

|  |  |
| --- | --- |
|  | Strongly Connected Components (Kasuraja’s Algo): |
| **TRIE\_NP\_HARD (SLRTCE)** | 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(); |
| **TEAM NOTEBOOK** | Stack.pop();  if (visited[v] == false) |
| **(ICPC KOLKATA)** | {  gr.DFSUtil(v, visited); |
|  | cout << endl; |
|  | } |
|  | }} |

|  |  |  |
| --- | --- | --- |
| Bit Manipulation: | 4 | Articulation Point (cut-vertices):  void APUtil(LL u, bool visited[], LL disc[],  LL low[], LL parent[], bool ap[])  {  static LL time = 0; LL children = 0; visited[u] = true;  disc[u] = low[u] = ++time; list<LL>::iterator i;  for (i = adj[u].begin(); i != adj[u].end(); ++i)  {  LL v = \*i;  if (!visited[v])  {  children++; parent[v] = u;  APUtil(v, visited, disc, low, parent, ap); if (parent[u] == NIL && children > 1)  ap[u] = true;  if (parent[u] != NIL && low[v] >= disc[u]) ap[u] = true;  }.  else if (v != parent[u])  low[u] = min(low[u], disc[v]);  }  }  void AP()  {  bool \*visited = new bool[V]; LL \*disc = new LL[V];  LL \*low = new LL[V]; LL \*parent = new LL[V]; bool \*ap = new bool[V]; |
| Techniques: | 4 |
| STL DS: | 5 |
| **STL Algorithms:** | **5** |
| **Number Theory:** | **6** |
| Probability: | 8 |
| Extended Euclid’s Algorithm: | 8 |
| Segmented Sieve for primes | 9 |
| Modular power | 9 |
| Matrix Exponentiation | 10 |
| Euler’s totient: | 10 |
| Largest power of p that divides n! | 11 |
| nCr (with lucas Theorem): | 11 |
| Chinese Remainder Theorem | 12 |
| Wilson’s theorem | 12 |
| Inclusion-Exclusion: | 12 |
| Number of solutions to a linear eqn: | 13 |
| Sum of GP: | 13 |
| Ternary Search (max of unimodal function): | 14 |
| **Data Structures:** | **14** |
| Iterative trie: | 14 |
| Iterative segment tree: | 15 |
| Lazy Segment tree: | 16 |
| Policy based DS: | 17 |
| Union-Find: | 18 |
| **Graph Theory** | **18** |
| Dijkstra’s Algorithm: | 18 |
| Floyd Warshall(All pair) | 19 |

|  |  |  |
| --- | --- | --- |
| Bellman-Ford(for negative edges): | 20 | for (LL i = 0; i < V; i++)  {  parent[i] = NIL; visited[i] = false; ap[i] = false;  }  for (LL i = 0; i < V; i++) if (visited[i] == false)  APUtil(i, visited, disc, low, parent, ap); for (LL i = 0; i < V; i++)  if (ap[i] == true) cout << i << " ";  }  Bridges:  Replace condition for articulation point with if (low[v] > disc[u])  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, |
| Prim’s Algorithm for MST | 20 |
| LCA: | 21 |
| Topological Sort: | 22 |
| Strongly Connected Components (Kasuraja’s Algo): | 1 |
| Articulation Point (cut-vertices): | 2 |
| Bridges: | 3 |
| Euler path/circuit: | 3 |
| Hierholzer’s algorithm for directed graph: | 4 |
| Ford-Fulkerson max flow Algorithm: | 5 |
| Maximum Bipartite Matching: | 6 |
| **Geometry:** | **8** |
| Orientation: | 9 |
| Line intersection: | 9 |
| Circle intersection area: | 10 |
| Convex Hull: | 10 |
| Point in a polygon: | 12 |
| **Game Theory:** | **13** |
| **Pattern Matching:** | **13** |
| Suffix Arrays: | 13 |
| KMP Algorithm: | 15 |
| **Standard DP** | **17** |
| LCS: | 17 |
| Max contiguous subarray sum (Kadane’s Algo): | 18 |
| LIS in nlogn: | 19 |
| Coin Change Problem: | 20 |
| Rod Cutting Problem: | 20 |
| Sum Of Subset: | 21 |

[Catalan numbers: 21](#_TOC_250002)

[0/1 Knapsack: 22](#_TOC_250001)

[Egg Drop Problem: 22](#_TOC_250000)

Cap Assignment (bit-mask): 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
5. Meet in the middle (Solve left half, Solve right half, combine)
6. Greedy
7. DP
8. Analyse complexity carefully
9. Reduce the problem to some standard problem
10. Add m when doing modular arithmetic.
11. Carefully analyse reasoning behind adding small details in the Q.
12. Use exponential search in case of unbounded search.

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--) |
| STL DS:  stack<type> name empty(),size(),pop(),top(),push(x) | {  cout << circuit[i]; if (i)  cout<<" -> "; |
| queue<type> name empty(),size(),pop(),front(),back(),push(x) | }  } |
| priority\_queue <type> name | Bipartite graph: Coloring possible with 2 colors. |
| empty(),size(),pop(),top(),push(x) | Ford-Fulkerson max flow Algorithm: |
| deque<type> name | bool bfs(int rGraph[V][V], int s, int t, int parent[]) |
| pop\_front(),pop\_back(),push\_front(),push\_back(),size(),at(index),front() | { |
| ,back() | bool visited[V]; |
|  | memset(visited, 0, sizeof(visited)); |
| set/multiset/map/multimap<type>name | queue <int> q; |
| begin(),end(),size(),empty(),insert(val),erase(itr or val),find(val), | q.push(s); |
| lower\_bound(val),upper\_bound(val) | visited[s] = true; |
| (lower bound includes val, upper bound does not) | parent[s] = -1; |
| pair<type,type> name (first and second) | while (!q.empty()) |
|  | { |
| STL Algorithms: | int u = q.front(); q.pop(); |
| 1.sort(first\_iterator, last\_iterator) – To sort the given vector. | for (int v=0; v<V; v++) |
|  | { |
| 2. reverse(first\_iterator, last\_iterator) – To reverse a vector. | if (visited[v]==false && rGraph[u][v] > 0) |
|  | { |
| 3. \*max\_element (first\_iterator, last\_iterator) – To find the maximum | q.push(v); |
| element of a vector. | parent[v] = u; |
|  | visited[v] = true; |
| 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 | return (visited[t] == true); |
| summation of vector elements | } |

|  |  |
| --- | --- |
|  | int fordFulkerson(int graph[V][V], int s, int t) |
| 6. binary\_search(first\_iterator, last\_iterator, x) – Tests whether x exists in | { |
| sorted vector or not. | int u, v; |
|  | int rGraph[V][V]; |
| 7.lower\_bound(first\_iterator, last\_iterator, x) – returns an iterator pointing to | for (u = 0; u < V; u++) |
| the first element in the range [first,last) which has a value not less than ‘x’. | for (v = 0; v < V; v++) |
|  | rGraph[u][v] = graph[u][v]; |
| 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’. | int parent[V]; |
| 9.count(first\_iterator, last\_iterator,x) – To count the occurrences of x in | int max\_flow = 0; |
| vector. | while (bfs(rGraph, s, t, parent)) |
|  | { |
| 10.next\_permutation(first\_iterator, last\_iterator) – This modified the vector | int path\_flow = INT\_MAX; |
| to its next permutation. | for (v=t; v!=s; v=parent[v]) |
|  | { |
| 11.prev\_permutation(first\_iterator, last\_iterator) – This modified the vector | u = parent[v]; |
| to its previous permutation | path\_flow = min(path\_flow, rGraph[u][v]); |
|  | } |
| 12. random\_shuffle(arr.begin(), arr.end()); | for (v=t; v != s; v=parent[v]) |
|  | { |
| 13. ios\_base::sync\_with\_stdio(false); | u = parent[v]; |
| cin.tie(NULL); | rGraph[u][v] -= path\_flow; |
|  | rGraph[v][u] += path\_flow; |
| Number Theory: | }  max\_flow += path\_flow; |
|  | } |
| 1. To calculate sum of factors of a number, we can find the number of prime | return max\_flow; |
| 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 | Maximum Bipartite Matching: |
| primes. | bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[]) |
| 3. For rootn prime method, check for 2, 3 then: | {  // Try every job one by one |

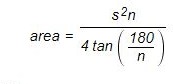
|  |  |
| --- | --- |
| for (i=5; i\*i<=n; i=i+6) n%i and n%(i+2) | for (int v = 0; v < N; v++) |
|  | { |
| 4. Number of divisors will be prime only if N=p^x where p is prime. | // If applicant u is interested in job v and v is |
|  | // not visited |
| 5. Kth prime factor= store smallest factor in seive and repeatedly divide with | if (bpGraph[u][v] && !seen[v]) |
| it to get the answer. | { |
|  | seen[v] = true; // Mark v as visited |
| 6. fib(n+m)=fib(n)fib(m+1)+fib(n-1)fib(m) | // If job 'v' is not assigned to an applicant OR |
|  | // previously assigned applicant for job v (which is matchR[v]) |
| 7. A number is Fibonacci if and only if one or both of (5\*n2 + 4) or (5\*n2 – | // has an alternate job available. |
| 4) is a perfect square | // Since v is marked as visited in the above line, matchR[v] |
|  | // in the following recursive call will not get job 'v' again |
| 8. every positive Every positive integer can be written uniquely as a sum of | if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR)) |
| distinct non-neighbouring Fibonacci numbers. | { |
|  | matchR[v] = u; |
| 9. Matrix multiplication | return true; |
| mul[i][j] += a[i][k]\*b[k][j]; | } |
|  | } |
| 10. Root n under mod p exists only if | } |
| n^((p-1)/2) % p = 1 | return false; |
|  | } |
| 11.divisibility by 4: last 2 digits divisible by 4 | int maxBPM(bool bpGraph[M][N]) |
| 12.divisibility by 8: last 3 digits divisible by 8 | {  // The value of matchR[i] is the applicant number |
| 13. Divisibility by 3,9: sum of digs divisible by 3,9 | // assigned to job i  int matchR[N]; |
| 14. Divisibility by 11: alternate (+ve,-ve) digit sum is divisible by 11 | memset(matchR, -1, sizeof(matchR)); |
| 15. Divisibility by 12: divisible by 3 and 4 | int result = 0; // Count of jobs assigned to applicants for (int u = 0; u < M; u++) |
| 16. Divisibility by 13: alternating sum in blocks of 3 (L to R) div 13 | {  // Mark all jobs as not seen for next applicant. |
| 17. Integral solution of ax+by=c exists if gcd(a,b) divides c | bool seen[N];  memset(seen, 0, sizeof(seen)); |

Probability:

// Find if the applicant 'u' can get a job if (bpm(bpGraph, u, seen, matchR))

result++;

}



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

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. }

1. LL x1, y1;
2. 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.

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))

1. 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

1. Polygon area (vertex cordinates):

| x1 · y2 + x2 · y3 + ... + xn · y1 − y1 · x2 − y2 · x3 − ... − yn · x1 | / 2

Segmented Sieve for primes

* 1. void segsieve(LL l,LL r) 2. {

1. LL limit = [floor](http://www.opengroup.org/onlinepubs/009695399/functions/floor.html)([sqrt](http://www.opengroup.org/onlinepubs/009695399/functions/sqrt.html)(r))+1;
2. vector<LL> prime;
3. sieve(limit, prime);
4. limit=r-l+1;
5. bool mark[limit+1];
6. [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++)

10. {

1. int loLim = [floor](http://www.opengroup.org/onlinepubs/009695399/functions/floor.html)(l/prime[i]) \* prime[i];
2. if (loLim < l)
3. loLim += prime[i]; 14.
4. for (int j=loLim; j<=r; j+=prime[i])
5. mark[j-l] = false;

17. }

18. }

Modular power

1. LL Mpow(LL x, unsigned LL y, LL m) 2. {

1. LL res = 1;
2. x = x % m;
3. 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;}

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;}

|  |  |
| --- | --- |
| Matrix Exponentiation | Circle intersection area: |
| LL power(LL F[3][3], LL n) | int areaOfIntersection(x0, y0, r0, x1, y1, r1){ |
| { | var rr0 = r0\*r0; |
| LL M[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}}; | var rr1 = r1\*r1; |
| if (n==1) | var c = Math.sqrt((x1-x0)\*(x1- x0) +(y1-y0)\*(y1- y0)); |
| return F[0][0] + F[0][1]; | var phi =(Math.acos((rr0+(c\*c)-rr1) /(2\*r0\*c)))\*2; |
| power(F, n/2); | var theta =(Math.acos((rr1+(c\*c)-rr0) /(2\*r1\*c)))\*2; |
| multiply(F, F); | var area1 = 0.5\*theta\*rr1 - 0.5\*rr1\*Math.sin(theta); |
| if (n%2 != 0) | var area2 = 0.5\*phi\*rr0 - 0.5\*rr0\*Math.sin(phi); |
| multiply(F, M); | return area1 + area2; |
| return F[0][0] + F[0][1] ; | } |
| } |  |
| LL findNthTerm(LL n) | Convex Hull: |
| {  LL F[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}} ;  return power(F, n-2);  } | Point nextToTop(stack<Point> &S)  {  Point p = S.top();  S.pop(); |
| Euler’s totient: | Point res = S.top();  S.push(p); |
| Number of integers coprime to n less than n | return res; |
| LL phi(LL n) | } |
| { |  |
| LL result = n; | int distSq(Point p1, Point p2) |
| for (LL p=2; p\*p<=n; ++p) | { |
| { | return (p1.x - p2.x)\*(p1.x - p2.x) + |
| if (n % p == 0) | (p1.y - p2.y)\*(p1.y - p2.y); |
| { | } |
| while (n % p == 0) |  |
| n /= p; | int compare(const void \*vp1, const void \*vp2) |
| result -= result / p; | { |
| } | Point \*p1 = (Point \*)vp1; |
| } | Point \*p2 = (Point \*)vp2; |

if (n > 1)

result -= result / n; return result;

}

int o = orientation(p0, \*p1, \*p2); if (o == 0)

return (distSq(p0, \*p2) >= distSq(p0, \*p1))? -1 : 1; return (o == 2)? -1: 1;

}

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. }

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++)

{

|  |  |
| --- | --- |
| 13. LL ncrpl(LL n,LL r, LL p) | while (orientation(nextToTop(S), S.top(), points[i]) != 2) |
| 14. { | S.pop(); |
| 15. if (r==0) | S.push(points[i]); |
| 16. return 1; | } |
| 17. int ni = n%p, ri = r%p; | while (!S.empty()) |
| 18. return (ncrpl(n/p, r/p, p) \* | { |
| 19. ncrp(ni, ri, p)) % p; | Point p = S.top(); |
| 20. } | cout << "(" << p.x << ", " << p.y <<")" << endl; |
|  | S.pop(); |
| Chinese Remainder Theorem | }  } |
| 1. LL crt(LL num[], LL rem[], LL k)  2. { | Point in a polygon: |
| 3. LL prod = 1; |  |
| 4. for (int i = 0; i < k; i++) | bool isInside(Point polygon[], int n, Point p) |
| 5. prod \*= num[i]; | { |
| 6. LL result = 0; | if (n < 3) return false; |
| 7. for (int i = 0; i < k; i++) | Point extreme = {INF, p.y}; |
| 8. { | int count = 0, i = 0; |
| 9. LL pp = prod / num[i]; | do |
| 10. LL inv,y; | { |
| 11. gcde(pp,num[i],&inv,&y); | int next = (i+1)%n; |
| 12. result += rem[i] \* inv \* pp; | if (doIntersect(polygon[i], polygon[next], p, extreme)) |
| 13. } | { |
| 14. return result % prod; | if (orientation(polygon[i], p, polygon[next]) == 0) |
| 15. } | return onSegment(polygon[i], p, polygon[next]); |
| For combining wrt a large number, use it 2 numbers at a time. |  |
|  | count++; |
| Wilson’s theorem  ((p-1)!)%p=-1 | }  i = next;  } while (i != 0); |
| Inclusion-Exclusion: | return count&1; // Same as (count%2 == 1)  } |
| (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;

}

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';

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:

Iterative trie:

int trie[MAX\_N \* 30][3], nxt; void trie\_init(int n) {

int nn = (n+2)\*30; for(int i=0; i<nn; i++)

trie[i][0] = trie[i][1] = trie[i][2] = -1; nxt = 1;

}

void trie\_insert(int v, int x) { int cur = 0;

for(int i=29; i>=0; i--) { int bit = v>>i & 1; if(trie[cur][bit]==-1)

trie[cur][bit] = nxt++; cur = trie[cur][bit];

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];

|  |  |
| --- | --- |
| trie[cur][2] = max(trie[cur][2], x); | for (int i = 0; i < n; i++) |
| } | suffixArr[i] = suffixes[i].index; |
| } | return suffixArr; |
|  | } |
| int trie\_getmax(int v, int m) { |  |
| int cur = 0, mx = -1; | void search(char \*pat, char \*txt, int \*suffArr, int n) |
| for(int i=29; i>=0; i--) { | { |
| int bit = v>>i & 1; | int m = strlen(pat); |
| if(m>>i & 1) | int l = 0, r = n-1; |
| cur = trie[cur][!bit]; | while (l <= r) |
| else { | { |
| int lt = trie[cur][!bit]; | int mid = l + (r - l)/2; |
| if(lt!=-1) mx = max(mx, trie[lt][2]); | int res = strncmp(pat, txt+suffArr[mid], m); |
| cur = trie[cur][bit]; | if (res == 0) |
| } | { |
| if(cur==-1) break; | cout << "Pattern found at index " << suffArr[mid]; |
| } | return; |
| if(cur!=-1) mx = max(mx, trie[cur][2]); | } |
| return mx; | if (res < 0) r = mid - 1; |
| } | else l = mid + 1; |
|  | } |
| Iterative segment tree:  void build() { | cout << "Pattern not found";  } |
| for (LL i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];} | KMP Algorithm: |
| void modify(LL p, LL value) { // set value at position p |  |
| for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];} | void KMPSearch(char \*pat, char \*txt) |
|  | { |
| LL query(LL l, LL r) { // sum on LLerval [l, r) | int M = strlen(pat); |
| LL res = 0; | int N = strlen(txt); |
| for (l += n, r += n; l < r; l >>= 1, r >>= 1) { | int lps[M]; |
| if (l&1) res += t[l++]; | computeLPSArray(pat, M, lps); |
| if (r&1) res += t[--r]; |  |
| } | int i = 0; // index for txt[] |
| return res; | int j = 0; // index for pat[] |
| } |  |

Lazy Segment tree:

LL lconstruct(LL \*a,LL \*st,LL ss,LL se,LL si)

{

if(ss==se)

{

st[si]=a[ss]; return st[si];

}

LL mid=ss+(se-ss)/2; st[si]=(lconstruct(a,st,ss,mid,si\*2+1)+lconstruct(a,st,mid+1,se,si\*2+2)); return st[si];

}

LL lgs(LL \*st,LL l,LL r,LL ss,LL se,LL si,LL \*lazy)

{

if(lazy[si])

//same as update if(ss>r||se<l||ss>se) return 0; if(l<=ss&&r>=se)

{

return st[si];

}

LL mid=ss+(se-ss)/2;

return (lgs(st,l,r,ss,mid,si\*2+1,lazy)+lgs(st,l,r,mid+1,se,si\*2+2,lazy));

}

void lupdate(LL \*st,LL ss,LL se,LL ql,LL qr,LL diff,LL si,LL \*lazy)

{

if(lazy[si])

{

st[si]=(st[si]+(se-ss+1)\*lazy[si]);

while (i < N)

{

if (pat[j] == txt[i])

{

j++; i++;

}

if (j == M)

{

printf("Found pattern at index %d n", i-j); j = lps[j-1];

}

else if (i < N && pat[j] != txt[i])

{

if (j != 0)

j = lps[j-1]; else

i = i+1;

}

}

}

void computeLPSArray(char \*pat, int M, int \*lps)

{

int len = 0;

lps[0] = 0; // lps[0] is always 0 int i = 1;

while (i < M)

{

if (pat[i] == pat[len])

{

len++; lps[i] = len; i++;

}

if(ss!=se)

{

lazy[si\*2+1]=(lazy[si\*2+1]+lazy[si]); lazy[si\*2+2]=(lazy[si\*2+2]+lazy[si]);

}

lazy[si]=0;

}

else // (pat[i] != pat[len])

{

if (len != 0)

{

len = lps[len-1];

}

else // if (len == 0)

if(ss>se||qr<ss||ql>se) return; if(ss>=ql&&se<=qr)

{

st[si]=(st[si]+(se-ss+1)\*diff); if(ss!=se)

{

lazy[si\*2+1]=(lazy[si\*2+1]+diff); lazy[si\*2+2]=(lazy[si\*2+2]+diff);

}

return;

}

if(ss!=se)

{

LL mid=ss+(se-ss)/2; lupdate(st,ss,mid,ql,qr,diff,si\*2+1,lazy); lupdate(st,mid+1,se,ql,qr,diff,si\*2+2,lazy);

}

st[si]=(st[2\*si+1]+st[2\*si+2]);

}

Policy based DS:

#include <ext/pb\_ds/assoc\_container.hpp> #include <ext/pb\_ds/tree\_policy.hpp> using namespace gnu\_pbds;

typedef tree<int, null\_type, less<int>, rb\_tree\_tag, tree\_order\_statistics\_node\_update> pbds;

insert(val),erase(),order\_of\_key(),find\_by\_order()

{

lps[i] = 0; i++;

}

}

}

}

Standard DP

LCS:

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

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++;

}

}

Graph Theory

Dijkstra’s Algorithm:

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;

}

void Dijkstra(LL src,LL V)

{

set< pair<LL, LL> > setds; vector<LL> dist(V, INF); setds.insert(make\_pair(0, src)); dist[src] = 0;

while (!setds.empty())

{

pair<int, int> tmp = \*(setds.begin()); setds.erase(setds.begin());

int u = tmp.second;

vector< pair<int, int> >::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

{

int v = (\*i).first;

int weight = (\*i).second;

if (dist[v] > dist[u] + weight)

{

if (dist[v] != INF) setds.erase(setds.find(make\_pair(dist[v], v)));

dist[v] = dist[u] + weight; setds.insert(make\_pair(dist[v], v));

}

}

}

}

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];

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;

}

|  |  |
| --- | --- |
| Bellman-Ford(for negative edges): | Coin Change Problem: |
| void BellmanFord(struct Graph\* graph, LL src) | int count( int S[], int m, int n ) |
| { | { |
| LL V = graph->V; | int table[n+1]; |
| LL E = graph->E; | memset(table, 0, sizeof(table)); |
| LL dist[V]; |  |
| for (LL i = 0; i < V; i++) | // Base case (If given value is 0) |
| dist[i] = INT\_MAX; | table[0] = 1; |
| dist[src] = 0; | for(int i=0; i<m; i++) |
| for (LL i = 1; i <= V-1; i++) | for(int j=S[i]; j<=n; j++) |
| { | table[j] += table[j-S[i]]; |
| for (LL j = 0; j < E; j++) |  |
| { | return table[n]; |
| 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 | Rod Cutting Problem:  LL cutRod(LL price[], LL n)  {  LL val[n+1]; val[0] = 0; LL i, j; |
| Prim’s Algorithm for MST | // Build the table val[] in bottom up manner and return the last entry |
|  | // from the table |
| void primMST() | for (i = 1; i<=n; i++) |
| { | { |
| priority\_queue<pair<LL,LL>,greater<pair<LL,LL>>> pq; | LL max\_val = INT\_MIN; |
| LL src = 0; | for (j = 0; j < i; j++) |
| vector<LL> key(V, INF); | max\_val = max(max\_val, price[j] + val[i-j-1]); |
| vector<LL> parent(V, -1); | val[i] = max\_val; |
| vector<bool> inMST(V, false); | } |
| pq.push(make\_pair(0, src)); |  |
| key[src] = 0; | return val[n];} |

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:

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]];

}

}

LL par[MAXN][MAXLOG]; // initially all -1 void dfs(LL v,LL p = -1){

par[v][0] = p; if(p + 1)

h[v] = h[p] + 1;

for(LL i = 1;i < MAXLOG;i ++)

if(par[v][i-1] + 1)

par[v][i] = par[par[v][i-1]][i-1]; for(auto u : adj[v]) if(p - u)

dfs(u,v);

}

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)]

LL LCA(LL v,LL u){

if(h[v] < h[u])

swap(v,u);

for(LL i = MAXLOG - 1;i >= 0;i --)

if(par[v][i] + 1 and h[par[v][i]] >= h[u]) v = par[v][i];

// now h[v] = h[u] if(v == u)

return v;

for(LL i = MAXLOG - 1;i >= 0;i --)

if(par[v][i] - par[u][i])

v = par[v][i], u = par[u][i]; return par[v][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);

# 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];

}

# Egg Drop Problem:

LL eggDrop(LL n, LL k)

{

LL eggFloor[n+1][k+1]; LL res;

LL i, j, x;

for (i = 1; i <= n; i++)

{

eggFloor[i][1] = 1;

eggFloor[i][0] = 0;

}

// We always need j trials for one egg and j floors. for (j = 1; j <= k; j++)

eggFloor[1][j] = j;

|  |  |  |
| --- | --- | --- |
| } | while (Stack.empty() == false)  {  cout << Stack.top() << " "; Stack.pop();  } | for (i = 2; i <= n; i++) |
| { |
| for (j = 2; j <= k; j++) |
| { |
| eggFloor[i][j] = INT\_MAX; |
| for (x = 1; x <= j; x++) |
| { |
| res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]); |
| if (res < eggFloor[i][j]) |
| eggFloor[i][j] = res; |
| } |
| } |
| } |
| return eggFloor[n][k]; |
| } |
| 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; |
| } |