L^3

Team Notebook

09/11/2018

Contents

1. Code Templates

```
#include <bits/stdc++.h> ----------------------------------//84
typedef long long ll; --------------------------------------//62
typedef unsigned long long ull; ----------------------------//ce
typedef std::pair<int, int> ii; ----------------------------//6f
typedef std::vector<int> vi; -------------------------------//96
typedef std::vector<vi> vvi; -------------------------------//b2
typedef std::vector<ii> vii; -------------------------------//56
const int INF = ~(1<<31); ----------------------------------//39
const ll LINF = (1LL << 60); -------------------------------//67
const double EPS = 1e-9; -----------------------------------//75
const double pi = acos(-1); --------------------------------//47
```

2. Data Structures

2.1. Fenwick Tree.

2.1.1. *Point-updates.*

2.1.2. *Range-updates.*

2.2. Mergesort Tree.

2.3. Segment Tree.

2.3.1. *Recursive (Point-update).*

```
struct segtree { -------------------------------------------//64
- int i, j, val; -------------------------------------------//06
- segtree *l, *r; ------------------------------------------//9d
- segtree(vi &ar, int _i, int _j) : i(_i), j(_j) { --------//88
--- if (i == j) { ------------------------------------------//f1
----- val = ar[i]; -----------------------------------------//ea
----- l = r = NULL; ----------------------------------------//9f
--- } else { -----------------------------------------------//e6
----- int k = (i+j) >> 1; ----------------------------------//50
----- l = new segtree(ar, i, k); ---------------------------//57
----- r = new segtree(ar, k+1, j); -------------------------//d0
----- val = l->val + r->val;  }  } -------------------------//ff
- void update(int _i, int _val) { --------------------------//28
--- if (i == _i and _i == j) { -----------------------------//b3
----- val = _val; ------------------------------------------//62
--- } else if (_i < i or j < _i) { -------------------------//0b
----- // do nothing ----------------------------------------//b7
--- } else { -----------------------------------------------//b9
----- l->update(_i, _val); ---------------------------------//92
----- r->update(_i, _val); ---------------------------------//c6
----- val = l->val + r->val;  }  } -------------------------//af
- int sum(int _i, int _j) { --------------------------------//fe
--- if (_i <= i and j <= _j) { -----------------------------//06
----- return val; ------------------------------------------//39
--- } else if (_j < i or j < _i) { -------------------------//40
----- return 0; --------------------------------------------//5a
--- } else { -----------------------------------------------//2f
----- return l->sum(_i, _j) + r->sum(_i, _j);  }  }  }; ---//04
```

2.3.2. *Iterative (Point-update and operation can be non-commutative).*

```
struct segtree { -------------------------------------------//64
- int n; ---------------------------------------------------//91
- int *vals; -----------------------------------------------//00
- segtree(int *ar, int n) { --------------------------------//cf
--- this->n = n; -------------------------------------------//db
--- vals = new int[2*n]; -----------------------------------//a1
--- for (int i = 0; i < n; ++i) ----------------------------//37
----- vals[n+i] = ar[i]; -----------------------------------//0b
--- for (int i = n-1; i > 0; --i) --------------------------//b3
----- vals[i] = vals[i<<1] + vals[i<<1|1];  } -------------//27
- void update(int i, int v) { ------------------------------//f7
--- for (vals[i += n] = v; i > 1; i >>= 1) ----------------//be
----- t[i>>1] = t[i] + t[i^1];  } -------------------------//49
- int query(int l, int r) { --------------------------------//0d
--- int res = 0; -------------------------------------------//8c
--- for (l += n, r += n; l < r; l >>= 1, r >>= 1) { -------//9b
----- if (l&1)  res += vals[l++]; --------------------------//e7
----- if (r&1)  res += vals[--r];  } -----------------------//b3
--- return res;  }  } --------------------------------------//80
```

2.3.3. *Lazy Propagation (Range-update).*

```
struct segtree { -------------------------------------------//64
- int i, j, val, temp_val = 0; -----------------------------//47
- segtree *l, *r; ------------------------------------------//cd
- segtree(vi &ar, int _i, int _j) : i(_i), j(_j) { --------//e0
--- if (i == j) { ------------------------------------------//74
----- val = ar[i]; -----------------------------------------//c8
----- l = r = NULL; ----------------------------------------//60
--- } else { -----------------------------------------------//e8
----- int k = (i + j) >> 1; --------------------------------//61
----- l = new segtree(ar, i, k); ---------------------------//75
----- r = new segtree(ar, k+1, j); -------------------------//a7
----- val = l->val + r->val;  }  } -------------------------//c7
- void visit() { -------------------------------------------//90
--- if (temp_val) { ----------------------------------------//e5
----- val += (j-i+1) * temp_val; ---------------------------//84
```

```
----- if (l) { --------------------------------------------//70
------- l->temp_val += temp_val; --------------------------//94
------- r->temp_val += temp_val;  } -----------------------//b2
----- temp_val = 0;  }  } ---------------------------------//46
- void increase(int _i, int _j, int _inc) { ---------------//ef
--- visit(); ----------------------------------------------//af
--- if (_i <= i && j <= _j) { -----------------------------//0a
----- temp_val += _inc; -----------------------------------//02
----- visit(); --------------------------------------------//b7
--- } else if (_j < i or j < _i) { ------------------------//4e
----- // do nothing ---------------------------------------//2f
--- } else { ----------------------------------------------//c5
----- l->increase(_i, _j, _inc); --------------------------//fc
----- r->increase(_i, _j, _inc); --------------------------//b0
----- val = l->val + r->val;  }  } ------------------------//81
- int sum(int _i, int _j) { -------------------------------//ae
--- visit(); ----------------------------------------------//86
--- if (_i <= i and j <= _j) { ----------------------------//1b
----- return val; -----------------------------------------//73
--- } else if (_j < i || j < _i) { ------------------------//96
----- return 0; -------------------------------------------//0d
--- } else { ----------------------------------------------//8c
----- return l->sum(_i, _j) + r->sum(_i, _j);  }  }  }; ---//f6
```

2.3.4. *Persistent (Point-update).*

```
struct node {  int l, r, lid, rid, val;  }; ---------------//63
struct segtree { ------------------------------------------//c9
- node *nodes; --------------------------------------------//10
- int n, node_cnt = 0; ------------------------------------//77
- segtree(int n) { ----------------------------------------//96
--- this->n = n; ------------------------------------------//83
--- nodes = new node[2*n];  } -----------------------------//f9
- int build (vi &ar, int l, int r) { ----------------------//21
--- if (l > r)  return -1; --------------------------------//3c
--- int id = node_cnt++; ----------------------------------//2c
--- nodes[id].l = l; --------------------------------------//06
--- nodes[id].r = r; --------------------------------------//89
--- if (l == r) { -----------------------------------------//51
----- nodes[id].lid = -1; ---------------------------------//fb
----- nodes[id].rid = -1; ---------------------------------//20
----- nodes[id].val = ar[l]; ------------------------------//0c
--- } else { ----------------------------------------------//ed
----- int m = (l + r) / 2; --------------------------------//63
----- nodes[id].lid = build(l, m); ------------------------//e7
----- nodes[id].rid = build(m+1, r); ----------------------//ff
----- nodes[id].val = nodes[nodes[id].lid].val + ----------//95
-------------------- nodes[nodes[id].rid].val;  } ---------//a3
--- return id;  } -----------------------------------------//31
- int update(int idx, int delta, int id) { ----------------//a0
--- if (id == -1) -----------------------------------------//23
----- return -1; ------------------------------------------//66
--- if (idx < nodes[id].l or nodes[id].r < idx) -----------//fb
----- return id; ------------------------------------------//6c
--- int nid = node_cnt++; ---------------------------------//8e
--- nodes[nid].l = nodes[id].l; ---------------------------//af
--- nodes[nid].r = nodes[id].r; ---------------------------//cc
--- nodes[nid].lid = update(idx, v, nodes[id].lid); -------//f3
--- nodes[nid].rid = update(idx, v, nodes[id].rid); -------//ef
--- nodes[nid].val = nodes[id].val + delta; ---------------//d1
--- return nid;  } ----------------------------------------//39
- int query(int id, int l, int r) { -----------------------//5c
--- if (r < nodes[id].l or nodes[id].r < l) ---------------//49
----- return 0; -------------------------------------------//ff
--- if (l <= nodes[id].l and nodes[id].r <= r) ------------//30
----- return nodes[id].val; -------------------------------//ef
--- return query(nodes[id].l, l, r) + ---------------------//97
---------- query(nodes[id].r, l, r);  }  }; ---------------//37
```

2.4. Sparse Table.

2.5. Sqrt Decomposition.

2.6. Treap.

2.6.1. *Explicit.*

2.6.2. *Implicit.*

2.6.3. *Persistent.*

2.7. Union Find.

```
struct union_find { ---------------------------------------//42
- vi p; union_find(int n) : p(n, -1) { } ------------------//28
- int find(int x) { return p[x] < 0 ? x : p[x] = find(p[x]); }
- bool unite(int x, int y) { ------------------------------//6c
--- int xp = find(x), yp = find(y); -----------------------//64
--- if (xp == yp) return false; ---------------------------//0b
--- if (p[xp] > p[yp]) swap(xp,yp); -----------------------//78
--- p[xp] += p[yp], p[yp] = xp; ---------------------------//88
--- return true; } ----------------------------------------//1f
- int size(int x) { return -p[find(x)]; } }; --------------//b9
```

3. Graphs

Using adjacency list:

```
struct graph { --------------------------------------------//32
- int n; --------------------------------------------------//d4
- vii *adj; -----------------------------------------------//7f
- int *dist; ----------------------------------------------//82
- graph(int n) { ------------------------------------------//42
--- this->n = n; ------------------------------------------//8a
--- adj = new vii[n]; -------------------------------------//0c
--- dist = new int[n]; } ----------------------------------//96
- void add_edge(int u, int v, int w) { --------------------//f8
--- adj[u].push_back({v, w}); -----------------------------//b9
--- /*adj[v].push_back({u, w});*/  } }; --------------------//2a
```

Using adjacency matrix:

```
struct graph { --------------------------------------------//32
- int n; --------------------------------------------------//d4
- int **mat; ----------------------------------------------//44
- graph(int n) { ------------------------------------------//9c
--- this->n = n; ------------------------------------------//67
--- mat = new int*[n]; ------------------------------------//2a
--- for (int i = 0; i < n; ++i) { -------------------------//ae
----- mat[i] = new int[n]; --------------------------------//f3
----- for (int j = 0; j < n; ++j) -------------------------//3a
------- mat[i][j] = INF; ----------------------------------//78
----- mat[i][i] = 0;  }  } --------------------------------//96
- void add_edge(int u, int v, int w) { --------------------//36
--- mat[u][v] = std::min(mat[u][v], w); -------------------//5c
- /*mat[v][u] = std::min(mat[v][u], w);*/  } }; -----------//6b
```

Using edge list:

```
struct edge { ---------------------------------------------//c7
- int u, v, w; --------------------------------------------//03
- edge(int u, int v, int w) : u(u), v(v), w(w) {} ---------//7f
- const bool operator <(const edge &other) const { --------//67
--- return w < other.w;  }  }; ----------------------------//c9
struct graph { --------------------------------------------//d4
- int n; --------------------------------------------------//29
- std::vector<edge> edges; --------------------------------//e6
- graph(int n) : n(n) {} ----------------------------------//1c
- void add_edge(int u, int v, int w) { --------------------//77
--- edges.push_back(edge(u, v, w));  } }; -----------------//8e
```

3.1. Single-Source Shortest Paths.

3.1.1. *Dijkstra.*

```
#include "graph_template_adjlist.cpp" ---------------------//76
void dijkstra(int s, int n, int *dist, vii *adj) { --------//ad
- for (int u = 0; u < n; ++u) -----------------------------//24
--- dist[u] = INF; ----------------------------------------//6d
- dist[s] = 0; --------------------------------------------//63
- std::priority_queue<ii, vii, std::greater<ii> > pq; -----//e0
- pq.push({0, s}); ----------------------------------------//a3
- while (!pq.empty()) { -----------------------------------//9b
--- int u = pq.top().second; ------------------------------//10
--- int d = pq.top().first; -------------------------------//7b
--- pq.pop(); ---------------------------------------------//15
--- if (dist[u] < d) --------------------------------------//b1
----- continue; -------------------------------------------//27
--- dist[u] = d; ------------------------------------------//96
--- for (auto &e : adj[u]) { ------------------------------//24
----- int v = e.first; ------------------------------------//94
----- int w = e.second; -----------------------------------//0f
----- if (dist[v] > dist[u] + w) { ------------------------//ca
------- dist[v] = dist[u] + w; ----------------------------//d0
------- pq.push({dist[v], v});  }  }  }  } ----------------//4a
```

3.1.2. *Bellman-Ford.*

```
#include "graph_template_adjlist.cpp" ---------------------//76
void bellman_ford(int s, int n, int *dist, vii *adj) { ----//f4
- for (int u = 0; u < n; ++u) -----------------------------//2c
--- dist[u] = INF; ----------------------------------------//ba
- dist[s] = 0; --------------------------------------------//50
- for (int i = 0; i < n-1; ++i) ---------------------------//a0
--- for (int u = 0; u < n; ++u) ---------------------------//94
----- for (auto &e : adj[u]) ------------------------------//96
------- if (dist[u] + e.second < dist[e.first]) -----------//3b
--------- dist[e.first] = dist[u] + e.second;  } ----------//b4
// you can call this after running bellman_ford() ---------//06
bool has_neg_cycle(int n, int *dist, vii *adj) { ----------//26
- for (int u = 0; u < n; ++u) -----------------------------//2f
--- for (auto &e : adj[u]) --------------------------------//e9
```

```
----- if (dist[e.first] > dist[u] + e.second) -------------//b4
------- return true; --------------------------------------//ba
- return false;  } ----------------------------------------//7b
```

3.2. All-Pairs Shortest Paths.

3.2.1. *Floyd-Washall.*

```
#include "graph_template_adjmat.cpp" ----------------------//6e
void floyd_warshall(int n, int **mat) { -------------------//af
- for (int k = 0; k < n; ++k) -----------------------------//61
--- for (int i = 0; i < n; ++i) ---------------------------//d3
----- for (int j = 0; j < n; ++j) -------------------------//43
------- if (mat[i][k] + mat[k][j] < mat[i][j]) ------------//a3
--------- mat[i][j] = mat[i][k] + mat[k][j];  } -----------//99
```

3.3. Strongly Connected Components.

3.3.1. *Kosaraju.*

3.4. Cut Points and Bridges.

3.5. Biconnected Components.

3.5.1. *Bridge Tree.*

3.5.2. *Block-Cut Tree.*

3.6. Minimum Spanning Tree.

3.6.1. *Kruskal.*

3.6.2. *Prim.*

3.7. Topological Sorting.

3.8. Euler Path.

3.9. Bipartite Matching.

3.9.1. *Alternating Paths Algorithm.*

3.9.2. *Hopcroft-Karp Algorithm.*

3.10. Maximum Flow.

3.10.1. *Edmonds-Karp.*

```
struct max_flow { -----------------------------------------//e7
- int n, s, t, *par, **c, **f; ----------------------------//07
- vi *adj; ------------------------------------------------//22
- max_flow(int n, int s, int t) : n(n), s(s), t(t) { ------//5b
--- adj = new std::vector<int>[n]; ------------------------//a8
--- par = new int[n]; -------------------------------------//71
--- c = new int*[n]; --------------------------------------//71
--- f = new int*[n]; --------------------------------------//f7
--- for (int i = 0; i < n; ++i) { -------------------------//0f
----- c[i] = new int[n]; ----------------------------------//c7
----- f[i] = new int[n]; ----------------------------------//ba
----- for (int j = 0; j < n; ++j) -------------------------//e9
------- c[i][j] = f[i][j] = 0;  }  } ----------------------//13
- void add_edge(int u, int v, int w) { --------------------//5f
--- adj[u].push_back(v); ----------------------------------//ce
--- adj[v].push_back(u); ----------------------------------//bb
--- c[u][v] += w;  } --------------------------------------//a0
- int res(int i, int j) { return c[i][j] - f[i][j]; } -----//36
- bool bfs() { --------------------------------------------//d0
--- std::queue<int> q; ------------------------------------//1b
--- q.push(this->s); --------------------------------------//f4
--- while (!q.empty()) { ----------------------------------//d8
----- int u = q.front();  q.pop(); ------------------------//e5
----- for (int v : adj[u]) { ------------------------------//c4
------- if (res(u, v) > 0 and par[v] == -1) { -------------//44
--------- par[v] = u; -------------------------------------//2a
--------- if (v == this->t) -------------------------------//79
----------- return true; ----------------------------------//85
--------- q.push(v);  }  }  } -----------------------------//f5
--- return false;  } --------------------------------------//da
- bool aug_path() { ---------------------------------------//08
--- for (int u = 0; u < n; ++u) ---------------------------//7c
----- par[u] = -1; ----------------------------------------//9f
--- par[s] = s; -------------------------------------------//df
--- return bfs();  } --------------------------------------//3c
- int calc_max_flow() { -----------------------------------//96
--- int ans = 0; ------------------------------------------//b0
--- while (aug_path()) { ----------------------------------//19
----- int flow = INF; -------------------------------------//bd
----- for (int u = t; u != s; u = par[u]) -----------------//22
------- flow = std::min(flow, res(par[u], u)); ------------//a6
----- for (int u = t; u != s; u = par[u]) -----------------//40
------- f[par[u]][u] += flow, f[u][par[u]] -= flow; -------//ef
----- ans += flow;  } -------------------------------------//ad
--- return ans;  }  }; ------------------------------------//75
```

3.10.2. *Dinic.*

```
struct max_flow { -----------------------------------------//e7
- int n, s, t, *adj_ptr, *dist, *par, **c, **f; -----------//b8
- vi *adj; ------------------------------------------------//a3
- max_flow(int n, int s, int t) : n(n), s(s), t(t) { ------//0a
--- adj = new std::vector<int>[n]; ------------------------//f7
--- adj_ptr = new int[n]; ---------------------------------//a4
--- dist = new int[n]; ------------------------------------//61
--- par = new int[n]; -------------------------------------//19
--- c = new int*[n]; --------------------------------------//9d
--- f = new int*[n]; --------------------------------------//ff
--- for (int i = 0; i < n; ++i) { -------------------------//a8
----- c[i] = new int[n]; ----------------------------------//e2
----- f[i] = new int[n]; ----------------------------------//df
----- for (int j = 0; j < n; ++j) -------------------------//e8
------- c[i][j] = f[i][j] = 0;  }  } ----------------------//37
- void add_edge(int u, int v, int w) { --------------------//d9
--- adj[u].push_back(v); ----------------------------------//74
--- adj[v].push_back(u); ----------------------------------//ca
--- c[u][v] += w;  } --------------------------------------//11
- int res(int i, int j) { return c[i][j] - f[i][j]; } -----//4e
- void reset(int *ar, int val) { --------------------------//99
--- for (int i = 0; i < n; ++i) ---------------------------//a7
----- ar[i] = val;  } -------------------------------------//3e
- bool make_level_graph() { -------------------------------//4f
--- reset(dist, -1); --------------------------------------//75
--- std::queue<int> q; ------------------------------------//fc
--- q.push(s); --------------------------------------------//d5
--- dist[s] = 0; ------------------------------------------//0f
--- while (!q.empty()) { ----------------------------------//1c
----- int u = q.front();  q.pop(); ------------------------//19
----- for (int v : adj[u]) { ------------------------------//28
------- if (res(u, v) > 0 and dist[v] == -1) { ------------//42
--------- dist[v] = dist[u] + 1; --------------------------//00
--------- q.push(v);  }  }  } -----------------------------//bd
--- return dist[t] != -1;  } ------------------------------//72
- bool next(int u, int v) { return dist[v] == dist[u] + 1; }
- bool dfs(int u) { ---------------------------------------//fb
--- if (u == t)   return true; ----------------------------//66
--- for (int &i = adj_ptr[u]; i < adj[u].size(); ++i) { ---//c3
----- int v = adj[u][i]; ----------------------------------//99
----- if (next(u, v) and res(u, v) > 0 and dfs(v)) { ------//07
------- par[v] = u; ---------------------------------------//eb
------- return true;  }  } --------------------------------//ad
--- dist[u] = -1; -----------------------------------------//94
--- return false;  } --------------------------------------//4b
- bool aug_path() { ---------------------------------------//6d
--- reset(par, -1); ---------------------------------------//80
--- par[s] = s; -------------------------------------------//60
--- return dfs(s);  } -------------------------------------//39
- int calc_max_flow() { -----------------------------------//22
--- int ans = 0; ------------------------------------------//a3
--- while (make_level_graph()) { --------------------------//50
----- reset(adj_ptr, 0); ----------------------------------//78
----- while (aug_path()) { --------------------------------//45
------- int flow = INF; -----------------------------------//1f
------- for (int u = t; u != s; u = par[u]) ---------------//97
--------- flow = std::min(flow, res(par[u], u)); ----------//ae
------- for (int u = t; u != s; u = par[u]) ---------------//76
--------- f[par[u]][u] += flow, f[u][par[u]] -= flow; -----//a1
------- ans += flow;  }  } --------------------------------//42
--- return ans;  }  }; ------------------------------------//16
```

3.11. Centroid Decomposition.

3.12. Least Common Ancestor.

3.12.1. *Binary Lifting.*

3.12.2. *Tarjan's Offline Algorithm.*

4. Strings

4.1. Z-algorithm.

4.2. Trie.

4.3. Hashing.

5. Dynamic Programming

5.1. Longest Common Subsequence.

5.2. Longest Increasing Subsequence.

5.3. Traveling Salesman.

6. Mathematics

6.1. Special Data Types.

6.1.1. *Fraction.*

6.1.2. *BigInteger.*

6.1.3. *Matrix.*

6.1.4. *Dates.*

6.2. **Binomial Coefficients.**

6.3. **Euclidean Algorithm.**

6.4. **Primality Test.**

6.4.1. *Optimized Brute Force.*

6.4.2. *Miller-Rabin.*

6.4.3. *Pollard's Rho Algorithm.*

6.5. **Sieve.**

6.5.1. *Sieve of Eratosthenes.*

6.5.2. *Divisor Sieve (Modified Sieve of Eratosthenes).*

6.5.3. *Phi Sieve.*

6.6. **Phi Function.**

6.7. **Modular Exponentiation.**

6.8. **Modular Multiplicative Inverse.**

6.9. **Chinese Remainder Theorem.**

6.10. **Numeric Integration (Simpson's Rule).**

6.11. **Fast Fourier Transform.**

6.12. **Josephus Problem.**

6.13. **Number of Integer Points Below a Line.**

7. Geometry

7.1. **Primitives.**

7.2. **Lines.**

7.3. **Circles.**

7.4. **Polygons.**

7.5. **Convex Hull (Graham's Scan).**

7.6. **Closest Pair of Points.**

7.7. **Rectilinear Minimum Spanning Tree.**

8. Other Algorithms

8.1. **Coordinate Compression.**

8.2. **2SAT.**

8.3. **Nth Permutation.**

8.4. **Floyd's Cycle-Finding.**

8.5. **Simulated Annealing.**

8.6. **Hexagonal Grid Algorithms.**

9. Useful Information (CLEAN THIS UP!!)

10. Misc

10.1. Debugging Tips.

- Stack overflow? Recursive DFS on tree that is actually a long path?
- Floating-point numbers
  - Getting `NaN`? Make sure `acos` etc. are not getting values out of their range (perhaps `1+eps`).
  - Rounding negative numbers?
  - Outputting in scientific notation?
- Wrong Answer?
  - Read the problem statement again!
  - Are multiple test cases being handled correctly? Try repeating the same test case many times.
  - Integer overflow?
  - Think very carefully about boundaries of all input parameters
  - Try out possible edge cases:
    - $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$
    - List is empty, or contains a single element
    - n is even, n is odd
    - Graph is empty, or contains a single vertex
    - Graph is a multigraph (loops or multiple edges)
    - Polygon is concave or non-simple
  - Is initial condition wrong for small cases?
  - Are you sure the algorithm is correct?
  - Explain your solution to someone.
  - Are you using any functions that you don't completely understand? Maybe STL functions?
  - Maybe you (or someone else) should rewrite the solution?
  - Can the input line be empty?
- Run-Time Error?
  - Is it actually Memory Limit Exceeded?

10.2. Solution Ideas.

- Dynamic Programming
  - Parsing CFGs: CYK Algorithm
  - Drop a parameter, recover from others
  - Swap answer and a parameter
  - When grouping: try splitting in two
  - 2^k trick
  - When optimizing
    - Convex hull optimization
      - $\mathrm{dp}[i] = \min_{j<i}\{\mathrm{dp}[j] + b[j] \times a[i]\}$
      - $b[j] \geq b[j+1]$
      - optionally $a[i] \leq a[i+1]$
      - $O(n^2)$ to $O(n)$
    - Divide and conquer optimization
      - $\mathrm{dp}[i][j] = \min_{k<j}\{\mathrm{dp}[i-1][k] + C[k][j]\}$
      - $A[i][j] \leq A[i][j+1]$
      - $O(kn^2)$ to $O(kn \log n)$
      - sufficient: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$, $a \leq b \leq c \leq d$ (QI)
    - Knuth optimization
      - $\mathrm{dp}[i][j] = \min_{i<k<j}\{\mathrm{dp}[i][k] + \mathrm{dp}[k][j] + C[i][j]\}$
      - $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
      - $O(n^3)$ to $O(n^2)$
      - sufficient: QI and $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$
- Greedy
- Randomized
- Optimizations
  - Use bitset (/64)
  - Switch order of loops (cache locality)
- Process queries offline
  - Mo's algorithm
- Square-root decomposition
- Precomputation
- Efficient simulation
  - Mo's algorithm
  - Sqrt decomposition
  - Store 2^k jump pointers
- Data structure techniques
  - Sqrt buckets
  - Store 2^k jump pointers
  - 2^k merging trick
- Counting
  - Inclusion-exclusion principle
  - Generating functions
- Graphs
  - Can we model the problem as a graph?
  - Can we use any properties of the graph?
  - Strongly connected components
  - Cycles (or odd cycles)
  - Bipartite (no odd cycles)
    - Bipartite matching
    - Hall's marriage theorem
    - Stable Marriage
  - Cut vertex/bridge
  - Biconnected components
  - Degrees of vertices (odd/even)
  - Trees
    - Heavy-light decomposition
    - Centroid decomposition
    - Least common ancestor
    - Centers of the tree
  - Eulerian path/circuit
  - Chinese postman problem
  - Topological sort
  - (Min-Cost) Max Flow
  - Min Cut
    - Maximum Density Subgraph
  - Huffman Coding
  - Min-Cost Arborescence
  - Steiner Tree
  - Kirchoff's matrix tree theorem
  - Prüfer sequences
  - Lovász Toggle
  - Look at the DFS tree (which has no cross-edges)
  - Is the graph a DFA or NFA?
    - Is it the Synchronizing word problem?
- Mathematics
  - Is the function multiplicative?
  - Look for a pattern
  - Permutations
    - Consider the cycles of the permutation
  - Functions
    - Sum of piecewise-linear functions is a piecewise-linear function
    - Sum of convex (concave) functions is convex (concave)
  - Modular arithmetic
    - Chinese Remainder Theorem
    - Linear Congruence
  - Sieve
  - System of linear equations
  - Values too big to represent?
    - Compute using the logarithm
    - Divide everything by some large value
  - Linear programming
    - Is the dual problem easier to solve?
  - Can the problem be modeled as a different combinatorial problem? Does that simplify calculations?
- Logic
  - 2-SAT
  - XOR-SAT (Gauss elimination or Bipartite matching)
- Meet in the middle
- Only work with the smaller half ($\log(n)$)
- Strings
  - Trie (maybe over something weird, like bits)
  - Suffix array
  - Suffix automaton (+DP?)
  - Aho-Corasick
  - eerTree
  - Work with $S + S$
- Hashing
- Euler tour, tree to array
- Segment trees
  - Lazy propagation
  - Persistent
  - Implicit
  - Segment tree of X
- Geometry
  - Minkowski sum (of convex sets)
  - Rotating calipers
  - Sweep line (horizontally or vertically?)
  - Sweep angle
  - Convex hull
- Fix a parameter (possibly the answer).
- Are there few distinct values?
- Binary search
- Sliding Window (+ Monotonic Queue)
- Computing a Convolution? Fast Fourier Transform
- Computing a 2D Convolution? FFT on each row, and then on each column
- Exact Cover (+ Algorithm X)
- Cycle-Finding
- What is the smallest set of values that identify the solution? The cycle structure of the permutation? The powers of primes in the factorization?
- Look at the complement problem

  - Minimize something instead of maximizing
- Immediately enforce necessary conditions. (All values greater than 0? Initialize them all to 1)
- Add large constant to negative numbers to make them positive
- Counting/Bucket sort

11. Formulas

- **Legendre symbol:** $\left(\frac{a}{b}\right) = a^{(b-1)/2} \pmod b$, b odd prime.
- **Heron's formula:** A triangle with side lengths a, b, c has area $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{a+b+c}{2}$.
- **Pick's theorem:** A polygon on an integer grid strictly containing i lattice points and having b lattice points on the boundary has area $i + \frac{b}{2} - 1$. (Nothing similar in higher dimensions)
- **Euler's totient:** The number of integers less than n that are coprime to n are $n\prod_{p|n}\left(1 - \frac{1}{p}\right)$ where each p is a distinct prime factor of n.
- **König's theorem:** In any bipartite graph $G = (L \cup R, E)$, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover. Let U be the set of unmatched vertices in L, and Z be the set of vertices that are either in U or are connected to U by an alternating path. Then $K = (L \setminus Z) \cup (R \cap Z)$ is the minimum vertex cover.
- A minumum Steiner tree for n vertices requires at most $n-2$ additional Steiner vertices.
- The number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set.
- **Lagrange polynomial** through points $(x_0, y_0), \dots, (x_k, y_k)$ is $L(x) = \sum_{j=0}^{k} y_j \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m}$
- **Hook length formula:** If λ is a Young diagram and $h_\lambda(i, j)$ is the hook-length of cell (i, j), then then the number of Young tableux $d_\lambda = n!/\prod h_\lambda(i, j)$.
- **Möbius inversion formula:** If $f(n) = \sum_{d|n} g(d)$, then $g(n) = \sum_{d|n} \mu(d) f(n/d)$. If $f(n) = \sum_{m=1}^{n} g(\lfloor n/m \rfloor)$, then $g(n) = \sum_{m=1}^{n} \mu(m) f(\lfloor \frac{n}{m} \rfloor)$.
- #primitive pythagorean triples with hypotenuse $< n$ approx $n/(2\pi)$.
- **Frobenius Number:** largest number which can't be expressed as a linear combination of numbers $a_1, \dots, a_n$ with non-negative coefficients. $g(a_1, a_2) = a_1a_2 - a_1 - a_2$, $N(a_1, a_2) = (a_1 - 1)(a_2 - 1)/2$. $g(d \cdot a_1, d \cdot a_2, a_3) = d \cdot g(a_1, a_2, a_3) + a_3(d - 1)$. An integer $x > (\max_i a_i)^2$ can be expressed in such a way iff. $x \mid \gcd(a_1, \dots, a_n)$.

11.1. Physics.

- **Snell's law:** $\frac{\sin\theta_1}{v_1} = \frac{\sin\theta_2}{v_2}$

11.2. Markov Chains.

A Markov Chain can be represented as a weighted directed graph of states, where the weight of an edge represents the probability of transitioning over that edge in one timestep. Let $P^{(m)} = (p_{ij}^{(m)})$ be the probability matrix of transitioning from state i to state j in m timesteps, and note that $P^{(1)}$ is the adjacency matrix of the graph. **Chapman-Kolmogorov:** $p_{ij}^{(m+n)} = \sum_k p_{ik}^{(m)} p_{kj}^{(n)}$. It follows that $P^{(m+n)} = P^{(m)}P^{(n)}$ and $P^{(m)} = P^m$. If $p^{(0)}$ is the initial probability distribution (a vector), then $p^{(0)}P^{(m)}$ is the probability distribution after m timesteps.

The return times of a state i is $R_i = \{m \mid p_{ii}^{(m)} > 0\}$, and i is *aperiodic* if $\gcd(R_i) = 1$. A MC is aperiodic if any of its vertices is aperiodic. A MC is *irreducible* if the corresponding graph is strongly connected.

A distribution π is stationary if $\pi P = \pi$. If MC is irreducible then $\pi_i = 1/\mathbb{E}[T_i]$, where T_i is the expected time between two visits at i. π_j/π_i is the expected number of visits at j in between two consecutive visits at i. A MC is *ergodic* if $\lim_{m\to\infty} p^{(0)}P^m = \pi$. A MC is ergodic iff. it is irreducible and aperiodic.

A MC for a random walk in an undirected weighted graph (unweighted graph can be made weighted by adding 1-weights) has $p_{uv} = w_{uv}/\sum_x w_{ux}$. If the graph is connected, then $\pi_u = \sum_x w_{ux}/\sum_v \sum_x w_{vx}$. Such a random walk is aperiodic iff. the graph is not bipartite.

An *absorbing* MC is of the form $P = \begin{pmatrix} Q & R \\ 0 & I_r \end{pmatrix}$. Let $N = \sum_{m=0}^{\infty} Q^m = (I_t - Q)^{-1}$. Then, if starting in state i, the expected number of steps till absorption is the i-th entry in $N1$. If starting in state i, the probability of being absorbed in state j is the (i, j)-th entry of NR.

Many problems on MC can be formulated in terms of a system of recurrence relations, and then solved using Gaussian elimination.

11.3. Burnside's Lemma.

Let G be a finite group that acts on a set X. For each g in G let X^g denote the set of elements in X that are fixed by g. Then the number of orbits

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|$$

$$Z(S_n) = \frac{1}{n}\sum_{l=1}^{n} a_l Z(S_{n-l})$$

11.4. Bézout's identity.

If (x, y) is any solution to $ax + by = d$ (e.g. found by the Extended Euclidean Algorithm), then all solutions are given by

$$\left(x + k\frac{b}{\gcd(a, b)}, y - k\frac{a}{\gcd(a, b)}\right)$$

11.5. Misc.

11.5.1. *Determinants and PM.*

$$det(A) = \sum_{\sigma\in S_n} \operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$perm(A) = \sum_{\sigma\in S_n}\prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$\begin{aligned} pf(A) &= \frac{1}{2^n n!}\sum_{\sigma\in S_{2n}} \operatorname{sgn}(\sigma)\prod_{i=1}^{n} a_{\sigma(2i-1),\sigma(2i)} \\ &= \sum_{M\in \mathrm{PM}(n)} \operatorname{sgn}(M)\prod_{(i,j)\in M} a_{i,j} \end{aligned}$$

11.5.2. *BEST Theorem.*

Count directed Eulerian cycles. Number of OST given by Kirchoff's Theorem (remove r/c with root) $\#\mathrm{OST}(G, r) \cdot \prod_v (d_v - 1)!$

11.5.3. *Primitive Roots.*

Only exists when n is $2, 4, p^k, 2p^k$, where p odd prime. Assume n prime. Number of primitive roots $\phi(\phi(n))$ Let g be primitive root. All primitive roots are of the form g^k where $k, \phi(p)$ are coprime.
k-roots: $g^{i\cdot\phi(n)/k}$ for $0 \le i < k$

11.5.4. *Sum of primes.*

For any multiplicative f:

$$S(n, p) = S(n, p-1) - f(p)\cdot(S(n/p, p-1) - S(p-1, p-1))$$

11.5.5. *Floor.*

$$\lfloor\lfloor x/y\rfloor/z\rfloor = \lfloor x/(yz)\rfloor$$
$$x\%y = x - y\lfloor x/y\rfloor$$

Practice Contest Checklist

- How many operations per second? Compare to local machine.
- What is the stack size?
- How to use printf/scanf with long long/long double?
- Are `__int128` and `__float128` available?
- Does MLE give RTE or MLE as a verdict? What about stack overflow?
- What is `RAND_MAX`?
- How does the judge handle extra spaces (or missing newlines) in the output?
- Look at documentation for programming languages.
- Try different programming languages: C++, Java and Python.
- Try the submit script.
- Try local programs: i?python[23], factor.
- Try submitting with `assert(false)` and `assert(true)`.
- Return-value from `main`.
- Look for directory with sample test cases.
- Make sure printing works.
- Remove this page from the notebook.