Aman Gupta
2019CS10673
Arpit Chauhan
2019CS10332

Assignment - 3
COL351 - Analysis and Design of Algorithms
Term: Fall 2021
Due Date: 24^{th} October, 2021

Problem 1
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, i.e. the points in Q are corners of the convex-polygon of smallest area that encloses all the points in P .
Design an O(nlogn) 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: In this problem, we use divide and conquer strategy to compute the convex hull of a set P of n input points $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$. We divide the given set of points into two sets, one with the leftmost half points and the other with the rightmost half points and then recursively calculate the convex hulls of both the halves. Then, we merge these convex hulls to obtain the required convex hull of the set P.
Given below is the pseudo code to compute the convex hull of set P using the above Divide-and-conquer approach.

Algorithm 1: Convex Hull using Divide and Conquer

1 If $n \leq 5$, use the brute force algorithm to find the convex hull of the given set of points.
2 Else, sort all the points according to their x co-ordinates and then, partition the given set into two sets, one with the leftmost(lowest x) half points and the other with the remaining points, say P_A and P_B respectively. The sorting is done only once initially and not in the further recursive steps.
3 Recursively compute the convex hulls of sets P_A and P_B using Divide and Conquer.
4 Find the upper and lower tangents for the convex hulls of sets P_A and P_B using Algorithm 2 and Algorithm 3 respectively.
5 Finally, remove all the points lying between the two tangents on both the hulls to obtain the Convex Hull for set P.

Algorithm 2: Finding the Upper Tangent(CH_A, CH_B)

1 Let CH_A be the left convex hull and CH_B be the right convex hull forming convex polygons C_A and C_B respectively. Let p be the rightmost point on C_A and q be the leftmost point on C_B.
2 While pq is not above both the polygons C_A and C_B, do
3 While pq extended passes through the inside of polygon C_A, do
4 Move p counter-clockwise to the next point on C_A
5 While pq extended passes through the inside of polygon C_B
6 Move q clockwise to the next point on C_B
7 Return pq

Algorithm 3: Finding the Lower Tangent(CH_A, CH_B)

1 Let CH_A be the left convex hull and CH_B be the right convex hull forming convex polygons C_A and C_B respectively. Let p be the rightmost point on C_A and q be the leftmost point on C_B.
2 While pq is not below both the polygons C_A and C_B, do
3 While pq extended passes through the inside of polygon C_A, do
4 Move p clockwise to the next point on C_A
5 While pq extended passes through the inside of polygon C_B, do
6 Move q counter-clockwise to the next point on C_B
7 Return pq

Runtime Analysis

We sort the points according to their x co-ordinate once at the start of the algorithm which takes O(nlogn) time. Algorithm 2 and 3 both take O(n) time because in each iteration of the while loop, we move either p or q in the same direction(up in the case of Upper tangent and low in the case of lower tangent) and hence, in atmost n/2 steps each for p and q, we reach the required tangent. Removing the points between the 2 tangents will also be O(n) as there are n points. Hence, the entire merging step of the two convex hulls takes O(n) time.
Let T(n) be the asymptotic running time of the algorithm described above. We can observe that it will follow the recurrence relation given below -

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) && \text{if } n > 5 \\ &= O(1) && \text{otherwise} \end{aligned}$$

Now, we use the substitution method to solve this recurrence.

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn/2) + cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &. \\ &. \\ &= 2^k T(n/2^k) + kcn \end{aligned} \tag{1}$$

Taking k = $log_2 n$, the first term in the final expression of T(n) comes out to be O(n) and the second term as O($nlog_2 n$), hence the overall time complexity of the algorithm comes out to be $\mathbf{O(}nlog_2 n\mathbf{)}$.

Proof of Correctness

Claim 1: During the execution of Algorithm 2, pq never meets the interior of C_A or C_B. Similarly for Algorithm 3.
Proof: We will prove this by induction. Initially, p is the rightmost point of the left half and q is the leftmost point of the right half. The claim is trivially true for this base case configuration. Now let at any point of time, the claim is true. Then in the next iteration of the while loop, p will be moved counter-clockwise to p' only if pq extended passes through the inside of polygon C_A which implies that p'q also does not meet the interior of C_A. Similarly for C_B and similarly for Algorithm 3. Thus, the claim is proved by induction.

Claim 2: The algorithm for finding the upper tangent terminates and we can always find it. Similarly for lower tangent.
Proof: Using claim 1, we can say that the inner while loops cannot run indefinitely because pq never meets the interior of C_A or C_B. In every iteration of the outer while loop for the upper tangent, some more boundary points go below pq hence in a finite number of iterations, all the points will be below pq and it will become the upper tangent. Similarly for lower tangent. Hence, both the algorithms always terminate and the claim is proved.

Claim 3: The final set of points obtained after merging the two smaller convex hulls enclose all the n points.
Proof: Since the upper tangent lies above both the polygons and the lower tangent lies below both the polygons, all the points lie between them. The left boundary of the left half convex hull and the right boundary of the right half convex hull remain intact in the merged Convex Hull and thus, all the n points lie in the polygon formed by the merged Convex Hull.

Claim 4: The polygon obtained after merging the two Convex hulls is Convex.
Proof: Let the left and right intersection point of the upper tangent be a and b respectively and similarly c and d respectively for the lower tangent. Now consider the points moving clockwise from b to d including both of them. Since they are part of the smaller right convex hull, they form a convex structure. Similarly since points moving clockwise from c to a are part of the left convex hull, those points also form a convex structure.
Since all the points lie below the upper tangent and above the lower tangent, the line segment joining any one point from the left polygon to another point in the right polygon will lie between the two tangents and cannot go outside the polygon as can also be observed from Figure 1. Hence, the polygon formed will be convex.

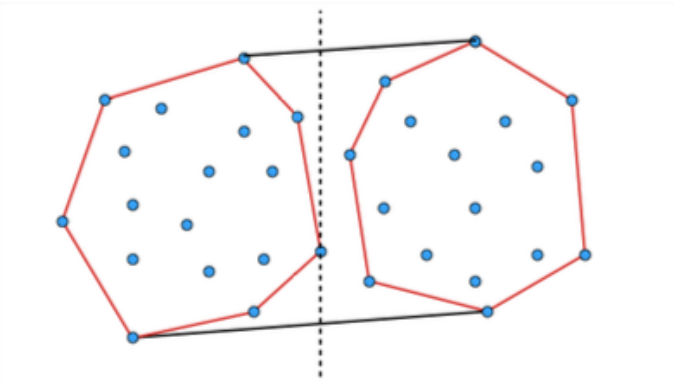

Figure 1: Merging Step

Claim 2 tells us that we will always be able to find the upper and lower tangents for the two polygons and thus, can execute Algorithm 1 to obtain the final convex hull. Claim 3 ensures that all the points are enclosed by the final polygon obtained and Claim 4 tells us that this polygon is convex. So, now we have obtained a subset Q of P which forms a convex polygon enclosing all the points of set P. Thus by the uniqueness of Convex Hull, we can say that the result of merging the two smaller convex hulls is the Convex Hull of the set P.
Hence, the algorithm is correct.

Problem 2
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 running time of the naive program is $O(n^2)$. Your task is to design an algorithm that computes all the forces F_j in O(nlogn) time.

Solution: The total net force on particle j, by Coulomb's Law, is equal to

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

which can be written as

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

Let us define p as the first summation inside the brackets and q as the second summation.Now, consider the following 3 polynomials,

$$A(x) = q_1x + q_2x^2 + ... + q_nx^n$$

$$B(x) = \frac{x}{1^2} + \frac{x^2}{2^2} + ... + \frac{x^{n-1}}{(n-1)^2}$$

$$C(x) = \frac{x}{(n-1)^2} + \frac{x^2}{(n-2)^2} + ... + \frac{x^{n-1}}{1^2}$$

Consider the polynomial products A(x)B(x) and A(x)C(x). We observe that,

Coefficient of x^j in $A(x)B(x) = \sum_{i<j} \frac{q_i}{(j-i)^2}$ and
Coefficient of x^{n+j} in $A(x)C(x) = \sum_{i>j} \frac{q_i}{(i-j)^2}$

Hence, the coefficient of x^j (say $[x^j]$) in the expression A(x)B(x) - $\frac{A(x)C(x)}{x^n}$ will be given by,

$$[x^j] = \sum_{i<j} \frac{q_i}{(j-i)^2} - \sum_{i>j} \frac{q_i}{(i-j)^2}$$

Thus, the total net force on particle j can be written as, $F_j = Cq_j[x^j]$. We will use Fast Fourier Transform(FFT) for calculating the 2 polynomial products and then, the force on each particle F_j can be directly calculated from the coefficients.

Algorithm 4: Computing net Force F_j on each particle

1 Compute coefficients of the polynomials A(x), B(x) and C(x) as described above.
2 Calculate the polynomial products P(x) = A(x)B(x) and Q(x) = A(x)C(x) using Fast Fourier Transform(FFT) as described during the lectures.
3 Let F = $F_1, F_2, ...F_n$ denote the net forces on the particles.
4 For j ← 1 to n, do
5 $g \leftarrow$ Coefficient of x^j in P(x)
6 $h \leftarrow$ Coefficient of x^{n+j} in Q(x)
7 $F_j \leftarrow Cq_j(g-h)$
8 Return F

Runtime Analysis

A(x),B(x) and C(x) are all polynomials of degree less than n with simple coefficients each of which can be calculated in constant time. Hence, the 3 polynomials can be calculated in O(n) time.
Next, we use FFT as described in class to calculate both the polynomial products, P(x) and Q(x) in O(nlogn) time each.
Then using the coefficients of P(x) and Q(x), we can calculate net Force, F_j in O(1) time each taking the overall time for this step to be O(n).
Therefore, the algorithm computes all the forces F_j in O(nlogn) overall time.

Proof of Correctness

Claim 1: Coefficient of x^j in $A(x)B(x), g = \sum_{i<j} \frac{q_i}{(j-i)^2}$

Proof: When we multiply A(x) and B(x), for every x^i term in A(x), product with x^{j-i} term in B(x) yields an addition of $\frac{q_i}{(j-i)^2}$ in the coefficient of x^j. Since B(x) has all indices positive,only terms with i<j will have contribution to the coefficient of x_j.Hence, the claim is proved.

Claim 2: Coefficient of x^{n+j} in $A(x)C(x), h = \sum_{i>j} \frac{q_i}{(i-j)^2}$

Proof: When we multiply A(x) and C(x), for every x^i term in A(x), product with x^{n+j-i} term in B(x) yields an addition of $\frac{q_i}{(i-j)^2}$ in the coefficient of x^{n+j}. Since B(x) has all indices less than n, n+j-i $<$ n which implies that only terms with i>j will have a contribution to the coefficient of x^{n+j}.Hence, the claim is proved.

The correctness of FFT follows from the discussion during the lectures.
Hence, using Claim 1 and Claim 2, we can say that the line 7 of the Algorithm described above correctly calculates the total net forces F_j on all the particles as per the Coulomb's Law i.e.

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

Thus, the algorithm is correct.

.

Problem 3.1
Let G = (V, E) be an unweighted undirected graph, and H = (V, E_H) be a undirected graph obtained from G that satisfy: (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 E_H be the distance-matrix of H.
Prove that the graph H = (V, E_H) can be computed from G in O(n^ω) time, where ω is the exponent of matrix-multiplication.

Solution:
H and G are both undirected and unweighted graphs. The graph H has the same vertex set as that of graph G, with some additional edges. As per the given condition an edge (x,y) exists in graph H if it is an edge in G or their exists a vertex z, such that, (x,z) and (z,y) are edges in G. Let A_G be the adjacency matrix of graph G.

Claim A_G^2(i,j) > 0, iff there exists a vertex k, such that, (i,k) and (k,j) are edges in G
Proof By matrix multiplication we can note that,

$$A_G^2(\text{i,j}) = \sum A_G(i, x) * A_G(x, j)$$

$A_G(i, x)$ is non-zero only for neighbours of i, and
$A_G(x, j)$ is non-zero only for neighbours of j.
Hence, A_G^2(i,j) > 0 if and only if i and j have a common neighbour in the graph G. Therefore, there exists a vertex k, such that, (i,k) and (k,j) are edges in G.

Claim The adjacency matrix of graph H, A_H equals $A_G \vee A_G^2$
Proof We can notice that since we have a logical or operation in A_G and its square, all edges of G are present in H. Additionally, we now need to prove that all vertices which are next neighbours in G are neighbours in H. Using the previous claims we can say that for next neighbour pairs in G, $A_G^2 > 0$ therefore this covers the second given condition of edges in H. A edge in H is either as direct edge which is covered by A_G or a next neighbour edge covered by A_G^2

Using the above two claims we find the adjacency matrix of H by calculating logical or operation of adjacency matrix of G and its square. Graph G has n vertices so the time taken in finding graph H is,
a) Obtaining the adjacency matrix of G - O(n^2)
b) Computing A_G^2 - O(n^ω), here ω is the exponent of matrix-multiplication
c) Taking logical Or of two matrices - O(n^2)
Since, $\omega \geqslant 2$, the overall time complexity is O(n^ω)

Problem 3.2
Argue that for any x,y ∈ V , D_H(x,y) = $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$

Solution: We will prove the given argument by the following claims:
Claim D_H(x,y) ≤ $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$
Proof: We can show that there exists a path in the graph H with length equal to ceil($D_G(x,y)/2$). There exists a path in the graph G with length $D_G(x,y)$. If this length is even then we can take every alternate vertex on the path to obtain a path of length $D_G(x,y)/2$ in the graph G. Else if the length is even then we can take every alternate vertex till the neighbour of y and then the edge from that neighbour to y. This will be of length ceil($D_G(x,y)/2$) in graph H. Since, there exits a path with length ceil($D_G(x,y)/2$) and $D_H(x,y)$ is the length of shortest path, hence the claim holds.

Claim D_H(x,y) ≥ $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$
Proof: We will prove this claim by contradiction. Assume that there exists a path in the graph H, whose length is lesser than ceil($D_G(x,y)/2$). Again considering two cases that $D_G(x,y)$ is odd and even. In even case, we have path in H whose length is less than $D_G(x,y)/2$. Since the graph H contains all edges of G and also edges between next neighbour, there will be a corresponding path in G, whose length will be less than 2*$D_G(x,y)/2$ (that is, $D_G(x,y)$). This leads to a contradiction because $D_G(x,y)$ is the shortest path length in graph G between x and y. Consider the case when $D_G(x,y)$ is odd. Hence, there exists a path of length strictly less than $(D_G(x,y)+1)/2$ in the graph H between vertex x and y. The corresponding length of path in the graph G will be 2*(Length of path in H). This will be strictly less that $D_G(x,y)+1$, and cannot be equal to $D_G(x,y)$ because its even, while $D_G(x,y)$ is odd. So it will be strictly less than $D_G(x,y)$. This again leads to a contradiction.

Using the above two claims we have proved that, D_H(x,y) = $\left\lceil \frac{D_G(x,y)}{2} \right\rceil$

Problem 3.3
Let A_G be adjacency matrix of G, and M = D_H A_G. Prove that for any x, y ∈ V, the following holds,

$$D_G(x,y) = \begin{cases} 2D_H(x,y) & M(x,y) \geq \text{degree}_G(y) \cdot D_H(x,y) \\ 2D_H(x,y) - 1 & M(x,y) \lneq \text{degree}_G(y) \cdot D_H(x,y) \end{cases}$$

Solution:
We will begin by obtaining the relation between M(x,y) and $degree_G(y).D_H(x,y)$. From matrix product we know,

$$M(\text{x,y}) = \sum D_H(x,z) * A_G(z,y)$$

The term $A_G(z,y)$ will be non-zero only if z is a neighbour of y, else it will be zero.
So, effectively M(x,y) is the sum of distances in graph H of vertex x from vertex v belonging neighbours of y in graph G.

Claim: For any neighbour v, of vertex y in graph G, $D_G(x,y) - 1 \leq D_G(x,v) \leq D_G(x,y) + 1$
Proof If the distance between (x,v) is string less than $D_G(x,y) - 1$ then, the distance of (x,y) will also be less than $D_G(x,y)$ which leads to a contradiction. Similarly, since there exists a path from x to v of length $D_G(x,y) + 1$ so $D_G(x,v)$ will be shorter than that.

So using the previous claim we can say that the distance between x and the neighbours of y in graph G can be, $D_G(x,y) - 1$, $D_G(x,y)$ or $D_G(x,y) + 1$. Let,
a = Number of neighbours of y with distance $D_G(x,y) - 1$
b = Number of neighbours of y with distance $D_G(x,y)$
c = Number of neighbours of y with distance $D_G(x,y) + 1$
Therefore, $degree_G(y)$ = a+b+c
Using the result obtained in part(b), we can see that:
M(x,y) = a.[ceil($D_G(x,y) - 1/2$)] + b.[ceil($D_G(x,y)/2$)] + c.[ceil($D_G(x,y) + 1/2$)].
$degree_G(y).D_H(x,y)$ = (a+b+c).[ceil($D_G(x,y)/2$)]

Claim M(x,y) ≥ $degree_G(y).D_H(x,y)$ if and only if $D_G(x,y)$ is even.
Proof Let $D_G(x,y)$ be p, then the equation M(x,y) - $degree_G(y).D_H(x,y)$, simplifies to
K(say) =a(ceil[p-1/2] - ceil[p/2]) + c(ceil[p+1/2]-ceil[p/2])
We can prove one half of the claim by contradiction, assume that p is odd and K ≥ 0, then we know K = a((p-1)/2 -(p+1)/2) + c((p+1)/2 - (p+1)/2) = -a. Since a is number of nodes it will be greater than zero. hence K will be less than zero which leads to contradiction. The other half of the proof can be done by assuming p to be even, then K = a(p/2 - p/2) + c((p+2)/2 - p/2) = c. Since c *geq* 0, K is also greater than zero. Hence, the claim is proved.

We can similarly prove that M(x,y) < $degree_G(y).D_H(x,y)$ if and only if $D_G(x,y)$ is odd. There will be strict in equality in this case because a is strictly greater than 0.
Using the claim in part(b) we know that when D_G is even, $D_H = (D_G)/2$ and when D_G is odd, $D_H = (D_G + 1)/2$. Combining this with the above claims we will reach to our proof.

Problem 3.4
Use (c) to argue that D_G is computable from D_H in $O(n^\omega)$ time.

Solution: We need to obtain D_G from D_H, we will use the procedure discussed in part(c), each steps takes the following amount of time,
1) Computing M matrix by multiplication of D_H and A_G. - $O(n^\omega)$
2) Computing degrees for all vertices of the graph G - $O(n^2)$
3) For all pair of points checking relation between M(x,y) and $degree_G(y).D_H(x, y)$ to assign value according to relation described in part(c). - $O(n^2)$.

Where, ω is the exponent of matrix multiplication, and $\omega \geq 2$.
Hence, the overall time complexity is $O(n^\omega)$

Problem 3.5
Prove that all-pairs-distances in n-vertex unweighted undirected graph can be computed in O(n^ω logn) time, if ω is larger than two.

Solution: We will use the above approach to reach to all pair shortest path matrix in O(n^ω logn) time, this is also known as **Seidel's algorithm**
Let A be the adjacency matrix of the graph.

Algorithm:
1) If the graph is a complete graph that is, if A is all 1 matrix then since all nodes are connected to each other return 1 matrix.
2) Keep computing the matrix D = (A $\vee$ A^2) for the adjacency matrix till we reach an all one matrix.
3) Since we know the distance matrix of an all one matrix we can calculate the same for the previous matrix using the method described in part(c).
4) Keep repeating the step (3) till we reach the distance matrix of A.

Time Complexity
1) Computing the squares each time takes O(n^ω time and this is done for a maximum of [log(n)] times because on each squaring the diameter of graph reduces
by a factor of 2. - O(n^ω logn)
2) Obtaining the distance matrix using the previous matrix distance vector requires O(n^ω time (part(d)) and is also done at max [log(n)] times.

Therefore, the overall complexity to calculate the shortest distance between all points is O(n^ω logn)

Problem 4.1
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.

Prove the following. [7 marks]

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

Solution:
We need to find the probability to that the length of the maximum chain in hash function H(.) should be greater than log(n). Let X_i be the length of the chain corresponding to the value i. We need to find the probability of X_i greater than log(n). Applying **union-bound** we get,
P(union($X_i \geq$ log(n))) $\geq$ Summation(P($X_i \geq$ log(n)))

Considering the probability for some constant p,
P($X_i \geq$ log(n)) = P(p.$X_i \geq$ plog(n)) = P($2^{p.X_i} \geq 2^{plog(n)}$)
which equals, P($2^{p.(X_i)} \geq n^p$)

Now we apply **Markov Inequality** on the inequality, we obtain,
P($2^{p.(X_i)} \geq n^p$) $\geq$ E[$2^{p.(X_i)}$]/n^p

Applying the same bound on the submission obtained after union-bound we get,
P(Max Chain Length $\geq$ log(n)) $\geq$ Summation(E[$2^{p.(X_i)}$])/n^p

Taking certain values of p we obtain the required the required inequality.
P(Max Chain Length $\geq$ log(n)) $\geq$ (1/n)
Note, that the bound is not valid for small number but is true for larger n.

Problem 4.2
Prove that for any given r $\in$ [1, p - 1], there exists at least (M/n)Cn subsets of U of size n in which maximum chain length in hash-table corresponding to H_r(x) is θ(n).

Solution:
We know that the universe consists of M elements. In the Hash function H(.), they will be mapped after taking modulo with n. There are in total n values possible after taking modulo, which are [0,1,2,....n-1]. Hence, all M elements will map to one of these n values by the Hash function H(.). By **Pigeonhole Principle**, one out of these n values will have more than M/n elements mapped to it. So if we take any n elements out of these M/n elements mapped to the same value, we will get a maximum chain length of the order θ(n). Therefore, the total number of such subsets is at least (M/n)Cn. Hence, the result is proved.

Problem 4.3
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 ∈ [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(), $H_r()$ 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:

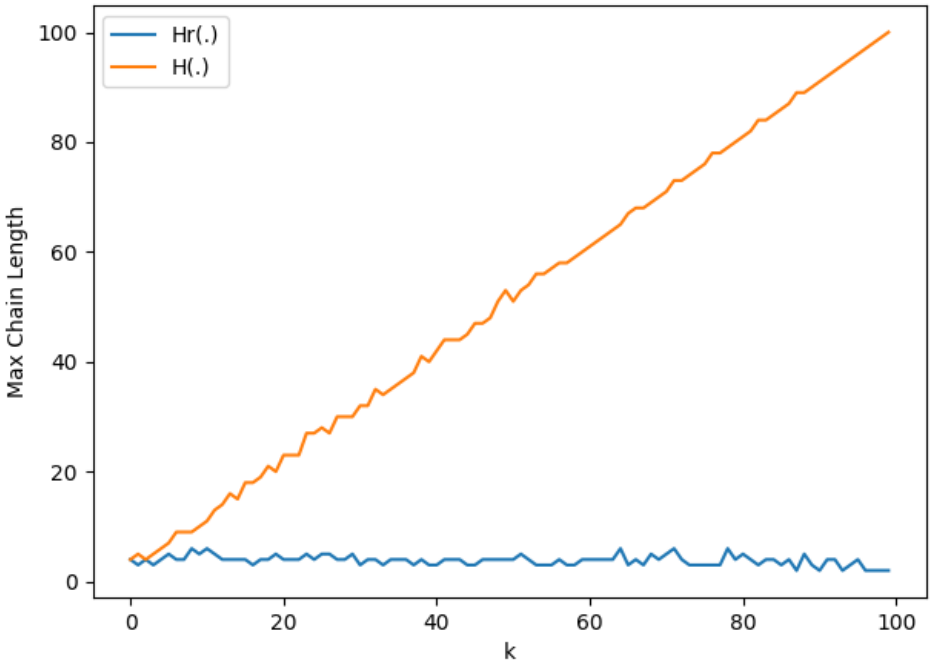

Figure 2: Plot of Max chain length for H(.) and $Hr(.)$

Implementation Details
The set S_k is set as $\{0, n, 2n, 3n, ..., (k-1)n\}$ and n−k random elements from the universe. The choice of r is made random for each S_k from the range [1,p-1]. p is taken as 17417.

Justification
Since the set S_k has atleast k elements that have the same modulo of zero mod(n). Therefore max chain length is atleast k in the case of H(.). The rest n - k elements do not affect the graph that significantly as they are randomly chosen.
In case of $H_r(.)$ the number are processed with a random r for a certain p, thus creating second universal. Therefore, the max length is this hash function is lesser than the hash function H(.).
The code is provided below.

```
import random
from matplotlib import pyplot as plt

M = 10000
n = 100
m_l = []
for k in range(1,n+1):
  S = []
  for i in range(k):
    S.append(i*n)
  for i in range(n-k):
    r = random.randint(0,9999)
    S.append(r)
  dic = {}
  for x in S:
    v = x%n
    if v in dic:
      dic[v] += 1
    else:
      dic[v] = 1
  m_l.append(max(list(dic.values())))

p = 17417
r_l = []
for k in range(1,n+1):
  S = []
  for i in range(k):
    S.append(i*n)
  for i in range(n-k):
    c = random.randint(0,9999)
    S.append(c)
  dic = {}
  r = random.randint(1,p-1)
  for x in S:
    v = ((r*x)%p)%n
    if v in dic:
      dic[v] += 1
    else:
      dic[v] = 1
  r_l.append(max(list(dic.values())))


plt.figure(figsize=(7,5))
plt.plot(r_l,label ="Hr(.)")
plt.plot(m_l,label = "H(.)")
plt.xlabel("k")
plt.ylabel("Max Chain Length")
plt.legend()
plt.savefig("Out.png")
```