**ISTANBUL TECHNICAL UNIVERSITY**

**COMPUTER ENGINEERING DEPARTMENT**

**BLG 336E**

# Analysis of Algorithms II HOMEWORK REPORT

| | | |
|---|---|---|
| **HOMEWORK NO** | : | 2 |
| **HOMEWORK DATE** | : | 18.04.2023 |

**150200915 : ADIL MAHMUDLU**

**Sprint 2023**

# Contents

# 1 EXPLANATION OF THE CODE AND THE SOLUTION

## 1.1 Convex Hull Solution

The Convex Hull Problem requires reading coordinates of the c cities from input file. The algorithm sorts the cities based on low-x coordinate, and then recursively partitions the cities list until at most 5 cities are left in each partition. Then, using brute force, convex hull of each of partitions are found and returned back for merging. Merging uses Bitangent Hull Merging method to find the merged hull. Bitangent Hull Merging finds upper and lower tangents of the partitioned hulls and merges them using these tangents. As this is recursion, all the partitions are merged until the whole points system's convex hull is found. This is called Divide and Conquer algorithm for Convex Hull problem. The program runs in $O(nlogn$ time and $O(n)$ space complexity. The functions that are necessary for implementation are:

- quadrant
- orientation
- compare
- merge
- bruteForce
- partition

### 1.1.1 quadrant function

```
function quadrant (a)
    if a.x >= 0
        if a.y >= 0
            return 1
        else
            return 4
    else
        if a.y >= 0
            return 2
        else
            return 3
```

The function return the quadrant of the point. It runs in $O(1)$ time complexity.

### 1.1.2 orientation function

```
function orientation (a, b, c)
    side <- (b.x - a.x)*(c.y - a.y) - (c.x - a.x)*(b.y - a.y)
    if side == 0
        return 0
    if side > 0
        return 1
    return -1
```

The function is utilized to find the orientation of the point c with respect to points a and b. The function uses cross product to find, standing on point a looking at point b, if point c is on the left, right, or on top of the ab line. Running time complexity and space complexity is $O(1)$.

### 1.1.3 compare function

```
function compare (a, b)
    aa <- {a.x - centet.x, a.y - center.y}
    bb <- {b.x - center.x, b.y - center.y}
    aq <- quadrant(aa)
    bq <- quadrant(bb)
    if aq != bq
        return aq < bq
    return aa.y*bb.x < bb.y*aa.x
```

Compare function, as its name suggests, compares to points according to their quadrant and degree from center. It puts the point with lower quadrant before the other, and if they have the same quadrant, puts the point with lower degree from center before the other. Space and time complexity is $O(1)$.

### 1.1.4 merge function

```
merge (a, b)
    na <- a.size, nb <- b.size
    ra <- 0, lb <- 0
    for i from 1 to na
        if a[i].x > a[ra].x
        ra <- i

for i from 1 to nb
    if b[i].x < b[lb].x
        lb <- i

inda <- ra, indb <- lb
done <- 0
while !done
    done <- 1
    while orientation(b[indb], a[inda], a[(inda+1)%na]) <= 0
        inda <- (inda+1) % na
    while orientation(a[inda], b[indb], b[(nb+indb-1)%nb]) >= 0
        indb <- (nb+indb-1) % nb
        done <- 0
uppera <- inda, upperb <- indb

inda <- ra, indb <- lb
done <- 0
while !done
    done <- 1
    while orientation(a[inda], b[indb], b[(indb+1)%nb]) <= 0
        indb <- (indb+1) % nb

    while orientation(b[indb], a[inda], a[(na+inda-1)%na]) >= 0
        inda <- (na+inda-1) % na
        done <- 0
lowera <- inda, lowerb <- indb
ind <- uppera
hull.insert(a[uppera])
while ind != lowera
    ind <- (ind+1)%na
    hull.insert(a[ind])
ind <- lowerb
hull.insert(b[lowerb])
while ind != upperb
    ind <- (ind+1)%nb
    hull.insert(b[ind])
leftmost <- 0
for i from 1 to hull.size
```

```
        if hull[i].x < hull[leftmost].x
            leftmost <- i
    rotate(hull, leftmost)
    return hull
```

Given function merges two convex hulls. As stated before, function first finds the upper and lower tangent and then constructs merged convex hull from partitions' hull points. It runs in $O(n1 + n2)$ time where n1 and n2 are the number of edges of the original convex hull polygons. Space complexity is $O(n1 + n2)$ as well, as the function creates a new convex hull vector of size n1+n2.

#### 1.1.5 bruteForce function

```
bruteForce (a)
    s <- set of long long pairs
    m <- a.size
    for i from 0 to m
        for j from i to m
            pos <- 0
            neg <- 0
            for k from 0 to m
                if orientation(a[i], a[j], a[k]) <= 0
                    neg <- neg+1
                if orientation(a[i], a[j], a[k]) >= 0
                    pos <- pos+1
            if pos == m or neg == m
                s.insert(a[i])
                s.insert(a[j])
    hull <- vector of long long pairs
    for e in s
        hull.insert(e)
    center <- {0, 0}
    n <- hull.size
    for i from 0 to n
        center.x = center.x + hull[i].x
        center.y = center.y + hull[i].y
        hull[i].x = hull[i].x * n
        hull[i].y = hull[i].y * n
    sort(hull)
    for i from 0 to n
```

```
        hull[i] <- {hull[i].x/n, hull[i].y/n}
    leftmost <- 0
    for i from 1 to n
        if hull[i].x < hull[leftmost].x
            leftmost <- i
    rotate(hull, leftmost)
    return hull
```

The bruteForce function goes through all the points in a given partition and checks whether a pair of points create an edge. It works by checking if all the points are in one side (or on top) of the line through the pair of points. It works in $O(n^3)$ time complexity where n is the number of points in the partition. But nevertheless, function is never executed above n=5, so it doesn't affect the performance. Space complexity is $O(1)$.

### 1.1.6 partition function

```
partition (a)
    n <- a.size
    if n <= 5
        return bruteForce(a)
    left, right <- vector of long long pairs
    for i from 0 to n/2
        left.insert(a[i])
    for i from n/2 to n
        right.insert(a[i])
    leftSide <- partition(left)
    rightSide <- partition(right)
    return merge(leftSide, rightSide)
```

The function recursively splits the points in two and calls itself until there are less than 6 points in a partition. Then, returning partitions are merged to create the merged convex hull. There are logn partitions and supposing merge function runs in $O(n)$, the running time complexity of the partition function is $O(nlogn)$. Space complexity on the other hand, is $O(n)$, given that it only splits points to left and right parts with n total points.

## 1.2 Prim Solution

Part 2 of the homework requires reading source bakeries and pounts of the cities from the input file. It is necessary to first calculate plow number for each pairs of cities and create a graph. Then, turn by turn, each bakery goes to the highest plow in neighbour city and if there are no bakeries there, opens up a branch in that city. This creates multiple ”forests” that are parts of Minimum Spanning Tree for the graph. The program runs in $O(n^2)$ time and space complexity. These are the functions necessary for the algorithm:

- createGraph
- bQueueEmpty
- prim

### 1.2.1 createGraph function

```
createGraph (pounts, adjMatrix, th)
    for i from 0 to pounts.size
        for j from i+1 to pounts.size
            plow <- abs(pounts[i] - pounts[j]
            if plow <= th * ((pounts[i] + pounts[j])/2) and plow != 0
                adjMatrix[i].insert{plow, j}
                adjMatrix[j].insert{plow, i}
```

The given function fills the adjacency matrix. Connects the cities provided the necessary requirement is met. Running time is $O(n^2)$, while space complexity is $O(n + e)$, where n is the number of cities and e is the number of edges between them.

### 1.2.2 bQueueEmpty function

```
bQueueEmpty (bQueue)
    for i from 0 to bQueue.size
        if !bQueue[i].empty
            return 0
    return 1
```

Function check every bakery’s priority queue to see if all of them are empty, and returns 1 if they are. Time complexity is $O(b)$ where b is the number of bakeries. Space complexity is $O(1)$.

### 1.2.3 prim function

```
void prim (bQueueadjMatrix, branches)
    b <-bQueue.size
    c <- adjMatrix.size
```

```
visited <- vector of size c initialized to 0s
turn <- 0
while !bQueueEmpty(bQueue)
    while !bQueue[turn].empty
        i <- bQueue[turn].top().city
        bQueue[turn].pop()
        if visited[i]
            continue
        branches[turn].insert(i)
        visited[i] <- 1
        for e in adjMatrix[i]
            plow <- e.plow
            city <- e.city
            if !visited[city]
                bQueue[turn].push(plow, city)
        break
    turn <- (turn + 1) % b
```

The core part of the program is in prim function. The bakeries take turns spreading, and the function gets the highest plow numbered neighbour of the cities current bakery has branched. This is carried out until the city we are looking at is not visited. Then the city is added to branches list of the bakery and its neighbours are added to priority queue of the bakery. The process is continued until there are no more cities left in any of the priority queues. The time complexity of the function is $O(belogn)$ where b is the number of bakery sources, n is the number of cities, and e is the number of roads between them. Space complexity is $O(n + e + b)$.

## 2 Why should you use the divide and conquer approach for this problem? How does this affects the complexity of the Convex-Hull problem?

There are practically three ways of solving Convex-Hull problem: Divide and Conquer, Brute Force, and Graham Scan.Normally, both divide and conquer method and graham scan method run on $O(nlong)$ time complexity, while brute force method runs in $O(n^3)$ time complexity. We used divide and conquer that uses brute force on small sample sizes, which yields $O(nlogn)$ as brute force on small sample size does not affect the performance.

## 3 How does increasing the number of the cities affects the memory complexity of the algorithms?

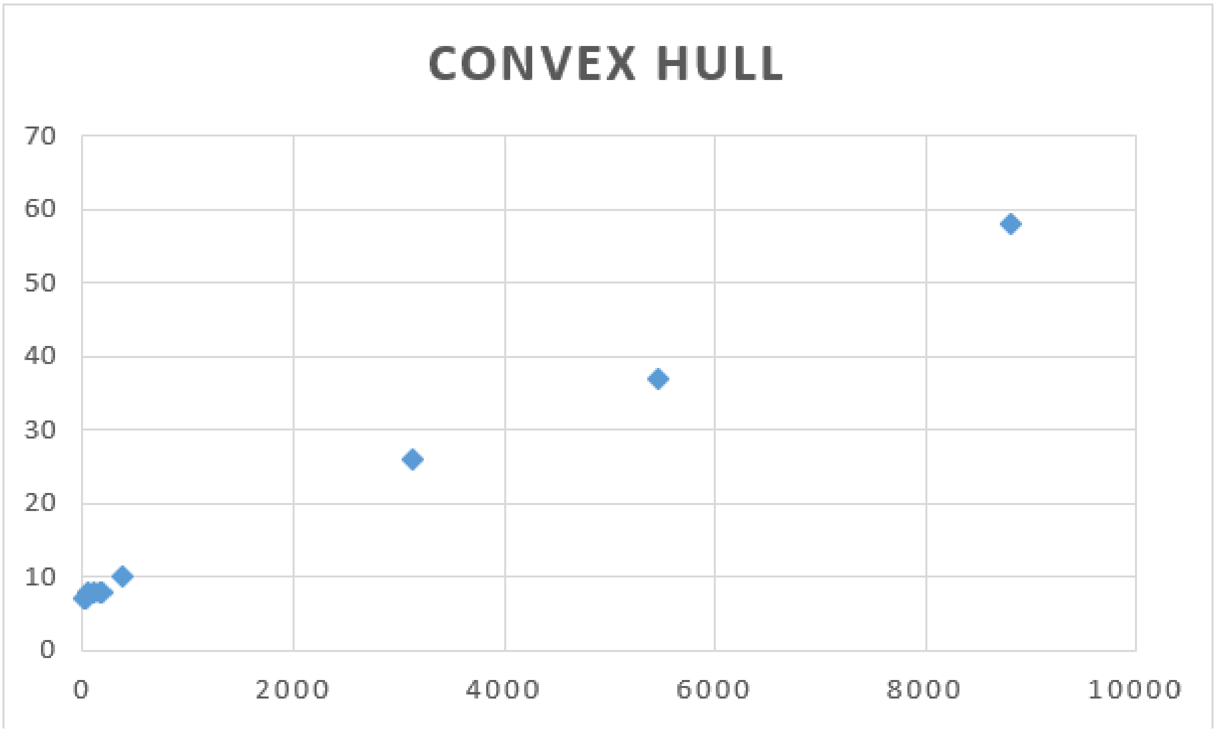

Figure 1: Scatter plot of Convex Hull runtime in ms/number of cities

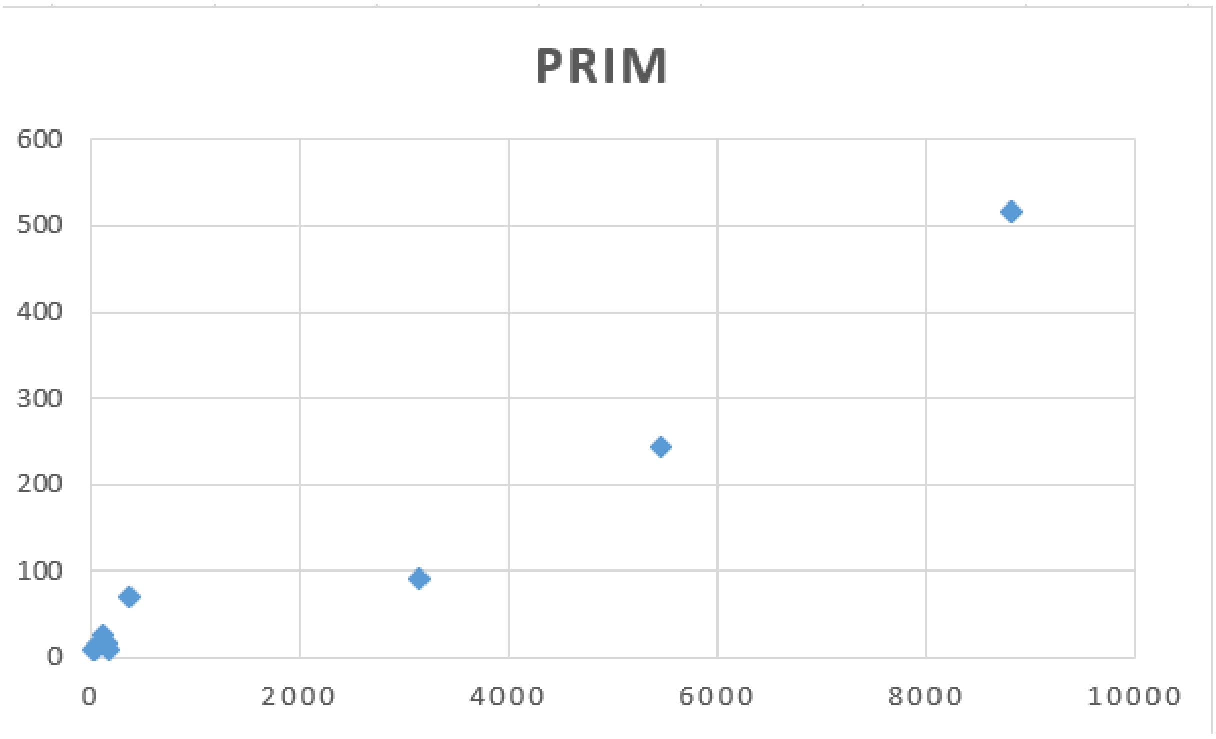

Figure 2: Scatter plot of Prim runtime in ms/number of cities