Cheatsheet

Jan

February 6, 2023

Contents

1 Dynamic Programming

1.1 Knapsack problem

```cpp
#include <iostream>
#include <vector>

int table[1001][1001];
int weight[1001];
int value[1001];

void knapsack()
{
    int N, W;
    std::cin >> N >> W;

    for(int i = 1; i <= N; i++) std::cin >> weight[i] >> value[i];

    for(int i = 0; i <= N; ++i)
    {
        for(int w = 0; w <= W; ++w)
        {
            if(i == 0 || w == 0) table[i][w] = 0;
            else if(weight[i] > w) table[i][w] = table[i - 1][w];
            else
            {
                table[i][w] = std::max(table[i-1][w], table[i-1][w-weight[i]] +
                    value[i]);
            }
        }
    }

    int result = table[N][W];

    int w = W;
    std::vector<int> stuff;
    for (int i = N; i > 0 && result > 0; i--)
```

```
    {
        if (result == table[i-1][w]) continue;
        else
        {
            stuff.push_back(i);
            result -= value[i];
            w -= weight[i];
        }
    }

    std::cout << table[N][W] << std::endl;
    std::cout << stuff.size() << std::endl;
    for(int i = 0; i < stuff.size(); i++) std::cout << stuff[i] << ' ';
    std::cout << std::endl;
}
```

1.2 Longest increasing subsequence

```
#include <vector>
#include <algorithm>

int table[100001];
int values[100001];

int long_inc_subseq(std::vector<int> nums)
{
        std::vector<int> dp;
        for (int i : nums)
    {
        // To have longest non-decreasing subsequence use std::upper_bound()
                int pos = std::lower_bound(dp.begin(), dp.end(), i) - dp.begin();
                if (pos == dp.size()) dp.push_back(i);
                else dp[pos] = i;
        }

        return dp.size();
}
```

2 Geometry

2.1 Polygons

```
#include <vector>

double area(std::vector<std::pair<int,int>> poly)
{
    double result = 0;
    for(int i = 0; i < poly.size()-1; i++) {
        result += poly[i].first*poly[i+1].second -
            poly[i+1].first*poly[i].second;
    }

    return result;
}
```

3 Graphs

3.1 Bellman-Ford

```
#include <vector>

struct Edge
{
    int target;
    int weight;
};

struct Node
{
    std::vector<Edge> adj;
    int dist = INT_MAX;
    int parent;
};

Node V[10001];

void bellmanford(int start, int N)
{
    V[start].dist = 0;
    for(int k = 0; k < N-1; k++) {
        for(int i = 0; i < N; i++) {
            for(Edge e : V[i].adj) {
                int new_dist = V[i].dist + e.weight;
                if(new_dist < V[e.target].dist) {
                    V[e.target].dist = new_dist;
```

```
                V[e.target].parent = i;
            }
        }
    }

    // To find negative cycles, run another iteration
    // if any distance change, negative cycle found
}
```

3.2 Condensation

```
#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>

struct Node
{
    std::vector<int> adj;
    std::vector<int> adj_rev;
    std::vector<int> adj_scc;
    bool used;
    bool in_scc_graph;
};

Node V[100001];
std::vector<int> order, component;

void dfs1(int v)
{
    V[v].used = true;
    for(int u : V[v].adj)
    {
        if(!V[u].used) dfs1(u);
    }

    order.push_back(v);
}

void dfs2(int v)
{
    V[v].used = true;
    component.push_back(v);

    for(int u : V[v].adj_rev)
    {
        if(!V[u].used) dfs2(u);
    }
}

void condensation(int n)
{
    for(int i = 1; i <= n; i++)
    {
        if(!V[i].used) dfs1(i);
    }

    for(int i = 1; i <= n; i++) V[i].used = false;

    std::reverse(order.begin(), order.end());

    std::vector<int> roots(n+1, 0);
    std::vector<int> root_nodes;

    for(int i : order)
    {
        if(!V[i].used)
        {
            dfs2(i);

            int root = component.front();
            for (int u : component) roots[u] = root;
            V[root].in_scc_graph = true;

            component.clear();
        }
    }

    for(int i = 1; i <= n; i++)
    {
        for(int u : V[i].adj)
        {
            int root_i = roots[i];
            int root_u = roots[u];

            if (root_u != root_i)
                V[root_i].adj_scc.push_back(root_u);
        }
    }
```

```
}

void solution()
{
    int n, m;
    std::cin >> n >> m;

    for(int i = 1; i <= n; i++)
    {
        V[i].adj.clear();
        V[i].adj_rev.clear();
        V[i].adj_scc.clear();
        V[i].used = false;
        V[i].in_scc_graph = false;
    }

    for(int i = 0; i < m; i++)
    {
        int x, y;
        std::cin >> x >> y;
        V[x].adj.push_back(y);
        V[y].adj_rev.push_back(x);
    }

    condensation(n);
}
```

3.3 DFS

```
#include <vector>
struct Node
{
    std::vector<int> adj;
    bool visited;
};

Node V[10001];

void dfs(int i)
{
    if(V[i].visited) return;
    V[i].visited = true;
    for(int k : V[i].adj) dfs(k);
}
```

3.4 Dijkstra

```
#include <vector>
#include <string>
#include <climits>
#include <queue>

struct Edge
{
    int target;
    int weight;
};

struct Node
{
    std::vector<Edge> adj;
    int dist = INT_MAX;
    int parent;
};

struct NodeDist
{
    int ind, dist;
    NodeDist(int i, int d)
    :ind(i),dist(d){}
};

bool operator<(NodeDist a, NodeDist b)
{
    return a.dist > b.dist;
}

Node V[10001];

void dijkstra(int start)
{
    std::priority_queue<NodeDist> Q;
    V[start].dist = 0;
    V[start].parent = -1;
    Q.push(NodeDist(start, 0));
    while (!Q.empty())
    {
        NodeDist nd = Q.top(); Q.pop();
        int k = nd.ind;
        int d = nd.dist;
        if(V[k].dist < d) continue;
```

```
        for(Edge e: V[k].adj)
        {
            int new_dist = d + e.weight;
            if(new_dist < V[e.target].dist)
            {
                V[e.target].dist = new_dist;
                V[e.target].parent = k;
                Q.push(NodeDist(e.target, new_dist));
            }
        }
    }
}
```

3.5 Floyd-Warshall

```
#include <utility>

int d[100][100];
int N, M;

void floydwarshall()
{
    for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                // Overflow possible
                d[i][j] = 1000000000;

    for(int i = 0; i < N; i++) d[i][i] = 0;

    // Set d[a][b] to weight of edge a--b
    for(int i = 0; i < M; i++)

    // Floyd-warshall
    for(int k = 0; k < N; k++)
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                d[i][j] = std::min(d[i][j], d[i][k] + d[k][j]);
}
```

3.6 Graph

```
#include <iostream>
#include <vector>

struct Edge
{
    int target;
    int weight;
};

struct Node
{
    std::vector<Edge> adj;
    int dist = INT_MAX;
    int parent;
};

Node V[10001];

void read_edge()
{
    int n, d, c;
    std::cin >> n >> d >> c;
    for(int i = 0; i < d; i++)
    {
        int a, b, s;
        std::cin >> a >> b >> s;
        V[b].adj.push_back({a, s});
    }
}
```

3.7 Maximum flow

```
#include <iostream>
#include <vector>

struct Edge
{
    int target;
    int capacity;
    int flow;
    Edge* back;

    Edge(int t, int c)
    :target(t), capacity(c){}
};
```

```
struct Node
{
    std::vector<Edge*> adj;
    bool visited = false;
    Edge* parent;
    int flow;
};

Node V[500];

int augment(int i, int t)
{
    if(V[i].visited) return 0;
    V[i].visited = true;
    if(i == t) return INT_MAX;
    for(Edge* e : V[i].adj) {
        if(e->capacity - e->flow <= 0) continue;
        int f = augment(e->target, t);
        if(f > 0) {
            f = std::min(f, e->capacity - e->flow);
            V[e->target].parent = e->back;
            return f;
        }
    }
    return 0;
}

int max_flow(int s, int t, int n)
{
    int total_flow = 0;
    for(int i = 0; i < n; i++)
    {
        for(Edge* e : V[i].adj) e->flow = 0;
    }

    while(true)
    {
        for(int i = 0; i < n; i++)
        {
            V[i].visited = false; V[i].parent = nullptr;
        }

        int flow = augment(s, t);
        if(flow == 0) break;
        total_flow += flow;
        int x = t;
        while(x != s)
        {
            V[x].parent->flow -= flow;
            V[x].parent->back->flow += flow;
            x = V[x].parent->target;
        }
    }

    return total_flow;
}

int main()
{
    int n, m, s, t;
    std::cin >> n >> m >> s >> t;

    for(int i = 0; i < n; i++)
    {
        V[i].adj.clear();
        V[i].visited = false;
    }

    for(int i = 0; i < m; i++)
    {
        int a, b, c;
        std::cin >> a >> b >> c;
        Edge* e1 = new Edge(b, c);
        // Zero if directed
        Edge* e2 = new Edge(a, 0);
        e1->back = e2;
        e2->back = e1;
        V[a].adj.push_back(e1);
        V[b].adj.push_back(e2);
    }

    std::cout << max_flow(s, t, n);

    return 0;
}
```

3.8 Minimum path cover

```
#include <iostream>
```

```cpp
#include <vector>

struct Node
{
    std::vector<int> adj;
    bool visited = false;
    int match = -1;
};

Node A[100001];
Node B[100001];

bool augment(int i, int n)
{
    if(A[i].visited) return false;
    A[i].visited = true;
    for(int j : A[i].adj)
    {
        if(B[j].match == -1 || augment(B[j].match, n))
        {
            B[j].match = i;
            return true;
        }
    }

    return false;
}

int matching(int n)
{
    for(int i = 1; i <= n; i++) B[i].match = -1;
    int M = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= n; j++) A[j].visited = false;
        if(augment(i, n)) M++;
    }

    return M;
}

void solution()
{
    int n, m;
    std::cin >> n >> m;

    for(int i = 1; i <= n; i++) A[i].adj.clear();

    for(int i = 0; i < m; i++)
    {
        int x, y;
        std::cin >> x >> y;
        A[x].adj.push_back(y);
    }

    std::cout << n - matching(n) << std::endl;
}
```

3.9 Prim

```cpp
#include <vector>
#include <queue>

struct Edge
{
    int target;
    int weight;
};

struct Node
{
    std::vector<Edge> adj;
    bool in_tree;
};

struct NodeWeight
{
    int i, w, p;
    NodeWeight(int i, int w, int p)
    :i(i),w(w),p(p){}
};

bool operator<(NodeWeight a, NodeWeight b)
{
    return a.w > b.w; // reverse for min-heap
}

Node V[10001];

void prim(int N)
```

```
{
    std::vector<NodeWeight> L;
    std::priority_queue<NodeWeight> Q;
    for(int i = 0; i < N; i++) Q.push({i, INT_MAX, -1});

    while(!Q.empty())
    {
        NodeWeight nw = Q.top(); Q.pop();
        int k = nw.i;
        if(V[k].in_tree) continue;
        V[k].in_tree = true;
        L.push_back(nw);
        for(Edge e : V[k].adj)
        {
            if(V[e.target].in_tree) continue;
            Q.push({e.target, e.weight, k});
        }
    }
}
```

3.10 TSP

```
#include <iostream>
#include <vector>
#include <string>
#include <climits>
#include <queue>
#include <cmath>

struct Edge
{
    int target;
    int weight;
};

struct Node
{
    double x, y;
    std::vector<Edge> adj;
    std::vector<int> children;
    bool in_tree = false;
    bool visited = false;
};

struct NodeWeight
{
    int i, w, p;
    NodeWeight(int i, int w, int p)
    :i(i),w(w),p(p){}
};

bool operator<(NodeWeight a, NodeWeight b)
{
    return a.w > b.w;
}

Node V[1000];

std::vector<NodeWeight> prim(int N)
{
    std::vector<NodeWeight> L;
    std::priority_queue<NodeWeight> Q;
    Q.push({0, INT_MAX, -1});

    while(!Q.empty())
    {
        NodeWeight nw = Q.top(); Q.pop();
        int k = nw.i;
        if(V[k].in_tree) continue;
        V[k].in_tree = true;
        if(nw.p != -1) V[nw.p].children.push_back(nw.i);
        L.push_back(nw);
        for(Edge e : V[k].adj)
        {
            if(V[e.target].in_tree) continue;
            Q.push({e.target, e.weight, k});
        }
    }

    return L;
}

void dfs(int start)
{
    if(!V[start].visited) std::cout << start << std::endl;
    V[start].visited = true;
    for(int i : V[start].children) if(!V[i].visited) dfs(i);
}

int main()
```

```
{
    int N;
    std::cin >> N;

    for(int i = 0; i < N; i++) std::cin >> V[i].x >> V[i].y;

    for(int i = 0; i < N; i++)
    {
        for(int j = i + 1; j < N; j++)
        {
            int dist = std::round(std::sqrt((V[i].x-V[j].x) * (V[i].x-V[j].x) +
                (V[i].y-V[j].y) * (V[i].y-V[j].y)));
            V[i].adj.push_back({j, dist});
            V[j].adj.push_back({i, dist});
        }
    }

    std::vector<NodeWeight> MST = prim(N);
    dfs(0);

    return 0;
}
```

4 Math

4.1 Formulas

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

$$a^{\varphi(n)} \equiv 1 \mod n$$

$$a^{-1} \equiv a^{\varphi(n)-1} \mod n$$

5 Template

```
#include <iostream>

void solution()
{

}

int main()
{
    int count;
    std::cin >> count;
    while(count--) solution();
    return 0;
}
```