COL351
Assignment 3

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

September 2021

Contents

1 Question 1 **2**

2 Question 2 **6**

3 Question 3 **8**
3.1 3.1 8
3.2 3.2 9
3.3 3.3 10
3.4 3.4 11
3.5 3.5 12

4 Question 4 **14**
4.1 4.1 14
4.2 4.2 14
4.3 4.3 15

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
    end for
    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
    end for
    $index_1 \leftarrow i_1, index_2 \leftarrow i_2$
    $L \leftarrow linejoiningi_1andi_2$
    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 $index_1$ and $index_2$

```
    while L crosses any polygon do                              ▷ lower tangent calculation
        while L crosses P_2 do
            index_2 ← (index_2 + 1) mod n_2   ▷ index_2 moves down since counter-clockwise
        end while
        while L crosses P_1 do
            index_1 ← (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
end procedure
```

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

```
procedure DIVIDE(P)                              ▷ P is sorted according to x-coordinate
    if size(P) ≤ 5 then
        return basecase(P)                       ▷ base case algorithm is defined below
    end if
    P_1 ← first half of P
    P_2 ← second half of P
    P_1_hull ← divide(P_1)
    P_2_hull ← divide(P_2)
    Q ← merge(P_1_hull, P_2_hull)
    return Q
end procedure
```

Algorithm 3 base case for divide

```
procedure BASECASE(P)    ▷ Because of less number of points, we directly check if an edge is a part of the convex hull or not
    initialise set    ▷ Set of all points in the hull
    for all i in range(size(P)) do
        for all j in range(i + 1, size(P)) do
            p ← P[i] and q ← P[j]
            a ← p.y − q.y
            b ← q.x − p.y
            c ← p.x × q.y − p.y × q.x
            neg, pos ← 0
            for all k in range(size(P)) do
                check on which side of line does the point line
                increment neg and/or pos corresponding above
            end for
            if pos = size(P) or neg = size(P) then
                set.add(P[i])
                set.add(P[j])
            end if
        end for
    end for
    Q is initialised as list of set
    middle ← [0, 0]
    for all iinrange(size(Q)) do


    end for

end procedure
```

Proof of termination:

Merge step: given P_1 and P_2, they are disjoint and have finite number of points $n_2 and n_2$ which means that the for loop runs are finite. Using all the points in P_1 and P_2 we find the upper and lower tangent. After obtaining the tangents, we calculate a set of points Q by iterating over at maximum $n_1 + n_2$ points by using 2 while loops. Q is the convex hull for the set $P_1 \cup P_2$.

For calculating the upper and lower tangents, we use nested while loops. In our implementation of finding tangents, we slowly go through pairs by changing one point at a time if the line intersects a polygon. All possible pairs are finite in number and we use its subset which is finite as well. Hence it terminates.

Divide step: if $size(P)$ is n, then the number of times divide is performed is finite and of the order $O(\log n)$. It is finite as n constantly gets divided into two parts and dividing stops at 5 or lesser number of points being considered. Base case computation also terminates in constant time (roughly small multiple of 5^3 computations) since it can be computed using basic mathematical formulae.

Following both merge and divide step explanations, the algorithm ultimately stop because divide step happens roughly $\log n$ times and every time the elements considered for merging are finite and not repeated. Hence the algorithm terminates.

Time complexity: We follow divide and conquer strategy by dividing P into P_1 and P_2 almost equal halves. Base case computational time complexity is constant as mentioned before. At every level, we perform merge step. As discussed in Proof of termination, the main computations performed are calculating tangents and actual merging.
Merging takes $O(n)$ time in worst case scenario i.e. all points in P_1 and P_2 are considered. For tangent calculation, we start at leftmost and rightmost points and travel up (for upper tangent) and down (for lower tangent). We perform this travelling by going from one point in P_1 to one point in P_2 guaranteed by the for and while loops. In worst case, we have to go through all of P_1 and P_2 which takes $O(n)$ time.
Hence merging takes overall $O(n)$ time.

$$T(n) = 2 \times T(n/2) + O(n) \tag{1}$$

Using master theorem, $a = 2$ and $b = 2$
$c = \log_b a = 1$
Hence final time complexity of algorithm becomes: $O(n \log n)$

Proof of correctness: We follow a simple divide and conquer algorithm with base case as defined above for $size(P) \leq 5$. Following is the proof of correctness of the forementioned algorithm using induction:
Base Case:

Induction Step:
We assume that the hypothesis is true for all $i < n$. Consider the merge step for n. The left and right hulls have sizes around $n/2$. The invariant of the merge function is that the points are returned in anti-clockwise order. We first compute the rightmost point in P_1 and the leftmost point in P_2.
Then, we find pair of points on the top of the hull so that the *merged* hull is convex. To compute the same, we keep on moving in counter-clockwise order in P_1 until we find a pair of points such that the line is *tangential* to P_1. We then check if this line is also *tangential* to P_2 (by *tangential*, we mean that the line doesn't cross the convex hull and intersects the polygon only at the vertex). If the line is not tangential *tangential* to P_2, we move clockwise in P_2. From geometry, we get that we can find such a point by only considering the *upper* half of the points in each of P_1 and P_2. Therefore, we won't visit a vertex again.
Similarly, we compute the lower pair of points by moving clockwise in P_1 and anti-clockwise in P_2.
Now, we know the *topmost* and *bottommost* indices in P_1 and P_2. We now take the points that lie between $upper_1$ and $lower_1$, as well as those that lie between $lower_2$ and $upper_2$, in this exact order (to maintain the invariant of returning the points in anti-clockwise order).
This completes the merge step and hence the induction.
Therefore, we have proven the correctness of our algorithm.

□

2 Question 2

Question 2

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

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

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:

$$\begin{aligned}
A(x) &= (0, q_1, q_2, \dots, q_n) \\
B(x) &= \left(-\frac{1}{(n-1)^2}, -\frac{1}{(n-2)^2}, \dots, -\frac{1}{1^2}, 0, \frac{1}{1^2}, \dots, \frac{1}{(n-2)^2}, \frac{1}{(n-1)^2}\right)
\end{aligned} \tag{3}$$

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:

$$\begin{aligned}
A(x) = &\qquad\qquad q_nx^n + \dots + \quad q_{j+1}x^{j+1} + \quad q_jx^j + \quad q_{j-1}x^{j-1} + \dots + q_1x^1 + 0x^0 \\
B(x) = \dots + &-\frac{1}{(n-j)^2}x^{j-1} + \dots + -\frac{1}{1^2}x^{n-2} + \ 0x^{n-1} + \quad \frac{1}{1^2}x^n + \dots + \frac{1}{(j-1)^2}x^{j+n-2} \\
&+ \cdots
\end{aligned} \tag{4}$$

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:

$$P(x)[j+n-1] = \sum_{k=1}^{n-j} q_{j+k} \cdot -\frac{1}{k^2} + 0 + \sum_{k=1}^{j-1} q_{j-k} \cdot \frac{1}{k^2} \tag{5}$$

Where $P(x)[p]$ denotes the coefficient of x^p in $P(x)$.
Equation 5 can be rewritten as:

$$\begin{aligned}
P(x)[j+n-1] &= \sum_{i=j+k,k=1}^{n-j} q_i \cdot -\frac{1}{(j-1)^2} + \sum_{i=j-k,k=1}^{j-1} q_i \cdot \frac{1}{(j-1)^2} \\
&= -\sum_{i=j+1}^{n} \frac{q_i}{(j-1)^2} + \sum_{i=1}^{j-1} \frac{q_i}{(j-1)^2} \\
&= \sum_{i<j} \frac{q_i}{(j-1)^2} - \sum_{i>j} \frac{q_i}{(j-1)^2} \\
&= \frac{F_j}{Cq_j} \\
\implies F_j &= P(x)[j+n-1] \times Cq_j
\end{aligned} \tag{6}$$

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

```
1: procedure COMPUTE FORCES((q, n))
2:     A ← q
3:     B ← [−1/(n−1)^2, −1/(n−2)^2, ..., −1/1^2, 0, 1/1^2, ..., 1/(n−2)^2, 1/(n−1)^2]
4:     P ← multiply(A, B)  ▷ multiply A(x) and B(x) using FFT "divide and conquer" algo
5:     F ← C[n : 2n − 1]   ▷ taking subarray corresponding to coefficients of x^(j+n−1)
6:     for i ∈ [1, n] do    ▷ 1-indexed array
7:         F[i] ← F[i] × Cq[i]
8:     end for
9:     return F
10: end procedure
```

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 6.

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. $\square$

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:

$$\begin{aligned} (A_H)_{ij} &= (A_G)_{ij} > 0 \vee (A_G^2)_{ij} > 0 \\ &= (A_G)_{ij} + (A_G^2)_{ij} > 0 \\ \implies A_H &= A_G + A_G^2 \succ 0 \end{aligned} \tag{7}$$

□

Therefore, the algorithm for computing A_H is:

Algorithm 5 Computing H

procedure ComputeH(G)
  $A_G \leftarrow \mathit{adjacency}(G)$
  $A_H \leftarrow A_G + A_G^2$
  $n \leftarrow |V_G|$
  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 $\mathit{graph}(A_H)$
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:

$$\begin{aligned} P_G(x,y) &= \{x, a_1, a_2, \ldots, a_k, y\} \\ \implies D_G(x,y) &= k+1 \end{aligned} \tag{8}$$

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

$$\begin{aligned} P_H(x,y) &= \{x, a_2, a_4, \ldots, a_{k-1}, y\} \\ \implies \mathit{length}(P_H(x,y)) &= \frac{k-1}{2} + 1 \\ &= \frac{k+1}{2} \\ &= \left\lceil \frac{D_G(x,y)}{2} \right\rceil \end{aligned} \tag{9}$$

When k is even, we have:

$$\begin{aligned} P_H(x,y) &= \{x, a_2, a_4, \ldots, a_k, y\} \text{ ((}a_k, y\text{) is the only edge of type 1)} \\ \implies \mathit{length}(P_H(x,y)) &= \frac{k}{2} + 1 \\ &= \frac{(k+1)+1}{2} \\ &= \left\lceil \frac{D_G(x,y)}{2} \right\rceil \end{aligned} \tag{10}$$

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 $Q_H(x,y)$:

$$\begin{aligned} Q_H(x,y) &= \{x, b_1, b_2, \ldots, b_m, y\} \\ \implies \mathit{length}(Q_H(x,y)) &= m+1 < \left\lceil \frac{D_G(x,y)}{2} \right\rceil, \text{ from assumption} \end{aligned} \tag{11}$$

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

$$\begin{aligned} Q_G(x,y) &= \{x, c_1, b_1, c_2, b_2, \ldots, c_m, b_m, c_{m+1}, y\}, \text{ } c_i \text{ may be the same as } b_i \\ \implies \mathit{length}(Q_G(x,y)) &\le 2m+2 < 2\left\lceil \frac{D_G(x,y)}{2} \right\rceil \\ \implies \mathit{length}(Q_G(x,y)) &< \begin{cases} D_G(x,y)+1, & D_G(x,y) \text{ is odd} \\ D_G(x,y), & D_G(x,y) \text{ is even} \end{cases} \end{aligned} \tag{12}$$

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 $\mathit{length}(Q_G(x,y))$ 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 3(c)

Question. *Let A_G be adjacency matrix of G, and $M = D_H \times A_G$. Prove that for any $x,y \in V$, the following holds:*

$$D_G(x,y) = \begin{cases} 2D_H(x,y) & M(x,y) \ge \deg_G(y) \cdot D_H(x,y) \\ 2D_H(x,y) - 1 & M(x,y) < \deg_G(y) \cdot D_H(x,y) \end{cases} \tag{13}$$

Solution. From Question 3.2, we know that $D_G(x,y)$ is equal to either of $2D_H(x,y)$ or $2D_H(x,y) - 1$. Therefore, we have two cases for when $D_G(x,y)$ is odd or even. We first consider the case when $D_G(x,y)$ is odd:

Claim 3.4. *When $D_G(x,y)$ is odd, we have $M(x,y) < \deg_G(y) \cdot D_H(x,y)$*

Proof. Consider the closest neighbour z_0 of y. Since $D_G(x,y)$ is odd, we have that $D_H(x,z_0) + 1 = D_H(x,y)$. It is easy to see this from the path of the case when k is even in Question 3.2 (the edge (z_0,y) is the additional edge on the path from x to y). We now show that for any other neighbour z of y, $D_H(x,z)$ cannot be larger than $D_H(x,y)$. This is true since There exists an edge (z_0, z) in H. Thus we have the following inequality:

$$\begin{aligned}
\sum_{z \in neighbour_G(y)} D_H(x,z) &= D_H(x,z_0) + D_H(x,z) \\
&< D_H(x,y) + (\deg(y)-1)D_H(x,y) \\
&< \deg(y) \cdot D_H(x,y) \\
\implies M(x,y) &< \deg_G(y) \cdot D_H(x,y)
\end{aligned} \tag{14}$$

This completes the proof of the claim. □

We will now prove the following claim for the even case:

Claim 3.5. *When $D_G(x,y)$ is even, we have $M(x,y) \geq \deg_G(y) \cdot D_H(x,y)$*

Proof. We now consider the farthest neighbour z_0 of y. This is at a distance of atmost $D_H(x,y)+1$. This is because there exists a path from x to y along with the edge y, z_0. Also, for any neighbour z, $D_H(x,z)$ cannot be smaller than $D_H(x,y)$. This is easy to prove via contradiction. Since, $D_G(x,y)$ is even, all edges in the path in H are of type 2. Therefore, if there was a path of shorter length, we would have a possible path containing an edge of type 1, which leads to a contradiction. Therefore, we have the following:

$$\begin{aligned}
\sum_{z \in neighbour_G(y)} D_H(x,z) &\geq \deg_G(y) \cdot D_H(x,y) \\
\implies M(x,y) &\geq \deg_G(y) \cdot D_H(x,y)
\end{aligned} \tag{15}$$

This completes the proof for theven case too. □

Therefore, for both cases, we have shown that the relations satisfied by x, y are different. Thus, we can use this condition to determine the value of D_G in terms of D_H. We restate the result:

$$D_G(x,y) = \begin{cases} 2D_H(x,y) & M(x,y) \geq \deg_G(y) \cdot D_H(x,y) \\ 2D_H(x,y) - 1 & M(x,y) < \deg_G(y) \cdot D_H(x,y) \end{cases} \tag{16}$$

□

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:     M ← D_H × adjacency(G)
3:     D_G ← init()
4:     for x ∈ V do
5:         for y ∈ V do
6:             if M(x, y) ≥ deg(G, y) · D_H(x, y) then
7:                 D_G(x, y) ← 2D_H(x, y)
8:             else
9:                 D_G(x, y) ← 2D_H(x, y) − 1
10:            end if
11:        end for
12:    end for
13:    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: $A_G \leftarrow adjacency(G)$
3: $H \leftarrow$ ComputeH(G)
4: **if** $H = G$ **then**
5: $D_G \leftarrow A_G$
6: $D_G \leftarrow$ all off-diagonal zero entries are set to ∞
7: **return** D_G
8: **end if**
9: $D_H \leftarrow$ AllPairDistances(H)
10: $D_G \leftarrow$ 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. *Prove that for any given $r \in [1, p-1]$, there exists at least ${}^{M/n}C_n$ subsets of U of size n in which maximum chain length in hash-table corresponding to $H_r(x)$ is $\Theta(n)$.*

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 mod 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:

$$y \bmod n = \{0, 1, \ldots, n-1, 0, 1, \ldots, n-1, (M/n \textit{ times}) \ldots, 0, 1, \ldots, n-1\} \tag{17}$$

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.

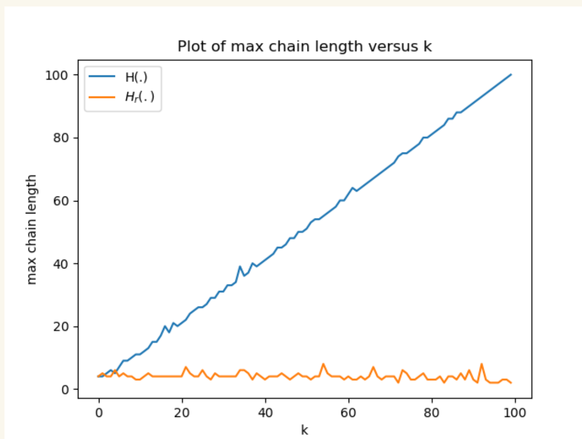

Figure 1: Plot

```
import random
from collections import Counter
import matplotlib.pyplot as plt


M = 10000
p = 10007
n = 100

def gen_sk(k: int):
    sk = [i * n for i in range(k)]
    for _ in range(k, n):
        sk.append(random.randint(0, M - 1))
    return sk
```

```python
def H(Sk: list):
    return [s % n for s in Sk]


def Hr(Sk: list):
    r = random.randint(1, p - 1)
    return [(r * s % p) % n for s in Sk]


def get_max_length(hashes: list):
    counts = Counter(hashes)
    return counts.most_common(1)[0][1]


if __name__ == "__main__":
    H_vals, Hr_vals = [], []
    for k in range(1, n + 1):
        Sk = gen_sk(k)
        H_vals.append(get_max_length(H(Sk)))
        Hr_vals.append(get_max_length(Hr(Sk)))

    ax = plt.axes()
    ax.set_xlabel("k")
    ax.set_ylabel("max chain length")
    ax.set_title("Plot of max chain length versus k")
    ax.plot(H_vals, label="H(.)")
    ax.plot(Hr_vals, label="$H_r(.)$")
    ax.legend()
    plt.savefig("4c.png")
```

□