

Contents

1	Template	1
2	Graphs	1
2.1	DFS	1
2.2	BFS	1
2.2.1	Shortest path on unweighted graph	2
2.3	Flood Fill	3
2.4	Topological Sort (Directed Acyclic Graph)	3
2.4.1	DFS Variation	3
2.4.2	Kahn's Algorithm	4
2.5	Bipartite Graph Check (Undirected Graph)	4
2.6	Cycle Check (Directed Graph)	5

1 Template

2 Graphs

2.1 DFS

$O(n + m)$

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

2.2 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 }

```

2.2.1 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 }

```

2.3 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 }

```

;

2.4 Topological Sort (Directed Acyclic Graph)

2.4.1 DFS Variation

$O(n + m)$

```

1  void dfs(ll at, ll n, vll adj[], bool visited[], vll &ts) {
2      if(visited[at])
3          return;
4
5      visited[at] = true;
6
7      vll neighbours = adj[at];
8      for(auto nex: neighbours)
9          dfs(nex.first, n, adj, visited);
10     ts.push_back(at);
11 }

```

2.4.2 Kahn's Algorithm

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

2.5 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   return flag;
22 }
23 }
```

2.6 Cycle Check (Directed Graph)

$O(n + m)$

```
1  enum { UNVISITED = -1, VISITED = -2,  EXPLORED=-3};
2
3  void cycleCheck(ll at, ll n ,vll adj[], int visited[], ll
4      dfs_parent[]) {
5      visited[at] = EXPLORED;
6
7      vll neighbours = adj[at];
8      for(auto nex: neighbours) {
9          if(visited[nex] == UNVISITED) {
10             // Tree edges (part of the DFS spanning tree)
11             dfs_parent[nex] = at;
12             cycleCheck(nex, n, adj, visited);
13         }
14         else if(visited[nex] == EXPLORED) {
15             if(nex == dfs_parent[at]) {
16                 // Trivial cycle
17                 // Do something
18             }
19             else {
20                 // Non trivial cycle - Back Edge ((u, v)
21                 // such that v is the ancestor of node u but
22                 // is not part of the DFS tree)
23                 // Do something
24             }
25         }
26         else if(visited[nex] == VISITED) {
27             // Forward/Cross edge ((u, v) such that v is a
28             // descendant but not part of the DFS tree)
29             // Do something
30         }
31     }
32     visited[at] = VISITED;
33 }
```