Luis Enrique Franco Marín

410921353

1. Traveling Sales person

- The ant colony optimization algorithm is a probabilistic method for resolving issues that can be reduced to finding good pathways via graphs. Multi-agent approaches inspired by the behavior of actual ants are referred to as artificial ants. Biological ants use pheromone-based communication as their primary mode of communication.

  Dynamic Programming is primarily a recursion optimization. We can use Dynamic Programming to optimize any recursive solution that involves repeated calls for the same inputs. The goal is to simply save the results of subproblems so that we don't have to recalculate them later. The temporal complexity of this simple optimization is reduced from exponential to polynomial.

  Branch and bound is a method for solving discrete and combinatorial optimization issues, as well as mathematical optimization problems. The set of candidate solutions is thought of as forming a rooted tree with the entire set at the root in a branch-and-bound method, which uses state space search to systematically enumerate them. The algorithm investigates the tree's branches, which are subsets of the solution set. Before enumerating a branch's candidate solutions, it's tested against upper and lower estimated bounds on the optimal solution, and it's eliminated if it can't yield a better solution than the algorithm's best so far.

- Pseudocodes

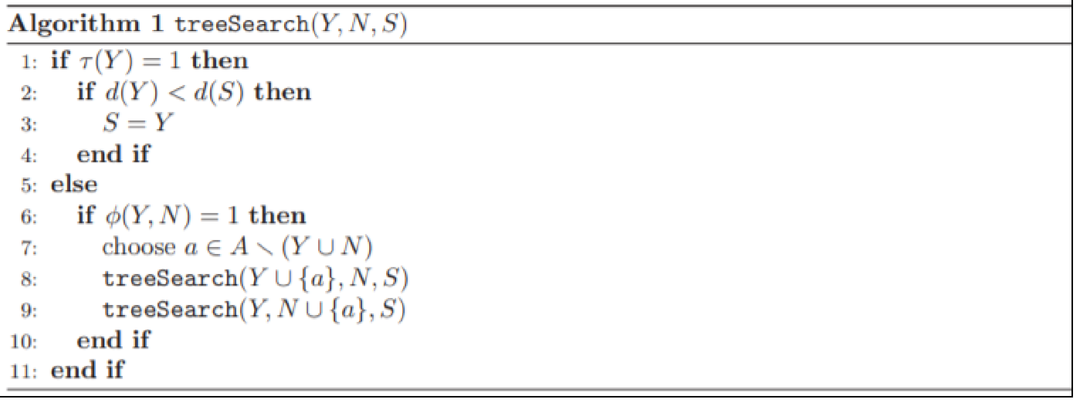
ACO

```
Initialize
For t=1 to iteration number do
    For k=1 to l do
        Repeat until ant k has completed a tour
            Select the city j to be visited next
            With probability $p_{ij}$ given by
            $$p_{ij}^{k} = \begin{cases} \frac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{s \in allowed_k} [\tau_{is}]^{\alpha} \cdot [\eta_{is}]^{\beta}} & j \in allowed_k \\ 0 & \text{otherwise} \end{cases}$$
        Calculate $L_k$
    Update the trail levels according to
    $\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$

    $\Delta\tau_{ij} = \sum_{k=1}^{l} \Delta\tau_{ij}^{k}$

    $$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ travels on edge } (i,j) \\ 0 & \text{otherwise} \end{cases}$$
End
```

DP

```
Algorithm 1: Dynamic Approach for TSP
  Data: $s$: starting point; $N$: a subset of input cities; $dist()$:
        distance among the cities
  Result: $Cost$ : TSP result
  $Visited[N] = 0$;
  $Cost = 0$;
  Procedure TSP($N$, $s$)
    $Visited[s] = 1$;
    if $|N| = 2$ and $k \neq s$ then
      $Cost(N, k) = dist(s, k)$;
      Return Cost;
    else
      for $j \in N$ do
        for $i \in N$ and $visited[i] = 0$ do
          if $j \neq i$ and $j \neq s$ then
            $Cost(N, j) = \min ( TSP(N - \{i\}, j) + dist(j, i))$
            $Visited[j] = 1$;
          end
        end
      end
    end
    Return $Cost$;
  end
```

Branch & Bound



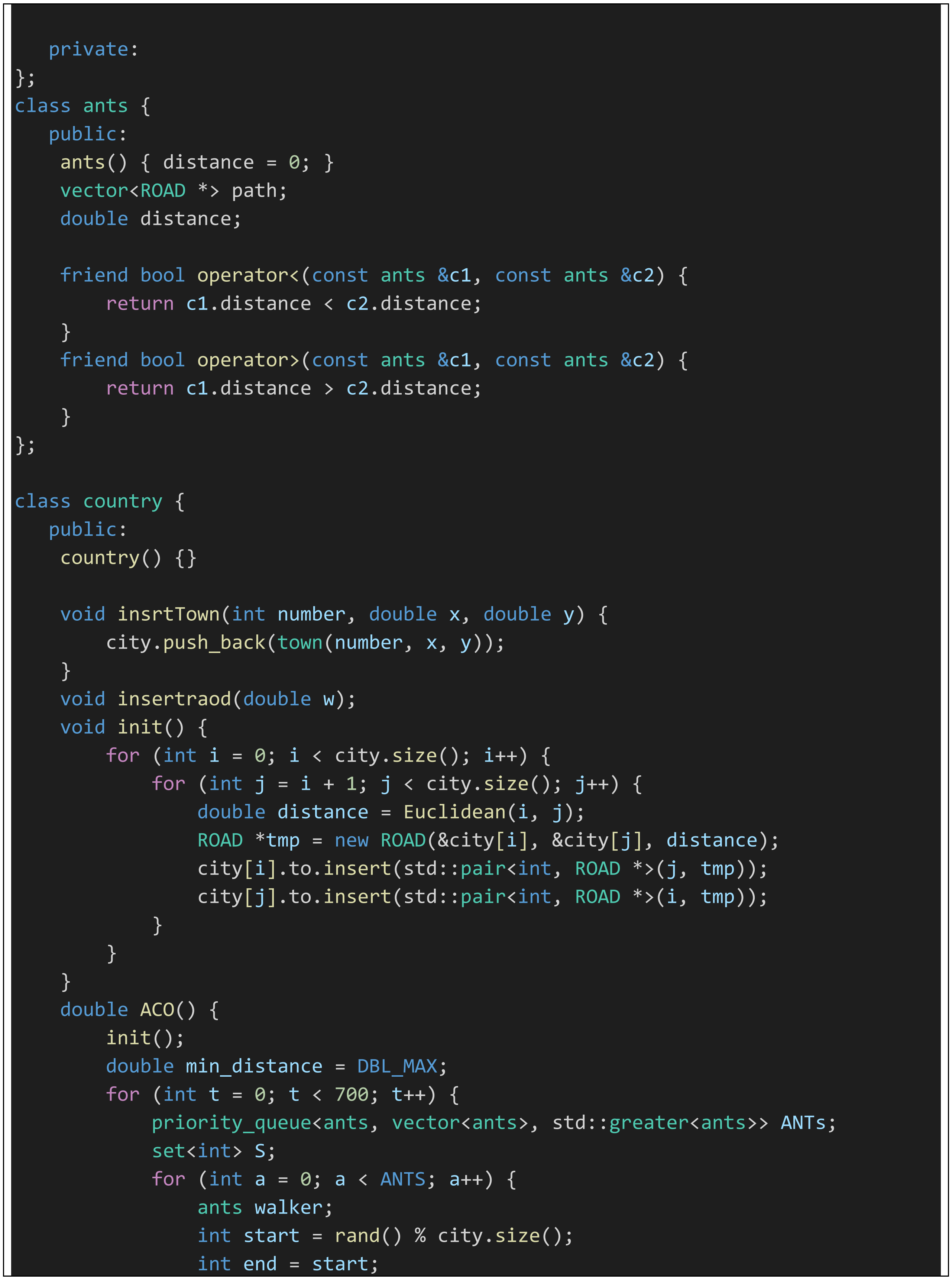
```
Algorithm 1 treeSearch($Y, N, S$)
if $\tau(Y) = 1$ then
    if $d(Y) < d(S)$ then
        $S = Y$
    end if
else
    if $\phi(Y, N) = 1$ then
        choose $a \in A \setminus (Y \cup N)$
        treeSearch($Y \cup \{a\}, N, S$)
        treeSearch($Y, N \cup \{a\}, S$)
    end if
end if
```

- Complexity analysis

Complexity	ACO	DP	B&B
Space	$O(n^2)$	$O(n^2)$	$O(n^2)$
Time	$O(n^3 \log n)$	$O(n^2 2^n)$	$O(n^2 2^n)$

- Codes:
  AOC:

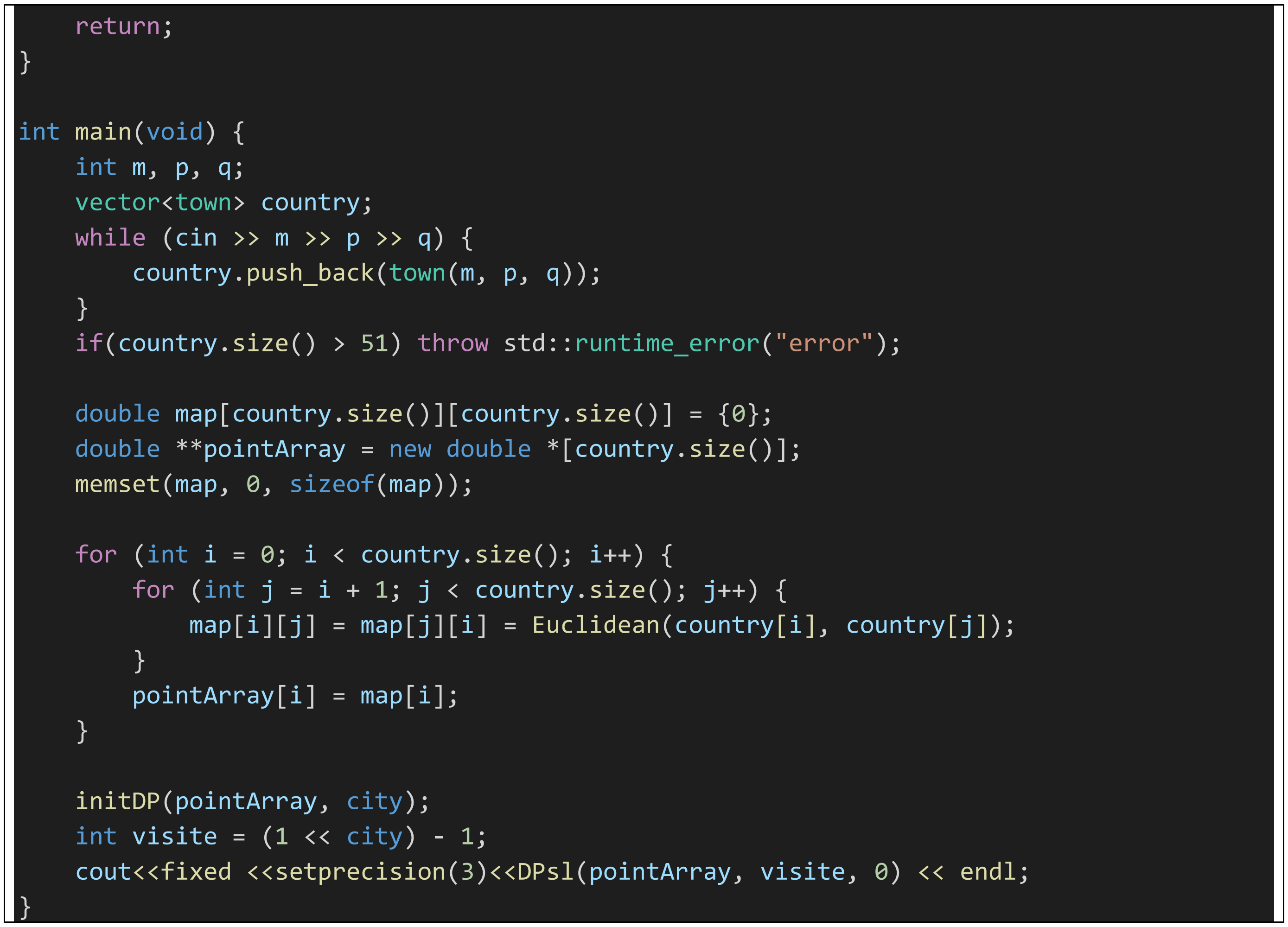
```cpp
#include <bits/stdc++.h>

using namespace std;

#define INF 0x3f3f3f3f
#define ALPHA 2
#define BETA 5
#define RHO 0.7
#define ANTS 5
#define DBL_MAX 1.7976931348623158e+308
#define DBL_MIN 2.2250738585072014e-308

double randFloat(double lowerbound, double upperbound);

class ROAD {
   public:
    ROAD() {}
    ROAD(class town *C, class town *D, double w) {
        this->C = C;
        this->D = D;
        this->w = w;
        this->re_w = 1 / w;
        this->Pheromones = 1000;
        this->score = re_w * Pheromones;
    }

    class town *C, *D;
    double w;
    double re_w, Pheromones, score;
};

class town {
   public:
    town(){};
    town(int n, double x, double y) {
        this->number = n;
        this->x = x;
        this->y = y;
    }

    double x, y;
    int number;
    unordered_map<int, ROAD *> to;
```



```
   private:
};
class ants {
   public:
    ants() { distance = 0; }
    vector<ROAD *> path;
    double distance;

    friend bool operator<(const ants &c1, const ants &c2) {
        return c1.distance < c2.distance;
    }
    friend bool operator>(const ants &c1, const ants &c2) {
        return c1.distance > c2.distance;
    }
};

class country {
   public:
    country() {}

    void insrtTown(int number, double x, double y) {
        city.push_back(town(number, x, y));
    }
    void insertraod(double w);
    void init() {
        for (int i = 0; i < city.size(); i++) {
            for (int j = i + 1; j < city.size(); j++) {
                double distance = Euclidean(i, j);
                ROAD *tmp = new ROAD(&city[i], &city[j], distance);
                city[i].to.insert(std::pair<int, ROAD *>(j, tmp));
                city[j].to.insert(std::pair<int, ROAD *>(i, tmp));
            }
        }
    }
    double ACO() {
        init();
        double min_distance = DBL_MAX;
        for (int t = 0; t < 700; t++) {
            priority_queue<ants, vector<ants>, std::greater<ants>> ANTs;
            set<int> S;
            for (int a = 0; a < ANTS; a++) {
                ants walker;
                int start = rand() % city.size();
                int end = start;
                for (int i = 0; i < city.size(); i++) {
                    S.insert(start);
                    int next = Roulette(start, S);
                    if (next == -1) break;

                    walker.distance += city[start].to[next]->w;
                    walker.path.push_back(city[start].to[next]);

                    start = next;
                }

                walker.distance += city[start].to[end]->w;
                walker.path.push_back(city[start].to[end]);

                S.clear();
                ANTs.push(walker);
            }

            if (ANTs.size() && min_distance > ANTs.top().distance) {
                min_distance = ANTs.top().distance;  // find min distance;
            }

            for (int i = 0; i < city.size(); i++) {
                for (int j = i + 1; j < city.size(); j++) {
                    city[i].to[j]->Pheromones *= RHO;
                }
            }

            for (int a = 0; a < ANTs.size(); a++) {
                double dis = 1 / ANTs.top().distance;
                for (int i = 0; i < ANTs.top().path.size(); i++) {
                    ANTs.top().path[i]->Pheromones += dis;
                }
                ANTs.pop();
            }

            for (int i = 0; i < city.size(); i++) {
                for (int j = i + 1; j < city.size(); j++) {
                    double tmp = ALPHA * city[i].to[j]->re_w * BETA *
city[i].to[j]->Pheromones;
                    if (tmp == 0) tmp = DBL_MIN;
                    city[i].to[j]->score = tmp;
                }
            }
        }
        return min_distance;
    }
    double Euclidean(int from, int to) {
        return sqrt((city[from].x - city[to].x) * (city[from].x - city[to].x) +
(city[from].y - city[to].y) * (city[from].y - city[to].y));
    }
    void print() {
        cout << "  ";
        for (int i = 0; i < city.size(); i++) {
            cout << setw(6) << i;
        }
        cout << endl;
        for (int i = 0; i < city.size(); i++) {
            cout << i << " ";
            for (int j = 0; j < city.size(); j++) {
                if (i == j) {
                    cout << setw(6) << 0;
                    continue;
                }
                cout.precision(4);
                cout << setw(6) << city[i].to[j]->w;
            }
            cout << endl;
        }
    }
    void printpath(ants);
    void clear() {
        for (int i = 0; i < city.size(); i++) {
            for (int j = i + 1; j < city.size(); j++) {
                delete city[i].to[j];
            }
        }
        city.clear();
        return;
    }

    vector<town> city;
    int Roulette(int start, set<int> S) {
        double wheel = GetWheel(start, S);
        wheel = randFloat(0, wheel);
        int next = -1;
        for (int i = 0; i < city.size() && wheel >= 0; i++) {
            if (S.count(i)) continue;
            wheel -= city[start].to[i]->score;
            next = i;
        }
        return next;
    }
    double GetWheel(int start, set<int> S) {
        double wheel = 0;
        for (int i = 0; i < city.size(); i++) {
            if (S.count(i)) continue;
            wheel += city[start].to[i]->score;
        }
        return wheel;
    }
};


double randFloat(double lowerbound, double upperbound) {
    return double(rand()) / (RAND_MAX) * (upperbound - lowerbound) + lowerbound;
}

int main(void) {

    int n, t = 0;
    double x, y;
    country thisCountry;
    double ans = 0;

    while (cin >> n >> x >> y) {
        thisCountry.insrtTown(n, x, y);
    }

    cout << thisCountry.ACO() << endl;

    return 0;
}
```

DP:

```cpp
#include <bits/stdc++.h>

using namespace std;
#define city 20
#define INF 0x3f3f3f3f
#define DBL_MAX 1.7976931348623158e+308

double DP[(1 << city)][city];

struct town {
    town(int num, int p, int q) {
        this->num = num;
        this->p = p;
        this->q = q;
    }
    int num, p, q;
};

double DPsl(double **map, int status, int at) {
    if (DP[status][at] != -1) {
        return DP[status][at];
    }
    int from = 1 << at;
    DP[status][at] = DBL_MAX;
    for (int i = 0; i < city; i++) {
        if (!(1 << i & status) || i == at) continue;
        DP[status][at] = min(DP[status][at], DPsl(map, status - from, i) + map[at][i]);
    }
    return DP[status][at];
}

double Euclidean(town from, town to) {
    return sqrt((from.p - to.p) * (from.p - to.p) + (from.q - to.q) * (from.q - to.q));
}

void initDP(double **map, int size) {
    for (int i = 0; i < 1 << city; i++) {
        for (int j = 0; j < city; j++) {
            DP[i][j] = -1;
        }
    }
    for (int i = 0; i < size; i++) {
        DP[1 << i][i] = map[0][i];
    }
```



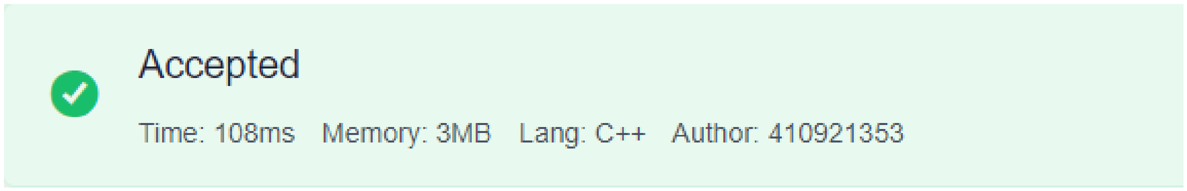
```
    return;
}

int main(void) {
    int m, p, q;
    vector<town> country;
    while (cin >> m >> p >> q) {
        country.push_back(town(m, p, q));
    }
    if(country.size() > 51) throw std::runtime_error("error");

    double map[country.size()][country.size()] = {0};
    double **pointArray = new double *[country.size()];
    memset(map, 0, sizeof(map));

    for (int i = 0; i < country.size(); i++) {
        for (int j = i + 1; j < country.size(); j++) {
            map[i][j] = map[j][i] = Euclidean(country[i], country[j]);
        }
        pointArray[i] = map[i];
    }

    initDP(pointArray, city);
    int visite = (1 << city) - 1;
    cout<<fixed <<setprecision(3)<<DPsl(pointArray, visite, 0) << endl;
}
```

Branch & Bound:

```
#include <bits/stdc++.h>

using namespace std;

const int N = 20;
int final_path[N + 1];
bool visited[N];
double solution = INT_MAX;

struct coordinate {
    int x;
    int y;
};

double firstMin(double adj[N][N], int i) {
    double min = INT_MAX;
    for (int j = 0; j < N; j++)
        if (adj[i][j] < min && i != j)
            min = adj[i][j];

    return min;
}

double secndMin(double adj[N][N], int i) {
    double first = INT_MAX;
    double second = INT_MAX;

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

        if (adj[i][j] <= first) {
            second = first;
            first = adj[i][j];
        }

        else if (adj[i][j] <= second && adj[i][j] != first)
            second = adj[i][j];
    }

    return second;
}

void recordTsp(double adj[N][N], double bound1, double curW, int level, int curPath[]) {
    if (level == N) {
        if (adj[curPath[level - 1]][curPath[0]] != 0) {
            double curSol = curW + adj[curPath[level - 1]][curPath[0]];

            if (curSol < solution)
                solution = curSol;
        }
        return;
    }

    for (int i = 0; i < N; i++) {
        if (adj[curPath[level - 1]][i] != 0 && visited[i] == false) {
            double tmp = bound1;
            curW += adj[curPath[level - 1]][i];

            if (level == 1)
                bound1 -= ((firstMin(adj, curPath[level - 1]) + firstMin(adj, i)) / 2);

            else
                bound1 -= ((secndMin(adj, curPath[level - 1]) + firstMin(adj, i)) / 2);

            if (bound1 + curW < solution) {
                curPath[level] = i;
                visited[i] = true;

                recordTsp(adj, bound1, curW, level + 1, curPath);
            }

            curW -= adj[curPath[level - 1]][i];
            bound1 = tmp;

            memset(visited, false, sizeof(visited));
            for (int j = 0; j <= level - 1; j++)
                visited[curPath[j]] = true;
        }
    }
}

void TSP(double adj[N][N]) {
    int curPath[N + 1];

    int bound1 = 0;
    memset(curPath, -1, sizeof(curPath));
    memset(visited, 0, sizeof(visited));
    for (int i = 0; i < N; i++)
        bound1 += (firstMin(adj, i) + secndMin(adj, i));

    bound1 = (bound1 & 1) ? bound1 / 2 + 1 : bound1 / 2;

    visited[0] = true;
    curPath[0] = 0;

    recordTsp(adj, bound1, 0, 1, curPath);
}

double distance(int x1, int y1, int x2, int y2) {
    return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2) * 1.0);
}

int main() {
    coordinate coorList[N];

    int n, x, y;
    for (int i = 0; i < N; i++) {
        scanf("%d%d%d", &n, &x, &y);
        coorList[i].x = x;
        coorList[i].y = y;
    }

    double adj[N][N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i == j)
                adj[i][j] = 0;

            else
                adj[i][j] = distance(coorList[i].x, coorList[i].y, coorList[j].x, 
coorList[j].y);
        }
    }

    TSP(adj);
    printf("%.3lf\n", solution);

    return 0;
}
```

- Results:

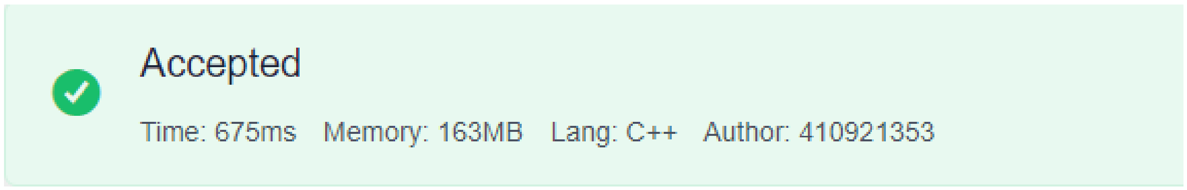
AOC:



Accepted

Time: 108ms Memory: 3MB Lang: C++ Author: 410921353

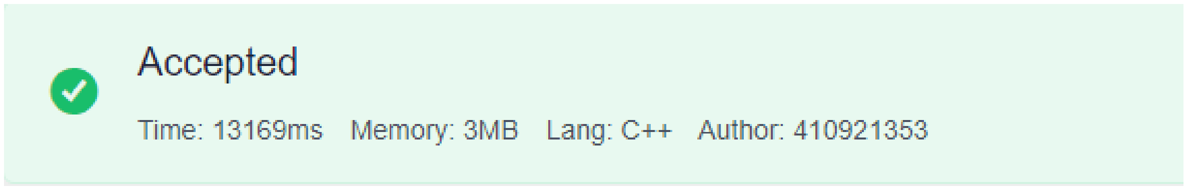
DP:



Accepted

Time: 675ms Memory: 163MB Lang: C++ Author: 410921353

Branch & Bound:



Accepted

Time: 13169ms Memory: 3MB Lang: C++ Author: 410921353

The **Ant colony optimization** clearly shows how the probabilistic approach yields a much more efficient solution.

2. F

3. Given a map, there are minefields and safe areas on the map. Find the number of minefields on the map. If there are no mines on the map, 0 is returned.

- Explain the design concept for the proposed algorithm. **(5%)**

The algorithm uses Depth first search to find the connected cells to the cell currently being tested. It uses recursion for all the cells that are connected and turn them into "safe places" with a 0 so they may no longer be iterated over. This ensures that while although using recursion, the graph is being iterated only one time.

- Pseudocode of your algorithms. **(5%)**

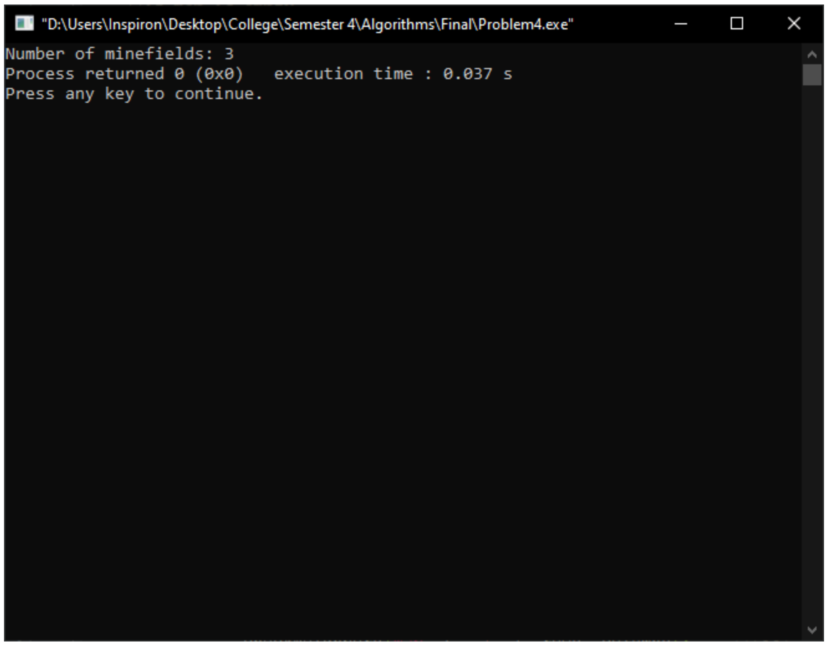
```
Initialize the map
Initialize count to 0
Iterate over rows
    Iterate over cells
        If a '1' is found
            Turn it into a 0
            Depth first search connected 1's in the 4 cardinal directions
            turn the 1's connected into a 0
            count++
            continue iteration
return count
```

- Implement algorithms for the problem. Please upload your code, output to e-learning **(10%)**

```
// C++Program to count islands in boolean 2D matrix
#include <bits/stdc++.h>
using namespace std;


void checkNeighbors(vector<vector<int>> &map, int i, int j, int rows, int
columns)
{
   //if map is empty
   if (i < 0 || j < 0 || i > (rows - 1) || j > (columns - 1) || map[i][j] != 1)
   {
           return;
```

```
    }
    //Checks the 4 cardinal directions
    if (map[i][j] == 1)
    {
            map[i][j] = 0;
            checkNeighbors(map, i + 1, j, rows, columns);      //RIGHT
            checkNeighbors(map, i - 1, j, rows, columns);      //LEFT
            checkNeighbors(map, i, j + 1, rows, columns);      //UP
            checkNeighbors(map, i, j - 1, rows, columns);      //DOWN
    }
}

int countMinefields(vector<vector<int>> &map)
{
    int rows = map.size();
    int columns = map[0].size();
    int count = 0;
    for (int i = 0; i < rows; i++)
    {
            for (int j = 0; j < columns; j++)
            {
                    if (map[i][j] == 1)
                    {
                            map[i][j] = 0;
                            count++;
                            checkNeighbors(map, i + 1, j, rows, columns);
//RIGHT
                            checkNeighbors(map, i - 1, j, rows, columns);
//LEFT
                            checkNeighbors(map, i, j + 1, rows, columns);
//UP
```

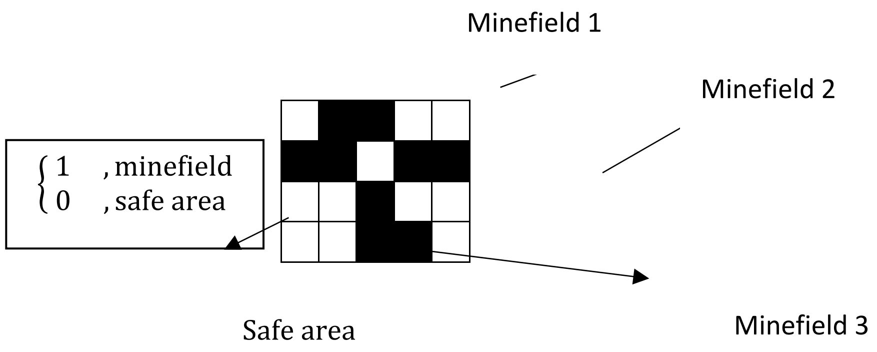
```
                                checkNeighbors(map, i, j - 1, rows, columns);
//DOWN
                      }
             }
   }
   return count;
}

int main()
{
   vector<vector<int>> map = {{0, 1, 1, 0, 0},
                                                              {1, 1, 0, 1, 1},
                                                              {0, 0, 1, 0, 0},
                                                              {0, 0, 1, 1, 0}};

   cout << "Number of minefields: " << countMinefields(map);
   return 0;
}
```



```
"D:\Users\Inspiron\Desktop\College\Semester 4\Algorithms\Final\Problem4.exe"
Number of minefields: 3
Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

- Analyze the complexity of your algorithms. **(5%)**
  Space complexity O(1)
  Time complexity O(n*m) //n being rows and m columns

```
Input:  map = [
                 [0,1,1,0,0]
            [1,1,0,1,1]
     [0,0,1,0,0]
          [0,0,1,1,0]
          ]
```



Output: 3

Explanation: There are three minefields on the map. Their areas are 4, 2, and 3.