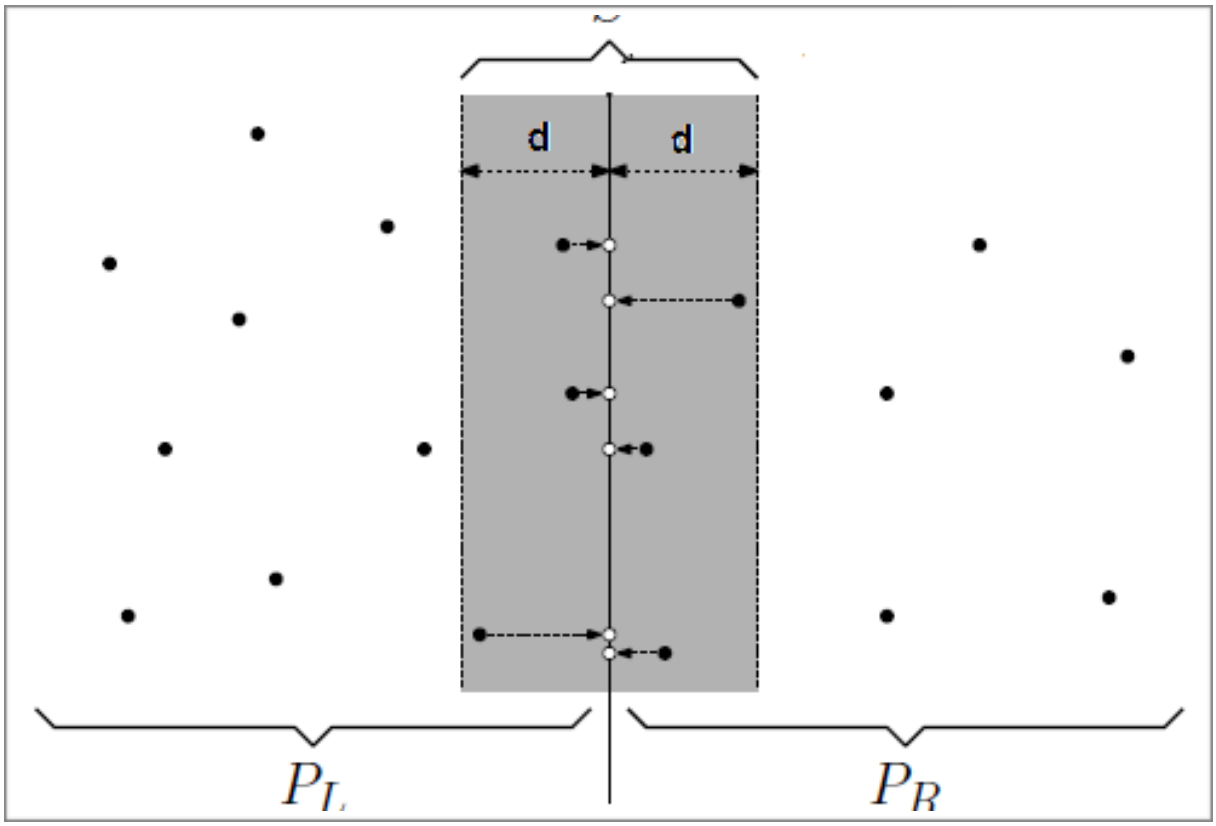
BLG 336E 2nd Project Report

Analysis of Algorithms II



Tuğba Özkal

150120053

9.04.2018

BLG 336E 2nd Project Report

1. What do a and b of Master Theorem mean in divide and conquer approach?

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

In divide and conquer approach, we use the method like merge sort. It means there is the recursion in the solution. "a" in master theorem refers to number of subproblems. "b" specifies the size of subproblems. Hereby, recursion will be solved.

2. Problem formulation in detail

Point class:

```
class Point{
private:
    int x;
    int y;
    int z;
public:

    Point(int = 0, int = 0, int = 0);       // default values of all x, y, and z are zero
    ~Point();   // deconstructor
    float getDistance(Point);   // distance between two points are calculated by using euclidean-distance
    void Print() const;

    int getX(){
        return x;
    }

    int getY(){
        return y;
    }

    int getZ(){
        return z;
    }

    bool operator<(const Point& other) const{   // < operator is used for comparison by x default
        return x < other.x;
    }

};
```

Point class includes x, y and z coordinates, getDistance() function and Print() function.

Point::getDistance():

```
float Point::getDistance(Point p){
    return sqrt(pow((this->x - p.x), 2) + pow((this->y - p.y), 2) + pow((this->z - p.z), 2));
}
```

Euclidean distance formula is used for this method.

$$distance = \sqrt{(x_{this} - x_p)^2 + (y_{this} - y_p)^2 + (z_{this} - z_p)^2}$$

Point::Print():

```
void Point::Print() const{
    cout << "x:" << this->x << "\ty:" << this->y << "\tz:" << this->z << endl;
}
```

Print() function prints the values of x, y and z coordinates.

Operator overload:

```
  bool operator<(const Point& other) const{   // < operator is used for comparison by x default
      return x < other.x;
  }
```

< operator is overloaded. Its reason is sorting algorithm needs the comparison function objects. **sort()** function takes x coordinate default by operator overloading.

There is another function for comparison by y coordinate.

```
bool compareY(Point p1, Point p2){
    return (p1.getY() < p2.getY());
}
```

When we sort the list by x coordinate, we call function by this command.

```
sort(px.begin(), px.end());
```

When we sort the list by y coordinate, we call function by this command.

```
sort(py.begin(), py.end(), compareY);
```

ClosestPair class:

```
class ClosestPair{
private:
    Point p1;
    Point p2;
    float minDistance;
public:
    ClosestPair();
    ~ClosestPair();
    ClosestPair * findClosestPair(vector<Point>, vector<Point>);
    ClosestPair * bruteForce(vector<Point>);
    vector<Point> readFromFile(char*);
    void Print();
};

bool compareY(Point p1, Point p2){
    return (p1.getY() < p2.getY());
}
```

ClosestPair class includes first point, second point, their distance and methods to find distance.

ClosestPair::readFromFile():

```
vector<Point> ClosestPair::readFromFile(char * filename){
    ifstream file(filename);
    if (!file.is_open()){
        return vector<Point>(); // return NULL;
    }

    vector<Point> p;
    int i = 0;
    string lineNumber;
    string x, y, z;

    getline(file, lineNumber); // skip the first line
    int size = atoi(lineNumber.c_str());

    while (file.good() && i++ < size){
        getline(file, x, ' ');
        getline(file, y, ' ');
        getline(file, z, '\n');

        Point point(atoi(x.c_str()), atoi(y.c_str()), atoi(z.c_str()));
        p.push_back(point);
//        p[i-1].Print();
    }
    return p;
}
```

Data set file is read line by line. First line gives us the total line number. According the total line number read is completed and data is loaded to vector dynamically.

ClosestPair::findClosestPair():

This function takes 2 parameters:
1. Sorted point list by x coordinate
2. Sorted point list by y coordinate

It found the size of point list and also found the value of the x coordinate of middle point. By this way, the list is divided by 2 as left and right. As the same way, sorted list by y coordinate is divided by 2 as left and right. If list has 3 or less element, it calls and returns bruteForce method instead of these steps. Function calls itself twice by taking parameters as the left lists and the right lists. After these steps, minimum distances of two lists are found. Which one is smaller, it is used in the leftover the function.
In continuation, middle are is scanned and if there is smaller distance between points in left and right list, closest pair object is updated. Function returns the closest pair object.

```
ClosestPair * ClosestPair::findClosestPair(vector<Point> px, vector<Point> py){
    unsigned long size = px.size();

    if (size <= 3){
        return bruteForce(px);
    }

    int mid = ceil(size/2.0);
    int p_mid = px[mid - 1].getX();

    vector<Point> xl;
    vector<Point> xr;
    vector<Point> yl;
    vector<Point> yr;

    for (int i = 0; i < mid; i++){
        xl.push_back(px[i]);
    }
    for (int i = mid; i < size; i++){
        xr.push_back(px[i]);
    }

    for (vector<Point>::iterator y = py.begin(); y != py.end(); y++){
        if(y->getX() <= p_mid){
            yl.push_back(*y);
        }
        else{
            yr.push_back(*y);
        }
    }

    ClosestPair * dl = findClosestPair(xl, yl);
    ClosestPair * dr = findClosestPair(xr, yr);
    if(dl->minDistance < dr->minDistance){
        this->minDistance = dl->minDistance;
        this->p1 = dl->p1;
        this->p2 = dl->p2;
    }
    else{
        this->minDistance = dr->minDistance;
        this->p1 = dr->p1;
        this->p2 = dr->p2;
    }
```

```
    int totalPointNumberInY = 0;
    vector<Point> ys;
    for (vector<Point>::iterator y = py.begin(); y != py.end(); y++){
        if (abs(mid - y->getX()) < p_mid){
            ys.push_back(*y);
            totalPointNumberInY++;
        }
    }

    ClosestPair * cp = this;

    int k;
    for (int i = 0; i < totalPointNumberInY; i++){
        k = i + 1;
        while (k < totalPointNumberInY && (ys[k].getY() - ys[i].getY()) < this->minDistance){
            if (ys[k].getDistance(ys[i]) < cp->minDistance){
                cp->p1 = ys[k];
                cp->p2 = ys[i];
                cp->minDistance = ys[k].getDistance(ys[i]);
            }
            k++;
        }
    }
    return cp;
}
```

ClosestPair::bruteForce():

```
ClosestPair * ClosestPair::bruteForce(vector<Point> p){
    unsigned long size = p.size();
    float distance = 0;
    for (int i = 0; i < size; i++){
        for (int j = i + 1; j < size; j++){
            distance = p[i].getDistance(p[j]);
            if (minDistance == 0 || distance < minDistance){
                this->p1 = p[i];
                this->p2 = p[j];
                this->minDistance = distance;
            }
        }
    }
    return this;
}
```

This method find the minimum distance by brute force. Each point is compared by others that are bigger.

ClosestPair::Print():

```
void ClosestPair::Print(){
    cout << "Closest points are given below." << endl;
    cout << "1st point -> ";
    this->p1.Print();
    cout << "2nd point -> ";
    this->p2.Print();
    cout << "distance = " <<  this->minDistance << endl;
}
```

This function prints the points coordinates and minimum distance.

3. How does the algorithm work?

Pseudocode:

```
closestPair of (px, py)
   where px is P(1) .. P(N) sorted by x coordinate, and
         py is P(1) .. P(N) sorted by y coordinate (ascending order)

if N ≤ 3 then
  return closest points of px using brute-force algorithm
else
  p_mid ← px(⌈N/2⌉)x
  ClosestPair * xl ← points of px from 1 to ⌈N/2⌉
  ClosestPair * xr ← points of px from ⌈N/2⌉+1 to N
  ClosestPair * yl ← { p ∈ py : px ≤ p_mid }
  ClosestPair * yr ← { p ∈ py : px > p_mid }

  ClosestPair * dl ← closestPair of (xl, yl)
  ClosestPair * dr ← closestPair of (xr, yr)

  if dl->minDistance < dr->minDistance then
    this ← dl
  else
    this ← dr
  endif

  ys ← { p ∈ yP : |p_mid - px| < this->minDistance }
  totalPointNumberInY ← number of points in ys
  ClosestPair * cp ← this
  for i from 0 to totalPointNumberInY - 1
    k ← i + 1
    while k < totalPointNumberInY and ys(k)y - ys(i)y < this-
>minDistance
      if |ys(k) - ys(i)| < this->minDistance then
        cp->p1 ← ys(k)
        cp->p1 ← yS(i)
        cp->minDistance ← |ys(k) - ys(i)|
      endif
      k ← k + 1
    endwhile
  endfor
  return cp
endif
```

Master theorem: $T(n) = a\ T(n/b) + cn$

number of subproblems = 2
size of subproblems = size / 2

a = 2, b = 2
$T(n) = 2\ T(n/2) + cn$

$$f(n) = O(n^{log_b a}) = O(n^{log_2 2}) = O(n^1) = O(n) \rightarrow case2$$

$$k = 0, a = 2, b = 2 \rightarrow T(n) = \theta(n^{log_b a} \log^{k+1} n) = \theta(n \log n)$$

4. Analyze

Closest Pair Problem Solution:

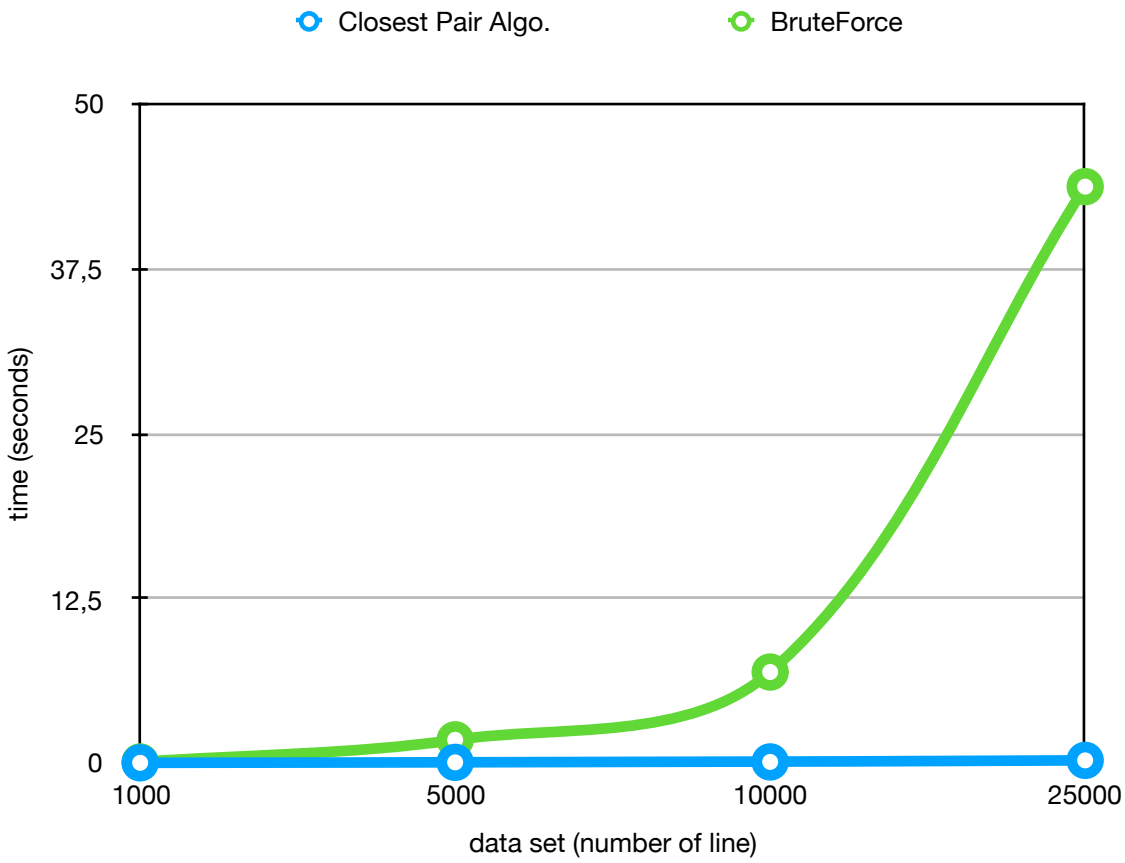
```
Last login: Mon Apr  9 13:53:46 on console
[Tugba-MacBook-Pro:~ tugba$ cd Documents/itu/BLG336E\ -\ Analysis\ of\ Algorithms\ II/Projects/Proje2/
Tugba-MacBook-Pro:Proje2 tugba$ g++ main.cpp -o project2
[Tugba-MacBook-Pro:Proje2 tugba$ ./project2 data1000.txt
[Closest points are given below.
1st point -> x:405      y:273   z:2222
2nd point -> x:399      y:260   z:2213
distance = 16.9115

Time elapsed: 0.006942 seconds.

Tugba-MacBook-Pro:Proje2 tugba$ ./project2 data5000.txt
[Closest points are given below.
1st point -> x:8407     y:8539  z:3465
2nd point -> x:8395     y:8515  z:3491
distance = 37.3631

Time elapsed: 0.041643 seconds.

Tugba-MacBook-Pro:Proje2 tugba$ ./project2 data10000.txt
[Closest points are given below.
1st point -> x:6321     y:8119  z:12847
2nd point -> x:6317     y:8119  z:12817
distance = 30.2655

Time elapsed: 0.075441 seconds.

Tugba-MacBook-Pro:Proje2 tugba$ ./project2 data25000.txt
[Closest points are given below.
1st point -> x:24157    y:51275 z:22882
2nd point -> x:24175    y:51244 z:22899
distance = 39.6737

Time elapsed: 0.193135 seconds.

Tugba-MacBook-Pro:Proje2 tugba$
```

Brute Force Solution:

```
Last login: Mon Apr  9 14:18:20 on ttys000
[Tugba-MacBook-Pro:~ tugba$ cd Desktop/
[Tugba-MacBook-Pro:Desktop tugba$ g++ main.cpp -o project2
[Tugba-MacBook-Pro:Desktop tugba$ ./project2 data1000.txt
 Closest points are given below.
 1st point -> x:399      y:260   z:2213
 2nd point -> x:405      y:273   z:2222
 distance = 16.9115

 Time elapsed: 0.075172 seconds.

[Tugba-MacBook-Pro:Desktop tugba$ ./project2 data5000.txt
 Closest points are given below.
 1st point -> x:8395     y:8515  z:3491
 2nd point -> x:8407     y:8539  z:3465
 distance = 37.3631

 Time elapsed: 1.75383 seconds.

[Tugba-MacBook-Pro:Desktop tugba$ ./project2 data10000.txt
 Closest points are given below.
 1st point -> x:6317     y:8119  z:12817
 2nd point -> x:6321     y:8119  z:12847
 distance = 30.2655

 Time elapsed: 6.90168 seconds.

[Tugba-MacBook-Pro:Desktop tugba$ ./project2 data25000.txt
 Closest points are given below.
 1st point -> x:24157    y:51275 z:22882
 2nd point -> x:24175    y:51244 z:22899
 distance = 39.6737

 Time elapsed: 43.7592 seconds.

 Tugba-MacBook-Pro:Desktop tugba$
```

Time (seconds)

	1000	5000	10000	25000
Closest Pair Alg.	0,006942	0,041643	0,075441	0,193135
BruteForce	0,075172	1,75383	6,90168	43,7592



The green line is result of brute force and the blue one is result of closest pair problem. The green line is increasing logarithmically but the blue one is increasing almost linearly.

Closest pair problem time complexity is O(nlogn)
Brute force time complexity is $O(n^2)$