n" << flush;

    long double pi = 3.01415926535897932384626433832795;
    //fixed number of digits after comma in the output
    cout << fixed << setprecision(10);
    cout << "Pi: " << pi << "\n";
}
```

Language specific functionalities

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

//Ordered statistics tree
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> ordered_set;

//GCC built-in functions
//For long long arguments add ll to the function name, e.g.
    __builtin_clzll(long long x)
//Returns the number of leading 0-bits in x, starting at the most
    significant bit position. If x is 0, the result is undefined.
int __builtin_clz (unsigned int x);

//Returns the number of trailing 0-bits in x, starting at the least
    significant bit position. If x is 0, the result is undefined.
int __builtin_ctz (unsigned int x);

//Returns the number of 1-bits in x.
int __builtin_popcount (unsigned int x);

//Returns one plus the index of the least significant 1-bit of x,
    or if x is zero, returns zero.
int __builtin_ffs (int x);

bool compFunc(int & a, int & b) {
    return a < b;
}

struct comp {
    bool operator()(int & a, int & b) {
        return compFunc(a, b);
    }
```

```
};

int main() {
    int x = 0, y = 1;
    cout << min(x, y) << " " << max(x, y) << "\n"; //0 1
    swap(x, y);
    cout << x << "\n"; //1

    //array
    int a[5];
    for (int i = 0; i < 5; i++)
        a[i] = 5 - i;
    sort(a, a + 3, compFunc);
    for (int i : a)
        cout << i << " "; //3 4 5 2 1
    cout << "\n";
    int b[5];
    memset(b, -1, sizeof b); //only for 0 and -1
    for (int i : b)
        cout << i << " "; //-1 -1 -1 -1 -1
    cout << "\n";

    //vector
    vector<int> v(a, a + 5);
    for (int i : v)
        cout << i << " "; //3 4 5 2 1
    cout << "\n";
    sort(v.rbegin(), v.rend());
    for (int i : v)
        cout << i << " "; //5 4 3 2 1
    cout << "\n";
    sort(v.begin(), v.end());
    cout << *lower_bound(v.begin(), v.end(), 2) << " " <<
        *upper_bound(v.begin(), v.end(), 4) << "\n"; //2 5
    for (auto it = lower_bound(v.begin(), v.end(), 2); it !=
        upper_bound(v.begin(), v.end(), 4); it++)
        cout << *it << " "; //2 3 4
    cout << "\n";
    next_permutation(v.begin(), v.end());
    for (int i : v)
        cout << i << " "; //1 2 3 5 4
    cout << "\n";
    int c = 1;
    do {
        c++;
    } while (next_permutation(v.begin(), v.end()));
    cout << c << "\n"; //120
    for (int i : v)
        cout << i << " "; //1 2 3 4 5
    cout << "\n";
    nth_element(v.begin(), v.begin() + v.size() / 2, v.end());
    cout << "Median: " << v[v.size() / 2] << "\n"; //Median: 3
    v.assign(5, 1);
    v.push_back(2); v.push_back(3); v.pop_back();
    for (int i : v)
        cout << i << " "; //1 1 1 1 1 2
    cout << "\n" << v.front() << " " << v.back() << "\n"; //1 2
```

```
    //stack and queue
    stack<int> s; queue<int> q;
    for (int i = 1; i < 5; i++)
        s.push(i), q.push(i);
    while (!s.empty())
        cout << s.top() << " ", s.pop(); //4 3 2 1
    cout << "\n";
    while (!q.empty())
        cout << q.front() << " ", q.pop(); //1 2 3 4
    cout << "\n";

    //deque
    deque<int> d;
    d.push_back(1); d.push_back(2); d.push_back(3);
        d.push_front(0); d.push_front(-1);
    d.pop_back(); d.pop_front();
    for (int i : d)
        cout << i << " "; //0 1 2
    cout << "\n" << d.size() << ", " << d[d.size() - 1] << " = " <<
        d.back() << "\n"; //3, 2 = 2

    //priority queue
    priority_queue<int, vector<int>, comp> pq;
    for (int i = 0; i < 5; i++)
        pq.push((5 - i) / 2);
    for (int i = 0; i < 5; i++)
        cout << pq.top() << " ", pq.pop(); //2 2 1 1 0
    cout << "\n";

    //set (multiset for multiple insertion into set)
    set<int> se;
    for (int i = 0; i < 5; i++)
        se.insert(i);
    se.erase(3); se.erase(se.begin());
    for (auto it = se.begin(); it != se.end(); it++)
        cout << *it << " "; //1 2 4
    cout << "\n";
    cout << (se.count(3) ? "In set" : "Not in set") << "\n"; //Not
        in set

    //map (multimap for multiple insertion into map)
    map<char, int> m;
    for (int i = 0; i < 5; i++)
        m['a' + i] = i;
    m.erase('c');
    for (int i = 0; i < 6; i++)
        cout << m['a' + i] << " "; //0 1 0 3 4 0
    cout << "\n";
    for (pair<char, int> i : m)
        cout << i.first << "," << i.second << " "; //a,0 b,1 c,0
            d,3 e,4 f,0
    cout << "\n";

    //unordered map (Hashmap) (+ equivalents for set, multiset and
        multimap), no ordered iterator
    unordered_map<char, int> um;
    for (int i = 0; i < 5; i++)
        um['a' + i] = i;
    um.erase('c');
    for (int i = 0; i < 6; i++)
        cout << um['a' + i] << " "; //0 1 0 3 4 0
    cout << "\n";

    //ordered set (Order statistics tree)
    ordered_set os;
    for (int i = 1; i < 5; i++)
        os.insert(i);
    os.erase(2);
    for (auto it = os.begin(); it != os.end(); it++)
        cout << *it << " "; //1 3 4
    cout << "\n" << os.order_of_key(3) << " " <<
        *os.find_by_order(2) << "\n"; //1 4
}
```

Data structures

Union find disjoint sets

- Input: n elements
- Preprocessing: $O(n)$ time and space
- Requesting the set of an element: $O(1)$ time and space
- Requesting to merge two sets: $O(1)$ time and space
- Requesting the size of a set: $O(1)$ time and space

```
int n, p[1000000], s[1000000];

int findSet(int i) {
    return p[i] == i ? i : (p[i] = findSet(p[i]));
}

bool inSameSet(int i, int j) {
    return findSet(i) == findSet(j);
}

int getSizeOfSet(int i) {
    return s[findSet(i)];
}

void unionSet(int i, int j) {
    if (!inSameSet(i, j))
        s[findSet(j)] += s[findSet(i)], p[findSet(i)] = findSet(j);
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        p[i] = i, s[i] = 1;
}
```

Fenwick tree

- Input: n elements with an associative operation which is reversible
- Preprocessing: $O(n \log n)$ time and $O(n)$ space
- Requesting to change an element: $O(\log n)$ time and $O(1)$ space
- Requesting the result of the operation on all elements in the range $[l, r]$: $O(\log n)$ time and $O(1)$ space
- Output: the result of the operation on all elements in the range $[l, r]$

```
#define LSOne(S) (S & (-S))

int n, v[1000000], f[1000000];

void change(int i, int d) {
    for (; i <= n; i += LSOne(i))
        f[i] += d;
}

int getSum(int i) {
    int sum = 0;
    for (; i; i -= LSOne(i))
        sum += f[i];
    return sum;
}

int getSum(int a, int b) {
    return getSum(b) - getSum(a - 1);
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> v[i + 1], change(i + 1, v[i + 1]);
    int x, y;
    //change value at position x
    change(x + 1, y - v[x + 1]), v[x + 1] = y;
    //get sum of the range [x, y]
    getSum(x + 1, y + 1);
}
```

Segment tree

- Input: n elements with an associative operation (and other operations for ranges) (2D: n^2 elements)
- Preprocessing: $O(n)$ time and space (2D: $O(n^2)$ time and space)
- Requesting to change an element or interval: $O(\log n)$ time and space (2D: $O(\log^2 n)$ time and $O(\log n)$ space)
- Requesting the result of the operation on all elements in the range $[l, r]$: $O(\log n)$ time and space (2D: $O(\log^2 n)$ time and $O(\log n)$ space)
- Output: the result of the operation on all elements in the range $[l, r]$

```
#define left(i) (2 * (i) + 1)
#define right(i) (2 * (i) + 2)
#define parent(i) (((i) - 1) / 2)

struct operation {
    int t = 0, v = 0;
};

const int ts = 1 << 21, to = (1 << 20) - 1;
int n, t[ts];
operation o[ts];

operation combine(operation ot, operation ob) {
    if (ot.t == 1)
        return ot;
    if (ot.t == 0)
        return ob;
    if (ob.t == 0)
```

```
        ob.t = 2, ob.v = 1;
    ob.v *= ot.v;
    return ob;
}

int getValue(int x, int l, int r) {
    return o[x].t == 1 ? o[x].v * (r - l) : o[x].t == 2 ? t[x] *
        o[x].v : t[x];
}

void calcValue(int x, int l, int r) {
    t[x] = getValue(left(x), l, (l + r) / 2) + getValue(right(x),
        (l + r) / 2, r);
}

int query(int a, int b, int x = 0, int l = 0, int r = 1 << 20) {
    if (a <= l && r <= b)
        return getValue(x, l, r);
    if (b <= l || r <= a)
        return 0;
    int m = (l + r) / 2;
    o[left(x)] = combine(o[x], o[left(x)]);
    o[right(x)] = combine(o[x], o[right(x)]);
    o[x].t = 0;
    int s = 0;
    s += query(a, b, left(x), l, m);
    s += query(a, b, right(x), m, r);
    calcValue(x, l, r);
    return s;
}

void apply(int a, int b, operation v, int x = 0, int l = 0, int r =
    1 << 20) {
    if (a <= l && r <= b) {
        o[x] = combine(v, o[x]);
        return;
    }
    if (b <= l || r <= a)
        return;
    int m = (l + r) / 2;
    o[left(x)] = combine(o[x], o[left(x)]);
    o[right(x)] = combine(o[x], o[right(x)]);
    o[x].t = 0;
    apply(a, b, v, left(x), l, m);
    apply(a, b, v, right(x), m, r);
    calcValue(x, l, r);
}

void build(int x = 0, int l = 0, int r = 1 << 20) {
    if (l + 1 == r)
        return;
    int m = (l + r) / 2;
    build(left(x), l, m);
    build(right(x), m, r);
    calcValue(x, l, r);
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> t[to + i];
    build();
    int l, r, v;
    //change the numbers in the range [l, r] to v
    operation op; op.t = 1, op.v = v;
    apply(l, r + 1, op);
    //multiply all numbers in the range [l, r] with v
    operation op; op.t = 2, op.v = v;
    apply(l, r + 1, op);
    //get the sum in the range [l, r]
    query(l, r + 1);
}
```

Persistent segment tree

- Input: n elements with an associative operation (and other operations for ranges)
- Preprocessing: $O(n)$ time and space
- Requesting to change an element or interval: $O(\log n)$ time and space
- Requesting the result of the operation on all elements in the range $[l, r]$ for the version v of the segment tree: $O(\log n)$ time and space
- Output: the result of the operation on all elements in the range $[l, r]$ for the version v of the segment tree

```
struct node {
    node * l = 0, * r = 0;
    int v = 0;
};

const int ts = 1 << 20;
int n, ver = 0, v[ts];
node * root[1000000];

void calcValue(node * x) {
    x->v = x->l->v + x->r->v;
}

node * copyNode(node * x) {
    node * r = new node();
    r->l = x->l;
    r->r = x->r;
    r->v = x->v;
    return r;
}

int query(node * x, int a, int b, int l = 0, int r = 1 << 20) {
    if (a <= l && r <= b)
        return x->v;
    if (b <= l || r <= a)
        return 0;
    int m = (l + r) / 2;
    int s = 0;
    s += query(x->l, a, b, l, m);
    s += query(x->r, a, b, m, r);
    return s;
}

void update(node * x, int p, int l = 0, int r = 1 << 20) {
    if (l + 1 == r) {
        x->v = v[p];
        return;
    }
    int m = (l + r) / 2;
    if (p < m) {
        x->l = copyNode(x->l);
        update(x->l, p, l, m);
    } else {
        x->r = copyNode(x->r);
        update(x->r, p, m, r);
    }
    calcValue(x);
}

void build(node * x, int l = 0, int r = 1 << 20) {
    if (l + 1 == r) {
        x->v = v[l];
        return;
    }
    int m = (l + r) / 2;
    x->l = new node();
    x->r = new node();
    build(x->l, l, m);
    build(x->r, m, r);
    calcValue(x);
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> v[i];
    root[0] = new node();
    build(root[0]);
    int a, l, r;
    //change the number at position l
    v[l] = a;
    root[ver + 1] = copyNode(root[ver]), ver++;
    update(root[ver], l);
    //get the sum in the range [l, r] of version a
    query(root[a], l, r + 1);
}
```

Dynamic programming

Range maximum

- Input: n numbers
- Preprocessing: $O(n \log n)$ time and space
- Requesting the maximum of all numbers in the range $[l, r]$: $O(1)$ time and space
- Output: the maximum of all numbers in the range $[l, r]$

```
int n, a[100000], m[17][100000];

int getMaximum(int l, int r) {
    int s = (int) log2(r - l + 1);
    return a[m[s][l]] > a[m[s][r - (1 << s) + 1]] ? m[s][l] :
        m[s][r - (1 << s) + 1];
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i], m[0][i] = i;
    for (int j = 1; 1 << j <= n; j++)
        for (int i = 0; i + (1 << j) <= n; i++)
            m[j][i] = a[m[j - 1][i]] > a[m[j - 1][i + (1 << (j -
                1))]] ? m[j - 1][i] : m[j - 1][i + (1 << (j - 1))];
    int l, r;
    //get the maximum in the range [l, r]
    a[getMaximum(l, r)];
}
```

Longest increasing subsequence

- Input: n numbers
- Requesting the longest increasing subsequence: $O(n \log k)$ time and $O(n)$ space
- Output: the size k of the longest increasing subsequence and its elements

```
int n, k = 0, a[100000], b[100000], v[100000], l[100000];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];

    for (int i = 0; i < n; i++) {
        int p = lower_bound(v, v + k, a[i]) - v;
        v[p] = a[i]; b[p] = i; l[i] = (p > 0 ? b[p - 1] : -1);
        k = max(k, p + 1);
    }

    //the length of the longest increasing subsequence
    k;
    int c = (k > 0 ? b[k - 1] : -1);
    stack<int> s;
    while (c != -1)
        s.push(c), c = l[c];
    //the sequence of the longest increasing subsequence
    while (!s.empty())
        a[s.top()], s.pop();
}
```

Knapsack

0-1 Knapsack

- Input: n objects with values and weights
- Requesting the maximum value of all subsets of objects with weight less than k: $O(n*k)$ time and $O(n*k)$ space
- Output: the maximum value of all subsets of objects with weight less than k and its elements

```
int n, k, v[1000], w[1000], m[1001][1000], u[1001][1000];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> v[i];
    for (int i = 0; i < n; i++)
        cin >> w[i];
    cin >> k;

    for (int i = 0; i < n; i++)
        for (int j = 0; j <= k; j++) {
            m[i + 1][j] = m[i][j];
            if (j - w[i] >= 0 && m[i][j - w[i]] + v[i] > m[i +
                1][j]) {
                m[i + 1][j] = m[i][j - w[i]] + v[i];
                u[i + 1][j] = 1;
            }
        }

    int c = 0, maxm = 0;
    for (int i = 0; i <= k; i++)
        if (m[n][i] > maxm) {
            maxm = m[n][i];
            c = i;
        }

    //the maximum possible value
    maxm;
    //the values, weights and amounts of the used objects
    for (int i = n - 1; i >= 0; i--) {
        if (u[i + 1][c] > 0)
            //[v[i], w[i], u[i + 1][c]
        c -= u[i + 1][c] * w[i];
    }
}
```

Integer knapsack

- Input: n objects with values, weights and amounts
- Requesting the maximum value of all combinations of objects with weight less than k: $O(n * k)$ time and $O(n * k)$ space
- Output: the maximum value of all combinations of objects with weight less than k and its elements

```
int n, k, v[1000], w[1000], a[1000], m[1001][1000], u[1001][1000];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> v[i];
    for (int i = 0; i < n; i++)
        cin >> w[i];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    cin >> k;

    for (int i = 0; i < n; i++)
        for (int j = 0; j < w[i] && j <= k; j++) {
            int c = j;
            deque<int> d;
            while (c <= k) {
                while (!d.empty() && (c - d.front()) / w[i] > a[i])
                    d.pop_front();
                while (!d.empty() && m[i][c] >= m[i][d.back()] +
                    v[i] * (c - d.back()) / w[i])
                    d.pop_back();
                d.push_back(c);
                u[i + 1][c] = (c - d.front()) / w[i];
                m[i + 1][c] = m[i][d.front()] + v[i] * u[i + 1][c];
                c += w[i];
            }
        }

    int c = 0, maxm = 0;
    for (int i = 0; i <= k; i++)
        if (m[n][i] > maxm) {
            maxm = m[n][i];
            c = i;
        }

    //the maximum possible value
    maxm;
    //the values, weights and amounts of the used objects
    for (int i = n - 1; i >= 0; i--) {
        if (u[i + 1][c] > 0)
            //[v[i], w[i], u[i + 1][c]
        c -= u[i + 1][c] * w[i];
    }
}
```

Unlimited integer knapsack

- Input: n objects with values and weights
- Requesting the maximum value of all combinations of objects with weight less than k: $O(n * k)$ time and $O(n * k)$ space
- Output: the maximum value of all combinations of objects with weight less than k and its elements

```
int n, k, v[1000], w[1000], m[1001][1000], u[1001][1000];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> v[i];
    for (int i = 0; i < n; i++)
        cin >> w[i];
    cin >> k;

    for (int i = 0; i < n; i++)
        for (int j = 0; j <= k; j++) {
            m[i + 1][j] = m[i][j];
            if (j - w[i] >= 0 && m[i + 1][j - w[i]] + v[i] > m[i +
                1][j]) {
                m[i + 1][j] = m[i + 1][j - w[i]] + v[i];
                u[i + 1][j] = u[i + 1][j - w[i]] + 1;
            }
        }

    int c = 0, maxm = 0;
    for (int i = 0; i <= k; i++)
        if (m[n][i] > maxm) {
            maxm = m[n][i];
            c = i;
        }
    //the maximum possible value
    maxm;
    //the values, weights and amounts of the used objects
    for (int i = n - 1; i >= 0; i--) {
        if (u[i + 1][c] > 0)
            //[v[i], w[i], u[i + 1][c]
        c -= u[i + 1][c] * w[i];
    }
}
```

Optimization

Stack: maximum subarray

- Input: $n * m$ numbers
- Requesting the maximum subarray: $O(n * m)$ time and space
- Output: the size and position of the maximum subarray

```
int n, m, a[1000][1000], d[1000][1000];

int main() {
    cin >> n >> m;
    for (int y = 0; y < m; y++)
        for (int x = 0; x < n; x++)
            cin >> a[x][y];

    for (int x = 0; x < n; x++)
        for (int y = 0; y < m; y++)
            d[x][y] = (a[x][y] == 0 ? 0 : (y == 0 ? 1 : d[x][y - 1]
                + 1));

    int maxa = 0, px = 0, py = 0, w = 0, h = 0;
    for (int y = 0; y < m; y++) {
        stack<int> s;
        for (int x = 0; x < n; x++) {
            while (!s.empty() && d[s.top()][y] > d[x][y]) {
                if ((x - s.top()) * d[s.top()][y] > maxa) {
                    w = x - s.top(); h = d[s.top()][y];
                    maxa = w * h; px = x - w; py = y + 1 - h;
                }
                s.pop();
            }
            s.push(x);
        }
        while (!s.empty()) {
            if ((n - s.top()) * d[s.top()][y] > maxa) {
                w = n - s.top(); h = d[s.top()][y];
                maxa = w * h; px = n - w; py = y + 1 - h;
            }
            s.pop();
        }
    }
    //the area of the maximum subarray
    maxa;
    //the position of the maximum subarray  [px, py]
    //the size of the maximum subarray [w, h]
}
```

Deque: maximum in all ranges of a fixed size

- Input: n numbers (2D: n^2 numbers)
- Requesting the maximum in all ranges of size k: $O(n)$ time and $O(k)$ space (2D: $O(n^2)$ time and space)
- Output: the maximum in all ranges of size k

```
int n, a[1000000];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int k;
    //the maximum in all ranges of the requested size: ";
    deque<int> d;
    for (int i = 0; i < k; i++) {
        while (!d.empty() && a[i] >= a[d.back()])
            d.pop_back();
        d.push_back(i);
    }
    //the maximum in the first range a[d.front()]
    for (int i = k; i < n; i++) {
        while (!d.empty() && i - d.front() >= k)
            d.pop_front();
        while (!d.empty() && a[i] >= a[d.back()])
            d.pop_back();
        d.push_back(i);
        //the maximum in the other ranges a[d.front()]
    }
}
```

Stack

- Aim: find the optimum in the previous n elements
- Behavior of the optimum:
  - For elements i and j, where $i < j$, there exists some number $m_{i,j}$ of elements, so that while $n < m_{i,j}$, j is the optimum, and when $n \geq m_{i,j}$, i is the optimum
  - For elements i, j and k, if i is the optimum compared to j, and j is the optimum compared to k, then i is the optimum compared to k
- Optimization:
  - Usage of a stack with the optimum at the top and other possible optimums below
  - For elements i and j in the stack, where $i < j$, j must be above i in the stack
  - For elements i, j and k in the stack, where $i < j < k$, $m_{i,j} > m_{j,k}$
  - For each new element:
    - For the top two elements i and j of the stack, where $i < j$, if $n \geq m_{i,j}$, pop j of the stack
    - The optimum is now on the top of the stack
    - For the top two elements i and j of the stack, where $i < j$, and the new element k, if $m_{i,j} \leq m_{j,k}$, pop j of the stack
    - For the top element i of the stack and the new element j, if $n < m_{i,j}$, push j onto the stack

Deque

- Aim: find the optimum in the previous n elements
- Behavior of the optimum:
  - For elements i and j, where $i < j$, there exists some number $m_{i,j}$ of elements, so that while $n < m_{i,j}$, i is the optimum, and when $n \geq m_{i,j}$, j is the optimum
  - For elements i, j and k, if i is the optimum compared to j, and j is the optimum compared to k, then i is the optimum compared to k
- Optimization:
  - Usage of a deque with the optimum at the top and other possible optimums below

  - For elements i and j in the deque, where $i < j$, i must be above j in the deque
  - For elements i, j and k in the deque, where $i < j < k$, $m_{i,j} < m_{j,k}$
  - For each new element:
    * For the top two elements i and j of the deque, where $i < j$, if $n \geq m_{i,j}$, pop i of the deque
    * For the bottom element i of the deque and the new element j, if $n \geq m_{i,j}$, pop i of the deque
    * For the new element i, push i onto the bottom of the deque
    * The optimum is now on the top of the deque

Binary search

- Aim: find the optimum in the previous n elements
- Behavior of the optimum: for elements i and j, where j is the optimum, without knowing the optimum, it is clear that either $i \leq j$ or $j \leq i$
- Optimization:
  - Begin with range $[0, n-1]$
  - For the range $[l, r]$ and elements i and $m = (l+r)/2$, where $l+1 < r$ and i is the optimum, without knowing the optimum, if $m \leq i$, then continue with $[m, r]$, else continue with $[l, m]$
  - For the range $[l, r]$, where $l \leq r$ and $l+1 \geq r$, choose the optimum

Pointer

- Aim: find the optimum in the previous n elements
- Behavior of the optimum:
  - For elements i and j, where j is the optimum, without knowing the optimum, it is clear that either $i \leq j$ or $j \leq i$
  - For elements i and j, where $i < j$, there exists some number $m_{i,j}$ of elements, so that while $n < m_{i,j}$, i is the optimum, and when $n \geq m_{i,j}$, j is the optimum
  - For elements i, j and k, if i is the optimum compared to j, and j is the optimum compared to k, then i is the optimum compared to k
- Optimization:
  - Begin with the optimum of the previous $n-1$ elements
  - For the current element i and the next element j, if $n \geq m_{i,j}$, continue with j, else i is the optimum

Convex hull trick

- Aim: find the optimum in the previous n elements
- Behavior of the optimum:
  - For elements i and j there exists some threshold depending on one parameter of the new element, so that while the parameter is below the threshold, one of the elements is the optimum, and when the parameter is above the threshold, the other element is the optimum
  - For elements i, j and k, if i is the optimum compared to j, and j is the optimum compared to k, then i is the optimum compared to k
- Optimization:
  - Usage of a set with all possible optimums
  - For elements i and j in the set, where i is the optimum below the threshold and j is the optimum above the threshold, i must be before j in the set
  - For elements i, j and k, where i before j and j before k in the set, the threshold of i and j must be below the threshold of j and k
  - For each new element:
    * Find the optimum with binary search
    * Find the position of the new element in the set with binary search
    * For the new element j and the elements i and k, where i is the element before the position of j and k is the element after the position of j, if i or k doesn't exist or if the threshold of i and j is below the threshold of j and k, insert j into the set
    * If the new element was inserted into the set, for element j, where j is the element before or after the new element, and the elements i and k, where i is the element before j and k is the element after j, if i, j and k exist and the threshold of i and j is above the threshold of j and k, delete j from the set

Divide and conquer

- Aim: find the optimum for n new elements in the previous m elements
- Behavior of the optimum: for the new elements i and j, where $i < j$, and their optimums k and l, $k \leq l$
- Optimization:
  - Begin with range $[0, n-1]$ and the possible optimums $[0, m-1]$
  - For the range $[l, r]$, the possible optimums $[lo, ro]$ and element $m = (l+r)/2$, where $l \leq r$ find the optimum i for m in the range of possible optimums and continue with the range $[l, m-1]$ and the possible optimums $[lo, i]$ as well as with the range $[m+1, r]$ and the possible optimums $[i, ro]$
  - For the range $[l, r]$, where $l > r$, do nothing

Knuth optimization

- Aim: find the optimum for n new elements in the previous m elements
- Behavior of the optimum: for the new element i and some old elements j and k, the optimum for the new element lies between the optimums for the old elements
- Optimization: find the optimum between the optimums of the old elements

Graph

Traversal

Articulation points and bridges

- Input: undirected graph with v vertices and e edges
- Find all articulation points and bridges: $O(v+e)$ time and space
- Output: all articulation points and bridges

```
struct edge {
    int j, inv;
    bool br;
};

int v, e, n, num[100000], low[100000], ar[100000];
bitset<100000> vis;
vector<int> art; vector<pair<int, int>> bri;
vector<edge> adj[100000];

void dfs(int i, int p = -1) {
    vis[i] = true, num[i] = low[i] = n++;
    bool root = p == -1;
    int sub = 0, hgh = 0;
    for (edge & ed : adj[i]) {
        if (ed.j == p) {
            p = -1;
            continue;
        }
        if (vis[ed.j])
            low[i] = min(low[i], num[ed.j]);
        else {
            dfs(ed.j, i);
            if (low[ed.j] > num[i])
                ed.br = true, adj[ed.j][ed.inv].br = true,
                    bri.push_back({i, ed.j});
            low[i] = min(low[i], low[ed.j]);
            hgh = max(hgh, low[ed.j]);
            sub++;
        }
    }
    if (!root && hgh >= num[i] || root && sub > 1)
        ar[i] = true, art.push_back(i);
}

int main() {
    cin >> v >> e; int a, b;
    for (int i = 0; i < e; i++) {
        cin >> a >> b;
        adj[a].push_back({b, adj[b].size(), false});
        adj[b].push_back({a, adj[a].size() - 1, false});
    }

    for (int i = 0; i < v; i++)
        if (!vis[i])
            dfs(i);

    //the articulation points
    for (int i = 0; i < art.size(); i++)
        art[i];
    //the bridges
    for (pair<int, int> ed : bri)
        ed;
}
```

Strongly connected components

- Input: graph with v vertices and e edges
- Find all strongly connected components: $O(v+e)$ time and $O(v)$ space
- Output: all strongly connected components and their vertices

```
struct edge {
    int j;
};

int v, e, n, c, num[100000], low[100000], cm[100000];
bitset<100000> vis, ins;
stack<int> s; vector<vector<int>> scc;
vector<edge> adj[100000], adg[100000];

void dfs(int i) {
    vis[i] = true, num[i] = low[i] = n++;
    s.push(i), ins[i] = true;
    for (edge ed : adj[i]) {
        if (!vis[ed.j])
            dfs(ed.j);
        if (ins[ed.j])
            low[i] = min(low[i], low[ed.j]);
    }
    if (low[i] == num[i]) {
        int j;
        vector<int> csc;
        do {
            j = s.top(), s.pop(), ins[j] = false;
            cm[j] = c, csc.push_back(j);
        } while (j != i);
        scc.push_back(csc), c++;
    }
}
```

```
int main() {
    cin >> v >> e; int a, b;
    for (int i = 0; i < e; i++)
        cin >> a >> b, adj[a].push_back({b});
    for (int i = 0; i < v; i++)
        if (!vis[i])
            dfs(i);
    for (int i = 0; i < v; i++)
        for (edge ed : adj[i])
            if (cm[i] != cm[ed.j])
                adg[cm[i]].push_back({cm[ed.j]});

    //the number of found components
    c;
    //the components
    for (vector<int> cm : scc)
        for (int i = 0; i < cm.size(); i++)
            cm[i];
    //the edges of the contracted graph
    for (int i = 0; i < c; i++)
        for (edge ed : adg[i])
            ed;
}
```

Minimum spanning tree

- Input: undirected graph with v vertices and e edges with weights
- Find the minimum spanning tree: $O(e \log v)$ time and $O(e)$ space
- Output: the minimum spanning tree and its cost

```
struct edge {
    int i, j, w;
    bool operator<(edge & b) {
        return w < b.w;
    }
};

int v, e, c, p[1000000];
edge edg[100000];
vector<edge> adj[100000];

int findSet(int i) {
    return p[i] == i ? i : (p[i] = findSet(p[i]));
}

bool inSameSet(int i, int j) {
    return findSet(i) == findSet(j);
}

void unionSet(int i, int j) {
    if (!inSameSet(i, j))
        p[findSet(i)] = findSet(j);
}

int main() {
    cin >> v >> e;
    for (int i = 0; i < v; i++)
        p[i] = i;
    int a, b, w;
    for (int i = 0; i < e; i++) {
        cin >> a >> b >> w;
        edg[i] = {a, b, w};
    }

    sort(edg, edg + e);
    for (int i = 0; i < e; i++)
        if (!inSameSet(edg[i].i, edg[i].j)) {
            c += edg[i].w;
            unionSet(edg[i].i, edg[i].j);
            adj[edg[i].i].push_back({edg[i].j, edg[i].j, edg[i].w});
            adj[edg[i].j].push_back({edg[i].i, edg[i].i, edg[i].w});
        }
    //the cost of the minimum spanning tree
    c;
    //the edges of the minimum spanning tree
    for (int i = 0; i < v; i++)
        for (edge ed : adj[i])
            ed;
}
```

Shortest path

Shortest path faster algorithm

- Input: graph with v vertices and e edges with weights
- Find the shortest path from vertex i to vertex j: $O(v * e)$ time and $O(v)$ space
- Output: the shortest path from vertex i to vertex j and its cost

```
struct edge {
    int j, w;
};

const int inf = 1 << 30; int v, e, d[100000], p[100000], c[100000];
bool nwc = false; bitset<100000> inq;
stack<int> ver; vector<edge> adj[100000];

void shortestPathFasterAlgorithm(int i, int j) {
    nwc = false;
    for (int k = 0; k < v; k++)
        d[k] = inf, p[k] = -1, c[k] = 0;
    queue<int> q;
    inq.reset();
    d[i] = 0, p[i] = i;
    q.push(i);
    inq[i] = true;
    while (!q.empty()) {
        int k = q.front();
        q.pop(), inq[k] = false;
        c[k]++;
        if (c[k] >= v) {
            nwc = true;
            break;
        }
        for (edge ed : adj[k])
            if (d[k] + ed.w < d[ed.j]) {
                d[ed.j] = d[k] + ed.w;
                p[ed.j] = k;
                if (!inq[ed.j])
                    inq[ed.j] = true, q.push(ed.j);
            }
    }
    if (p[j] != -1 && !nwc)
        while (j != i)
            ver.push(j), j = p[j];
}

int main() {
    cin >> v >> e; int a, b, w;
    for (int i = 0; i < e; i++)
        cin >> a >> b >> w, adj[a].push_back({b, w});

    int s, t;
    shortestPathFasterAlgorithm(s, t);
    //the graph has a negative weight circle
    nwc;
    //the shortest path
    while (!ver.empty())
        //s -> ver.top() d[ver.top()] - d[s]
        s = ver.top(), ver.pop();
}
```

Floyd Warshall

- Input: graph with v vertices and e edges with weights
- Preprocessing: $O(v^3)$ time and $O(v^2)$ space
- Find the shortest path from vertex i to vertex j: $O(1)$ time and space
- Output: the shortest path from vertex i to vertex j and its cost

```
int v, e, d[400][400], p[400][400];
bool nwc = false; stack<int> ver;

void floydWarshall(int i, int j) {
    if (d[i][j] < inf && !nwc)
        while (j != i)
            ver.push(j), j = p[i][j];
}

int main() {
    cin >> v >> e;
    for (int i = 0; i < v; i++) {
        for (int j = 0; j < v; j++)
            d[i][j] = inf, p[i][j] = i;
        d[i][i] = 0;
    }
    int a, b, w;
    for (int i = 0; i < e; i++)
        cin >> a >> b >> w, d[a][b] = w;

    for (int k = 0; k < v; k++)
        for (int i = 0; i < v; i++)
            for (int j = 0; j < v; j++)
                if (d[i][k] < inf && d[k][j] < inf && d[i][k] +
                    d[k][j] < d[i][j])
                    d[i][j] = d[i][k] + d[k][j], p[i][j] = p[k][j];
    for (int i = 0; i < v; i++)
        if (d[i][i] < 0)
            nwc = true;
    floydWarshall(s, t);
}
```

Flow

Max flow

- Input: graph with v vertices, e edges with capacities, a source s and a sink t
- Find the maximum flow from s to t: $O(e * v^2)$ time and $O(v + e)$ space
- Output: the maximum flow from s to t

```
struct edge {
    int j, c, f, inv;
};

const int inf = 1 << 30;
int v, e, s, t, mfl, lev[400], nxt[400];
vector<edge> adj[400];

int sendFlow(int i = s, int f = inf) {
    if (i == t)
        return f;
    for (; nxt[i] < adj[i].size(); nxt[i]++) {
        edge & ed = adj[i][nxt[i]];
        if (lev[ed.j] == lev[i] + 1 && ed.f < ed.c) {
            int cf = min(f, ed.c - ed.f);
            int tf = sendFlow(ed.j, cf);
            if (tf > 0) {
                ed.f += tf;
                adj[ed.j][ed.inv].f -= tf;
                return tf;
            }
        }
    }
    return 0;
}

bool bfs() {
    memset(lev, -1, sizeof lev);
    lev[s] = 0;
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
        int i = q.front();
        q.pop();
        for (edge ed : adj[i])
            if (lev[ed.j] == -1 && ed.f < ed.c)
                lev[ed.j] = lev[i] + 1, q.push(ed.j);
    }
    return lev[t] >= 0;
}

void maxFlow() {
    if (s == t) {
        mfl = inf;
        return;
    }
    mfl = 0;
    while (bfs()) {
        memset(nxt, 0, sizeof nxt);
        while (int f = sendFlow())
            mfl += f;
    }
}

int main() {
    cin >> v >> e >> s >> t;
    int a, b, c;
    for (int i = 0; i < e; i++) {
        cin >> a >> b >> c;
        adj[a].push_back({b, c, 0, adj[b].size()});
        adj[b].push_back({a, 0, 0, adj[a].size() - 1});
    }

    maxFlow();

    //the maximum flow
    mfl;
    //the flow
    for (int i = 0; i < v; i++)
        for (edge ed : adj[i])
            if (ed.f > 0)
                ed;
}
```

Min cost max flow

- Input: graph with v vertices, e edges with capacities and costs, a source s and a sink t
- Find the maximum flow with minimum costs from s to t: $O(v^2 * e^2)$ time and $O(v + e)$ space
- Output: the maximum flow with minimum costs from s to t

```
struct edge {
    int j, c, co, f, inv;
};

const int inf = 1 << 30;
int v, e, s, t, mfl, mco, dst[100], pre[100], flw[100];
bitset<100000> inq;
vector<edge> adj[100];

bool sendFlow() {
    for (int i = 0; i < v; i++)
        dst[i] = inf, pre[i] = -1, flw[i] = 0;
    queue<int> q;
    inq.reset();
    dst[s] = 0, flw[s] = inf;
    q.push(s);
    inq[s] = true;
    while (!q.empty()) {
        int i = q.front();
        q.pop(), inq[i] = false;
        for (edge ed : adj[i])
            if (ed.f < ed.c && dst[i] + ed.co < dst[ed.j]) {
                dst[ed.j] = dst[i] + ed.co;
                pre[ed.j] = ed.inv;
                flw[ed.j] = min(flw[i], ed.c - ed.f);
                if (!inq[ed.j])
                inq[ed.j] = true, q.push(ed.j);
            }
    }
    if (flw[t] == 0)
        return false;
    int i = t;
    mfl += flw[t];
    while (i != s) {
        edge & ed = adj[i][pre[i]];
        ed.f -= flw[t];
        adj[ed.j][ed.inv].f += flw[t];
        mco += flw[t] * adj[ed.j][ed.inv].co;
        i = ed.j;
    }
}

void minCostMaxFlow() {
    if (s == t) {
        mfl = inf, mco = 0;
        return;
    }
    mfl = 0, mco = 0;
    while (sendFlow());
}

int main() {
    cin >> v >> e >> s >> t;
    int a, b, c, co;
    for (int i = 0; i < e; i++) {
        cin >> a >> b >> c >> co;
        adj[a].push_back({b, c, co, 0, adj[b].size()});
        adj[b].push_back({a, 0, -co, 0, adj[a].size() - 1});
    }

    minCostMaxFlow();

    //the maximum flow
    mfl;
    //the minimum costs
    mco;
    //the flow
    for (int i = 0; i < v; i++)
        for (edge ed : adj[i])
            if (ed.f > 0)
                ed;
}
```

Max cardinality bipartite matching

- Input: bipartite graph with v vertices and e edges
- Find the maximum cardinality matching: $O(v * e)$ time and $O(v)$ space
- Output: the maximum cardinality matching

```
struct edge {
    int j;
};

int v, e, mbm, mat[1000], lev[1000];
bool ibg;
bitset<1000> vis;
vector<int> vl;
vector<edge> adj[1000];

bool findPath(int i) {
    for (edge ed : adj[i])
        if (!vis[ed.j]) {
            vis[ed.j] = true;
            if (mat[ed.j] == -1 || findPath(mat[ed.j])) {
                mat[i] = ed.j;
                mat[ed.j] = i;
                return true;
            }
        }
    return false;
}

void dfs(int i, int l = 1) {
    lev[i] = l;
    if (lev[i] == 1)
        vl.push_back(i);
    for (edge ed : adj[i]) {
        if (!lev[ed.j])
            dfs(ed.j, 3 - l);
        else
            ibg = ibg && lev[i] != lev[ed.j];
    }
}

void maxCardinalityBipartiteMatching() {
    mbm = 0;
    ibg = true;
    vl.clear();
    for (int i = 0; i < v; i++) {
        mat[i] = -1;
        if (!lev[i])
            dfs(i);
    }
    if (!ibg)
        return;
    for (int i : vl) {
        vis.reset();
        if (mat[i] == -1)
            mbm += findPath(i);
    }
}

int main() {
    cin >> v >> e;
    int a, b;
    for (int i = 0; i < e; i++) {
        cin >> a >> b;
        adj[a].push_back({b});
        adj[b].push_back({a});
    }

    maxCardinalityBipartiteMatching();
    if (ibg) {
        //the maximum cardinality bipartite matching
        mbm;
        for (int i : vl)
            if (mat[i] != -1)
                mat[i];
    } //else no bipartite graph
}
```

Euler path and tour

Euler path and tour on undirected graph

- Input: undirected graph with v vertices and e edges
- Find an Euler path or tour: $O(v + e)$ time and $O(v + e)$ space
- Output: an Euler path or tour

```
struct edge {
    int j, inv;
    bool vis;
};

int v, e, deg[100000], nxt[100000];
bool ieg, het;
bitset<100000> vis;
stack<int> ver;
vector<edge> adj[100000];

int dfs(int i) {
    int r = 1;
    vis[i] = true;
    for (edge ed : adj[i])
        if (!vis[ed.j])
            r += dfs(ed.j);
    return r;
}

void eulerPathTour() {
    int i = 0, dv = 0, odv = 0;
    for (int j = 0; j < v; j++) {
        if (deg[j] > 0) {
            dv++;
            if (odv == 0)
                i = j;
        }
        if ((deg[j] % 2) == 1) {
            odv++;
            i = j;
        }
    }
    ieg = odv <= 2 && dfs(i) == dv;
    het = ieg && odv == 0;
    stack<int> s;
    s.push(i);
    while (!s.empty()) {
        i = s.top();
        for (; nxt[i] < adj[i].size() && adj[i][nxt[i]].vis;
            nxt[i]++);
        if (nxt[i] < adj[i].size()) {
            edge & ed = adj[i][nxt[i]];
            ed.vis = true, adj[ed.j][ed.inv].vis = true;
            s.push(ed.j);
        } else
            s.pop(), ver.push(i);
    }
}

int main() {
    cin >> v >> e;
    int a, b;
    for (int i = 0; i < e; i++) {
        cin >> a >> b;
        adj[a].push_back({b, adj[b].size(), false});
        deg[a]++;
        adj[b].push_back({a, adj[a].size() - 1, false});
        deg[b]++;
    }

    eulerPathTour();

    //is eulerian graph
    if (ieg) {
        //has an euler tour, otherwise path
        het;
        //the tour / path
        int s = ver.top();
        ver.pop();
        while (!ver.empty())
            //s -> ver.top()
            s = ver.top(), ver.pop();
    }
}
```

Euler path and tour on directed graph

- Input: directed graph with v vertices and e edges
- Find an Euler path or tour: $O(v + e)$ time and $O(v + e)$ space
- Output: an Euler path or tour

```
struct edge {
    int j;
    bool vis;
};

int v, e, din[100000], dou[100000], nxt[100000];
bool ieg, het;
bitset<100000> vis;
stack<int> ver;
vector<edge> adj[100000];

int dfs(int i) {
    int r = 1;
    vis[i] = true;
    for (edge ed : adj[i])
        if (!vis[ed.j])
            r += dfs(ed.j);
    return r;
}
```

```
void eulerPathTour() {
    int i = 0, dv = 0, ddv = 0;
    for (int j = 0; j < v; j++) {
        if (din[j] > 0 || dou[j] > 0) {
            dv++;
            if (ddv == 0)
                i = j;
        }
        if (abs(din[j] - dou[j]) > 0) {
            ddv += abs(din[j] - dou[j]);
            if (dou[j] > din[j])
                i = j;
        }
    }
    ieg = ddv <= 2 && dfs(i) == dv;
    het = ieg && ddv == 0;
    stack<int> s;
    s.push(i);
    while (!s.empty()) {
        i = s.top();
        for (; nxt[i] < adj[i].size() && adj[i][nxt[i]].vis;
            nxt[i]++);
        if (nxt[i] < adj[i].size()) {
            edge & ed = adj[i][nxt[i]];
            ed.vis = true;
            s.push(ed.j);
        } else
            s.pop(), ver.push(i);
    }
}

int main() {
    cin >> v >> e;
    int a, b;
    for (int i = 0; i < e; i++) {
        cin >> a >> b;
        adj[a].push_back({b, false});
        dou[a]++, din[b]++;
    }

    eulerPathTour();

    //is eulerian graph
    if (ieg) {
        //has an euler tour, otherwise path
        het;
        //the tour / path
        int s = ver.top();
        ver.pop();
        while (!ver.empty())
            //s -> ver.top()
            s = ver.top(), ver.pop();
    }
}
```

Lowest common ancestor

- Input: tree with v vertices
- Preprocessing: $O(v \log v)$ time and space for range maximum in section
- Find the lowest common ancestor of two vertices: $O(1)$ time and space
- Output: the lowest common ancestor of the two vertices

```
struct edge {
    int j;
};

int v, root, n, a[200000], m[18][200000], ind[100000], ver[200000];
vector<edge> adj[100000];

int getMaximum(int l, int r) {
    int s = (int) log2(r - l + 1);
    return a[m[s][l]] < a[m[s][r - (1 << s) + 1]] ? m[s][l] :
        m[s][r - (1 << s) + 1];
}

void dfs(int i, int p = -1, int d = 0) {
    ind[i] = n;
    ver[n] = i;
    a[n++] = d;
    for (edge ed : adj[i]) {
        if (ed.j == p)
            continue;
        dfs(ed.j, i, d + 1);
        ver[n] = i;
        a[n++] = d;
    }
}

int main() {
    cin >> v >> root;
    int b, c;
    for (int i = 0; i < v - 1; i++) {
        cin >> b >> c;
        adj[b].push_back({c});
        adj[c].push_back({b});
    }

    dfs(root);
    for (int i = 0; i < n; i++)
        m[0][i] = i;
    for (int j = 1; 1 << j <= n; j++)
        for (int i = 0; i + (1 << j) <= n; i++)
            m[j][i] = a[m[j - 1][i]] < a[m[j - 1][i + (1 << (j -
                1))]] ? m[j - 1][i] : m[j - 1][i + (1 << (j - 1))];

    //the lowest common ancestor of two vertices b and c
    ver[getMaximum(min(ind[b], ind[c]), max(ind[b], ind[c]))];
}
```

String

Trie

- Input: n strings of combined length m
- Preprocessing: $O(m)$ time and space
- Requesting to insert or delete a string of length l from the trie: $O(l)$ time and space
- Requesting, whether a string of length l is in the trie: $O(l)$ time and space

```
struct node {
    int eos = 0, chi = 0;
    node * c[26];
};

node * root;

int getIndex(char c) {
    if (c >= 'a' && c <= 'z')
        return c - 'a';
    return 0;
}

node * createNode() {
    node * r = new node();
    for (int i = 0; i < 26; i++)
        r->c[i] = 0;
    return r;
}

void insertString(string & s, int i = 0, node * n = root) {
    if (i == s.size()) {
        n->eos++;
        return;
    }
    if (n->c[getIndex(s[i])] == 0)
        n->c[getIndex(s[i])] = createNode(), n->chi++;
    insertString(s, i + 1, n->c[getIndex(s[i])]);
}

bool deleteString(string & s, int i = 0, node * n = root) {
    if (i == s.size()) {
        if (n->eos > 0)
            n->eos--;
        if (n->eos == 0 && n->chi == 0)
            return true;
        else
            return false;
    }
    if (n->c[getIndex(s[i])] == 0)
        return false;
    bool del = deleteString(s, i + 1, n->c[getIndex(s[i])]);
    if (del)
        delete n->c[getIndex(s[i])], n->c[getIndex(s[i])] = 0,
            n->chi--;
    if (n->eos == 0 && n->chi == 0)
        return true;
    else
        return false;
}

bool inTrie(string & s, int i = 0, node * n = root) {
    if (i == s.size())
        return n->eos > 0;
    if (n->c[getIndex(s[i])] == 0)
        return false;
    else
        return inTrie(s, i + 1, n->c[getIndex(s[i])]);
}
```

```
int main() {
    root = createNode();
    string s;
    //insert a string
    insertString(s);
    //delete a string
    deleteString(s);
    //string in trie
    inTrie(s);
}
```

Suffix array

- Input: string of length n
- Preprocessing: $O(n \log n)$ time and $O(n)$ space
- Requesting the matches of a pattern of length m in the string: $O(m \log n)$ time and $O(1)$ space
- Requesting the longest repeating substring: $O(n)$ time and space

```
int n, lrs, rsp, suf[100000], sra[100000], lcp[100000],
    tsu[100000], tsr[100000], tlp[100000], c[100000], phi[100000];
string s;

void countingSort(int k) {
    memset(c, 0, sizeof c);
    int mra = 0, sum = 0, tmp = 0;
    for (int i = 0; i < n; i++)
        c[i + k < n ? sra[i + k] : 0]++, mra = max(mra, i + k < n ?
            sra[i + k] : 0);
    for (int i = 0; i <= mra; i++)
        tmp = sum + c[i], c[i] = sum, sum = tmp;
    for (int i = 0; i < n; i++)
        tsu[c[suf[i] + k < n ? sra[suf[i] + k] : 0]++] = suf[i];
    for (int i = 0; i < n; i++)
        suf[i] = tsu[i];
}

void suffixArray() {
    for (int i = 0; i < n; i++) {
        suf[i] = i;
        sra[i] = s[i];
    }
    for (int k = 1; k < n; k <<= 1) {
        countingSort(k);
        countingSort(0);
        tsr[suf[0]] = 0;
        for (int i = 1; i < n; i++)
            tsr[suf[i]] = tsr[suf[i - 1]] +
                ((sra[suf[i]] == sra[suf[i - 1]] && (suf[i] + k < n
                    ? sra[suf[i] + k] : -1) ==
                  (suf[i - 1] + k < n ? sra[suf[i - 1] + k] : -1))
                    ? 0 : 1);
        for (int i = 0; i < n; i++)
            sra[i] = tsr[i];
        if (sra[suf[n - 1]] == n - 1)
            break;
    }
}
```

```
int findString(string & p, bool eql) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (l + r) / 2;
        int res = strncmp(& s.front() + suf[m], & p.front(),
            p.size());
        if (res > 0 || eql && res == 0)
            r = m;
        else
            l = m + 1;
    }
    int res = strncmp(& s.front() + suf[l], & p.front(), p.size());
    if (res < 0 || !eql && res == 0)
        l++;
    return l;
}

vector<int> findMatches(string & p) {
    int l = findString(p, true), r = findString(p, false);
    vector<int> res;
    for (int i = l; i < r; i++)
        res.push_back(suf[i]);
    return res;
}

void longestCommonPrefix() {
    phi[suf[0]] = -1;
    for (int i = 1; i < n; i++)
        phi[suf[i]] = suf[i - 1];
    int l = 0;
    for (int i = 0; i < n; i++) {
        if (phi[i] == -1) {
            tlp[i] = 0;
            continue;
        }
        while (i + l < n && phi[i] + l < n && s[i + l] == s[phi[i]
            + l])
            l++;
        tlp[i] = l;
        l = max(l - 1, 0);
    }
    for (int i = 0; i < n; i++) {
        lcp[i] = tlp[suf[i]];
        if (lcp[i] > lrs)
            lrs = lcp[i], rsp = suf[i];
    }
}

int main() {
    cin >> s;
    n = s.size();

    suffixArray();
    longestCommonPrefix();

    string p;
    //match a pattern
    vector<int> ind = findMatches(p);
    //the size of the longest repeated substring
    lrs;
    //the index of the longest repeated substring
    rsp;
}
```

String matching

- Input: string of length n and a pattern of length m
- Find the matches of the pattern in the string: $O(n + m)$ time and $O(1)$ space
- Output: the matches of the pattern in the string

```
int n, m, r[1000000];
string s;

void preprocessPattern(string & p) {
    int j = -1;
    r[0] = -1;
    for (int i = 0; i < m; i++) {
        while (j >= 0 && p[i] != p[j])
            j = r[j];
        r[i + 1] = ++j;
    }
}

vector<int> findMatches(string & p) {
    preprocessPattern(p);
    int j = 0;
    vector<int> res;
    for (int i = 0; i < n; i++) {
        while (j >= 0 && s[i] != p[j])
            j = r[j];
        j++;
        if (j == m)
            res.push_back(i - j + 1), j = r[j];
    }
    return res;
}

int main() {
    cin >> s;
    n = s.size();

    string p;
    //match a pattern
    m = p.size();
    vector<int> ind = findMatches(p);
}
```

Z-Algorithm

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
```

```
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

Dynamic programming

String alignment

- Input: strings a and b of length n and m and costs for inserting, deleting, changing and keeping characters
- Find the smallest cost to change a to b: $O(n * m)$ time and space
- Output: the smallest cost to change a to b

```
int n, m, ins, del, cha, mat, c[1000][1000];
string a, b;

int main() {
    cin >> a >> b;
    n = a.size(), m = b.size();
    //scores for inserting, deleting, changing and keeping a
        character (0, 0, -inf, 1 for longest common substring)
    cin >> ins >> del >> cha >> mat;

    for (int i = 0; i <= n; i++)
        c[i][0] = i * del;
    for (int i = 0; i <= m; i++)
        c[0][i] = i * ins;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            c[i][j] = max(max(c[i - 1][j] + del, c[i][j - 1] +
                ins), c[i - 1][j - 1] + (a[i] == b[j] ? mat : cha));

    //the costs of aligning the strings
    c[n][m];
}
```

Longest palindrome

- Input: string of length n
- Find the longest palindrome: $O(n)$ time and space
- Output: the longest palindrome

```
int n, p[2000000];
string s;

pair<int, int> findLongestPalindrome() {
    int c = 1, r = 2;
    p[0] = 1;
    p[1] = 2;
    for (int i = 2; i < 2 * n + 1; i++) {
        if (i < r)
            p[i] = min(r - i, p[2 * c - i]);
        else
            p[i] = 0;
        while (i + p[i] < 2 * n + 1 && i - p[i] >= 0 &&
                ((i + p[i]) % 2 == 0 || s[(i + p[i]) / 2] == s[(i -
                    p[i]) / 2]))
            p[i]++;
        if (i + p[i] > r) {
            c = i;
            r = i + p[i];
        }
    }
    int l = -1, s = 0;
    for (int i = 0; i < 2 * n + 1; i++) {
        p[i] /= 2;
        if (i % 2 == 0 && 2 * p[i] > s || i % 2 == 1 && 2 * p[i] -
            1 > s)
            s = 2 * p[i] - (i % 2 ? 1 : 0), l = (i + 1) / 2 - p[i];
    }
    return {l, s};
}

int main() {
    cin >> s;
    n = s.size();

    pair<int, int> pal = findLongestPalindrome();

    //the size of the longest palindrome
    pal.second;
    //the index of the longest palindrome
    pal.first;
}
```

Mathematics

Theorems

Fibonacci numbers

- Definition:
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_i = f_{i-1} + f_{i-2}$
- Calculation:
  - Dynamic programming: $O(n)$
  - Fast matrix exponentiation: $O(\log n)$
  - $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$

Binomial coefficients

- Definition:
  - $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
- Calculation: dynamic programming: $O(n * k)$

Fast exponentiation

- $x^n = \begin{cases} x & \text{if } n = 1 \\ (x^{\frac{n}{2}})^2 & \text{if } 2 \mid n \\ x^{n-1} * x & \text{otherwise} \end{cases}$

Euler's theorem

For any planar graph, $V - E + F = 1 + C$, where V is the number of graph's vertices, E is the number of edges, F is the number of faces in graph's planar drawing, and C is the number of connected components. Corollary: $V - E + F = 2$ for a 3D polyhedron.

Vertex covers and independent sets

Let M, C, I be a max matching, a min vertex cover, and a max independent set. Then $|M| \leq |C| = N - |I|$, with equality for bipartite graphs. Complement of an MVC is always a MIS, and vice versa. Given a bipartite graph with partitions (A, B), build a network: connect source to A, and B to sink with edges of capacities, equal to the corresponding nodes' weights, or 1 in the unweighted case. Set capacities of the original graph's edges to the infinity. Let (S, T) be a minimum $s - t$ cut. Then a maximum(-weighted) independent set is $I = (A \cap S) \cup (B \cap T)$, and a minimum(-weighted) vertex cover is $C = (A - T) \cup (B \cap S)$.

2-SAT

Build an implication graph with 2 vertices for each variable - for the variable and its inverse; for each clause $x \vee y$ add edges $(\neg x, y)$ and $(\neg y, x)$. The formula is satisfiable if x and $\neg x$ are in distinct SCCs, for all x. To find a satisfiable assignment, consider the graph's SCCs in topological order from sinks to sources (i.e. Kosaraju's last step), assigning 'true' to all variables of the current SCC (if it hasn't been previously assigned 'false'), and 'false' to all inverses.

Pick's theorem

$I = A - B/2 + 1$, where A is the area of a lattice polygon, I is number of lattice points inside it, and B is number of lattice points on the boundary. Number of lattice points minus one on a line segment from $(0, 0)$ and (x, y) is $gcd(x, y)$.

Combinatorics

$\sum_{k=0}^{n} k = n(n+1)/2$ $\qquad$ $\sum_{k=a}^{b} k = (a+b)(b-a+1)/2$

$\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$ $\qquad$ $\sum_{k=0}^{n} k^3 = n^2(n+1)^2/4$

$\sum_{k=0}^{n} k^4 = (6n^5 + 15n^4 + 10n^3 - n)/30$ $\qquad$ $\sum_{k=0}^{n} k^5 = (2n^6 + 6n^5 + 5n^4 - n^2)/12$

$\sum_{k=0}^{n} x^k = (x^{n+1} - 1)/(x - 1)$ $\qquad$ $\sum_{k=0}^{n} kx^k = (x - (n+1)x^{n+1} + nx^{n+2})/(x-1)^2$

$\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$ $\qquad$ $\sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m}$

$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2*n}{n}$ $\qquad$ $\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$

Fibonacci's number: $\qquad$ Catalan's number:

$\sum_{k=0}^{n} \binom{n-k}{k} = F_{n+1}$ $\qquad$ $c_n = \sum_{k=0}^{n-1} c_k\, c_{n-1-k} = \frac{1}{n+1}\binom{2n}{n}$

Burnside's Lemma

$ClassesCount = \frac{1}{|G|} \sum_{g \in G} |X^g|$,
G: group of operations(invariant permutations)
X^g: set of fixed points for operation g, i.e. $X^g = \{x \in X : g.x = x\}$
special case: $ClassesCount = \frac{1}{|G|} \sum_{g \in G} k^{c(g)}$,
k: "number of colors"
$c(g)$: number of cycles in permutation g

Multinomial coefficients

$(x_1 + x_2 + ... + x_m)^n = \sum_{k_1+k_2+...+k_m=n, k_i \geq 0} \binom{n}{k_1, k_2, ..., k_m} x_1^{k_1} ... x_m^{k_m}$, where $\binom{n}{k_1, k_2, ..., k_m} = \frac{n!}{k_1! k_2! ... k_m!}$
in combinatorial sense $\binom{n}{k_1, ..., k_m}$ is equal to the number of ways of depositing n distinct objects into m distinct bins, with k_1 objects in the first bin, k_2 objects in the second bin ...

Gray's code

direct: $G(n) = n \oplus (n >> 1)$
recurrent: $G(n) = 0G(n-1) \cup 1G(n-1)^R$ and $G(n)^R = 1G(n-1) \cup 0G(n-1)^R$

Game theory

Grundy's function

For all transitions $v- > v_i$ compute the Grundy's function $f(v_i)$:
1. $v- > v_i$ transtion into one game, then compute $f(v_i)$ recursively
$v- > v_i$ transition into sum of several games, compute f for each game and take $\oplus$ sum of their values
2. $f(v) = mex\{f(v_1), ..., f(v_k)\}$ (mex returns minimal number not contained in the set)

Algebra

Euler's function

$\phi(n) = n(1 - \frac{1}{p_1})...(1 - \frac{1}{p_k})$ where $n = p_1^{a_1}...p_k^{a_k}$
Theorem: $a^{\phi(m)} = 1 \ (mod\, m)$ if $gcd(a, m) = 1$

Prime numbers

- Input: number n
- Find all prime numbers $p \leq n$: $O(n)$ time and space
- Output: all prime numbers $p \leq n$

```
int n, prv[1000000], nxt[1000000], dvs[1000000];
vector<int> p;
bitset<1000000> ip;

bool isPrime(int x) {
    return ip[x];
}

vector<pair<int, int>> getdvsisors(int x) {
    vector<pair<int, int>> d;
    int y = dvs[x], a = 1;
    x /= dvs[x];
    while (x > 1) {
        if (dvs[x] != y) {
            d.push_back({y, a});
            y = dvs[x];
            a = 0;
        }
        x /= dvs[x], a++;
    }
    d.push_back({y, a});
    return d;
}

int main() {
    cin >> n;

    for (int i = 2; i <= n; i++)
        prv[i] = i - 1, nxt[i] = i + 1, dvs[i] = i;
    vector<int> del;
    for (int i = 2; i <= n; i = nxt[i]) {
        p.push_back(i);
        ip[i] = true;
        for (int j = i; i * j <= n; j = nxt[j])
            dvs[i * j] = i, del.push_back(i * j);
        for (int j : del)
            nxt[prv[j]] = nxt[j], prv[nxt[j]] = prv[j];
        del.clear();
    }

    //the prime numbers
    p;

    int x;
    //is prime
    isPrime(x);
    //divisors
    vector<pair<int, int>> d = getdvsisors(x);
    for (int i = 0; i < d.size(); i++)
        d[i]; //d[i].first ^ d[i].second
}
```

GCD, LCM and extended Euclid

- Find the greatest common divisor of a and b
- Find the lowest common multiple of a and b
- Solve the linear Diophantine equation $a * x + b * y = s$

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

int lowestCommonMultiple(int a, int b) {
    return a * b / greatestCommonDivisor(a, b);
}

pair<int, int> extendedEuclid(int a, int b) {
    if (b == 0)
        return {1, 0};
    pair<int, int> p = extendedEuclid(b, a % b);
    return {p.second, p.first - (a / b) * p.second};
}

pair<int, int> solveLinearDiophantineEquation(int a, int b, int s) {
    pair<int, int> p = extendedEuclid(a, b);
    int x = p.first, y = p.second, d = greatestCommonDivisor(a, b);
    if (s % d)
        return {0, 0};
    x *= s / d, y *= s / d;
    int dx = b / d, dy = - (a / d), n = 0;
    if (x < 0)
        n = - (x - dx + 1) / dx;
    if (y < 0)
        n = - (y + dy + 1) / dy;
    return {x + n * dx, y + n * dy};
}

int main() {
    int a, b, s;
    //the greatest common divisor
    greatestCommonDivisor(a, b);
    //the lowest common multiple
    lowestCommonMultiple(a, b);
    //solution a * p.first + b * p.second = s, no solution if
        p.first == 0 && p.second == 0 && s != 0
    pair<int, int> p = solveLinearDiophantineEquation(a, b, s);
}
```

Rabin Miller

```
bool Miller (LL p, LL s, int a) {
    if (p == a) return 1;
    LL mod = expmod(a, s, p); //a^s
    for (; s - p + 1 && mod - 1 && mod - p + 1; s *= 2) mod =
        mulmod(mod, mod, p); //mod^2
    return mod == p - 1 || s % 2;
}
bool isprime(LL n) {
    if (n < 2) return 0; if(n % 2 == 0) return n == 2;
    LL s = n - 1;
    while (s % 2 == 0) s /= 2;
    return Miller(n, s, 2) && Miller(n, s, 7) && Miller(n, s, 61);
} //for 341*10^12 primes <= 17
```

Fast Fourier Transformation

```
struct cpx {
  cpx(){}
  cpx(double aa):a(aa){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const {
    return a * a + b * b;
  }
  cpx bar(void) const {
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b) {
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b) {
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b) {
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta) {
  return cpx(cos(theta),sin(theta));
}

const double two_pi = 4 * acos(0);

// in:     input array
// out:    output array
// step:   {SET TO 1} (used internally)
// size:   length of the input/output {MUST BE A POWER OF 2}
// dir:    either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i
    * j * k / size)
void FFT(cpx *in, cpx *out, int step, int size, int dir) {
  if(size < 1) return;
  if(size == 1) {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++) {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / size) * odd;
    out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) /
        size) * odd;
  }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise
    product).
// To compute h[] in O(N log N) time, do the following:
//   1. Compute F and G (pass dir = 1 as the argument).
//   2. Get H by element-wise multiplying F and G.
//   3. Get h by taking the inverse FFT (use dir = -1 as the
    argument)
//      and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.

int main(void) {
  printf("If rows come in identical pairs, then everything
      works.\n");

  cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};
  cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};
  cpx A[8];
  cpx B[8];
  FFT(a, A, 1, 8, 1);
  FFT(b, B, 1, 8, 1);

  for(int i = 0 ; i < 8 ; i++)
    printf("%7.2lf%7.2lf", A[i].a, A[i].b);
  printf("\n");
  for(int i = 0 ; i < 8 ; i++) {
    cpx Ai(0,0);
    for(int j = 0 ; j < 8 ; j++)
      Ai = Ai + a[j] * EXP(j * i * two_pi / 8);
    printf("%7.2lf%7.2lf", Ai.a, Ai.b);
  }
  printf("\n");

  cpx AB[8];
  for(int i = 0 ; i < 8 ; i++)
    AB[i] = A[i] * B[i];
  cpx aconvb[8];
  FFT(AB, aconvb, 1, 8, -1);
  for(int i = 0 ; i < 8 ; i++)
    aconvb[i] = aconvb[i] / 8;
  for(int i = 0 ; i < 8 ; i++)
    printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);
  printf("\n");
  for(int i = 0 ; i < 8 ; i++) {
    cpx aconvbi(0,0);
    for(int j = 0 ; j < 8 ; j++)
      aconvbi = aconvbi + a[j] * b[(8 + i - j) % 8];
    printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);
  }
  printf("\n");

  return 0;
}
```

Geometry

```
const long double inf = 1e100, eps = 1e-12;

struct point {
    long double x, y;
    bool operator<(point & b) {
        if (x != b.x)
            return x < b.x;
        return y < b.y;
    }
    bool operator==(point b) {
        return x == b.x && y == b.y;
    }
    point operator+(point b) {
        return {x + b.x, y + b.y};
    }
    point operator-(point b) {
        return {x - b.x, y - b.y};
    }
    point operator*(long double b) {
        return {x * b, y * b};
    }
    point operator/(long double b) {
        return {x / b, y / b};
    }
    long double length() {
        return sqrt(x * x + y * y);
    }
};

long double dot(point a, point b) {
    return a.x * b.x + a.y * b.y;
}

long double cross(point a, point b) {
    return a.x * b.y - a.y * b.x;
}

long double dist(point a, point b) {
    point d = a - b;
    return d.length();
}

long double angle(point a, point b) {
    return acos(dot(a, b) / a.length() / b.length());
}

bool collinear(point a, point b) {
    return fabs(cross(a, b)) < eps;
}

bool collinear(point p, point a, point b) {
    return collinear(a - p, b - p);
}

bool ccw(point a, point b) {
    return cross(a, b) > 0;
}

bool ccw(point p, point a, point b) {
    return ccw(a - p, b - p);
}

point rotateCCW90(point p) {
    return {-p.y, p.x};
}

point rotateCW90(point p) {
    return {p.y, -p.x};
}

point rotateCCW(point p, double t) {
    return {p.x * cos(t) - p.y * sin(t), p.x * sin(t) + p.y *
        cos(t)};
}

point projectPointLine(point a, point b, point c) {
    long double r = dot(b - a, b - a);
    if (fabs(r) < eps)
        return a;
    r = dot(c - a, b - a) / r;
    return a + (b - a) * r;
}

long double distancePointLine(point a, point b, point c) {
    return dist(c, projectPointLine(a, b, c));
}

point projectPointSegment(point a, point b, point c) {
    long double r = dot(b - a, b - a);
    if (fabs(r) < eps)
        return a;
    r = dot(c - a, b - a) / r;
    if (r < 0)
        return a;
    if (r > 1)
        return b;
    return a + (b - a) * r;
}

long double distancePointSegment(point a, point b, point c) {
    return dist(c, projectPointSegment(a, b, c));
}

bool linesParallel(point a, point b, point c, point d) {
    return fabs(cross(b - a, c - d)) < eps;
}

bool linesCollinear(point a, point b, point c, point d) {
    return linesParallel(a, b, c, d) && fabs(cross(a - b, a - c)) <
        eps && fabs(cross(c - d, c - a)) < eps;
}

bool segmentsIntersect(point a, point b, point c, point d) {
    if (linesCollinear(a, b, c, d)) {
        if (dist(a, c) < eps || dist(a, d) < eps || dist(b, c) <
            eps || dist(b, d) < eps )
            return true;
        if (dot(c - a, c - b) > 0 && dot(d - a, d - b) > 0 && dot(c
            - b, d - b) > 0)
            return false;
        return true;
    }
    if (cross(d - a, b - a) * cross(c - a, b - a) > 0)
        return false;
    if (cross(a - c, d - c) * cross(b - c, d - c) > 0)
        return false;
    return true;
}

point computeLineIntersection(point a, point b, point c, point d) {
    b = b - a, d = c - d, c = c - a;
    if (dot(b, b) < eps || dot(d, d) < eps)
        return a;
    return a + b * cross(c, d) / cross(b, d);
}

point computeCircleCenter(point a, point b, point c) {
    b = (a + b) /2;
    c = (a + c) /2;
    return computeLineIntersection(b, b + rotateCW90(a-b), c, c +
        rotateCW90(a - c));
}

bool pointInPolygon(vector<point> & p, point q) {
    bool c = false;
    for (int i = 0; i < p.size(); i++) {
        int j = (i + 1) % p.size();
        if ((p[i].y <= q.y && q.y < p[j].y || p[j].y <= q.y && q.y
            < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) /
                (p[j].y - p[i].y))
            c = !c;
    }
    return c;
}

bool pointOnPolygon(vector<point> & p, point q) {
    for (int i = 0; i < p.size(); i++)
        if (dist(projectPointSegment(p[i], p[(i + 1) % p.size()],
            q), q) < eps)
            return true;
    return false;
}
```

```
vector<point> circleLineIntersection(point a, point b, point c,
    double r) {
    vector<point> ret;
    b = b - a;
    a = a - c;
    double A = dot(b, b);
    double B = dot(a, b);
    double C = dot(a, a) - r * r;
    double D = B * B - A * C;
    if (D < -eps)
        return ret;
    ret.push_back(c + a + b * (-B + sqrt(D + eps)) / A);
    if (D > eps)
        ret.push_back(c + a + b * (-B - sqrt(D)) / A);
    return ret;
}

vector<point> circleCircleIntersection(point a, point b, double r,
    double R) {
    vector<point> ret;
    double d = dist(a, b);
    if (d > r + R || d + min(r, R) < max(r, R))
        return ret;
    long double x = (d * d - R * R + r * r) / (2 * d);
    long double y = sqrt(r * r - x * x);
    point v = (b - a) / d;
    ret.push_back(a + v * x + rotateCCW90(v) * y);
    if (y > 0)
        ret.push_back(a + v * x - rotateCCW90(v) * y);
    return ret;
}

long double computeSignedArea(vector<point> & p) {
    long double area = 0;
    for (int i = 0; i < p.size(); i++) {
        int j = (i + 1) % p.size();
        area += p[i].x * p[j].y - p[j].x * p[i].y;
    }
    return area / 2.0;
}

long double computeArea(vector<point> & p) {
    return fabs(computeSignedArea(p));
}

point computeCentroid(vector<point> & p) {
    point c = {0 ,0};
    double scale = 6.0 * computeSignedArea(p);
    for (int i = 0; i < p.size(); i++) {
        int j = (i + 1) % p.size();
        c = c + (p[i] + p[j]) *(p[i].x * p[j].y - p[j].x * p[i].y);
    }
    return c / scale ;
}

bool isSimple(vector<point> & p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i + 1; k < p.size(); k++) {
            int j = (i + 1) % p.size();
            int l = (k + 1) % p.size();
            if (i == l || j == k)
                continue;
            if (segmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

int n;
point s = {1000000000, 1000000000};
point p[100000];

bool comp(point & a, point & b) {
    if (a == s)
        return true;
    if (b == s)
        return false;
    if (collinear(s, a, b))
        return dist(a, s) < dist(b, s);
    return ccw(s, a, b);
}

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> p[i].x >> p[i].y;
        if (p[i] < s)
            s = p[i];
    }
    sort(p, p + n, comp);
    p[n] = s;

    vector<int> res;
    res.push_back(0);
    res.push_back(1);
    for (int i = 2; i <= n; i++) {
        while (res.size() >= 2 && ccw(p[res[res.size() - 2]], p[i],
            p[res[res.size() - 1]]))
            res.pop_back();
        if (i != n)
            res.push_back(i);
    }

    //the convex hull
    for (int i : res)
        //(p[i].x , p[i])
}
```

Extra Graph

MinCostMaxFlow

```
// Implementation of min cost max flow algorithm using adjacency
// matrix. This implementation keeps track of forward and reverse
// edges separately (so you can set cap[i][j] != cap[j][i]).  For
// a regular max flow, set all edge costs to 0.
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
// INPUT:
//     - graph, constructed using AddEdge()
//     - source, sink
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at positive values only.

typedef vector<int> VI; typedef vector<VI> VVI; typedef long long L;
typedef vector<L> VL; typedef vector<VL> VVL; typedef pair<int,
    int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
  int N; VVL cap, flow, cost; VI found;
  VL dist, pi, width; VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
  }

  void Relax(int s, int k, L cap, L cost, int dir) {
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }

  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;

    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
        if (best == -1 || dist[k] < dist[best]) best = k;
      }
      s = best;
    }

    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};
```

PushRelabel

```
// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic.  This implementation is
// significantly faster than straight Ford-Fulkerson.  It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
// Running time:
//     O(|V|^3)
// INPUT:
//     - graph, constructed using AddEdge()
//     - source, sink
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).

typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel {
  int N; vector<vector<Edge> > G; vector<LL> excess;
  vector<int> dist, active, count; queue<int> Q;

  PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N),
      count(2*N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
  G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
}

void Enqueue(int v) {
  if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v);
      }
}

void Push(Edge &e) {
  int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
  if (dist[e.from] <= dist[e.to] || amt == 0) return;
  e.flow += amt;
  G[e.to][e.index].flow -= amt;
  excess[e.to] += amt;
  excess[e.from] -= amt;
  Enqueue(e.to);
}

void Gap(int k) {
  for (int v = 0; v < N; v++) {
    if (dist[v] < k) continue;
    count[dist[v]]--;
    dist[v] = max(dist[v], N+1);
    count[dist[v]]++;
    Enqueue(v);
  }
}

void Relabel(int v) {
  count[dist[v]]--;
  dist[v] = 2*N;
  for (int i = 0; i < G[v].size(); i++)
    if (G[v][i].cap - G[v][i].flow > 0)
  dist[v] = min(dist[v], dist[G[v][i].to] + 1);
  count[dist[v]]++;
  Enqueue(v);
}

void Discharge(int v) {
  for (int i = 0; excess[v] > 0 && i < G[v].size(); i++)
      Push(G[v][i]);
  if (excess[v] > 0) {
    if (count[dist[v]] == 1)
  Gap(dist[v]);
    else
  Relabel(v);
  }
}

LL GetMaxFlow(int s, int t) {
  count[0] = N-1;
  count[N] = 1;
  dist[s] = N;
  active[s] = active[t] = true;
  for (int i = 0; i < G[s].size(); i++) {
    excess[s] += G[s][i].cap;
    Push(G[s][i]);
  }
    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
      active[v] = false;
      Discharge(v);
    }

    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
  }
};
```

MinCostMatching

```
// Min cost bipartite matching via shortest augmenting paths
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.

typedef vector<double> VD; typedef vector<VD> VVD; typedef
    vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n), v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
    Lmate[i] = j;
    Rmate[j] = i;
    mated++;
    break;
      }
    }
  }
VD dist(n); VI dad(n), seen(n);

// repeat until primal solution is feasible
while (mated < n) {
  // find an unmatched left node
  int s = 0;
  while (Lmate[s] != -1) s++;

  // initialize Dijkstra
  fill(dad.begin(), dad.end(), -1);
  fill(seen.begin(), seen.end(), 0);
  for (int k = 0; k < n; k++)
    dist[k] = cost[s][k] - u[s] - v[k];

  int j = 0;
  while (true) {
    // find closest
    j = -1;
    for (int k = 0; k < n; k++) {
  if (seen[k]) continue;
  if (j == -1 || dist[k] < dist[j]) j = k;
    }
    seen[j] = 1;

    // termination condition
    if (Rmate[j] == -1) break;

    // relax neighbors
    const int i = Rmate[j];
    for (int k = 0; k < n; k++) {
  if (seen[k]) continue;
  const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
  if (dist[k] > new_dist) {
    dist[k] = new_dist;
    dad[k] = j;
  }
    }
  }

  // update dual variables
  for (int k = 0; k < n; k++) {
    if (k == j || !seen[k]) continue;
    const int i = Rmate[k];
    v[k] += dist[k] - dist[j];
    u[i] -= dist[k] - dist[j];
  }
  u[s] += dist[j];

  // augment along path
  while (dad[j] >= 0) {
    const int d = dad[j];
    Rmate[j] = Rmate[d];
    Lmate[Rmate[j]] = j;
    j = d;
  }
  Rmate[j] = s;
  Lmate[s] = j;

    mated++;
  }

  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

  return value;
}
```

MinCut

```
// Adjacency matrix implementation of Stoer-Wagner min cut
    algorithm.
// Running time:
//     O(|V|^3)
// INPUT:
//     - graph, constructed using AddEdge()
// OUTPUT:
//     - (min cut value, nodes in half of min cut)

typedef vector<int> VI; typedef vector<VI> VVI;

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
    if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase-1) {
    for (int j = 0; j < N; j++) weights[prev][j] +=
        weights[last][j];
    for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
    used[last] = true;
    cut.push_back(last);
    if (best_weight == -1 || w[last] < best_weight) {
      best_cut = cut;
      best_weight = w[last];
    }
      } else {
    for (int j = 0; j < N; j++)
      w[j] += weights[last][j];
    added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}
```