

1	Init	1
2	STL	2
3	Datastructures	4
4	Math	6
5	Graph	10
6	Geometry	18
7	String	22
8	Other	23

1 Init

1.1 Template

```

1  #include <bits/stdc++.h>
2
3  void test(){
4
5  }
6  int main(){
7      //initialize
8      int T;
9      std::cin >> T;
10     while(T-->0) test();
11 }
```

1.2 Debug compile options

```
g++ -Wall -Wextra -pedantic -g -std=c++11 -O2 -Wshadow -Wformat=2 -Wfloat-equal -Wconversion -Wlogical-op
-Wcast-qual -Wcast-align -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC -fsanitize=address -fsanitize=undefined
-fstack-protector
```

1.3 Complexity

Value	Complexity	Algorithms
$n < 10^{18}$	$O(1)$, $O(\text{polylog}(N))$	Binary search, Functions (math)
$n < 10^9$	$O(\sqrt{N})$	Prime check, factorization
$n < 10^6$	$O(N)$, $O(N \log(N))$	Greedy, Sorting, Binary search + Greedy, Divide and conquer, 1D Dynamic programming
$n < 10^3$	$O(N^2)$, $O(N^2 \log(N))$	2D Dynamic programming, All-pair shortest path
$n < 100$	$O(N^3)$	Max-flow, Various unoptimized traversals
$n < 20$	$O(2^N)$	Combinations
$n < 16$	$O(N \cdot 2^N)$, $O(N^2 \cdot 2^N)$	Bitmask dynamic programming
$n < 10$	$O(N!)$	Permutations

1.4 Common errors

- Loop bounds (especially with DP)
- Array bounds (make slightly bigger)
- Initialization (init and reset)
- Incorrect output format
- Wrong nesting
- Precision (use epsilon)
- Overflow (use 64-bit when in doubt)
- Invalid expressions (divide by zero, segfault)
- Index offset
- Rounding (floor, ceil or normal)
- Read complete input
- Boundary cases
- Wrong variable, copying mistakes, etc

2 STL

2.1 I/O

```
1 std::ios::sync_with_stdio(false); //speedup IO (dont combine with printf/scanf)
2 std::cin.ignore(n); //ignore n characters before continue
3 std::getline(std::cin, s); //reads whole line into string
4 std::cin >> std::noskipws; //dont skip whitespace
5 std::cout << std::fixed; //use fixed-point notation
6 std::cout << std::setprecision(n); //set the precision to n decimals
7 std::cout << std::setw(n); //set the length each output will contain
8 std::cout << std::setfill(c); //set the char to fill the remaining chars from above
```

2.2 String

```
1 //read a line
2 std::string s;
3 std::getline(std::cin, s);
4 //string split
5 std::vector<std::string> split(const std::string &s, char delim) {
6     std::vector<std::string> elems;
7     std::stringstream ss(s);
8     std::string item;
9     while (std::getline(ss, item, delim)) {
10         elems.push_back(item);
11     }
12     return elems;
13 }
14 //substring
15 str.substr(pos, LENGTH); //till end if LENGTH is empty
16 //string find
17 str.find("substring") //return std::string::npos if not found, else position first letter
```

2.3 Containers

```
1 std::vector<int> vec;
2 //sort from small to large
3 bool cmp(int a, int b){
4     return a < b;
5 }
6 std::sort(vec.begin(), vec.end(), cmp);
7 std::sort(vec.begin(), vec.end(), std::less<int>());
8 //use std::greater<int>() for large to small and std::less<int>() for small to large
9 std::find(vec.begin(), vec.end(), NEEDLE); //use .find() if available (map/set)
10 std::count(vec.begin(), vec.end(), NEEDLE); //use .find() if available (map/set)
11 //find union/intersection/difference -- both need to be SORTED already
12 std::vector<int> tmp(vec.size()+vec2.size());
13 std::set_intersection(vec.begin(), vec.end(), vec2.begin(), vec2.end(), tmp.begin());
14 std::set_union(vec.begin(), vec.end(), vec2.begin(), vec2.end(), tmp.begin());
15 std::set_difference(vec.begin(), vec.end(), vec2.begin(), vec2.end(), tmp.begin());
16 //filter unique elements
17 auto last = std::unique(vec.begin(), vec.end());
18 vec.erase(last, v.end());
```

2.4 String streams

```
1 //c-style for complex parsing
2 int in;
3 sscanf(str.c_str(), "%d", &in)
```

```
4 //convert number to string -- itoa()
5 std::ostringstream os;
6 os << (int) i;
7 os.str();
8 //convert string to number(s) -- atoi() or atol()
9 std::istringstream is(str);
10 is >> i;
```

2.5 Constants

```
1 INT_MIN
2 INT_MAX
3 LLONG_MIN
4 LLONG_MAX
5 PI //defined in header
6 EPS //defined in header
```

2.6 Math

```
1 ceil(a); //round up
2 floor(a); //round down
3 round(a); //round nearest
4 atan2(a, b); //atan with two parameters
5 frac_part = modf(d, intpart); //split up in parts (int_part is a double!)
6 std::__gcd(a, b); //greatest common divisor a, b
```

2.7 Permutations

$O(N!)$

```
1 std::vector arr;
2 std::sort(arr.begin(), arr.end());
3 do {
4     //work with permutation
5 } while (next_permutation(arr.begin(), arr.end()));
```

2.8 Binary search

$O(\log(N))$

```
1 std::multimap<int, int> m;
2 m.lower_bound(i); //first element equal or higher then i
3 m.higher_bound(i); //first element higher then i
4 std::pair<auto iter, auto iter> p = m.equal_range(i); //get all with equal value
```

2.9 Bit twiddling hacks

```
1 in |= (1 << a); //enable bit at position a
2 in ^= (1 << a); //toggle bit at position a
3 if(in & (1 << a)); //check bit at position a
4 //get last bit set (least signifcant)
5 c = (b & -b)
6 //ll for the long long versions
7 __builtin_popcount(in); //count the amount of bits set
8 __builtin_ffs(in); //give the least signifcant index + 1 in binary representation
9 __builtin_clz(in); //returns the number of leading zeros
10 __builtin_ctz; //give number of trailing zeros
```

3 Datastructures

3.1 BIT

$O(\log(N))$

```

1 //use a std::map to save very large tables or for simple 2D (faster below)
2 int lst[MAXN];
3
4 int sum(int b){
5     int sum = 0; b+=1;
6     for(; b; b=(b&(-b))){
7         sum += lst[b];
8     }
9     return sum;
10 }
11
12 int update(int b, int v){
13     b+=1;
14     for(; b<MAXN; b+=(b&(-b))){
15         lst[b] += v;
16     }
17 }

```

3.2 2D-BIT

$O(N \log(N))$

```

1 //online  $O(N (\log(N))^2)$  is also possible using std::map<int, int>
2 std::pair<int, int> rs[MAXN]; //list of points (y,x)
3 std::vector<int> lst2[MAXN];
4 std::vector<int> order[MAXN];
5
6 int sum(int b, int c){
7     int sum = 0; b+=1; c+=1;
8     for(; b; b=(b&(-b))){
9         int h = upper_bound(order[b].begin(), order[b].end(), c) - order[b].begin();
10        for(int d = h; d; d=(d&(-d))){
11            sum += lst2[b][d];
12        }
13    }
14    return sum;
15 }
16
17 void update(int b, int c, int v){
18     b+=1; c+=1;
19     int unt = 0;
20     for(; b<MAXN; b+=(b&(-b))){
21         int h = upper_bound(order[b].begin(), order[b].end(), c) - order[b].begin();
22         for(int d = h; d<lst2[b].size(); d+=(d&(-d))){
23             lst2[b][d] += v;
24         }
25     }
26 }
27
28 void init(){
29     std::sort(rs, rs+MAXN);
30     for(int i=0; i<MAXN; ++i){
31         for(int j=rs[i].second+1; j<MAXN; j+=(j&(-j))){
32             order[j].push_back(rs[i].first+1);
33             lst2[j].push_back(0);
34         }
35     }
36     for(int i=0; i<MAXN; ++i) lst2[i].push_back(0);

```

37 | }

3.3 Segment tree

```

1 // inclusion segment tree (a node includes both it endpoints!)
2 struct Node{
3     Node(): li(0), ri(0), l(0), r(0) {}
4     int li;
5     int ri;
6     Node *l;
7     Node *r;
8
9     int val;
10
11     bool hupd;
12     int upd;
13 };
14
15 Node *r;
16
17 Node *build(int li, int ri) {
18     Node *n = new Node;
19     n->li = li;
20     n->ri = ri;
21     n->upd = 0;
22     if(li == ri){
23         n->val = 0; //init
24     }else{
25         int mi = (li+ri)/2;
26         n->r = build(li, mi);
27         n->l = build(mi+1, ri);
28
29         n->val = std::min(n->r->val, n->l->val); // merge
30     }
31     return n;
32 }
33
34 void pushd(Node *n){
35     if(!n->hupd) return;
36     n->l->val = n->upd; //update
37     n->r->val = n->upd; //update
38     n->l->upd = n->upd; //split (move old updates)
39     n->r->upd = n->upd; //split (move old updates)
40     n->l->hupd = true;
41     n->r->hupd = true;
42     n->hupd = false;
43     n->upd = 0;
44 }
45
46 int query(Node *n, int li, int ri){
47     if(ri < n->li || n->ri < li){
48         //outside
49         return INT_MAX;
50     }else if(li <= n->li && n->ri <= ri){
51         //inside
52         return n->val;
53     }
54     pushd(n);
55
56     int la = query(n->l, li, ri);

```

```

57     int ra = query(n->r, li, ri);
58     return std::min(la, ra); // merge
59 }
60
61 void update(Node *n, int li, int ri, int v){
62     if(ri < n->li || n->ri < li){
63         //outside
64         return;
65     }else if(li <= n->li && n->ri <= ri){
66         //inside
67         if(n->ri != n->li){
68             pushd(n);
69             n->upd = v; //split
70             n->hupd = true;
71         }
72         n->val = v; //update
73         return;
74     }
75     pushd(n);
76
77     update(n->l, li, ri, v);
78     update(n->r, li, ri, v);
79     n->val = std::min(n->r->val, n->l->val); // merge
80 }

```

3.4 Union-Find

 $O(\alpha)$

```

1  int pr[MAXN];
2  int sz[MAXN];
3
4  int find(int k){
5      if(pr[k] == k) return k;
6      else return pr[k] = find(pr[k]);
7  }
8  void merge(int a, int b){
9      a = find(a); b = find(b);
10     if(a == b) return;
11     if(sz[a] > sz[b]) std::swap(a, b);
12
13     pr[a] = b;
14     sz[b] += sz[a];
15     sz[a] = 0;
16 }

```

4 Math

4.1 GCD/LCM

 $O(\log(K))$

```

1  int gcd(int a, int b){
2      if(b == 0) return a;
3      return gcd(b, a%b);
4  }
5
6  int lcm(int a, int b){
7      return a*b/gcd(a,b);
8  }

```

4.2 Extended Euclidean

$O(\log(K))$

```

1 //determines a and b satisfying s*a+t*b == gcd(a, b)
2 std::pair<int, int> extgcd(int a, int b){
3     int s = 0, old_s = 1;
4     int t = 1, old_t = 0;
5     int r = b, old_r = a;
6     while(r){
7         int q = old_r/r;
8         int sv;
9         sv = r;
10        r = old_r - q*r; old_r = sv;
11        sv = s;
12        s = old_s - q*s; old_s = sv;
13        sv = t;
14        t = old_t - q*t; old_t = sv;
15    }
16    return std::make_pair(old_s, old_t);
17    //return s, t if you want with result = zero
18 }

```

4.3 Modular multiplicative inverse

$O(\log(K))$

```

1 //needs extended euclidean
2 int mod_inverse(int a, int m){
3     if(gcd(a, m) != 1) return -1; //inverse does not exist
4     int inv = extgcd(a, m).first;
5     if(inv < 0) {
6         int mlt = inv/m;
7         if(inv % m) mlt--;
8         inv -= mlt*m;
9     }
10    return inv;
11 }

```

4.4 Modular exponentiation

$O(\log(K))$

```

1 //trick also works with matrices
2 long long expmod(long long a, long long b, long long m){
3     long long res = 1;
4     a = a%m;
5     while(b > 0){
6         if((b%2) == 1) res = (res*a)%m;
7         b >>= 1;
8         a = (a*a) % m;
9     }
10    return res;
11 }

```

4.5 Fibonacci

$O(N)$

```

1 int fib(int n){
2     int a = 0, b = 1;
3     for(int i=0; i<n; ++i){
4         int t = b;
5         b = a+b;
6         a = t;
7     }

```

```

8     return a;
9 }

```

4.6 Combinations

 $O(N^2)$

```

1 int cmb[MAXN] [MAXN];
2 int comb(int i, int j){
3     if(i < 0) return 0;
4     else if(j == 0) return 1;
5     else if(cmb[i][j] != -1) return cmb[i][j];
6     else return cmb[i][j] = comb(i-1,j-1)+comb(i-1,j);
7 }

```

4.7 Gaussian elimination

 $O(N^3)$

```

1 int N; //amount of rows (columns = rows + 1)
2 double aug[MAXN] [MAXN];
3 double ans[MAXN];
4 void gaussian_elimination() {
5     // the forward elimination phase
6     for (int i = 0; i < N - 1; i++) {
7         int l = i;
8
9         // which row has largest column value
10        for (int j = i + 1; j < N; j++) if (fabs(aug[j][i]) > fabs(aug[l][i])) l = j;
11        // swap this pivot row, reason: minimize floating point error
12        for (int k = i; k <= N; k++){
13            double t = aug[i][k];
14            aug[i][k] = aug[l][k];
15            aug[l][k] = t;
16        }
17        // the actual forward elimination phase
18        for (int j = i + 1; j < N; j++)
19            for (int k = N; k >= i; k--) aug[j][k] -= aug[i][k] * aug[j][i] / aug[i][i];
20
21    }
22
23    // the back substitution phase
24    for (int j = N - 1; j >= 0; j--) {
25        double t = 0.0;
26        for (int k = j + 1; k < N; k++) t += aug[j][k] * ans[k];
27        ans[j] = (aug[j][N] - t) / aug[j][j];
28    }
29 }

```

4.8 Prime generation

 $O(N \log(\log(N)))$

```

1 std::vector<long long> prms;
2 long long is_prm[MAXN];
3
4 //generate primes needed for functions below
5 void init_prms(){
6     for(long long i=0; i<MAXN; ++i) is_prm[i] = true;
7     for(long long i=2; i<MAXN; ++i){
8         if(!is_prm[i]) continue;
9         prms.push_back(i);
10        for(long long j=2*i; j<MAXN; j+=i) is_prm[j] = false;
11    }

```


12 }
}

4.9 Is prime

$O(\sqrt{N})$

```
1 bool is_prime(long long n){
2     for(size_t i=0; i<prms.size(); ++i){
3         if(prms[i]*prms[i] > n) return true;
4         if((n % prms[i]) == 0) return false;
5     }
6     return true;
7 }
```

4.10 Factorize

$O(N \log(\log(N)))$

```
1 //returns list of factors and how many times they occur
2 std::vector<std::pair<int, int> > factors(long long n){
3     std::vector<std::pair<int, int> > v;
4     for(size_t i=0; i<prms.size(); ++i){
5         long long k = 0;
6         while((n % prms[i]) == 0){
7             ++k;
8             n/=prms[i];
9         }
10        if(k) v.push_back(std::make_pair(prms[i], k));
11    }
12    if(n!=1) v.push_back(std::make_pair(n, 1));
13    return v;
14 }
```

4.11 Is prime (fast)

$O(\text{polylog}(N))$

```
1 //deterministic until 2^64 -- needs long long expmod probably!
2 bool miller_rabin(long long n){
3     long long prms[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
4
5     long long s = 0, d = n-1;
6     while((d % 2) == 0){
7         d/=2;
8         s++;
9     }
10    for(int i=0; i<12; ++i){
11        if(prms[i] >= n) break;
12        if(expmod(prms[i], d, n) == 1) continue;
13        long long c = 1;
14        int j = 0;
15        for(j=0; j<s; ++j){
16            if(expmod(prms[i], c*d, n) == n-1) break;
17            c *= 2;
18        }
19        if(j == s) return false;
20    }
21    return true;
22 }
```

5 Graph

5.1 Header

```

1 struct Edge{
2     //add constructor if convenient
3     int f; //from
4     int t; //to
5     int d; //distance
6
7     int cap; //capacity (max flow)
8     int flw; //flow (max flow)
9     Edge *rev; //reverse edge (in case undirected)
10
11     bool use; //edge is used (euler-tour)
12 };
13
14 struct Node{
15     int n; //index
16     std::vector<Edge*> ed; //adjacency list
17 };
18
19 Node nd[MAXN]; //nodes
20 int dis[MAXN]; //distance to node
21
22 int dpt[MAXN]; //depth node (SCC/bridge)
23 int low[MAXN]; //low-link node (SCC/bridge)
24 //next edge of a node, used to keep track of next path in a non-recursive dfs
25 std::vector<Edge*>::iterator eit[MAXN];
26
27 Edge *frm[MAXN]; //edge used to node
28 bool vis[MAXN]; //visited
29
30 int incnt[MAXN]; //active indegree count (topo-sort)
31
32 void init(){
33     for(int i=0; i<MAXN; ++i){
34         nd[i].n = i;
35         for(size_t j=0; j<nd[i].ed.size(); ++j) delete nd[i].ed[j];
36         nd[i].ed.clear();
37         dis[i] = INT_MAX;
38         dpt[i] = -1;
39         low[i] = INT_MAX;
40         frm[i] = 0;
41         vis[i] = false;
42         incnt[i] = 0;
43     }
44 }

```

5.2 BFS

```

1 void bfs(int F){
2     for(int i=0; i<MAXN; ++i) dis[i] = INT_MAX;
3     std::queue<int> q; //replace by stack for dfs
4     dis[F] = 0;
5     q.push(F);
6     while(!q.empty()){
7         int p = q.front();
8         q.pop();
9         for(auto iter = nd[p].ed.begin(); iter != nd[p].ed.end(); ++iter){

```

$O(E+V)$

```

10         Edge *e = *iter;
11         if(dis[e->t] == INT_MAX){
12             dis[e->t] = dis[p] + 1;
13             q.push(e->t);
14         }
15     }
16 }
17 }

```

5.3 Dijkstra

$O((E+V)\log(V))$

```

1  int phi[MAXN];
2  #define POT(u,v) (phi[u] - phi[v])
3
4  void dijkstra(int F){
5      for(int i=0; i<MAXN; ++i) dis[i] = INT_MAX, phi[i] = 0;
6
7      std::priority_queue<std::pair<int,
↵ int>, std::vector<std::pair<int, int> >, std::greater<std::pair<int, int> > > > pq;
8      dis[F] = 0;
9      pq.push(std::make_pair(0, F));
10     while(!pq.empty()){
11         std::pair<int, int> p = pq.top();
12         pq.pop();
13         if(dis[p.second] != p.first) continue;
14         for(auto iter = nd[p.second].ed.begin(); iter != nd[p.second].ed.end(); ++iter){
15             Edge *e = *iter;
16             if(p.first + e->d + POT(p.first, e->t) < dis[e->t]){
17                 dis[e->t] = p.first + e->d + POT(p.first, e->t);
18                 pq.push(std::make_pair(dis[e->t], e->t));
19             }
20         }
21     }
22
23     //addition if we want to work with negative-weight paths (min-cost max-flow)
24     //for(int i=0; i<MAXN; ++i) if(phi[i] < INT_MAX) phi[i] += dis[i];
25 }

```

5.4 Floyd-Warshall

$O(V^3)$

```

1  int dism[MAXN][MAXN];
2
3  void floyd_warshall(){
4      for(int k=0; k<MAXN; ++k){
5          for(int i=0; i<MAXN; ++i){
6              for(int j=0; j<MAXN; ++j){
7                  //line depends on exercise (this is maximum edge on minimum path)
8                  dism[i][j] = std::min(dism[i][j], std::max(dism[i][k], dism[k][j]));
9              }
10          }
11      }
12 }

```

5.5 Bellman-Ford

$O(E V)$

```

1  bool bellman_ford(int F){
2      //compute shortest path
3      for(int i=0; i<MAXN; ++i) dis[i] = INT_MAX;
4      dis[F] = 0;

```

```

5   for(int k=0; k<MAXN-1; ++k){
6       for(int i=0; i<MAXN; ++i){
7           for(auto iter = nd[i].ed.begin(); iter != nd[i].ed.end(); ++iter){
8               Edge *e = *iter;
9               if(dis[i] + e->d < dis[e->t]){
10                  dis[e->t] = dis[i] + e->d;
11              }
12          }
13      }
14  }

15
16  //check for negative weight path
17  for(int i=0; i<MAXN; ++i){
18      for(auto iter = nd[i].ed.begin(); iter != nd[i].ed.end(); ++iter){
19          Edge *e = *iter;
20          if(dis[i] + e->d < dis[e->t]) return false;
21      }
22  }
23  return true;
24  }

```

5.6 Minimum spanning tree

$O(E \log(V))$

```

1  //needs union-find
2  bool cmp(Edge *e1, Edge *e2) { return e1->d < e2->d; }
3
4  std::vector<Edge*> kruskal(std::vector<Edge*> &edg){
5      std::vector<Edge*> vec;
6
7      std::sort(edg.begin(), edg.end(), cmp);
8      for(size_t i=0; i<edg.size(); ++i){
9          int a = find(edg[i]->f);
10         int b = find(edg[i]->t);
11         if(a != b){
12             vec.push_back(edg[i]);
13             merge(a, b);
14         }
15     }
16
17     return vec;
18 }

```

5.7 Biconnected components

$O(E+V)$

```

1  std::vector<int> biconnected(int r){
2      for(int i=0; i<MAXN; ++i) eit[i] = nd[i].ed.begin();
3      std::vector<int> res; //result nodes
4
5      std::stack<int> st; //stack
6      st.push(r);
7      dpt[r] = 0;
8      while(!st.empty()){
9          int c = st.top();
10         st.pop();
11
12         //add to result
13         if(eit[c] == nd[c].ed.end()){
14             int cnt = 0;
15             for(auto iter = nd[c].ed.begin(); iter != nd[c].ed.end(); ++iter){
16                 int n = (*iter)->t;

```

```

17         if(dpt[c] != 0 && low[n] >= dpt[c] && dpt[n] == dpt[c]+1){ //other edges
18             res.push_back(c);
19             break;
20         }else if(dpt[c] == 0 && dpt[n] == dpt[c]+1){ //root
21             dpt[n] = 0;
22             ++cnt;
23         }
24     }
25     if(dpt[c] == 0 && cnt >= 2) res.push_back(c); //root
26     continue;
27 }
28
29 //init
30 if(low[c] == INT_MAX) low[c] = dpt[c];
31
32 //loop through children (non-recursive so we should come back to this node)
33 st.push(c);
34 auto iter = eit[c];
35 for(; iter!=nd[c].ed.end(); ++iter){
36     int n = (*iter)->t;
37     if(dpt[n] != -1){
38         if(dpt[n] < dpt[c]-1) low[c] = std::min(low[c], dpt[n]); // back edge
39         else if(dpt[n] > dpt[c]) low[c] = std::min(low[c], low[n]); // forward edge
40     }else{
41         dpt[n] = dpt[c]+1;
42         st.push(n);
43         break;
44     }
45 }
46 eit[c] = iter;
47 }
48 return res;
49 }

```

5.8 Strongly connected components

$O(E+V)$

```

1  int tarjan_ind = 0;
2  std::vector<std::vector<int>> tarjan(int r){
3      for(int i=0; i<MAXN; ++i) eit[i] = nd[i].ed.begin();
4      std::vector<std::vector<int>> res; //result components
5
6      std::stack<int> st; //stack
7      std::stack<int> cp; //current component
8      st.push(r);
9      dpt[r] = ++tarjan_ind;
10     while(!st.empty()){
11         int c = st.top();
12         st.pop();
13
14         //init
15         if(low[c] == INT_MAX){
16             cp.push(c);
17             low[c] = dpt[c];
18         }
19
20         //add to result
21         if(eit[c] == nd[c].ed.end()){
22             if(low[c] == dpt[c]){
23                 res.push_back(std::vector<int>());
24                 while(true){

```

```

25         int n = cp.top();
26         dpt[n] = INT_MAX; low[n] = INT_MAX;
27         res.back().push_back(n);
28         cp.pop();
29         if(n == c) break;
30     }
31 }
32 continue;
33 }
34
35 //loop through children (non-recursive so we should come back to this node)
36 st.push(c);
37 auto iter = eit[c];
38 for(; iter!=nd[c].ed.end(); ++iter){
39     int n = (*iter)->t;
40     if(dpt[n] != -1){
41         if(dpt[n] < dpt[c]) low[c] = std::min(low[c], dpt[n]); // back edge
42         else if(dpt[n] > dpt[c]) low[c] = std::min(low[c], low[n]); // forward edge
43     }else{
44         dpt[n] = ++tarjan_ind;
45         st.push(n);
46         break;
47     }
48 }
49 eit[c] = iter;
50 }
51 return res;
52 }

```

5.9 2-SAT

$O(E+V)$

```

1 //needs tarjan (MAXN = 2x the amount of variables)
2 int asgn[MAXN]; //start from 2 all even indexes contain normal var
3
4 //give terms with negative as negotiation (make sure no zero used!)
5 int comp[MAXN];
6 int casg[MAXN];
7 bool two_sat(std::vector<std::pair<int, int> > terms){
8     for(int i=0; i<MAXN; ++i) comp[i] = casg[i] = -1;
9
10    //build graph
11    for(size_t i=0; i<terms.size(); ++i){
12        std::pair<int, int> p = terms[i];
13        Edge *e = new Edge();
14        if(p.first < 0) e->f = -2*p.first;
15        else e->f = 2*p.first+1;
16        if(p.second < 0) e->t = -2*p.second+1;
17        else e->t = 2*p.second;
18        nd[e->f].ed.push_back(e);
19        e = new Edge();
20        if(p.second < 0) e->f = -2*p.second;
21        else e->f = 2*p.second+1;
22        if(p.first < 0) e->t = -2*p.first+1;
23        else e->t = 2*p.first;
24        nd[e->f].ed.push_back(e);
25    }
26
27    //apply tarjan
28    std::vector<std::vector<int> > all;
29    for(int k=0; k<MAXN; ++k){

```

```

30         if(dpt[k] != -1) continue;
31         std::vector<std::vector<int>> > vec = tarjan(k);
32         for(size_t
↪ i=0; i<vec.size(); ++i) for(size_t j=0; j<vec[i].size(); ++j) comp[vec[i][j]] = i+all.size();
33         all.insert(all.end(), vec.begin(), vec.end());
34     }
35
36     //reverse topological traverse
37     for(size_t i=0; i<all.size(); ++i){
38         if(casg[i] == -1) casg[i] = true;
39
40         for(size_t j=0; j<all[i].size(); ++j){
41             int chk = comp[all[i][j]]/2;
42             if(comp[chk*2] == comp[chk*2+1]) return false;
43
44             if(all[i][j] % 2) casg[comp[all[i][j]-1]] = !casg[i];
45             else casg[comp[all[i][j]+1]] = !casg[i];
46         }
47     }
48
49     //set assignment (2 contains assignment var 1, 4 assignment var 2 etc...)
50     for(size_t
↪ i=0; i<all.size(); ++i) for(size_t j=0; j<all[i].size(); ++j) asgn[all[i][j]/2] = casg[i];
51     return true;
52 }

```

5.10 Max Flow

$O(V^2E)$

```

1 //bfs is safer (unless path length is limited and integer)
2 //for min-cost replace by Bellman-Ford or Dijkstra with potentials
3 int dfs(int a, int b){
4     if(a == b) return INT_MAX;
5     if(vis[a]) return 0;
6     vis[a] = true;
7
8     for(size_t i=0; i<nd[a].ed.size(); ++i){
9         Edge *e = nd[a].ed[i];
10
11         int cap = (e->cap-e->flw)+e->rev->flw;
12         if(cap == 0) continue;
13         else{
14             int k = dfs(e->t, b);
15             if(k == 0) continue;
16             frm[e->t] = e;
17             return std::min(cap, k);
18         }
19     }
20     return 0;
21 }
22
23 int max_flow(int a, int b){
24     int mf = 0;
25
26     frm[a] = 0;
27     while(true){
28         for(int i=0; i<MAXN; ++i) vis[i] = false;
29         int f = dfs(a, b);
30         if(f == 0) break;
31         mf += f;
32     }

```

```

33     int lst = b;
34     while(frm[lst]){
35         Edge *nr = frm[lst];
36
37         int rf = std::min(nr->cap-nr->flw, f);
38         nr->flw += rf;
39         nr->rev->flw -= f-rf;
40
41         lst = frm[lst]->f;
42     }
43 }
44 return mf;
45 }

```

5.11 Bipartite matching

 $O(VE)$

```

1  int mtch[MAXN]; //only the size of one side
2  bool aug(int n) {
3      if (vis[n]) return false;
4      vis[n] = true;
5
6      for (size_t i = 0; i < nd[n].ed.size(); i++) {
7          Edge *e = nd[n].ed[i];
8          //try match with edge (if available or previous can be rematched)
9          if (mtch[i] == -1 || aug(mtch[e->t])) {
10             mtch[e->t] = n;
11             return true;
12         }
13     }
14     return false;
15 }
16
17 void bipartite_matching() {
18     for (int i = 0; i < MAXN; i++) mtch[i] = -1;
19
20     int M = 0; //contains the maximum matching
21     for (int i = 0; i < MAXN; i++) { //try to start match from i
22         for (int j = 0; j < MAXN; j++) vis[j] = false;
23         if (aug(i)) M++;
24     }
25 }

```

5.12 Euler tour

 $O(V^2E)$

```

1  //start from one of the two non-even edges (if trail)
2  std::list<int> euler_tour(int s){
3      std::list<int> ans;
4      for(int i=0; i<MAXN; ++i) eit[i] = nd[i].ed.begin();
5
6      std::stack<int> st;
7      st.push(s);
8      while(!st.empty()){
9          int c = st.top();
10         st.pop();
11
12         auto iter = eit[c];
13         if(iter == nd[c].ed.end()){
14             ans.push_front(c);
15             continue;
16         }

```



```

17
18     st.push(c);
19     for(; iter != nd[c].ed.end(); ++iter){
20         Edge *e = *iter;
21         if(e->use) continue;
22         e->use = e->rev->use = true;
23         st.push(e->t);
24         break;
25     }
26     eit[c] = iter;
27 }
28
29 return ans;
30 }

```

5.13 Topological sort

$O(E+V)$

```

1  int ecnt[MAXN];
2
3  std::vector<int> toposort(){
4      std::vector<int> res;
5      std::queue<int> q;
6      for(int i=0; i<MAXN; ++i){
7          if(incnt[i] == 0) q.push(i);
8      }
9
10     while(!q.empty()){
11         int c = q.front();
12         q.pop();
13         res.push_back(c);
14         for(auto iter = nd[c].ed.begin(); iter != nd[c].ed.end(); ++iter){
15             Edge *e = (*iter);
16             --incnt[e->t];
17             if(incnt[e->t] == 0) q.push(e->t);
18         }
19     }
20     return res;
21 }

```

5.14 LCA

$O(E+V)$

```

1  //set MAXK to the log2 of MAXN
2  int par[MAXN][MAXK]; //example of sparse table idea
3  void construct(int k, int p = -1){
4      if(dpt[k] != -1) return;
5
6      if(p == -1) dpt[k] = 0;
7      else dpt[k] = dpt[p]+1;
8
9      //compute parents
10     par[k][0] = p;
11     for(int i=1; i<MAXK; ++i){
12         if(par[k][i-1] == -1) break;
13         par[k][i] = par[par[k][i-1]][i-1];
14     }
15
16     //dfs children
17     for(size_t i=0; i<nd[k].ed.size(); ++i){
18         int t = nd[k].ed[i]->t;
19     }

```

```

20     construct(t, k);
21 }
22 }
23
24 int query(int a, int b){
25     if(dpt[a] < dpt[b]) std::swap(a, b);
26
27     //level out
28     int k = 0;
29     for(; k<MAXK; ++k){
30         if(par[a][k] == -1 || dpt[par[a][k]] < dpt[b]) break;
31     }
32     --k;
33     while(dpt[a] != dpt[b]){
34         a = par[a][k];
35         while(k>=0 && (par[a][k] == -1 || dpt[par[a][k]] < dpt[b])) --k;
36     }
37
38     //go up
39     for(k=0; k<MAXK; ++k){
40         if(par[a][k] == par[b][k]) break;
41     }
42     --k;
43     while(k >= 0){
44         a = par[a][k];
45         b = par[b][k];
46         while(k >= 0 && par[a][k] == par[b][k]) --k;
47     }
48     //do last jump if necessary
49     if(a != b){
50         k++;
51
52         a = par[a][k];
53         b = par[b][k];
54     }
55
56     return a;
57 }

```

6 Geometry

6.1 Headers

```

1  #define PI (2*std::acos(0.0))
2  #define EPS 1e-9
3
4  struct Vec{
5      Vec(): x(0), y(0), z(0) {}
6      Vec(Vec p1, Vec p2): x(p2.x-p1.x), y(p2.y-p1.y), z(p2.z-p1.z) {}
7      Vec(double i, double j, double k = 0): x(i), y(j), z(k) {}
8      double x;
9      double y;
10     double z;
11
12     bool operator<(const Vec &o){
13         if(fabs(x - o.x) > EPS) return x < o.x;
14         else if(fabs(y - o.y) > EPS) return y < o.y;
15         else return z < o.z;
16     }
17     bool operator==(const Vec &o){
18         return (fabs(x-o.x) < EPS) && (fabs(y-o.y) < EPS) && (fabs(z-o.z) < EPS);

```

```

19     }
20 };
21 Vec operator+(const Vec &a, const Vec &b){
22     return Vec(a.x+b.x, a.y+b.y, a.z+b.z);
23 }
24 Vec operator*(const double d, Vec b){
25     return Vec(d*b.x, d*b.y, d*b.z);
26 }
27
28 struct Line{
29     Line(Vec i, Vec j): b(i), d(j) {}
30     Vec b; //base
31     Vec d; //direction
32 };
33 Line fromPoints(Vec i, Vec j){
34     return Line(i, Vec(i, j));
35 }
36
37 double dot(Vec a, Vec b){ //dot product
38     return a.x*b.x+a.y*b.y+a.z*b.z;
39 }
40 double cross(Vec a, Vec b){ //cross product (2D)
41     return a.x*b.y-a.y*b.x;
42 }
43 Vec cross_vec(Vec a, Vec b){
44     return Vec(a.y*b.z-b.y*a.z, a.z*b.x-b.z*a.x, a.x*b.y-a.y*b.x);
45 }
46 double len_sq(Vec a){ //give the squared length of a vector
47     return dot(a, a);
48 }
49 double len(Vec a){ //squares the squared length
50     return std::sqrt(len_sq(a));
51 }
52
53 double angle(Vec p, Line l){ //return the angle between the line and the point
54     Vec c(l.b, p);
55     return acos(dot(c, l.d)/std::sqrt(len_sq(c)*len_sq(l.d)));
56 }
57 bool colinear(Vec p, Line l){ //returns if a point is on a line
58     return fabs(cross(Vec(l.b, p), l.d)) < EPS;
59 }
60 double ccw(Vec p, Line l){ //return true if p is left of l
61     return cross(Vec(l.b, p), l.d) < 0;
62 }

```

6.2 Distances

```

1 double dist(Vec p1, Vec p2){
2     return hypot(p1.x-p2.x, p1.y-p2.y);
3 }
4
5 double dist(Vec p, Line l){
6     Vec c(l.b, p);
7     double u = dot(c, l.d)/len_sq(l.d);
8     if(u < 0.0) return dist(l.b, p);
9     else if(u > 1.0) return dist(l.b+l.d, p);
10    return dist(p, l.b+u*l.d);
11 }

```

0(1)

6.3 Intersect

0(1)

```

1 std::pair<bool, Vec> intersect(Line l1, Line l2){
2     Vec v = Vec(l1.b, l2.b);
3     double c = cross(l1.d, l2.d);
4     if(fabs(c) < EPS){
5         if((-EPS <= dot(v, l1.d) && dot(v, l1.d) <= len_sq(l1.d)+EPS) ||
6            (-EPS <= dot(v, l2.d) && dot(v, l2.d) <= len_sq(l2.d)+EPS))
7             return std::make_pair(true, Vec()); //colinear and touching
8             else return std::make_pair(false, Vec()); //either colinear but not touching or parallel
9     }else{
10        double t = cross(v, l1.d)/c;
11        double u = cross(v, l2.d)/c;
12        if(-EPS <=
↪ t && t <= 1+EPS && -EPS <= u && u <= 1+EPS) return std::make_pair(true, l1.b+u*l1.d); //intersects
13        else return std::make_pair(false, Vec()); //not intersecting
14    }
15 }
```

6.4 Area

0(N)

```

1 //returns signed area
2 double areaPolygon(const std::vector<Vec> &v){ //ADD FIRST POINT AGAIN!
3     double res = 0.0;
4     for(size_t i=0; i<v.size()-1; ++i){
5         res += v[i].x*v[i+1].y-v[i+1].x*v[i].y;
6     }
7     return res/2.0;
8 }
9
10 double areaCircle(double ab, double bc, double ca){
11     double pm = ab+bc+ca;
12     double sp = pm/2;
13     return std::sqrt(sp*(sp-ab)*(sp-bc)*(sp-ca));
14 }
```

6.5 Miscellaneous

0(N)

```

1 /* TEST THESE */
2
3 bool inPolygon(Vec pt, const std::vector<Vec> &v){//ADD FIRST POINT AGAIN!
4     double sum = 0.0;
5     for(size_t i=0; i<v.size()-1; ++i){
6         if(ccw(v[i+1], fromPoints(pt, v[i]))){ //extend this to handle border
7             sum += angle(v[i+1], fromPoints(pt, v[i]));
8         }else sum -= angle(v[i+1], fromPoints(pt, v[i]));
9     }
10    return fabs(fabs(sum) - 2*PI) < EPS;
11 }
12
13 Vec centroid(const std::vector<Vec> &v){//ADD FIRST POINT AGAIN!
14     Vec ans;
15     for(size_t i=0; i<v.size()-1; ++i){
16         double spc = v[i].x*v[i+1].y-v[i+1].x*v[i].y;
17         ans.x += (v[i].x+v[i+1].x)*spc;
18         ans.y += (v[i].y+v[i+1].y)*spc;
19     }
20     ans = (1/(6.0*areaPolygon(v)))*ans;
21     return ans;
22 }
```

```

23
24 double radiusInCircle(double ab, double bc, double ca){
25     double sp = 0.5*(ab+bc+ca);
26     return areaCircle(ab, bc, ca)/sp;
27 }
28
29 double radiusCircumCircle(double ab, double bc, double ca){
30     return ab*bc*ca/(4.0*areaCircle(ab, bc, ca));
31 }
32
33 std::pair<int, Vec> inCircleTriangle(Vec pa, Vec pb, Vec pc){
34     double r = radiusInCircle(len(Vec(pa, pb)), len(Vec(pb, pc)), len(Vec(pa, pc)));
35     if(fabs(r) < EPS) return std::make_pair(r, Vec());
36
37     double ratio = len(Vec(pa, pb))/len(Vec(pa, pc));
38     Vec pt = pb + (ratio/(1+ratio))*Vec(pb, pc);
39     Line l1 = fromPoints(pa, pt); //check this line
40
41     ratio = len(Vec(pb, pa))/len(Vec(pb, pc));
42     pt = pa + (ratio/(1+ratio))*Vec(pa, pc);
43     Line l2 = fromPoints(pb, pt); //check this line
44
45     return std::make_pair(r, intersect(l1, l2).second);
46 }

```

6.6 Convex-hull

```

1 _____ O(N log(N)) _____
2 Vec pivot(0, 0); //will contain the point that is used as pivot
3
4 bool angle_cmp(Vec p1, Vec p2){
5     Line l = Line(pivot, Vec(pivot, p2));
6     if(colinear(p1, l)) return dist(pivot, p1) < dist(pivot, p2);
7     return !ccw(p1, l);
8 }
9
10 std::vector<Vec> convex_hull(std::vector<Vec> v){ //DONT ADD FIRST POINT AGAIN
11     std::vector<Vec> ans;
12     if(v.size() <= 3) return v;
13
14     int s = 0;
15     for(size_t i=1; i<v.size(); ++i){
16         if(v[i].y+EPS < v[s].y || (fabs(v[i].y - v[s].y) < EPS && v[i].x < v[s].x)) s = i;
17     }
18
19     std::swap(v[0], v[s]);
20     pivot = v[0];
21     sort(++v.begin(), v.end(), angle_cmp);
22
23     ans.push_back(v.back()); ans.push_back(v[0]); ans.push_back(v[1]);
24     size_t i = 2;
25     while(i < v.size()){
26         size_t j = ans.size()-1;
27         if(ccw(ans[j-1], Line(ans[j], v[i]))) ans.push_back(v[i++]);
28         else ans.pop_back();
29     }
30     return ans;
31 }

```

7 String

7.1 Edit distance

 $O(N^2)$

```

1  int dp[MAXN][MAXN]; //edit distance dp table
2  inline int edit_distance(std::string a, std::string b){
3      dp[0][0] = 0;
4      for(size_t i=1; i<=a.size(); ++i) dp[i][0] = i;
5      for(size_t j=1; j<=b.size(); ++j) dp[0][j] = j;
6      for(size_t i=1; i<=a.size(); ++i){
7          for(size_t j=1; j<=b.size(); ++j){
8              dp[i][j] = std::min(dp[i][j-1], dp[i-1][j])+1; //add character: +1
9              if(a[i-1] == b[j-1]) dp[i][j] = std::min(dp[i][j], dp[i-1][j-1]); //same character: no cost
10             else dp[i][j] = std::min(dp[i][j], dp[i-1][j-1]+1); //replace character +1
11         }
12     }
13     return dp[a.size()][b.size()];
14 }
```

7.2 KMP

 $O(N)$

```

1  int bt[MAXN]; //back table created preprocessing pattern
2  void kmpPreprocess(std::string P) {
3      size_t i = 0, j = -1; bt[0] = -1;
4      while (i < P.size()) {
5          while (j >= 0 && P[i] != P[j]) j = bt[j];
6          i++; j++;
7          bt[i] = j;
8      }
9  }
10
11 void kmpSearch(std::string T, std::string P) {
12     kmpPreprocess(P); //preprocess first
13
14     size_t i = 0, j = 0;
15     while (i < T.size()) {
16         while (j >= 0 && T[i] != P[j]) j = bt[j];
17         i++; j++;
18         if (j == P.size()) {
19             //pattern is found in text at index i-j -- DO STUFF HERE
20             j = bt[j];
21         }
22     }
23 }
```

7.3 Suffix array

 $O(N \log(N)^2)$

```

1  //MAXN should be 2x normal
2  int SA[MAXN];
3
4  int RA[MAXN];
5  int tempRA[MAXN];
6  int SK;
7  bool cmp(int a, int b){
8      if(RA[a] == RA[b]) return RA[a+SK] < RA[b+SK];
9      else return RA[a] < RA[b];
10 }
11
12 void constructSA(std::string T){
13     T += '$';
```

```

14     int N = T.size();
15     for(int i=0; i<2*N; ++i) {
16         if(i < N) RA[i] = T[i];
17         else RA[i] = 0;
18         SA[i] = i;
19     }
20     int r = 0;
21     for(SK=1; SK<N; SK<=<=1){
22         std::sort(SA, SA+N, cmp); //can also use 2x counting sort
23
24         tempRA[SA[0]] = r = 0;
25         for(int i=0; i<N; ++i){
26             if(RA[SA[i]] != RA[SA[i-1]] || RA[SA[i]+SK] != RA[SA[i-1]+SK]) ++r;
27             tempRA[SA[i]] = r;
28         }
29
30         for(int i=0; i<N; ++i) RA[i] = tempRA[i];
31         if(RA[SA[N-1]] == N-1) break;
32     }
33 }

```

7.4 Longest common subsequence

$O(N)$

```

1  int LCP[MAXN]; //longest common prefix for suffix at position i
2
3  int PLCP[MAXN];
4  int PHI[MAXN];
5  void constructLCP(std::string T){
6      T += '$';
7      int N = T.size();
8
9      PHI[SA[0]] = -1;
10     for(int i=1; i<N; ++i) PHI[SA[i]] = SA[i-1];
11
12     int L = 0;
13     for(int i=0; i<N; ++i){
14         if(PHI[i] == -1) {
15             PLCP[i] = 0;
16             continue;
17         }
18         while(T[i+L] == T[PHI[i] + L]) L++;
19         PLCP[i] = L;
20         L = std::max(L-1, 0);
21     }
22     for(int i=0; i<N; ++i) LCP[i] = PLCP[SA[i]];
23 }

```

8 Other

8.1 Binary search (integer)

$O(\log(N))$

```

1  bool can(double f){
2      //add implementation here
3      return true;
4  }
5
6  int binsearch(int lo, int hi){
7      while(lo < hi){
8          int mid = (lo+hi)/2;
9

```

```

10         if(can(mid)) lo = mid+1;
11         else hi = mid;
12     }
13     return lo;
14 }

```

8.2 Binary search (floating-point)

$O(\log(N))$

```

1 double binsearch(double lo, double hi){
2     double mid, ans;
3     while(std::fabs(hi-lo) > EPS){
4         mid = (lo+hi)/2.0;
5         if(can(mid)) {
6             hi = mid;
7             ans = mid;
8         }else lo = mid;
9     }
10    return ans;
11 }

```

8.3 Counting sort

$O(N)$

```

1 const int MAXNUM = 100;
2 int ind[MAXNUM]; //contains new index after sort
3 int tmp[MAXN]; //temporary sort array
4
5 void counting_sort(std::vector<int> arr) { //stable counting sort
6     for(int i=0; i<MAXNUM; ++i) ind[i] = 0;
7
8     for(size_t i=0; i<arr.size(); ++i){
9         ind[arr[i]]++;
10    }
11    int sum = 0;
12    for(int i=0; i<MAXNUM; ++i){
13        int t = ind[i];
14        ind[i] = sum;
15        sum += t;
16    }
17    for(size_t i=0; i<arr.size(); ++i){
18        tmp[ind[arr[i]]++] = arr[i];
19    }
20    for(size_t i=0; i<arr.size(); ++i){
21        arr[i] = tmp[i];
22    }
23 }

```

8.4 Generate permutations

$O(N^K)$

```

1 std::list<std::vector<int> > gen_permutation(std::vector<int> &opts, int sz){
2     std::list<std::vector<int> > all;
3     if(sz == 0){
4         for(size_t i=0; i<opts.size(); ++i) all.push_back({opts[i]});
5         return all;
6     }
7     for(size_t i=0; i<opts.size(); ++i){
8         std::list<std::vector<int> > lst = gen_permutation(opts, sz-1);
9         for(auto iter = lst.begin(); iter != lst.end(); ++iter) iter->push_back(opts[i]);
10        all.insert(all.end(), lst.begin(), lst.end());
11    }

```



```

12     return all;
13 }

```

8.5 Generate combinations

$O(N \binom{N}{k})$

```

1  std::list<std::vector<int> > gen_combinations(std::vector<int> &opts, int k){
2      std::vector<bool> has(opts.size());
3      std::fill(has.begin() + k, has.end(), true);
4
5      std::list<std::vector<int> > all;
6      do {
7          std::vector<int> comb;
8          for (size_t i = 0; i < has.size(); ++i) {
9              if (!has[i]) comb.push_back(opts[i]);
10             }
11             all.push_back(comb);
12         } while (std::next_permutation(has.begin(), has.end()));
13
14         return all;
15     }

```

8.6 LIS

$O(N \log(N))$

```

1  int ldp[MAXN]; //lis dp array
2  int lin[MAXN]; //lin[inp.size()-1] contains last index
3  int frm[MAXN]; //contains index number came from
4  int lis(std::vector<int> inp){ //non-decreasing lis
5      for(int i=0; i<MAXN; ++i) ldp[i] = INT_MAX;
6
7      for(size_t i=0; i<inp.size(); ++i){
8          int k = std::upper_bound(ldp, ldp+MAXN, inp[i])-ldp;
9          ldp[k] = inp[i];
10         lin[k] = i;
11         if(k == 0) frm[i] = -1;
12         else frm[i] = lin[k-1];
13     }
14
15     return std::upper_bound(ldp, ldp+MAXN, INT_MAX-1)-ldp;
16 }

```