Table of contents

Time complexity

Harmonic series: 1/1 + 1/2 + 1/3 + … + 1/n = ln(n)

n	Worst AC Algorithm (10^8 in 3s)	Examples
<=10..11	O(n!), O(n^6)	Exhausting permutations
<=15..18	O(2^n * n^2)	DP TSP
<=18..22	O(2^n * n)	DP Bitmask
<=100	O(n^4)	DP 3 dimensions
<=400	O(n^3)	Floyd Warshall
<=2000	O(n^2 logn)	2 nested for loops + Tree struct.
<=10^4	O(n^2)	Bubble/Selection/Insertion sort
<=10^6	O(nlogn)	Merge sort, Segment tree
<=10^8	O(n), O(1) or O(logn)	IO bottleneck: n<=10^6

Master theorem

For an algorithm that run in time complexity $T(n)$,

If $T(n)$ can be written as a recurrence formula in form of $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

If we divide original problem into a subproblems, parameter reduced from n to $\frac{n}{b}$

We have $T(n) = \begin{cases} O(n^d) & if\ d > \log_b a \\ O(n^d logn) & if\ d = \log_b a \\ O\left(n^{\log_b a}\right) & if\ d < \log_b a \end{cases}$

check.bat

```
@echo off
:loop
    gen.exe > data.in
    std.exe < data.in > std.out
    main.exe < data.in > main.out
    fc std.out main.out
if not errorlevel 1 goto loop
pause
goto loop
```

Guide to solving OI Problems: **Draft first, then code;** Total complexity ≤ $2.5*10^7 = 5000^2$

'The sooner you start to code, the longer the program will take'
'If you have too many special cases, then you are doing it wrong'

- What question type is it? Graph 以小見大? DP 以小推大? Greedy? Observation? Implementation?
- Simulate small test cases (by hand/machine) first, Write out the equation (greedy)
- Add objects one by one (recurrence relation): Question reduction
- Before submit: `long long`? **fixed**? ÷0? Odd/Even? Special cases (0/1)? Var. name (`i/j`, `n/m`)? Sorted?
- Problem reading: Contiguous? More vs at least? Permutation/Sequence/Subarray?
- Binary search on answer? Sorting? Two pointers? Partial sum? BIT?
- Problem solving techniques: Backwards solving, Simulation, Observation, Separate by type (small k))
- Optimization: Queries: Precomputation/sorting, batching, finding cycles, "change of axis"/summarize
- Divide&Conquer: Assume subproblem already solved, how to combine subproblems to main problem

0. STL: (Enabling C++11: Tools/Compiler Options/Settings/Code Generation/Language Standard/GNU)

'I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it.' - Bill Gates
'The standard library saves programmers from having to reinvent the wheel.' – Bjarne Stroustrup

Code Compilation

```
g++ -static -std=c++11 -Wno-unused-result -fno-optimize-sibling-calls
-fno-strict-aliasing -DONLINE_JUDGE -lm -s -O2 -o program.exe program.cpp
```

Template (main.cpp)

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#define pii pair<int,int>
#define pb push_back
#define mp make_pair
#define mt make_tuple
#define DEBUG 0
#define cerr if (DEBUG) cerr
#define fprintf if (DEBUG) fprintf
#define local freopen("in.txt","r",stdin);
#define test cerr<<"hi\n";
#define tr(x) cerr<<#x" "<<x<<'\n';
#define fi first
#define se second
#define HH1 402653189
#define HH2 1610612741
#define INF 0x3f3f3f3f
#define tii tuple<int,int,int>
#define npos string::npos

using namespace std;

const double PI = 2 * acos(0.0);

int main(){
    //freopen("input.txt","r",stdin);
    //freopen("output.txt","w",stdout);
    ios_base::sync_with_stdio(0);   // cin.tie(0);

}
```

Containers: bool empty();, void clear();

```
// vector<T>: Dynamically sized array;
// Can compare using lexicographical order: < <= == >= > !=
vector<int> a;  vector<int> a(100);  vector<int> a(n, 100);
vector<int> a = {2,4,2,5,7};    vector<vector<int> > a;
cout << v.back();  // returns last element
vector<vector<int>> a(n, vector<int> (m)); // 2D vector size (n,m)
a.push_back(123); // do not use a[x]=i on unitialized size
for (auto it = a.begin(); it! = a.end(); ++it)  sum+=*it;
for (auto x:a)  sum+=x;
a.erase(a.begin()+2); // 3rd only;  a.erase(a.begin(),a.begin()+2); on range
a.insert(a.begin()+2, 200);   // insert element before that pointed by Itr
```

```
// pair<T1,T2>: Store two data of type T1 and T2
// Can be compared (use .fi first)
pair<string, int> p; p = {"Sowon",173}; p=mp("a",0);
p.first = "Sowon", p.second = 173;
```

```
struct node{
    int count;
    int child[26];
}; vector<node> trie;
```

```
// list<T>: Bidirectional linked list;
// O(1) insertion, O(1) erase, No random access
list<int> l{10,20};
l.push_back(30);  l.push_front(0);
cout << l.front() << endl; // 0
cout << l.back() << endl;  // 30
auto it = l.begin();   // >0  10  20  30
++it;                  //  0 >10  20  30
it=l.erase (it);       //  0 >20  30
++it;                  //  0  20  >30
it=l.insert(it,25);    //  0  20  >25  30
l.insert(l.end(), 40); //  0  20  >25  30 40
cout << *it << endl;   //  25
for (auto it=l.rbegin(); it!=l.rend(); ++it){
    cout << *it << ' '; // 40 30 25 20 0
}
l.sort();  // sorts linked list, O(nlogn)
```

```
tuple <int,char> foo (10,'x');
auto bar = make_tuple ('x', 14);
get<1>(bar) = 100;
char c = get<0>(bar); int x;
tie (x, c) = foo;
tie (ignore, x) = bar;
```

```
// deque<T>: Double-ended queue (Re: P004 Broken Keyboard, M1313 Bookstack)
// O(1) push_front/pop_front/push_back/pop_back
// O(1) random access, NOT contiguous memory. About 3x slower than vector
// stack<T>: push_back -> push, pop_back -> pop, back -> top
// queue<T>: push_back -> push, pop_front -> pop
deque<int> dq {4,5,6};
dq.push_front(3);
dq.push_back(7);
cout << dq[2] << endl; // 5
dq.pop_front();
cout << dq[2] << endl; // 6
```

```
Stack: First in last out
1. Balanced parantheses
2. Expression evaluation
3. SCC   4. Graham's scan

Queue: First in first out
0. Josephus Problem
1. BFS
```

```
// priority_queue<T>: Max heap
// O(1) top(), O(logn) pop() and push()
priority_queue<int> pq;
pq.push(1), pq.push(2), pq.push(3);
cout << pq.top() << endl; // 3
pq.pop();   cout << pq.top() << endl; // 2
priority_queue<int, vector<int>, greater<int> > pq2; // Min heap
auto cmp = [](pii x, pii y) {return x.se > y.se;};   // Custom comparator
priority_queue<pii, vector<pii>, decltype(cmp) > pq3 (cmp);
```

```
//bitset<L>: L 0/1 bool values. For small values (e.g. alphabet) use int
bitset<10000000> bs; bs[0]=1;    bitset<10> s(string("0010011010"));
for (int i=0; i<n; i++).    bitset<10> s(42);   // exhaust subset, read r->l
    cin >> x, bs|=bs<<x;
int  x.count();  // Return number of ones
void s.set(); s.reset(); s.flip();  // Applied on all bits
bool s.all(); s.any(); s.none();    // Check whole range of bits
```

<set>, <map>: O(logn) find, head, insert, erase; Unordered set/map: O(1); Multiset: O(# of key) count

Frequency map: `m[key]++; if (!--m[key]) m.erase(key);` Multimap: `map<int,set>`

Re: http://codeforces.com/blog/entry/21853; `distance(it1, it2)` is O(it2-it1) for BdrIt

```
set<int> s={2,5,6,8}; // Initialization
cout << s.insert(7).se;  // returns {BdIt, bool},  7 was actually inserted
cout << s.insert(2).se;  // returns {BdIt, false}, 2 was already inside s
cout << s.count(7);   // 1
s.erase(8); s.erase(10); // s now contains {2,5,6,7}
cout << s.size();     // 4
cout << *s.begin();   // 2, s is always sorted
cout << *s.rbegin();  // 7
cout << (s.find(11)==s.end()); // returns iterator to element
auto it = *s.lower_bound(x);
auto it2 = *m.find(k); // returns iterator to pair<const K, M>
```

```
struct Hasher {
    size_t operator()(pii x) const{
        return hash<ll>()( ((ll)x.fi) ^ (((ll)x.second)<<32) );
}};
unordered_set<pii, Hasher> us;
us.reserve(1024); // power of 2, depends on # of insert
us.max_load_factor(0.25);
```

```
multiset erase: if (s.count(x)) s.erase(s.find(x)); // single, .erase; //all
```

Policy based data structures: (Re: https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/)

Has same functions as <set> and <map>. For <multiset>, insert <pii> with key and id

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> indexed_set;

indexed_set a;

int main(){
    a.insert(7); a.insert(3); a.insert(2); a.insert(9);
    auto x = s.find_by_order(2);  // return It to element at pos, *x == 7
    cout << s.order_of_key(7) << endl;  // 2
    cout << s.order_of_key(6) << endl;  // 2
    cout << s.order_of_key(8) << endl;  // 3
}
```

pb_ds priority_queue: (Re: https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/pq_design.html)

Has same functions as std::priority_queue, but with modify, erase, join

```
#include <ext/pb_ds/priority_queue.hpp>
__gnu_pbds::priority_queue<int> pq;
auto it = pq.push(0); pq.modify(it, 3);
pq.erase_if(bind1st(less<int>(), 500));  pq.join (pq2);
```

ext/rope: (Re: http://codeforces.com/blog/entry/10355). O(logn) for all operations

Insert arbitrary blocks of array to any position and erase them; 2x slower than implicit cartesian tree

```
#include <ext/rope>
using namespace __gnu_cxx;
int main(){
    rope<int> v;  v.pb(123); rope<int> cur = v.substr(1, r - 1 + 1);
    v.erase(1, r - 1 + 1);
    v.insert(v.mutable_begin(), cur);
}
```

<algorithm>: Range is defined as [first, last) [l, r) where last/r is the element *past* the last element

```
all_of/any_of/none_of (InputIt l, InputIt r, UnPred p)
  • is_permutation (FwdIt l1, FwdIt r1, FwdIt l2, [BinPred p ==])
  • equal (InputIt l1, InputIt r1, InputIt l2, [BinPred p ==])
  • is_sorted (FwdIt l1, FwdIt r1, FwdIt l2, [BinPred p <])
  • find_if (InputIt l, InputIt r, UnPred p);   // returns iterator
  • max/min_element (FwdIt l, FwdIt r, [cmp <]) // returns iterator
  • find_end (FwdIt l1, FwdIt r1, FwdIt l2, FwdIt r2, [BinPred p ==])
      ○  // returns la1 if not found; pos = it – l1;
  • count    (InpIt l, InpIt r, T value)
  • count_if (InpIt l, InpIt r, UnPred p)
  • accumulate (InpIt l, InpIt r, T init, [BinPred])
      ○  [](string a, int b){return a + to_string(b);})
  • for_each (InpIt l, InpIt r, UnPred p);
  • transform (InpIt l, InpIt r, OutIt l, UnOp op);
      ○  [](T c) -> T { return c; }
```

```
Un/Bin Predicate:
lambda function

[=](type foo){
    return val;
}

dist = it2 – it1;
```

```
reverse (BdrIt l, BdrIt r);
  • random_shuffle (RdmIt l, RdmIt r);
  • rotate (FwdIt l, FwdIt n_l, FwdIt r);
  • prev/next_permutation (BdrIt l, BdrIt r);
  • unique (FwdIt l, FwdIt r, [BinPred p ==]); // return It to range end()
      ○  auto la = unique (v.begin(), v.end()); v.erase(la, v.end());
  • copy (FwdIt l, FwdIt r, back_inserter(to_vector), [UnPred p]);
  • swap_ranges (FwdIt l1, FwdIt r1, FwdIt l2);  [iter_]swap
```

```
fill (FwdIt l, FwdIt r, const T& value);
  • iota (FwdIt l, FwdIt r, T value); // starting from value, +1 @
  • memset(a, byte, sizeof(a));
  • memcpy(dest, source, sizeof (dest));
```

```
set_difference/intersection/union/symmetric_difference // sorted array
  • (InpIt l1, IptIt r1, InpIt l2, InpIt r2, back_inserter(to_vector));
```

<cstdio>, <iostream>: Input format

```
while (gets(s) != NULL)
while (scanf("%d", &x))  while (cin >> n >> m)
sprintf(s, "%d", 18); sscanf("06", "%d", &x);
fflush(stdout);
fprintf(stderr,"%d",debug);  cerr << debug
scanf(" %c", &c); // mind the space
cin.ignore(1024, '\n');
```

```
string line, word[100];
getline(cin, line);
stringstream ss (line);
while (ss >> word[i])
    i++;
```

```
string s = to_string(123);
int x = stoi("123"); // x= stoi(s, nullptr, base)

char buf[30]; int x;
itoa(x,buf,16); // convert to base 16 c_string
```

```
double: in %lf, out %.12lf   append .0     long double: i/o %Lf   append L
long long: %lld/ %l64d       append LL
fgets (x, sizeof(x), stdin);
scanf("\n%s", t1); // read cstring on next line    s=(string) t1;
scanf("\n%[^\n]", t1); // gets, scanset/negated scanset
```

```
5: at least 5 char;   -5: left align
05: 0 padded til 5;   .5: at least 5 digit
```

Floating point (Re: Precision error, M1702 Archery). Use long double / pii fraction / expand if needed

```
a==b   fabs(a-b) <  eps
a> b       a >  b+eps
a< b       a+eps <  b
```

```
double ceil() floor() trunc()    (int)
a>=b       a >= b+eps
a<=b       a+eps <= b
```

<string>

```
string s = "abcdefg";  // abcdefgcd
string t (a,5);        // aaaaa
s+="cd"; cout << s << endl;       // abcdefgcd
cout << s.length() << endl;       // 9
cout << s.empty() << endl;        // 0
cout << s.substr(2) << endl;      // cdefgcd
cout << s.substr(1, 4)  << endl;  // bcde
cout << s.substr(5, 10) << endl;  // fgcd    last k char: .substr(len-k,k);
cout << s.find("cd") << endl;     // 2, O(nk)
cout << s.find("cd",2) << endl;   // 2, O(nk)
cout << s.rfind("cd") << endl;    // 7, O(nk)
cout << (s.find("xyz") == string::npos); // 1
s.c_str(); // return c character array object

int c=0; for (int i=s.find(sub); i!=npos; i=s.find(sub,i+1)) c++;  // count

s.erase(2,3);     // abfgcd (pos, len)
s.insert(3,'!');  // abf!fcd,  insert before pos
```

```
s.find_last_of  ("cd", pos=s.length())  // 8, O(nk)   find_last_not_of
s.find_first_of ("cd", pos=0)           // 2, O(nk)   find_last_not_of
```

Bitwise operations (Re: http://codeforces.com/blog/entry/15643 C++ Tricks) (unsigned long long)

```
__builtin_clzll (x)        // Number of leading  0s from most  sig. 1-bit
__builtin_ctzll (x)        // Number of trailing 0s from least sig. 1-bit
__builtin_popcountll (x)   // Number of 1-bits in x
__builtin_ffsll (x)        // Index (1 based) of least sig. 1 bit  (0->0)
63 - __builtin_clzll(x)    // Index (0 based) of most sig. 1 bit   (0->63)
```

```
Turn on the j-th item (0 based): OR |
    S |= 1 << j
Check if the j-th item of the set is on: AND &
    S & 1 << j
Turn off the j-th item of the set: NOT ~
    S &= ~(1 << j)
Toggle a bit: XOR ^
    S ^= (1 << j)
Value of the least significant 1 bit
    T = (S & (-S))
Turn on all bits in a set of size n
    S = (1 << n) - 1
```

```
Modulus by N (2^k)
    S & (N-1)
Test if power of 2:
    (S & (S-1)) ==0
Turn off last 1 bit:
    S & (S-1)
Turn on last 0 bit
    S | (S+1)
Turn off last consec. 1
    S & (S+1)
Turn on last consec. 0
    S | (S-1)
```

Gray code: Avoid "transitional states" between two consecutive binary numbers, change 1 bit @

```
unsigned int BinaryToGray (unsigned int num) { return num ^ (num >> 1); }
unsigned int GrayToBinary (unsigned int num) {
    unsigned int mask = num >> 1;
    while (mask != 0) num = num ^ mask, mask = mask >> 1;
    return num;
}
```

Random

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
shuffle(permutation.begin(), permutation.end(), rng);

for (int i = 1; i < N; i++)
    swap(permutation[i], permutation[uniform_int_distribution<int>(0, i)(rng)]);
```

Runtime: To test clock, use volatile int for-loop

```
auto T = clock();
cout << double(clock() - T) / CLOCKS_PER_SEC << endl;   // output running time
```

1. Searching and Sorting, Optimization

When to sort: Processing order, Greedy, Optimal order (cmp), Properties, nth_element, hang seng index

Binary Search (on function/answer): Efficient way to check possibility, (faster brute force)

Re: Seating Plan/Avoiding the crows, GCJ18 Bit Party, Optimal Bowing (DP); Finding roots of equation

```
int lo = N_MIN - 1, hi = N_MAX + 1;
while (hi – low > 1){           // while (fabs(hi-lo) > EPS), or forloop iterations
    int mi = (lo + hi) / 2;     // binary_search: elements partitioned into 2:
    if (check(mi)) hi = mi;     // put hi or lo here: see which side is true
    else lo = mi;               // i.e. now is FFTT, swap hi and lo for TTFF
}
return hi;          // lo/hi: LHS/RHS of check() turning pt, exceed if none
```

Binary Search STL: distance is O(1) for RndAccIt, O(n) for FwdIt; arr is sorted/partitioned w.r.t. val

Static lower_bound: O(n) precomputation, index array (Re: S073 SOS): if (a[i]==-1) a[i]=a[i+1]

```
int a[10] = {1, 3, 5, 5, 5, 7, 9, 11, 11, 13};
cout << binary_search(a, a+10, 5) << endl; // 1
cout << binary_search(a, a+10, 8) << endl; // 0
cout << distance(a, lower_bound(a, a+10, 5)) << endl; // 2   1st element >= 5
cout << distance(a, lower_bound(a, a+10, 8)) << endl; // 6
cout << distance(a, upper_bound(a, a+10, 5)) << endl; // 5   1st element > 5
cout << distance(a, upper_bound(a, a+10, 8)) << endl; // 6
cout << lower_bound(a, a+10, 21) - a << endl; // 10      lb-- for largest <5
```

Ternary Search (Optimization on function y=f(x), single peak and strictly sloping)

```
double lo = lb, hi = ub, m1, m2;
while (hi – lo > EPS){
    m1=(lo * 2 + hi) / 3;
    m2=(lo + hi * 2) / 3
    if (f(m1) > f(m2)) lo = m1;
    else hi = m2;
}
printf("Minimum %.12lf\n",lo);
```

```
Find largest x s.t. f(x)<f(x+1), ans=x+1
int lo=min-1, hi=max+1;
while (hi-lo>1){
    int m=(hi+lo)/2;
    if (f(x)<f(x+1)) lo=m;  // 1st half T
    else hi=m;
}
cout<<hi<<endl;
```

Sliding window (dynamic); Find min len subarray containing m distinct elements

Two pointers: Think about eliminating areas that the answer must not appear in the array

```
for (int i=0, j=0; i<n;i++){ // minimum length range s.t. contain m distinct
    for (j=j;j<n && s.size()<m;j++){ s[a[j]]++; } // slide right until ok
    if (s.size()==m && j-i < mc){    // current interval is ok, update ans
        mc=j-i;   ans={i,j};
    }
    if (!--s[a[i]]) s.erase(a[i]);   // update for sliding left by 1
}
cout << ans.fi +1 << " " << ans.se;  // originally is 0 based open interval
```

Sliding window (fixed)

```
for (int i=0;i<min(n,m);i++) s[a[i]]++; ans=s.size();  // ans for [0..k)
for (int i=0;i+m<n;i++){   // don't let right reach over end of array
    if (!--s[a[i]]) s.erase(a[i]);   // update for sliding left by 1
    s[a[i+m]]++;   ans=max(ans,(int)s.size());   // slide right by 1
}
cout << ans;
```

Unique with index

```
vector<pair<int, pii> > v;   // {val, [range)}
v.pb({a[0],{0,-1}});

for (int i=1;i<n;i++)
    if (a[i]!=a[i-1]) v.back().se.se=i, v.pb({a[i],{i,n}});
```

Sorting STL (return void)

Sorting queries: Think of "natural greater than" from eliminating already processed items, then bsearch

```
sort (RndIt l, RndIt r, [cmp]);   // use lambda function for cmp
sort (v.rbegin(), v.rend()); // Sort in reverse
stable_sort (RndIt l, RndIt r, [cmp]);     // Keep order of == elements
nth_element (RndIt l, RndIt nth, RndIt r); // cout << v[v.size()/2]; // median
```

Selection, Bubble Sort

```
// Selection sort: Min # pairwise swaps (Re: Cycles in sorting, discretize)

for (int i=n-1;i>=1;i--){
    int maxj=0;
    for (int j=1;j<=i;j++){
        if (arr[j]>arr[maxj]) maxj=j;
    }
    swap(arr[i],arr[maxj]);
}

// Bubble sort: Min # of adjacent swaps (inversions)

for (int i=n-1;i>=1;i--){
    for (int j=0;j<i;j++){
        if (arr[j]>arr[j+1]) swap(arr[j],arr[j+1]);
} }
```

Finding inversions (Min # of adjacent swaps, # {I,j} s.t. a[i] > a[j], i<j) using BIT

- Merge sort: At any step in merge(): if (a[i] > a[j]), there are (mid – i) inversions. Left right subarrays are sorted, so all remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) are greater than a[j]
- AVL tree: Each node also stores the subtree size; When inserting, if a[i] is smaller than current node, we increase inversion count by 1 plus the number of nodes in right subtree of current node.

```
// BIT inversion index, suitable for a[i]<=1e6, otherwise discretize first

int ans = 0;
for (int i = 1; i <= n; i++)
    upd(a[i], 1), ans += ask(a[i] - 1);

// Triplet inversion count:
1. Store an x[i] for #elements >a[i] in front of i
2. Perform inversion index again, but increase by x[i] instead of 1
```

Discretization

For intervals, try storing and *sorting* the start/end points with pii {pos, 1/-1} instead.

```
vector<int> dis; map<int, int> m; int val[100005];

for (int i = 0; i < n; i++) dis.pb(a[i]);
sort(dis.begin(), dis.end());
dis.erase(unique(dis.begin(),dis.end()), dis.end());
for (int i = 0; i < dis.size(); i++) m[dis[i]] = i, val[i] = dis[i];
```

2. Data Structures (Re: Algorithm Gym)

Partial Sum (1D/2D/Diagonal): Update: O(n), Query: O(1): Increasing/Monotonic, so can b-search

```
for (int i=1;i<=n;i++){
    ps[i]=ps[i-1]+a[i-1];   // ps[0]==0
partial_sum(a,a+n,ps+1);
cout << ps[4]-ps[0]; // returns sum(a[0..4) = a[0..3])
```

Difference Array (1D/2D): Update: O(1), Query: O(n): To regen, use partial sum

```
for (int i=0;i<n;i++)
    df[i]=a[i]-(i==0 ? 0 : a[i-1]);
df[a]+=v, df[b+1]-=v; // increase [a,b] by v. df[] is raise all from here
for (int cur=0, i=0;i<n;i++)
    cur+=df[i]; // cout << cur;  true val in array
```

1D Binary Indexed Tree: Update: O(logn), Query: O(logn); 1-based

2D BIT: Use a 2D array and 2 for loops, or use sparse tree

Lower bound: https://codeforces.com/blog/entry/61364?locale=en

```
int T[MAX_N];  memset(T,0,sizeof(T));

void upd(int x, int val){    // increase a[x] by val
    for (x; x <= n; x += x & -x)  T[x] += val;
}   // upd(l, val); upd(r+1, -val);

int ask(int p){     // sum of a[1..p] (closed interval)
    int ans = 0;
    for (int i = p; i > 0; i -= i & -i)  ans += T[i];
    return ans;
}   // return ask(y) – ask(x-1)

void init(int n){
    for (int i = 1; i <= n; i++){
        int j = (i + (i & -i)); F[i] += a[i];
        if (j <= n) F[j] += F[i];
    }
}

int lower_bound(int v) { // calling lower_bound on ps[]
    int sum = 0, pos = 0;
    for(int i = LOGN; i >= 0; i--) {
    if (pos + (1 << i) < MAXN && sum + T[pos + (1 << i)] < v) {
        sum += T[pos + (1 << i)];
        pos += (1 << i);
    } }
    return pos + 1; // 'pos': position of largest value less than 'v'
}
```

BIT Range update & Range query: Use 2x BIT

```
update(A, B, V):          // closed interval
    T1.add(A, V)          // add V to any X>=A
    T1.add(B+1, -V)       // cancel previously added V from any X>=B+1

    T2.add(A, (A-1)*V)    // add a fix for (A-1)s V didn't exist before A
    T2.add(B+1, -B*V)     // remove the fix, add more (B-A+1)*V to any query
                          // X>=B+1. => -(A-1)*V -(B-A+1)*V => -B*V

sum(X):
    return T1.sum(X)*X - T2.sum(X)
```

Interval tree offline query: How many segments in this interval? (1-based, closed interval) (CF 369E)

```
#define MAXN 1000006        // MAXN: max length
int BIT[MAXN+10],ans[MAXN];     // ans of query
int n,q;
vector<pair<pii,int> > query;   // query[L] = {R, id_query}
vector<pii> seg;

void upd(int x){
    for (; x<=MAXN; x+=x&-x) BIT[x]++;
}

int ask(int x){
    int ret=0;
    for (; x>0; x-=x&-x) ret+=BIT[x];
    return ret;
}

void work(vector<pii> &seg, vector<pair<pii,int> > &query){
    vector<int> *_e = new vector<int>[MAXN+1];
    vector<pii> *_q = new vector<pii>[MAXN+1];

    for (auto x:seg)   _e[x.fi].pb(x.se);
    for (auto x:query) _q[x.fi.fi].pb({x.fi.se, x.se}); // x.se: id

    for (int i=MAXN; i>=1; i--){
        for (int x:_e[i]) upd(x);
        for (auto x:_q[i]) ans[x.se]+=ask(x.fi);
} }

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

    cin >> n >> q;
    for (int i=0;i<n;i++){          // init/generate segments
        int a,b; cin >> a >> b;
        seg.pb({a,b});
    }

    for (int i=0;i<q;i++){          // init/generate queries
        int a,b; cin >> a >> b;
        query.pb({{a,b},i});    // i: id
    }

    work(seg, query);

    for (int i=0;i<m;i++) cout << ans[i] << "\n";
}
```

Sparse Table RMQ: Precomputation: O(nlogn), Query: O(1) Re: Binary Lifting (LCA)

Store the value of each interval of length 2^k for each k. When query, overlap left and right segments.

```
void build() {
    for (int j=0,b=31-__builtin_clz(MAXN);j<=b;j++)
    for (int i=0;i+(1<<j)-1<MAXN;i++)
        st[i][j]= (j?min(st[i][j-1],st[i+(1<<(j-1))][j-1]):a[i]);
}
```

```
int ask(int l, int r){            // return value of min a[l..r]
    int len = (int) log2(r-l+1);  // 31-__builtin_clz(r-l+1)
    return min(st[l][x], st[r-(1<<len)+1][len];
}
```

Disjoint Set: Use DFS on static DSU; "Opposite DSU" (Online bipartite checking): Store in[x], par[x]

Finding connected components in undirected graph (joining/adding components; Re: Flood fill combine)

```
int get (int x){
    if (x==p[x]) return x;
    else return p[x] = get(p[x]);
}

void init (int n){
    for (int i=0; i<=n; i++) p[i]=i, sz[i]=1;
}

int ask (int x, int y){
    return get(x) == get(y);
}

void merge (int x, int y){  // merge x onto y
    x=get(x), y=get(y);
    if (x==y) return;
    p[x]=y;
    sz[y] += sz[x];
}

int getCnt (int x){
    return sz[get(x)];
}
```

MO's algorithm (M1731 Mock Quizzes): Processing order; O((N + Q) * sqrt(N) * F); => Transition!

```
struct query{
    int n,k,id;
};

int Q,n,k,bs=sqrt(MAX_N); long long cur = 1; // value when n=0, k=0
long long ans[MAX_Q]; query a[MAX_Q];

for (int i=0;i<Q;i++) { // reading queries
    cin >> a[i].n >> a[i].k;
    a[i].id=i;
}

sort (a,a+Q,[](query x, query y){
    if (x.k/bs == y.k/bs){
        if (x.k/bs%2) return x.n < y.n;  // adj. up and down for 2x speed
        else return x.n > y.n;
    }
    return x.k/bs < y.k/bs;
});

for (int i=0;i<Q;i++) {
    while (n<a[i].n) {cur = 2 * cur - ncr(n,k) + MOD, n++, cur%=MOD;}
    while (n>a[i].n) {cur = (cur + ncr(n-1,k))* inv2 % MOD, n--, cur%=MOD;}

    while (k<a[i].k) {cur = cur + ncr(n,k+1), k++, cur%=MOD;}
    while (k>a[i].k) {cur = cur - ncr(n,k) + MOD, k--, cur%=MOD;}

    ans[a[i].id]=cur;
}

for (int i=0;i<Q;i++) cout << ans[i] << endl;
```

Sliding window minimum: Amortized O(1) insertion/deletion

```
void sliding_window_minimum(std::vector<int> & ARR, int K) {
    // pair<int, int> represents the pair (ARR[i], i)
    std::deque< std::pair<int, int> > window;
    for (int i = 0; i < ARR.size(); i++) {
        while (!window.empty() && window.back().first >= ARR[i]) // <= max
            window.pop_back();
        window.push_back(std::make_pair(ARR[i], i));

        while(window.front().second <= i - K)
            window.pop_front();

        std::cout << (window.front().first) << ' ';
} }
```

Grab minimum `l<i` s.t. `a[j]<a[i]` for all `l<j<i`

```
stack<int> s;
s.push(0);  l[0]=0;
for (int i=1;i<n;i++) {
    while (!s.empty() && A[s.top()]<A[i] s.pop();
    if(s.empty()) l[i]=0;
    else l[i]=s.top()+1;
    s.push(i);
}
```

2D BIT (Sparse, Offline, 0-based)

```
vector<int> points[MAX_X], trees[MAX_X]; int n, m;

void addPoint(int x, int y){
    for(++x; x < MAX_X; x+=x&-x)   // m is the first dimension, add x until M
        points[x].pb(y);
}

void update(int x, int y, int v){
    for(++x; x < MAX_X; x+=x&-x){
        // Go through the first dimension
        auto it = lower_bound(points[x].begin(), points[x].end(), y);
        // Second dimension: find the corresponding record for starting point
        for(int j = it - points[x].begin() + 1; j <= points[x].size(); j+=j&-j)
            trees[x][j] += v;
} }

int ask(int x, int y){
    int ret = 0;
    for(++x; x > 0; x-=x&-x) {
        auto it = upper_bound(points[x].begin(), points[x].end(), y);
        int j = it - points[x].begin();
        for(j; j > 0; j-=j&-j){
            ret += trees[x][j];
            if(j == 0) break;
    } }
    return ret;
}

void init() { // call this after adding all update points
    for (int x = 0; x < MAX_X; x++) {
        sort(points[x].begin(), points[x].end());
        trees[x].resize(points[x].size() + 5);
} }
```

Segment Tree: Build: O(n), Update: O(logn), Query: O(logn); Open interval [l, r)

```
#define MAXN 100005              // RMQ with range increase update
int segT[MAXN<<2], segd[MAXN<<2]; // lazy prop. For range upd, range ask

void build(int id=1, int l=0, int r=n){
    segd[id] = 0;
    if (l+1 == r){        // at leaf (unit-length)
        segT[id] = a[l]; // query value for unit-length node
        return;
    }
    build(id<<1, l, l+r>>1);
    build(id<<1|1, l+r>>1, r);

    segT[id] = max(segT[id<<1], segT[id<<1|1]);
}

void push(int id){ // apply 'below' upd range, pass upd to children
    segT[id<<1] += segd[id];      // paint: T[id<<1] = T[id<<1|1] = T[id];
    segT[id<<1|1] += segd[id];    // apply(id,val,l,r)
    segd[id<<1] += segd[id];      // better have push_need and push_val
    segd[id<<1|] += segd[id];
    segd[id]=0;   // NO_MARK = 0x3fffffff; (0 may be special!)
}

void upd(int x, int y, int v, int id=1, int l=0, int r=n){ // single: y=x+1
    if (y<=l || r<=x) return;
    if (x<=l && r<=y){  // cur node completely within upd range, so apply
        segT[id] += v; // actual query results
        segd[id] += v; // passing on upd to children
        return;
    }
    push(id); // upd only covers cur node partially, so push

    upd(x,y,v,id<<1,l,l+r>>1);
    upd(x,y,v,id<<1|1,l+r>>1,r);

    segT[id] = max(segT[id<<1], segT[id<<1|1]); // apply 'above' upd range
}

void modify(int x, int val) { // Single-node modification
    segT[L + x].val += val;  // L is the lowest power of 2 >= N
    for (int i = (L + x) >> 1; i >= 1; i >>= 1)
        segT[i] = max(nodes[i << 1], nodes[(i << 1) | 1]);
}

int getpos(int id=1, int l=0, int r=n){ // position of first max in whole T
    if (l+1 == r) return l;
    push(id);
    if (segT[id<<1] == 0) return getpos(id<<1, l, l+r>>1);
    else return getpos(id<<1|1, l+r>>1, r);
}

int ask(int x, int y, int id=1, int l=0, int r=MAXN) {  // call ask(l,r)
    if (x>=r || y<=l) return -1;        // sentiel value not affecting ans
    if (x<=l && y>=r) return T[id];     // query is as wide as node range

    push(id, l, r);   // children not yet updated, so push

    return max(ask(x, y, id<<1, l, l+r>>1), ask (x, y, id<<1|1, l+r>>1, r);
}
```

Range write bit + Range count

```
#define MAXN 100005 //  CF558E (substring counting sort)

using namespace std;

int T[30][MAXN<<2]; // [type of charcter] [count]
int D[30][MAXN<<2];
int a[MAXN],n,q; string s;
int cur[30];

void build (int id=1, int l=0, int r=n){
    if (l+1==r) {
        T[a[l]][id]=1;
        return;
    }
    build(id<<1,l,l+r>>1);
    build(id<<1|1, l+r>>1,r);

    for (int i=0;i<26;i++) T[i][id] = T[i][id<<1] + T[i][id<<1|1];
}

int cnt(int x, int y, int t, int id=1, int l=0, int r=n){

    if (y<=l || r<=x) return 0;
    if (x<=l && r<=y) return T[t][id];

    if (D[t][id]!=-1){
        T[t][id<<1] =   D[t][id]*((l+r>>1) -l);
        T[t][id<<1|1] = D[t][id]*(r- (l+r>>1));
        D[t][id<<1] = D[t][id<<1|1] = D[t][id];
        D[t][id]=-1;
    }

    return cnt(x,y,t,id<<1,l,l+r>>1) + cnt(x,y,t,id<<1|1,l+r>>1,r);
}

void upd(int x, int y, int t, int v, int id=1, int l=0, int r=n){
    if (y<=l || r<=x) return;
    if (x<=l && r<=y){
        T[t][id]=v*(r-l);
        D[t][id]=v; return;
    }
    if (D[t][id]!=-1){
        T[t][id<<1] =   D[t][id]*((l+r>>1) -l);
        T[t][id<<1|1] = D[t][id]*(r- (l+r>>1));
        D[t][id<<1] = D[t][id<<1|1] = D[t][id];
        D[t][id]=-1;
    }
    upd(x,y,t,v,id<<1,l,l+r>>1);
    upd(x,y,t,v,id<<1|1,l+r>>1,r);

    T[t][id]=T[t][id<<1] + T[t][id<<1|1];
}

// usage: cnt(x,y,i); upd(x,y,i,0);
```

Maximum contiguous subsequence sum

Online MCSS: Node store sum, prefix mcss, suffix mcss, mcss=max(mcss(l), mcss(r), suffix(l)+prefix(r))

Lazy propagation intuition

- T[node] tries to store the value for that range. But we don't want to update the whole tree, so store some updates in D[node] waiting to be pushed down. D[node] == -1 when nothing to push as to avoid pushing 0 being special.
- When to push down? Only when making modifications/query that will slice segment of cur. query.
- When to push up (rewrite current node after updating children)? When range queries have to be supported, we push up during updates.
- Assume some processed data is given/maintained => can I compute the answer within a reasonable amount of time (still viable if I add constant / log factor of overhead)
- Ensure lazy mark is good enough to both batch multiple updates and push the updates without loss

Parallel binary search

```
// assume T[] is a segtree storing range max
// search for first apperance of something >= v, if all < v, return n

int _ask(int v, int id = 1, int l = 0, int r = n) {
    // only works when ans is in current segment

    if(T[id] < v) return n;         // everything is < v

    if(l + 1 == r) {
        if(T[id] >= v) return l;
        else return n; // just to be safe
    }

    if (T[id<<1] >= v) { // if lhs stores something >= v, look into it, discard rhs
        return _ask(v, id<<1, l, l+r>>1);
    }
    else return _ask(v, id<<1|1, l+r>>1, r);    // vice versa
}

int ask(int start, int v){
    int id = getID(start); // returns segment containing a[start, start + 1]
    while(T[id].v < v){
        if(T[id].r == n){ // everything after start is peeked, found nothing,stop
            return n;
        }
        if(id%2 == 0) id /= 2;      // id is a left child, walk to parent
        else if(id%2 == 1) id++;    // else, walk to the sibling
    }

    // now, id is a segment which starts later than "start", contains sth gte v
    return _ask(v, id, T[id].l, T[id].r);
}
```

<table>
<tr><td colspan="16">1: [0, 16)</td></tr>
<tr><td colspan="8">2: [0, 8)</td><td colspan="8">3: [8, 16)</td></tr>
<tr><td colspan="4">4: [0, 4)</td><td colspan="4">5: [4, 8)</td><td colspan="4">6: [8, 12)</td><td colspan="4">7: [12, 16)</td></tr>
<tr><td colspan="2">8: [0, 2)</td><td colspan="2">9: [2, 4)</td><td colspan="2">10: [4, 6)</td><td colspan="2">11: [6, 8)</td><td colspan="2">12: [8, 10)</td><td colspan="2">13: [10, 12)</td><td colspan="2">14: [12, 14)</td><td colspan="2">15: [14, 16)</td></tr>
<tr><td>16: 0</td><td>17: 1</td><td>18: 2</td><td>19: 3</td><td>20: 4</td><td>21: 5</td><td>22: 6</td><td>23: 7</td><td>24: 8</td><td>25: 9</td><td>26: 10</td><td>27: 11</td><td>28: 12</td><td>29: 13</td><td>30: 14</td><td>31: 15</td></tr>
</table>

TODO: Split merge treap

Rectangle union area: Input n, each rectangle corners; Don't push updates down

```
#define MAXN 400005
int T[MAXN<<2][2],D[MAXN<<2]; // sum,cnt>0
int n,aa[MAXN],bb[MAXN]; long long ans;
vector<pair<pii,pii> > q,rect,event,ev[MAXN];   // event: {x,val}, {ylo, yhi}
vector<int> x, y; unordered_map<int,int> mx,my; // discretization

void upd(int x, int y, int v, int id=1, int l=0, int r=MAXN){ // inc range
    if (x>=r || y<=l) return;
    if (x<=l && y>=r) {
        T[id][0]+=v;
        T[id][1]=T[id][0]>0?bb[r]-bb[l]:T[id<<1][1] + T[id<<1|1][1];
        return;
    }
    upd(x,y,v,id<<1,  l,l+r>>1);
    upd(x,y,v,id<<1|1,l+r>>1,r);

    T[id][1]=T[id][0]>0?bb[r]-bb[l]:T[id<<1][1] + T[id<<1|1][1];
}

int ask(int x, int y, int id=1, int l=0, int r=MAXN){
    if (x>=r || y<=l) return 0;
    if (x<=l && y>=r) return T[id][1];

    return ask(x,y,id<<1,l,l+r>>1) + ask(x,y,id<<1|1,l+r>>1,r);
}

int main(){

    cin >> n;
    for (int i=0;i<n;i++) {
        int a,b,c,d; cin >> a >> b >> c >> d;
        x.pb(a), y.pb(b);
        x.pb(c), y.pb(d);
        rect.pb({{a,b},{c,d}});
    }

    sort(x.begin(),x.end()); sort(y.begin(),y.end());  // discretization
    auto lx = unique(x.begin(),x.end()); x.erase(lx,x.end());
    auto ly = unique(y.begin(),y.end()); y.erase(ly,y.end());

    for (int i=0;i<x.size();i++) mx[x[i]]=i, aa[i]=x[i];
    for (int i=0;i<y.size();i++) my[y[i]]=i, bb[i]=y[i];

    for (auto x:rect){   // event list making
        ev[mx[x.fi.fi]].pb({{mx[x.fi.fi], 1},{my[x.fi.se],my[x.se.se]}});
        ev[mx[x.se.fi]].pb({{mx[x.se.fi], -1},{my[x.fi.se],my[x.se.se]}});
    }

    int cur=0, oldlen = 0;  // answer calculation, shift line algorithm
    for (int i=0;i<x.size();i++){
        int newx = i;
        ans+=(long long)(aa[newx]-aa[cur]) * oldlen;
        for (auto x:ev[i])
            upd(x.se.fi, x.se.se, x.fi.se);
        cur = newx, oldlen = ask(0,MAXN);
    }

    cout << ans;
}
```

3. Graph: From vertices, edges to states

Special graphs:

- **Chain**: All vertices have 2 neighbors, except two of them only have 1 neighbor (Cycle: all degree 2)
  - No cycles, no branches; Process the nodes from head to tail
- **Star**: A tree with one internal node and k leaves; No cycles, maximum distance = 2;
- **Unicyclic**: Connected graph with exactly one cycle: |V|=|E|; Formed by adding 1 edge to tree
  - Work on cycle first, process subtrees connected to cycle one by one (T172 City Reform)
- Complete: Every pair of distinct vertices is connected by a unique edge; |E|=|V|*(|V|-1)/2
- Planar: Vertices and edges can be drawn in a plane such that no two edges intersect
  - More efficient algorithms can be applied: (Re: http://courses.csail.mit.edu/6.889/fall11/)

Graph representation: Adjacency list: O(E) For undirected graph, call new_edge twice

```
vector<pair<int, int> > e[100005];

void new_edge(int from, int to, int weight){
    e[from].pb({to, weight});
}

void query(int x){   // Querying all adjacent nodes of given node (x)
    for (int i=0;i<e[x].size();i++)
        printf("%d\n",e[x][i].first);
}
```

Grid Graph

```
const int dx4[] = { -1,0,1,0 };   const int dx8[] = { -1,0,1,0,1,1,-1,-1 };
const int dy4[] = { 0,1,0,-1 };   const int dy8[] = { 0,1,0,-1,1,-1,1,-1 };

bool visited[my][mx];

int valid(int x, int y){
    if (x < 0 || y < 0) return 0;
    if (x >= mx || y >= my) return 0;
    if (isWall(x, y)) return 0;
    return 1;
}

void query (int x, int y){
    // Querying all adjacent nodes of given node (x,y)
    for (int dir = 0; dir < 4; dir++){
        int nx = x + dx4[dir];
        int ny = y + dy4[dir];
        if (!valid(nx, ny)) continue;
        printf("%d %d\n", nx, ny);
} }
```

Topological Sort: O(V+E): Detect Cycles Lexi. smallest: priority_queue All orders: backtracking

Longest path: Initialize `dist[]` to 0. `dist[w]=max(dist[from] + dist[from][w])`, process by order

```
for (all vertices v)
    if (indegree[v]==0) push v into queue q      // priority_queue also ok
while (q is not empty){
    x = q.deque();                  // order.push_back(x)
    for (all nodes connected to x)
        if (--in[v]==0) q.push(v)
}
```

DFS: O(V+E) Example: Flood Fill (Static DSU), DFS Tree (Re: Connectivity)

No need to worry about stack overflow! Relax with using recursion! (remember base case)

```
dfs (v){
    visited[v] = 1;  // color[v] = gray
    st[v] = time++;
    for (all vertex w adjacent to v)
        if (visited[w] == 0)    dfs(w)
    ft[v] = time     // color[v] = black, rev_ts.pb(v);
}
```

BFS: O(V+E) Application: Shortest path, State searching (Water jug problem), BFS Tree

"BFS queue only contains elements from at max two successive levels of the BFS tree."

Multisource BFS: O(V+E): Push all the sources into the queue first

```
bfs (x){
visited[x]=1, distance[x]=0;
q.push(x);
while (q.size()!=0){
    t = q.front(); q.pop();
    for (all vertex w adjacent to t){   // int w:v[t]
        if (visited[w]==0){
            visited[w]=1;
            q.push(w);      distance[q] = distance[t] + 1
} } }
```

Bidirectional BFS: O(V+E): Run two searches simultaneously from initial and goal, stop when meet

```
bds {
    q.push(start), q.push(end);
    while (q.size()!=0){
        t = q.front(); q.pop();
        for (all vertex w adjacent to t) {
            if w is not visited
                calculate distance to w from original direction
                push w into the queue
            else if w was visited from other direction
                calculate distance to w
                return distance[starting][w] + distance[w][ending]
} } }
```

Iterating DFS: O(V+E)

```
ids (){
    limit=0;
    while goal is not found
        limit++;
        dfs(root,limit)
}
```

```
dfs (node,depth){
    if (depth<0) return;
    process the node
    for all neighbour w of node
        if (w is not visited)
            dfs(w, depth-1);   }
```

0/1 BFS (Re: http://codeforces.com/blog/entry/22276)

When edge weights are only 0 and x (x>=0), cannot apply otherwise (Re: BFS queue depth)

```
deque d, d.push_front(source)
while (d.empty()==false){
    t = d.front, d.pop_front();
    for (all edges e of form (vertex, u)){
        if (travelling e relaxes distance to u)
            relax dist[u]
            if (e.weight==1) d.push_back(u)
            else d.push_front(u)
} }
```

Shortest path tree (at vertex v): Rooted at v, path distance from v to other vertex u is shortest

- Well-defined iff there are no negative cycles (should not revisit any nodes, <= V-1 edges)
- Subpaths of shortest paths from u to v are shortest paths
- Lexicographically smallest shortest path: DAG u->v iff d[v]=d[u] + cost(u,v), then tsort/try 1..n
- Kth shortest path: For each vertex, store the least k distances, then dp

Dijkstra: Single source shortest path, No negative cycle; O(ElogE + V)

Minimax distance (directed): dis[u]=min(dis[u], max(dis[v], e[u][v])); Floyd-Warshall also works;

```
priority_queue <pii, vector<pii>, greater<pii> > q;      // {dis, node}
pq.push({0,st}), memset(dis,INF,sizeof(dis)), dis[st]=0; // push start
while (q.size())
    int v, w; tie(w,v) = q.top();
    q.pop()                                  // negative edge weight, use below:
    if (vis[v]) continue;  vis[v] = 1;   // if (w > dis[u]) continue;
    for (auto u:e[v])
        if (dis[u.fi] > dis[u.fi] + u.se)   // relax
            dis[u.fi] = dis[u.fi] + u.se,   // par[u.fi]=v
            pq.push({dis[u.fi], u.fi});
```

Bellman-Ford: Single source shortest path; O(VE); Re: Shortest path with decreasing edge weights

- Negative cycle detection: for each edge (u,v,w), if (dis[u] + w < dis[v]), then contain –ve cycle
- Given m inequalities of the form $x_i \leq x_j + c$ => Create source s to all x, edge j–>i with cost c

```
Iterate |V|-1 times
    for each edge e
        if (dis[e.to] > dis[e.from] + e.cost)
            dis[e.to] = dis[e.from] + e.cost    // par[e.to] = e.from
```

SPFA: Single source shortest path; Small label first (SLF), Largest label last (LLL); O(E) avg, O(VE) worst

```
push node 1 into the queue q
while (q is not empty)
    node u = q.front(); q.pop();
    for each edge e from u
        if (dis[e.to] > dis[u] + e.cost)
            dis[e.to] = dis[u] + e.cost
            if (node e.to is not in q)     // use Boolean array
                push node e into q
```

Floyd-Warshall: All pairs shortest path; Negative edges, no negative cycles; O(V^3) <=> V<=100

- Shortest cycle in directed graph: Find `min(dis[i][j]+dis[j][i])` / Init `dis[i][i]=INF`
- Negative cycle detection: Iff there exists `dis[i][i] < 0`

```
memset(dis,INF,sizeof(dis));  set all dis[i][i] to 0    // nxt[i][i] = i
for each edge (u,v)    dis[u][v] = w                    // nxt[u][v] = v

for m=1..n          // Remember middle goes first
    for u=1..n   for v=1..n
        if (dis[u][v] > dis[u][m] + dis[m][v])
            dis[u][v] = dis[u][m] + dis[m][v]           // nxt[u][v] = nxt[u][m]

vector<int> gen_path (int u, int v) {
    vector<int> ret = {u};
    while (u != v)
        u = nxt[u][v], ret.pb(u);
    return ret;
}
```

Path extraction (with par[])

```
for (auto t = end; t != -1; t = par[t])  path.pb(t);
reverse(path.begin(), path.end());
```

Minimum spanning tree: Undirected, Minimum total cost connecting all nodes (V-1 edges); Not unique!

- Uniqueness: Multiset of edge weights is unique; Min cost edge is unique => included in any MST
- Maximum spanning tree: Negate all edge lengths / reverse Kruskal => Maximin path (undirected)
- Minimum spanning subgraph (MST with fixed edges): Just continue running Kruskal
- Minimum spanning forest (K (fixed) subtrees): Run Kruskal, but break when have k components

Prim: Minimum spanning tree, Greedy, (Similar to Dijkstra); O(E + VlogV), for dense graph

```
pq.push((mp(0,1)) // {edge weight, node}, start from any node
while (pq.size())
    int dis, v; tie(dis,v) = pq.top(); pq.pop();  // cheapest frm prev
    if (vis[v]) continue; vis[v]=1;
    TODO: use this edge to connect to node v      // ans+=dis;
    for (pii u:e[v]) pq.push(u);
```

Kruskal: Minimum spanning tree, DSU; O(ElogV + Eα(V))

```
sort edges by costs (low to high)  // For choosing shortest available edge
initialize p[] // p[i]=i for all i
for each edge (u-v)
    if (find(u) != find(v)) // If u and v are not connected yet
        union(u,v)          // connect component of u with that of v
        TODO: use this edge to connect node u and node v  // ans+=cost
```

Connectivity (Re: Graph IV): Determination of edge (u,v)

Tree edge	Edge that form a tree	`par[v] == u`
Back edge	Go from a node to its ancestor => Cycles Cycle v. length = `st[v] − st[u] + 1`	`st[u] > st[v] && ft[u] < ft[v]`
Cross edge	Other edges, in the same tree or diff. tree	`st[u] > st[v] && ft[u] > ft[v]`
Cut edge (Bridge)	When removed, # connected component will increase (Re: articulation point)	Q: does the subtree have back edge to v ancestor? A: No => ok!

Cut edges: Undirected Graph: O(N+M);

Biconnected component: Mark all bridges, DFS from 1 to N but do not use bridges to travel.

In every bridge connected component, we can reach v-u in two ways without duplicate edges => shrink

```
dfs (v):
    set v as visited; st[v] = low[v] = t++;
    for (all child u of v):
        if (u is not visied)
            par[u] = v; dfs(u);
            low[v]=min(low[v],low[u])
            if (low[u]>=st[v] && (v != root || #child (v)>1 )) v cut vertex
            if (low[u]>st[v]) v-u bridge
        else if (u != par[v])
            low[v]=min(low[v],st[u])
```

SCC: If every vertex v and u, could reach to every vertex u (including v) Re: Transform to DAG

Weakly connected: Shrink each SCC into a point, then each chain is a WCC (Re: M1321 u-v or v-u)

2SAT:

```
dfs (v):        mark v as visited;
                for (all unvisited u:e[v]) dfs(u);
                V.pb(v);      // V store finishing time

rdfs (v,k):     mark v as visited;   v is belong to component k
                for (all unvisited u:r[v]) dfs(u);

find_scc():     for (1 to n) if (not visited i) dfs(i);
                for (n to 1) if (not visited V[i]) rdfs(V[i], k++);
```

Tree: A graph with n-1 edges, no cycles or self-loops, unique path between any two nodes

Ancestor detection in O(1): Flatten the tree to store subtree information, then use data structure

- v is ancestor of u iff `(st[v]<st[u] && ft[u]<ft[v])`
- x is an ancestor of y iff x occurs before y in preorder traversal and after y in post-order traversal

Tree traversal: Tree linearization on subtree (Pre-order, Post-order) . Or tree DP (root it first!)

- Binary search on tree (with queries): During DFS, store vector of root-to-current-vertex weight array
- Node heights: `d(v) = 1 + max(children(v));` Depth: `d(v)=d(par(v))+1`

Pre-order: st[], store node val Process node Traverse left, Traverse right	In-order: Only in binary tree Traverse left, Process node, Traverse right	Post-order: re children ans, ft[] Traverse left, traverse right Process node

Sparse LCA RMQ

```
int tab[LGN][MAXN<<1];
vector<int> e[MAXN], en[MAXN], ed[MAXN], euler, d[MAXN];

void dfs(int now, int prv){
    en[now] = euler.size();
    d[now] = d[prv]+1;
    euler.push_back(now);
    for(auto x: e[now]){
        if(x != prv){
            dfs(x, now);
            euler.push_back(now);
        }
    }
    ed[now] = euler.size();

    return;
}

void build(){
    int n = euler.size(); // n = 2*n-1
    for(int i = 0; i < n; i++){
        tab[0][i] = euler[i];
    }

    for(int i = 1; i < LGN; i++){
        for(int j = 0; j+(1<<i) <= n; j++){
            int x = tab[i-1][j];
            int y = tab[i-1][j+(1<<(i-1))];
            if(d[x] < d[y]) tab[i][j] = x;
            else tab[i][j] = y;
    } }

    return;
}

int ask(int x, int y){
    if (en[x] > en[y]) swap(x, y);
    x = en[x]; y = ed[y];
    int h = sizeof(int)*8 - __builtin_clz(y - x) - 1; // lg2(y - x)
    int u = tab[h][x];
    int v = tab[h][y - (1<<h)];

    if(d[u] < d[v]) return u;
    else return v;
}
```

Euler tour technique (Re: https://codeforces.com/blog/entry/18369)

Perform DFS. push_back node when arriving, [coming back from child], leaving.

- Subtree node sum: Subtree is a subarray, use BIT: `add(fi(v),x); sum(fi(v), la(v))`
- Sum on path: `add(fi(v),x), add(la(v), -x); sum_from_root(0, fi(v))`
- LCA (faster, linear memory): `v[argmin(fi(u), fi(v) in h]`

Difference array on trees: Given queries, each time increase all node from v to u by 1

```
sum[u]++, sum[par[v]]--;  // assume v is ancestor of u
dfs(v):                   // run dfs in root
    for all child u
        dfs(u); d[v]=d[v]+d[u];
```

Binary lifting LCA: Path splitting; Range path min on edges; Store the parent and answer for 2^i

```
int par[MAXN][LGN + 1], mx[MAXN][LGN + 1]; // LGN = 20 for 1e6
int h[MAXN]; memset(par, -1, sizeof(par)); dfs(1);

void dfs (int v,int p = -1, int w = INF){
    par[v][0] = p;
    /* mx[v][0] = w; */
    if (p != -1) h[v] = h[p] + 1;    // h is depth
    for (auto u : adj[v])
        if(u.fi != p) dfs(u.fi, v, u.se);
}

void build() {
    for (int i = 1; i <= LGN; i++) {
        for (int v = 1; v <= n; v++) {
            par[v][i] = par[par[v][i - 1]][i - 1];
            /* mx[v][i]  = min(mx[v][i - 1], mx[par[v][i - 1]][i - 1]); */
} } }

int walk (int u, int step, int &cur) { // walk steps upwards, O(lgn)
    for (int i = LGN; i >= 0; i--) {
        if (step & (1 << i)) {
            /* cur = min(cur, mx[u][i]); */
            u = par[u][i];
    } }
    return u;
}

int ask (int u, int v){     // lca
    /* int cur = INF; */
    if (h[u] < h[v]) swap(u, v); // now u is lower than v

    u = walk(u, h[u] - h[v], cur);

    if (v == u) return v; // return cur;

    for(int i = LGN; i >= 0; i--)
        if(par[v][i] != par[u][i]){
            v = par[v][i], u = par[u][i];
            /* cur = min(cur, mx[v][i], mx[u][i]); */
        }

    return par[v][0];
    /* return min({cur, mx[v][0], mx[u][0]}); */
}

// call dfs() for each node, then call build()
```

Centers and Centroid: Both at most 2 in one tree

- Center: The node that minimizes remoteness (distance from furthest node)
  - Finding: Topological sort from leaf; → All diameters in T must go through center
- Centroid: Node when removed, minimizes largest remaining component <= floor(n/2) in size;
  - Finding: Store subtree size of each node and n-s(v), then try all nodes

Tree diameter

- For each root, pair the two longest paths (depth of children) (Special: 1 child)
- BFS from any node, then BFS again from that furthest node => Acyclic (ok)/ general graphs (no)?
- Counting diameters: From the center, count # of nodes with depth n/2.

```
vector<int> e[MAXN];
int ans;
int dfs(int u,int p){
    int sum = 0;
    int max1 = 0, max2 = 0;
    for(auto v:e[u])
        if(v!=p) {
            int tmp = dfs(v,u);
            sum = max(sum, tmp);
            if(ans>max1)  max2 = max1,max1 = ans;
            else          max2 = max(ans,max2);
        }
    sum = max(sum, max1+max2);
    ans = max1 + 1;
    return sum;
}
```

Bipartite Matching (MCBM): Find the bi parts! MinEC = V - MCBM, MCBM = MinVC

Min edge cover = Max independent set: Max subset of vertices s.t. no two vertices represent an edge

Min vertex cover: Min set of vertices s.t. each edge in the graph is incident to at least one vertex

Unweighted Bipartite Matching (Hungarian Matrix-like): Augmenting path algorithm: O(VE)

Common bipartite graph: Grid, Gender, Row/Column, Odd/even, Tree

```
const int MAXN = 205;   // Number of vertices on each side.
int NX, NY;
int cx[MAXN], cy[MAXN]; // cx[i] (cy[i]) is companion of i-th vertex on l/r
bool G[MAXN][MAXN], visited[MAXN]; // G stores the graph. G[i][j] == 1 iff the i-th
vertex on the left and the j-th vertex on the right are connected.

int dfs(int u) {
    for (int v = 1; v <= NY; v++) {
        if (G[u][v] && !visited[v]) {
            visited[v] = 1;
            if (cy[v] == -1 || dfs(cy[v])) {
                cx[u] = v; cy[v] = u; return 1;
    } } }
    return 0;
}

int maxMatch() {
    int ans = 0;
    memset(cx, -1, sizeof(cx));
    memset(cy, -1, sizeof(cy));
    for (int i = 1; i <= NX; i++)
        if (cx[i] == -1)
            memset(visited, 0, sizeof(visited)), ans += dfs(i);
    return ans;}
```

Finding Eulerian tours (through every edge once) (DFS); Condition: 0 or 2 odd-degree nodes

```
vector E
dfs (v):
        color[v] = gray
        for u in adj[v]:
                erase the edge v-u and dfs(u)
        color[v] = black
        push v at the end of e
```

Powers of adjacency matrix A (unweighted)

- The $(i,j)^{th}$ entry of A^k counts the number of walks of length k from i to j
- Diameter D of the graph is the smallest $\mathbb{N}$ s.t. $A^D + A^{D-1} = (A+I)^D$ has no zero entries
- Index γ of the graph is the smallest $\mathbb{N}$ s.t. A^γ has no zero entries (non-bipartite iff $D \leq \gamma \leq 2D$)
- If there are two columns r_i, r_j in A^d which are orthogonal, then G is necessarily bipartite

Graph Colouring

- If the degree of vertex is small, then it should be easier to color it: (with X colors to choose from)
- For vertices with degree < X, we can easily find a color regardless how other vertices are colored
- Exhaust the color of vertices with degree >= X only
- Bipartiteness: Exhaust each color to be {1,2}, {3,4}, {5,6}, etc. Then bicolor each set with DFS/BFS.
- Planar graph: V-E+F=2, Can always be 4-colorable.
- Graph coloring is NP complete for k>=3. Use recursive backtracking instead (check prev. vertices)

Shortest cycle in undirected graph

- Alternative: Remove edge then Dijkstra for the two nodes
- Shortest cycle in directed graph: Set dis[i][i] to INF in Floyd Warshall

```
int n, m, u, v, w;
long long ans;
long long a[205][205], dis[205][205];

int main () {
    scanf("%d%d", &n, &m);
    ans = INF;
    memset(a, INF, sizeof(a));
    memset(dis, INF, sizeof(dis));
    for (int i = 0; i < m; i++) {
        scanf("%d%d%d", &u, &v, &w);
        a[u][v] = a[v][u] = dis[u][v] = dis[v][u] = w;
    }
    for (int k = 1; k <= n; k++){
        for (int i = 1; i <= k - 1; i++)
            for (int j = i + 1; j <= k - 1; j++)
                ans = min(ans, dis[i][j] + a[i][k] + a[k][j]);
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
    }
    printf("%lld\n", ans);
    return 0;
}
```

Graph Domination (NP)

Maximum independent set: $O((n/2)^2 2^{n/2})$: Meet in the middle, Bitmask DP

A set of vertices in a graph, no of which are adjacent; Also called: stable set / colclique / anticlique

```
#define MAXN 45 // CF1105E
bool adj[MAXN][MAXN]; // 0-based
int dp1[1 << (MAXN >> 1)], dp2[1 << (MAXN >> 1)]; // array instead of unorderd_map
int n;

int main(){
    buildGraph();

    int mi = n / 2;

    for (long long bs = 1; bs < (1LL << mi); bs += 1LL) {
        // for first n/2 nodes, compute max for each bitset
        vector<int> cur;
        for (int node = 0; node < mi; node++) {
            if ((1LL << node) & bs) {
                cur.pb(node);
                dp1[bs] = max(dp1[bs], dp1[bs ^ (1LL << node)]);
        } }
        bool invalid = 0;
        for (int x : cur) for (int y : cur) if (adj[x][y]) invalid = 1;
        if (not invalid) dp1[bs] = cur.size();
    }

    for (long long bs = (1LL << mi); bs < (1LL << n); bs += (1LL << mi)) {
        // do the same for next n/2 nodes
        vector<int> cur;
        for (int node = mi; node < n; node++) {
            if ((1LL << node) & bs) {
                cur.pb(node);
                dp2[bs >> mi] = max(dp2[bs >> mi], dp2[(bs ^ (1LL << node))>> mi]);
        } }
        bool invalid = 0;
        for (int x : cur) for (int y : cur) if (adj[x][y]) invalid = 1;
        if (not invalid) dp2[bs >> mi] = cur.size();
    }

    int ans = 0;

    for (long long bs = 0; bs < (1LL << mi); bs += 1LL) {
        // meet in the middle, get the appropiate bitset from other side
        long long getHalf = ((1LL << (n - mi)) - 1) << mi;
        for (int node = 0; node < mi; node++) {
            if ((1LL << node) & bs) {
                for (int i = mi; i < n; i++) {
                    if (adj[node][i]) {
                        getHalf &= (~(1LL << i));
        } } } }
        ans = max(ans, dp1[bs] + dp2[getHalf >> mi]);
    }

    cout << ans;
}
```

TODO: Euler tour, random walk, Cycle

Minimum dominating set:

A set of vertices in a graph, s.t. every vertex not in the set is adjacent to at least 1 member of the set

Minimum vertex cover:

A set of vertices s.t. each edge of the graph is incident to at least one vertex of the set

Minimum edge cover:

A set of edges s.t. each vertex of the graph is incident to at least one edge of the set

Maximum matching:

A set of edges without common vertices

(All) maximum cliques:

Subset of vertices, all pairs adjacent to each other, complete subgraph

Chinese Postman Problem:

Stable Marriage:

2D BIT of boolean: O(Qlog2(N))

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#define mp make_pair
using namespace std;
using namespace __gnu_pbds;
typedef pair<int, int> pii;
typedef tree<pii, null_type, less<pii>, rb_tree_tag,
tree_order_statistics_node_update> OST;

const int N = 100001;

OST bit[N];

void insert(int x, int y) {
    for(int i = x; i < N; i += i & -i)
        bit[i].insert(mp(y, x));
}

void remove(int x, int y) {
    for(int i = x; i < N; i += i & -i)
        bit[i].erase(mp(y, x));
}

int query(int x, int y) {
    int ans = 0;
    for(int i = x; i > 0; i -= i & -i)
        ans += bit[i].order_of_key(mp(y+1, 0));
    return ans;
}
```

4. Max Flow = min cut; Multi-source/Multi-sink => Add one big source/sink with INF cap, 0 cost
Capacities relate to feasibility, Edge cost relate to optimality/preference

Dinic Max Flow: $O(V^2E)$; $O(\min(V^{2/3}, \sqrt{E}) * E)$ unit capacity, $O(\sqrt{V}E)$ bipartite/unit network

```
#define MAXN 505
int cap[MAXN][MAXN],N,S,T,lay[MAXN],f,flow,vis[MAXN];
queue<int> q;

int bfs() {
    memset(lay,0,sizeof(lay));
    while (!q.empty()) q.pop();
    q.push(S);
    lay[S]=1;
    while (!q.empty()) {
        int cur=q.front(); q.pop();
        for (int v=0;v<N;v++) {
            if (cap[cur][v]>0 && lay[v]==0) {
                lay[v]=lay[cur]+1;
                q.push(v);
                if (v==T) return lay[T];
    } } }
    return 0;
}

int dfs(int cur,int lc) {
    if (cur==T) return lc;
    vis[cur]=1;
    for (int v=0;v<N;v++) {
        if (cap[cur][v]>0 && lay[v]==lay[cur]+1 && !vis[v]) {
            int tmp=dfs(v,min(lc,cap[cur][v]));
            if (tmp>0) {
                cap[cur][v]-=tmp;
                cap[v][cur]+=tmp;
                return tmp;
    } } }
    return 0;
}

void addEdge(int u, int v, int val, int type){
    cap[u][v]=val;
    cap[v][u]=val*type;
}

int main() {

    // Assign N (number of vertices), S (source) and T (sink) here.
    // Vertices numbered from 0 to N - 1. Hence S and T should be in [0, N).

    // addEdge (u, v, capacity, [DIRECTED | UNDIRECTED]);

    while (bfs()) {
        while (1) {
            memset(vis,0,sizeof(vis));
            f=dfs(S,1e9);
            if (f==0) break;
            flow+=f;
    } }
    cout << flow << endl;
}
```

Dinic Max Flow: (haleyk)

```
#include <cstdio>
#include <queue>
using namespace std;

typedef int flow_t;
const int MAXN = 505, MAXM = 100005, DIRECTED = 0, UNDIRECTED = 1;
const flow_t INFTY = 0x3fffffff;
int N, S, T, now;
struct edge {
    flow_t remain;
    int endVertexId, nextEdgeId;
}e[MAXM << 1];
struct vertex {
    int firstEdgeId, level, firstUnsaturEdgeId;
}v[MAXN];

void _addEdge(int begin, int end, flow_t c) {
    e[now].remain = c;
    e[now].endVertexId = end;
    e[now].nextEdgeId = v[begin].firstEdgeId;
    v[begin].firstEdgeId = now++;
}

void addEdge(int begin, int end, flow_t c, int edgeType) {
    _addEdge(begin, end, c);
    _addEdge(end, begin, edgeType * c);
}

void init() {
    now = 0;
    for (int i = 0; i < N; ++i) v[i].firstEdgeId = -1;
}

bool markLevel(){
    for (int i = 0; i < N; ++i)
        v[i].level = -1, v[i].firstUnsaturEdgeId = v[i].firstEdgeId;
    v[S].level = 0;
    queue<int> Q;
    Q.push(S);
    while (!Q.empty()) {
        int x = Q.front(); Q.pop();
        for (int i = v[x].firstEdgeId; i >= 0; i = e[i].nextEdgeId)
            if (e[i].remain && v[e[i].endVertexId].level < 0)
                    v[e[i].endVertexId].level = v[x].level + 1,
                    Q.push(e[i].endVertexId);
    }
    return v[T].level > 0;
}

flow_t extendFlow(int x, flow_t flow) {
    if (x == T) return flow;
    flow_t t, total = 0;
    for (int &i = v[x].firstUnsaturEdgeId; i >= 0; i = e[i].nextEdgeId) {
        if (v[e[i].endVertexId].level == v[x].level + 1 && e[i].remain) {
            if (t = extendFlow(e[i].endVertexId, min(flow, e[i].remain)))
            e[i].remain -= t, e[i ^ 1].remain += t, flow -= t, total += t;
            if (0 == flow) break;
    } }
    return total;
```

```
}

flow_t Dinic() {
    flow_t flow, total = 0;
    while (markLevel())
        while (flow = extendFlow(S, INFTY))
            total += flow;
    return total;
}

void buildGraph() {
    // Assign N (number of vertices), S (source) and T (sink) here.
    // Vertices numbered from 0 to N - 1. So S and T should be in [0, N).
    init();
    // Add edges here
    addEdge (u, v, capacity, [DIRECTED | UNDIRECTED]);
}

int main() {
    int nCase, n, m;
    scanf("%d", &nCase);
    while (nCase--) {
        scanf("%d%d", &n, &m);
        buildGraph();
        flow_t ans = Dinic();
    }
    return 0;
}
```

Dinic Min Cost Max Flow: O(V^2E); Add source of capacity [req. flow], check total >= req. flow

Min Cost Fixed Flow: Add an edge of capacity [flow] from S to each source

```
typedef int flow_t, cost_t;
const int MAXN = 405, MAXM = 1505, DIRECTED = 0, UNDIRECTED = 1;
const flow_t FLOW_INFTY = 0x3fffffff;
const cost_t COST_INFTY = 0x3fffffff;
int N, S, T, now, K;
bool inQ[MAXN];
struct edge {
    flow_t remain;
    cost_t cost;
    int endVertexId, nextEdgeId;
}e[MAXM << 1];
struct vertex {
    int firstEdgeId, firstUnsaturEdgeId;
    cost_t level;
}v[MAXN];

void _addEdge(int begin, int end, flow_t c, cost_t w) {
    e[now].remain = c;
    e[now].cost = w;
    e[now].endVertexId = end;
    e[now].nextEdgeId = v[begin].firstEdgeId;
    v[begin].firstEdgeId = now++;
}

void addEdge(int begin, int end, flow_t c, int edgeType, cost_t w = 1) {
    _addEdge(begin, end, c, w);
    _addEdge(end, begin, edgeType * c, -w);
}
```

```
void init() {
    now = 0;
    for (int i = 0; i < N; ++i) v[i].firstEdgeId = -1, inQ[i] = false;
}

bool markLevel(){ // SPFA
    for (int i = 0; i < N; ++i)
        v[i].level = COST_INFTY, v[i].firstUnsaturEdgeId =
                            v[i].firstEdgeId, inQ[i] = false;
    v[S].level = 0;
    queue<int> Q;
    Q.push(S);
    inQ[S] = true;
    while (!Q.empty()) {
        int x = Q.front();
        Q.pop();
        inQ[x] = false;
        for (int i = v[x].firstEdgeId; i >= 0; i = e[i].nextEdgeId) {
            if (e[i].remain && v[e[i].endVertexId].level > v[x].level + e[i].cost)
{
                v[e[i].endVertexId].level = v[x].level + e[i].cost;
                if (!inQ[e[i].endVertexId])
                Q.push(e[i].endVertexId), inQ[e[i].endVertexId] = true;
    } } }
    return v[T].level < COST_INFTY;
}

flow_t extendFlow(int x, flow_t flow) {
    if (x == T) return flow;
    inQ[x] = true;
    flow_t t, total = 0;
    for (int &i = v[x].firstUnsaturEdgeId; i >= 0; i = e[i].nextEdgeId) {
        if (v[e[i].endVertexId].level == v[x].level + e[i].cost && e[i].remain
&& !inQ[e[i].endVertexId]) {
            if (t = extendFlow(e[i].endVertexId, min(flow, e[i].remain)))
            e[i].remain -= t, e[i ^ 1].remain += t, flow -= t, total += t;
            if (0 == flow) break;
    } }
    inQ[x] = false;
    return total;
}

flow_t Dinic() {
    flow_t flow, total = 0;
    cost_t cost = 0;
    while (markLevel())
        while (flow = extendFlow(S, FLOW_INFTY)) // min-cost flow: && cost <= 0
            total += flow, cost += flow * v[T].level;
    return cost; // Return total in maxFlow; return cost in minCostMaxFlow
}

void buildGraph() {
    // Assign N (number of vertices), S (source) and T (sink) here.
    // Vertices numbered from 0 to N - 1. So S and T should be in [0, N).
    init();
    // Add edges here
}
```

5. Number theory

Modular Arithmetic: d|n => n is divisible by d

- $(x + y)\ mod\ m = (x\ mod\ m + y\ mod\ m)\ mod\ m, (x * y)\ mod\ m = (x\ mod\ m * y\ mod\ m)\ mod\ m$
- $(x - y)\ mod\ m = (x\ mod\ m - y\ mod\ m + m)\ mod\ m$
- $(x/y)\ mod\ m = (x * y^{-1})\ mod\ m$ (modular inverse)
- Negative numbers (a<0): `a%b = ((a%b) + b) % b`

Modpow; Modular inverse: $A^{-1} = A^{\varphi(m)-1}$ $A^{-1} = A^{p-2}$ where p is prime

```
long long mpow(long long a, long long p, long long m){
    if (p == 0) return 1LL % m;
    if (p % 2) return a * mpow(a, p - 1, m) % m;
    long long t = mpow(a, p / 2, m);
    return t * t % m;
} }   // 3 18132 17 => 13
```

```
cin >> s;
reverse(s.begin(),s.end());
for (int i=0;i<s.length();i++)
    k+=(s[i]-'0')*mp(10,i,m);
    k%=m;
```

Euclidean Algorithm: O(logn). Worse case 2 cont. Fib. numbers; `lcm (a,b) = a / gcd(a, b) * b;`

- For any positive integers a b, let r=a%b. Then `gcd (a,b) = gcd (b, r)`, iterate until r==0
- For coprime (a, b), highest value that cannot be represented as ax + by (x, y >= 0) is a*b-(a+b)

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

Extended Euclidean Algorithm: O(logn); Find integral solutions of Ax + By = c * gcd(A, B);

- Solution shift: `x' = c * x + k * B/gcd(A,B);` `y' = c * y - k * A/gcd(A,B)`

```
pll egcd (long long a, long long b) { // fi * a + se * b = gcd
    if (a > b) {
        pll ret = egcd(b, a);
        return mp(ret.se, ret.fi);
    }
    if (a == 0) return mp(0, 1);
    pll p = egcd(b % a, a);
    return mp(p.se - (b / a) * p.fi, p.fi);
}

pll crt(vector<pll> v) { // all modulo are coprime
    // {gcd(m,n)=1, x % m = a, x % n = b} => x % mn = adn + bcm , s.t. cm + dn = 1
    // d = inv(n)(mod m), c = inv(m)(mod n) => if m, n primes, n=n^(m-2), m=m^(n-2)

    vector<long long> a, m, p; // ai = mi % pi
    long long ans = 0, s = 1, x, y;
    for (auto eqt : v) m.pb(eqt.fi), p.pb(eqt.se),
                        a.pb(m.back() % p.back()), s *= p.back();

    for (int i = 0; i < v.size(); i++) {
        tie(x, y) = egcd(p[i], s / p[i]);
        ans += y * s / p[i] * a[i];
    }
    ans %= s;   if (ans < 0) ans += s;
    return mp(ans, s);
}

vector<pll> eqt = {mp(1, 3), mp(4, 5), mp(6, 7)}; // x = mi (mod pi); mp(mi, pi)
auto crt_ans = crt(eqt) // ans: {34 105}, x % se = fi

void inverse(int a, int m) { // gcd(A, M) = 1 => Ax + My = 1, inv(A) = x
    auto op = egcd(a, m); return op.fi;
}

// linear congurence: ax = b (mod n) => ax = ny + b => ax - ny = c * gcd(a, n) = b
```

Euler's phi/totient function φ(n): Number of coprime integers to n in [1..n]

```
ans = n;  // input n
for each distinct prime factor pi of N, ans *= (pi-1); ans /= pi;
return ans
```

Factorization and sieve

```
#define MX 15000000
int u[MX+5], p[MX+5], pn; // u: min prime factor; p: primes; pn: #primes

void seive(){
    for (i=2; i<=MX; i++) {
        if(!u[i]) u[i] = p[++pn] = i;
        for (j=1; i*p[j]<=MX; j++){
            u[i*p[j]] = p[j];
            if(i%p[j] == 0) break;
} } }

vector<int> factorize(int x){
    vector<int> a;
    for(int j=x;j>1;)
        a.pb(u[j]), j/=u[j];
    return a;
}
```

Maximal number of divisors of any n-digit number (Re: OEIS A066150)

```
4, 12, 32, 64, 128, 240, 448, 768, 1344, 2304,                    //10^9
4032, 6720, 10752, 17280, 26880, 41472, 64512, 103680, 161280    //10^18
```

Number of prime numbers under 10^n (Re: OEIS A006880)

```
4, 25, 168, 1229, 9592, 78498, 664579, 5761455, 50847534,            //10^9
455052511, 4118054813, 37607912018, 346065536839, 3204941750802, 29844570422669,
279238341033925, 2623557157654233, 24739954287740860 //10^18
```

Theorems/Conjectures:

- Prime number conjectures (any even number >2 is sum of two primes), Pythagorean triple
- Every +ve integer is a unique sum of one or more distinct non-consecutive Fib. numbers (greedy)

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \frac{n^2+n}{2} \text{ (the triangular numbers)}$$

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3+3n^2+n}{6} \text{ (the square pyramidal numbers)}$$

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \left[\frac{n(n+1)}{2}\right]^2 = \frac{n^4+2n^3+n^2}{4} \text{ (the squared triangular numbers)}$$

$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{6n^5+15n^4+10n^3-n}{30}$$

Simplify fraction (Re: Radial sweep, Slope)

$\gcd(a/b, c/d) = \gcd(a,c)/\mathrm{lcm}(b,d)$ $\mathrm{lcm}(a/b, c/d) = \mathrm{lcm}(a,c)/\gcd(b,d)$

```
pii simplify(pii x){
    if (x.se==0) return {1,0};
    if (x.fi==0) return {0,1};
    int g = __gcd(abs(x.fi),abs(x.se));
    if (x.se<0) g*=-1;
    return {x.fi/g, x.se/g};
}
```

Combinatorics

Try to think using DP method first. If the constraints are too large, then try nCr O(1) method

nCr mod p (prime): O(n) preprocess, O(1) query: n<=10^6

Small p, large n: Lucas' Theorem ($O(p^2 \log_p n)$): $C_r^n = \prod_{i=0}^{k} C_{r_i}^{n_i} \ (mod\ p), n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_0$

```
fact[0] = fact[1] = minv[0] = minv[1] = finv[0] = finv[0] = 1;
for (long long i = 2; i < MAXN; i++){
    fact[i] = i * fact[i-1] % MOD;
    minv[i] = minv[MOD % i] * (MOD – MOD / i) % MOD;
    finv[i] = minv[i] * finv[i - 1] % MOD;
}
for each query (r,n), output fact[n] * finv[r] % MOD * finv[n - r] % MOD
```

Exact nCr: O(r) query

```
long long ncr (long long n, long long r) {
    if (r > n) return 0;
    if (r > n / 2) return ncr(n, n - r);
    long long ret = 1;  for (int i = 1; i <= r; i++) ret = ret * (n – i + 1) / i;
    return ret;
}
```

Pascal's triangle: $C_r^n = C_r^{n-1} + C_{r-1}^{n-1}$;

```
for (int i = 1; i <= n; i++){   // pas[n][r] stores C(n, r)
    pas[i][0] = pas[i][i] = 1 % m;
    for (int j = 1; j < i; j++){
        pas[i][j] = (pas[i - 1][j] + pas[i - 1][j - 1]) % m
```

Catalan number: $A_n = \sum_{i=1}^{n} A_{i-1} A_{n-i} \quad A_n = C_n^{2n} - C_{n-1}^{2n} = \frac{C_n^{2n}}{n+1} = \frac{2m(2m-1)}{(m+1)m} * A_{m-1},\ A_0 = 1$

- # valid string containing n pairs of parenthesis, #ways n+1 factors completely parenthesized
- Number of binary trees with n nodes => Number of ways n edges arranged in binary tree.
- Number of ways a convex polygon of n+2 sides can be triangulated
- Number of monotonic paths along the edges of a n*n grid that do not pass above the diagonal
- Reflection trick: If the positions drops to -1, we can flip the following steps and end up at -2

Pigeon Hole Principle:

- Suppose there are M balls and N boxes, then we can find a box containing >= ceil(m/n) balls.
- Let s[1..N] be an array of N integers. Then there exists a subarray s[L..R] with sum divisible by N.
- If there are (2N+1) integers in the range [1, 2^N), then can pick 3 int satisfying triangle inequality

TODO:

- Burnside lemma, Polya enumeration theorem
- Stirling numbers, Bell numbers, integer partitioning
- Carmichael function
- Pollard's rho function (integer factorization in O(sqrt(p))
- Mobius
- ModGraph
- Finding the next number in a sequence: Method of common differences
- Postfix calculator (Infix to postfix)
- STAT2601: nHr, Discrete distributions, Generating function, Inclusion-Exlcusion
- Alpha beta pruning, Minimax tree

Game theory (ST Function, xor sum, Nim): If all else fails, draw the NP/AB table / Decision tree!

- At a particular position, both players have same set of moves, Finite moves, Perfect information

Table drawing:

Ending position is a P position.
If initial position is N position, P1 wins; If initial position is P position, P2 wins;

Suppose we can reach positions $X_1, X_2, X_3, \ldots X_K$ from current position X_{cur}.
If all X_i are N positions, X_{cur} is a P position. Otherwise, X_{cur} is an N position

SG Function:

- MEX of a set: Smallest non-negative integers *not* in the set. (MEX(empty set) = 0)
- Graph: Topological sort (with reverse graph)!
- Game split into two: $SG(G_1) = SG(G_{11})$ xor $SG(G_{12})$

Suppose we can reach positions $X_1, X_2, X_3, \ldots X_K$ from current position X_{cur}.
$SG(X_{cur}) = mex(\{SG(X_1), SG(X_2), \ldots, SG(X_K)\})$

A position is P position iff SG value ==0. A position is N position iff SG value > 0.
Ending position (P) has SG value == 0.

Suppose we have analyzed each independent game using SG function. Let P_i be the position in G_i

Then $SG(P_1, P_2, \ldots P_N) = SG(P_1)$ xor $SG(P_2)$ xor … xor $SG(P_N)$
For multiple games, it is a P position iff xor sum of SG value of all states = 0

Algorithmic games:

```
bool isWinning(position pos) {
    moves[] = possible positions to which I can move from position pos;
    for (all x in moves)
        if (!isWinning(x)) return true;
    return false;
}

int grundyNumber(position pos) {
    moves[] = possible positions to which I can move from pos
    set s;
    for (all x in moves) s.insert (grundyNumber(x));
    // return smallest non-negative integer not in the set s;
    int ret=0;
    while (s.contains(ret)) ret++;
    return ret;
}
```

Nim 2: N stone piles, remove some number (>0) of stones from the pile

From $G_{X,Y}$ we can reach ($G_{0,Y}$, … $G_{X-1,Y}$, $G_{X,0}$ … $G_{X,Y-1}$) $SG(G_{X,Y}) = SG(G_X)$ xor $SG(G_Y)$; $SG(G_K)=K$
General solution: <u>If $A_1 \wedge A_2 \wedge \ldots \wedge A_N = 0$ (xor sum), P2 wins. Otherwise, P1 wins.</u>

Winning strategy:
If xor sum==0, After P1's move, xor sum!=0. There exists a move for P2 s.t. after the move xor sum=0
If xor sum!=0, P1 should move to make xor sum =0. Then use the strategy above.

The move: Find the most significant 1 in xor sum => There exists a number with '1' in that place => reduce (minus) that number (A_i) so that xor-sum is 0.

Player 2 wants xor sum ==0. So add a new pile where A_{n+1} = xor sum.
Nim 1: Misère game: Change the strategy at the end of the game.
=> Choose a move so that there is an odd number of heaps with one stick.

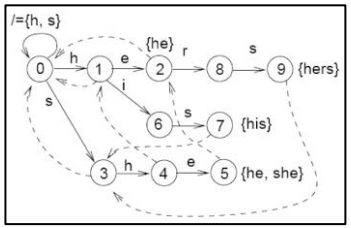
6. String: Pattern is what we want to find, Text is the large string

KMP: p[i] = Length of longest suffix of S[0…i] which matches prefix of S[0…i], apart from whole string

```
const int LEN_PATTERN = 10005, LEN_TEXT = 1000005;
int next[LEN_PATTERN], matchTo[LEN_TEXT];
char s[LEN_PATTERN], t[LEN_TEXT]; // s: pattern string; t: text
// Note that both strings should be indexed from 1

void KMP() {
    int lenS = strlen(s + 1), lenT = strlen(t + 1), p = 0;
    for (int i = 2; i <= lenS; ++i) {
        while (p && s[i] != s[p + 1]) p = next[p];
        next[i] = (s[i] == s[p + 1]) ? ++p : 0;
    }
    p = 0;
    for (int i = 1; i <= lenT; ++i) {
        while (p && t[i] != s[p + 1]) p = next[p];
        matchTo[i] = (t[i] == s[p + 1]) ? ++p : 0;
    }
}
```

AC Automaton / TrieGraph

```
const int TOTLEN = 1000006, MAXN = 100005, SIGMA = 52;

string str[MAXN];
int trie[TOTLEN][SIGMA], fail[TOTLEN], nxt[TOTLEN][SIGMA];
int nxt_node = 1, n;
int has_str[TOTLEN]; // bitset. Don't use __int128

// fail[s]: ending node of the proper prefix of some pattern which is the longest
proper suffix of s
// has_str[s]: indexes of all words ending at s

int toInt(char c) {
    if ('a' <= c && c <= 'z') return c - 'a';
    else return c - 'A' + 26;
}

void insert(int id) { // insert string[id] into the trie
    int cur = 0;
    for (char c : str[id]) {
        if (trie[cur][toInt(c)] == -1) // no next node
            trie[cur][toInt(c)] = nxt_node++;
        cur = trie[cur][toInt(c)];
    }
    has_str[cur] |= (1 << id); // str[id] can terminate at this node
}

void buildAC() {
    memset(trie, -1, sizeof(trie));
    for (int id = 0; id < n; id++)
        insert(id);

    memset(fail, -1, sizeof(fail));
    memset(nxt, -1, sizeof(nxt));

    queue<int> q;

    for (int ch = 0; ch < SIGMA; ch++) // build initial edges for depth-1 nodes
        if (trie[0][ch] >= 1) {
            fail[trie[0][ch]] = 0;
            q.push(trie[0][ch]);
        }
        else trie[0][ch] = 0; // if no edge, loop back to self. modifies trie edge!

    while (q.size()) {
        int now = q.front(); q.pop();
        for (int ch = 0; ch < SIGMA; ch++) {
            if (trie[now][ch] != -1) { // has trie edge

                int f = fail[now];
                while (trie[f][ch] == -1) // go back until node has ch as child
                    f = fail[f];

                f = trie[f][ch]; // forwards 1 char, suffix fail points to this

                fail[trie[now][ch]] = f;
                has_str[trie[now][ch]] |= has_str[f];

                q.push(trie[now][ch]);
} } } }

int nextState(int cur, char c) { // at this node, where to go next with edge c?
    if (nxt[cur][toInt(c)] != -1) return nxt[cur][toInt(c)];
    if (trie[cur][toInt(c)] != -1)
        return nxt[cur][toInt(c)] = trie[cur][toInt(c)];
    else return nxt[cur][toInt(c)] = nextState(fail[cur], c);
}

void search(string text) {
    int cur = 0;
    for (int i = 0; i < text.length(); i++) {
        char c = text[i];

        cur = nextState(cur, c);

        if (not has_str[cur]) // no string ending here
            continue;

        for (int j = 0; j < n; j++) {
            if (!!(has_str[cur] & (1 << j))) {
                cout<<str[j]<<": [" << i - str[j].length() + 1<<", " << i << "]\n";
} } } }

int main() {
    // init n, str[] here.
    n = 4; string input[] = {"he", "she", "his", "hers"};
    for (int i = 0; i < n; i++) str[i] = input[i];

    buildAC();

    for (int i = 0; i < nxt_node; i++) {
        cout << "node " << i << ": " << fail[i] << ' ';
        for (int j = 0; j < n; j++)
            cout << !!(has_str[i] & (1 << j));
        cout << '\n';
    }
    // 0 0 0 1 2 0 3 0 3
    // 2: 1000, 5: 1100, 7: 0010, 9: 0001
    search("ahishers"); // his: [1, 3], he: [4, 5], she: [3, 5], hers: [4, 7]
}
```



Trie: Rooted tree, each edge has a character

```
short trie[MAXN][26], nxt_node = 1;
bool ending[MAXN]; // MAXN = SIGMA * TOT_LENGTH, 1 new node for each character

void insert(string s) {
    int cur = 0;
    for (char c : s) {
        if (!trie[cur][c - 'a'])
            trie[cur][c - 'a'] = nxt_node++;
        cur = trie[cur][c - 'a'];
    }
    ending[cur] = 1;
}

bool inside_trie(string s) {
    int cur = 0;
    for (int i = 0; i < s.length(); i++) {
        if (trie[cur][s[i] - 'a']) cur = trie[cur][s[i] - 'a'];
        else return 0; // match first i characters
    }
    if (ending[cur]) cout << "inside the trie\n";
    else cout << "prefix of something in the trie\n";
    return 1;
}
```

Manacher: len[i] = (length of longest palindrome centered here – 1) / 2

```
vi manacher(string input){
    string str;
    for(auto x: input) str += x + "$";
    str.pop_back(); // a$b$c$d

    int n = str.length();
    vi len(n, 1);

    for(int i = 1, rightmost = 0, pivot = 0; i < n; i++){
        if(rightmost > i){
            int j = pivot-(i-pivot);
            len[i] = min(len[j], rightmost-i);
        }

        for (; i>=len[i] && i+len[i]<n && str[i-len[i]]==str[i+len[i]]; len[i]++);

        if (i+len[i] > rightmost){
            rightmost = i+len[i];
            pivot = i;
        } }

    return len;
}
```

Z Algorithm: Z[i] = Length of longest substring starting from S[i] which is also a prefix of S

```
vi init(string str){
    int n = str.length();
    vi z(n, 0);
    // z[0] = 0;  special case
    for(int i = 1, rightmost = 0, pivot = 0; i < n; i++){
        if(rightmost > i){
            int j = i-pivot;
            z[i] = min(z[j], rightmost-i);
        }

        for (; i+z[i] < n && str[z[i]] == str[i+z[i]]; z[i]++);

        if (i+z[i] > rightmost){
            pivot = i;
            rightmost = i+z[i];
        }
    }

    return z;
}

vi match(string pattern, string text){
    string str = pattern+"$"+text;
    vi z = init(str);
    vi match_position;
    for(int i = pattern.length()+1; i < str.length(); i++){
        if(z[i] == pattern.length()){
            match_position.push_back(i);
        }
    }

    return match_position;
}
```

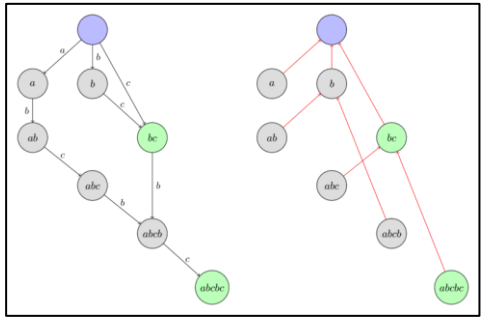
Hashing: Equivalence in O(1); Multiple string search: Use unordered_set to store hashed patterns;

```
#define MOD 1000000007LL // CF559B
#define inv256 285156252

char s[200005], t[200005];
long long hs[200005],ht[200005],mpow[200005],minv[200005];

int cmp(int l1, int r1, int l2, int r2){ // closed interval
    int h1 = (hs[r1]-hs[l1-1]+MOD)%MOD * minv[l1-1] % MOD;
    int h2 = (ht[r2]-ht[l2-1]+MOD)%MOD * minv[l2-1] % MOD;

    return h1==h2;
}

int main(){
    mpow[0]=minv[0]=1;
    for (int i=1;i<200005;i++) mpow[i] = mpow[i - 1] * 256 % MOD;
    for (int i=1;i<200005;i++) minv[i] = minv[i - 1] * inv256 % MOD;

    scanf("%s%s",s,t); n = strlen(s);

    for (int i=1;i<=n;i++) {
        hs[i]=(hs[i-1]+s[i-1]*mpow[i]%MOD)%MOD;
        ht[i]=(ht[i-1]+t[i-1]*mpow[i]%MOD)%MOD;
    }
    ...
}
```

Suffix Automaton; Re: https://cp-algorithms.com/string/suffix-automaton.html

- #states <= 2n – 1 (n = s.length()), #transitions <= 3n – 4

```
struct state {
    int len, link; // terminal flag if necessary
    map<char, int> next; // list of transitions.
};

const int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last; // state corresponding to the entire string at the moment

void sa_init() {
    st[0].len = 0; st[0].link = -1;
    sz++; last = 0;
}

void sa_extend(char c) { // adds the next character to the SAM
    int cur = sz++;
    st[cur].len = st[last].len + 1;
    int p = last;
    while (p != -1 && !st[p].next.count(c)) {
        st[p].next[c] = cur;
        p = st[p].link;
    }
    if (p == -1) {
        st[cur].link = 0;
    } else {
        int q = st[p].next[c];
        if (st[p].len + 1 == st[q].len) {
            st[cur].link = q;
        } else {
            int clone = sz++;
            st[clone].len = st[p].len + 1;
            st[clone].next = st[q].next;
            st[clone].link = st[q].link;
            while (p != -1 && st[p].next[c] == q) {
                st[p].next[c] = clone; p = st[p].link;
            }
            st[q].link = st[cur].link = clone;
    } }
    last = cur;
} // O(nlogk). Use O(nk) memory for O(n) construction (array of sz k in each
state), and additional a list of all transitions*/
```



Suffix Automaton applications:

- Check for occurrence: Given a text T, and multiple patterns P, check if P appears as a substring of T
  - Build a SAM of T. To check if a pattern P appears in T, we follow the transitions, starting from t0, according to the characters of P. If at some point there doesn't exists a transition, then the pattern P doesn't appear as a substring of T. If we can process the entire P this way, then P appears in T.
  - O(length(P)) time for each string P. Algorithm finds length of longest prefix of P that appears in T.
- Number of different substrings: O(length(s))
  - Build SAM of S. Each substring of S corresponds to some path in the automaton. The number of different substrings is equal to the number of different paths in the automaton starting at t0.
  - Given that the suffix automaton is a DAG, the number of different ways can be computed via DP.
  - Namely, let d[v] be the number of ways, starting at the state v (including the path of length zero). Then we have the recursion: $d[v]=1+\sum w:(v,w,c)\in DAWG\ d[w]$ (i.e. d[v] can be expressed as the sum of answers for all ends of the transitions of v)
  - The number of different substrings is the value d[t0]−1 (since we don't count the empty substring).
- Total length of all different substrings: O(length(s))
  - Solution is similar to the previous one, only now it is necessary to consider two quantities for the dynamic programming part: the number of different substrings d[v] and their total length ans[v].
  - We already described how to compute d[v] in the previous task. The value ans[v] can be computed using the recursion: $ans[v]=\sum w:(v,w,c)\in DAWG\ d[w]+ans[w]$
  - We take the answer of each adjacent vertex w, and add to it d[w] (since every substrings is one character longer when starting from the state v).
- Lexicographically k-th substring: O(length(s)) preprocessing, O(length(ans) * k) per query
  - The solution of this problem is based on the idea of the previous two problems. The lexicographically k-th substring corresponds to the lexicographically k-th path in the suffix automaton. Therefore after counting the number of paths from each state, we can easily search for the k-th path starting from the root of the automaton.
- Smallest cyclic shift: O(length(s))
  - SAM for the string S+S, will contain in itself as paths all the cyclic shifts of the string S.
  - Problem is reduced to finding the lexicographically smallest path of length length(S): we start in the initial state and greedily pass through the transitions with the minimal character.
- Number of occurrences: O(length(T)) preprocessing, O(length(P)) query
  - For a given text T. We have to answer multiple queries. For each given pattern P we have to find out how many times the string P appears in the string T as substring.
  - We construct the SAM for T. Next we do the following preprocessing: for each state v in the automaton we calculate the number cnt[v] that is equal to the size of the set endpos(v). In fact all strings corresponding to the same state v appear in the text T an equal amount of times, which is equal to the number of positions in the set endpos. However we cannot construct the sets endpos explicitly, therefore we only consider their sizes cnt.
  - To compute them we proceed as follows. For each state, if it was not created by cloning (and if it is not the initial state t0), we initialize it with cnt=1. Then we will go through all states in decreasing order of their length len, and add the current value cnt[v] to the suffix links: cnt[link(v)] += cnt[v].
  - Thus we can compute the quantities cnt for all states in the automaton in O(length(T)) time.
  - After that answering a query by just looking up the value cnt[t], where t is the state corresponding to the pattern, if such a state exists. Otherwise answer with 0. Query takes O(length(P)) time.
- First occurrence position: O(length(s)) preprocessing, O(length(P)) query
  - Given a text T and multiple queries. For each query string P we want to find the position of the first occurrence of P in the string T (the position of the beginning of P).
  - Build SAM. Additionally, precompute position firstpos for all states in the automaton, i.e. for each state v we want to find the position firstpos[v] of the end of the first occurrence. i.e. we want to find in advance the minimal element of each set endpos
  - To maintain these positions firstpos we extend the function sa_extend(). When we create a new state cur, we set: firstpos(cur)=len(cur)−1.
  - And when we clone a vertex q as clone, we set: firstpos(clone)=firstpos(q) (since the only other option for a value would be firstpos(cur) which is definitely too big)
  - Answer for a query is simply firstpos(t)−length(P)+1, where t is the state corresponding to string P.
- Shortest non-appearing string
  - Given string S and alphabet. We have to find a string of smallest length, that doesn't appear in S.
  - DP on SAM of S: Let d[v] be the answer for the node v, i.e. we already processed part of the substring, are currently in the state v, and want to find the smallest number of characters that have to be added to find a non-existent transition.
  - Computing d[v]: If there is not transition using at least one character of the alphabet, then d[v]=1. Otherwise one character is not enough, and so we need to take the minimum of all answers of all transitions: $d[v]=1+\min w:(v,w,c)\in SAd[w]$. Answer to the problem will be d[t0], and the actual string can be restored using the computed array d[].

7. Classical Dynamic Programming: Avoid recomputation of subproblems!

Longest Increasing Subsequence (DP on subset/subsequence)

Let dp[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that dp[i] is the last element of the LIS.
Then, L(i) can be recursively written as:
L(i) = 1 + max(L(j)) where 0 < j < i and dp[j] < dp[i]; or
L(i) = 1, if no such j exists.
To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.

nlogn: L(i) represents the smallest ending value of all length-I LISs found so far, L(i-1) < L(i)

Digit DP

```
#include <cstdio> // CF 1073E: Sum of numbers s.t. #distinct digits <= k
const int mod = 998244353;
int tot, num[20];
long long k, l, r;
inline void up(long long &a, long long b) {if ((a += b) >= mod) a -= mod;}

struct node {
    long long cnt, sum;
} f[20][1 << 10];

long long pw[20];
node calc(int x, int lim, int S) {
    if (x==0) return (node) {1, 0};
    if (!lim && f[x][S].cnt==0) return f[x][S];
    node F = (node) {0, 0};
    for (int i = lim ? num[x] : 9; i>=0; i--) {
        int nxt;
        if (S==0 && i==0) nxt = 0;
        else nxt = S | 1 << i;
        if (__builtin_popcount(nxt) > k) continue;
        node tmp = calc(x - 1, lim && i == num[x], nxt);
        up(F.cnt, tmp.cnt);
        up(F.sum, (tmp.sum + tmp.cnt * pw[x - 1] % mod * i) % mod);
    }
    if (!lim) f[x][S] = F;
    return F;
}
long long solve(long long x) {
    tot = 0;
    while (x)
        num[++tot] = x % 10, x /= 10;
    return calc(tot, 1, 0).sum;
}
int main() {
    __builtin_memset(f, -1, sizeof f);
    scanf("%lld%lld%lld", &l, &r, &k);
    pw[0] = 1; for (int i = 1; i < 20; i++) pw[i] = pw[i - 1] * 10 % mod;
    printf("%lld\n", (solve(r) - solve(l - 1) + mod) % mod);
    return 0;
}
```

TSP: O($2^n * n^2$): Bitmask DP

```
int c[18][18], n, m, k; // CF 580D;
int a[18];
ll dp[1 << 18][18];

int main() {
    cin >> n >> m >> k;
    for (int i=0; i<n; i++)
        cin >> a[i];
    while (k--) {
        int x, y, z;
        cin >> x >> y >> z;
        c[x-1][y-1] = z;
    }

    // init
    for (int i=0; i<n; i++)
        dp[1 << i][i] = a[i];

    for (int mask=1; mask<(1<<n); mask++) {
        for (int i=0; i<n; i++) {
            if (mask & (1 << i)) {
                for (int j=0; j<n; j++) {
                    if (!(mask & (1 << j))) {
                        dp[mask | (1 << j)][j] = max(
                            dp[mask | (1 << j)][j],
                            dp[mask][i] + a[j] + c[i][j]);
    } } } } }

    ll z = 0;

    for (int mask=0; mask<(1 << n); mask++) {
        if (__builtin_popcount(mask) != m)
            continue;
        for (int i=0; i<n; i++)
            z = max(z, dp[mask][i]);
    }
    cout << z << '\n';
}
```

8. DP/Greedy intuition

DP Basic concepts:

- The problem has optimal sub-structures: The subproblem (overlapping) is part of the solution of the original problem. (avoid computing repeated sub-problems)
- Think of the state first, then the transition formula afterwards.
  - DP[index][choices you have for index]
  - DP[node][steps] (Re: shortest k-edge path in graph)
  - DP[value][# used] (subsequence, partitions)
- Bottom-up DP: Tabular method, visits and fills the value of all states
  - Determine the required set of parameters that uniquely describe the problem (state)
  - If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state.
  - Initialize the base cases
  - With the base-case cells/states in the DP table already filled, determine the cells/states that can be filled next (transitions) / For current state, see which states it relies on
  - Faster if many sub-problems are revisited as there is no overhead from recursive calls
- Top-down DP: Recursion with memorization (Easier to think of the states!)
  - Natural transformation from the normal complete search recursion
  - Computes the sub-problems only when necessary
- Rolling array: Store only the last 2 rows
  - Problems that involve the same row: Is the range of the "same row' correct? Re: 飛揚的小鳥
- For queries, don't reset the DP table, instead precompute and answer in O(1)
- Displaying the optimal solution:
  - Bottom-up DP: store (a list of) parent information at each state
  - Top-down DP: After transitioning to this state, can I build the optimal big solution?
- Common DP states:
  - DP on graph (counting # of shortest paths, path using k edges)
  - DP on interval: Where you cut the interval, how many you have used (Re: substrings)
  - DP on two strings: Ending point of the two strings (Re: Longest common subsequence)
  - DP on tree: Root it first, then for each node, think of the answer retrieval between it and parent/children
  - DP on DAG: to avoid unnecessary backtracking, travel in topological order
    - Transform into DAG: Longest/shortest path, Count number of paths
    - DP on grid: For current cell, which cells could it have come from?
  - DP on TSP O(n^2 * 2^n): Bitmask for visited cities + ending node. Preprocess the dist matrix first
  - Knapsack problem / Subsets (Bitmask DP)

Greedy:

- Greedy property: If we make a choice that seems like the best at the moment and proceed to solve the remaining subproblem, we reach the optimal solution. We never have to reconsider prev. choices.
- Optimal substructure: Optimal solution to the problem contains optimal solutions to sub-problems.
- Sort the input for greedy strategy: Find some criteria that a<b in the optimal sol, then sorting function
  - Intervals: Sort by increasing left endpoint, then decreasing right endpoint
- If the input size is small enough to use complete search or DP, use those.
  Only use a greedy algorithm if the input size given in the problem statement are too large.
- Economics in OI: MR=MC, Social cost vs Private cost, Low hanging fruit principle;

9. Exhaustion / Brute force

- Compute the 'potential worth' of a partial (and still valid) solution. If the potential worth is inferior to the worth of the current best found valid solution so far, we can prune the search there.
- Symmetry (Re: 8 queens problem, Sudoku)
- Pre-computation (generating the table, then hardcode!)
- Filtering (remove incorrect) vs Generating (prune in the middle)
- Backwards solving (Precomputation)

Exhaust permutation

```
do {
    ...
} while (next_permutation(a, a+n);
```

Exhaust selection (Fixed r)

```
for (int i=0; i<n; i++)
    for (int j=i+1; j<n; j++)
        for (int k=j+1; k<n; k++) ...
```

Exhaust bitset

```
for (int i=0; i<(1<<n); i++){
    reset();
    for (int j=0; j<n; j++)
        if (i & (1<<j))   // j-th item inside subset
    if ( CONDITION_OK ) break;
}
```

Exhaust subset

```
for (int s = (S - 1) & S; s; s = (s - 1) & S)
```

Exhaust size k bitset

```
template<typename T>
void subset(int k, int n, T&& f) {
    int t = (1 << k) - 1;
    while (t < 1 << n) {
        f(t);
        int x = t & -t, y = t + x;
        t = ((t & ~y) / x >> 1) | y;
    }
}
```

Recursive backtracking (Hamolitan path) (Graph coloring)

```
vector <int> path
void dfs(int v){
    s.pb(v);
    if (have enough edges in path) ok=1;   // s.size()==number_of_edges
    for (each vertex u connected to v){
        if (u is not used) {
            mark u as used;   dfs(u);   mark u as not used;
            path.pop_back();
} } }
```

10. Numerical

Matrix library (hqwhuang)

```
const ll MOD = 1e9+7;
inline double Mod(double v,double mod=MOD){
    return v;
}
inline ll Mod(ll v, ll mod = MOD){
    return (v+mod)%mod;
}
typedef vector<vector<ll>> matl;
typedef vector<vector<double>> matd;
template<class T> vector<T> operator*(const T & a,vector<T> v){
    rep(i,0,v.size()-1)
        v[i]=Mod(v[i]*a);
    return v;
}
template<class T> vector<T> operator+(vector<T> v,const vector<T> &w){
    rep(i,0,v.size()-1)
        v[i]=Mod(v[i]+w[i]);
    return v;
}
template<class T> vector<vector<T>> matE(T n){//正方形
    vector<vector<T>> re(n,vector<T>(n));
    rep(i,0,n-1)
        re[i][i] = 1;
    return re;
}
template<class T> vector<vector<T>> matE(vector<vector<T>> mat){//正方形
    return matE((T)(mat.size()));
}
template<class T>
vector<vector<T>> transpose(const vector<vector<T>> &a,vector<vector<T>> &re){//长方
形
    re.resize(a[0].size(),vector<T>(a.size()));
    rep(i,0,a[0].size()-1)
        rep(j,0,a.size()-1)
            re[i][j] = a[j][i];
    return re;
}
template<class T> T operator*(const vector<T> &a,const vector<T> &b){
    T re = 0;
    rep(i,0,a.size()-1)
        re = Mod(re+Mod(a[i]*b[i]));
    return re;
}
template<class T> vector<vector<T>> operator*(const vector<vector<T>> &a,const
vector<vector<T>> &b_){//长方形
    vector<vector<T>> b;
    transpose(b_,b);
    vector<vector<T>> re(a.size(),vector<T>(b.size()));
    rep(i,0,a.size()-1)
        rep(j,0,b.size()-1)
            re[i][j] = a[i]*b[j];
    return re;
}
template<class T>
vector<vector<T>> pow(vector<vector<T>> a,ll n){//正方形
    vector<vector<T>> re;
    if(n==0)
        return matE(a);
    re = pow(a,n/2);
    return re*re*(n%2?a:matE(a));
}
```

Gaussian elimination (System of linear equations)

```
// To use, build a matrix of coefficients and call run(mat, R, C).
// If the i-th variable is free, row[i] will be -1, otherwise ans[i].
//
// Time complexity: O(R * C^2)
//
// Constants to configure:
// - MAXC is the number of columns
// - eps is the epsilon value

namespace Gauss {
  const int MAXC = 1001;

  int row[MAXC];
  double ans[MAXC];

  void run(double mat[][MAXC], int R, int C) {
    REP(i, C) row[i] = -1;

    int r = 0;
    REP(c, C) {
      int k = r;
      FOR(i, r, R) if (fabs(mat[i][c]) > fabs(mat[k][c])) k = i;
      if (fabs(mat[k][c]) < eps) continue;

      REP(j, C+1) swap(mat[r][j], mat[k][j]);
      REP(i, R) if (i != r) {
        double w = mat[i][c] / mat[r][c];
        REP(j, C+1) mat[i][j] -= mat[r][j] * w;
      }
      row[c] = r++;
    }

    REP(i, C) {
      int r = row[i];
      ans[i] = r == -1 ? 0 : mat[r][C] / mat[r][i];
    }
  }
}
```

Absorbing Markov Chain

Solving linear recurrences

LP Simplex

11. Ad-hoc

Roman Numerals

```
string intToRoman(int num) {
    if(num <= 0) return "";
    string ret = "";
    static int number[13] = {1000, 900, 500, 400, 100,90, 50, 40, 10, 9, 5, 4, 1};
    static string flags[13] = {"M","CM", "D", "CD", "C", "XC", "L", "XL", "X",
"IX", "V", "IV", "I"};
    for(int i = 0; i < 13 && num > 0; i++){
        if(num < number[i]) continue;
            // cout<< i << " " << number[i] << " - "<<flags[i] << endl;
        while(num >= number[i]){
            num-= number[i];
            ret += flags[i];
        }
    }
    return ret;
}
int romanToInt(string s) {
    int tagVal[256];
    tagVal['I'] = 1;tagVal['V'] = 5;
    tagVal['X'] = 10;   tagVal['C'] = 100;
    tagVal['M'] = 1000; tagVal['L'] = 50;
    tagVal['D'] = 500;
    int val = 0;
    for(int i = 0; i < s.length(); i++){
        if (i+1 >= s.length() || tagVal[s[i+1]] <=tagVal[s[i]]) val +=tagVal[s[i]];
        else val -= tagVal[s[i]];
    }
    return val;
}
```

Dates

```
int dateToInt (int m, int d, int y){ // 1 based
    return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

void intToDate (int jd, int &m, int &d, int &y){
    int x, n, i, j;
    x = jd + 68569;
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}
// simple version: (d+=m<3?y--:y-2,23*m/9+d+4+y/4-y/100+y/400)%7
string intToDay (int jd){
    static string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
    return dayOfWeek[jd % 7];
}
// 3/24/2004, 2453089, Wed
```

Faster IO

```
template <typename T> inline void read(T& x){
    int w = 1; T data = 0; char ch = getchar();
    while(!isdigit(ch) && ch != '-') ch = getchar();
    if(ch == '-') w = -1, ch = getchar();
    while(isdigit(ch)) data = 10 * data + ch - '0', ch = getchar();
    x = data * w;
}
template <typename T> void write(T x){
    if (x < 0) putchar('-'),x = ~x + 1;
    if (x > 9) write(x / 10);
    putchar( x % 10 + '0');
}
template <typename T> inline void writeln(T x){
    write(x), putchar('\n');
}

// usage: read(n); write(n);
```

Largest area under histogram

```
int N; long long h[100001], b[100000], bl, ans;

int main() {
    cin >> n;  for (int i = 0; i < N; i++) cin >> h[i];
    for (int i = 0; i <= N; i++) {
        b[bl] = i;
        while (bl > 0 && h[i] < h[b[bl - 1]]) {
            ans = max(ans, h[b[bl - 1]] * (i - b[bl - 1]));
            bl--;
        }
        h[b[bl++]] = h[i];
    }
    cout << ans;
}
```

12. Visualizer

SVG Graphics / Visualizer (Circles, Lines, Polygons)

```
<!DOCTYPE html>
<html><body>
<svg width="600" height="600">
    <circle cx="50" cy="50" r="40"
        stroke="red" stroke-width="4" fill="blue" />
    <circle cx="60" cy="60" r="40"
        stroke="red" stroke-width="4" fill="#000000" />
    <line x1="0" y1="0" x2="200" y2="200"
        style="stroke:rgb(255,0,0);stroke-width:2" />
    <polygon points="200,10 250,190 160,210"
        style="fill:lime;stroke:purple;stroke-width:1" />
</svg>
</body></html>
```

CSS Grid/Graph Paper

```
<style>
    table, tr, td{ border: 1px solid black; }
    table { border-collapse: collapse; }
    td{ padding:8px; }
</style>

<table> <tr> <td></td> <td></td>...  </tr>...</table>
```