Bayram Barış Sarı

In this project, I tried to solve a closest pair problem in 3D space by the help of divide-and-conquer algorithm. This algorithm divides problem to subproblems, and solve them(conquer) after it becomes easy, then it combines solutions. Therefore, it uses recursions and its complexity is calculated by **master method**. The procedure of master method is the same with divide-and-conquer algorithm, it creates subproblems recursively until these subproblems becomes an easy problem, then solves and combines these subproblems. Its formula as follows:

$$T(n) = a\,T(n/b) + f(n)$$

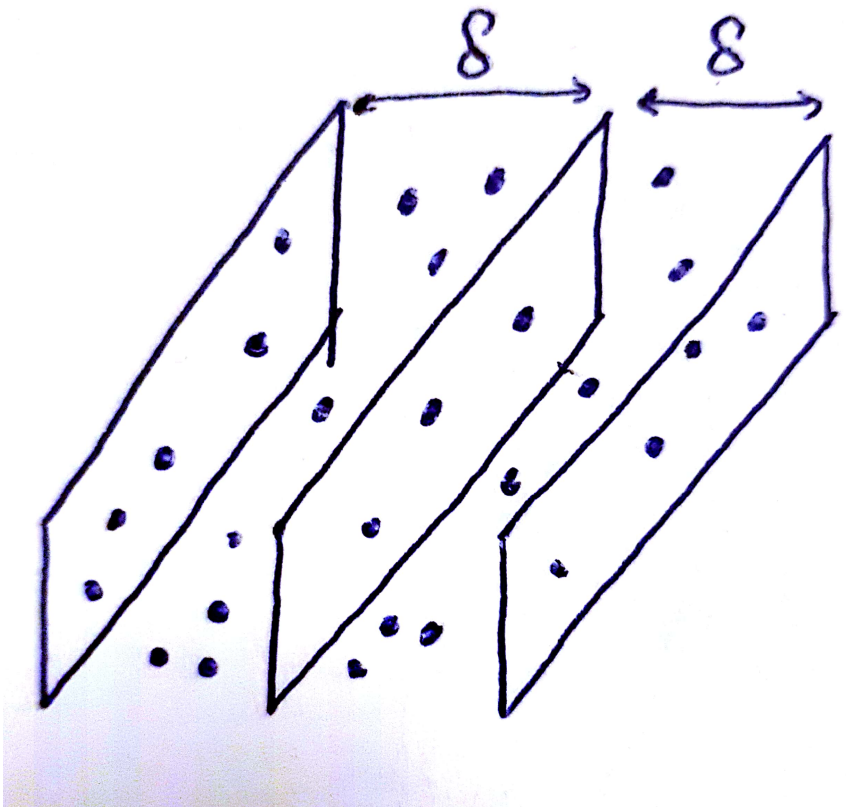
In this formula, 'a' means subproblems and 'n/b' means the size of subproblems-'a'.

In our problem, if we compare euclidean distances between all points, it takes $\boldsymbol{O(n^2)}$ time. This method is called **Brute Force** and it's **not the solution** of this project.

At the beginning, I have used quick sort algorithm that I implemented for AoA-I lecture to sort the data according to its x-coordinate. The time complexity for quick sort is $\boldsymbol{O(nlogn)}$. Because I thought this space as a rectangular prism, with edges are x,y and z, and I wanted to divide this rectangular prism from the middle until it becomes small enough to solve the problem. Let's call this space S. When it's divided into two parts, S1 and S2, it means that two sub-rectangular prisms of S are created by hyperplane at the x median of S. Now, there are **three situations for closest pair of points**: ***two points are in S1, two points are in S2 or one point is in S1, and the other point is in S2***.

Firstly, we need to find **min** = **(smallest distance in S1, smallest distance in S2)**. Then, we need to create a new set of points that are at most "**min"** far away from hyperplane that divides. The picture below explains this new set better:

Note : min is shown as δ in this figure. There is a hyperplane at the middle, and the points between δ and hyperplane are added to the set for third possibility(one point is in S1, the other one is in S2).



When this steps are applied recursively, the smallest distance between two points in the data can be found in ***O(nlogn)*** time.

Let's look at the functions more closely:

```
double Coordinate::find_min(std::vector<Coordinate> &c) {
    std::vector<Coordinate> eliminated;
    double min, min_left=0, min_right=0, min_lr=0;
    if(c.size() <= 4)
        return helper_min(c);
    std::vector<Coordinate> temp1;
    std::vector<Coordinate> temp2;
    divide(c, temp1, temp2);
    min_left = find_min(temp1);
    min_right = find_min(temp2);
    min = min_right;
    if(min_left<min_right)
        min = min_left;
    /** New set of points are created*/
    eliminated = eliminate_points(c, min);
    min_lr = helper_min(eliminated);
    if (min_lr<min)
        min = min_lr;
    return min;
}
```

This function recursively divides the data into two parts until size of these parts are equal or smaller than 4, and then it returns the result of helper_min function for these parts. When the smallest distance of 2-sub spaces are found, min is set to the smallest distance and new set of points are created for third possibility. Then, it's also sent to helper_min function and the minimum result is returned.

```
}
void Coordinate::divide(std::vector<Coordinate> orig,
std::vector<Coordinate> &sub1, std::vector<Coordinate> &sub2) {
    int index = orig.size()/2;
    int i=0;
    std::vector<Coordinate>::iterator it;
    for(it = orig.begin(); it != orig.end(); ++it){
        if(i < index){
            sub1.push_back(*it);
        }else{
            sub2.push_back(*it);
        }
        i++;
    }
}
```

That's the divide function. It takes two empy vectors and the original vector, and then divides the original vector into two parts. It takes ***O(n)*** time to divide because of for loop.

```
double Coordinate::helper_min(std::vector<Coordinate> c) {
    double dist=0, min= 1000000;
    std::vector<Coordinate>::iterator i;
    std::vector<Coordinate>::iterator j;
    for(i = c.begin(); i != c.end(); ++i) {
        for (j = std::next(i); j != c.end(); ++j) {
            dist = distance(*i, *j);
            if (dist <= min) {
                min = dist;
            }
        }
        total_calculation++;
    }
    return min;
}
```

This function checks the distances of all point pairs and returns the minimum distance. It takes at most ***O(6)*** time because size of c is at most 4.

```
std::vector<Coordinate>
Coordinate::eliminate_points(std::vector<Coordinate> c, double min) {
    std::vector<Coordinate> temp;
    std::vector<Coordinate>::iterator it;
    int median = c[c.size()/2].getx();
    for(it = c.begin(); it != c.end(); ++it){
        if(abs((*it).getx()-median) < min){
            temp.push_back(*it);
        }
    }
    return temp;
}
```

This function creates a new set of points in the space that shown in the picture above and returns the vector of these points. It takes ***O(n)*** time. N = size of c, number of point pairs.

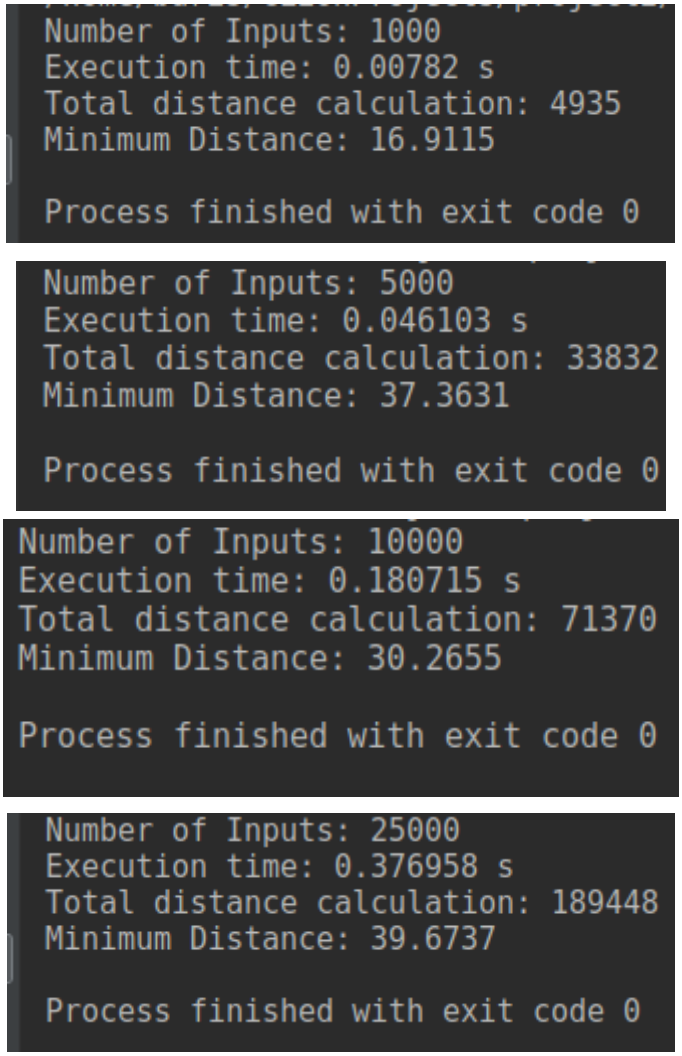
```
double Coordinate::distance(Coordinate c1, Coordinate c2) {
    // Distance = sqrt( (x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2)
    return sqrt(pow(c1.getx()-c2.getx(),2)+(pow(c1.gety()-c2.gety(),2))
+(pow(c1.getz()-c2.getz(),2)));
}
```

This function calculates the euclidean distance between two points and returns it. It takes ***O(1)*** time.

So, $T(n) = 2\,T(n/2) + f(n) = 2\,T(n/2) + O(n) + O(n) + O(n\log n)$,
(first O(n) is for divide,second O(n) is for eliminate_points, and O(nlogn) for quicksort)

$T(n) = 2\,T(n/2) + O(n\log n)$. This is case-2 of the master theorem.
$a = 2$, $b = 2$, and $a/b = 1 = c$ (c comes from $n^c \log^k n$).
So, ***$T(n) = \Theta(n\log n)$***

When the algorithm is executed with 4 different data sets, the results are as follows:



As it has seen obviously, when the number of inputs is increased, total distance calculation and execution time also increase.