COL351
Assignment 3

Mallika Prabhakar (2019CS50440)
Sayam Sethi (2019CS10399)

September 2021

Contents

1 Question 1

Question 1

Question. *The Convex Hull of a set P of n points in $x - y$ plane is a minimum subset Q of points in P such that all points in P can be generated by a convex combination of points in Q. In other words, the points in Q are corners of the convex-polygon of smallest area that encloses all the points in P. Design an $O(n \log n)$ time Divide-and-Conquer algorithm to compute the convex hull of a set P of n input points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.*

Solution. We implement a divide and conquer strategy. We divide the set of points P in 2 almost equal groups namely P_1 and P_2 and recursively compute their convex hull. Base case is when the number of points to be divided is 5 or less. We use the number 5 as a base case since no convex hull exists for 2 points. P_1 and P_2 are merged by finding the upper and lower tangents of the mentioned polygons. The merger results in generating the convex hull Q of all the points in the set $P_1 \cup P_2$.

Algorithm 1 Merge algorithm for divide and conquer solution for convex hull

procedure MERGE(P_1, P_2)
    $n_1 \leftarrow size(P_1), n_2 \leftarrow size(P_2)$
    $i_1 \leftarrow 0, i_2 \leftarrow 0$
    for all i in $range(n_1)$ **do** ▷ rightmost point in P_1
        if $P_1[i] > P_1[i_1]$ **then**
            $i_1 \leftarrow i$
        end if
        for all i in $range(n_2)$ **do** ▷ leftmost point in P_2
            if $P_2[i] < P_2[i_2]$ **then**
                $i_2 \leftarrow i$
            end if
            $index_1 \leftarrow i_1, index_2 \leftarrow i_2$
            $L \leftarrow line joining i_1 and i_2$
            while L crosses any polygon **do** ▷ upper tangent calculation
                while L crosses P_1 **do**
                    $index_1 \leftarrow (index_1 + 1) \mod n_1$ ▷ $index_1$ moves up since counter-clockwise
                end while
                while L crosses P_2 **do**
                    $index_2 \leftarrow (n_2 + index_2 - 1) \mod n_2$ ▷ $index_2$ moves up since counter-clockwise
                end while
            end while
            $upper_1 \leftarrow index_1, upper_2 \leftarrow index_2$
            $index_1 \leftarrow i_1, index_2 \leftarrow i_2$
            $L \leftarrow line joining i_1 and i_2$
            while L crosses any polygon **do** ▷ lower tangent calculation
                while L crosses P_2 **do**
                    $index_2 \leftarrow (index_2 + 1) \mod n_2$ ▷ $index_2$ moves down since counter-clockwise
                end while
                while L crosses P_1 **do**
                    $index_1 \leftarrow (n_1 + index_1 - 1) \mod n_1$ ▷ $index_1$ moves down since counter-clockwise
                end while
            end while

```
        lower_1 ← index_1, lower_2 ← index_2
        Q' is initialised          ▷ Q' contains convex hull in a counter-clockwise manner
        Q'.add(P_1[upper_1])
        index ← upper_1
        while index ≠ lower_1 do
            index ← (index + 1) mod n_1
            Q'.add(P_1[index])
        end while
        Q'.add(P_2[lower_2])
        index ← lower_2
        while index ≠ upper_2 do
            index ← (index + 1) mod n_2
            Q'.add(P_2[index])
        end while
        return Q'
```

Algorithm 2 Merge algorithm for divide and conquer solution for convex hull

```
procedure DIVIDE(P')
    if size(P') then
        return

        procedure ALGORITHM(P)
            sort(P)                         ▷ sorted according to x-coordinate
            Q ← divide(P)
            return Q
        end procedure
```

□

2 Question 2

Question 2

Question. *The total net force on particle j, by Coulomb's Law, is equal to*

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2} \tag{1}$$

Design an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Solution. We will use polynomial multiplication to solve this question. Consider the polynomials:

$$\begin{aligned} A(x) &= (0, q_1, q_2, \ldots, q_n) \\ B(x) &= \left(-\frac{1}{(n-1)^2}, -\frac{1}{(n-2)^2}, \ldots, -\frac{1}{1^2}, 0, \frac{1}{1^2}, \ldots, \frac{1}{(n-2)^2}, \frac{1}{(n-1)^2}\right) \end{aligned} \tag{2}$$

In the above representation, only the coefficients of $A(x), B(x)$ are shown. The degrees of $A(x)$ and $B(x)$ are n and $2n-2$ respectively. Now, in the product $P(x) = A(x) \cdot B(x)$, consider the coefficient of x^{j+n-1}. To visualise this, we will write the polynomials as:

$$\begin{aligned} A(x) &= \quad q_nx^n + \cdots + \quad q_{j+1}x^{j+1} + \quad q_jx^j + \quad q_{j-1}x^{j-1} + \cdots + q_1x^1 + 0x^0 \\ B(x) &= \cdots + \quad -\frac{1}{(n-j)^2}x^{j-1} + \cdots + \quad -\frac{1}{1^2}x^{n-2} + \quad 0x^{n-1} + \quad \frac{1}{1^2}x^n + \cdots + \frac{1}{(j-1)^2}x^{j+n-2} \\ &\quad + \cdots \end{aligned} \tag{3}$$

Multiplication of corresponding terms gives terms with power of x as $j+n-1$, and thus formally, the coefficient of x^{j+n-1} can be written as:

$$P(x)[j+n-1] = \sum_{k=1}^{n-j} q_{j+k} \cdot -\frac{1}{k^2} + 0 + \sum_{k=1}^{j-1} q_{j-k} \cdot \frac{1}{k^2} \tag{4}$$

Where $P(x)[p]$ denotes the coefficient of x^p in $P(x)$.
Equation 4 can be rewritten as:

$$\begin{aligned} P(x)[j+n-1] &= \sum_{i=j+k,k=1}^{n-j} q_i \cdot -\frac{1}{(j-1)^2} + \sum_{i=j-k,k=1}^{j-1} q_i \cdot \frac{1}{(j-1)^2} \\ &= -\sum_{i=j+1}^{n} \frac{q_i}{(j-1)^2} + \sum_{i=1}^{j-1} \frac{q_i}{(j-1)^2} \\ &= \sum_{i<j} \frac{q_i}{(j-1)^2} - \sum_{i>j} \frac{q_i}{(j-1)^2} \\ &= \frac{F_j}{Cq_j} \\ \implies F_j &= P(x)[j+n-1] \times Cq_j \end{aligned} \tag{5}$$

Therefore, we have derived an alternate method for computing F_j. Since this involves computing product of polynomials, we can perform the polynomial product in $O(n \log n)$ since both $A(x), B(x)$ are polynomials of degree $O(n)$. Once $C(x)$ has been computed, we can then compute F_j in $O(1)$ for each j by dividing the corresponding coefficient with Cq_j. The exact algorithm is given as:

Algorithm 3 Computing F_j for $j \in \{1, 2, \ldots, n\}$

1: **procedure** COMPUTE FORCES((q, n))
2: $\quad A \leftarrow q$
3: $\quad B \leftarrow \left[-\frac{1}{(n-1)^2}, -\frac{1}{(n-2)^2}, \ldots, -\frac{1}{1^2}, 0, \frac{1}{1^2}, \ldots, \frac{1}{(n-2)^2}, \frac{1}{(n-1)^2}\right]$
4: $\quad P \leftarrow \mathit{multiply}(A, B)$ ▷ multiply $A(x)$ and $B(x)$ using FFT "divide and conquer" algo
5: $\quad F \leftarrow C[n : 2n-1]$ ▷ taking subarray corresponding to coefficients of x^{j+n-1}
6: $\quad$ **for** $i \in [1, n]$ **do** ▷ 1-indexed array
7: $\quad\quad F[i] \leftarrow F[i] \times Cq[i]$
8: $\quad$ **end for**
9: $\quad$ **return** F
10: **end procedure** =0

Proof of Correctness: The proof of correctness of the *FFT Algorithm* has been discussed in the lectures. The correctness of lines $5-8$ has been proved from Equation 5.

Time Complexity: All operations except the *FFT Algorithm* are $O(n)$ operations. The complexity of *FFT Algorithm* has been shown to be $O(d \log d)$ where d is the degree of the polynomial. Since the degrees of $A(x), B(x)$ are $O(n)$, the *FFT Algorithm* can be computed in $O(n \log n)$ time.

Therefore, all forces F_j can be computed in $O(n \log n)$ time. This completes the design of the algorithm along with proof of correctness and time complexity. □

3 Question 3

3.1 3.1

Question 3(a)

Question. *Prove that the graph $H = (V, E_H)$ can be computed from G in $O(n^\omega)$ time, where ω is the exponent of matrix-multiplication.*

Proof. Enumerate the vertices V in G as $\{1, 2, \ldots, |V| = n\}$ and let A_G be the adjacency matrix of G. Consider the term A_G^2. From **Lemma 1** of Lecture 22, we know that A_G^2 is positive only if there exists a walk of length *exactly* 2. Therefore, we have the following claim:

Claim 3.1. *The adjacency list for graph H is given as $A_G + A_G^2 \succ 0$, where A_G is adjacency matrix of G.*

Proof. From definition of H, we have that edges in graph H consists of all edges of graph G and end points of walks of length 2. Therefore, E_H has all edges of walks of length 1 and 2. In other words, $(A_H)_{ij}$ is positive only if there exists a walk of length 1 or 2 between nodes i, j. This can be formally written as:

$$\begin{aligned} (A_H)_{ij} &= (A_G)_{ij} > 0 \vee (A_G^2)_{ij} > 0 \\ &= (A_G)_{ij} + (A_G^2)_{ij} > 0 \\ \implies A_H &= A_G + A_G^2 \succ 0 \end{aligned} \tag{6}$$

□

Therefore, the algorithm for computing A_H is:

Algorithm 4 Computing H

procedure ComputeH(G)
  $A_G \leftarrow adjacency(G)$
  $A_H \leftarrow A_G + A_G^2$
  $n \leftarrow |V_G|$
  for $i, j \in [1, n] \times [1, n]$ **do**
    if $(A_H)_{ij} > 0$ **then**
      $(A_H)_{ij} \leftarrow 1$
    else
      $(A_H)_{ij} \leftarrow 0$
    end if
  end for
  return $graph(A_H)$
end procedure

Time Complexity Computing A_G^2 will take $O(n^\omega)$ time. All other steps take $O(n^2)$ time. We know that $\omega \geq 2$. Therefore, the overall time complexity of the algorithm will be $O(n^\omega)$. Therefore, we have proposed an algorithm which computes the graph H via its adjacency matrix in $O(n^\omega)$ time. This completes the proof. □

3.2 3.2

Question 3(b)

Question. *Argue that for any $x, y \in V$, $D_H(x, y) = \left\lceil \frac{D_G(x,y)}{2} \right\rceil$*

Solution. We will prove the given statement by first showing that there exists a path of length $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$ for each x, y in H. We will then prove that we cannot have a shorter path length in H.
Note: For this and subsequent parts, we call edges which are directly in G as edges of *type* 1 and the other edges as edges of *type* 2.

Claim 3.2. *For each $x, y \in V$, there exists a path of length $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$ in graph H, corresponding to the shortest path in G.*

Proof. Let the shortest path between x, y in G be given as:

$$\begin{aligned} P_G(x,y) &= \{x, a_1, a_2, \ldots, a_k, y\} \\ \implies D_G(x,y) &= k + 1 \end{aligned} \tag{7}$$

We now have two cases, when k is odd and when k is even. For the case when k is odd, we have:

$$\begin{aligned} P_H(x,y) &= \{x, a_2, a_4, \ldots, a_{k-1}, y\} \\ \implies \mathit{length}(P_H(x,y)) &= \frac{k-1}{2} + 1 \\ &= \frac{k+1}{2} \\ &= \left\lceil \frac{D_G(x,y)}{2} \right\rceil \end{aligned} \tag{8}$$

When k is even, we have:

$$\begin{aligned} P_H(x,y) = \{x, a_2, a_4, \ldots, a_k, y\} \;&((a_k, y) \text{ is the only edge of type 1}) \\ \implies \mathit{length}(P_H(x,y)) &= \frac{k}{2} + 1 \\ &= \frac{(k+1)+1}{2} \\ &= \left\lceil \frac{D_G(x,y)}{2} \right\rceil \end{aligned} \tag{9}$$

Therefore we have shown the correctness of the claim for both cases of k. ☐

We will now show that there cannot exist a path between x, y of shorter length in H.

Claim 3.3. *The shortest distance between x, y is given exactly as $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$*

Proof. We will prove the claim using contradiction. Assume that there exists a shorter path $Q_H(x,y)$:

$$
\begin{aligned}
Q_H(x,y) &= \{x, b_1, b_2, \ldots, b_m, y\} \\
\implies length(Q_H(x,y)) &= m + 1 < \left\lceil \frac{D_G(x,y)}{2} \right\rceil \text{, from assumption}
\end{aligned} \tag{10}
$$

Consider the edges in G corresponding to this path $Q_H(x,y)$:

$$
\begin{aligned}
Q_G(x,y) &= \{x, c_1, b_1, c_2, b_2, \ldots, c_m, b_m, c_{m+1}, y\}, \text{ } c_i \text{ may be the same as } b_i \\
\implies length(Q_G(x,y)) &\le 2m + 2 < 2\left\lceil \frac{D_G(x,y)}{2} \right\rceil \\
\implies length(Q_G(x,y)) &< \begin{cases} D_G(x,y) + 1, & D_G(x,y) \text{ is odd} \\ D_G(x,y), & D_G(x,y) \text{ is even} \end{cases}
\end{aligned} \tag{11}
$$

We know that $D_G(x,y)$ is the shortest path in G between vertices x, y. Therefore, we have that such a path cannot exist if $D_G(x,y)$ is even and in the case when $D_G(x,y)$ is odd, we notice that the inequality in $length(Q_G(x,y))$ has an even number $(2m+2)$ in the RHS. Therefore, the equality cannot hold in this case as well. Thus, we have arrived at a condradiction on the length of shortest x, y path in G. Therefore, $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$ is the shortest path in H. □

Thus, from Claim 3.2 and Claim 3.3 we have shown that $D_H(x,y) = \left\lceil \frac{D_G(x,y)}{2} \right\rceil$.
Hence, proved. □

3.3 3.3

Question 1

Question. *Question*

Solution.

Claim 3.4.

Proof. □

Algorithm 5 DP solution for partitioning

1:

□

3.4 3.4

Question 3(d)

Question. *Use Question 3.3 to argue that D_G is computable from D_H in $O(n^\omega)$ time.*

Proof. We will first propose the algorithm and then prove its correctness and time complexity.

Algorithm 6 Computing D_G from D_H

1: **procedure** ComputeDg(G, D_H)
2: $\quad M \leftarrow D_H \times adjacency(G)$
3: $\quad D_G \leftarrow init()$
4: $\quad$ **for** $x \in V$ **do**
5: $\qquad$ **for** $y \in V$ **do**
6: $\qquad\quad$ **if** $M(x, y) \geq \deg(G, y) \cdot D_H(x, y)$ **then**
7: $\qquad\qquad D_G(x, y) \leftarrow 2D_H(x, y)$
8: $\qquad\quad$ **else**
9: $\qquad\qquad D_G(x, y) \leftarrow 2D_H(x, y) - 1$
10: $\qquad\quad$ **end if**
11: $\qquad$ **end for**
12: $\quad$ **end for**
13: $\quad$ **return** D_G
14: **end procedure**

Algorithm 6 computes the matrix D_G using the idea proven in Question 3.4. Therefore, from the proof given in Question 3.4, we can compute D_G.
Time Complexity Line 2 in Algorithm 6 takes $O(n^\omega)$ time. The nested for loop takes $O(n^2)$ time since each iteration takes $O(1)$ time. Therefore the total running time is $O(n^\omega)$ ($\omega > 2$).
Therefore, we have used the proof of Question 3.4 to arrive at an $O(n^\omega)$ solution for computing D_G. This completes the proof. $\square$

3.5 3.5

Question 3(e)

Question. *Prove that all-pairs-distances in n-vertex unweighted undirected graph can be computed in $O(n^\omega \log n)$ time, if ω is larger than two.*

Solution. We propose the following algorithm for computing all-pairs-distances:

Algorithm 7 Computing all-pairs-distances

```
1: procedure AllPairDistances(G)
2:     A_G ← adjacency(G)
3:     H ← ComputeH(G)
4:     if H = G then
5:         D_G ← A_G
6:         D_G ← all off-diagonal zero entries are set to ∞
7:         return D_G
8:     end if
9:     D_H ← AllPairDistances(H)
10:    D_G ← ComputeDg(G, D_H)
11:    return D_G
12: end procedure
```

This is a recursive algorithm that we use to compute the all-pairs-shortest distances. To prove the same, we will prove the correctness of the algorithm using reverse induction on the depth of the recursive calls.

Base Case If H is the same as G, then each component in G is fully connected. Therefore, the distance matrix will be the same as the adjacency matrix and the off-diagonal entries that are 0 will be ∞ since there is no path between such vertices.
Inductive Step We assume that it is true for depth $i+1$, now consider the call at depth i. We have already shown the correctness of line $3, 10$ in Question 3.1 and Question 3.4 respectively. Additionally from the inductive assumption, we know that D_H is indeed the distance of graph H. Therefore, our recursive algorithm is correct.

However, we still have to prove termination. To do the same, we notice that any two vertices that have a path between them have a path of length $< n$. Additionally, the distance halves at each step as proved in Question 3.1. Therefore, the algorithm terminates in $O(\log n)$ calls.
Time Complexity As stated above, the number of calls to `AllPairDistances` is $O(\log n)$. Each call of the function takes $O(n^\omega)$ time as shown in Question 3.1 and Question 3.4. Therefore, the total time complexity of the algorithm is $O(n^\omega \log n)$.
This completes the algorithm along with proof of correctness and time complexity. $\square$

4 Question 4

4.1 4.1

Question 1

Question. *Question*

Solution.

Claim 4.1.

Proof. □

Algorithm 8 DP solution for partitioning

1:

□

4.2 4.2

Question 1

Question. *Question*

Solution. Since M is fairly large, we will assume that $p \approx M$ and that M is a multiple of n. Since we have to show inequality, choosing the lower bound of p works since the term only increases for larger p. Therefore, $rx \bmod p$ maps exactly to a set which is the same as $\{0, 2, \ldots, p-1 \approx M-1\}$ (from lectures 26, 27). Additionally, since we have assumed that M is a multiple of n, we get that:

$$y \bmod n = \{0, 1, \ldots, n-1, 0, 1, \ldots, n-1, (M/n \; times) \ldots, 0, 1, \ldots, n-1\} \tag{12}$$

Where y is the mapping of $\{0, 2, \ldots, M-1\}$ via the function $rx \text{mod} p$. Now, each *bucket* contains atleast M/n elements. Therefore, taking n elements from any of this bucket will give a maximum chain length of $n = \Theta(n)$. There are $n \cdot {}^{M/n}C_n$ such subsets of U. The above arguments are for $p \approx M$. If $p > M$, the number of such subsets will only increase.
Therefore, we have shown that there are *atleast* ${}^{M/n}C_n$ such subsets of U of size n with maximum chain length in $H_r()$ is $\Theta(n)$. □

4.3 4.3

Question 4(c)

Question. *Implement $H()$ and $H_r()$ in Python/Java for $M = 10^4$ and the following different choices of sets of size $n = 100$: For $k \in [1, n]$, S_k is union of $\{0, n, 2n, 3n, \ldots, (k-1)n\}$ and $n - k$ random elements in U.*
Obtain a plot of Max-chain-length for hash functions $H()$, $H_r()$ over different choices of sets S_k defined above. Note that you must choose a different random r for each choice of S_k. Provide a justification for your plots.

Solution. We observe that the maximum chain length increases almost linearly for $H()$ while it remains approximately constant for $H_R()$. This is because as k increases, the set S_K becomes less and less random. $H(x) = 0$ for $x = i \cdot n$ and this is what amounts to the maximum chain length as k increases. However, in the case of $H_r()$, the set S_k is transformed to a random set and therefore the maximum chain length remains constant with an exepctation of 2.

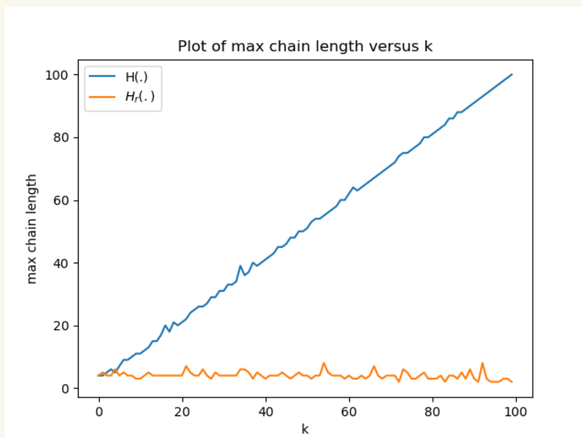

Figure 1: Plot

□