Contents

1 Template

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll              long long
#define vll             vector<ll>
#define pll             pair<ll, ll>
#define vpll            vector<pll>
#define endl '\n'
#define all(xs)         xs.begin(), xs.end()
#define found(x, xs) (xs.find(x) != xs.end())
```

2 Algebra

2.1 All divisors

$O(\sqrt{n})$

```
vll divisors(ll n) {
    vll divs;
    for (ll i = 1; 1LL * i * i <= n; i++) {
        if (n % i == 0) {
            divs.push_back(i);
            if (i != n / i) {
                divs.push_back(n / i);
            }
        }
    }

    return divs;
}
```

2.2 Primality test

$O(\sqrt{n})$

```
bool isPrime(ll n)
{
    if(n!=2 && n % 2==0)
        return false;

    for(ll d=3; d*d <= n; d+=2)
    {
        if(n % d==0)
            return false;
    }

    return n >= 2;
}
```

2.3 Binary exponentiation

$O(\log n)$

```
ll binpow(ll a, ll b) {
    ll res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

2.4 Greatest common divisor

$O(\log \min(a, b))$

```
1  ll gcd (ll a, ll b) {
2      while (b) {
3          a %= b;
4          swap(a, b);
5      }
6      return a;
7  }
```

2.4.1 Least common multiple

```
1  ll lcm(ll a, ll b) {
2      return a / gcd(a, b) * b;
3      }
```

3 Graphs

3.1 DFS

$O(n + m)$

```
1  void dfs(ll at, ll n ,vpll adj[], bool visited[]) {
2      if(visited[at])
3          return;
4
5      visited[at] = true;
6
7      vpll neighbours = adj[at];
8      for(auto nex: neighbours)
9          dfs(nex.first, n, adj, visited);
10 }
```

3.2 BFS

$O(n + m)$

```
1  void bfs(ll s, ll n, vll adj[]) {
2      bool visited[n] = {0};
3      visited[s] = true;
4
5      queue<ll> q;
6      q.push(s);
7      while (!q.empty())
8      {
```

```
 9          vll neighbours = adj[q.front()];
10          for(auto nex: neighbours) {
11              if(!visited[nex]) {
12                  visited[nex]=true;
13                  q.push(nex);
14              }
15          }
16          cout << q.front() << '\n';
17          q.pop();
18      }
19  }
```

3.2.1   Shortest path on unweighted graph

$O(n+m)$

```
 1  vll solve(ll s, ll n, vll adj[]) {
 2      bool visited[n] = {0};
 3      visited[s] = true;
 4
 5      queue<ll> q;
 6      q.push(s);
 7      vll prev(n, -1);
 8      while (!q.empty())
 9      {
10          vll neighbours = adj[q.front()];
11          for(auto nex: neighbours) {
12              if(!visited[nex]) {
13                  visited[nex]=true;
14                  q.push(nex);
15                  prev[nex] = q.front();
16              }
17          }
18          q.pop();
19      }
20
21      return prev;
22  }
23
24  vll reconstructPath(ll s, ll e, vll prev) {
25      vll path;
26      for(ll i=e; i!=-1; i=prev[i])
27          path.push_back(i);
28
29      reverse(path.begin(), path.end());
30
31      if(path[0]==s)
32          return path;
33      else {
```

```
34        vll place;
35        return place;
36    }
37 }
38
39 vll bfs(ll s, ll e, ll n, vll adj[]) {
40     vll prev = solve(s, n, adj);
41
42     return reconstructPath(s, e, prev);
43 }
```

3.3   Flood Fill

$O(n+m)$

```
1  int dir_y[] = {};
2  int dir_x[] = {};
3
4  int ff(int i, int j, char c1, char c2) {
5      if ((i < 0) || (i >= n)) return 0;
6      if ((j < 0) || (j >= m)) return 0;
7      if (grid[i][j] != c1) return 0;
8
9      int ans = 1;
10     grid[i][j] = c2;
11
12     for (int d = 0; d < 8; ++d)
13         ans += floodfill(i+dir_y[d], j+dir_x[d], c1, c2);
14
15     return ans;
16 }
```

;

3.4   Topological Sort (Directed Acyclic Graph)

3.4.1   DFS Variation

$O(n+m)$

```
1  void dfs(ll at, ll n ,vpll adj[], bool visited[], vll &ts) {
2      if(visited[at])
3          return;
4
5      visited[at] = true;
6
7      vpll neighbours = adj[at];
8      for(auto nex: neighbours)
9          dfs(nex.first, n, adj, visited);
10     ts.push_back(at);                      // Only change
```

5

```
11  }
```

3.4.2   Kahn's Algorithm

```
1   priority_queue<ll, vll, greater<ll>> pq;
2   for(ll at=0; at<n; at++)              // Push all sources of
        connected components in graph
3       if(in_degree[at] == 0)
4           pq.push(at);
5
6   while(!pq.empty()) {
7       ll at = pq.top(); pq.pop();
8       vll neighbors = adj[at];
9       for(auto nex: neighbors) {
10          in_degree[nex]--;
11          if(in_degree[nex]>0) continue;
12          pq.push(nex);
13      }
14  }
```

3.5   Bipartite Graph Check (Undirected Graph)

$O(n+m)$

```
1   bool isBipartite(ll s, ll n, vll adj[]) {
2       queue<ll> q;
3       q.push(s);
4       vll color(n, -1); color[s]=0;
5       bool flag = true;
6       while (!q.empty())
7       {
8           vll neighbours = adj[q.front()];
9           for(auto nex: neighbours) {
10              if(color[nex] == -1) {
11                  color[nex] = 1-(color[q.front()]);
12                  q.push(nex);
13              }
14              else if(color[nex] == color[q.front()]) {
15                  flag = false;
16                  break;
17              }
18          }
19          q.pop();
20      }
21
22      return flag;
23  }
```

3.6 Cycle Check (Directed Graph)

$O(n+m)$

```
enum { UNVISITED = -1, VISITED = -2,  EXPLORED=-3};

void cycleCheck(ll at, ll n ,vll adj[], int visited[], ll
    dfs_parent[]) {
    visited[at] = EXPLORED;

    vll neighbours = adj[at];
    for(auto nex: neighbours) {
        if(visited[nex] == UNVISITED) {
            // Tree edges (part of the DFS spanning tree)
            dfs_parent[nex] = at;
            cycleCheck(nex, n, adj, visited);
        }
        else if(visited[nex] == EXPLORED) {
            if(nex == dfs_parent[at]) {
                // Trivial cycle
                // Do something
            }
            else {
                // Non trivial cycle - Back Edge ((u, v)
                    such that v is the ancestor of node u but
                     is not part of the DFS tree)
                // Do something
            }

        }
        else if(visited[nex] == VISITED) {
            // Forward/Cross edge ((u, v) such that v is a
                descendant but not part of the DFS tree)
            // Do something
        }

    }

    visited[at] = VISITED;
}
```

4 Math Formulas

4.1 Sum of an arithmetic progression

$S_n = \frac{n}{2}(a_1 + a_n)$

4.2   Permutation with repeated elements

$P_n = \frac{n!}{n_1! n_2! \ldots n_k!}$