Glouton

```
#include "allAlgo.h"

list<int> glouton(vector<XY> points)
{
    list<int> ordre = {0};
    vector<int> num;
    for (int i = 1; i < points.size(); i++)        n
    {
        num.push_back(i);                                n
    }
    int numberOfPoints = points.size();

    XY last = points[0];
    points.erase(points.begin());

    for(int z = 1; z < numberOfPoints; z++)        n
    {
        double smallestDistance = numeric_limits<double>::max();
        int indexSmallestDistancePoint = 0;
        for(int i = 0; i < points.size(); i++)          n(n-z)
        {
            double presentDistance = sqrt(pow(last.x-points[i].x,2) +
pow(last.y-points[i].y,2));                                           n(n-z)
            if(presentDistance < smallestDistance)
            {
                smallestDistance = presentDistance;
                indexSmallestDistancePoint = i;
            }
        }

        last = points[indexSmallestDistancePoint];
        ordre.push_back(num[indexSmallestDistancePoint]);
        points.erase(points.begin() + indexSmallestDistancePoint);
        num.erase(num.begin() + indexSmallestDistancePoint);
    }

    ordre.push_back(0);

    return ordre;
}
```

$$\sum_{z=1}^{n} \sum_{i=0}^{n-z} 1$$

$$= \sum_{z=1}^{n} (n-z)$$

$$= n^2 - \frac{n(n+1)}{2}$$

$$= n^2 - \frac{n^2}{2} - \frac{n}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2} \text{ donc } \in \Theta(n^2)$$

$\Theta(\max(n, n^2)) = \Theta(n^2)$

Approximatif

```
#include "allAlgo.h"
#include <bits/stdc++.h>
list<int> approximatif(vector<XY> & points)
{
    list<int> results;
    int numberOfPoints = points.size();

    int **graph = new int*[numberOfPoints];
       for(int i = 0; i < numberOfPoints; i++){  n
              graph[i] = new int[numberOfPoints];
              for(int j = 0; j < numberOfPoints; j++){  n^2
                     graph[i][j] = 0;                          n^2
              }
       }

    for(int i = 0; i < numberOfPoints; i++){      n
              for(int j = 0; j < numberOfPoints; j++){  n^2
                     graph[i][j] = graph[j][i] = sqrt(pow(points[j].x-points[i].x,2) +
pow(points[j].y-points[i].y,2));                               n^2
              }
       }

    int parent[numberOfPoints];

    int key[numberOfPoints];

    bool mstSet[numberOfPoints];

    for (int i = 0; i < numberOfPoints; i++){     n
        key[i] = INT_MAX;                              n
        mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < numberOfPoints - 1; count++)
    {                                              n
        int min = INT_MAX;
        int min_index;

        for (int v = 0; v < numberOfPoints; v++){      n^2
            if (mstSet[v] == false && key[v] < min){
                min = key[v];                                  < 4n^2
                min_index = v;
            }
        }
```

```
        mstSet[min_index] = true;

        for (int v = 0; v < numberOfPoints; v++){  n

            if (graph[min_index][v] && mstSet[v] == false && graph[min_index][v] <
key[v]) { ???????
                parent[v] = min_index;
                key[v] = graph[min_index][v];
            }

        }

    }

    cout<<"Edge \tWeight\n";
    for (int i = 1; i < numberOfPoints; i++){ n
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
    }

    int **graphEulerian = new int*[(numberOfPoints-1)*2];

    for (int i = 0; i < (numberOfPoints-1)*2; i++){ 2n
        graphEulerian[i]=new int[2];
    }

    for (int i = 1; i < numberOfPoints; i++){ n
        graphEulerian[i-1][0] = parent[i];
        graphEulerian[i-1][1] = i;
        graphEulerian[i+numberOfPoints-2][0] = i;
        graphEulerian[i+numberOfPoints-2][1] = parent[i];
    }

    bool used[(numberOfPoints-1)*2];
    for (int i = 0; i < (numberOfPoints-1)*2; i++){ 2n
        used[0] = false;
    }

    vector<int> eulerianCycle;
    eulerianCycle.push_back(0);

    for (int i = 0; i < (numberOfPoints-1)*2 -1; i++){ 2n
        for (int j= 0; j < (numberOfPoints-1)*2; j++){ 4n^2
            if(!used[j] && graphEulerian[j][0] == eulerianCycle.back()){
                used[j] = true;                              < 4n^2
                eulerianCycle.push_back(graphEulerian[j][1]);
                break;
            }
        }
    }

    bool added[numberOfPoints];
    for (int i = 0; i < numberOfPoints; i++){   n
```

```
        used[0] = false;
    }

    for (int i = 0; i < eulerianCycle.size(); i++){  n
        if(!added[eulerianCycle[i]]){

            results.push_back(eulerianCycle[i]);
            added[eulerianCycle[i]] = true;
        }
    }
    results.push_back(results.front());

    return results;
}
```

Note : les complexités notées < $4n^2$ veulent dire que le if aurait probablement enlevé un certain nombre d’exécution mais puisque nous avions déjà des boucles imbriqués qui avait une complexité de n^2 il n’était pas nécessaire de les calculer.

$\Theta(\max(n, n^2, < n^2)) = \Theta(n^2)$

Dynamique

```
#include "allAlgo.h"
#include <algorithm>
#include <bitset>

vector<vector<double>> getSimpleDistanceVector(vector<XY> & points);
void getDistanceVector(vector<vector<double>> & D, vector<vector<list<int>>> &
ordre, vector<vector<double>> & dist, vector<int> & S);
list<int> getFinalDistance(vector<XY> & points, vector<vector<double>> & D,
vector<vector<list<int>>> & ordre, vector<int> & S);

void setS(vector<int> & tmp, vector<int> & S, vector<int> & num, int k, int
offset = 0)
{
   if (k == 0) {
       S.push_back(1 << tmp[0]);
       for (size_t i = 1; i < tmp.size(); i++)                n
       {
               S.back() |= 1 << tmp[i];
       }
       return;
   }
   for (int i = offset; i <= num.size() - k; i++) {                  n
       tmp.push_back(num[i]);
       setS(tmp, S, num, k-1, i+1);                           n^2
       tmp.pop_back();
   }
}
```

```
list<int> dynamique(vector<XY> & points)
{
   vector<int> num;
   for (int i = 0; i < points.size() - 1; i++)                   n
   {
       num.push_back(i);                                          n
   }

   vector<int> S;
   S.push_back(0);
   for (size_t i = 1; i < num.size(); i++)                        n
   {
       vector<int> tmp;
       setS(tmp, S, num, i);                                          n^3
   }

   vector<vector<double>> dist = getSimpleDistanceVector(points);   n^2

   vector<vector<double>> D;
   vector<vector<list<int>>> ordre;
   for (int i = 1; i < points.size(); i++)                            n
   {
       vector<double> di(S.size());
       di[0] = sqrt(pow(points[0].x - points[i].x, 2) + pow(points[0].y -
points[i].y, 2));
       D.push_back(di);

       vector<list<int>> oi(S.size());
       list<int> f = {i, 0};
       oi[0] = f;
       ordre.push_back(oi);
   }

   getDistanceVector(D, ordre, dist, S);                          n^2 2^2n

   return getFinalDistance(points, D, ordre, S);                  n^2
}

vector<vector<double>> getSimpleDistanceVector(vector<XY> & points)
{
   vector<vector<double>> D;

   int size = points.size();

   for(int i = 1; i < size; i++)                                          n
   {
       vector<double> di;

       for (size_t j = 1; j < size; j++)                              n^2
       {
           double toAdd = -1;
           if(i != j)
           {
               toAdd = sqrt(pow(points[i].x - points[j].x, 2) + pow(points[i].y
- points[j].y, 2));
           }
           di.push_back(toAdd);
       }
```

```
        D.push_back(di);
    }

    return D;
}

vector<int> findBitPosition(int n)
{
    vector<int> posVec;
    int pos = 0;
     while (n) {                                                          n
        if(n & 1)
        {
            posVec.push_back(pos);
        }

        n = n >> 1;
        pos++;
    }
     return posVec;
}

void getDistanceVector(vector<vector<double>> & D, vector<vector<list<int>>> &
ordre, vector<vector<double>> & dist, vector<int> & S)
{
    int rowSize = D.size();
    int colSize = S.size();

    for(int i = 1; i < colSize; i++)                                     2^n
    {
        vector<int> pos = findBitPosition(S[i]);

        for (size_t j = 0; j < rowSize; j++)                          n 2^n
        {
            double toAdd = -1;
            list<int> ordreToAdd = {-1};
            if(!(S[i] & (1 << j)))
            {
                vector<double> results;
                vector<list<int>> resultsOrdre;
                int top = pos.size() - 1;
                for (int k = 0; k < pos.size(); k++)                 n^2 2^n
                {
                    uint shift = ~(1 << pos[k]);
                    int si = S[i] & shift;
                    int col = 0;
                    while(S[col] != si)                              n^2 2^2n
                    {
                        col++;
                    }
                    if(D[pos[k]][col] != -1)
                    {
                        results.push_back(D[pos[k]][col] + dist[j][pos[k]]);
                        resultsOrdre.push_back(ordre[pos[k]][col]);
                        resultsOrdre.back().push_front(j + 1);
                    }
                }
                double smallest = -1;
                list<int> ordreMin = {-1};
                for (size_t d = 0; d < results.size(); d++)    n^3 2^n
```

```
                {
                    if(smallest == -1 || smallest > results[d])
                    {
                        smallest = results[d];
                        ordreMin = resultsOrdre[d];
                    }
                }

                toAdd = smallest;
                ordreToAdd = ordreMin;
            }

            D[j][i] = toAdd;
            ordre[j][i] = ordreToAdd;
        }
    }
}

list<int> getFinalDistance(vector<XY> & points, vector<vector<double>> & D,
vector<vector<list<int>>> & ordre, vector<int> & S)
{
   double smallest = -1;
   int idxISmallest = 0;
   int idxJSmallest = 0;
   for (size_t i = 0; i < D.size(); i++)                             n
   {
       for (size_t j = D[i].size() - D.size() + 1; j < D[i].size(); j++)
                                                                  n^2
       {
           if(D[i][j] != -1)
           {
               double val = D[i][j] + D[i][0];
               if(smallest == -1 || smallest > val)
               {
                   smallest = val;
                   idxISmallest = i;
                   idxJSmallest = j;
               }
           }
       }
   }
   ordre[idxISmallest][idxJSmallest].push_front(0);

   return ordre[idxISmallest][idxJSmallest];
}
```

$O(n^2\ 2^n)$