**\*BackTracking //O(N!)**

class Permute{

int \*p,

int N;

public:

Permute(int l[], int n) : p(l), N(n) { heaps(N-1); simple(N-1); }

void heaps(int N) {

if(N == 0) {

print();

return;

}

for(int i = 0; i <= N; i++) {

heaps(N – 1);

swap( ( N % 2 != 0 ? p[0] : p[i] ), p[N]);

}

}

void simple(int N) {

if(N == 0) {

print();

return;

}

for(int i = 0; i <= N; i++) {

swap(p[i], p[N]);

simple(N – 1);

swap(p[i], p[N]);

}

}

};

**\*Magic Square**

class MagicSquare{

int \*s, n, rowSum;

public:

MagicSquare(int n) : n(n) , s(new int[n\*n]) {

for(inti = 0; i < n\*n; i++)

s[i] = i+1;

rowSum = n\* (n\*n + 1) / 2;

generate(n\*n – 1);

}

**//O(N!) = O((n^2)!)**

void generate(int N) {

if(N == 0) {

check();

return;

}

for(inti = 0; i <= N; i++) {

generate(N – 1);

swap( N%2 != 0 ? s[0] : s[i], s[N]);

}

}

//O(n^2)

void check() {

//0 1 2

int c = 0;

for(inti = 0; i < n; i++) {

int sum = 0;

for(int j = 0; j < n; j++, c++)

sum += s[c];

if(sum != rowSum)

return;

}

//0 n 2n

for(inti = 0; i < n; i++) {

int c = i, sum = 0;

for(int j = 0; j < n; j++, c+= n)

sum += s[c];

if(sum != rowSum)

return;

}

//diagonal

int d1 = 0, d2 = 0;

for(inti = 0; i < n; i++) {

d1 += s[i \* n + i];

d2 += s[i \* n + n-1 – i];

}

if(d1 != rowSum || d2 != rowSum)

return;

print();

}

};

**\*N Queen //剪枝不剪枝时间复杂度O(n^n) 与O(n!)**

int Nqueen(int n) {

int res =0;

vector<int> pos(n, -1);

helper(n, pos, 0, res);

return res;

}

void helper(int n, vector<int>& pos, int row, int& res) {

if(row == n) {res++; return;}

int col;

for(col = 0; col < n; col++) {

bool flag = true;

for(int queen = 0; queen < row; queen++) {

if(pos[queen] == col || pos[queen] – queen == col – row || pos[queen] + queen == col + row) {

flag = false;

break;

}

}

if(flag) {

pos[row] = col;

helper(n, pos, row + 1, res);

}

}

}

**\*sudoku //剪枝不剪枝时间复杂度O(n^(n^2)) 与O((n!)^n)**

**\*DFS Time Complexity: O(V+E) ~ O(V^2)**

void enterDFS(int v){

vector<bool> visited(V, false);

DFS(v, visited);

}

DFS(int v, vector<bool> visited) {

visited[v] = true;

cout << v << “ “;

list<int>::iterator i;

for(i = adj[v].begin(); i!= adj[v].end(); i++) {

if(!visited[\*i])

DFS(\*i, visited);

}

}

//O(V^2)

void iterativeDFS(int v) {

vector<bool> visited(V, false); //O(V)

stack<int> s;

s.push(v); //O(1) 放循环里面会变成 V次赋值

visited[v] = true;

while(!s.empty()) { //O(V)

v = s.top(); s.pop();

cout << v << “ “;

list<int>::iterator i;

for(i = adj[v].begin(); i!= adj[v].end(); i++) { //O(V-1)

if(!visited[\*i])

s.push(\*i);

visited[\*i] = true;

}

}

}

**\*BFS Time Complexity: O(V+E) ~ O(V^2)**

void iterativeBFS(int v) {

vector<bool> visited(V, false);

queue<int> q;

q.push(v);

visited[v] = true;

while(!q.empty()) {

v = q.front(); q.pop();

cout << v << “ “;

list<int>::iterator i;

for(i = adj[v].begin(); i!= adj[v].end(); i++) {

if(!visited[\*i])

q.push(\*i);

visited[\*i] = true;

}

}

}

**\*isConnected //O(V^2)**

bool isConnected(int v) {

vector<bool> visited(V, false);

queue<int> q;

q.push(v);

visited[v] = true;

int count = 1;

while(!q.empty()) {

v = q.front(); q.pop();

cout << v << “ “;

list<int>::iterator i;

for(i = adj[v].begin(); i!= adj[v].end(); i++) {

if(!visited[\*i])

q.push(\*i);

visited[\*i] = true;

count ++;

}

}

return count == V;

}

\***Bellman-Ford: what is the cheapest way to get to V2?**

**// O(VE)** space O(V)

// all shortest path is O(v^2 \* E) = O(V^4)

void BellmanFord(int start) {

vector<int> cost(V, 1e8);

cost[start] = 0;

for(inti = 0; i < V – 1; i++) {

for (auto e = Es.begin(); e != Es.end(); e++) {

int u = e.first;

int v = e.second;

int weight = e.weight;

if(cost[v] > cost[u] + weight)

cost[v] = cost[u] + weight;

}

}

}

**\*Floyd-Warshall: find the cheapest way to get from anywhere to anywhere**

Bellman-Ford and Floyd-Warshall with path tracking

//Time complexity - O(v^3)

//Space complexity - O(V^2)

void Floyd-Warshall(int g[][V]) {

int dist[][V], i, j, k;

int path[][V];

for(i = 0; i < V; i++) {

for (j = 0; j < V; j++) {

dist[i][j] = g[i][j];

if(dist[i][j] != INF && i != j)

path[i][j] = j;

else

path[i][j] = -1;

}

}

for(k = 0; k < V; k++) {

for(i = 0; i < V; i++) {

for(j = 0; j < V; j++) {

if(dist[i][j] > dist[i][k] + dist[k][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

path[i][j] = path[i][k];

}

}

}

}

}

void printPath(int u, int v, int path[][V]) {

vector<int> ans;

if(path[u][v] == -1) {

cout << “[]”; return;

}

cout << u << “ “;

while(u != v) {

u = path[u][v];

ans.push\_back(u);

}

for(auto i : ans)

cout << i << “ ”;

cout << endl;

}

**\*Prim (find the minimum spanning tree)**

//Space complexity - O(E + V)

//Time complexity - **O(ElogV)**

void prim() {

priority\_queue<pair<int, int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

int src = 0;

vector<int> parent(V, -1)

vector<int> key(V, INF);

vector<bool> inMST(V, false);

pq.push({0, src});

key[src] = 0;

while(!pq.empty()) {

int u = pq.top().second; pq.pop();

inMST[u] = true;

list<pair<int,int>>::iterator it;

for(it = adj[u].begin(); it!= adj[u].end(); it++) {

int v = \*it.first;

int weigh = \*it.second;

if(inMST[v] == false && key[v] > weight) {

key[v] = weight;

pq.push({key[v], v});

parent[v] = u;

}

}

}

for(inti = 1; i < V; i++)

cout << parent[i] << “—“ << i << endl;

}

**\*Kruskal (find the minimum spanning tree)**

//Space complexity - O(E + V)

//Time complexity - **O(ElogV)**

int kruskal() {

int res = 0; //min-spanning tree weight

sort(edges.begin(), edges.end());

DisjointSets ds(V);

vector<pair<int, pair<int, int>>::iterator it;

for(it = edges.begin(); it!= edges.end(); it++) {

int u = it->second.first;

int v = it->second.second;

int setU = ds.find(u);

int setV = ds.find(v);

if(setU != setV) {

cout << u << “—“ << v << endl;

res += it->first;

ds.merge(setU, setV);

}

}

return res;

}