MARMARA UNIVERSITY
FACULTY OF ENGINEERING

CSE-2246 Analysis Of Algorithms

Homework - 2

Due Date: 06.06.2023

	Department	Student Id	Name Surname
1	CSE	150120027	Mert Muslu
2	CSE	150120051	Erkut Dönmez
3	CSE	150121539	Gülsüm Ece Günay

1. INTRODUCTION

The Half-Travelling Salesman Problem, known as H-TSP, is an NP-Hard optimization problem. TSP assumes that the salesman tries to find the shortest possible routes with n cities. The important point is that the salesman can not visit the same vertex again, and has to return to the same starting vertex. However, for the H-TSP, the salesman should visit the ⌈n/2⌉ vertex from the n vertex. The difference between TSP and H-TSP is that TSP focuses on finding the shortest route. However, besides this point, H-TSP also focuses on hierarchical constraints due to several proposed routes that it can go through.

For the given problem, there are numerous feasible solutions. However, all possible solutions do not meet the optimization in terms of time and space, which are reasonably crucial points for large inputs.

2. ANALYZE/HEURISTIC

2.1 ALGORITHMS

Our algorithm begins by computing the distances between all cities and storing them in a two-dimensional array. Subsequently, it identifies the two cities that generate the shortest distance and initiates the exploration from both cities towards the next closest city. This iterative process continues until half of the cities have been visited. Once the shortest path has been determined for each starting city, we ascertain which starting city corresponds to the overall shortest path. Finally, we compile the shortest path and the visited cities in the designated order, and write them to our output file.

What did we do to make our algorithms more efficient?

a. Merge Sort:

Our algorithm calculates the distance between cities by looping $O(n^2/2)$ times for a given input n. After that, we apply the merge sort for the array in that we store the given input values, however, instead of changing any value -id, x-axis, y-axis- we created a new array and transferred the projective order of id's according to x and y coordinates into the array with respect to the id's. Then, when calculating the distance between cities, we avoided calculating the distance between cities with significantly different x and y coordinates based on this order. This was because the basic operation in calculating the distance was the most time-consuming factor affecting the time complexity. By incorporating mergeSort as an eliminator factor in the distance calculation, it allowed us to reduce the complexity from n^2 to $n^2 * \log_2 n$ n / n, while the mergeSort algorithm itself cost us $\Theta(n * \log n)$.

b. Hashing:

While calculating the shortest distances between two cities, we created the visited array in order not to go back to cities that have already been visited (in order not to go to the same city twice). When we stored the ids of the visited cities in this array, we were checking this by opening another for loop inside the 2 nested for loops. This increased the time complexity to O(n^3) and caused it to run in 424 seconds for input with n = 15112. But using the hashing method we learned in the data structures course, we controlled this by using the hashing algorithm F(x) = x instead of the for loop. In the defining part of our array, we gave -1 for all values, and by applying the hashing algorithm array[id] = id to the cities visited, we equated the values of the indexes of the cities visited in the array to their ids. In this way, instead of looping through the loop, if the value of the city we checked in our array was not -1, we checked that city as visited. In fact, we could have gained a lot in terms of space efficiency by using linear probing or quadratic probing, but since the given inputs are not very large, we set our goal in favor of time efficiency. Therefore, our hashing algorithm is F(x) = x.

c. Creation of distance array:

While creating the distance array we've used the code that given below:

distance[i][j] = (abs(pow((cities[i].x - cities[j].x),2) + pow((cities[i].y - cities[j].y),2)));

However, our observations revealed that utilizing that formula to create the distance array resulted in significant time costs. As an alternative approach, we realized that when calculating the array, such as distance[3][5] between cities 3 and 5, there was no need to calculate the value for distance[5][3] since it held the same value as distance[3][5]. To avoid redundant calculations, we assigned a large value to distance[5][3] that would never be used. Essentially, our two-dimensional array represented a symmetric matrix, and we eliminated the need to compute unnecessary values. Additionally, we set a large value for the distance between a city and itself as it would always be a zero and never used in our calculations.For instance, in the case of distance[0][0], which represents the distance between the 0th city and itself, we assigned a very large value instead of zero. This was done to avoid potential issues that could arise from using zero in our program.

d. Buffer

We create a struct that defines x and y coordinates along with city IDs.We stored the city data(coordinates and city number) we retrieved from our input file in an array. We dynamically allocated the array using malloc function. Initially, the allocated space was only 2 * 12 bytes (in any case we wanted to initialize it with a higher size and multiplied the size of our struct with two) because the struct contained three variables which are id,x and y and those variables created as an integer, which totalled 12 bytes. Since the size of our input file can vary, we used the "realloc" function to double the size of our array whenever the number of cities exceeded its current capacity.This way, we used buffer logic. By doing so, we avoided the need to initialize the array with a very high value and increased space efficiency.

2.2. TIME COMPLEXITY OF OUR PROGRAM

The algorithms for which we need to calculate the actual time complexity of the program are as follows.

MergeSort : (Between Lines 16-104)

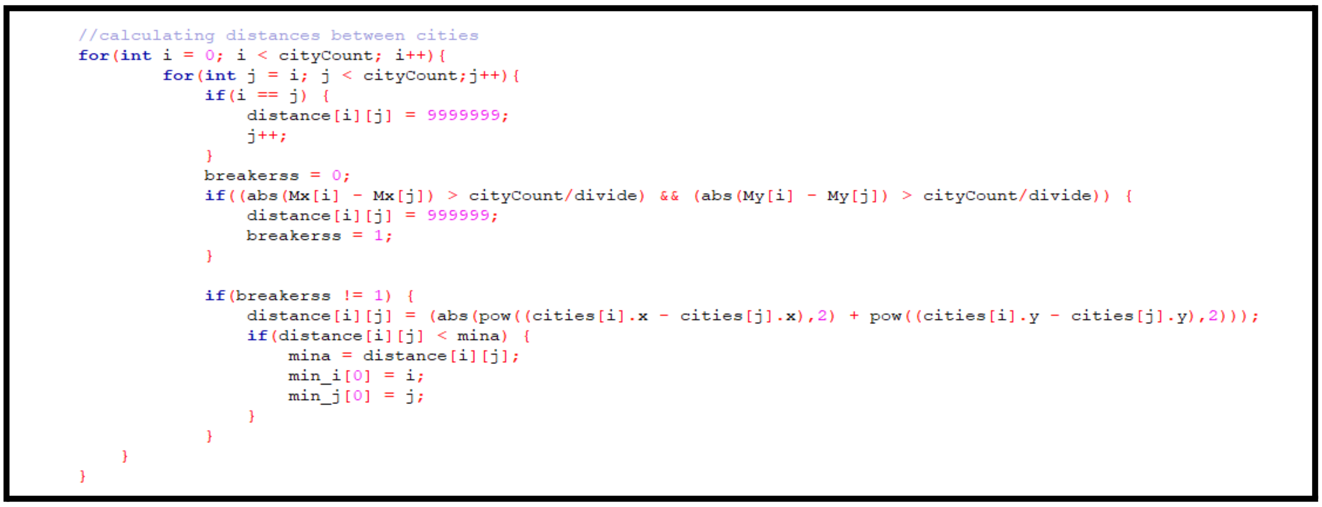
Basic Operation: Merging of two sorted subarrays into a single sorted array.

Space Efficiency: in-place

Stability: Yes

Time Efficiency: O(n * logn)

Calculation of Distance: <u>(Between Lines 203-224)</u>



```
//calculating distances between cities
for(int i = 0; i < cityCount; i++){
        for(int j = i; j < cityCount;j++){
            if(i == j) {
                distance[i][j] = 9999999;
                j++;
            }
            breakerss = 0;
            if((abs(Mx[i] - Mx[j]) > cityCount/divide) && (abs(My[i] - My[j]) > cityCount/divide)) {
                distance[i][j] = 999999;
                breakerss = 1;
            }

            if(breakerss != 1) {
                distance[i][j] = (abs(pow((cities[i].x - cities[j].x),2) + pow((cities[i].y - cities[j].y),2)));
                if(distance[i][j] < mina) {
                    mina = distance[i][j];
                    min_i[0] = i;
                    min_j[0] = j;
                }
            }
    }
}
```

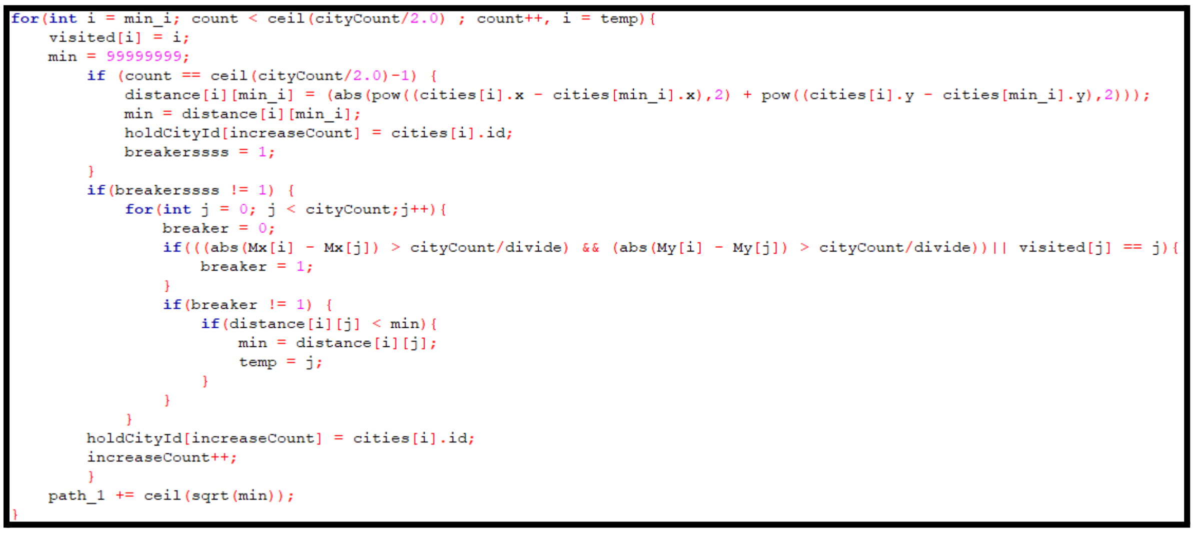
Basic Operation : Calculation of distance (Power)

Space Efficiency: in-place

Stability: No (To aim for gaining time efficiency, stability of the algorithm is ignored.)

Time Efficiency : C_{Worst} : $O(n^2)$ C_{Best} : $\Omega(n * \log_2 n)$

Calculation of Path: <u>(Between Lines 245-271)</u>



```
for(int i = min_i; count < ceil(cityCount/2.0) ; count++, i = temp){
    visited[i] = i;
    min = 99999999;
        if (count == ceil(cityCount/2.0)-1) {
            distance[i][min_i] = (abs(pow((cities[i].x - cities[min_i].x),2) + pow((cities[i].y - cities[min_i].y),2)));
            min = distance[i][min_i];
            holdCityId[increaseCount] = cities[i].id;
            breakerssss = 1;
        }
        if(breakerssss != 1) {
            for(int j = 0; j < cityCount;j++){
                breaker = 0;
                if(((abs(Mx[i] - Mx[j]) > cityCount/divide) && (abs(My[i] - My[j]) > cityCount/divide))|| visited[j] == j){
                    breaker = 1;
                }
                if(breaker != 1) {
                    if(distance[i][j] < min){
                        min = distance[i][j];
                        temp = j;
                    }
                }
            }
        holdCityId[increaseCount] = cities[i].id;
        increaseCount++;
        }
    path_1 += ceil(sqrt(min));
}
```

Basic Operation : Assignment(Equalization) or Comparison

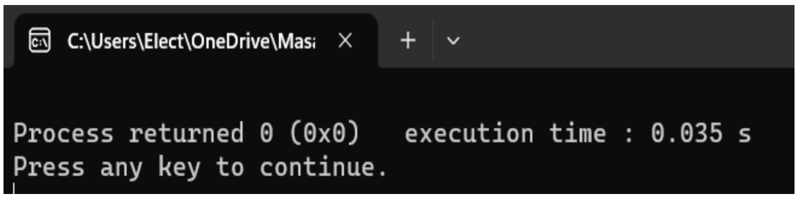
Space Efficiency : not in-place (because of F(x) = x hashing algorithm)

Stability : No, When minimum distance occurs more than once, “min” identifier holds the lowest “i” value’s distance.

Time Efficiency: **C_{Worst} : $O(n^2)$** **C_{Best} : $\Omega(n * \log_2 n)$**

2.3 TIME SPAN FOR INPUT FILES

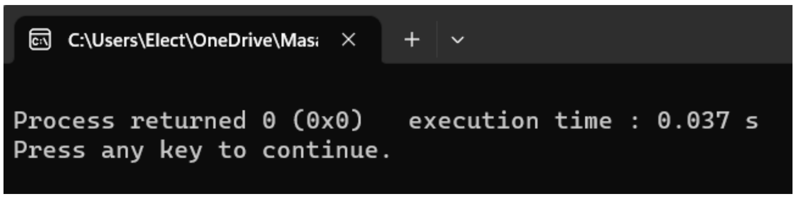
1. First txt (76 inputs): 0.035 s



```
C:\Users\Elect\OneDrive\Mas

Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

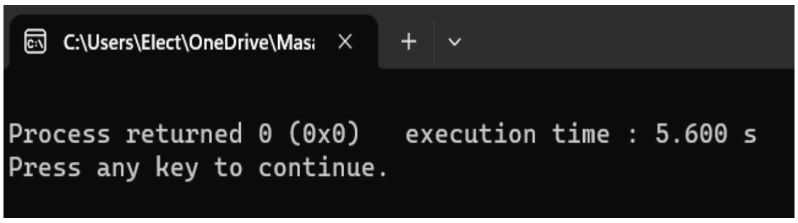
2. Second txt(280 inputs): 0.037 s



```
C:\Users\Elect\OneDrive\Mas

Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

3. Third txt(15112 inputs): 5.60 s



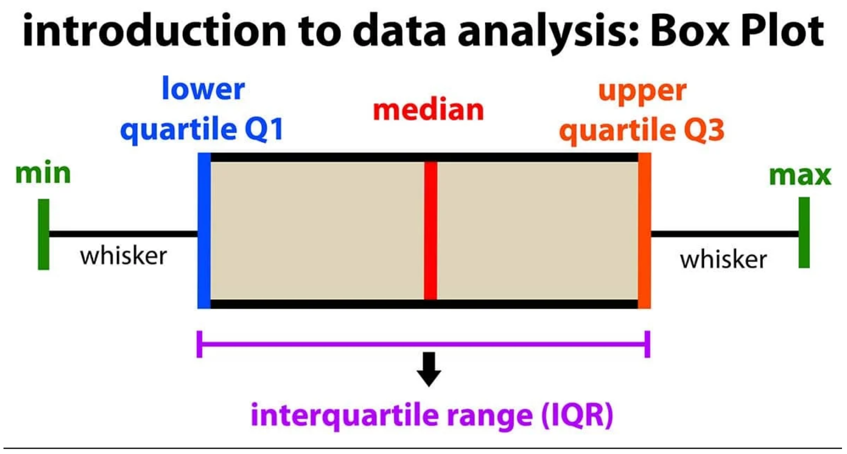
```
C:\Users\Elect\OneDrive\Mas

Process returned 0 (0x0)   execution time : 5.600 s
Press any key to continue.
```

3. WHAT DID COME TO OUR MINDS? (ADDITIONALLY)

While working on the project, several ideas came to our minds. One of the ideas involved eliminating cities through statistical analysis. The underlying concept was to identify outlying cities by utilizing statistics. We planned to employ the interquartile range, a statistical measure, to handle the large input files efficiently. However, upon examining our input files, we realized that there might not be as many outlying values as initially anticipated. Consequently, we explored alternative approaches to eliminate unnecessary values that we would never visit. Since the problem at hand is H-TSP rather than TSP, it was crucial for us to be highly selective in determining our path, as we do not need to travel through all the cities. This led us to the algorithm mentioned earlier.

IQR (Interquartile Range) is explained furthermore:


introduction to data analysis: Box Plot
lower quartile Q1
median
upper quartile Q3
min
max
whisker
whisker
interquartile range (IQR)

Upon analyzing the provided figure, our plan involved calculating the lower quartile and upper quartile ranges based on the given x and y coordinates. Subsequently, we would determine the lower and upper fences using the respective formulas. By applying these calculations, we aimed to identify cities that fall below the lower fence or exceed the upper fence, classifying them as outliers. These outliers would be excluded from any path calculations, as they are not deemed necessary to visit.

4. CONCLUSION

To sum up, Half-Travelling Salesman Problem is an NP-hard problem, which means there are not any known ways to optimize this problem in a reasonable amount of time and memory. As the input size enlarges, the number of possible routes and memory consumption also increases. Moreover, the growth occurs exponentially, which means that it makes searching impractical. For efficiency, we try to shrink observed data as much as possible with reasonable algorithms -merge sort, hashing, etc.- so that we deal with quite less amount of input. However, in order to enhance the program's efficiency in terms of time, we opted to store all the required data in arrays. This decision resulted in improved time efficiency, although at the expense of space efficiency.

5. DIVISION OF LABOR

Report: Mert Muslu, Erkut Dönmez, Gülsüm Ece Günay

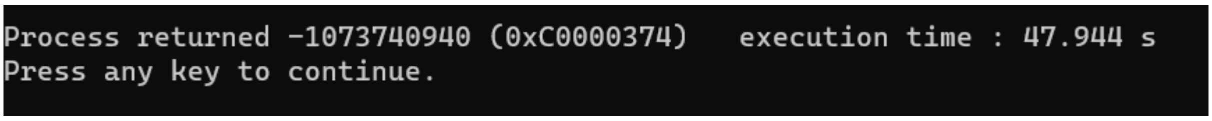
Algorithms: Hashing -Mert, Erkut; Buffer -Mert, Ece; MergeSort -Mert.

Input and Memory issues: Mert, Ece, Erkut

6. INCOMPLETED PARTS

When the input size exceeds 1GB, we encounter an error with the message "Process returned -1073740940 (0xC0000374)" and find that the output.txt file is empty. This issue arises due to our usage of a 2D integer array called "distance". Specifically, when the input value reaches 33810, the distance array requires more than 4GB of memory allocation, leading to the error.

For the test-input-3.txt, our terminal is like the one given below, and our test-output-3.txt is empty. When we change the data type of the distance array with "unsigned long long int" instead of "int", we get an output file that includes city ID's but randomly ordered and the path is greater than sexillion.



```
Process returned -1073740940 (0xC0000374)   execution time : 47.944 s
Press any key to continue.
```

7. REFERENCES

1. **GeeksforGeeks. (2023b, May 31).** ***Merge sort - data structure and algorithms tutorials*****. GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/**

2. **Tümer, B. (2023)** ***Hashing, week7.pptx*** **[CSE 2025 Data Structures].**