|  |  |
| --- | --- |
| **Shadow Brokers**  **(IIT Patna)**  INDEX  [Bit Manipulation:](#_gjdgxs) 4  [Techniques:](#_30j0zll) 4  [STL DS:](#_1fob9te) 5  [**STL Algorithms:**](#_3znysh7) **5**  [**Number Theory:**](#_2et92p0) **6**  [Probability:](#_tyjcwt) 7  Sieve of Eratosthenes 7  [Extended Euclid’s Algorithm:](#_3dy6vkm) 8  S[egmented Sieve for prime](#_1t3h5sf)s 8  [Modular power](#_4d34og8) 9  [Matrix Exponentiation](#_2s8eyo1) 10  [Euler’s totient:](#_17dp8vu) 10  [Largest power of p that divides n!](#_3rdcrjn) 11  [nCr (with lucas Theorem):](#_26in1rg) 11  [Chinese Remainder Theorem](#_lnxbz9) 12  [Wilson’s theorem](#_35nkun2) 12  [Inclusion-Exclusion:](#_1ksv4uv) 12  [Number of solutions to a linear eqn:](#_44sinio) 13  [Sum of GP:](#_2jxsxqh) 13  [Ternary Search (max of unimodal function):](#_z337ya) 14  [**Data Structures:**](#_3j2qqm3) **14**  [Iterative trie:](#_1y810tw) 14  [Iterative segment tree:](#_4i7ojhp) 15  [Lazy Segment tree:](#_2xcytpi) 16  [18](#_3whwml4)  [**Graph Theory**](#_qsh70q) **18**  [Dijkstra’s Algorithm:](#_3as4poj) 18  [Floyd Warshall(All pair)](#_1pxezwc) 19  [Bellman-Ford(for negative edges):](#_49x2ik5) 20  [Prim’s Algorithm for MST](#_2p2csry) 20  [LCA:](#_147n2zr) 20  [Topological Sort:](#_3o7alnk) 22  [Strongly Connected Components (Kasuraja’s Algo):](#_23ckvvd) 1  [Articulation Point (cut-vertices) and bridges:](#_ihv636) 23  [Euler path/circuit:](#_1hmsyys) 3  [Hierholzer’s algorithm for directed graph:](#_41mghml) 4  [Ford-Fulkerson max flow Algorithm:](#_2grqrue) 5  [Maximum Bipartite Matching:](#_vx1227) 6  [**Geometry:**](#_3fwokq0) **8**  [Orientation:](#_1v1yuxt) 9  [Line intersection:](#_4f1mdlm) 9  [Circle intersection area:](#_2u6wntf) 10  [Convex Hull:](#_19c6y18) 10  [Point in a polygon:](#_3tbugp1) 12  [**Game Theory:**](#_28h4qwu) **13**  [**Pattern Matching:**](#_nmf14n) **13**  [Suffix Arrays:](#_37m2jsg) 13  KMP(STL) 13  [KMP Algorithm:](#_1mrcu09) 15  [**Standard DP**](#_46r0co2) **17**  [LCS:](#_2lwamvv) 17  [Max contiguous subarray sum (Kadane’s Algo):](#_111kx3o) 18  [LIS in nlogn:](#_3l18frh) 19  [Coin Change Problem:](#_206ipza) 20  [Rod Cutting Problem:](#_4k668n3) 20  [Sum Of Subset:](#_2zbgiuw) 21  [Catalan numbers:](#_1egqt2p) 21  [0/1 Knapsack:](#_3ygebqi) 22  [Egg Drop Problem:](#_2dlolyb) 22  [Cap Assignment (bit-mask):](#_sqyw64) 23  **Other String Algorithms**[**:**](#_nmf14n) **23**  Manacher’s Algorithm 23  Z Algorithm 23 Bit Manipulation: 1. To multiply by 2^x : S = S<<x  2. To divide by 2^x : S = S>>x  3. To set jth bit : S|=(1<<j)  4. To check jth bit : T = S &(1<<j) (If T=0 not set else set)  5. To turn off jth bit : S&=~(1<<j)  6. To flip jth bit : S^=(1<<j)  7. To get value of LSB: T = (S &(-S)) (Gives 2^position)  8. To turn on all bits S = (1<<n) - 1  in a set of size n: Techniques: 1. For counting problems, try counting number of incorrect ways instead of correct ways.  2. Prune Infeasible/Inferior Search Space Early  3. Utilize Symmetries  4. Try solving the problem backwards  5.Binary Search the answer  6. Meet in the middle (Solve left half, Solve right half, combine)  7. Greedy  8. DP  9. Analyse complexity carefully  10. Reduce the problem to some standard problem  11. Add m when doing modular arithmetic.  12. Carefully analyse reasoning behind adding small details in the Q.  13. Use exponential search in case of unbounded search. STL DS: **stack<type> name**  empty(),size(),pop(),top(),push(x)  **queue<type> name**  empty(),size(),pop(),front(),back(),push(x)  **priority\_queue <type> name**  empty(),size(),pop(),top(),push(x)  **deque<type> name**  pop\_front(),pop\_back(),push\_front(),push\_back(),size(),at(index),front(),back()  **set/multiset/map/multimap<type>name**  begin(),end(),size(),empty(),insert(val),erase(itr or val),find(val),  lower\_bound(val),upper\_bound(val)  (lower bound includes val, upper bound does not)  pair<type,type> name (first and second) STL Algorithms: 1.sort(first\_iterator, last\_iterator) – To sort the given vector. 2. reverse(first\_iterator, last\_iterator) – To reverse a vector. 3. \*max\_element (first\_iterator, last\_iterator) – To find the maximum element of a vector.  4. \*min\_element (first\_iterator, last\_iterator) – To find the minimum element of a vector. 5. accumulate(first\_iterator, last\_iterator, initial value of sum) – Does the summation of vector elements  6. binary\_search(first\_iterator, last\_iterator, x) – Tests whether x exists in sorted vector or not. 7.lower\_bound(first\_iterator, last\_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value not less than ‘x’. 8.upper\_bound(first\_iterator, last\_iterator, x) – returns an iterator pointing to the first element in the range [first,last) which has a value greater than ‘x’.  9.count(first\_iterator, last\_iterator,x) – To count the occurrences of x in vector.  10.next\_permutation(first\_iterator, last\_iterator) – This modified the vector to its next permutation. 11.prev\_permutation(first\_iterator, last\_iterator) – This modified the vector to its previous permutation  12. random\_shuffle(arr.begin(), arr.end());  13. \_\_builtin\_popcount(unsigned int) Number Theory: 1. To calculate sum of factors of a number, we can find the number of prime factors and their exponents. N = ae1 \* be2 \* ce3 … Then sum = (1 + a + a^2….)(1 + b + b^2 .. )...  Number of factors=(a+1)\*(b+1)...  2.Every even integer greater than 2 can be expressed as the sum of 2 primes.  3. For rootn prime method, check for 2, 3 then:  for (i=5; i\*i<=n; i=i+6) n%i and n%(i+2)  4. Number of divisors will be prime only if N=p^x where p is prime.  5. Kth prime factor= store smallest factor in seive and repeatedly divide with it to get the answer.  6. fib(n+m)=fib(n)fib(m+1)+fib(n-1)fib(m)  7. A number is Fibonacci if and only if one or both of (5\*n2 + 4) or (5\*n2 – 4) is a perfect square  8. every positive Every positive integer can be written uniquely as a sum of distinct non-neighbouring Fibonacci numbers.  9. Matrix multiplication  mul[i][j] += a[i][k]\*b[k][j];  10. Root n under mod p exists only if  n^((p-1)/2) % p = 1  11.divisibility by 4: last 2 digits divisible by 4  12.divisibility by 8: last 3 digits divisible by 8  13. Divisibility by 3,9: sum of digs divisible by 3,9  14. Divisibility by 11: alternate (+ve,-ve) digit sum is divisible by 11  15. Divisibility by 12: divisible by 3 and 4  16. Divisibility by 13: alternating sum in blocks of 3 (L to R) div 13  17. Integral solution of ax+by=c exists if gcd(a,b) divides c Probability:   P(A∩B) = P(A) + P(B) - P(A∪B)  Probability of A if B has happened:  P(A|B) = P(A∩B) / P(B) expected value is the sum of: [(each of the possible outcomes) × (the probability of the outcome occurring)].  Var(X) = E(X^2) – m^2  Seive of Eratostones: vector<ll> prime; void SieveOfEratosthenes(ll n)  {   bool prim[n+1];   memset(prim, true, sizeof(prim));  prime.pb(2);  for(ll i=4; i<=n; i+=2) prim[i] = false;   for(ll i=3; i<=n; i+=2){  if(prim[i] ){  prime.pb(i);  for(ll j=2\*i; j<=n; j+=i) prim[i] = false;  } }} Extended Euclid’s Algorithm:  1. LL gcde(LL a,LL b,LL \*x,LL \*y) 2. { 3. if (a == 0) 4. { 5. \*x = 0, \*y = 1; 6. return b; 7. } 8. LL x1, y1; 9. LL gcd = gcde(b%a, a, &x1, &y1); 10. \*x = y1 - (b/a) \* x1; 11. \*y = x1; 12. return gcd; 13. }   To find inverse of a wrt m:  gcde(a,m,&x,&y);  x is the inverse of a. Segmented Sieve for primes  1. void segsieve(LL l,LL r) 2. { 3. LL limit = [floor](http://www.opengroup.org/onlinepubs/009695399/functions/floor.html)([sqrt](http://www.opengroup.org/onlinepubs/009695399/functions/sqrt.html)(r))+1; 4. vector<LL> prime; 5. sieve(limit, prime); 6. limit=r-l+1; 7. bool mark[limit+1]; 8. [memset](http://www.opengroup.org/onlinepubs/009695399/functions/memset.html)(mark, true, [sizeof](http://www.opengroup.org/onlinepubs/009695399/functions/sizeof.html)(mark));   //True= is prime   1. for (int i = 0; i < prime.size(); i++) 2. { 3. int loLim = [floor](http://www.opengroup.org/onlinepubs/009695399/functions/floor.html)(l/prime[i]) \* prime[i]; 4. if (loLim < l) 5. loLim += prime[i]; 7. for (int j=loLim; j<=r; j+=prime[i]) 8. mark[j-l] = false; 9. } 10. }  Modular power  1. LL Mpow(LL x, unsigned LL y, LL m) 2. { 3. LL res = 1; 4. x = x % m; 5. while (y > 0) 6. { 7. if (y & 1) 8. res = (res\*x) % m; 9. y = y>>1; // y = y/2 10. x = (x\*x) % m; } 11. Return res;}  Matrix Exponentiation LL power(LL F[3][3], LL n) {  LL M[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}};  if (n==1)  return F[0][0] + F[0][1];  power(F, n/2);  multiply(F, F);  if (n%2 != 0)  multiply(F, M);  return F[0][0] + F[0][1] ; }  LL findNthTerm(LL n) {  LL F[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}} ;  return power(F, n-2); } Euler’s totient: Number of integers coprime to n less than n  LL phi(LL n)  {  LL result = n;  for (LL p=2; p\*p<=n; ++p)  {  if (n % p == 0)  {  while (n % p == 0)  n /= p;  result -= result / p;  }  }  if (n > 1)  result -= result / n;  return result;  } Largest power of p that divides n! // Returns largest power of p that divides n! int largestPower(int n, int p) {  // Initialize result  int x = 0;    // Calculate x = n/p + n/(p^2) + n/(p^3) + ....  while (n)  {  n /= p;  x += n;  }  return x; } nCr (with lucas Theorem):  1. LL ncrp(LL n, LL r, LL p) 2. { 3. LL C[r+1]; 4. [memset](http://www.opengroup.org/onlinepubs/009695399/functions/memset.html)(C, 0, [sizeof](http://www.opengroup.org/onlinepubs/009695399/functions/sizeof.html)(C)); 5. C[0] = 1; 6. for (LL i = 1; i <= n; i++) 7. { 8. for ( LL j = min(i, r); j > 0; j--) 9. C[j] = (C[j] + C[j-1])%p; 10. } 11. return C[r]; 12. } 13. LL ncrpl(LL n,LL r, LL p) 14. { 15. if (r==0) 16. return 1; 17. int ni = n%p, ri = r%p; 18. return (ncrpl(n/p, r/p, p) \* 19. ncrp(ni, ri, p)) % p; 20. }  Chinese Remainder Theorem  1. LL crt(LL num[], LL rem[], LL k) 2. { 3. LL prod = 1; 4. for (int i = 0; i < k; i++) 5. prod \*= num[i]; 6. LL result = 0; 7. for (int i = 0; i < k; i++) 8. { 9. LL pp = prod / num[i]; 10. LL inv,y; 11. gcde(pp,num[i],&inv,&y); 12. result += rem[i] \* inv \* pp; 13. } 14. return result % prod; 15. }   For combining wrt a large number, use it 2 numbers at a time. Wilson’s theorem ((p-1)!)%p=-1 Inclusion-Exclusion: (A U B)= add 1 at a time, subtract 2 at a time …… Number of solutions to a linear eqn: LL countSol(LL coeff[], LL start, LL end, LL rhs)  {  // Base case  if (rhs == 0)  return 1;    LL result = 0; // Initialize count of solutions    // One by subtract all smaller or equal coefficiants and recur  for (LL i=start; i<=end; i++)  if (coeff[i] <= rhs)  result += countSol(coeff, i, end, rhs-coeff[i]);  return result;  } Sum of GP: long long gp(LL r, LL p,LL m){  if(p==0)  return 1;  if(p==1)  return 1;  LL ans=0;  if(p%2==1){  ans=Mpow(r,p-1,m);  ans=(ans+((1+r)\*gp(Mpow(r,2,m),(p-1)/2,m))%m)%m;  }  else{  ans=((1+r)\*gp(Mpow(r,2,m),p/2,m))%m;  }  return ans;  } Ternary Search (max of unimodal function): double ts(double start, double end) {  double l = start, r = end;   for(int i=0; i<200; i++) {  double l1 = (l\*2+r)/3;  double l2 = (l+2\*r)/3;  //cout<<l1<<" "<<l2<<endl;  if(func(l1) > func(l2)) r = l2; else l = l1;  }  return func(r); } Data Structures: **Segment Tree**  ll tree[4\*N] ,a[N] ;  ll buildTree( int pos,int l, int r) {  if(l==r) return tree[pos] = a[l] ;  else {  int mid = (l + r)>>1 ;  tree[pos] = max(buildTree(pos<<1,l,mid),buildTree(pos<<1|1,mid+1,r)) ;  }  }  ll rangeMaxQuery(int qlow, int qhigh, int l, int r, int pos) {  if(qlow<=l && qhigh>=r) return tree[pos] ;  if(qlow>r || qhigh<l) return 9e18 ;  int mid = (l+r)>>1;  return max(rangeMaxQuery(qlow, qhigh,l,mid,pos<<1),rangeMaxQuery(qlow, qhigh,mid+1,r,pos<<1|1)) ;  }  ll updateTree(int l,int r, int qlow, int qhigh, int pos, int value) {  if(l > r || l >qhigh || r < qlow) return 0;  if(l==r) return tree[pos] += value ;  int mid=(l+r)>>1;  return tree[pos] = max(updateTree(l,mid,qlow,qhigh,pos<<1,value) ,updateTree(mid+1,r,qlow,qhigh,pos<<1|1,value)) ;  }  **Lazy Sement Tree**  // no change in build function  void updlzy(int l,int r,int pos){  if(lazy[pos]){  tree[pos]+=lazy[pos];  if(l!=r) lazy[pos<<1]+=lazy[pos],lazy[pos<<1|1]+=lazy[pos];  }  lazy[pos]=0;  }  ll update(int l,int r,int ul,int ur,int val,int pos){  updlzy(l,r,pos);  if(l>ur || ul>r) return 0;  else if(ul<=l && r<=ur){  tree[pos]+=val;  lazy[pos<<1]+=val,lazy[pos<<1|1]+=val;  return tree[pos];  }  int mid=(l+r)>>1;  tree[pos]=max(update(l,mid,ul,ur,val,pos<<1),updlzy(mid+1,r,ul,ur,val,pos<<1|1));  }  ll query(int l,int r,int ql,int qr,int pos){  updlzy(l,r,pos);  if(l>ur || ul>r) return 0;  else if(ql<=l && r<=qr) return tree[pos];  int mid=(l+r)>>1;  return max(query(l,mid,ql,qr,pos<<1),query(mid+1,r,ql,qr,pos<<1|1));  }  **Union-Find:**  LL find(struct subset subsets[], LL i){  if (subsets[i].parent != i)  subsets[i].parent = find(subsets, subsets[i].parent);  return subsets[i].parent;  }  void Union(struct subset subsets[], LL x, LL y){  LL xroot = find(subsets, x);  LL yroot = find(subsets, y);  // Attach smaller rank tree under root of high rank tree  if (subsets[xroot].rank < subsets[yroot].rank)  subsets[xroot].parent = yroot;  else if (subsets[xroot].rank > subsets[yroot].rank)  subsets[yroot].parent = xroot;  else{  subsets[yroot].parent = xroot;  subsets[xroot].rank++;  }  }  **Trie:**  struct node {  node \*child[26];  bool end;  };  node \*getNode() {  node \*p = new node;  p->end = 0;  for (int i = 0; i < 26; i++)  p->child[i] = NULL;  return p;  }  void insert(node \*root, string key) {  node \*tmp = root;  for (int i = 0; i < key.length(); i++) {  int index = key[i] - 'a';  if (!tmp->child[index])  tmp->child[index] = getNode();  tmp = tmp->child[index];  }  pCrawl->isEndOfWord = 1;  }  bool search(node \*root, string key)  {  node \*tmp = root;    for (int i = 0; i < key.length(); i++) {  int index = key[i] - 'a';  if (!tmp->child[index])  return 0;  tmp = tmp->child[index];  }  return (tmp && tmp->end);  }  int main(){  // after taking input  node \*root = getNode();  For(i,0,N)  insert(root,s[i]);  } Graph TheoryDijkstra’s Algorithm: priority\_queue< pi, vector<pi> , greater<pi> > pq;  vector<ll> dist(n+1,intmax);  dist[1] = 0;  pq.push(mp(0,1));  parent[1] = -1;  while(!pq.empty()){  ll u = pq.top().ss;  pq.pop();  FOR0(i,v[u].size()){  ll wt = v[u][i].ss;  ll vertex = v[u][i].ff;  if (dist[vertex]>dist[u]+wt){  dist[vertex] = dist[u] + wt;  parent[vertex] = u;  pq.push(mp(dist[vertex],vertex));  }  }  } Floyd Warshall(All pair) for (k = 0; k < V; k++)  for (i = 0; i < V; i++)  for (j = 0; j < V; j++)  if (dist[i][k] + dist[k][j] < dist[i][j])  dist[i][j] = dist[i][k] + dist[k][j]; Bellman-Ford(for negative edges): void BellmanFord(struct Graph\* graph, LL src)  {  LL V = graph->V;  LL E = graph->E;  LL dist[V];  for (LL i = 0; i < V; i++)  dist[i] = INT\_MAX;  dist[src] = 0;  for (LL i = 1; i <= V-1; i++)  {  for (LL j = 0; j < E; j++)  {  LL u = graph->edge[j].src;  LL v = graph->edge[j].dest;  LL weight = graph->edge[j].weight;  if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])  dist[v] = dist[u] + weight;  }  }//to check for negative weight cycle, repeat above  } // if shorter path is found, cycle exists Prim’s Algorithm for MST void primMST()  {  priority\_queue< pi, vector<pi> , greater<pi> > pq;  LL src = 0;  vector<LL> key(V, INF);  vector<LL> parent(V, -1);  vector<bool> inMST(V, false);  pq.push(make\_pair(0, src));  key[src] = 0;  while (!pq.empty())  {  LL u = pq.top().second;  pq.pop();  inMST[u] = true; // Include vertex in MST  list< pair<LL, LL> >::iterator i;  for (i = adj[u].begin(); i != adj[u].end(); ++i)  {  LL v = (\*i).first;  LL weight = (\*i).second;  if (inMST[v] == false && key[v] > weight)  {  key[v] = weight;  pq.push(make\_pair(key[v], v));  parent[v] = u;  }  }}}   LCA: **Pre-processing: O(nlogn) , Query: O(logn)**  vector <int> tree[MAXN];  int depth[MAXN];  int parent[MAXN][level];  // pre-compute depth for each node and their first parent(2^0th parent)  void dfs(int cur, int prev){  depth[cur] = depth[prev] + 1;  parent[cur][0] = prev;  for (int i=0; i<tree[cur].size(); i++) {  if (tree[cur][i] != prev)  dfs(tree[cur][i], cur);  }  }  void precomputeSparseMatrix(int n){  for (int i=1; i<level; i++){  for (int node = 1; node <= n; node++){  if (parent[node][i-1] != -1)  parent[node][i]=parent[parent[node][i-1]][i-1];  } }}  int lca(int u, int v){  if (depth[v] < depth[u]) swap(u, v);  int diff = depth[v] - depth[u];  for (int i=0; i<level; i++)  if ((diff>>i)&1)  v = parent[v][i];  if (u == v) return u;  for (int i=level-1; i>=0; i--)  if (parent[u][i] != parent[v][i]){  u = parent[u][i];  v = parent[v][i];  }  return parent[u][0];  } Topological Sort: void topologicalSortUtil(LL v, bool visited[],  stack<LL> &Stack)  {  visited[v] = true;  list<LL>::iterator i;  for (i = adj[v].begin(); i != adj[v].end(); ++i)  if (!visited[\*i])  topologicalSortUtil(\*i, visited, Stack);  Stack.push(v);  }  void topologicalSort()  {  stack<LL> Stack;  bool \*visited = new bool[V];  for (LL i = 0; i < V; i++)  visited[i] = false;  for (LL i = 0; i < V; i++)  if (visited[i] == false)  topologicalSortUtil(i, visited, Stack);  while (Stack.empty() == false)  {  cout << Stack.top() << " ";  Stack.pop();  } Manacher's Algorithm: **return longest palindromic substring in O(n).** string manacher(string s){   ll len = s.length();  string ne = "@";  fr(i,len)  ne+= "#"+s[i] ;  ne += "#$";   len = ne.size();    ll p[len+1] = {0}, c=0,r=0;  fre(i,len-2){  ll imirror = 2\*c-i;  if(r>i) p[i] = min(r-i, p[imirror]);  while(ne[i+1+p[i]]==ne[i-1-p[i]]) p[i]++;  if(i+p[i]>r) c=i, r = i+p[i];  }  ll mlen = 0, cind = 0;  fre(i,len-2) {  if(p[i]>mlen) mlen = p[i], cind = i;  }  return s.substr((cind-mlen-1)/2, mlen); } Z Algorithm: **O( c.length() + s.length() )** **String c need to be find out in string s; z[i] stores the maximum length of substring starting from ith position which is prefix of a. We need to find how many times z[i] = c.length() a = c+'&' + s where & is character that is not present in either of the strings.** void zalgo(string s, string c ){  string a = c+"#"+ s;  ll n = a.length();  ll z[n+1], l=0,r=0,k ;  z[0] = 0;  fre(i,n-1){  if(i>r){  l = r = i;  while(r<n && a[r]==a[r-l]) r++;  z[i] = r-l;  r--;  }  else {  k = i-l;  if(z[k]< r-i+1) z[i] = z[k];  else{  l = i;  while(r<n && a[r]==a[r-l] )r++;  z[i] = r-l;  r--;  }} }  ll m = c.length(), ans=0;  fre(i,n-1)  {  if(z[i]== m)  ans++;  }}  MACROS :  //rg99  #include<bits/stdc++.h>  #define FOR0(i,n) for(ll i=0;i<n;i++)  #define FOR1(i,n) for(ll i=1;i<=n;i++)  #define FORl(i,l,n) for(ll i=l;i<n;i++)  using namespace std;  #define pi pair<ll,ll>  #define pb push\_back  #define ll long long  #define ld long double  #define ff first  #define ss second  #define mp make\_pair  #define vi vector<ll>  #define sync ios\_base::sync\_with\_stdio(0); cin.tie(0); cout.tie(0);  #define endl '\n'  #define sp <<" "<<  #define intmax 1e14  ll power(ll num,ll g,ll mod){    if(g==0)return 1;    if(g%2==1)return (num\*power((num\*num)%mod,g/2,mod))%mod;    return power((num\*num)%mod,g/2,mod);  }  Closest Pair:  Suppose that we have processed points 1 to N - 1 (ordered by X) and the shortest distance we have found so far is h. We now process point N and try to find a point closer to it than h. We maintain a set of all already-processed points whose X coordinates are within h of point N, as shown in the light grey rectangle. As each point is processed, it is added to the set, and when we move on to the next point or when h is decreased, points are removed from the set. The set is ordered by y coordinate. A balanced binary tree is suitable for this, and accounts for the log N factor. To search for points closer than h to point N, we need only consider points in the active set, and furthermore we need only consider points whose y coordinates are in the range yN - h to yN + h (those in the dark grey rectangle). This range can be extracted from the sorted set in O(log N) time, but more importantly the number of elements is O(1) (the exact maximum will depend on the metric used), because the separation between any two points in the set is at least h. It follows that the search for each point requires O(log N) time, giving a total of O(N log N)  Line Segment Intersections :  We’ll start by considering the problem of returning all intersections in a set of horizontal and vertical line segments. Since horizontal lines don’t have a single X coordinate, we have to abandon the idea of sorting objects by X. Instead, we have the idea of an event: an X coordinate at which something interesting happens. In this case, the three types of events are: start of a horizontal line, end of a horizontal line, and a vertical line. As the sweep line moves, we’ll keep an active set of horizontal lines cut by the sweep line, sorted by Y value. To handle either of the horizontal line events, we simply need to add or remove an element from the set. Again, we can use a balanced binary tree to guarantee O(log N) time for these operations. When we hit a vertical line, a range search immediately gives all the horizontal lines that it cuts. If horizontal or vertical segments can overlap there is some extra work required, and we must also consider whether lines with coincident endpoints are considered to intersect, but none of this affects the computational complexity.  If the intersections themselves are required, this takes O(N log N + I) time for I intersections. By augmenting the binary tree structure (specifically, by storing the size of each sub-tree in the root of that sub-tree), it is possible to count the intersections in O(N log N) time.  In the more general case, lines need not be horizontal or vertical, so lines in the active set can exchange places when they intersect. Instead of having all the events pre-sorted, we have to use a priority queue and dynamically add and remove intersection events. At any point in time, the priority queue contains events for the end-points of line-segments, but also for the intersection points of adjacent elements of the active set (providing they are in the future). Since there are O(N + I) events that will be reached, and each requires O(log N) time to update the active set and the priority queue, this algorithm takes O(N log N + I log N) time. The figure below shows the future events in the priority queue (blue dots); note that not all future intersections are in the queue, either because one of the lines isn’t yet active, or because the two lines are not yet adjacent in the active list.  Area of Union of Rectangles  Given a set of axis-aligned rectangles, what is the area of their union? Like the line-intersection problem, we can handle this by dealing with events and active sets. Each rectangle has two events: left edge and right edge. When we cross the left edge, the rectangle is added to the active set. When we cross the right edge, it is removed from the active set.  We now know which rectangles are cut by the sweep line (red in the diagram), but we actually want to know the length of sweep line that is cut (the total length of the solid blue segments). Multiplying this length by the horizontal distance between events gives the area swept out between those two events.  We can determine the cut length by running the same algorithm in an inner loop, but rotated 90 degrees. Ignore the inactive rectangles, and consider a horizontal sweep line that moves top-down. The events are now the horizontal edges of the active rectangles, and every time we cross one, we can simply increment or decrement a counter that says how many rectangles overlap at the current point. The cut length increases as long as the counter is non-zero. Of course, we do not increase it continuously, but rather while moving from one event to the next.  With the right data structures, this can be implemented in O(N2) time (hint: use a boolean array to store the active set rather than a balanced binary tree, and pre-sort the entire set of horizontal edges). In fact the inner line sweep can be replaced by some clever binary tree manipulation to reduce the overall time to O(N log N), but that is more a problem in data structures than in geometry, and is left as an exercise for the reader. The algorithm can also be adapted to answer similar questions, such as the total perimeter length of the union or the maximum number of rectangles that overlap at any point. | Strongly Connected Components (Kasuraja’s Algo): void fillOrder(int v, bool visited[], stack<int> &Stack)  {  visited[v] = true;  list<int>::iterator i;  for(i = adj[v].begin(); i != adj[v].end(); ++i)  if(!visited[\*i])  fillOrder(\*i, visited, Stack);  Stack.push(v);  }  void printSCCs()  {  stack<int> Stack;  bool \*visited = new bool[V];  for(int i = 0; i < V; i++)  visited[i] = false;  // Fill vertices in stack according to their finishing times  for(int i = 0; i < V; i++)  if(visited[i] == false)  fillOrder(i, visited, Stack);  Graph gr = getTranspose();  for(int i = 0; i < V; i++)  visited[i] = false;  while (Stack.empty() == false)  {  // Pop a vertex from stack  int v = Stack.top();  Stack.pop();  if (visited[v] == false)  {  gr.DFSUtil(v, visited);  cout << endl;  }  }} Articulation points and Bridges: **v** : vector used to store adjacency list  **visited** : boolean array to keep track of nodes visited  **disc** : int array to store discovered time of vertex  **low** is int array to which stores, for every vertex v, the discovery time of the earliest discovered vertex to which v or any vertices in the subtree rooted at v is having a back edge. initialized by INFINITY. **parent** : int array used to store parent of each node. **is** : bool array if ith vertex is an articulation point.  **time** : used to keep track of discovered time. **ans** : vector of pair<int ,int> used to store bridges.  void dfs(ll x, ll time) {  visited[x] = true;  disc[x] = low[x] = time+1;  ll child = 0;  fr(i,v[x].size()) {  ll a = v[x][i];  if(a==parent[x]) continue;   if(visited[a]) low[x] = min(low[x] , disc[a] );  else {  child++;  parent[a] = x;  dfs(a,time+1);  low[x] = min(low[x], low[a]);  if(parent[x]==-1 && child>1)  is[x] = true,num++;  else if(parent[x]!=-1 && low[a]>=disc[x])  is[x] = true,num++;  if(low[a]>disc[x])  ans.pb(mp(x,a));  }} } Euler path/circuit: Euler path in undirected graph:  Graph is connected and all vertices have even degree except or 2 have odd degrees.  Euler Circuit in undirected graph:  All vertices have even degree and graph is connected.  Euler circuit in directed graph:  All vertices are a part of a single strongly connected component and indegree and outdegree of all vertices is same, Hierholzer’s algorithm for directed graph: void printCircuit(vector< vector<int> > adj)  {  unordered\_map<int,int> edge\_count;    for (int i=0; i<adj.size(); i++)  {  edge\_count[i] = adj[i].size();  }    if (!adj.size())  return;  stack<int> curr\_path;  vector<int> circuit;  curr\_path.push(0);  int curr\_v = 0;    while (!curr\_path.empty())  {  if (edge\_count[curr\_v])  {  curr\_path.push(curr\_v);  int next\_v = adj[curr\_v].back();  edge\_count[curr\_v]--;  adj[curr\_v].pop\_back();  curr\_v = next\_v;  }  else  {  circuit.push\_back(curr\_v);  curr\_v = curr\_path.top();  curr\_path.pop();  }  }  for (int i=circuit.size()-1; i>=0; i--)  {  cout << circuit[i];  if (i)  cout<<" -> ";  }  }  Bipartite graph: Coloring possible with 2 colors. Ford-Fulkerson (Edmond Karp) max flow Algorithm: O(EV^3)  bool bfs(int rGraph[V][V], int s, int t, int parent[])  {  bool visited[V];  memset(visited, 0, sizeof(visited));  queue <int> q;  q.push(s);  visited[s] = true;  parent[s] = -1;  while (!q.empty())  {  int u = q.front();  q.pop();    for (int v=0; v<V; v++)  {  if (visited[v]==false && rGraph[u][v] > 0)  {  q.push(v);  parent[v] = u;  visited[v] = true;  }  }  }  return (visited[t] == true);  }  int fordFulkerson(int graph[V][V], int s, int t)  {  int u, v;  int rGraph[V][V];  for (u = 0; u < V; u++)  for (v = 0; v < V; v++)  rGraph[u][v] = graph[u][v];    int parent[V];    int max\_flow = 0;  while (bfs(rGraph, s, t, parent))  {  int path\_flow = INT\_MAX;  for (v=t; v!=s; v=parent[v])  {  u = parent[v];  path\_flow = min(path\_flow, rGraph[u][v]);  }  for (v=t; v != s; v=parent[v])  {  u = parent[v];  rGraph[u][v] -= path\_flow;  rGraph[v][u] += path\_flow;  }  max\_flow += path\_flow;  }  return max\_flow;  } Dinic’s Algorithm:  **O(VE^2)** const int MAXN = ...;  const int INF = 1000000000;    int n, c[MAXN][MAXN], f[MAXN][MAXN], s, t, d[MAXN], ptr[MAXN], q[MAXN];   bool bfs() {  int qh=0, qt=0;  q[qt++] = s;  memset (d, -1, n \* sizeof d[0]);  d[s] = 0;  while (qh < qt) {  int v = q[qh++];  for (int to=0; to<n; ++to)  if (d[to] == -1 && f[v][to] < c[v][to]){  q[qt++] = to;  d[to] = d[v] + 1;  }}  return d[t] != -1; }  int dfs (int v, int flow) {  if (!flow) return 0;  if (v == t) return flow;  for (int & to=ptr[v]; to<n; ++to) {  if (d[to] != d[v] + 1) continue;  int pushed = dfs (to, min (flow, c[v][to] - f[v][to]));  if (pushed) {  f[v][to] += pushed;  f[to][v] -= pushed;  return pushed;  }  }  return 0; }  int dinic()  {  int flow = 0;  for (;;) {  if (!bfs()) break;  memset (ptr, 0, n \* sizeof ptr[0]);  while (int pushed = dfs (s, INF))  flow += pushed;  }  return flow; } Maximum Bipartite Matching: **O(M\*N\*N)**  bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[])  {  // Try every job one by one  for (int v = 0; v < N; v++)  {  // If applicant u is interested in job v and v is  // not visited  if (bpGraph[u][v] && !seen[v])  {  seen[v] = true; // Mark v as visited  // If job 'v' is not assigned to an applicant OR  // previously assigned applicant for job v (which is matchR[v])  // has an alternate job available.  // Since v is marked as visited in the above line, matchR[v]  // in the following recursive call will not get job 'v' again  if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR))  {  matchR[v] = u;  return true;  }  }  }  return false;  }  int maxBPM(bool bpGraph[M][N])  {  // The value of matchR[i] is the applicant number  // assigned to job i  int matchR[N];  memset(matchR, -1, sizeof(matchR));    int result = 0; // Count of jobs assigned to applicants  for (int u = 0; u < M; u++)  {  // Mark all jobs as not seen for next applicant.  bool seen[N];  memset(seen, 0, sizeof(seen));    // Find if the applicant 'u' can get a job  if (bpm(bpGraph, u, seen, matchR))  result++;  }  return result;  } Geometry: 1.Area of a regular polygon(equal sides)  2. Angle between (m1, b1) and (m2, b2):  arctan ((m2 − m1) / (m1 · m2 + 1))  3. Triangle: Area = a · b · sin γ / 2  • Area = | x1 · y2 + x2 · y3 + x3 · y1 − y1 · x2 − y2 · x3 − y3 · x1 | / 2  • Heron’s formula:  Let s = (a + b + c) / 2; then Area = s⋅(s − a)⋅(s − b)⋅(s − c)  4. Circle: (x − xc)^2+ (y − yc)^2= r^2  5.Polygon area (vertex coordinates):  | x1 · y2 + x2 · y3 + ... + xn · y1 − y1 · x2 − y2 · x3 − ... − yn · x1 | / 2 Orientation: LL orientation(PoLL p1, PoLL p2, PoLL p3)  {  LL val = (p2.y - p1.y) \* (p3.x - p2.x) -  (p2.x - p1.x) \* (p3.y - p2.y);    if (val == 0) return 0; // colinear    return (val > 0)? 1: 2; // clock or counterclock wise  } Line intersection: bool onSegment(PoLL p, PoLL q, PoLL r)  {  if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&  q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))  return true;  return false;  }  bool doIntersect(PoLL p1, PoLL q1, PoLL p2, PoLL q2)  {  LL o1 = orientation(p1, q1, p2);  LL o2 = orientation(p1, q1, q2);  LL o3 = orientation(p2, q2, p1);  LL o4 = orientation(p2, q2, q1);  if (o1 != o2 && o3 != o4)  return true;  if (o1 == 0 && onSegment(p1, p2, q1)) return true;  if (o2 == 0 && onSegment(p1, q2, q1)) return true;  if (o3 == 0 && onSegment(p2, p1, q2)) return true;  if (o4 == 0 && onSegment(p2, q1, q2)) return true;    return false;} Circle intersection area: int areaOfIntersection(x0, y0, r0, x1, y1, r1){  var rr0 = r0\*r0;  var rr1 = r1\*r1;  var c = Math.sqrt((x1-x0)\*(x1- x0) +(y1-y0)\*(y1- y0));  var phi =(Math.acos((rr0+(c\*c)-rr1) /(2\*r0\*c)))\*2;  var theta =(Math.acos((rr1+(c\*c)-rr0) /(2\*r1\*c)))\*2;  var area1 = 0.5\*theta\*rr1 - 0.5\*rr1\*Math.sin(theta);  var area2 = 0.5\*phi\*rr0 - 0.5\*rr0\*Math.sin(phi);  return area1 + area2;  } Convex Hull: Point nextToTop(stack<Point> &S)  {  Point p = S.top();  S.pop();  Point res = S.top();  S.push(p);  return res;  }  int distSq(Point p1, Point p2)  {  return (p1.x - p2.x)\*(p1.x - p2.x) +  (p1.y - p2.y)\*(p1.y - p2.y);  }  int compare(const void \*vp1, const void \*vp2)  {  Point \*p1 = (Point \*)vp1;  Point \*p2 = (Point \*)vp2;  int o = orientation(p0, \*p1, \*p2);  if (o == 0)  return (distSq(p0, \*p2) >= distSq(p0, \*p1))? -1 : 1;  return (o == 2)? -1: 1;  }  void convexHull(Point points[], int n)  {  int ymin = points[0].y, min = 0;  for (int i = 1; i < n; i++)  {  int y = points[i].y;  if ((y < ymin) || (ymin == y &&  points[i].x < points[min].x))  ymin = points[i].y, min = i;  }  swap(points[0], points[min]);  p0 = points[0];  qsort(&points[1], n-1, sizeof(Point), compare);  int m = 1;  for (int i=1; i<n; i++)  {  // Keep removing i while angle of i and i+1 is same  while (i < n-1 && orientation(p0, points[i],  points[i+1]) == 0)  i++;  points[m] = points[i];  m++;  }  if (m < 3) return;  stack<Point> S;  S.push(points[0]);  S.push(points[1]);  S.push(points[2]);  for (int i = 3; i < m; i++)  {  while (orientation(nextToTop(S), S.top(), points[i]) != 2)  S.pop();  S.push(points[i]);  }  while (!S.empty())  {  Point p = S.top();  cout << "(" << p.x << ", " << p.y <<")" << endl;  S.pop();  }  } Point in a polygon: bool isInside(Point polygon[], int n, Point p)  {  if (n < 3) return false;  Point extreme = {INF, p.y};  int count = 0, i = 0;  do  {  int next = (i+1)%n;  if (doIntersect(polygon[i], polygon[next], p, extreme))  {  if (orientation(polygon[i], p, polygon[next]) == 0)  return onSegment(polygon[i], p, polygon[next]);    count++;  }  i = next;  } while (i != 0);  return count&1; // Same as (count%2 == 1)  } Game Theory: 1. If nim-sum is non-zero, player starting first wins.  2. Mex: smallest non-negative number not present in a set.  3. Grundy=0 means game lost.  4. Grundy=mex of all possible next states.  5. Sprague-Grundy theorem:  If a game consists of sub games (nim with multiple piles)  Calculate grundy number of each sub game (each pile)  Take xor of all grundy numbers:  If non-zero, player starting first wins. Pattern Matching:Suffix Arrays: struct suffix  {  int index; // To store original index  int rank[2]; // To store ranks and next rank pair  };  int cmp(struct suffix a, struct suffix b)  {  return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):  (a.rank[0] < b.rank[0] ?1: 0);  }  int \*buildSuffixArray(char \*txt, int n)  {  struct suffix suffixes[n];  for (int i = 0; i < n; i++)  {  suffixes[i].index = i;  suffixes[i].rank[0] = txt[i] - 'a';  suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;  }  sort(suffixes, suffixes+n, cmp);  int ind[n];  for (int k = 4; k < 2\*n; k = k\*2)  {  int rank = 0;  int prev\_rank = suffixes[0].rank[0];  suffixes[0].rank[0] = rank;  ind[suffixes[0].index] = 0;  for (int i = 1; i < n; i++)  {  if (suffixes[i].rank[0] == prev\_rank &&  suffixes[i].rank[1] == suffixes[i-1].rank[1])  {  prev\_rank = suffixes[i].rank[0];  suffixes[i].rank[0] = rank;  }  else  {  prev\_rank = suffixes[i].rank[0];  suffixes[i].rank[0] = ++rank;  }  ind[suffixes[i].index] = i;  }  for (int i = 0; i < n; i++)  {  int nextindex = suffixes[i].index + k/2;  suffixes[i].rank[1] = (nextindex < n)?  suffixes[ind[nextindex]].rank[0]: -1;  }  sort(suffixes, suffixes+n, cmp);  }  // Store indexes of all sorted suffixes in the suffix array  int \*suffixArr = new int[n];  for (int i = 0; i < n; i++)  suffixArr[i] = suffixes[i].index;  return suffixArr;  void search(char \*pat, char \*txt, int \*suffArr, int n)  {  int m = strlen(pat);  int l = 0, r = n-1;  while (l <= r)  {  int mid = l + (r - l)/2;  int res = strncmp(pat, txt+suffArr[mid], m);  if (res == 0)  {  cout << "Pattern found at index " << suffArr[mid];  return;  }  if (res < 0) r = mid - 1;  else l = mid + 1;  }  cout << "Pattern not found";  } KMP Algorithm(STL):  std::size\_t found = a.find(b, 0); while(found != std::string::npos) {  std::cout << "found!" << '\n';  found = a.find(b, found+1); } KMP Algorithm(STL):  **KMP b stores the string(pattern)  we need to find it occurrences in string a. and vector v stores occurrences of b in a**  void kmp(string a, string b){  vector<ll> v;  ll n = a.length() , m = b.length();  /\* Compute temporary array pre[m] to maintain  size of suffix which is same as prefix \*/  ll pre[m] , i=1, j=0;  pre[0] = 0;  while(i<m) {  if(b[i]==b[j])  pre[i] = j+1, i++, j++;  else if(b[i]!=b[j]){  if(j==0) pre[i]=0, i++;  else j = pre[j-1];  }}  i=0, j=0;  /\* Search for pattern in text. \*/  while(i<n) {  if(a[i]==b[j]){  i++, j++;  if(j==m){  v.pb(i+1-m);  j = pre[j-1];  }}  else{  if(j==0) i++;  else j =pre[j-1];  }}} Standard DPLCS: void lcs( char \*X, char \*Y, LL m, LL n )  {  LL L[m+1][n+1];  for (LL i=0; i<=m; i++)  {  for (LL j=0; j<=n; j++)  {  if (i == 0 || j == 0)  L[i][j] = 0;  else if (X[i-1] == Y[j-1])  L[i][j] = L[i-1][j-1] + 1;  else  L[i][j] = max(L[i-1][j], L[i][j-1]);  }  }  // Following code is used to prLL LCS  LL index = L[m][n];  char lcs[index+1];  lcs[index] = '\0'; // Set the terminating character  LL i = m, j = n;  while (i > 0 && j > 0)  {  if (X[i-1] == Y[j-1])  {  lcs[index-1] = X[i-1]; // Put current character in result  i--; j--; index--; // reduce values of i, j and index  }  else if (L[i-1][j] > L[i][j-1])  i--;  else  j--;  }  cout << "LCS of " << X << " and " << Y << " is " << lcs;  } Max contiguous subarray sum (Kadane’s Algo): LL maxSubArraySum(LL a[], LL size)  {  LL max\_so\_far = a[0];  LL curr\_max = a[0];    for (LL i = 1; i < size; i++)  {  curr\_max = max(a[i], curr\_max+a[i]);  max\_so\_far = max(max\_so\_far, curr\_max);  }  return max\_so\_far;  } LIS in nlogn: LL CeilIndex(std::vector<LL> &v, LL l, LL r, LL key) {  while (r-l > 1) {  LL m = l + (r-l)/2;  if (v[m] >= key)  r = m;  else  l = m;  }  return r;  }    LL LongestIncreasingSubsequenceLength(std::vector<LL> &v) {  if (v.size() == 0)  return 0;    std::vector<LL> tail(v.size(), 0);  LL length = 1; // always poLLs empty slot in tail    tail[0] = v[0];  for (size\_t i = 1; i < v.size(); i++) {  if (v[i] < tail[0])  tail[0] = v[i];  else if (v[i] > tail[length-1])  tail[length++] = v[i];  else  tail[CeilIndex(tail, -1, length-1, v[i])] = v[i];  }    return length;  } Coin Change Problem: int count( int S[], int m, int n )  {  int table[n+1];  memset(table, 0, sizeof(table));    // Base case (If given value is 0)  table[0] = 1;  for(int i=0; i<m; i++)  for(int j=S[i]; j<=n; j++)  table[j] += table[j-S[i]];    return table[n];  } Rod Cutting Problem: LL cutRod(LL price[], LL n)  {  LL val[n+1];  val[0] = 0;  LL i, j;    // Build the table val[] in bottom up manner and return the last entry  // from the table  for (i = 1; i<=n; i++)  {  LL max\_val = INT\_MIN;  for (j = 0; j < i; j++)  max\_val = max(max\_val, price[j] + val[i-j-1]);  val[i] = max\_val;  }    return val[n];} Sum Of Subset: bool isSubsetSum(LL set[], LL n, LL sum)  {  bool subset[n+1][sum+1];  for (LL i = 0; i <= n; i++)  subset[i][0] = true;  for (LL i = 1; i <= sum; i++)  subset[0][i] = false;  for (LL i = 1; i <= n; i++)  {  for (LL j = 1; j <= sum; j++)  {  if(j<set[i-1])  subset[i][j] = subset[i-1][j];  if (j >= set[i-1])  subset[i][j] = subset[i-1][j] ||  subset[i - 1][j-set[i-1]];  }  }  return subset[n][sum];  } Catalan numbers: **1, 1, 2, 5, 14, 42, 132, 429, 1430,........**  C(n) =(1/(n+1)) \* choose(2n, n);  C(n+1) = Summation(i = 0 to n) [C(i) \* C(n-i)] 0/1 Knapsack: LL knapSack(LL W, LL wt[], LL val[], LL n)  {  LL i, w;  LL K[n+1][W+1];  for (i = 0; i <= n; i++)  {  for (w = 0; w <= W; w++)  {  if (i==0 || w==0)  K[i][w] = 0;  else if (wt[i-1] <= w)  K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);  else  K[i][w] = K[i-1][w];  }  }  return K[n][W];  } Cap Assignment (bit-mask): long long int countWaysUtil(int mask, int i)  {  if (mask == allmask) return 1;  if (i > 100) return 0;  if (dp[mask][i] != -1) return dp[mask][i];  long long int ways = countWaysUtil(mask, i+1);  int size = capList[i].size();  for (int j = 0; j < size; j++)  {  if (mask & (1 << capList[i][j])) continue;  else ways += countWaysUtil(mask | (1 << capList[i][j]), i+1);  ways %= MOD;  }  return dp[mask][i] = ways;  } |