Billy Saysavath
Duy Do
Phung Tran

CPSC 335 – Section 3
Fall 2015
Assignment #3

I. Exhaustive Optimization Algorithm:

a. Pseudocode:

```
#Clock starts at this point
  Dist = farthest(n,P)
  bestDist [:n]= Dist
  A [n]

  for i in range(0, n-1):
    A[i] = i;

def print_perm( n, A[], sizeA, P, bestSet, bestDist ):
     i, dist
     if (n == 1):
          dist = sqrt( pow( P[A[i]].x – P[A[sizeA-1]].x, 2 ) + pow( P[A[i]].y – P[A[sizeA-1]].y, 2 ))
          if( disk < bestDist ):
                    bestDist = dist
     else
          for i in range (0, sizeA):
                    print_perm(n – 1, A, sizeA, P, bestSet, bestDist)
                    if ( n % 2 == 0 ):
                              A[i], A[n-1] = A[n-1], A[i]
                    else:
                              A[0], A[n-1] = A[n-1], A[0]
          print_perm(n – 1, A, sizeA, P, bestSet, bestDist)

def farthest(n, P):
    max_dist=0;
     i, j;
     dist;

    for i in range(0, n):
       for j in range(0, n):
          dist = (P[i].x - P[j].x)*(P[i].x - P[j].x) + (P[i].y - P[j].y)*(P[i].y - P[j].y)
          if (max_dist < dist):
             max_dist = dist
```

```
34
35    return sqrt(max_dist)
```

b. Analyze:
   - Line 2 calls farthest function, and it takes O(n^2) steps.
   - Line 3 takes n steps. Line 4 takes a constant step, so we say 1.
   - Line 6 takes n steps to fill the values in a permutation array A.
   - Then, the program calls print_perm. In print_perm, line 10 takes 2 steps.
   - In an if-else statement, line 12 takes 1 step. Line 13 takes 1 + max(1, 0) step.
   - Line 16, the for-loop will take sizeA steps, which equals to number of points m.
   - Multiply the recursive call to print_perm at line 17 which, as we learned in a class, takes O(n!), and 1 + max(1, 1) for if-else statement from line 18 to 21.
   - Lastly, we add another O(n!) for a recursive call to print_perm. We have the following:

$$T(n) = n^2 + n + 1 + n + 2 + 1 + \max\left(1 + 1 + \max(1,0)\,, m\left(n! + 1 + \max\,(1,1)\right) + n!\right)$$

$$T(n) = n^2 + 2n + 4 + \max\,(1 + 1 + 1, m(n! + 1 + 1) + n!)$$

$$T(n) = n^2 + 2n + 4 + \max\,(3, m(n! + 2) + n!)$$

$$T(n) = n^2 + 2n + 4 + \max\,(3, m \cdot n! + 2m + n!)$$

$$T(n) = m \cdot n! + n! + 2m + n^2 + 2n + 4$$

We have, $O(n) = m \cdot n!$ where m = n; therefore,

$$T(n) \in O(n) = n \cdot n!$$

II. Approximation algorithms

a. Pseudocode:

```
// allocate space for the INNA set of indices of the points
M = new int[n];
// set the best set to be the list of indices, starting at 0
for( i=0 ; i<n ; i++)
   M[i] = i;

// Start the chronograph to time the execution of the algorithm at this point

// allocate space for the Visited array of Boolean values
Visited = new bool[n];
// set it all to False
for( i = 0;  i< n; i++)
   Visited[i] = false;
```

```
// calculate the starting vertex A
A = farthest_point(n,P);
// add it to the path
I = 0;
M[i] = A;

// set it as visited
Visited[A] = true;

for(i=1; i<n; i++) {
   // calculate the nearest unvisited neighbor from node A
   B = nearest(n, P, A, Visited);

   // node B becomes the new node A
   A = B;
   // add it to the path
   M[i] = A;
   Visited[A] = true;
}

// calculate the length of the Hamiltonian cycle
dist = 0;
for (i=0; i < n-1; i++)
   dist += sqrt((P[M[i]].x - P[M[i+1]].x)*(P[M[i]].x - P[M[i+1]].x) +
         (P[M[i]].y - P[M[i+1]].y)*(P[M[i]].y - P[M[i+1]].y));

dist += sqrt((P[M[0]].x - P[M[n-1]].x)*(P[M[0]].x - P[M[n-1]].x) +
        (P[M[0]].y - P[M[n-1]].y)*(P[M[0]].y - P[M[n-1]].y));

// End the chronograph to time the loop at this point

  def farthest_point(n, P):
        farthest_point = 0
        max_dist, dist
        i, j
        max_dist = sqrt( (P[0].x - P[n-1].x)*(P[0].x - P[n-1].x)  + (P[0].y - P[n-1].y)*(P[0].y - P[n-1].y) )

        for i in range[0, n-1]:
               for j in range[0, n-1]:
                   dist = sqrt(  (P[i].x - P[j].x)*(P[i].x - P[j].x) +  (P[i].y - P[j].y)*(P[i].y - P[j].y) )
                   if (max_dist < dist) :
                        max_dist = dist;
                        farthest_point = i
```

```
57          return farthest_point
58
59    def nearest(n, P, A, Visited):
60            min_dist, dist
61            nearest, i
62            for i in range[0, n]:
63                  if ( !Visited[i] ):
64                        min_dist = sqrt(  (P[A].x - P[i].x)*(P[A].x - P[i].x) +  (P[A].y - P[i].y)*(P[A].y - P[i].y) )
65                        nearest = i
66            for i in range[0, n]:
67                  if ( !Visited[i] ):
68                        dist = sqrt(  (P[A].x - P[i].x)*(P[A].x - P[i].x) +  (P[A].y - P[i].y)*(P[A].y - P[i].y) );
69                        if (min_dist > dist):
70                              min_dist = dist
71                              nearest = i
72
73             return nearest;
```

b. Analyze:

- Line 2 takes 1 step
- Line 4-5, the for-loop takes n steps
- Line 10 takes 1 step
- Line 12-13 take n steps
- Line 16 calls farthest_point() which takes n^2
    - farthest_point() begins at line 45
        - Lines 46-48, and line 57 each takes 5 steps
        - Line 51, outer loop, iterates i from 0 to n-1 which takes n steps
            - Line 52, inner loop, iterates i from 0 to n-1 which steps n steps
                - Line 53 takes 1 step
                - Line 54-56 takes 1 + max(2, 0) which equals 3 steps
- Line 23 iterates i from 1 to n-1 steps which takes n-2 steps. Multiply to a time calling nearest() we have n^2-n
    - Line 10 calls nearest which takes n time
    - nearest() begins at line 33
        - Lines 60, 61, and 73 each takes 1 step = 3 steps
        - Line 62 iterates from 0 to n-1 takes n steps
            - Lines 63-65 take 1 + max(2,0) which is 3 steps
        - Line 66 iterates from 0 to n-1 takes n steps
            - Lines 67-71 takes 1 + max(1+1+max(2,0),0) which takes 5 steps
    - Lines 27-31 each take 3 steps,
- Line 35 takes 1 step.
- Line 36-38 iterates from 0 to n – 2, so it takes n-1 steps
- Line 40 takes 1 step

Altogether, we have:

$$T(n) = 1 + n + 1 + n + n^2 + n^2 - n + 1 + n - 1 + 1$$

$$T(n) = 2n^2 + 4n + 2$$

Therefore, $T(n) \in O(n^2)$