COL351 Assignment 3

Shrey Patel
2019CS10400

Rahul Chhabra
2019CS11016

October 27, 2021

1. Convex Hull

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 $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$.

Solution:

Goal:
Our main goal is to design an algorithm which taken in input a set of points and returns the set of vertices of convex hull in some order (say counter-clockwise).
We will use Divide and Conquer strategy to design our algorithm.

Algorithm for returning the set of vertices of convex hull:

1. Sort the points according to their x-coordinate and resolve conflicts on basis of y-coordinate as a part of pre-processing step.
2. Divide points into left half and right half using the median of x coordinates and recursively compute the set of vertices of corresponding convex hulls.
3. It can be seen that the two convex hulls horizontally separable as we have divided the points according to their x-coordinates.
4. Find the upper and lower tangents and add their endpoints to the union of both convex hull.
5. Remove vertices which lie between the two tangents.

Algorithm for finding lower tangent:
We are given vertices of left and right convex hulls and we need to find the lower tangent.

1. Take the vertex a with largest x coordinate among vertices of left convex hull and vertex b with smallest x coordinate from right convex hull. In case there are more than candidates, take the one with smallest y-coordinate.
2. While the predecessor of a in left set or the successor of b in right set is below the ab line, if the predecessor of a in left set is below ab line then update a to predecessor(a) otherwise update b to successor(b)
3. Final vertices a and b are the endpoints of required tangent.

In similar way the upper tangent can also be calculated.

We can check whether a point lies above or below a line by putting it in the line equation and checking the sign of the equation.This is taken from basic school maths.

Note:
predecessor(a) is the next point in hull in clockwise direction or equivalently the previous point in hull in counter-clockwise direction
successor(a) is the next point in hull in counterclockwise direction or equivalently the previous point in hull in clockwise direction
Without loss of generality, we have made two assumptions, one is that no three points are collinear and other is no two points have same x coordinate. Our algorithm can handle these cases with minor addition of some edge cases

Lemma 1.1: The algorithm for finding the lower tangent is correct.
Proof:
To prove this, we will first show that the tangent returned doesn't pass through interior of any of the hulls.

Lemma 1.1.1 In any iteration, the line joining the left point and right point doesn't pass through the interior of left or right convex hull.

Proof:
We will use induction to prove this lemma.

Base Case: Since the points in both convex hulls are separated initially, so the line joining the rightmost point of left hull and leftmost point of right hull cannot pass through any of the hulls. Base case holds.

Induction Hypothesis: Let at any iteration k, the line joining left point and right point doesn't pass through any of the hulls.

Induction Step: Without loss of generality, let's assume that at $(k+1)^{th}$ step, left point is updated to predecessor(left point) which implies that a' = predecessor(left point) lies below the line L joining left point and right point. Suppose the line joining the left point and right point passes through the interior of left hull. Now since L' = line joining a' and right point passes through the interior of left hull which implies left point lies below L'. This contradicts the fact that a' lies below L. Since a' and left point are two distinct points so So at any iteration , the line joining the left point and right point doesn't pass through the interior of any hull.

Algorithm 1 Algorithm for returning the vertices of convex hull in counter-clockwise order

```
procedure FindConvexHull(s[]):
    sort set s in non decreasing order according to their x coordinate(break ties on basis of y coordinate)
    n ← size of set s
    return ConvexHullUtil(s, 1, n)
procedure ConvexHullUtil(s[], i, j):
    medianIndex ← (i+j)/2
    leftHull ← ConvexHullUtil(s, i, medianIndex)
    rightHull ← ConvexHullUtil(s, medianIndex + 1, j)
    return merge(leftHull, rightHull)
procedure merge(leftHull, rightHull):
    lowerTangent ← findLowerTangent(leftHull, rightHull)
    upperTangent ← findUpperTangent(leftHull, rightHull)
    topLeft ← upperTangent[0]
    bottomLeft ← lowerTangent[0]
    topRight ← upperTangent[1]
    bottomRight ← lowerTangent[1]
    p ← number of vertices in left hull
    q ← number of vertices in right hull
    while topLeft ≠ bottomLeft do
        topLeft = (topLeft - 1) mod p
        if topLeft == bottomLeft then break
        end if
        remove topLeft from leftHull
    end while
    while topRight ≠ bottomRight do
        topRight = (topRight + 1) mod q
        if topRight == bottomRight then break
        end if
        remove topRight from RightHull
    end while
    result ← []
    topLeft ← upperTangent[0]
    bottomLeft ← lowerTangent[0]
    topRight ← upperTangent[1]
    bottomRight ← lowerTangent[0]
    while topLeft ≠ bottomLeft do
        add lefHull[topLeft] into the result
        topLeft = (topLeft + 1) mod p
    end while
    Add bottomLeft to result
    while topRight ≠ bottomRight do
        add rightHull[topRight] into result
        bottomRight = (bottomRight + 1) mod q
    end while
    Add topRight to result
    return result
```

```
procedure findLowerTangent(leftHull, rightHull):
    left point ← point in left hull with largest x coordinate
    right point ← point in right hull with smallest x coordinate
    condition1 ← predecessor of left point in leftHull is below lines formed by joining left point and right point
    condition2 ← successor of right point in rightHull is below lines formed by joining left point and right point
    while condition1 or condition2 do
        if condition1 then
            left point = predecessor(left point)
        else
            right point = successor(right point)
        end if
        condition1 ← predecessor of left point in leftHull is below lines formed by joining left point and right point
        condition2 ← successor of right point in rightHull is below lines formed by joining left point and right point
    end while
    return left point, right point
procedure findUpperTangent(leftHull, rightHull):
    left point ← point in left hull with largest x coordinate
    right point ← point in right hull with smallest x coordinate
    condition1 ← successor of left point in leftHull is below lines formed by joining left point and right point
    condition2 ← predecessor of right point in rightHull is below lines formed by joining left point and right point
    while condition1 or condition2 do
        if condition1 then
            left point = successor(left point)
        else
            right point = successor(right point)
        end if
        condition1 ← successor of left point in leftHull is below lines formed by joining left point and right point
        condition2 ← predecessor of right point in rightHull is below lines formed by joining left point and right point
    end while
    return left point, right point
```

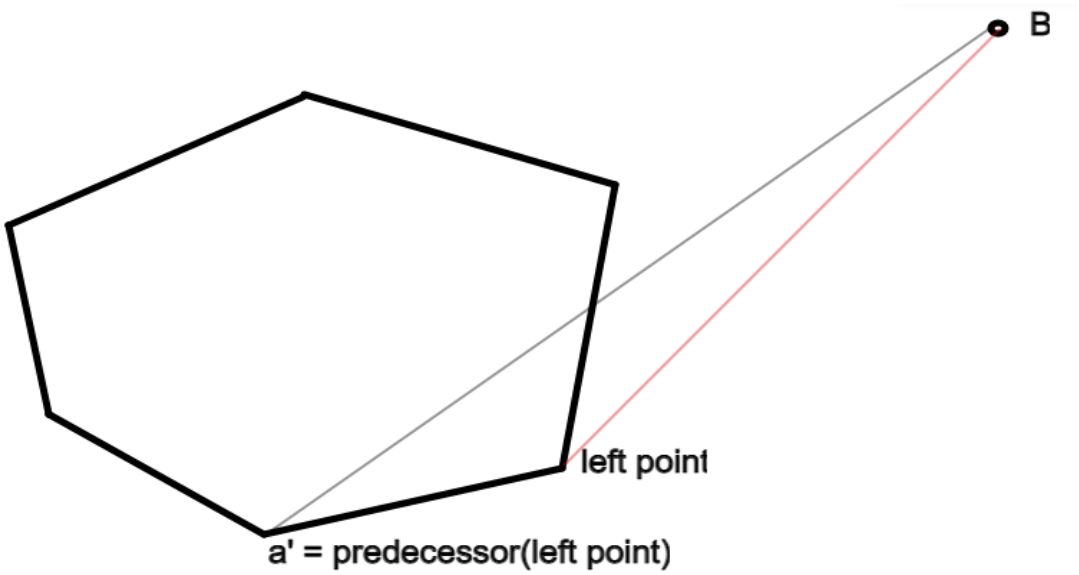

Figure 1: Pictorial Representation of a left point and its predecessor

Lemma 1.1.2: The while loop in the algorithm eventually terminates.

We can show that the left point will never cross the left most point of left hull and right point will never cross the rightmost point of right hull. Suppose at some iteration k, the left point coincides with the leftmost point of left hull. Now the predecessor of this point will definitely lie above the line joining left and right point (by arguments from lemma 1.1.1) so this our algorithm will not move forward and similarly for rightmost point of right hull, our algorithm will not move forward so will eventually terminate. These arguments hold because the two hulls are horizontally seperable.

Lemma 1.2: The algorithm for finding the lower tangent is correct.
Proof:
Same as above with suitable changes **Lemma 1.3:** The algorithm returns the required Convex hull

Proof by Strong induction on number of vertices n

Base Case:(n = 3) Since a convex hull needs to have atleast 3 vertices. If the points are not collinear then we can say that the given points form a triangle so the convex hull of minimum area will have all three vertices. Base case is true.

Induction Hypothesis: Let the above algorithm be correct for all set of points S such that $|S| < n$ i.e. the above algorithm returns the set of vertices of convex hull formed by S in counter clockwise order.

Induction Step: According to the algorithm we first divide the points into left half and right half of nearly equal size and recursively find the the vertices of convex hull corresponding to them. Since the cardinality of both left and right set is less than n so according to Induction Hypothesis the set returned by algorithm for left and right part contains the vertices of corresponding convex hulls in counter clockwise order. We

also have coordinates of end points of upper and lower tangent. We remove the points lying between the the upper and lower tangent for left hull and right hull and merge them in the required order.

Lemma 1.3.1: The polygon obtained after merging both hulls is a convex hull.
Proof: Suppose there exists two points (a,b) and (c,d) such that line joining these two points passes through the region outside the polygon. Since by Induction hypothesis, let and right parts are convex hulls, these two points cannot lie on the same side which means one point lies on the left hull(say (a,b)) and other on the right hull(say (c,d)). This Without loss of generality, let's assume that this line passes through the region above the polygon. Let T be the upper tangent of left and right convex hulls,So one of the endpoints of upper tangent must lie below this line but this implies that one of the points (a,b) and (c,d) lies above the upper tangent which contradicts the fact that T is the upper tangent for left and right convex hulls.Since the polygon so obtained is a convex hull , By principle of mathematical induction, Our algorithm returns a convex hull for given points.

Since the polygon formed is a convex hull and its vertices are subset of given set of points, we can say that this is indeed the convex hull with smallest area.

2. Particle Interaction

Some physicists are working on interactions among large numbers of very small charged particles. Basically, their set-up works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points 1, 2, 3, ..., n on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to:

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

However, the running time of the naive algorithm is O(n^2). Your task is to design an algorithm that computes all the forces F_j in O(n *log* n) time.

Solution:
We will solve this problem using multiplication of polynomials by Fast Fourier Transform, and we assume the correctness of the FFT algorithm and its time complexity. We will first split the given expression for F_j in to two parts and solve each of them independently as follows:

$$F_j = Cq_j(T_1 \text{ - } T_2)$$
$$\text{where } T_1 = \sum_{i<j} \left(\frac{q_i}{(j-i)^2}\right) \text{ and } T_2 = \sum_{i>j} \left(\frac{q_i}{(j-i)^2}\right)$$

Note that since C is constant and the charge q_j is a free variable for the summation, they can be taken out of the summation.

Also, we define three polynomials for using FFT as follows:

1. $A(x) = \sum_{i=0}^{n} a_i\ x^i$ where $a_i = q_i\ \forall\ 0 < \text{i} \leq \text{n}$, and $a_0 = 0$.
2. $B(x) = \sum_{i=0}^{n} b_i\ x^i$ where $b_i = q_{n-i}\ \forall\ 0 \leq \text{i} < \text{n}$, and $b_n = 0$.
3. $C(x) = \sum_{i=0}^{n} c_i\ x^i$ where $c_i = \left(\frac{1}{i^2}\right)\ \forall\ 0 < \text{i} \leq \text{n}$, and $c_0 = 0$.

Now, we express the terms T_1 and T_2 in terms of the co-efficient of the above defined polynomials.

$T_1 = \sum_{i<j} \left(\frac{q_i}{(j-i)^2}\right) \implies$
$T_1 = \left(\frac{q_1}{(j-1)^2}\right) + \left(\frac{q_2}{(j-2)^2}\right) + ... + \left(\frac{q_{j-1}}{(1)^2}\right) \implies$
$T_1 = a_1c_{j-1} + a_2c_{j-2} + ... + a_{j-1}c_1$

Now, since $a_0 = c_0 = 0$, we can add two extra terms a_0c_j and a_jc_0 to RHS without any change in the value.

$T_1 = a_0c_j + a_1c_{j-1} + a_2c_{j-2} + ... + a_{j-1}c_1 + a_jc_0 \implies$
$T_1 = \sum_{i=0}^{j} a_ic_{j-i} \implies$
$T_1 =$ coefficient of x^j in the product $A(x) * C(x)$

Similarly, for T2,

$T_2 = \sum_{i>j} \left(\frac{q_i}{(j-i)^2}\right) \implies$
$T_2 = \left(\frac{q_{j+1}}{(1)^2}\right) + \left(\frac{q_{j+2}}{(2)^2}\right) + ... + \left(\frac{q_n}{(n-j)^2}\right) \implies$
$T_2 = b_{n-j-1}c_1 + b_{n-j-2}c_2 + ... + b_0c_{n-j}$

Now, since $c_0 = 0$, we can add an extra term $b_{n-j}c_0$ to RHS without any change in the value.

$T_2 = b_{n-j}c_0 + b_{n-j-1}c_1 + b_{n-j-2}c_2 + ... + b_0c_{n-j} \implies$
$T_2 = \sum_{i=0}^{n-j} b_{n-j-i}c_i \implies$
$T_2 =$ coefficient of x^{n-j} in the product $B(x) * C(x)$

Thus, $F_j = Cq_j$(co-eff of x^j in $A(x)*C(x)$ - co-eff of x^{n-j} in $B(x)*C(x)$). So, all that remains is to compute the polynomial products $A(x) * C(x)$ and $B(x) * C(x)$.

Algorithm:

1. Compute $D(x) = A(x) * C(x) = \sum_{i=0}^{2*n} d_i x^i$.
2. Compute $E(x) = B(x) * C(x) = \sum_{i=0}^{2*n} e_i x^i$.
3. For all indices $j \in \{1, 2, ..., n\}$: $F_j = Cq_j(d_j - e_{n-j})$.

Proof of correctness:
The proof of steps 1 and 2 follow from the correctness of FFT algorithm, which we assume to be true. And the proof of the step 3(rest of the problem formulation) is as discussed before the algorithm.

Time Complexity:
From FFT, time taken to multiply two polynomials of degree n = O(n *log* n).

- Time complexity of step 1: O(n *log* n).
- Time complexity of step 2: O(n *log* n).
- Time complexity of step 3: n * O(1) = O(n).

So, **Total time complexity of the algorithm = O(n *log* n)**.

3. Distance computation using Matrix Multiplication

Let $G = (V, E)$ be an unweighted undirected graph, and $H = (V, E_H)$ be an undirected graph obtained from G which satisfies: $(x, y) \in E_H$ if and only if $(x, y) \in E$ or there exists a $w \in V$ such that $(x, w), (w, y) \in E$. Further, let D_G denote the distance-matrix of G, and D_H be the distance-matrix of H.

(a) 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.

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

(c) Let A_G be the adjacency matrix of G, and $M = D_H * A_G$. Prove that for any $x,y \in V$, then,

$$D_G(x, y) = \begin{cases} 2\mathrm{D}_H(x, y) & M(x,y) \geq degree_G(y).D_H(x, y) \\ 2\mathrm{D}_H(x, y) \text{ - } 1 & M(x,y) < degree_G(y).D_H(x, y) \end{cases}$$

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

(e) Prove that all-pairs-distances in n-vertex unweighted undirected graph can be computed in $O(n^\omega \ log \ n)$ time, if $\omega > 2$.

From lecture notes, product of any two square matrices of order n takes $O(n^\omega)$ and time, we use this fact as it is without proof.

Property 3.1: (i,j) $\in E_H \iff$ either (i,j) $\in E_G$ or $\exists$ a vertex k such that (i,k), (k,j) $\in E_G$.

Solution for (a):

Let A_G denote the adjacency matrix of the graph G of order n, i.e. $A_{ij} = 1$ if the edge $(i,j) \in E_G$ otherwise 0. Thus, an adjacency matrix and a graph have a one-to-one correspondence, meaning that a graph can be derived entirely from its adjacency matrix. So, we need to derive the adjacency matrix of H A_H from A_G.

Note that we assume that the graphs have no self-loops and so, $(A_G)_{ii} = (A_H)_{ii} = 0 \ \forall \ 1 \leq i \leq n$.

Algorithm:

1. Compute the matrix $(A_G)^2$ using the matrix product algorithm.
2. Obtain the matrix A' = $A_G + (A_G)^2$.
3. Obtain the matrix A_H as: For all $1 \leq$ i,j $\leq$ n: $(A_H)_{ij} = 1$ if $A'_{ij} > 0$ and 0 otherwise.

Property 3.2: *All entries of the matrices defined above are non-negative since adjacency matrices have either 0 or 1 as their entries and any matrix product involves only summation of values.*

Claim 3.1: A_H is the adjacency matrix of the graph H.
Proof of Claim 3.1:
To prove this, we need to prove that for all vertices i,j, (i,j) $\in E_H$ if and only if $(A_H)_{ij} = 1$. Note that from the step 3, the statement $(A_H)_{ij} = 1$ is equivalent to $A'_{ij} > 0$. So, we need to prove that (i,j) $\in E_H$ if and only if $A'_{ij} > 0$.

($\Rightarrow$) Suppose that for a pair of vertices i and j, (i,j) $\in E_H$. Then we need to prove that $A'_{ij} > 0$. From property 3.1, there can be two cases: (*Note that these two cases are not mutually exclusive but the proof still holds since they are exhaustive.*

- **Case 1:** (i,j) $\in E_G$. Then from the definition of adjacency matrix, $(A_G)_{ij} > 0$. Now, A' = A_G + $(A_G)^2$ where $(A_G)_{ij} > 0$ and $(A_G)^2_{ij} \geq 0$ from property 3.2. Thus, $A'_{ij} > 0$. Hence Proved.
- **Case 2:** $\exists$ a vertex k such that (i,k), (k,j) $\in E_G$. So, for the same vertex k, $(A_G)_{ik} > 0$ and $(A_G)_{kj} > 0$ from the definition of adjacency matrix. So, $(A_G)_{ik} * (A_G)_{kj} > 0$ and thus, $\sum_{l=1}^{n} (A_G)_{il} * (A_G)_{lj} = (A_G)^2_{ij} > 0$. Also from property 3.2, $(A_G)_{ij} \geq 0$. So, A' = A_G + $(A_G)^2 > 0$. Hence Proved.

So, the forward inference holds for both the cases.

($\Leftarrow$) Suppose for some pair of vertices *i,j*, $A'_{i,j}$. Then we need to prove that the edge (i,j) $\in E_H$. From our assumption and the definition of A', A_G + $(A_G)^2 > 0$. From property 3.2, $(A_G)_{ij} \geq 0$ and $(A_G)^2_{ij} \geq 0$ $\forall$ 1 $\leq$ *i,j* $\leq$ n. So, combining our assumption and property 3.2, we have two exhaustive cases (possibly non-exclusive):

- **Case 1:** $(A_G)_{ij} > 0$. Then (i,j) $\in A_G$, and so using property 3.1, we have that (i,j) $\in A_H$. Hence Proved.
- **Case 2:** $(A_G)^2_{ij} > 0$. From matrix product, $(A_G)^2_{ij} = \sum_{l=1}^{n} (A_G)_{il} * (A_G)_{lj} > 0$ and since all the entries of the adjacency matrix are non-negative, we have that $\exists$ at least one such 1 $\leq$ k $\leq$ n for which $(A_G)_{ik} * (A_G)_{kj} > 0$ which means that $(A_G)_{ik} > 0$ and $(A_G)_{kj} > 0$. So, $\exists$ a vertex v such that (i,k), (k,j) $\in E_G$. From property 3.1, (i,j) $\in E_H$. Hence Proved.

So, the converse also holds for both the cases. And therefore the claim holds.

Time Complexity:
We assume and directly use that the product of two matrices of order n take O(n^ω) time.

- Computing $(A_G)^2_{ij} = (A_G)_{ij} * (A_G)_{ij}$: O($n^\omega$).
- Computing A' = A_G + $(A_G)^2$, which is obtained by n^2 addition operations is O(n^2).
- Obtaining A_H from A', which involves comparing n^2 entries, is O(n^2).

Since, it is given that $\omega > 2$, the total time complexity is O(n^ω).

Solution for (b):

We need to prove that $D_H(x,y) = \lceil \left(\frac{D_G(x,y)}{2} \right) \rceil$ for any two vertices *x,y*. Let P = $\{z_0 = x, z_1, ..., z_L = y\}$ be the shortest path joining some pair of vertices x,y in graph G. Now, if P' is the shortest path joining x,y then, we need to prove that the length of P' is $\lceil \left(\frac{L}{2} \right) \rceil$.

First, we prove that $\exists$ a path in H joining x,y of length $\lceil \left(\frac{L}{2} \right) \rceil$. Consider the path P as defined above. Since, (z_0,z_1), (z_1,z_2) $\in E_G$, from property 3.1, (z_0,z_2) $\in E_H$. Similarly, (z_i,z_{i+2}) $\in E_H$ $\forall$ 0 $\leq$ i $<$ L-1. Therefore consider the path Q in H formed by the all the edges (z_0,z_2), (z_2,z_4), ... such that it joins x and y i.e. Q = $\{z_0 = x, z_2, ..., z_2k\} \cup \{z_L = y\}$. There are two possible cases depending on whether L is odd or even:

- **Case 1:** L is even. Then since the first set on RHS of Q contains only the even indexed elements, z_L also belongs to that set. So, Q = $\{z_0 = x, z_2, ..., z_L\}$ and so the length of the path is L/2 which is equal to $\lceil \left(\frac{L}{2} \right) \rceil$ when L is even.

- **Case 2:** L is odd. Then L-1 is even and so, Q = $\{z_0 = x, z_2, ..., z_{L-1}\} \cup \{z_L = y\}$. Length of part 1 is (L-1)/2. So, total length of Q = (L+1)/2 = $\lceil(\frac{L}{2})\rceil$ when L is odd.

Thus, we proved that we can obtain a path Q in H joining x and y which has a length $\lceil(\frac{L}{2})\rceil$.

Now, we require to prove that there doesn't exist any path Q' joining x,y in H such that len(Q') < $\lceil(\frac{L}{2})\rceil$, where L is the length of the shortest path joining x and y in G. We will prove this by contradiction. Assume that ∃ a path Q' of length say K, joining x and y in H such that K < $\lceil(\frac{L}{2})\rceil$. Let Q' = $\{x = w_0, w_1, ...w_K = y\}$.

Now, since (w_i,w_{i+1}) ∈ E_H, from property 3.1, ∀ 0 ≤ i < K, either (w_i,w_{i+1}) ∈ E_G or ∃ some vertex w such that (w_i,w), (w,w_{i+1}) ∈ E_G. So, if we extend the path Q' to some path P' to join x,y in G, then for every edge in Q' we add at most one edge in P', and thus len(P') ≤ 2*K.

From assumption, we have that K < $\lceil(\frac{L}{2})\rceil$. Now there are two cases:

- **Case 1:** L is even. Then K < L/2 or 2*K < L. From above deduction, ∃ a path P' of length at most 2*K which is less than K which joins x and y in G. This contradicts our initial assumption that L is the length of the shortest path joining x and y in G.
- **Case 2:** L is odd. Then K < (L+1)/2 or 2*K < L+1 or 2*K ≤ L. But since L is odd and 2*K is even, 2*K ≠ L. Just like case 1, this contradicts our initial assumption.

So, by contradiction, there cannot exist any such path Q' in H whose length is less than $\lceil(\frac{L}{2})\rceil$. So, for any pair of vertices x,y we have that $D_H(x,y) = \lceil\left(\frac{D_G(x,y)}{2}\right)\rceil$. Hence Proved.

Solution for (c):

It is given that:
M = $D_H * A_G$ $\implies$
M(x,y) = $\sum_{z=1}^{n} D_H(x,z) * A_G(z,y)$ $\implies$
M(x,y) = $\sum_{z \in N_G(y)} D_H(x,z)$
where $N_G(y)$ = set of neighbours of y in G, so $|N_G(y)| = degree_G(y)$

From part (b), we proved that $D_H(x,y) = \left(\frac{D_G(x,y)}{2}\right)\rceil$. So, there can be two possible cases depending on whether $D_G(x,y)$ is even or odd. So, we will prove this part using case analysis:

- **Case 1:** When $D_G(x,y)$ is even. So, $D_G(x,y) = 2 * D_H(x,y)$. Let Q = $\{a_0 = x, a_1, a_2, ...a_L = y\}$ be the shortest path of length L=$D_H(x,y)$, which joins x and y in H. Then because (a_i,a_{i+1}) ∈ E_H, there must exist some vertex b_i such that (a_i,b_i), (b_i,a_{i+1}) ∈ E_G ∀ 0 ≤ i < L so that the path Q can be extended to another path P = $\{a_0 = x, b_0, a_1, b_1, a_2, b_2, ...b_{L-1}, a_L = y\}$ of length 2*L = $2 * D_H(x,y) = D_G(x,y)$ which joins x and y in G. No other part joining x,y in G is shorter than P because its length is $D_G(x,y)$. Thus, path P is optimal.

  Now, let $N_G(y) = \{w_0, w_1, w_2....\}$ be the neighbours of y in G. There are two kinds of neighbours:

  - **Subcase 1:** A neighbour w lies on an optimal path P in G. In this case, w is similar to the vertex b_{L-1} defined above. But w ∈ Q i.e. the optimal path in H. In Q, we have that $D_H(x,y) = D_H(x,a_{L-1}) + 1$. Similarly, since the edge (a_{L-1},$b_{L-1} = w$) ∈ E_H, we have that $D_H(x,w) = D_H(x,a_{L-1} + 1$. From the above two equations, $D_H(x,y) = D_H(x,w)$.
  - **Subcase 2:** A neighbour w doesn't lie on the optimal path P in G. Then we prove that $D_H(x,y) \leq D_H(x,w)$. We prove this by contradiction. Lets assume that ∃ such neighbour w of y such that $D_H(x,y) > D_H(x,w)$. Then, since the edge (w,y) ∈ E_H, $D_H(x,w) + 1 \leq D_H(x,y)$. We found a

better path from x to y through the neighbour w in H with possibly shorter length than current value of $D_H(x,y)$. We have a contradiction since we assumed that w doesn't lie on the optimal path in H. So, $D_H(x,y) \leq D_H(x,w)$.

So, for some neighbours w, we have $D_H(x,y) = D_H(x,w)$ while for the rest $D_H(x,y) \leq D_H(x,w)$.

$\forall$ w $\in N_G(y)$, $D_H(x,y) \leq D_H(x,w) \implies$
$\sum_{w \in N_G(y)} D_H(x,w) \geq |N_G(y)|$ * $D_H(x,y) \implies$
M(x,y) $\geq degree_G(y) * D_H(x,y)$

So, when $D_G(x,y)$ is even, **M(x,y)** $\geq degree_G(y) * D_H(x,y)$.
Hence Proved.

- **Case 2:** When $D_G(x,y)$ is odd. So, $D_G(x,y) = 2 * D_H(x,y)$ - 1. Let Q = $\{a_0 = x, a_1, a_2, ...a_L = y\}$ be the shortest path of length L=$D_H(x,y)$, which joins x and y in H. Then because (a_i,a_{i+1}) $\in E_H$, there must exist some vertex b_i such that (a_i,b_i), (b_i,a_{i+1}) $\in E_G$ $\forall$ 0 $\leq$ i $<$ L-1 so that the path Q can be extended to another path P = $\{a_0 = x, b_0, a_1, b_1, a_2, b_2, ...a_{L-1}, a_L = y\}$ of length 2*(L-1) + 1 = 2*L - 1 = $2 * D_H(x,y) = D_G(x,y)$ which joins x and y in G. No other part joining x,y in G is shorter than P because its length is $D_G(x,y)$. Thus, path P is optimal.

  Now, let $N_G(y) = \{w_0, w_1, w_2....\}$ be the neighbours of y in G. There are two kinds of neighbours:

  - **Subcase 1:** A neighbour w lies on an optimal path P in G. In this case, w is similar to the vertex a_{L-1} defined above. So, w $\in$ Q i.e. the optimal path in H. In Q, we have that $D_H(x,y) = D_H(x,a_{L-1}) + 1 = D_H(x,w) + 1$. So, $D_H(x,w) < D_H(x,y)$
  - **Subcase 2:** A neighbour w doesn't lie on the optimal path P in G. Now, we know that at least one neighbour of y must lie on the optimal path since y is reachable from x in H. Let that neighbour be w'. Then from subcase 1 above, we have that $D_H(x,w') + 1 = D_H(x,y)$. Now, we know that (w',y), (y,w) $\in E_G$ and so, (w',w) $\in E_H$. So, since w is not on optimal path to y, w will also be at the same distance from x as y in H, or $D_H(x,w') + 1 = D_H(x,w)$. Thus, we have that $D_H(x,w) = D_H(x,y)$.

  So, for some neighbours w, we have $D_H(x,w) < D_H(x,y)$ while for the rest $D_H(x,w) = D_H(x,y)$. But since $\exists$ at least one neighbour w such that w lies on optimal path,

  $\sum_{w \in N_G(y)} D_H(x,w) < |N_G(y)|$ * $D_H(x,y)$
  M(x,y) $< degree_G(y) * D_H(x,y)$

  So, when $D_G(x,y)$ is odd, **M(x,y)** $< degree_G(y) * D_H(x,y)$. Hence Proved.

So, we proved the given formula using the two distinct cases derived in part (b).

Solution for (d):

Algorithm:

- First obtain the matrix M as the product $D_H * A_G$.
- Then for each pair x,y, calculate the term $degree_G(y).D_H(x,y)$. If it is greater than M(x,y) then allot $D_G(x,y)$ = 2*$D_H(x,y)$ - 1, else allot $D_G(x,y)$ = 2*$D_H(x,y)$.

The correctness of the above algorithm follows from part (c)

Time Complexity

- Matrix product for calculating M: O(n^ω)
- Calculating $D_G(x, y)$ for a given pair x,y takes O(1) time. So, total time for all such pairs = O(n^2).

So, **Total time complexity = O(n^ω)**, since it is given that $\omega > 2$.

Solution for (e):

Given a graph G, we need to calculate the matrix D_G. We use the following algorithm,

Algorithm:

1. Define a sequence of graphs $H_0 = G$, H_1,..... H_k, such that each graph in the sequence is the subgraph of its successor graph and each successor graph can be obtained by applying the rule given in the problem statement, i.e. $\forall\ 0 \leq$ i $<$ k, *(x, y)* $\in E_{H_{i+1}}$ if and only if *(x, y)* $\in E_{H_i}$ or there exists a $w \in V$ such that *(x, w), (w, y)* $\in E_{H_i}$.
2. Starting from A_G, we calculate adjacency matrices for each successive graph is the above sequence using the method constructed in part(a), i.e.

$$\text{A'} = A_{H_i} + (A_{H_i})^2$$
$$\forall \text{ x,y, } A_{H_{i+1}}(x,y) = \begin{cases} 1 & A'(x,y) > 0 \\ 0 & otherwise \end{cases}$$

   until $A_{H_{i+1}} = A_{H_i}$ i.e. $A_{H_{k+1}} = A_{H_k}$
3. For H_k, $D_{H_k} = A_{H_k}$, since H_k will have completely connected components.
4. So, starting from H_k, compute successive distance matrices using the algorithm described in part (d) as:
   - Calculate the matrix M = $D_{H_{i+1}} * A_{H_i}$.
   - $D_{H_i}(x,y) = \begin{cases} 2\text{D}_{H_{i+1}}(x,y) & M(x,y) \geq degree_{H_i}(y).D_{H_{i+1}}(x,y) \\ 2\text{D}_{H_{i+1}}(x,y) \text{ - } 1 & M(x,y) < degree_{H_i}(y).D_{H_{i+1}}(x,y) \end{cases}$

Correctness and Time Complexity:
The proof of step 2 follows from the correctness argument in part (a). Also, we know from part(a) that the time required to calculate the adjacency matrix of one graph is O(n^ω). And since we need k such iterations, the total time complexity of step 2 is **k * O(n^ω)**.

Similarly, the proof of step 4 follows from the correctness argument in part (c). While the time complexity per iteration can be derived from part(d) as O(n^ω). Again, since there are k iterations, the total time complexity of step 4 id **k * O(n^ω)**.

We terminate from step 2 when the consecutive adjacency matrices are identical. This occurs only in graphs G' which obey the following property: If x,y and z are three distinct vertices in G', then
(x,y), (y,z) $\in E_{G'} \implies$ (x,z) $\in E_{G'}$, or in other words, edge relation is transitive. This is possible only when the graph G' is fully connected. Thus, the graph H_k is fully connected. So, if $\exists$ a path P between two vertices x,y in H_k, then (x,y) $\in E_{H_k}$, or in other words, $D_{H_k} = A_{H_k}$. Thus, we proved step 3.

Thus, the total time complexity as of now is **k * O(n^ω)** and we try to find the value of k. Note that from step 4, while going from $D_{H_{i+1}}$ to D_{H_i}, the distance values are at most doubled. So, for some pair of vertices x,y, after j iterations of the above algorithm, we have that $D_{H_{k-j}}(x,y) \leq 2^j * D_{H_k}(x,y)$. To calculate $D_G(x,y)$, j=k and so, $D_G(x,y) \leq 2^j * D_{H_k}(x,y)$. And if x and y are connected in G, then the maximum path length possible is O(n), where n is the number of vertices, and since x and y will also be connected in H_k, we have that $D_{H_k} = A_{H_k} = 1$. So, 2^k = O(n) or k = O(log n).

So, the **total time complexity is O($n^\omega * logn$)**

4. Universal Hashing

Let U = [0, M - 1] be a universe of M elements, p be a prime number in range [M, $2M$], and $n(«M)$ be an integer. Consider the following hash functions (where $r \in [1, p\ \ 1]$):

$$H(x) := (x) \bmod n$$
$$H_r(x) := ((rx) \bmod p) \bmod n$$

(a) Initialize S to empty. Repeat n times: Choose a random integer in U and add it to S. Prove that:

$$Prob(\text{max-chain-length in hash table of } S \text{ under hash-function } H(\cdot) > log_2 n) \leq \left(\frac{1}{n}\right)$$

(b) 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 the which maximum chain length in hash-table corresponding to $H_r(x)$ is $\Theta(n)$.

(c) 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, $2n$, $3n$, . . . , $(k\ \ 1)n$ and $n\ \ k$ random elements in U.

Obtain a plot of Max-chain-length for hash functions H() and $H_r()$ over different choices of sets S_k as defined above. Note that you must choose a different random r for each choice of S_k. Provide a justification for your plots.

Solution for (a):

Since the n numbers are chosen randomly, for any x Prob(H(x) = i) = $\frac{1}{n}$ and Prob(H(x) $\neq$ i) = 1- $\frac{1}{n}$

Let X_{ij} be the indicator random variable for having exactly k elements at i^{th} position i.e. $X_{ij} = 1$ iff length of chain at i^{th} position is k.

Let's first calculate Prob($X_{ij} = 1$). For this we need j elements to have hash value equal to i and remaining n-j elements should have hash value other than i.
We can select j elements from a set of n elements in $\binom{n}{j}$ ways so Prob($X_{ij} = 1$) = $\binom{n}{j}(\frac{1}{n})^j(1-\frac{1}{n})^{n-j}$

Let Y_{ij} be the indicator random variable for having **atleast** k elements at i^{th} position i.e. $Y_{ij} = 1$ iff length of chain at i^{th} position is atleast k.
So by similiar argument as above: Prob($Y_{ij} = 1$) $\leq \binom{n}{j}\ (\frac{1}{n})^j$
Now,
Prob(max-chain-length in hash table of S under hash-function H(·) > k) $\leq$ Prob($\exists$ i such that length of chain at i^{th} position is greater than k)
$\leq \sum_{l=0}^{n-1} P(Y_{lk} = 1)$
$\leq n * P(Y_{lk} = 1)$
$\leq$ n * $\binom{n}{k}\ (\frac{1}{n})^k$
$\leq$ n * $\frac{n!}{(n-k)!(k!)}\ (\frac{1}{n})^k$
$\leq$ n * $\frac{n^k}{(k!)}\ (\frac{1}{n})^k$
$\leq$ n $(\frac{1}{k!})$
we can note that for any k : k! = k* (k-1) * (k-2) * (k-3) 3*2 1
There are atleast $\frac{k}{2}$ terms lower bounded by (k/2). so k! $\geq (\frac{k}{2})^{\frac{k}{2}}$
$\implies \frac{1}{k!} \geq \frac{1}{(\frac{k}{2})^{\frac{k}{2}}}$

Now :$(\frac{log_2 n}{2})^{\frac{log_2 n}{2}} \geq (\frac{2^5}{2})^{\frac{log_2 n}{2}} for\ n \geq 2^{2^5}$
$\geq (2)^{2*log_2 n} for\ n \geq 2^{32}$
$\geq (2)^{log_2 n^2} for\ n \geq 2^{32}$
$\geq n^2 for\ n \geq 2^{32}$
$\implies$ n $(\frac{1}{k!}) \leq n * \frac{1}{n^2} \leq \frac{1}{n}$

By presenting a better lower bound for k!, we can present a good lower bound for values of n.That's Proved.

Solution for (b):
Since the cardinality of U is M and we are first creating a mapping [0,M - 1]-> [0,p - 1] where p belongs to [M, 2M] defined by $H_r(x)$ So after applying this map to our M elements, after this we are taking mod with n so basically all m elements are mapped to n slots.This situation is equivalent to throwing M balls in n buckets where i^{th} ball is thrown in $H_r(i)$ th bin. So by pigeonhole principle, there exists atleast one bucket containing M/n elements. Now from this bucket, if we were to choose n elements there are $\binom{M/n}{n}$ possible subsets to choose from and for all these subsets the maximum chain length will be $\theta(n)$ so max chain length is lower bounded by constant multiple of n and also upper bounded by a constant multiple of n. So we will get $\binom{M/n}{n}$ possible subsets for which max chain length will be theta n. This can be further explained by the fact that the balls(elements) present in a particular bucket have same hash value so all theses subsets of M/n values will have their corresponding inverse value in U . This implies that there are atleast $\binom{M/n}{n}$ subsets which maps to the same value. Since r is given to us, also these elements will go to same slot giving us a max chain length of θ(n).

Solution for (c):

Code:

```
import random
import matplotlib.pyplot as plt
n = 100
M = 10000
def isprime(n):
    for i in range(2,n//2 + 1):
        if n%i == 0:
            return False
    return True
def getPrime(a,b):
    k = a
    while k<=b:
        if isprime(k):
            return k
        k = k + 1
    return -1
p = getPrime(M, 2*M)

def h(x):
    return x % n
def hr(x):
    r = random.randint(1, p)

    return ((r*x) %p) %n
```

```
hs = []
hrs = []

def solve(k):
    global hs, hrs
    sk1 = [i*n for i in range(k)]
    sk2 =  random.sample(range(1, M), n - k)
    sk = sk1 + sk2
    counts_h = [0 for i in range(n)]
    counts_hr = [0 for i in range(n)]
    for i in range(n):
        h_val = h(sk[i])
        hr_val = hr(sk[i])
        counts_h[h_val] +=1
        counts_hr[hr_val] += 1
    max_h = 0
    max_hr = 0
    for i in range(n):
        if counts_h[i] > max_h:
            max_h  =counts_h[i]
        if counts_hr[i] > max_hr:
            max_hr = counts_hr[i]
    hs.append(max_h)
    hrs.append(max_hr)



for k in range(1, n+1):
    solve(k)

ks = [i for i in range(1, n +1)]
plt.plot(hs , ks ,color = 'red',label= "(x) mod n")
plt.plot(hrs, ks, color = 'green', label = "((r x) mod p ) mod n")
plt.xlabel("Value of k")
plt.legend(loc="lower right")
plt.ylabel("Maximum chain length")
plt.title("Plot of maximum chain length for different values of k")
plt.show()
plt.savefig("A3.png")
print("Plot saved to ./A3.png")
```

Plot:

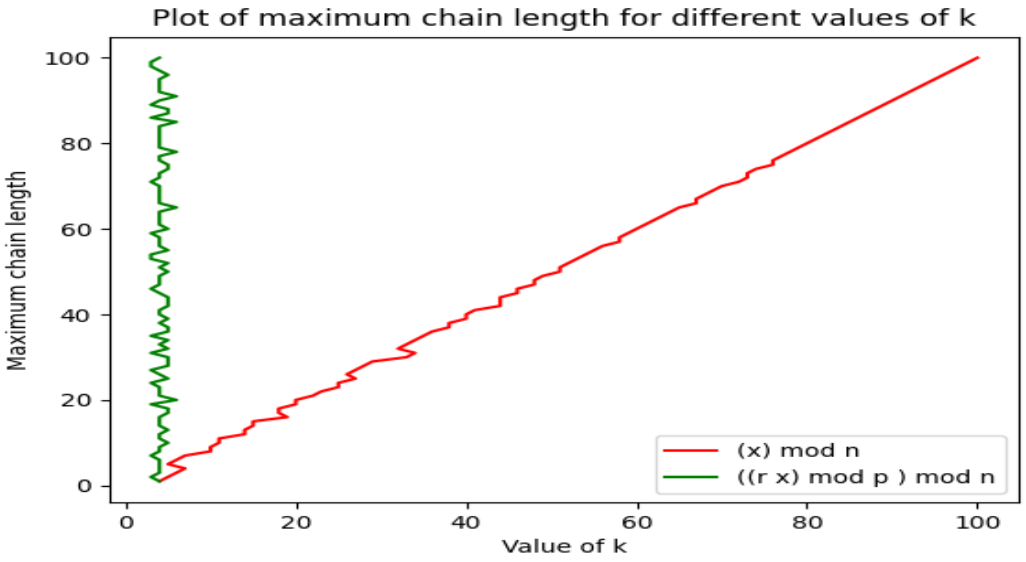

Figure 2: Plot for Maximum chain length for different values of k

Explanation:

The graph corresponding to function H(x) = (x) mod n shows behaviour as y = x line . Since first k element of our set S_k are always mapped to same slot i.e. 0 so maximum chain length always remains close to value of k and since other n-k elements are generated randomly, they can be termed as "good set" for function H so they are spread almost uniformly in our set and hence graph remains almost close to y = x.

The graph corresponding to function $H_r(x)$ = ((r x) mod p) mod n shows behaviour as y = constant line where the constant lies near 1 and 2 . Since we have added randomisation in the function also, the n - k elements still remain "good set" for this function but due to factor r which is a random number, the first k elements also become randomly distributed and become a good set for function $H_r(x)$, So all the n elements are spread nearly uniformly and hence the maximum chain length obtained in this is nearly constant i.e. O(1) and is near 2 and 3 which states that this is a very hood hash function as compared to H(x)