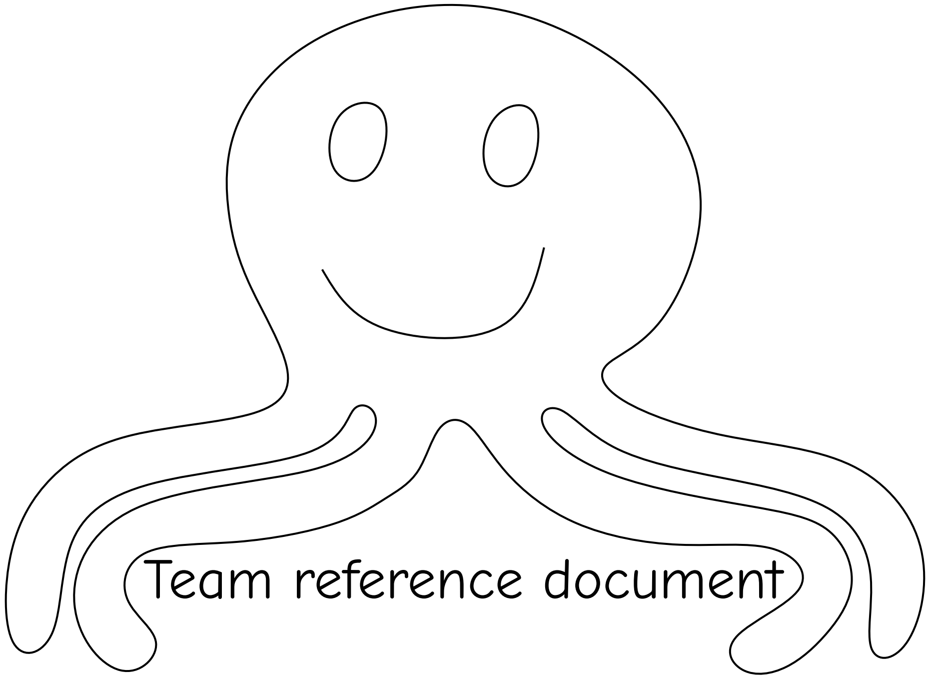

Team reference document

Kiel University

05. July 2025

Contents

Template

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<vi> vvi;
typedef vector<vii> vvii;
#define fi first
#define se second
#define eb emplace_back
#define pb push_back
#define mp make_pair
#define mt make_tuple
#define endl '\n'
#define ALL(x) (x).begin(), (x).end()
#define RALL(x) (x).rbegin(), (x).rend()
#define SZ(x) (int)(x).size()
#define FOR(a, b, c) for (auto a = (b); (a) < (c); ++(a))
#define F0R(a, b) FOR (a, 0, (b))
template <typename T>
bool ckmin(T& a, const T& b) { return a > b ? a = b, true : false; }
template <typename T>
bool ckmax(T& a, const T& b) { return a < b ? a = b, true : false; }
#ifndef DEBUG
#define DEBUG 0
#endif
#define dout if (DEBUG) cerr
#define dvar(...) " [" « #__VA_ARGS__ ": " « (__VA_ARGS__) « "] "
```

.vimrc

```
map 0 ^
sy on
```

Code icon by Font Awesome

Math

Geometric Sum

Geometric Sum for $q \neq 1$:

$$\sum_{k=1}^{n} q^k = \frac{1-q^{n+1}}{1-q}$$

For $|q| < 1$ the corresponding infinite sum converges:

$$\sum_{i=1}^{\infty} q^k = \frac{1}{1-q}$$

Picks Theorem

For every polygon with integer-only coordinates with following holds for the area A, amount of interior integer points i and amount of boundary points b:

$$A = i + \frac{b}{2} - 1$$

Calculation of border points: $\mathcal{O}(b \cdot \log(\text{maxVal}))$

```
ll onEdge(vii pts) { // Assume pts[0] = pts[SZ(pts) - 1]
  ll res = SZ(pts);
  F0R (i, SZ(pts) - 1)
    res += abs(gcd(pts[i].fi - pts[i + 1].fi, pts[i].se - pts[i + 1].se));
  return res;
}
```

For polygon area see the geometry section.

Eulers formular for planar graphs

For every planar graph G the following holds: $V - E + F = 2$

Combinatorics

Binomial coefficient

Number of possible sets with k elements selected from n elements.

$$\binom{n}{k} = \binom{n}{n-k} = \frac{n!}{k!(n-k)!} = \frac{n}{k}\binom{n-1}{k-1} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

n \ k	0	1	2	3	4	5	6	7	8	9
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1			
7	1	7	21	35	35	21	7	1		
8	1	8	28	56	70	56	28	8	1	
9	1	9	36	84	136	136	84	36	9	1

Catalan numbers

Use cases:

- The number of valid groupings of n pairs of parentheses
- The number of diagonal-avoiding paths on $n \times n$ grid
- The number of ways to triangulate a regular polygon with $n+2$ sides
- The number of rooted full binary trees with n internal nodes
- The number of rooted trees with n edges
- The number of ways to correctly parenthesize an ordered expression of $n+1$ items with a binary operation

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \begin{cases} 1 & n = 0 \lor n = 1 \\ \sum\limits_{k=0}^{n-1} C_k C_{n-1-k} & \text{otherwise} \end{cases}$$

Values C_0 to C_{10}: $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796$

Euler numbers

The number of permutations $1, \ldots, n$ with exactly k non-decreasing segments.

$$\left\langle {n \atop k} \right\rangle = \sum_{l=0}^{k} (-1)^l \binom{n+1}{l} (k+1-l)^n = \begin{cases} 1 & k = 0 \lor k = n \\ k \left\langle {n-1 \atop k} \right\rangle + (n-k+1) \left\langle {n-1 \atop k-1} \right\rangle & \text{otherwise} \end{cases}$$

n \ k	0	1	2	3	4	5	6	7
0	1							
1	1	1						
2	1	4	1					
3	1	11	11	1				
4	1	26	66	26	1			
5	1	57	302	302	57	1		
6	1	120	1191	2416	1191	120	1	
7	1	247	4293	15619	15619	4293	247	1

Stirling Numbers of the first kind

Number of permutations $1, \ldots, n$ with exactly k cycles.

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{cases} 1 & k = n \\ 0 & k = 0 \land n > 0 \\ \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} & \text{otherwise} \end{cases}$$

n \ k	0	1	2	3	4	5	6	7	8
0	1								
1	0	1							
2	0	1	1						
3	0	2	3	1					
4	0	6	11	6	1				
5	0	24	50	35	10	1			
6	0	120	274	225	85	15	1		
7	0	720	1764	1624	735	175	21	1	
8	0	5040	13068	13132	6769	1960	322	28	1

Stirling Numbers of the second kind

Number of partitions of n elements into k sets.

$$\left\{ {n \atop k} \right\} = \begin{cases} 1 & k = n \\ 0 & k = 0 \land n > 0 \\ \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\} & \text{otherwise} \end{cases}$$

$n \backslash k$	0	1	2	3	4	5	6	7	8
0	1								
1	0	1							
2	0	1	1						
3	0	1	3	1					
4	0	1	7	6	1				
5	0	1	15	25	10	1			
6	0	1	31	90	65	15	1		
7	0	1	63	301	350	140	21	1	
8	0	1	127	966	1701	1050	266	28	1

Derangements

Amount of permutations of a set with n elements such that no element is at its starting position.

$$\text{der}(n) \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n-1)(\text{der}(n-1) + \text{der}(n-2)) & \text{otherwise} \end{cases}$$

First values: $1, 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 1334961, 14684570$

Primes

2	3	5	7	11	13	17	19	23	29_{10}	31	37	41
43	47	53	59	61	67	71_{20}	73	79	83	89	97	101
103	107	109	113_{30}	127	131	137	139	149	151	157	163	167
173_{40}	179	181	191	193	197	199	211	223	227	229_{50}	233	227

10^i	1	2	3	4	5	6	7	8	9
cnt	4	25	168	1229	9592	78498	664579	5761455	50847534

Number of Divisors

$\leq 10^i$	1	2	3	4	5	6	7	8	9
cnt	4	12	32	64	128	240	448	768	1344

Hypergeometric distribution

Set with N elements of which K have a wanted property. The probability of k elements having property K when choosing n is

$$H = \frac{\binom{K}{k} \cdot \binom{N-K}{n-k}}{\binom{N}{n}}$$

Newton Method

The intersections of a function f with the x-axis can be approximated iteratively:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The method converges towards the solution quadratically therefore doubleing the amount of correct decimal places every iteration.

Number Theory

Extended Euclid $\mathcal{O}(\log(a+b))$

Calculates a and b such that $\gcd(x, y) = ax + by$

```
ll gcd(ll x, ll y, ll& a, ll& b) {
  if (y) {
    ll res = gcd(y, x % y, b, a);
    return b -= x / y * a, res;
  }
```

```
  return a = 1, b = 0, x;
}
```

Modular Exponentiation $\mathcal{O}(\log b)$

```
ll modpow(ll a, ll b, ll m) {
  a %= m;  // normal pow with m = inf
  ll res = 1;
  while (b > 0) {
    if (b & 1) res = res * a % m;
    a = a * a % m;
    b >>= 1;
  }
  return res;
}
```

Eulers Totoid Function $\mathcal{O}(\sqrt{n})$

$\phi(n)$: Number of Integers in $[1, n]$ that are coprime to n.

p prime, $k \in \mathbb{N} \Rightarrow \phi(p^k) = p^k - p^{k-1}$

a, b coprime $\Rightarrow \phi(a \cdot b) = \phi(a) \cdot \phi(b)$

$a, b \in \mathbb{N} \Rightarrow \phi(a \cdot b) = \phi(a) \cdot \phi(b) \cdot \frac{gcd(a,b)}{\phi(gcd(a,b))}$

P prime factors of $n \Rightarrow \phi(n) = n \cdot \prod_{p \in P}(1 - \frac{1}{p})$

```
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if(n % i == 0) {
            while(n % i == 0) n /= i;
            result -= result / i;
        }
    }
    if(n > 1) result -= result / n;
    return result;
}
```

Data Structures

Fenwick tree (range) query: $\mathcal{O}(\log n)$, (point) update $\mathcal{O}(\log n)$

```
template<typename T>
struct FT {
  int n;
  vector<T> A;
  FT(int sz) : n{sz}, A(n, 0) {}
  T query(int i) {
    T sum = 0;
    for (--i; i >= 0; i = (i & (i + 1)) - 1) sum += A[i];
    return sum;
  }
  T query(int i, int j) { return query(j) - query(i); }
  void update(int i, T add) {
    for (; i < n; i |= i + 1) A[i] += add;
  }
  // lb assumes query(i, i) >= 0 forall i in [1, n]
  // returns min p >= 1, so that [1, p] >= sum
  // if [1, n] < sum, return n + 1
  /* TODO: 0 indexed
  int lb(T sum) {
    int pos = 0;
    for (int pw = 1 << 25; pw; pw >>= 1)
      if (pos + pw <= n && sum > A[pos | pw]) sum -= A[pos |= pw];
    return pos + !!sum;
  }
  */
};
```

2D Fenwick tree (range) query: $\mathcal{O}(\log n \log m)$, (point) update $\mathcal{O}(\log n \log m)$

```
template<typename T>
struct FT2D {
  int n;
  vector<FT<T>> fts;
  FT2D(int sz1, int sz2) : n{sz1}, fts(n + 1, FT<T>(sz2)) {};
  T query(int i, int j1, int j2) {
    T sum = 0;
    for (--i; i >= 0; i = (i & (i + 1)) - 1) sum += fts[i].query(j1, j2);
    return sum;
  }
  T query(int i1, int i2, int j1, int j2) {
    return query(i2, j1, j2) - query(i1, j1, j2);
  }
  void update(int i, int j, T add) {
    for (; i < n; i |= i + 1) fts[i].update(j, add);
  }
};
```

Segment tree build: $\mathcal{O}(n)$, (range) query: $\mathcal{O}(\log n)$, (point) update $\mathcal{O}(\log n)$

```
template<typename T, typename F>
struct ST {
  using value_type = T;
  using merge_type = F;
  const int n;
  const T e;
  F merge;
  vector<T> data;
  ST(int sz, T _e, F m) : n{sz}, e{_e}, merge{m}, data(2 * n, e) {}
  void build() {
```

```
    for (int i = n - 1; i; --i)
      data[i] = merge(data[i << 1], data[i << 1 | 1]);
  }
  T query(int l, int r) {
    T li = e, ri = e;
    for (l += n, r += n; l < r; r >>= 1, l >>= 1) {
      if (l & 1) li = merge(li, data[l++]);
      if (r & 1) ri = merge(data[--r], ri);
    }
    return merge(li, ri);
  }
  void update(int i, T val) {
    for (data[i += n] = val; i > 1; i >>= 1)
      data[i >> 1] = merge(data[i & ~1], data[i | 1]);
  }
};
```

Union Find/DSU

n Elements, $m \leq n$ Operations: $\mathcal{O}(m \cdot \alpha(m, n)) \approx \mathcal{O}(n)$

```
struct DSU {
  DSU(int size) : msize(size), data(size, -1) {}
  bool sameSet(int a, int b) { return find(a) == find(b); }
  int find(int x) {
    return data[x] < 0 ? x : data[x] = find(data[x]);
  }
  bool join(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (data[a] > data[b]) swap(a, b);
    data[a] += data[b], data[b] = a;
    return --msize, true;
  }
  int size() { return msize; }
  int size(int a) { return -data[find(a)]; }
  int msize;
  vi data;
};
```

1D Sparse Table

build: $\mathcal{O}(n \log n)$ query: $\mathcal{O}(1)$

For any idempotent function

```
template <typename T, typename F>
struct SPT {
  vector<vector<T>> d;
  F f;
  SPT (int n, F _f) : d(32 - __builtin_clz(n), vector<T>(n)), f{_f} {}
  SPT (const vector<T>& v, F _f) : SPT(SZ(v), _f) {
    d[0] = v;
    build();
  }
  void build() {
    for (int j = 1; (1 << j) <= SZ(d[0]); ++j) {
      for (int i = 0; i + (1 << j) <= SZ(d[0]); ++i) {
        d[j][i] = f(d[j - 1][i], d[j - 1][i + (1 << (j - 1))]);
      }
    }
  }
  T query(int l, int r) { // [l, r)
    int k = 31 - __builtin_clz(r - l);
    return f(d[k][l], d[k][r - (1 << k)]);
  }
};
```

2D Sparse Table

build: $\mathcal{O}(nm \log n \log m)$ query: $\mathcal{O}(1)$

```
typedef vector<vvi> vvvi;
typedef vector<vvvi> vvvvi;
struct SPT2D {
  vvvvi spT;
  int n, m, log2n, log2m;

  SPT2D(vvi& A) : n(SZ(A)), m(SZ(A[0])),
        log2n(33 - __builtin_clz(n)),
        log2m(33 - __builtin_clz(m)) {
    spT.assign(n, vvvi(log2n, vvi(m, vi(log2m))));

    F0R (ir, n) {
      F0R (ic, m)
        spT[ir][0][ic][0] = A[ir][ic];
      for (int jc = 1; (1 << jc) <= m; ++jc)
        for (int ic = 0; ic + (1 << jc) <= m; ++ic)
          spT[ir][0][ic][jc] =
              min(spT[ir][0][ic][jc - 1],
                  spT[ir][0][ic + (1 << (jc - 1))][jc - 1]);
    }
    for (int jr = 1; (1 << jr) <= n; ++jr)
      for (int ir = 0; ir + (1 << jr) <= n; ++ir)
        for (int jc = 0; (1 << jc) <= m; ++jc)
          for (int ic = 0; ic + (1 << jc) <= m; ++ic)
            spT[ir][jr][ic][jc] =
                min(spT[ir][jr - 1][ic][jc],
                    spT[ir + (1 << (jr - 1))][jr - 1][ic][jc]);
  }

  int query(int r1, int r2, int c1, int c2) { //r2, c2 are exclusive
    int rk = 31 - __builtin_clz(r2 - r1);
    int ck = 31 - __builtin_clz(c2 - c1);

    int cc = c2 - (1 << ck);
```

```
    int rr = r2 - (1 « rk);
    return min({spT[r1][rk][c1][ck], spT[r1][rk][cc][ck],
                spT[rr][rk][c1][ck], spT[rr][rk][cc][ck]});
  }
};
```

Dynamic Programming

Knapsack $\mathcal{O}(n\sum_{i=1}^{n} p_i)$

```
const int inf = 1e9;
int knapsack(const vi& w, const vi& p, int B) {
  ll maxP = accumulate(ALL(p), 0);

  vvi dp(maxP + 1, vi(SZ(w), inf));
  fill(ALL(dp[0]), 0);
  dp[p[0]][0] = w[0];

  FOR(t, 1, maxP + 1) {
    FOR(i, 1, SZ(w)) {
      dp[t][i] = dp[t][i - 1];
      if(t - p[i] >= 0)
        ckmin(dp[t][i], dp[t - p[i]][i - 1] + w[i]);
    }
  }

  int res = 0;
  FOR(i, maxP + 1)
    if(dp[i][SZ(w) - 1] <= B)
      ckmax(res, i);
  return res;
}
```

TSP $\mathcal{O}(n2^n)$

```
const int INF = 1e9;
vvi dp, adj; // adjacency matrix
int tsp_calc(int pos, int start, int mask) {
  if ((1 « SZ(adj)) - 1 == mask) return adj[pos][start];
  if (dp[pos][mask] != -1) return dp[pos][mask];
  int minV = INF;
  FOR (i, SZ(adj))
    if (i != pos && !(mask & (1 « i)))
      ckmin(minV, adj[pos][i] + tsp_calc(i, start, mask | (1 « i)));
  return dp[pos][mask] = minV;
}
int tsp(int start = 0) {
  dp.assign(SZ(adj), vi(1 « SZ(adj), -1));
  FOR(i, SZ(adj)) ckmin(adj[i][i], 0);
  return tsp_calc(start, start, 1 « start);
}
```

Subset Sum $\mathcal{O}(n\sum_{i=1}^{n} v_i)$

```
const int mxSum = 1000;
bitset<mxSum + 1> subSetSum(const vi& v) {
  bitset<mxSum + 1> dp;
  dp[0] = 1;
  for (int i : v) dp |= dp « i;
  return dp;
}
```

Edit Distance $\mathcal{O}(nm)$

```
const int inf = 1e9;

int editDistance(const string& a, const string& b) {
  vvi dp(SZ(a) + 1, vi(SZ(b) + 1, inf));
  FOR(i, SZ(a) + 1) {
    FOR(j, SZ(b) + 1) {
      if(!i) dp[i][j] = j;
      else if(!j) dp[i][j] = i;
      else {
        dp[i][j] = 1 + min({dp[i - 1][j - 1],
                            dp[i][j - 1], dp[i - 1][j]});
        if(a[i - 1] == b[j - 1]) ckmin(dp[i][j], dp[i - 1][j - 1]);
      }
    }
  }
  return dp[SZ(a)][SZ(b)];
}
```

Longest Increasing Subsequence $\mathcal{O}(n\log n)$

```
const int inf = 1e9;

int lis(const vi& a) {
  vi dp(SZ(a) + 1, inf);
  dp[0] = -inf;
  FOR(i, SZ(a)) {
    int ind = upper_bound(ALL(dp), a[i]) - dp.begin();
    if(dp[ind - 1] < a[i] && a[i] < dp[ind])
      dp[ind] = a[i];
  }
  return lower_bound(ALL(dp), inf) - dp.begin() - 1;
}
```

Longest Common Subsequence $\mathcal{O}(nm)$

```
int lcs(const string& s, const string& t) {
  int n = SZ(s), m = SZ(t);
  vvi dp(n + 1, vi(m + 1, 0));
  dp[n - 1][m - 1] = s[n - 1] == t[m - 1];
  for (int i = n - 2; ~i; --i)
    dp[i][m - 1] = s[i] == t[m - 1] ? 1 : dp[i + 1][m - 1];
  for (int i = m - 2; ~i; --i)
    dp[n - 1][i] = s[n - 1] == t[i] ? 1 : dp[n - 1][i + 1];
  for (int i = n - 2; ~i; --i)
    for (int j = m - 2; ~j; --j)
      dp[i][j] = max({dp[i + 1][j + 1] + (s[i] == t[j]),
                      dp[i + 1][j], dp[i][j + 1]});
  return dp[0][0];
}
```

Graphs

Topological Sort $\mathcal{O}(|V| + |E|)$

A priority queue can be used if further sorting is necessary.

```
int N;
vvi adj(N);
vi in(N);  // in degree for every node
vi toposort() {
  vi q; // Result saved in q
  FOR (i, N)
    if (!in[i]) q.pb(i);
  FOR (i, SZ(q))
    for (int v : adj[q[i]])
      if (!--in[v]) q.pb(v);
  return q;
}
```

SCC Tarjan $\mathcal{O}(|V| + |E|)$

```
vvi adj;
vi dfs_num, dfs_low, S;
vector<bool> onStack;
int dfsCounter;
void scc(int v, vvi& sccs) {
  dfs_num[v] = dfs_low[v] = dfsCounter++;
  S.push_back(v);
  onStack[v] = true;
  for (int u : adj[v]) {
    if (dfs_num[u] == -1) scc(u, sccs);
    if (onStack[u]) ckmin(dfs_low[v], dfs_low[u]);
  }
  if (dfs_num[v] == dfs_low[v]) {
    sccs.eb(); int u;
    do {
      u = S.back();
      S.pop_back();
      onStack[u] = 0;
      sccs.back().pb(u);
    } while (u != v);
  }
}
vvi scc() {
    dfs_num.assign(SZ(adj), -1);
    dfs_low.assign(SZ(adj), 0);
    onStack.assign(SZ(adj), 0);
    dfsCounter = 0;
    vvi sccs;
    FOR (i, SZ(adj))
        if (!~dfs_num[i]) scc(i, sccs);
    return sccs;
}
```

Articulation Points $\mathcal{O}(|V| + |E|)$

```
vi dfsNum, low;
int dfsCounter = 0;
vvi adj;
int artiDfs(int v, vi& a, int p = -1) {
  dfsNum[v] = low[v] = dfsCounter++;
  int children = 0;
  bool aP = false;
  for (int u : adj[v]) {
    if (dfsNum[u] == -1) {
      ckmin(low[v], artiDfs(u, a, v));
      if (low[u] >= dfsNum[v] && p != -1 && !aP) {
        a.pb(v);
        aP = true;
      }
      children++;
    } else if (u != p)
      ckmin(low[v], dfsNum[u]);
  }
  if (p == -1 && children > 1) a.pb(v);
  return low[v];
}
vi findArtiPoints() {
  dfsNum.assign(SZ(adj), -1);
  low.assign(SZ(adj), -1);
  dfsCounter = 0;
  vi res;
```

```
  FOR (v, SZ(adj))
    if (dfsNum[v] == -1) artiDfs(v, res);
  return res;
}
```

Bridges $\mathcal{O}(|V| + |E|)$

```
vvi adj;
vi dfsNum, low;
int dfsCounter = 0;
int bridgeDfs(int v, vii& b, int p = -1) {
  dfsNum[v] = low[v] = dfsCounter++;
  for (int u : adj[v]) {
    if (dfsNum[u] == -1) {
      ckmin(low[v], bridgeDfs(u, b, v));
      if (low[u] > dfsNum[v]) b.eb(v, u);
    } else if (u != p)
      ckmin(low[v], dfsNum[u]);
  }
  return low[v];
}
vii findBridges() {
  vii bridges;
  dfsNum.assign(SZ(adj), -1);
  low.assign(SZ(adj), -1);
  FOR (v, SZ(adj))
    if (dfsNum[v] == -1) bridgeDfs(v, bridges);
  return bridges;
}
```

Minimal Spanning Tree - Kruskal $\mathcal{O}(|E| \log |V|)$

```
template <typename W, typename C = less<tuple<W, int, int>>>
tuple<bool, W, vi> kruskal(int V, vector<tuple<W, int, int>>& edges, C cmp = C
  sort(ALL(edges), cmp); DSU dsu(V); vi mst;
  W w = 0;
  for (int i = 0; SZ(dsu) > 1 && i < SZ(edges); ++i) {
    auto [d, a, b] = edges[i];
    if (dsu.join(a, b)) mst.pb(i), w += d;
  }
  return mt(SZ(dsu) == 1, w, mst);
}
```

Shortest Paths

Dijkstra $((|E| + |V|) \log |V|)$

```
template <typename D = int>
vector<D> dijkstra(int start, const vector<vector<pair<int, D>>>& adj, const D INF = 1e9
  vector<D> dist(SZ(adj), INF);
  set<pair<D, int>> q;
  q.emplace(0, start);
  dist[start] = 0;
  while (!q.empty()) {
    auto [d, v] = *q.begin(); q.erase(q.begin());
    if (dist[v] < d) continue;
    for (auto [u, du] : adj[v])
      if (ckmin(dist[u], d + du)) q.emplace(d + du, u);
  }
  return dist;
}
```

Bellman Ford $\mathcal{O}(|E||V|)$

Check for negative cycles:
$dist$ still changes in a $|V|$'th relaxation step with $dist_i = 0$ initially for all i.

```
const int inf = 1e9;
// vertex a, vertex b, distance
vector<tuple<int, int, int>> edges;
int V;
// Returns empty vector on negative cycle
vi bellmanFord(int start) {
  vi dist(V, inf);
  dist[start] = 0;
  bool negCycle = false;
  FOR (i, V) {
    negCycle = false;
    for (auto [a, b, d] : edges)
      if (dist[a] < inf && ckmin(dist[b], dist[a] + d))
          negCycle = true;
  }
  return negCycle ? vi() : dist;
}
```

Bellman Ford with Queue $\mathcal{O}(|E||V|)$

This approach may be faster

```
const int inf = 1e9;
vvii adj;
// returns empty vector if there's a neg cycle
vi bellmanFordQueue(int start) {
  vi dist(SZ(adj), inf);
```

```
  queue<int> q;
  vector<bool> inQ(SZ(adj), false);
  vi cnt(SZ(adj), 0); // cnt number of relaxations for neg cycles
  q.push(start);
  dist[start] = 0; inQ[start] = true;
  while (!q.empty()) {
    int v = q.front(); q.pop();
    inQ[v] = false;
    for (auto [u, d] : adj[v])
      if (ckmin(dist[u], dist[v] + d)) {
        if (++cnt[u] > SZ(adj)) return vi();
        if (!inQ[u]) q.push(u), inQ[u] = true;
      }
  }
  return dist;
}
```

Floyd Warshall $\mathcal{O}(|V|^3)$

```
const ll INF = 1e18;
vector<vector<ll» adj;  // adjacency matrix
bool negCycle = false;
void floydWarshall() {
  FOR (k, SZ(adj)) FOR (i, SZ(adj)) FOR (j, SZ(adj))
    if(adj[i][k] != INF && adj[k][j] != INF)
      ckmin(adj[i][j], adj[i][k] + adj[k][j]);
  FOR (k, SZ(adj)) if(adj[k][k] < 0) negCycle = true;
}
```

Forest

Lowest Common Ancestor build: $\mathcal{O}(|V|\log|V|)$ query: $\mathcal{O}(1)$

```
struct LCA {
  vi height, first;
  SPT<int, function<int(int, int)» spt;
  LCA(vvi& adj, int root = 0) : height(SZ(adj), -1), first(SZ(adj), -1),
      spt(2 * SZ(adj) - 1, [self = this](int a, int b) {
        return self->height[a] < self->height[b] ? a : b;
      }) {
    int idx = 0;
    dfs(adj, root, idx);
    spt.build();
  }
  void dfs(vvi& adj, int v, int& idx, int h = 0) {
    first[v] = idx;
    spt.d[0][idx++] = v;
    height[v] = h;
    for (int u : adj[v]) if (first[u] == -1) {
      dfs(adj, u, idx, h + 1);
      spt.d[0][idx++] = v;
    }
  }
  int query(int a, int b) {
    ii m = minmax(first[a], first[b]);
    return spt.query(m.fi, m.se);
  }
};
```

LCA with binary lifting build: $\mathcal{O}(|V|\log|V|)$ query: $\mathcal{O}(\log|V|)$

```
struct LCA {
  int n, logN, root;
  vvi up;
  vi h;
  LCA(vvi& adj, int r = 0)
      : n{SZ(adj)}, logN{31 - __builtin_clz(n)},
        root{r}, up(n, vi(logN + 1, root)),
        h(n, -1) {
    build(adj);
  }
  void build(vvi& adj) {
    queue<int> q;
    q.push(root);
    h[root] = 0;
    while (SZ(q)) {
      int v = q.front();
      q.pop();
      for (int u : adj[v])
        if (h[u] == -1) {
          h[u] = h[v] + 1;
          q.push(u);
          up[u][0] = v;
        }
    }
    FOR (exp, 1, logN + 1)
      FOR (v, n)
        if (up[v][exp - 1] != -1)
          up[v][exp] = up[up[v][exp - 1]][exp - 1];
  }
  int jumpUp(int v, int amt) {
    for (int i = 0; v != -1 && (1 « i) <= amt; ++i)
      if (amt & (1 « i))
        v = up[v][i];
    return v;
  }
  int query(int v, int u) {
    v = jumpUp(v, max(0, h[v] - h[u]));
    u = jumpUp(u, max(0, h[u] - h[v]));
```

```
    if (u == v) return u;
    for (int l = logN; ~l; --l) {
      int jmpU = up[u][l], jmpV = up[v][l];
      if (jmpU == -1 || jmpV == -1) continue;
      if (jmpU != jmpV) {
        u = jmpU;
        v = jmpV;
      }
    }
    return up[v][0];
  }
};
```

Heavy-light decomposition build: $\mathcal{O}(|V|)$, query/update: $\mathcal{O}(\log^2|V|)/\mathcal{O}(\log|V|)$

```
template<typename T, typename F>
struct HLD {
  int n;
  vi par, sz, height, in, pos;
  vvi paths;
  ST<T, F> st;
  HLD(vvi& adj, const vector<T>& val, T unit, F merge, int root = 0)
    : n{SZ(adj)}, par(n), sz(n, 1), height(n), in(n), pos(n),
      st{n, unit, merge} {
    dfssz(adj, root);
    vi order;
    dfsbuild(adj, root, order);
    int j = 0;
    for (auto it = order.crbegin(); it != order.crend(); ++it)
      for (int v : paths[*it]) st.data[st.n + (pos[v] = j++)] = val[v];
    st.build();
  }
  int dfssz(vvi& adj, int v, int h = 0, int p = -1) {
    par[v] = p; height[v] = h;
    for (int u : adj[v])
      if (p != u) sz[v] += dfssz(adj, u, h + 1, v);
    return sz[v];
  }
  void dfsbuild(vvi& adj, int v, vi& order, int p = -1, bool hvy = false) {
    if (hvy) paths[in[v] = in[p]].pb(v);
    else {
      in[v] = SZ(paths);
      paths.pb({v});
    }
    int h = -1;
    for (int u : adj[v])
      if (p != u) {
        if (sz[u] > sz[v] / 2) h = u;
        else dfsbuild(adj, u, order, v);
      }
    if (~h) dfsbuild(adj, h, order, v, true);
    if (paths[in[v]][0] == v) order.pb(in[v]);
  }
  void update(int v, T val) { st.update(pos[v], val); }
  T queryPath(int a, int b) {
    T v = st.e;
    while (in[a] != in[b]) {
      if (height[paths[in[a]][0]] < height[paths[in[b]][0]]) swap(a, b);
      v = st.merge(v, st.query(pos[paths[in[a]][0]], pos[a] + 1));
      a = par[paths[in[a]][0]];
    }
    if (height[a] > height[b]) swap(a, b);
    return st.merge(v, st.query(pos[a], pos[b] + 1));
  }
  T querySubtree(int v) {
    return st.query(pos[v], pos[v] + sz[v]);
  }
};
```

Flow

Max-flow min-cut theorem. The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts

Edges for flow algorithms

```
template <typename F>
struct edge {
  edge(int from, int to, F capacity, F flow = 0)
      : mfrom(from), mto(to), mcapacity(capacity), mflow(flow) {}
  int mfrom, mto;
  F mcapacity, mflow;
  int other(int v) { return v == mfrom ? mto : mfrom; }
  F capacity(int v) { return v == mfrom ? mcapacity : 0; }
  F flow(int v) { return v == mfrom ? mflow : -mflow; }
  void adjust(int v, F amount) {
    mflow += v == mfrom ? amount : -amount;
  }
};
template <typename F>
ostream& operator<<(ostream& o, const edge<F>& e) {
  return o << e.mfrom << "-- " << e.mflow << '/'
           << e.mcapacity << " -->" << e.mto;
}
```

Edmonds Karp $\mathcal{O}(|V||E|^2)$

```
#include "flowedge.cc"
template <typename F = ll>
struct EK {
  vvi adj;
  vector<edge<F>> edges;
  int S, T;
  EK(int n, int s = -1, int t = -1) { reset(n, s, t); }
  int add(int from, int to, F c = numeric_limits<F>::max(), F f = 0) {
    edges.eb(from, to, c, f);
    adj[from].pb(SZ(edges) - 1);
    adj[to].pb(SZ(edges) - 1);
    return SZ(edges) - 1;
  }
  void clear() { edges.clear(); adj.clear(); }
  void reset(int n, int s = -1, int t = -1) {
    clear();
    adj.resize(n + (s == -1) + (t == -1));
    S = s == -1 ? n : s;
    T = t == -1 ? n + (s == -1) : t;
  }
  F augment(int s, int t) {
    vii p(SZ(adj), {-1, -1});
    queue<int> q;
    p[s] = mp(-1, 0);
    q.push(s);
    while (!q.empty()) {
      int v = q.front();
      if (v == t) break;
      q.pop();
      for (int i : adj[v]) {
        auto& e = edges[i];
        if (p[e.other(v)].se == -1 && e.flow(v) < e.capacity(v)) {
          p[e.other(v)] = mp(v, i); q.push(e.other(v));
        }
      }
    }
    if (p[t].se == -1) return 0;
    F mf = numeric_limits<F>::max();
    for (ii c = p[t]; c.fi != -1; c = p[c.fi])
      ckmin(mf, edges[c.se].capacity(c.fi) - edges[c.se].flow(c.fi));
    for (ii c = p[t]; c.fi != -1; c = p[c.fi])
      edges[c.se].adjust(c.fi, mf);
    return mf;
  }
  F maxflow() { return maxflow(S, T); }
  F maxflow(int s, int t) {
    F maxflow = 0;
    while (F plus = augment(s, t)) maxflow += plus;
    return maxflow;
  }
};
```

Dinic $\mathcal{O}(|V|^2|E|)$

```
#include "flowedge.cc"
template <typename F = ll>
struct DC {
  vector<edge<F>> edges;
  vvi adj;
  vi dist, ptr;
  int S, T, N;
  DC(int n, int m = 0, int s = -1, int t = -1) {
    reset(n, m, s, t);
  }
  void buildMatchingEdges(int m) {
    FOR (i, N) add(S, i, 1);
    FOR (i, m) add(N + i, T, 1);
  }
  int add(int from, int to, F c = numeric_limits<F>::max(), F f = 0) {
    edges.eb(from, to, c, f);
    adj[from].pb(SZ(edges) - 1);
    adj[to].pb(SZ(edges) - 1);
    return SZ(edges) - 1;
  }
  int match(int from, int to) { return add(from, N + to, 1);}
  vii matching() {
    vii res; res.reserve(maxflow());
    for (const auto& e : edges)
      if (e.mflow == 1 and e.mfrom != S and e.mto != T)
        res.eb(e.mfrom, e.mto - N);
    return res;
  }
  void clear() { edges.clear(); adj.clear(); }
  void reset(int n, int m = 0, int s = -1, int t = -1) {
    clear();
    adj.resize((N = n) + m + (s == -1) + (t == -1));
    S = s == -1 ? n + m : s;
    T = t == -1 ? n + m + (s == -1) : t;
    if (m != 0) buildMatchingEdges(m);
  }
  bool bfs(int s, int t) {
    dist.assign(SZ(adj), SZ(adj));
    queue<int> q;
    q.push(s);
    dist[s] = 0;
    while (SZ(q)) {
      int v = q.front(); q.pop();
      for (int i : adj[v]) {
        auto& e = edges[i];
```

```
        if (dist[e.other(v)] == SZ(adj) && e.flow(v) < e.capacity(v)) {
          dist[e.other(v)] = dist[v] + 1;
          q.push(e.other(v));
        }
      }
    }
    return dist[t] < SZ(adj);
  }
  F dfs(int v, int t, F available) {
    if (v == t || !available) return available;
    F pushed = 0;
    for (; ptr[v] < SZ(adj[v]); ++ptr[v]) {
      auto& e = edges[adj[v][ptr[v]]];
      if (dist[v] + 1 != dist[e.other(v)])
        continue;
      F wasPushed = dfs(e.other(v), t,
                        min(available - pushed, e.capacity(v) - e.flow(v)));
      pushed += wasPushed;
      e.adjust(v, wasPushed);
      if (pushed == available) return pushed;
    }
    return pushed;
  }
  F maxflow() {
    return maxflow(S, T);
  }
  F maxflow(int s, int t) {
    F f = 0;
    for (;;) {
      if (!bfs(s, t)) return f;
      ptr.assign(SZ(adj), 0);
      f += dfs(s, t, numeric_limits<F>::max());
    }
  }
};
using BM = DC<int>;
```

Push Relabel $\mathcal{O}(|V|^3)$

```
#include "flowedge.cc"
template <typename F = ll>
struct PR {
  vi label, currentEdge;
  vector<F> excess;
  queue<int> active;
  vvi adj;
  vector<edge<F>> edges;
  int S, T;
  PR(int n, int s = -1, int t = -1) { reset(n, s, t); }
  int add(int from, int to, F c = numeric_limits<F>::max(), F f = 0) {
    edges.eb(from, to, c, f);
    adj[from].pb(SZ(edges) - 1);
    adj[to].pb(SZ(edges) - 1);
    return SZ(edges) - 1;
  }
  void clear() { edges.clear(); adj.clear(); }
  void reset(int n, int s = -1, int t = -1) {
    clear();
    adj.resize(n + (s == -1) + (t == -1));
    S = s == -1 ? n : s;
    T = t == -1 ? n + (s == -1) : t;
  }
  void push(int v, edge<F>& e) {
    F more = min(excess[v], e.capacity(v) - e.flow(v));
    excess[e.other(v)] += more;
    excess[v] -= more;
    e.adjust(v, more);
    if (more && excess[e.other(v)] == more) active.push(e.other(v));
  }
  void relabel(int v) {
    int m = numeric_limits<int>::max();
    for (int i : adj[v]) {
      auto& e = edges[i];
      if (e.flow(v) < e.capacity(v)) ckmin(m, label[edges[i].other(v)]);
    }
    if (m < numeric_limits<int>::max()) label[v] = m + 1;
  }
  void discharge(int v) {
    while (excess[v]) {
      auto& e = edges[adj[v][currentEdge[v]]];
      if (label[v] - 1 == label[e.other(v)] &&
          e.flow(v) < e.capacity(v))
        push(v, e);
      else if (SZ(adj[v]) == ++currentEdge[v]) {
        currentEdge[v] = 0;
        relabel(v);
      }
    }
  }
  F maxflow(int s, int t) {
    currentEdge.assign(SZ(adj), 0);
    label.assign(SZ(adj), 0);
    excess.assign(SZ(adj), 0);
    excess[s] = numeric_limits<F>::max();
    label[s] = SZ(adj);
    for (int i : adj[s]) push(s, edges[i]);
    while (!active.empty()) {
      if (active.front() != s && active.front() != t)
        discharge(active.front());
      active.pop();
    }
```

```
    F maxflow = 0;
    for (int i : adj[s]) maxflow += edges[i].flow(s);
    return maxflow;
  }
};
```

Minimum s-t cut

To find a minimal s-t cut find all nodes that are reachable in the residual network for a network w/ maximum flow from s. This is the s part of the cut. All other nodes belong to the t part.

Closure Problem

A closure of a directed graph is a set of vertices with no outgoing edges. The closure problem is the task to find the maximum weighted closure. Solvable through reduction to a maximum flow problem: Add source and target, connect all the vertices with positive weight w to the source with capacity w and connect all the vertices with negative weight w to the target with capacity $-w$. All of the edges in the original graph have infinite capacity in the new graph. The weight of the maximum weighted closure is equal to the sum of all positive weighted vertices in the original graph minus the maximum flow in the constructed graph.

Strings

</> Trie/Prefix Tree $\mathcal{O}(n)$ for set and get

```
template <typename T, int E = 26, typename V = char, V base = 'a'>
struct Trie {
  using str = basic_string<V>;
  vector<array<int, E>> nxt;
  vector<T> v;
  Trie() : nxt{array<int, E>{}}, v(1, -1) {}
  void set(const str& s, T val) {
    int it = 0;
    for (V c : s) {
      if (!nxt[it][c - base]) {
        nxt[it][c - base] = SZ(nxt);
        nxt.eb();
        v.eb();
      }
      it = nxt[it][c - base];
    }
    v[it] = val;
  }
  T get(const str& s) {
    int it = 0;
    for (V c : s) {
      if (!nxt[it][c - base]) return T();
      it = nxt[it][c - base];
    }
    return v[it];
  }
};
```

</> Prefix Function $\mathcal{O}(n)$

For a string s return an array in which the i-th entry is the length of the longest proper prefix of $s[0, \ldots, i]$ which is also a suffix. Note that the 0-th entry is 0

```
vi prefixFunction(const string& s) {
  vi prefix(SZ(s));
  FOR (i, 1, SZ(s)) {
    int j = prefix[i - 1];
    while (j > 0 && s[i] != s[j]) j = prefix[j - 1];
    if (s[i] == s[j]) ++j;
    prefix[i] = j;
  }
  return prefix;
}
```

</> KMP $\mathcal{O}(n + m)$

Returns a list with all the starting indeces where the pattern matches the text.

```
vi preprocess(string& s) {
  vi fail(SZ(s) + 1);
  fail[0] = -1;
  for (int i = 0, j = -1; i < SZ(s);) {
    while (j >= 0 && s[i] != s[j]) j = fail[j];
    ++i, ++j;
    fail[i] = j;
  }
  return fail;
}
vi match(string& text, string& pattern) {
  vi matches, fail(preprocess(pattern));
  for (int i = 0, j = 0; i < SZ(text);) {
    while (j >= 0 && text[i] != pattern[j]) j = fail[j];
    ++i; ++j;
    if (j == SZ(pattern)) {
      matches.pb(i - j);
      j = fail[j];
    }
```

```
  }
  return matches;
}
```

Manachers $\mathcal{O}(n)$

Returns array where $P[i]$ contains the length of the palindrome with mid-point i. There are extra entries for where the mid-point is inbetween letters. Example for $abbaac$:

```
| a | b | b | a | a | a | c |
0 1 0 1 4 1 0 1 2 3 2 1 0 1 0
```

```
vi manacher(string s) {
  string T = "#";   // Assume that '#' is not in the alphabet
  for (char c : s) T += c, T += '#';
  vi P(SZ(T));
  int c = 0, R = 0;
  FOR (i, 1, SZ(T) - 1) {
    if (R > i) P[i] = min(R - i, P[2 * c - i]);
    for (int r = i + 1 + P[i], l = i - 1 - P[i];
         r < SZ(T) && l >= 0 && T[l] == T[r]; l--, r++)
      P[i]++;
    if (i + P[i] > R) c = i, R = i + P[i];
  }
  return P;
}
```

Aho-Corasick Automaton build: $\mathcal{O}(\sum |t_j|)$, query: $\mathcal{O}(|S|)$

Data structure to search how often n fixed strings $t_1, \ldots, t_n$ are contained in a variable string S. Strings t_j may each have values.

```
template <typename T, int E = 26, typename V = char, V base = 'a'>
struct AHC {
  using str = basic_string<V>;
  T e;
  vector<array<int, E>> nxt;
  vi fail;
  vector<T> val;
  AHC(T _e) : e{_e}, nxt(1, array<int, E>()), fail(1, 0), val(1, e) {}
  AHC(T _e, const vector<pair<str, T>>& strs)
      : e{_e}, nxt(1, array<int, E>()), fail(1, 0), val(1, e) {
    for (const auto& [s, v] : strs) insert(s, v);
    build();
  }
  void reserve(size_t sz) {
    nxt.reserve(sz);
    fail.reserve(sz);
    val.reserve(sz);
  }
  void insert(const str& s, T v) {
    int curr = 0;
    for (V c : s) {
      if (!nxt[curr][c - base]) {
        nxt[curr][c - base] = SZ(nxt);
        nxt.eb();
        val.pb(e);
      }
      curr = nxt[curr][c - base];
    }
    val[curr] += v;
  }
  void build() {
    fail.assign(SZ(nxt), 0);
    queue<int> q;
    FOR (i, E)
      if (nxt[0][i]) q.push(nxt[0][i]);
    while (!q.empty()) {
      int curr = q.front();
      q.pop();
      FOR (i, E) {
        if (nxt[curr][i]) {
          fail[nxt[curr][i]] = nxt[fail[curr]][i];
          val[nxt[curr][i]] += val[nxt[fail[curr]][i]];
          q.push(nxt[curr][i]);
        } else
          nxt[curr][i] = nxt[fail[curr]][i];
      }
    }
  }
  T query(const str& s) {
    int curr = 0;
    T res = e;
    for (V c : s) {
      if (nxt[curr][c - base])
        curr = nxt[curr][c - base];
      else
        while (curr && !nxt[curr][c - base]) curr = fail[curr];
      res += val[curr];
    }
    return res;
  }
};
```

Geometry

Geometry

```
#define xx real()
#define yy imag()
const double EPS = 1e-9;
const double INF = numeric_limits<double>::max();
using pt = complex<double>;
struct Line {
  double a, b, c;
};  // ax + by + c = 0
double dot(pt a, pt b) { return a.xx * b.xx + a.yy * b.yy; }
double cross(pt a, pt b) { return a.xx * b.yy - a.yy * b.xx; }
double dir(pt a, pt b, pt c) { return cross(b - a, c - a); }
bool cw(pt a, pt b, pt c) { return dir(a, b, c) < 0; }
bool ccw(pt a, pt b, pt c) { return dir(a, b, c) > 0; }
bool collinear(pt a, pt b, pt c) { return abs(dir(a, b, c)) < EPS; }
// Angle between a and b with o as origin (ccw).
// Return value in [0, 2PI)
double angle(pt a, pt b) {
  double ang = arg(a) - arg(b);
  return ang < 0 ? ang + 2 * M_PI : ang;
}
double angle(pt a, pt b, pt o) { return angle(b - o, a - o); }
// Theta in radiens
pt rotate(pt a, double theta) { return a * polar(1.0, theta); }
Line ptToLine(pt p1, pt p2) {
  if (abs(real(p1) - real(p2)) < EPS) {
    return {1.0, 0.0, -real(p1)};
  } else {
    double a = -(imag(p1) - imag(p2)) / (real(p1) - real(p2)),
           c = -(a * real(p1)) - imag(p1);
    return {a, 1.0, c};
  }
}
bool areParallel(Line l1, Line l2) {
  return abs(l1.a - l2.a) < EPS && abs(l1.b - l2.b) < EPS;
}
bool areSame(Line l1, Line l2) {
  return areParallel(l1, l2) && abs(l1.c - l2.c) < EPS;
}
pt intersectPt(Line l1, Line l2) {
  // Does not handle if same or parrallel
  if (areParallel(l1, l2)) return pt(-INF, -INF);
  double x =
      (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  double y;
  if (abs(l1.b) < EPS)
    y = -(l1.a * x + l1.c);
  else
    y = -(l2.a * x + l2.c);
  return pt(x, y);
}
double distToLine(pt p, pt a, pt b, bool segment = false) {
  pt ap = p - a, ab = b - a;
  double u = dot(ap, ab) / (abs(ab) * abs(ab));
  if (segment) {
    if (u < 0.0) return abs(p - a);  // a is closest
    if (u > 1.0) return abs(p - b);  // b is closest
  }
  return abs(p - a - ab * u);      // closest is in segment.
}
```

Polygon inPolygon: $\mathcal{O}(\log n)$, area: $\mathcal{O}(n)$, isConvex: $\mathcal{O}(n)$

```
bool inTriangle(pt a, pt b, pt c, pt p) {
  return
    abs(-abs(dir(a, b, c)) + abs(dir(a, b, p))
        + abs(dir(a, p, c)) + abs(dir(p, b, c))) < EPS;
}
// poly must be sorted in clockwise direction.
// returns true if point is on edge of poly.
bool inPolygon(const vector<pt>& poly, pt p) {
  double sum = 0;
  FOR(i, SZ(poly)) {
    double ang = angle(poly[i], poly[(i + 1) % SZ(poly)], p);
    if(ang > M_PI) ang -= 2*M_PI; // we want angle (-PI, PI] not [0, 2PI)
    sum += ang;
  }
  return abs(abs(sum) - 2*M_PI) < EPS;
}
// poly must be sorted in clockwise direction.
// poly[0] = poly[SZ(poly) - 1]
// returns true if point is on edge of poly.
bool inConvexPolygon(const vector<pt>& poly, pt p) {
  int l = 1, r = SZ(poly) - 2;
  while (l < r) {
    int mid = (l + r) / 2;
    if (cw(poly[0], poly[mid], p))
      l = mid + 1;
    else
      r = mid;
  }
  return inTriangle(poly[0], poly[l], poly[l - 1], p);
}
double area(const vector<pt>& p) {
  double res = 0.0;
  FOR (i, SZ(p))
```

```
    res += cross(p[i], p[(i + 1) % SZ(p)]);
  return abs(res) / 2;
}
bool isConvex(const vector<pt>& p) {
  if (SZ(p) < 3) return false;
  bool isLeft = ccw(p[0], p[1], p[2]) || collinear(p[0], p[1], p[2]),
      convex = true;
  FOR (i, SZ(p))
    convex &= isLeft == (ccw(p[i], p[(i + 1) % SZ(p)], p[(i + 2) % SZ(p)])
                 || collinear(p[i], p[(i + 1) % SZ(p)], p[(i + 2) % SZ(p)]));
  return convex;
}
```

Convex Hull $\mathcal{O}(n \log n)$

```
// careful with inputs for n < 2
vector<pt> convexHull(vector<pt>& pts) {
  int n = SZ(pts);
  sort(ALL(pts),
       [](pt a, pt b) { return mp(a.xx, a.yy) < mp(b.xx, b.yy); });
  vector<pt> up, down;
  up.eb(pts[0]); down.eb(pts[0]);
  FOR (i, 1, n) {
    // for colinear poins change ccw->!cw & cw->!ccw
    if (i == n - 1 || ccw(pts[0], pts[n - 1], pts[i])) {
      while (SZ(up) > 1 &&
             ccw(up[SZ(up) - 2], up[SZ(up) - 1], pts[i]))
        up.pop_back();
      up.eb(pts[i]);
    }
    if (i == n - 1 || cw(pts[0], pts[n - 1], pts[i])) {
      while (SZ(down) > 1 &&
             cw(down[SZ(down) - 2], down[SZ(down) - 1], pts[i]))
        down.pop_back();
      down.eb(pts[i]);
    }
  }
  vector<pt> ans(up);
  ans.insert(ans.end(), 1 + RALL(down));
  return ans;
}
```

Utils

Fast IO

```
int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
}
```

128 Bit Integer

```
ostream& operator«(ostream& o, __int128_t n) {
  if (n < 0) {
    o « '-';
    n *= -1;
  }
  ll mod = 1e18;
  string s;
  do {
    unsigned long long digits = n % mod;
    string dStr = to_string(digits);
    if (digits != n)
      s = string(18 - dStr.length(), '0') + dStr + s;
    else
      s = dStr + s;
    n = (n - digits) / mod;
  } while (n);
  return o « s;
}
```

Buildin

```
// Returns one plus the index of the least significant 1-bit of x, or
// if x is zero, returns zero.
int __builtin_ffs(unsigned int x);
int __builtin_clz(unsigned int x); // count leading zeroes
int __builtin_ctz(unsigned int x); // count trailing zeroes
int __builtin_popcount(unsigned int x); // count set one bits
// unsigned long: Postfix 'l', unsigned long long: Postfix 'll'

// Rotate bits of x (left|right) by n places.
unsigned int rotl(unsigned int x, int n);
unsigned int rotr(unsigned int x, int n);
```

Bit Operations

```
int msb(unsigned int x) {
  for (int i = 1; i <= 16; i «= 1) x |= x » i;
  return x - (x » 1);
}
```

```cpp
int lsb(int x) { return x & -x; }
bool oppositeSign(int x, int y) { return (x ^ y) < 0; }
bool isPowOf2(int x) { return x && !(x & (x - 1)); }
void allSubsets(int m) {
  for(int i = m; ; --i &= m) {
      /* */
      if(!i) break; // account for empty set
  }
}
void allSupersets(int m, int nx) {
  for (int i = m; i < nx; ++i |= m) { /* */ }
}
```

Ternary Search $\mathcal{O}(\log(r - l))$

```cpp
// f unimodal function on [l, r]
double terSearch(double l, double r, function<double(double)> f,
                 bool mx = true) {
  const double eps = 1e-9;
  while (abs(r - l) > eps) {
    double m1 = l + (r - l) / 3;
    double m2 = r - (r - l) / 3;
    if (mx == (f(m1) < f(m2))) l = m1;
    else r = m2;
  }
  return l;
}
```