noiref

ngmh

December 2019

Contents

1 Data Structures

Data Structure	Precomputation / Update	Query	Memory	Notes
Prefix Sum	O(N) / X	O(1)	O(N)	Associative Functions (+, XOR)
Sparse Table	O(N log N) / X	O(1)	O(N log N)	Non-Associative Functions (max, gcd)
Fenwick Tree	X / O(log N)	O(log N)	O(N)	Prefix Sum with Updates
Segment Tree	X / O(log N)	O(log N)	O(4N)	Allows more Information

Table 1: Quick Summary of Data Structures

1.1 Prefix Sums

Prefix Sums rely on the Principle of Inclusion and Exclusion. By adding and subtracting the correct prefixes, we can determine the answer for any subarray.
This idea can be extended to multiple dimensions as well, but beyond 2 it gets slightly cancerous.
Query: O(1)
Update: X

1.1.1 1D

```
1  //Query - 1-Indexed
2  int query(int s, int e){
3      return ps[e]-ps[s-1];
4  }
5
6  //Precomputation
7  for(int i = 1; i <= n; i++) ps[i] = ps[i-1]+a[i];
```

1.1.2 2D

```
1  //Query - 1-Indexed
2  int query(int x1, int y1, int x2, int y2){
3      return ps[x2][y2]-ps[x1-1][y2]-ps[x2][y1-1]+ps[x1-1][y1-1];
4  }
5
6  //Precomputation
7  for(int i = 1; i <= r; i++){
8      for(int j = 1; j <= c; j++){
9          ps[i][j] = ps[i-1][j]+ps[i][j-1]-ps[i-1][j-1]+a[i][j];
10     }
11 }
```

1.2 Sparse Table

Sparse Tables to me, are Prefix Sums on steroids. Instead of using prefixes, we use subarrays with sizes which are powers of 2. Then, any query will need at most 2 of the calculated subarrays.
Query: O(1)
Update: X

1.2.1 1D

```
1  //Query
2  int query(int l, int r){
3      r++;
4      int p = 31-__builtin_clz(r-l);
5      return __gcd(sp[p][l], sp[p][r-(1<<p)]);
```

```
 6  }
 7
 8  //Precomputation
 9  h = floor(log2(n));
10  for(int i = 0; i < n; i++) sp[0][i] = a[i];
11  for(int i = 1; i <= h; i++){
12      for(int j = 0; j+(1<<i) <= n; j++){
13          sp[i][j] = __gcd(sp[i-1][j], sp[i-1][j+(1<<(i-1))]);
14      }
15  }
```

1.2.2   2D

```
 1  Not Implemented Yet!
```

1.3   Fenwick Trees

Fenwick Trees essentially act the same as prefix sums, except that they can perform updates. However, the complexity of code varies depending on what kind of queries and updates are needed. One cool use case is for range maximum, but only works when updates are strictly non-decreasing.
Query: O(log N)
Update: O(log N)

1.3.1   Point Update, Range Query

```
 1  int ls(int x){ return (x)&(-x); }
 2
 3  void pu(int i, int v){
 4      for(; i <= n; i += ls(i)) fw[i] += v;
 5  }
 6  int pq(int i){
 7      int t = 0;
 8      for(; i; i -= ls(i)) t += fw[i];
 9      return t;
10  }
11  int rq(int s, int e){
12      return pq(e)-pq(s-1);
13  }
```

1.3.2   Range Update, Point Query

```
 1  int ls(int x){ return (x)&(-x); }
 2
 3  //PURQ Code (PU, PQ)
 4
 5  void ru(int s, int e, int v){
 6      pu(s, v);
 7      pu(e+1, -v);
 8  }
```

1.3.3   Range Update, Range Query

```
1   int ls(int x){ return (x)&(-x); }
2
3   //PURQ Code (PU, PQ)
4   //Modify functions:
5   //int pu(*tree...
6
7   void ru(int s, int e, int v){
8       pu(fw1, s, v);
9       pu(fw1, e+1, -v);
10      pu(fw2, s, -v*(s-1));
11      pu(fw2, e+1, v*e);
12  }
13  int ps(int i){
14      return pq(fw1, i)*i+pq(fw2, i);
15  }
16  int rq(int s, int e){
17      return ps(e)-ps(s-1);
18  }
```

1.4 Segment Trees

Segment Trees need more memory and code to implement, but are much more powerful, since more information can be stored in each node. They can be used for anything really, e.g. finding how many elements are larger than K, Maxsum.

Query: O(log N)

Update: O(log N)

1.4.1 1D

```
1   struct node {
2       int s, e, m, v;
3       node *l, *r;
4       node(int _s, int _e){
5           s = _s; e = _e; m = (s+e)/2; v = 0;
6           if(s != e){
7               l = new node(s, m);
8               r = new node(m+1, e);
9           }
10      }
11      void pu(int x, int y){
12          if(s == e){ v = y; return; }
13          if(x <= m) l->pu(x, y);
14          if(x > m) r->pu(x, y);
15          v = min(l->v, r->v);
16      }
17      int rq(int x, int y){
18          if(s == x && e == y) return v;
19          if(y <= m) return l->rq(x, y);
20          if(x > m) return r->rq(x, y);
21          return min(l->rq(x, m), r->rq(m+1, y));
22      }
23  } *root;
```

1.4.2 Lazy Propagation - Range Add

```
1  int pu(){
2      if(s == e){ v += lazy; lazy = 0; return v; }
3      v += lazy;
4      l->lazy += lazy; r->lazy += lazy;
5      lazy = 0;
6      return v;
7  }
8
9  void ru(int x, int y, int z){
10     if(s == x && e == y){ lazy += z; return; }
11     if(y <= m) l->ru(x, y, z);
12     else if(x > m) r->ru(x, y, z);
13     else l->ru(x, m, z), r->ru(m+1, y, z);
14     v = max(l->pu(), r->pu());
15 }
16
17 int rq(int x, int y){
18     pu();
19     if(s == x && e == y) return pu();
20     if(y <= m) return l->rq(x, y);
21     if(x > m) return r->rq(x, y);
22     return max(l->rq(x, m), r->rq(m+1, y));
23 }
```

1.4.3 Lazy Propagation - Range Add + Set

```
1  Not Implemented Yet!
```

1.4.4 Maxsum Tree

```
1  struct node {
2      ll s, e, m, ps, ss, ms, ts;
3      node *l, *r;
4      node(ll _s, ll _e){
5          s = _s; e = _e; m = (s+e)/2; ps = ss = ms = ts = 0;
6          if(s != e){
7              l = new node(s, m);
8              r = new node(m+1, e);
9          }
10     }
11     void pu(ll x, ll y){
12         if(s == e){ ps = ss = ms = ts = y; return; }
13         if(x <= m) l->pu(x, y);
14         if(x > m) r->pu(x, y);
15         //New Prefix Max - Left Prefix, Left + Right Prefix
16         ps = max(l->ps, l->ts+r->ps);
17         //New Suffix Max - Right Suffix, Left Suffix + Right
18         ss = max(r->ss, r->ts+l->ss);
19         //Total Sum - Left + Right
20         ts = l->ts+r->ts;
21         //Maxsum - Left Suffix + Right Prefix, Left, Right,
22         //         Total, Left Maxsum, Right Maxsum
23         ms = max({l->ss+r->ps, ps, ss, ts, l->ms, r->ms});
24     }
25     ll ans(){
26         return ms;
```

```
27        }
28  } *root;
```

1.4.5   Order Statistic Tree - More than K

```
1  Not Implemented Yet!
```

1.4.6   Order Statistic Tree - More than K + Updates

```
1  Not Implemented Yet!
```

1.4.7   2D

```
 1  struct node2D {
 2      int s, e, m;
 3      node1D *maxi;
 4      node2D *l, *r;
 5      node2D(int a, int b, int c, int d){
 6          s = a; e = b; m = (s+e)/2;
 7          maxi = new node1D(c, d);
 8          if(s != e){
 9              l = new node2D(s, m, c, d);
10              r = new node2D(m+1, e, c, d);
11          }
12      }
13      void pu(int a, int b, int v){
14          if(s == e){ maxi->pu(b, v); return; }
15          if(a <= m) l->pu(a, b, v);
16          else r->pu(a, b, v);
17          maxi->pu(b, max(l->maxi->rq(b, b), r->maxi->rq(b, b)));
18      }
19      int rq(int a, int b, int c, int d){
20          if(s == a && e == b) return maxi->rq(c, d);
21          if(b <= m) return l->rq(a, b, c, d);
22          if(a > m) return r->rq(a, b, c, d);
23          return max(l->rq(a, m, c, d), r->rq(m+1, b, c, d));
24      }
25  } *root;
```

2 Graph Theory

Graph Algorithm	Complexity	Notes
DFS	O(N)	Flood Fill, Trees
BFS	O(N)	Unweighted Shortest Path
Floyd-Warshall	$O(N^3)$	All Pairs Shortest Path
Dijkstra	O(E log E)	Single Source Shortest Path
TSP	$O(2^N)$	Tour All Nodes
UFDS	O(1)	Checking Connectedness
MST - Kruskal	O(E log E)	Greedy
MST - Prim's	O(E log E)	Dijkstra
Bipartite Matching	$O(N^{5/2})$	2 Sets of Nodes
Articulation Points	O(N+E)	Find Splitting Nodes
Bridges	O(N+E)	Find Splitting Edges
SCC	O(N+E)	Nodes Can Reach Everyone

Table 2: Quick Summary of General Graph Algorithms

2.1 Depth First Search

Depth First Search goes as far in as possible, before coming all the way back out. It can be problematic on general graphs, but works fine and is easier to use for trees.
Time Complexity: O(N)

```
1  void dfs(int x, int p){
2      for(auto it : adj[x]){
3          if(it != p){
4              par[it] = x;
5              dep[it] = dep[x]+1;
6              dist[it] = dist[x]+1;
7              dfs(it, x);
8          }
9      }
10 }
```

2.2 Breadth First Search

Breadth First Search slowly spreads out from the source before going deeper in. It works well on general graphs but is slightly more tedious to code.
It can also find shortest paths on 0-1 weighted graphs. This is done using a deque and pushing 0s to the front and 1s to the back.
Time Complexity: O(N)

```
1  d[nx][ny] = 0;
2  q.push(pi(sx, sy));
3  while(!q.empty()){
4      pi f = q.front(); q.pop();
5      for(int i = 0; i < 4; i++){
6          nx = f.first+dx[i];
7          ny = f.second+dy[i];
8          if(nx < 0 || ny < 0 || nx >= h || ny >= w) continue;
9          if(d[nx][ny] != -1) continue;
10         d[nx][ny] = d[f.first][f.second]+1;
11         q.push(pi(nx, ny));
12     }
13 }
```

2.3 Floyd-Warshall

Floyd-Warshall is very slow, so it is unlikely to be used. However, if there are too many edges, using Dijkstra from each node will still end up slower.
Time Complexity: $O(N^3)$

```
1  for(int i = 0; i < n; i++){
2      for(int j = 0; j < n; j++){
3          if(i == j) adj[i][j] = 0;
4          else adj[i][j] = INT_MAX;
5      }
6  }
7  for(int k = 0; k < n; k++){
8      for(int i = 0; i < n; i++){
9          for(int j = 0; j < n; j++){
10             adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
11         }
12     }
13 }
```

2.4 Dijkstra

Dijkstra is much faster at shortest paths, but complexity is dependent on the number of edges.
Time Complexity: O(N)

```
1  priority_queue<pi, vector<pi>, greater<pi> > pq;
2  dist[s] = 0;
3  pq.push(pi(0, s));
4  while(!pq.empty()){
5      pi f = pq.top(); pq.pop();
6      for(auto it:adj[f.second]){
7          if(dist[it.first] == -1 || dist[it.first] > f.first+it.second){
8              dist[it.first] = f.first+it.second;
9              pq.push(pi(dist[it.first], it.first));
10         }
11     }
12 }
```

2.5 Travelling Salesman Problem

The Travelling Salesman Problem uses DP on Bitmask to solve it. The Bitmask indicates which nodes have already been visited.
Time Complexity: $O(2^N)$

```
1  long long adj[14][14], dp[8200][14];
2
3  //start with 1, 0
4  //mask is 0 visited, at node 0
5  long long tsp(long long mask, long long pos){
6      if(mask == visited) return adj[pos][0];
7      if(dp[mask][pos] != -1) return dp[mask][pos];
8      long long ans = LLONG_MAX;
9      for(int i = 0; i < m; i++){
10         if((mask&(1<<i)) == 0){
11             long long newans = adj[pos][i]+tsp(mask|(1<<i), i);
12             ans = min(ans, newans);
```

```
13              }
14          }
15      return dp[mask][pos] = ans;
16  }
17
18
19  visited = (1 << m)-1;
20  res = tsp(1, 0);
21  if(res < 0) cout << "-1";
22  else cout << res;
```

2.6   Travelling Salesman Problem - BFS

```
1  Not Implemented Yet!
```

2.7   Union Find Disjoint Subset

Union Find Disjoint Subset connects subsets of nodes together. Using techniques like path compression reduce it's time complexity to O(1).
Time Complexity: O(1)

```
1  int root(int x){
2      if(p[x] == -1) return x;
3      return p[x] = root(p[x]);
4  }
5  void connect(int x, int y){
6      p[root(x)] = root(y);
7  }
```

2.8   Minimum Spanning Tree

By stripping off larger edges, we are left with a tree such that the weights of edges between nodes are the minimum possible.
Time Complexity: O(E log E)

2.8.1   Kruskal

Kruskal is a greedy algorithm by processing starting from the smallest edge. It runs in O(E), but is O(E log E) because it needs to sort the edges.
Time Complexity: O(E log E)

```
1  sort(edgs.begin(), edgs.end());
2  for(auto it:edgs){
3      if(root(it.second.first) != root(it.second.second)){
4          connect(it.second.first, it.second.second);
5          cost += it.first;
6      }
7  }
```

2.8.2   Prim's

Prim's is literally Dijkstra, but using max instead of +. It can also be used to find minimum maximum edge from a single source.
Time Complexity: O(E log E)

```
1  priority_queue<pi, vector<pi>, greater<pi> > pq;
2  dist[s] = 0;
3  pq.push(pi(0, s));
4  while(!pq.empty()){
5      pi f = pq.top(); pq.pop();
6      for(auto it:adj[f.second]){
7          if(dist[it.first] == -1 || dist[it.first] > max(f.first,it.second)){
8              dist[it.first] = max(f.first, it.second);
9              pq.push(pi(dist[it.first], it.first));
10         }
11     }
12 }
```

2.9 Bipartite Matching

A Bipartite Graph has nodes in 2 sets, such that edges only run across both sets. A chooses edges such that each node only has a maximum of one edge.
Time Complexity: O($N^{5/2}$)

```
1  bool dfs(int u){
2      if(v[u]) return 0;
3      v[u] = 1;
4      for(auto v:adj[u]){
5          if(match[v] == -1 || dfs(match[v])){
6              match[v] = u;
7              match[u] = v;
8              return 1;
9          }
10     }
11     return 0;
12 }
13 memset(match, -1, sizeof(match));
14 for(auto lit:ho){
15     for(auto rit:adj[lit]){
16         if(match[rit] == -1){
17             match[rit] = lit;
18             match[lit] = rit;
19             mcbm++;
20             break;
21         }
22     }
23 }
24 for(auto lit:ho){
25     if(match[lit] == -1){
26         memset(v, 0, sizeof(v));
27         mcbm += dfs(lit);
28     }
29 }
30 mis = n-mcbm;
```

2.10 Articulation Points

Articulation Points are nodes that split the graph when they are removed. They are found using Tarjan's.
Time Complexity: O(N+E)

```
1  void atp(int node, int depth){
2      vi[node] = 1;
3      dep[node] = depth;
4      low[node] = depth;
5      for(auto it:adj[node]){
6          if(!vi[it]){
7              par[it] = node;
8              atp(it, depth+1);
9              chi[node]++;
10             if(low[it] >= dep[node]) atps[node]++;
11             low[node] = min(low[node], low[it]);
12         } else if(it != par[node]){
13             low[node] = min(low[node], dep[it]);
14         }
15     }
16 }
17 atp(0, 0);
18 // root -> chi[i]
19 // other -> atps[i]+1
```

2.11 Bridges

Bridges are edges that split the graph when they are removed. They are found using Tarjan's.
Time Complexity: O(N+E)

```
1  void bridges(int x, int par){
2      if(dep[x] != -1) return;
3      dep[x] = low[x] = co++;
4      int t = 0;
5      for(auto it:adj[x]){
6          if(it == par && t == 0){ t++; continue; }
7          if(dep[it] != -1){
8              if(low[it] > dep[x]) bgs.push_back(pi(x, it));
9              low[x] = min(low[x], low[it]);
10             continue;
11         }
12         bridges(it, x);
13         if(low[it] > dep[x]) bgs.push_back(pi(x, it));
14         low[x] = min(low[x], low[it]);
15     }
16 }
17 for(int i = 1; i <= n; i++) bridges(i, 0);
```

2.12 Strongly Connected Components

Strongly Connected Components are sets of nodes where all nodes can visit each other. After compressing them into single nodes, the new graph formed is a DAG (Directed Acyclic Graph).
Time Complexity: O(N+E)

```
1  stack<int> s;
2  vector<int> cur, adj[100001];
3  set<int> adjscc[100001];
4  vector<vector<int> > comps;
5  void scc(int v){
6      idxs[v] = idx;
7      lowlink[v] = idx;
```

```
 8        idx++;
 9        s.push(v);
10        ins[v] = 1;
11        for(auto w:adj[v]){
12            if(idxs[w] == -1){
13                scc(w);
14                lowlink[v] = min(lowlink[v], lowlink[w]);
15            } else if(ins[w]){
16                lowlink[v] = min(lowlink[v], idxs[w]);
17            }
18        }
19        if(lowlink[v] == idxs[v]){
20            cur.clear();
21            w = 0;
22            while(w != v){
23                w = s.top(); s.pop();
24                ins[w] = 0;
25                cur.push_back(w);
26            }
27            comps.push_back(cur);
28        }
29 }
30 idx = 1;
31 memset(idxs, -1, sizeof(idxs));
32 for(int i = 1; i <= n; i++){
33     if(idxs[i] == -1) scc(i);
34 }
35 memset(com, -1, sizeof(com));
36 for(int i = 0; i < comps.size(); i++){
37     for(auto it:comps[i]) com[it] = i;
38 }
39 for(int i = 1; i <= n; i++){
40     for(auto it:adj[i]){
41         if(com[i] != com[it]) adjscc[com[i]].insert(com[it]);
42     }
43 }
```

2.13   Trees

2.13.1   Diameter

```
 1 pi dfs(int node, int par, int dist){
 2     pi b = pi(node, dist);
 3     for(auto it:adj[node]){
 4         if(it.first != par){
 5             pi c = dfs(it.first, node,dist+it.second);
 6             if(c.second > b.second) b = c;
 7         }
 8     }
 9     return b;
10 }
11 f = dfs(0, -1, 0);
12 l = dfs(f.first, -1, 0);
13 // l.second
```

2.13.2 2^K Decomposition

```
1  int par(int x, int k){
2      for(int i = 19; i >= 0; i--){
3          if(k >= (1<<i)){
4              if(x == -1) return x;
5              x = p[x][i];
6              k -= (1<<i);
7          }
8      }
9      return x;
10 }
11
12 memset(p, -1, sizeof(p));
13 dfs(0);
14 for(int k = 1; k <= 19; k++){
15     for(int i = 0; i < n; i++){
16         if(p[i][k-1] != -1) p[i][k] = p[p[i][k-1]][k-1];
17     }
18 }
```

2.13.3   Lowest Common Ancestor

```
1  int lca(int x, int y){
2      if(dep[x] < dep[y]) swap(x, y);
3      for(int k = 19; k >= 0; k--){
4          if(p[x][k] != -1 && dep[p[x][k]] >= dep[y]) x = p[x][k];
5      }
6      if(x == y) return x;
7      for(int k = 19; k >= 0; k--){
8          if(p[x][k] != p[y][k]){
9              x = p[x][k];
10             y = p[y][k];
11         }
12     }
13     return p[x][0];
14 }
```

2.13.4   All Pairs Shortest Path

```
1  int distance(int x, int y){
2      return dist[x]+dist[y]-2*dist[lca(x, y)];
3  }
```

2.13.5   Preorder

```
1  // UNTESTED
2  void dfs(int x, int par){
3      c++;
4      pre[x] = c;
5      for(auto it:adj[x]){
6          if(it != par) dfs(adj[it], x);
7      }
8  }
```

2.13.6 Postorder

```
1  // UNTESTED
2  void dfs(int x, int par){
3      c++;
4      for(auto it:adj[x]){
5          if(it != par) dfs(adj[it], x);
6      }
7      post[x] = c;
8  }
```

2.13.7 Subtree to Range

```
1  int dfs(int x, int par){
2      c++;
3      pre[x] = c;
4      for(auto it:adj[x]){
5          if(it != par) rig[pre[x]] = max(rig[pre[x]], dfs(it, x));
6      }
7      if(rig[pre[x]] == 0) rig[pre[x]] = pre[x];
8      return rig[pre[x]];
9  }
10 // node -> pre[x]
11 // children -> pre[x]+1, rig[pre[x]]
```

2.13.8 Leaf Pruning

```
1  for(int i = 1; i <= n; i++){
2      if(adj[i].size() == 1) leafs.push(i);
3  }
4  while(!leafs.empty()){
5      u = leafs.front(); leafs.pop();
6      t = adj[u][0];
7      adj[t].erase(find(adj[t].begin(), adj[t].end(), u));
8      if(adj[t].size() == 1) leafs.push(t);
9      adj[u].erase(find(adj[u].begin(), adj[u].end(), t));
10 }
```

2.13.9 Weighted Maximum Independent Set

```
1  int mis(int v, bool take, int p){
2      if(dp[v][take] != -1) return dp[v][take];
3      int ans = take*c[v];
4      for(auto it:adj[v]){
5          if(it == p) continue;
6          int temp = mis(it, 0, v);
7          if(!take) temp = max(temp, mis(it, 1, v));
8          ans += temp;
9      }
10     return dp[v][take] = ans;
11 }
12 void ans(int v, bool take, int p){
13     for(auto it:adj[v]){
14         if(it == p) continue;
15         int temp0 = dp[it][0], temp1 = (take ? -1 : dp[it][1]);
```

```
16        if(temp0 > temp1) ans(it, 0, v);
17        else { a.push_back(it); ans(it, 1, v); }
18     }
19 }
20 mis(0, 0, -1);
21 mis(0, 1, -1);
22 if(dp[0][1] > dp[0][0]){ a.push_back(0); ans(0, 1, -1); }
23 else ans(0, 0, -1);
```

2.13.10 Heavy-Light Decomposition

```
1  int dfs(int node){
2      int size = 1, max_c = 0;
3      for(auto it:adj[node]){
4          if(it.s != par[node]){
5              par[it.s] = node; dep[it.s] = dep[node]+1;
6              int c_size = dfs(it.s);
7              size += c_size;
8              if(c_size > max_c){ max_c = c_size; heav[node] = it.s; }
9          }
10     }
11     return size;
12 }
13 void decomp(int node, int hea){
14     head[node] = hea; pos[node] = c_pos++;
15     if(heav[node] != -1) decomp(heav[node], hea);
16     for(auto it:adj[node]){
17         if(it.s == par[node]) continue;
18         if(it.s != heav[node]) decomp(it.s, it.s);
19         root->update(pos[it.s], it.f);
20     }
21 }
22 int query(int a, int b){
23     int res = 0;
24     for(; head[a] != head[b]; b = par[head[b]]){
25         if(dep[head[a]] > dep[head[b]]) swap(a, b);
26         res = max(res, root->query(pos[head[b]], pos[b]));
27     }
28     if(dep[a] > dep[b]) swap(a, b);
29     res = max(res, root->query(pos[a]+1, pos[b]));
30     return res;
31 }
32 dfs(0);
33 decomp(0, 0);
```

2.13.11 Centroid Decomoposition

```
1  ll dfs1(ll u, ll p, ll l){
2      sub[u] = 1;
3      for(auto it : adj[u]){
4          if(ban[it.f] != -1) continue;
5          if(it.f == p) continue;
6          if(l) dst[it.f][l-1] = dst[u][l-1]+it.s;
7          sub[u] += dfs1(it.f, u, l);
8      }
9      return sub[u];
```

```
10  }
11
12  ll dfs2(ll u, ll p, ll n){
13      for(auto it : adj[u]){
14          if(ban[it.f] != -1) continue;
15          if(it.f != p && sub[it.f] > n/2){
16              return dfs2(it.f, u, n);
17          }
18      }
19      return u;
20  }
21
22  void build(ll u, ll p, ll l){
23      ll n = dfs1(u, p, l);
24      ll cent = dfs2(u, p, n);
25      if(p == -1) p = cent;
26      par[cent] = p;
27      ban[cent] = l;
28      for(auto it : adj[cent]){
29          if(ban[it.f] != -1) continue;
30          dst[it.f][l] = it.s;
31          build(it.f, cent, l+1);
32      }
33  }
34
35  void update(ll x){
36      ll lvl = ban[x];
37      ll y = x;
38      while(lvl != -1){
39          ans[y] = min(ans[y], dst[x][lvl]);
40          st.push(y);
41          y = par[y];
42          lvl--;
43      }
44  }
45
46  ll query(ll x){
47      ll res = LLONG_MAX/3;
48      ll lvl = ban[x];
49      ll y = x;
50      while(lvl != -1){
51          res = min(res, ans[y]+dst[x][lvl]);
52          y = par[y];
53          lvl--;
54      }
55      return res;
56  }
57
58  //Initialisation
59  memset(ban, -1, sizeof(ban));
60  build(0, -1, 0);
61  for(ll i = 0; i < n; i++) ans[i] = LLONG_MAX/3;
62
63  //Adding a node to be considered
64  update(X);
65
66  //Querying with all considered nodes
67  query(Y);
```

```
68   while(!st.empty()){ ans[st.top()] = LLONG_MAX/3; st.pop(); }
```

3 Dynamic Programming

DP Algorithm	Complexity	Notes
Coin Change	O(NV)	
Coin Combinations	O(NV)	
Knapsack - 0-1	O(NS)	
Knapsack - 0-K	O(log2(K)+NS)	
LIS - Naive	O(N^2)	
LIS - DS / Optimal	O(N log N)	
LCS	O(N^2)	
LCS - LIS	O(N log N)	
Digits	O(10—N—)	
Convex Hull Speedup	O(N)	Amortized
Divide and Conquer	O(N log N)	

Table 3: Quick Summary of General Dynamic Programming Algorithms

3.1 Coin Change

```
1   dp[0] = 0;
2   for(int i = 1; i <= v; i++){
3       dp[i] = INT_MAX;
4       for(int j = 0; j < n; j++){
5           if(i >= c[j]) dp[i] = min(dp[i], dp[i-c[j]]+1);
6       }
7   }
```

3.2 Coin Combinations

```
1   ways[0] = 1;
2   for(int i = 0; i < c; i++){
3       for(int j = 1; j <= v; j++){
4           if(j >= coins[i]) ways[j] += ways[j-coins[i]];
5       }
6   }
```

3.3 Knapsack

3.3.1 0-1

```
1   for(int i = 0; i < n; i++){
2       for(int j = s; j >= w[i]; j--){
3           dp[j] = max(dp[j], dp[j-w[i]]+v[i]);
4       }
5   }
```

3.3.2 0-K

```
1  //further speedup:
2  //take top s/w most valued items
3  //for every possible item
4  //not just multiple copies of one item
5
6  for(int i = 0; i < n; i++){
7      cin >> v >> w >> k;
8      k = min(k, s/w);
9      c = 1;
10     while(true){
11         if(k < c) break;
12         items.push_back(pi(v*c, w*c));
13         k -= c;
14         c *= 2;
15     }
16     if(k != 0) items.push_back(pi(v*k, w*k));
17 }
18 for(auto it :items){
19     for(int i = s; i >= it.second; i--){
20         dp[i] = max(dp[i], dp[i-it.second]+it.first);
21     }
22 }
```

3.4 Longest Increasing Subsequence

3.4.1 N^2

```
1  for(int i = 0; i < n; i++){
2      for(int j = 0; j < i; j++){
3          if(a[j] < a[i]) lis[i] = max(lis[i], lis[j]);
4      }
5      lis[i]++;
6      ans = max(ans, lis[i]);
7  }
```

3.4.2 $NlogN$

```
1  for(int i = 0; i < n; i++){
2      t = query(a[i]-1)+1;
3      update(a[i], t);
4      ans = max(ans, t);
5  }
```

3.4.3 Optimal

```
1  for(int i = 0; i < n; i++){
2      int p = lower_bound(dp, dp+l, a[i])-dp;
3      dp[p] = a[i];
4      l = max(p+1, l);
5  }
```

3.5 Longest Common Subsequence

3.5.1 N^2

```cpp
int lcs(int al, int bl){
    if(al == 0 || bl == 0) return 0;
    if(dp[al][bl] != -1) return dp[al][bl];
    dp[al][bl] = max(lcs(al-1, bl), lcs(al, bl-1));
    if(a[al-1] == b[bl-1]) dp[al][bl] = max(dp[al][bl], lcs(al-1, bl-1)+1);
    return dp[al][bl];
}
```

3.5.2 Longest Increasing Subsequence

```cpp
for(int i = 0; i < n; i++){ cin >> a[i].first; a[i].second = i+1; }
for(int i = 0; i < m; i++) cin >> b[i];
sort(a, a+n);
for(int i = 0; i < m; i++){
    p = lower_bound(a, a+n, pi(b[i], 0));
    if(p != a+n) c.push_back(p->second);
}
// Perform LIS on c
```

3.6 Digits

```cpp
long long derp(int idx, int prev, int same, int allzero){
    if(idx == num.size()) return dp[idx][prev][same][allzero] = !allzero;
    if(dp[idx][prev][same][allzero] != -1) return dp[idx][prev][same][allzero];
    long long sum = 0, limit;
    if((!allzero && same) || (allzero && idx == 0)) limit = num[idx];
    else limit = 9;
    for(int i = 0; i <= limit; i++){
        if(i == 4) continue;
        if(allzero){
            if(i == 0) sum += derp(idx+1, 10, 1, 1);
            else sum += derp(idx+1, i, (idx == 0 && i == limit), 0);
        } else if(!(i == 3 && prev == 1)){
            if(same && i == num[idx]) sum += derp(idx+1, i, 1, 0);
            else sum += derp(idx+1, i, 0, 0);
        }
    }
    return dp[idx][prev][same][allzero] = sum;
}
void dcmp(long long x){
    num.clear();
    while(x > 0){ num.push_back(x%10); x /= 10; }
    reverse(num.begin(), num.end());
}
long long solve(long long x){
    if(memo.count(x) != 0) return memo[x];
    memset(dp, -1, sizeof(dp));
    dcmp(x);
    return memo[x] = derp(0, 10, 1, 1);
}
```

3.7 Convex Hull Speedup

This speeds up any DP which is a quadratic function. A similar idea can also be done for linear functions but just using a set.
The important part is knowing how to rearrange the transition to get coefficients.
Query: O(1) Update: O(1)

```
1  // dp(x) = max(dp(i)+f(p(x)-p(i)))
2  //        = max(dp(i)+a(p(x)-p(i))^2+b(p(x)-p(i))+c)
3  //        = max(dp(i)+ap(x)^2-2ap(x)p(i)+ap(i)^2+bp(x)-bp(i))+c
4  //        = max(dp(i)+ap(i)^2-bp(i)-2ap(x)p(i))+ap(x)^2+bp(x)+c
5  //        = max(c(i)+m(i)p(x))+v(x)
6  // c(i) = dp(i)+ap(i)^2-bp(i)
7  // m(i) = -2ap(i)
8  // v(x) = ap(x)^2+bp(x)+c
9  long long func(pi line, long long x){
10     return line.first*x+line.second;
11 }
12 long double intersection(long long m1, long long c1, long long m2, long long c2){
13     return (long double)(c2-c1)/(m1-m2);
14 }
15 long double intersect(pi x, pi y){
16     return intersection(x.first, x.second, y.first, y.second);
17 }
18 long long query(long long x){
19     while(hull.size() > 1){
20         if(func(hull[0], x) < func(hull[1], x)){
21             hull.pop_front();
22         } else break;
23     }
24     return func(hull[0], x);
25 }
26 void insert(long long m, long long c){
27     pi line = pi(m, c);
28     while(hull.size() > 1){
29         long long s = hull.size();
30         if(intersect(hull[s-1], line) <= intersect(hull[s-2], line)){
31             hull.pop_back();
32         } else break;
33     }
34     hull.push_back(line);
35 }
36 insert(0, 0);
37 for(int i = 1; i <= n; i++){
38     dp[i] = query(ps[i])+a*ps[i]*ps[i]+b*ps[i]+c;
39     insert(-2*a*ps[i], dp[i]+a*ps[i]*ps[i]-b*ps[i]);
40 }
```

3.8 Divide and Conquer

```
1  long long cost(int s, int e){
2      return (ps[e]-ps[s-1])*(e-s+1);
3  }
4  void dnc(int s, int e, long long x, int y, int k){
5      if(s > e) return;
6      int m = (s+e)/2, best = 0;
7      dp[m][k] = LLONG_MAX/2;
```

```
8        for(int i = x; (i <= y && i <= m); i++){
9            if(dp[m][k] > dp[i][!k]+cost(i+1, m)){
10               dp[m][k] = dp[i][!k]+cost(i+1, m);
11               best = i;
12           }
13       }
14       if(s < m) dnc(s, m-1, x, best, k);
15       if(m < e) dnc(m+1, e, best, y, k);
16   }
17   for(int i = 1; i <= n; i++) dp[i][0] = LLONG_MAX/2;
18   for(int i = 1; i <= g; i++){
19       for(int j = 1; j <= n; j++) dp[j][i%2] = LLONG_MAX/2;
20       dnc(0, n, 0, n, i%2);
21   }
```

4 Math

4.1 Greatest Common Divisor

```
1   // Alternatively, find highest common powers for each factor
2   int gcd(int a, int b){
3       if(b == 0) return a;
4       return gcd(b, a%b);
5   }
```

4.2 Lowest Common Multiple

```
1   // Alternatively, find highest powers for each factor
2   int lcm(int a, int b){
3       return (a*b)/gcd(a, b);
4   }
```

4.3 Modular Functions

4.3.1 Multiplication

```
1   int mulmod(int a, int b, int m) {
2       int res = 0;
3       while(b > 0) {
4       if(b % 2 == 1) res = (res+a)%m;
5           a = (a*2)%m;
6           b /= 2;
7       }
8       return res % m;
9   }
```

4.3.2 Exponentiation

```
1   int powmod(int a, int b, int m) {
2       int res = 1;
3       while(b > 0) {
4       if(b % 2 == 1) res = mulmod(res, a, m);
5           a = mulmod(a, a, m);
6           b /= 2;
```

```
7        }
8        return res % m;
9    }
```

4.4   Primes

4.4.1   Sieve of Eratosthenes

```
1    memset(prime, 1, sizeof(prime));
2    prime[0] = prime[1] = 0;
3    for(int i = 2; i <= 1000000; i++){
4        if(prime[i]){
5            for(int j = 2; i*j <= 1000000; j++){
6                prime[i*j] = 0;
7            }
8        }
9    }
```

4.4.2   Prime Factorisation

```
1    while(x % 2 == 0){ cnt[2]++; x /= 2; }
2    for(int i = 3; i*i <= x+1; i += 2){
3        while(x % i == 0){ cnt[i]++; x /= i; }
4        if(x == 1) break;
5    }
6    if(x > 1) cnt[x]++;
```

4.5   Fibonacci

```
1    typedef pair<pi, pi> matrix;
2    matrix multiply(matrix x, matrix y, long long z){
3        return matrix(
4                pi((x.f.f*y.f.f+x.f.s*y.s.f)%z,
5                  (x.f.f*y.f.s+x.f.s*y.s.s)%z),
6                pi((x.s.f*y.f.f+x.s.s*y.s.f)%z,
7                  (x.s.f*y.f.s+x.s.s*y.s.s)%z));
8    }
9    matrix square(matrix x, long long y){
10       return multiply(x, x, y);
11   }
12   matrix power(matrix x, long long y, long long z){
13       if(y == 1) return x;
14       if(y % 2 == 0) return square(power(x, y/2, z), z);
15       return multiply(x, square(power(x, y/2, z), z), z);
16   }
17   long long fibo(long long n, long long m){
18       matrix x = matrix(pi(1, 1), pi(1, 0));
19       return power(x, n, m).f.s;
20   }
```

4.6 $^{n}C_{k}$

```
1  if(k > n-k) k = n-k;
2  for(int i = 0; i < k; i++){
3      ans *= (n-i);
4      ans /= (i+1);
5  }
```

5 Algorithms

5.1 Discretisation

```
1  for(int i = 0; i < n; i++){
2      cin >> a[i];
3      b[i] = a[i];
4  }
5  sort(b, b+n);
6  for(int i = 0; i < n; i++){
7      d = lower_bound(b, b+n, a[i])-b;
8      a[i] = d+1;
9  }
```

5.2 Binary Search

```
1  while(mini < maxi){
2      medi = mini+(maxi-mini)/2;
3      if(can(medi)) maxi = medi;
4      else mini = medi+1;
5  }
```

5.3 Mo's Algorithm

```
1  //queries : index, left, right
2  //answers : index, answer
3
4  bool cmp(pii x, pii y){
5      if(x.second.first/blk != y.second.first/blk){
6          return x.second.first/blk < y.second.first/blk;
7      }
8      if(x.second.first/blk & 1){
9          return x.second.second < y.second.second;
10     }
11     return x.second.second > y.second.second;
12 }
13
14 blk = sqrt(n);
15 sort(qs, qs+q, cmp);
16 for(int i = 0; i < q; i++){
17     l = qs[i].second.first; r = qs[i].second.second;
18     while(lft > l){
19         //remove a[lft-1]
20         lft--;
21     }
22     while(rgt <= r){
23         //add a[rgt]
24         rgt++;
```

```
25        }
26        while(lft < l){
27            //add a[lft]
28            lft++;
29        }
30        while(rgt > r+1){
31            //remove a[rgt-1]
32            rgt--;
33        }
34        ans[i] = pi(qs[i].first, cur);
35    }
36  sort(ans, ans+q);
```

6 Miscellaneous

6.1 Macros + Functions + Variables

```
1  #define f first
2  #define s second
3  #define pb push_back
4  #define ins insert
5  int ls(int x){ return (x)&(-x); }
6  mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
7  // 4 Directions
8  int dx[]={0, 0, -1, 1};
9  int dy[]={-1, 1, 0, 0};
10 // 8 Directions
11 int dx[]={0, 0, -1, 1, -1, 1, -1, 1};
12 int dy[]={-1, 1, -1, 1, 0, 0, 1, -1};
13 // Knight Moves
14 int dx[]={-1, -2, 1, 2, 2, 1, -2, 1};
15 int dy[]={-2, -1, -2, -1, 1, 2, 1, 2};
```

6.2 Compile Commands

6.2.1 Compile

```
1  g++ -c "%f" -std=c++11
```

6.2.2 Build

```
1  g++ -o "%e" "%f" -std=c++11
```

6.2.3 Command Line

```
1  g++ "file.cpp" -o "file" -std=c++11
```

6.2.4 Simple Script

```
1  int main(int argc, char **argv){
2      system(g++ -o (argv[1]) (argv[1]).cpp -std=c++11);
3  }
```

6.3 Input/Output

6.3.1 Fast

```
1 ios_base::sync_with_stdio(false);
2 cin.tie(0);
```

6.3.2 Faster

```
1 inline int read_int() {
2     int x = 0;
3     char ch = getchar_unlocked();
4     while (ch < '0' || ch > '9') ch = getchar_unlocked();
5     while (ch >= '0' && ch <= '9'){
6         x = (x << 3) + (x << 1) + ch - '0';
7         ch = getchar_unlocked();
8     }
9     return x;
10 }
```

6.4 Pruning

```
1 auto start = chrono::high_resolution_clock::now();
2 auto end = chrono::high_resolution_clock::now();
3 auto elapse = chrono::duration<double>(end-start);
4 if(elapse.count() > 2.9) break;
```

6.5 Optimise

```
1 int __attribute__((optimize("Ofast"), target("arch=sandybridge"))) f(){}
```