Intro

Shortcuts

```
typedef vector<int> vi;
typedef long long ll;
typedef pair<int, int> ii;

const int UNVISITED = -1;
const int INF = 1e9;
const double EPS = 1e-9;
const double PI = acos(-1.0); // alternative (2.0 * acos(0.0))
```

Complexity

Modern CPU compute 100M in 3s.

n	Worst AC Algorithm	Problem
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations
$\leq [15..18]$	$O(2^n n^2)$	e.g. DP TSP
$\leq [18..22]$	$O(2^n n)$	e.g. DP with bitmask
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions
≤ 400	$O(n^3)$	e.g. Floyd Warshall's
$\leq 2K$	$O(n^2 logn)$	e.g. 2 loops + a tree-related DS
$\leq 10K$	$O(n^2)$	e.g. Selection/Insert sort
$\leq 1M$	$O(nlogn)$	e.g. Building Segment Tree
$\leq 100M$	$O(n)$	I/O bottleneck

Limits

32-bit int $2^{31} - 1 = 2147483647 \approx 10^{10}$

64-bit signed long long upper limit $2^{63} - 1 = 9223372036854775807 \approx 10^{18}$

Math

TODO tables of 2^x, $!x, x1..13$

TODO simple geometric formulas for volumes etc?

TODO sin/cos

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

Primes

```
// 100 first primes
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

// Some larger primes
104729 1299709 9999991 15485863 179424673 2147483647 32416190071
112272535095293 54673257461630679457

// prime sieve with prime checking
const int MAX_SIEVE = 1e7; // 1e7 in a few seconds

ll _sieve_size;
bitset<MAX_SIEVE + 10> bs;
vi primes;

void sieve(ll upperbound) {
    _sieve_size = upperbound + 1;
    bs.set();
    bs[0] = bs[1] = 0;
    for (ll i = 2; i <= _sieve_size; ++i)
        if (bs[i]) {
            for (ll j = i * i; j <= _sieve_size; j += i)
                bs[j] = 0;
            primes.push_back((int)i);
        }
}

bool isPrime(ll N) { // works for N <= (last prime in primes)^2
    if (N <= _sieve_size) return bs[N]; // O(1) sieve check for small primes
    for (int i = 0; i < (int)primes.size(); ++i) // brute force for larger
        if (N % primes[i] == 0) return false;
    return true; // more time if N is prime!
}
```

Java BigInteger

```
BigInteger.ZERO // constants
i.mod(m) // base number conversion
i.isProbablePrime(10) // Probabilistic prime testing
i.gcd(k)
x.modPow(y, n) // calculate x^y mod n

// Catalan numbers with BigInteger
import java.util.Scanner;
import java.math.BigInteger;

class Main {
    public static BigInteger[] mem;

    public static BigInteger cat(int n) {
        if (n == 0) return BigInteger.ONE;
        if (mem[n] != null) return mem[n];

        BigInteger k = BigInteger.valueOf(2 * (2 * n - 1)).multiply(cat(n -
            1));
        return mem[n] = k.divide(BigInteger.valueOf(n + 1));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        mem = new BigInteger[11]; // adjust as necessary
        while (sc.hasNextInt()) {
            System.out.println(cat(sc.nextInt()));
        }
    }
}
```

Fibonacci

$[0..15]$ $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610$

$F(0) = 0, F(1) = 1$

$F(n) = F(n-1) + F(n-2)$

Combinatorics

$C(n, 0) = C(n, n) = 1$

$C(n, k) = C(n-1, k-1) + C(n-1, k)$

Catalan numbers

$[0..10]$ $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796$

1. $Cat(n)$ Count the number of distinct binary trees with n vertices.
2. Count number of expressions counting n correctly matched pairs of parentheses.
3. Count ways a convex polygon can be triangulated.

$Cat(0) = 1$

$Cat(n) = \frac{2(2n-1)}{n+1} * Cat(n-1)$

Extended Euclid: Linear Diphantine Equation

```
int x, y, d; // answer, give d = gcd(a, b)
void extendedEuclid(int a, int b) { // solve a*x + b*y = d
    if (b == 0) { x = 1; y = 0; d = a; return; }
    extendedEuclid(b, a % b);
    int x1 = y, y1 = x - (a /b) * y;
    x = x1; y = y1;
}
```

Cycle Finding

```
// find position and length of the repeated pattern in a generated sequence
ii floydCycleFinding(int x0) { // define int f(int x) which generates the
    sequence
    // 1st phase, hare 2x speed of turtoise
```

```
    int tortoise = f(x0), hare = f(f(x0));
    while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
    // 2nd phase, find mu, same speed
    int mu = 0; hare = x0;
    while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); ++mu;
        }
    // 3rd phase, find lambda, hare moves tortoise still
    int lambda = 1; hare = f(tortoise);
    while (tortoise != hare) { hare = f(hare); ++lambda; }
    return ii(mu, lambda); // mu: start of cycle, lambda: cycle length
}
```

Game Theory

The Nim Game. Two players take turns to remove objects from distinct heaps. On each turn, a player must remove at least one object and may remove any number of objects, but only from the same heap. For the starting player to win, $n_1\hat{}...\hat{}n_k \neq 0$. (bitwise xor)

DP

LIS O$(nlogk)$

```
vi lis(vi a) {
    int L[MAX];

    vi dp(a.size());

    int lis = 0;
    for (int i = 0; i < a.size(); ++i) {
        // LIS ending at a[i] is at length pos + 1
        int pos = lower_bound(L, L + lis, a[i]) - L;
        L[pos] = a[i];
        dp[i] = pos + 1;

        if (pos + 1 > lis) {
            lis = pos + 1;
        }
    }

    return dp; // Return lis array
}
```

Data structures

Union Find

```
class UnionFind { // rank ordered with path compression
public:
    UnionFind(int n) {
        rank.assign(n, 0);
        p.assign(n, 0);
        set_size.assign(n, 1);
        num_sets = n;
        for (int i = 0; i < n; ++i)
            p[i] = i;
    }

    int find_set(int i) { return (p[i] == i) ? i : (p[i] = find_set(p[i])); }
    bool is_same_set(int i, int j) { return find_set(i) == find_set(j); }
    void union_set(int i, int j) {
        if (!is_same_set(i, j)) {
            --num_sets;
            int x = find_set(i), y = find_set(j);
            if (rank[x] > rank[y]) {
                p[y] = x;
                set_size[x] += set_size[y];
            }
            else {
                p[x] = y;
                set_size[y] += set_size[x];
                if (rank[x] == rank[y]) rank[y]++;
            }
        }
    }
    int num_disjoint_sets() { return num_sets; }
    int size_of_set(int i) { return set_size[find_set(i)]; }
private:
    vi rank, p, set_size;
    int num_sets;
};
```

Fenwick Tree

```
// Ideal to answer dynamic Range Sum Queries
struct FenwickTree {
  vi ft;
  FenwickTree() {}
  // initialization: n + 1 zeroes, ignore index 0
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  int rsq(int b) { // returns RSQ(1, b), O(log n)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum;
  }
  int rsq(int a, int b) { // returns RSQ(a, b), O(log n)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1));
  }
  // adjusts value of the k-th element by v
  void adjust(int k, int v) { // O(log n)
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v;
  }
};
```

Segment Tree

```
class SegmentTree { // Max range query. Change >= to <= for min.
    vi st, a;

    int n;
```

```
    int left(int p) { return p << 1; } // Same as binary heap
    int right(int p) { return (p << 1) + 1; }

    void build(int p, int l, int r) { // O(n log n)
        if (l == r)
            st[p] = l;
        else {
            build(left(p), l, (l + r) / 2);
            build(right(p), (l + r) / 2 + 1, r);
            int p1 = st[left(p)], p2 = st[right(p)];
            st[p] = (a[p1] >= a[p2]) ? p1 : p2; // Build max
        }
    }

    int rmq(int p, int l, int r, int i, int j) { // O(log n)
        if (i > r || j < l) return -1; // outside of range
        if (l >= i && r <= j) return st[p]; // inside range

        int p1 = rmq(left(p), l, (l + r) / 2, i, j);
        int p2 = rmq(right(p), (l + r) / 2 + 1, r, i, j);

        if (p1 == -1) return p2;
        if (p2 == -1) return p1;
        return (a[p1] >= a[p2]) ? p1 : p2; // Return max inside
    }

    // Support for dynamic updating. O(log n)
    int update_point(int p, int l, int r, int idx, int new_value) {
        int i = idx, j = idx;
        if (i > r || j < l)
            return st[p];

        if (l == i && r == j) {
            a[i] = new_value;
            return st[p] = l;
        }

        int p1, p2;
        p1 = update_point(left(p), l, (l + r) / 2, idx, new_value);
        p2 = update_point(right(p), (l + r) / 2, r, idx, new_value);

        return st[p] = (a[p1] >= a[p2]) ? p1 : p2; // Max query
    }
public:
    SegmentTree(const vi &_a) {
        a = _a; n = (int) a.size(); // Copy for local use
        st.assign(4 * n, 0); // Large enough of zeroes
        build(1, 0, n - 1);
    }

    // Return index of max O(log n)
    int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }

    // Update index to a new value.
    int update_point(int idx, int new_value) {
        return update_point(1, 0, n - 1, idx, new_value);
    }
};
```

Graph

Kruskal MST

```
// use union find class
int kruskal_mst(vector<pair<int, ii> > &EdgeList, int V) {
    int mst_cost = 0;
    UnionFind UF(V);
    for (int i = 0; i < EdgeList.size(); ++i) {
        pair<int, ii> front = EdgeList[i];
        if (!UF.isSameSet(front.second.first, front.second.second)) {
            mst_cost += front.first;
            UF.unionSet(front.second.first, front.second.second);
        }
    }

    return mst_cost;
}
```

Bipartite check

```
bool is_bipartite(int s) {
    qi q; q.push(s);
    vi color(n, INF); color[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int j = 0; j < (int)adjs[u].size(); ++j) {
            ii v = adjs[u][j];
            if (color[v.first] == INF) {
                color[v.first] = 1 - color[u];
                q.push(v.first);
            }
            else if (color[v.first] == color[u]) {
                return false;
            }
        }
    }
    return true;
}
```

Maximum Bipartite Cardinality Matching

```
vector<vi> AdjList; // initialize
vi match, vis;

int aug(int l) { // return 1 if augmenting path is found, 0 otherwise
    if (vis[l]) return 0;
    vis[l] = 1;
    for (int j = 0; j < (int)AdjList[l].size(); ++j) {
```

```
        int r = AdjList[l][j];
        if (match[r] == -1 || aug(match[r])) {
            match[r] = l;
            return 1;
        }
    }
    return 0;
}

// in main
int MCBM = 0; // result
match.assign(V, -1);
for (int l = 0; l < n; ++l) {
    vis.assign(n, 0);
    MCBM += aug(l);
}
```

Articulation points and bridges

```
void articulationPointAndBridge(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
    for (int j = 0; j < (int)AdjList[u].size(); ++j) {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == UNVISITED) {
            dfs_parent[v.first] = u;
            if (u == dfsRoot) rootChildren++;

            articulationPointAndBridge(v.first);

            if (dfs_low[v.first] >= dfs_num[u])
                articulation_vertex[u] = true;
            if (dfs_low[v.first] > dfs_num[u])
                printf(" Edge (%d,%d) is a bridge\n", u, v.first);
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
        }
        else if (v.first != dfs_parent[u]) // a back edge and not direct
            cycle
            dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
    }
}

// in main
dfsNumberCounter = 0;
dfs_num.assign(V, UNVISITED);
dfs_low.assign(V, 0);
dfs_parent.assign(V, 0);
articulation_vertex.assign(V, 0);
printf("Bridges:\n");
for (int i = 0; i < V; ++i)
    if (dfs_num[i] == UNVISITED) { // special case for root
        dfsRoot = i; rootChildren = 0;
        articulationPointAndBridge(i);
        articulation_vertex[dfsRoot] = (rootChildren > 1);
    }
// articulation_vertex contains Articulation Points
```

Dijkstra

```
vector<vector<ii> > AdjList; // pair<node, cost>
int V, E, s, t;

int dijsktra(int s, int t) { // variant will leave duplicate nodes in queue
    vi dist(V, INF);
    dist[s] = 0;
    priority_queue<ii, vector<ii>, greater<ii> > pq;
    pq.push(ii(0, s));
    while (!pq.empty()) {
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if (d > dist[u]) continue; // important check
        for (int j = 0; j < (int)AdjList[u].size(); ++j) {
            ii v = AdjList[u][j];
            if (dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second; // relax
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }
    return dist[t];
}
```

Bellman Ford

```
int bellman_ford(int s, int t) { // O(VE) when using adj list
    vi dist(V, INF); dist[s] = 0;
    for (int i = 0; i < V - 1; ++i) // relax all edges V-1 times
        for (int u = 0; u < V; ++u)
            for (int j = 0; j< (int)AdjList[u].size(); ++j) {
                ii v = AdjList[u][j]; // record SP spanning here if needed
                dist[v.first] = min(dist[v.first], dist[u] + v.second);
            }

    return dist[t];
}

// check if there exists a negative cycle
bool hasNegativeCycle = false;
for (int u = 0; u < V; ++u)
    for (int j = 0; j< (int)AdjList[u].size(); ++j) {
        ii v = AdjList[u][j];
        if (dist[v.first] > dist[u] + v.second) // if still possible
            hasNegativeCycle = true;            // then neg cycle exists
    }
```

Euler Tour

```
list<int> cyc; // list for fast insertion in middle

void EulerTour(list<int>::iterator i, int u) {
```

```
    for (int j = 0; j < (int)AdjList[u].size(); ++j) {
        ii v = AdjList[u][j];
        if (v.second) {
            v.second = 0; // mark as to be removed
            for (int k = 0; k < (int)AdjList[v.first].size(); ++k) {
                ii uu = AdjList[v.first][k]; // remove bi-directional
                if (uu.first == u && uu.second) {
                    uu.second = 0;
                    break;
                }
            }
            EulerTour(cyc.insert(i, u), v.first);
        }
    }
}

// inside main
cyc.clear();
EulerTour(cyc.begin(), A); // cyc contains euler tour starting at A
for (list<int>::iterator it = cyc.begin(); it != cyc.end(); ++it)
    printf("%d\n", *it); // the Euler tour
```

Edmond Karp

```
// setup res, s, t, AdjList as global variables
int res[MAXN][MAXN], mf, f, s, t;
vi p;
vector<vi> AdjList; // Don't forget backward edges!

void augment(int v, int minEdge) { // traverse BFS spanning tree from s to t
    if (v == s) { f = minEdge; return; } // record minEdge in a global
        variable f
    else if (p[v] != -1) {
        augment(p[v], min(minEdge, res[p[v]][v]));
        res[p[v]][v] -= f; res[v][p[v]] += f;
    }
}

int edmond_karp() {
    mf = 0;
    while (1) { // run bfs
        f = 0;
        bitset<MAXN> vis; vis[s] = true; // bitset is faster
        queue<int> q; q.push(s);
        p.assign(MAXN, -1); // record the BFS spanning tree, from s to t
        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (u == t) break; // stop bfs if we reach t
            for (int j = 0; j < (int)AdjList[u].size(); ++j) { // faster with
                AdjList
                int v = AdjList[u][j];
                if (res[u][v] > 0 && !vis[v])
                    vis[v] = true, q.push(v), p[v] = u;
            }
        }
        augment(t, INF);
        if (f == 0) break;      // we cannot send any more flow, terminate
        mf += f;                // we can still send a flow, increase the max
            flow!
    }
    return mf;
}
```

Flood Fill

```
// need grid, R, C
int dr[8] = { 1, 1, 0, -1, -1, -1, 0, 1 };
int dc[8] = { 0, 1, 1, 1, 0, -1, -1, -1 };

// Return size of CC
int floodfill(int r, int c, char c1, char c2) {
    if (r < 0 || r >= R || c < 0 || c >= C) return 0;
    if (grid[r][c] != c1) return 0;

    int ans = 1; // Because vertex (r, c) has c1 as its color
    grid[r][c] = c2; // Color it
    for (int d = 0; d < 8; ++d)
        ans += floodfill(r + dr[d], c + dc[d], c1, c2);
    return ans;
}
```

Topological Sort

```
vi ts; // Result in reverse order
void topo(int u) {
    seen[u] = 1; // Init to false
    for (int i = 0; i < (int)adj_list[u].size(); ++i) {
        ii v = adj_list[u][i];
        if (!seen[v.first])
            topo(v.first);
    }
    ts.push_back(u);
}

// use
ts.clear();
// init seen to false
for (int i = 0; i < n; ++i)
    if (!seen[i]) topo(i);
```

Strongly Connected Components

```
vi dfs_num, dfs_low, S, visited;

void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
    S.push_back(u); // stores u in a vector based on order of visitation
    visited[u] = 1;
    for (int j = 0; j < (int)AdjList[u].size(); ++j) {
```

```
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == UNVISITED)
            tarjanSCC(v.first);
        if (visited[v.first])
            dfs_low[u] - min(dfs_low[u], dfs_low[v.first]);
    }

    if (dfs_low[u] == dfs_num[u]) { // if this is a root (start) of an SCC
        printf("SCC %d:", ++numSCC); // this part is done after recursion
        while (1) {
            int v = S.back(); S.pop_back(); visited[v] = 0;
            printf(" %d", v);
            if (u == v) break;
        }
        printf("\n");
    }
}

// in main
dfs_num.assign(V, UNVISITED);
dfs_low.assign(V, 0);
visited.assign(V, 0);
dfsNumberCounter = numSCC = 0;
for (int i = 0; i < V; ++i)
    if (dfs_num[i] == UNVISITED)
        tarjanSCC(i);
```

Chinese Postman

```
// Weight of euler tour in connected graph.
// Need to fill d[][] with min cost between any two nodes. Do floyd warshall
    before.
int memo[1 << MAX]; // dp bitmask memo structure

// Min cost of increasing by one the degree of set of the given odd vertices,
     to make them even.
int min_cost(int s) {
    if (s == 0) return 0;
    if (memo[s] != 0) return memo[s];

    int best = -1;

    int x = 0; // Choose our first node to switch as the first node with odd
        values we can find.
    while (((s >> x) & 1) == 0) ++x; // x = number of trailing zeros

    // Try to combine with all other odd value nodes.
    for (int y = x + 1; y < n; ++y) {
        if ((s >> y) & 1 == 0) continue;

        int comb = s ^ (1 << x) ^ (1 << y); // Switch off the selected nodes.

        // Cost will be to combine these two nodes + combining the rest.
        int cost = d[x][y] + min_cost(comb);
        if (best == -1 || cost < best)
            best = cost;
    }

    return memo[s] = best;
}
```

String

Knuth-Morris-Pratt

```
int b[MAXN]; // back table
void kmpPreprocess(string P) {
    int i = 0, j = -1; b[0] = -1;
    while (i < P.size()) {
        while (j >= 0 && P[i] != P[j]) j = b[j];
        ++i; ++j;
        b[i] = j;
    }
}

void kmpSearch(string T, string P) { // does P match T?
    kmpPreprocess(P); // must prepare P
    int i = 0, j = 0;
    while (i < T.size()) {
        while (j >= 0 && T[i] != P[j]) j = b[j];
        ++i; ++j;
        if (j == P.size()) {
            printf("P is found at index %d in T\n", i - j);
            j = b[j]; // prepare for next possible match
        }
    }
}
```

Edit Distance

```
vector<vi> dp;
int edit_distance(string A, string B) { // align A with B
    dp.assign((int)A.size() + 1, vi()); // dynamic dp matrix
    for (int i = 0; i <= A.size(); ++i)
        dp[i].assign((int)B.size() + 1, 0);

    for (int i = 1; i <= A.size(); ++i)
        dp[i][0] = i * -1; // delete substring A[1..i], score -1
    for (int i = 1; i <= B.size(); ++i)
        dp[0][i] = i * -1; // insert space in B[1..i], score -1

    for (int i = 1; i <= A.size(); ++i)
        for (int j = 1; j <= B.size(); ++j) {
            // Match +2, Mismatch -1
            dp[i][j] = dp[i - 1][j - 1] + (A[i - 1] == B[j - 1] ? 2 : -1);
            dp[i][j] = max(dp[i][j], dp[i - 1][j] - 1); // delete
            dp[i][j] = max(dp[i][j], dp[i][j - 1] - 1); // insert
        }
```

```
    return dp[A.size()][B.size()]; // max alignment score
}
```

Longest Common Subsequence

```
vector<vi> dp;
int lcs(string A, string B) { // turn edit distance into lcs
    dp.assign((int)A.size() + 1, vi()); // dynamic dp matrix
    for (int i = 0; i <= A.size(); ++i)
        dp[i].assign((int)B.size() + 1, 0); // all edge cases 0

    for (int i = 1; i <= A.size(); ++i)
        for (int j = 1; j <= B.size(); ++j) {
            // Match 1, Mismatch -INF
            dp[i][j] = dp[i - 1][j - 1] + (A[i - 1] == B[j - 1] ? 1 : -INF);
            dp[i][j] = max(dp[i][j], dp[i - 1][j]); // delete cost 0
            dp[i][j] = max(dp[i][j], dp[i][j - 1]); // insert cost 0
        }

    return dp[A.size()][B.size()]; // max alignment score
}
```

Suffix Array

```
// Suffix Array is a simpler version of Suffix Tree.
// It is slower to construct, O(n log n) vs O(n)
// but it's a lot simpler to program.

// ex. find all Longest Common Substrings of a and b, O(n log n)
string T = a + "$" + b + "#"; // Chars lower, combine input strings
n = T.size(); m = b.size(); // for ease of programming
constructSA(T); // Construct Suffix Array
computeLCP(T); // LCS depends on LCP, so must do this

set<string> res = allLCS(T); // Can also use LCS()
if (res.empty()) printf("No common sequence.\n");
for (set<string>::iterator i = res.begin(); i != res.end(); ++i) {
    printf("%s\n", i->c_str());
}

// ex. find Longest Repeated Substring (min 2 times), O(n log n)
T += "$"; // input string T, append '$'
n = T.size(); // for ease of programming
constructSA(T); // Construct Suffix Array
computeLCP(T); // LRS depends on LCP

pair<string, int> ans = LRS(T); // LRS string and #repetitions
if (ans.first.size()) printf("%s %d\n", ans.first.c_str(), ans.second);
else printf("No repetitions found!\n");

// impl
const int MAXN = 100010; // ok up to ~100k
int RA[MAXN], tmpRA[MAXN]; // rank array + tmp
int SA[MAXN], tmpSA[MAXN]; // suffix array + tmp
int c[MAXN]; // freq table for counting sort
int n, m; // globals for T and P
int Phi[MAXN]; // for computing longest common prefix
int PLCP[MAXN];
int LCP[MAXN]; // LCP[i] stores the LCP between previous suffix T + SA[i-1]
                                                    // and current suffix T + SA[i]

void countingSort(int k, int n) { // sort RA, res in SA
    int sum, maxi = max(300, n); // up to 255 ASCII chars of length n
    memset(c, 0, sizeof c);
    for (int i = 0; i < n; ++i) // count freq of each integer rank
        c[i + k < n ? RA[i + k] : 0]++;
    for (int i = sum = 0; i < maxi; ++i) {
        int t = c[i]; c[i] = sum; sum += t;
    }
    for (int i = 0; i < n; ++i) // shuffle suffix array if necessary
        tmpSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];
    for (int i = 0; i < n; ++i) // update suffix array
        SA[i] = tmpSA[i];
}

void constructSA(string &T) { // Construct Suffix Array in O(n log n)
    int n = T.size();
    for (int i = 0; i < n; ++i) RA[i] = T[i];
    for (int i = 0; i < n; ++i) SA[i] = i;
    for (int k = 1; k < n; k <<= 1) { // repeat sort log n times
        countingSort(k, n); // radix sort
        countingSort(0, n); // stable sort on first item
        int r = 0; tmpRA[SA[0]] = 0; // re-rank from rank r = 0
        for (int i = 1; i < n; ++i) {
            // if same pair => r otherwise increase rank
            if (RA[SA[i]] == RA[SA[i - 1]] && RA[SA[i] + k] == RA[SA[i - 1] +
                k])
                tmpRA[SA[i]] = r;
            else
                tmpRA[SA[i]] = ++r;
        }
        for (int i = 0; i < n; ++i) // update rank array
            RA[i] = tmpRA[i];
        if (RA[SA[n - 1]] == n - 1) break; // optimization
    }
}

void computeLCP(string &T) { // Longest Common Prefix, O(n)
    Phi[SA[0]] = -1;
    for (int i = 1; i < n; ++i)
        Phi[SA[i]] = SA[i - 1];
    for (int i = 0; i < n; ++i) {
        int L = 0;
        if (Phi[i] == -1) { PLCP[i] = 0; continue; }
        while (T[i + L] == T[Phi[i] + L]) ++L;
        PLCP[i] = L;
```

```
        L = max(L - 1, 0);
    }
    for (int i = 0; i < n; ++i)
        LCP[i] = PLCP[SA[i]];
}

int owner(int idx) { return (idx < n - m - 1) ? 1 : 2; }

// Longest Common Substring in O(n)
ii LCS() { // return <LCS length, index>, where SA[index] gives index in T
    int idx = 0, maxLCP = -1;
    for (int i = 1; i < n; ++i)
        if (owner(SA[i]) != owner(SA[i - 1]) && LCP[i] > maxLCP)
            maxLCP = LCP[i], idx = i;
    return ii(maxLCP, idx);
}

set<string> allLCS(string &T) { // return all unique longest substrings O(n
    log n)
    int maxLCP = -1;
    set<string> res;
    for (int i = 0; i < n; ++i) {
        if (owner(SA[i]) == owner(SA[i - 1])) continue;
        if (LCP[i] == 0) continue;
        if (LCP[i] > maxLCP) res.clear();
        if (LCP[i] >= maxLCP) {
            maxLCP = LCP[i];
            res.insert(T.substr(SA[i], maxLCP));
        }
    }
    return res;
}

// Longest Repeated Substring (substring 2 times or more)
ii LRS() { // returns <LRS length, index>, where SA[index] gives index in T
    int idx = 0, maxLCP = -1;
    for (int i = 1; i < n; i++)
        if (LCP[i] > maxLCP)
          maxLCP = LCP[i], idx = i;
    return ii(maxLCP, idx);
}

pair<string, int> LRS(string &T) { // return LRS and #repetitions
    int maxLCP = -1, rep = 0;
    string s;
    for (int i = 1; i < n; i++) {
        string curr = T.substr(SA[i], LCP[i]);
        if (LCP[i] > maxLCP) {
            maxLCP = LCP[i]; rep = 2;
            s = curr;
        }
        else if (s == curr) ++rep;
    }
    return make_pair(s, rep);
}
```

Geometry

Points and Lines

```
struct point_i { // prefer
    int x, y;
    point_i() { x = y = 0; }
    point_i(int _x, int _y) : x(_x), y(_y) { }
};
struct point { // only if double needed, prefer ints
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) { }
    bool operator < (point other) const {
        if (fabs(x - other.x) > EPS) // EPS comparison!
            return x < other.x;
        return y < other.y;
    }
    bool operator == (point other) const { // EPS comparison
        return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS));
    }
};
// Euclidian distance
double dist(point p1, point p2) { return hypot(p1.x - p2.x, p1.y - p2.y); }

// A vector is not a point here
struct vec { double x, y; vec(double _x, double _y) : x(_x), y(_y) { } };

vec toVec(point a, point b) { return vec(b.x - a.x, b.y - a.y); }
vec scale(vec v, double s) { return vec(v.x * s, v.y * s); }

// Move a point
point translate(point p, vec v) { return point(p.x + v.x, p.y + v.y); }

double dot(vec a, vec b) { return a.x * b.x + a.y * b.y; }
double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

// Closest point to the line defined by a and b (must be different!)
double distToLine(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    c = translate(a, scale(ab, u));
    return dist(p, c);
}

// Closest point to line segment between a and b (OK if a == b)
double distToLineSegment(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    if (u < 0.0) { c = point(a.x, a.y); return dist(p, a); }
    if (u > 1.0) { c = point(b.x, b.y); return dist(p, b); }
```

```
    return distToLine(p, a, b, c);
}

// ax + by + c = 0, b = 0.0 if vertical, 1.0 otherwise
struct line { double a, b, c; };

void pointsToLine(point p1, point p2, line &l) {
    if (fabs(p1.x - p2.x) < EPS) { // special for vertical
        l.a = 1.0; l.b = 0.0; l.c = -p1.x;
    }
    else {
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0;
        l.c = -(double)(l.a * p1.x) - p1.y;
    }
}

bool areParallel(line l1, line l2) { // check a & b
    return (fabs(l1.a - l2.a) < EPS) && (fabs(l1.b - l2.b) < EPS);
}

bool areSame(line l1, line l2) { // check c
    return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPS);
}

// Check lines, not line segments
bool areIntersect(line l1, line l2, point &p) {
    if (areParallel(l1, l2)) return false;
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
    if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
    else                  p.y = -(l2.a * p.x + l2.c);
    return true;
}
```

Circles

```
// 2: inside, 1: border, 0: outside. Exakt int calc
int insideCircle(point_i p, point_i c, int r) {
    int dx = p.x - c.x, dy = p.y - c.y;
    int Euc = dx * dx + dy * dy, rSq = r * r;
    return Euc < rSq ? 2 : Euc == rSq ? 1 : 0;
}

// Given to points p1, p2 and the radius of a circle.
// Return if there can be a circle with the given radius and
// if so return it's center. To get both possible centers,
// call again with p1 and p2 swapped.
bool circle2PtsRad(point p1, point p2, double r, point &c) {
    double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
                (p1.y - p2.y) * (p1.y - p2.y);
    double det = r * r / d2 - 0.25;
    if (det < 0.0) return false;
    double h = sqrt(det);
    c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
    c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
    return true;
}
```

Convex Hull

TODO