Contents

1 Template

```
#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

$O(\sqrt{n})$

```cpp
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

$O(\sqrt{n})$

```cpp
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

$O(\log n)$

```cpp
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

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

```
ll gcd (ll a, ll b) {
  while (b) {
      a %= b;
      swap(a, b);
  }
  return a;
}
```

2.4.1 Least common multiple

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

3 Graphs

3.1 DFS

$O(n + m)$

```
void dfs(ll at, ll n ,vpll adj[], bool visited[]) {
    if(visited[at])
        return;

    visited[at] = true;

    vpll neighbours = adj[at];
    for(auto nex: neighbours)
        dfs(nex.first, n, adj, visited);
}
```

3.2 BFS

$O(n + m)$

```
void bfs(ll s, ll n, vll adj[]) {
    bool visited[n] = {0};
    visited[s] = true;
```

```cpp

    queue<ll> q;
    q.push(s);
    while (!q.empty())
    {
        vll neighbours = adj[q.front()];
        for(auto nex: neighbours) {
            if(!visited[nex]) {
                visited[nex]=true;
                q.push(nex);
            }
        }
        cout << q.front() << '\n';
        q.pop();
    }
}
```

3.2.1 Shortest path on unweighted graph

$O(n+m)$

```cpp
vll solve(ll s, ll n, vll adj[]) {
    bool visited[n] = {0};
    visited[s] = true;

    queue<ll> q;
    q.push(s);
    vll prev(n, -1);
    while (!q.empty())
    {
        vll neighbours = adj[q.front()];
        for(auto nex: neighbours) {
            if(!visited[nex]) {
                visited[nex]=true;
                q.push(nex);
                prev[nex] = q.front();
            }
        }
        q.pop();
    }

    return prev;
}

vll reconstructPath(ll s, ll e, vll prev) {
    vll path;
    for(ll i=e; i!=-1; i=prev[i])
        path.push_back(i);

```

```
    reverse(path.begin(), path.end());

    if(path[0]==s)
        return path;
    else {
        vll place;
        return place;
    }
}

vll bfs(ll s, ll e, ll n, vll adj[]) {
    vll prev = solve(s, n, adj);

    return reconstructPath(s, e, prev);
}
```

3.3 Flood Fill

$O(n+m)$

```
int dir_y[] = {};
int dir_x[] = {};

int ff(int i, int j, char c1, char c2) {
    if ((i < 0) || (i >= n)) return 0;
    if ((j < 0) || (j >= m)) return 0;
    if (grid[i][j] != c1) return 0;

    int ans = 1;
    grid[i][j] = c2;

    for (int d = 0; d < 8; ++d)
        ans += floodfill(i+dir_y[d], j+dir_x[d], c1, c2);

    return ans;
}
```

;

3.4 Topological Sort (Directed Acyclic Graph)

3.4.1 DFS Variation

$O(n+m)$

```
void dfs(ll at, ll n ,vpll adj[], bool visited[], vll &ts) {
    if(visited[at])
        return;

    visited[at] = true;
```

```

    vpll neighbours = adj[at];
    for(auto nex: neighbours)
        dfs(nex.first, n, adj, visited);
    ts.push_back(at);                           // Only change
}
```

3.4.2 Kahn's Algorithm

```
priority_queue<ll, vll, greater<ll>> pq;
for(ll at=0; at<n; at++)          // Push all sources of
    connected components in graph
    if(in_degree[at] == 0)
        pq.push(at);

while(!pq.empty()) {
    ll at = pq.top(); pq.pop();
    vll neighbors = adj[at];
    for(auto nex: neighbors) {
        in_degree[nex]--;
        if(in_degree[nex]>0) continue;
        pq.push(nex);
    }
}
```

3.5 Bipartite Graph Check (Undirected Graph)

$O(n+m)$

```
bool isBipartite(ll s, ll n, vll adj[]) {
    queue<ll> q;
    q.push(s);
    vll color(n, -1); color[s]=0;
    bool flag = true;
    while (!q.empty())
    {
        vll neighbours = adj[q.front()];
        for(auto nex: neighbours) {
            if(color[nex] == -1) {
                color[nex] = 1-(color[q.front()]);
                q.push(nex);
            }
            else if(color[nex] == color[q.front()]) {
                flag = false;
                break;
            }
        }
        q.pop();
```

```
    }

    return flag;
}
```

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

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

4.3 Check if is geometric progression

$a_i^2 = a_{i-1}a_{i+1}$

5 Facts

5.1 XOR

5.1.1 Self-inverse property

To cancel a XOR, you can XOR again the same value because $a \oplus a = 0$, so $(value \oplus a) \oplus a = value$

5.1.2 Identity element

$a \oplus 0 = a$

5.1.3 Commutative

$a \oplus b = b \oplus a$

5.1.4 Associative

$(a \oplus b) \oplus c = a \oplus (b \oplus c)$