Indian Institute of Technology Delhi

# COL351 Analysis and Design of Algorithms: Assignment 3


Ishita Chaudhary: 2019CS10360

Mohammed Jawahar Nausheen: 2019CS10371

# Contents

# Convex Hull

Finding the convex hull for a given set of points involves finding a subset of the given points joining which gives us a convex polygon of minimum area that encloses all of the given points. We adopt a divide and conquer strategy in solving the problem.

***Divide*** Divide the set of points into two equal-sized groups according to the median in x coordinate.

***Recursively solve*** for convex hull for the two left and right groups separately.

***Combine*** The two convex hulls are combined by finding the upper and lower tangents to create one common convex polygon enclosing all the points. Finding the tangents needs to be done in linear time.

In the proposed algorithm, to check if the line $C_1[u_1]C_2[u_2]$ crosses the polygon say $C_2$, we check if the point $C_2[(u_2 + 1)modn_2]$ lies below the line. If it does, then the three points in order will be in a clockwise orientation. This can be checked from the slopes of the two lines easily. In this way, this step takes only some O(1) computation. Similarly, to check if the line $C_1[u_1]C_2[u_2]$ lies above $C_1$, $C_1[(n_1 + u_1 - 1)modn_1]$, $C_1[u_1]$ and $C_2[u_2]$ should lie in a clockwise orientation. Similar checks are made in the case of lower tangent.

***Correctness***: Assuming that we have the correct convex hulls for the sub-problems, we will show that the while loops terminate only when we find the tangents to the two convex polygons. Since, all the points lie below the upper tangent and above the lower tangent, we can claim that the condition of enclosing all the points inside the polygon is satisfied by the figure formed. Furthermore, the tangents are chosen such that the angles at the points of tangency are convex, thus the generated polygon is also convex.

For the upper tangent, starting from the right-most point of the left polygon and left-most point in the right polygon, we gradually move the ends of the required tangent in the clockwise direction on $C_2$ and anti-clockwise direction on $C_1$. If $C_1[u_1]$ and $C_2[u_2]$ are the ends of the line and the line crosses $C_2$ then $C_2[(u_2 + 1)modn_2]$ should be lying above this line and the orientation is not clockwise, the loop continues. The same applies for points from $C_1$. Finally, when the orientation is clockwise, the point $C_2[(u_2+1)modn_2]$ will have to lie below the obtained tangent line. Additionally, since $C_2$ is a convex polygon, all other points will also lie below this tangent. Similar argument also holds for $C_1$

In a similar way, we can see that the line for lower tangent gradually moves downwards until all points lie above it.

***Complexity Analysis***: The problem is reduced into two sub-problems merging which requires O(n) time. Finding the right-most and left-most points of the convex hulls of sub-problems takes at most O(n) time. Finding the upper tangent again takes at most O(n) time since checking for orientation (crossing) takes O(1) time and each vertex can be visited at most once. Similarly, finding the lower tangent also takes O(n) time. The recursion to the problem now reduces to T(n) = 2T(n/2) + O(n) which when simplified becomes O(nlogn) according to Master's Theorem.

**Function** `Orientation`$((x_1, y_1), (x_2, y_2), (x_3, y_3))$**:**
result $\leftarrow (y_2 - y_1)(x_3 - x_2) - (x_2 - x_1)(y_3 - y_2)$
**if** *result == 0* **then return** *collinear*
**else if** *result > 0* **then return** *clockwise*
**else return** *counter-clockwise*

**Function** `ConvexHull`(*points P*)**:**
**if** *size(P)* $\leq$ *5* **then** solve trivially
Find median point $x_m$ according to x-coordinate
$P_1 \leftarrow$ Set of points to the left of $x_m$ and $x_m$
$P_2 \leftarrow$ Set of points to the right of $x_m$

Lists of vertices in clockwise direction
$C_1 \leftarrow$ `ConvexHull(`$P_1$`)`
$C_2 \leftarrow$ `ConvexHull(`$P_2$`)`

`/* Find upper tangent */`
$u_1 \leftarrow$ index of right-most vertex of $C_1$
$u_2 \leftarrow$ index of left-most vertex of $C_2$
**while** *line* $C_1[u_1]C_2[u_2]$ *crosses* $C_1$ *or* $C_2$ **do**
  **while** `Orientation` *(* $C_1[u_1], C_2[u_2], C_2[(u_2 + 1) mod n_2]$*) is not clock-wise* **do**
   $u_2 \leftarrow (u_2 + 1) mod n_2$
  **while** `Orientation` *(* $C_1[(n_1 + u_1 - 1) mod n_1], C_1[u_1], C_2[u_2]$*) is not clock-wise*
   **do** $u_1 \leftarrow (n_1 + u_1 - 1) mod n_1$
**end**
`/* Find lower tangent */`
$l_1 \leftarrow$ index of right-most vertex of $C_1$
$l_2 \leftarrow$ index of left-most vertex of $C_2$
**while** *line* $C_1[l_1]C_2[l_2]$ *crosses* $C_1$ *or* $C_2$ **do**
  **while** `Orientation` *(*$C_1[l_1], C_2[l_2], C_2[(n_2 + l_2 - 1) mod n_2]$ *is not counter-clockwise* **do** $l_2 \leftarrow (n_2 + l_2 - 1) mod n_2$
  **while** `Orientation` *(* $C_1[(l_1 + 1) mod n_1], C_1[l_1]C_2[l_2]$*) is not counter-clockwise*
   **do** $l_1 \leftarrow (l_1 + 1) mod n_1$
**end**

result $= \phi$
Add all points of $C_1$ from $l_1$ to $u_1$ in clockwise direction. Add all points from $u_2$ to $l_2$ in clockwise direction.
**return** result

# Particle Interaction

Given that charged particles $q_1, q_2, q_3, ..., q_n$ (the charge may be positive or negative) are placed at regular spacing along a straight line, at positions 1, 2, 3, ..., n on the real line respectively.

We want to calculate the total net force on each charged particle in O(n logn) time. We will solve this problem by polynomial multiplication using Fast Fourier Transformation, which is an O(n logn) algorithm.

Consider the following two polynomials,

Q(x)=$q_1 + q_2x + q_3x^2 + ... + q_nx^{n-1}$, and

D(x)=$(-1/(n-1)^2) + (-1/(n-2)^2)x + (-1/(n-3)^2)x^2 + ... + (1/(n-1)^2)x^{2n-2}$

**Algorithm:** The coefficient vectors of the above two polynomials are Q= $(q_1, q_2, q_3, ..., q_n)$ and D= $(-1/(n-1)^2, -1/(n-2)^2, ..., -1, 0, 1, ..., 1/(n-2)^2, 1/(n-1)^2)$. Since, the degree of Q(x) (= n-1) < degree of D(x) (= 2n-2); for polynomial multiplication, both the polynomials need to have same number of coefficients. Hence, we add (n-1) more terms to Q(x) with zero coefficient.

We have to find the vector F, the coefficient list of F(x)= Q(x)*D(x), followed by multiplying each term with its corresponding charge and coulomb's constant.

- The desired length is found first, which is $min_i$: $2^i \geq |Q| + |D| - 1$. The lists Q and D are padded with zeroes to make their length same as the desired length.
- The FFT or Fast Fourier Transformation of both D and Q are calculated.
- Further, the dot product (or point-wise multiplication) of these two are performed and stored in vector P.
- Next, inverse FFT of P is calculated, and multiplied with coulomb's constant and the corresponding charge at the position. The algorithm terminates.

**Correctness of Algorithm:** As we know, that Fast Fourier Transformation is used to multiply polynomials, and gives the desired results; we only need to show that the multiplication of polynomials Q(x) and D(x) followed by multiplying each term with its corresponding charge and coulomb's constant gives the expression,

$F_j = \sum_{i<j} \text{C}q_iq_j/(j-i)^2$ - $\sum_{i>j} \text{C}q_iq_j/(j-i)^2$

$\implies \text{F}_j = \text{C}q_j(\sum_{i<j} q_i/(j-i)^2$ - $\sum_{i>j} q_i/(j-i)^2)$

The coefficient vectors Q and D can be represented as

$$Q_i = \begin{cases} q_{i+1} \text{ if i} \in [0, \text{n-1}] \\ 0 \text{ otherwise} \end{cases}$$

$$D_i = \begin{cases} \text{-1}/(n-1-i)^2 \text{ if i} \in [0, \text{n-2}] \\ 0 \text{ if i=n-1} \\ 1/(i+1-n)^2 \text{ if i} \in [\text{n, 2n-2}] \end{cases}$$

Define F as the convolution of vectors D and Q.

**Claim:** $F_{n+j-2} = \sum_{i<j} q_i/(j-i)^2$ - $\sum_{i>j} q_i/(j-i)^2$

**Proof:** By the definition of convolution, we can write,

$F_{n+j-2} = \sum_{i=0}^{n+j-2} Q_iD_{n+j-2-i}$

Break the summation at i = j-2, j-1 and n-1. $\implies \text{F}_{n+j-2} = \sum_{i=0}^{j-2} Q_iD_{n+j-2-i} + Q_{j-1}D_{n-1} + \sum_{i=j}^{n-1} Q_iD_{n+j-2-i} + \sum_{i=n}^{n+j-2} Q_iD_{n+j-2-i}$

As we know $D_{n-1}$=0, and $Q_i$=0 for i $\geq$ 0.

$\implies \text{F}_{n+j-2} = \sum_{i=0}^{j-2} Q_iD_{n+j-2-i} + \sum_{i=j}^{n-1} Q_iD_{n+j-2-i}$

$\implies F_{n+j-2} = \sum_{i=0}^{j-2} q_{i+1}/(j-i-1)^2 + \sum_{i=j}^{n-1} q_{i+1}/(i-j+1)^2$
$\implies F_{n+j-2} = \sum_{i=1}^{j-1} q_i/(j-i)^2 + \sum_{i=j+1}^{n} q_i/(i-j)^2$
The claim is proved.

To obtain the net force on particle j, use $F_{n+j-2}$. This is then multiplied by corresponding $q_j$ and coulomb's constant to obtain the expression needed. Hence, proved.

**Function** `multiplyPolynomial(`*charge list Q, integer n*`):`

```
initialise C                                          /* coulomb's constant */

initialise length:=1
initialise empty list P, F, D
forall i ∈ [−(n − 1), n − 1] do
    if i < 0 then
        D.add(-1/i^2)
    end

    if i = 0 then
        D.add(0)
    end

    if i > 0 then
        D.add(1/i^2)
    end
end
while length < length(Q) + length(D) do
    length=length*2
end
while length(Q) < length do
    Q.add(0)
end
while length(D) < length do
    D.add(0)
end
F_Q = FFT(Q)
F_D = FFT(D)
forall i ∈ [0, length − 1] do
    P.add(F_Q[i]*F_D[i])
end
iFFT_P = inverseFFT(P)
forall i ∈ [0, n − 1] do
    F.add(C*Q[i]*iFFT_P[n-2+i])
end
return F
```

**Complexity Analysis:** First, we analyze the complexity of the *FFT* and *inverseFFT* algorithms. The initialisation of data and checks take O(1) time. Further, initialising the odd and even coefficient list take O(m) time, where m is the length of coefficient list given as input. Now, the recursive calls the function itself twice, i.e., 2T(m/2). The next for loop iterates over m/2 values and is O(m). The only difference between *FFT* and *inverseFFT* is an extra for loop in *inverseFFT* at the end, which is O(m).

**Function** `FFT(`*coefficient list C*`):`

initialise w:=1, $w_n$:=$e^{2\pi i/n}$, n:= length(C)
**if** $n = 1$ **then**
  return C
**end**
initialise empty lists $C_{even}$, $C_{odd}$
**forall** $i \in [0, n-1]$ **do**
  **if** $i$ *is even* **then**
    $C_{even}$.add(C[i])
  **end**
  **if** $i$ *is odd* **then**
    $C_{odd}$.add(C[i])
  **end**
**end**
$Y_{even}$ = FFT($C_{even}$)
$Y_{odd}$ = FFT($C_{odd}$)
**forall** $k \in [0, n/2-1]$ **do**
  Y[k]= $Y_{even}$ [k]+ w$Y_{odd}$ [k]
  Y[k+ n/2]= $Y_{even}$ [k]- w$Y_{odd}$ [k]
  w = w$w_n$
**end**
**return** $Y$

**Function** `inverseFFT(`*coefficient list C*`):`

initialise w:=1, $w_n$:=$e^{-2\pi i/n}$, n:= length(C)
**if** $n = 1$ **then**
  return C
**end**
initialise empty lists $C_{even}$, $C_{odd}$
**forall** $i \in [0, n-1]$ **do**
  **if** $i$ *is even* **then**
    $C_{even}$.add(C[i])
  **end**
  **if** $i$ *is odd* **then**
    $C_{odd}$.add(C[i])
  **end**
**end**
$Y_{even}$ = FFT($C_{even}$)
$Y_{odd}$ = FFT($C_{odd}$)
**forall** $k \in [0, n/2-1]$ **do**
  Y[k]= ($Y_{even}$ [k]+ w$Y_{odd}$ [k])
  Y[k+ n/2]= ($Y_{even}$ [k]- w$Y_{odd}$ [k])
  w = w$w_n$
**end**
**forall** $k \in [0, n-1]$ **do**
  Y[k]= Y[k]/n
**end**
**return** $Y$

Hence, the complexity of both these algorithms can be determined by simplifying the expression, T(m)=2T(m/2)+O(m)+constant, which is O(m logm).

Now, coming to the *multiplyPolynomial*, the initialisation takes constant time. As we know, the charge list has *n* elements and distance list has *2n-1* elements. The first while loop iterates over log(n + (2n-1)) times, i.e, O(logn). The next two for loops iterate over log(n + (2n-1))-n and log(n + (2n-1))-(2n-1) times respectively. Note that the size of charge list and distance list is still O(n). Next, we call *FFT* and *inverseFFT* once, along with a for loop that iterates for O(logn). The complexity of this is O(3mlogm), where m=O(n), which is nothing but O(nlogn); the next for loop is O(m) = O(n). Hence, the overall time complexity of *multiplyPolynomial* is O(nlogn).

# Distance Computation Using Matrix Multiplication

Given G= (V,E) is an unweighted undirected graph, and H= (V, $E_H$) is an undirected graph obtained from G: (x,y) $\in E_H \iff$ (x,y) $\in$ E or there $\exists$ a $w \in$ V such that (x,w), (w,y) $\in$ E.

Notations Used: $D_G$ denotes the distance matrix of graph G, and $D_H$ denotes the distance matrix of graph H.

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

Let $A_G$ denote the adjacency matrix for graph G, then (x,w), (w,y) $\in$ E implies, $A_G$(x,w) = 1 and $A_G$(w,y) = 1 $\iff \sum A_G(x,w) * A_G(w,y) > 0 \iff A_G^2(x,y) > 0$.
Hence, the adjacency matrix $A_H$ can be defined as $A_H(x,y) = 1$ iff $A_G(x,y) = 1$ or $A_G^2(x,y) > 0$. Computing $A_G^2$ can be done in $O(n^w)$ time using matrix multiplication. Computing $A_H$ takes $O(n^2)$ more time, thus the overall time complexity is $O(n^w)$.

## 3.2 (b)

***Claim*** : $D_H(x,y) = \lceil D_G(x,y)/2 \rceil$
The proof is case wise.
***Case 1***: Consider $D_G(x,y)$ to be even, say 2k. Let this 2k length path be x, $v_1, v_2, ....v_{2k-1}$, y. Now according to the definition of H, (x,$v_1$), ($v_1$, $v_2$) $\in$ E implies, (x,$v_2$) $\in E_H$. Hence the path (x,$v_1$),($v_1$,$v_2$) can be replaced by the single edge, (x,$v_2$) in H. Similarly, ($v_2, v_4$), ($v_4, v_6$), ... ($v_{2k-2}$,y) can be used to form a path of length k from x to y. To show that $D_H(x,y)$ is indeed k, we will show that a shorter path cannot exist in H. Consider a d < k length path in H. We convert it to a walk in G by replacing (u,v) with (u,w) and (w,v) from definition of H. Then the walk created is at most of length 2d < 2k. This is a contradiction since $D_G(x,y) = 2k$.
***Case 2***: We now deal with the case of odd length path, say 2k-1, x, $v_1, v_2, ....v_{2k-2}$, y. From case 1, $D_H(x, v_{2k-2})$ is k-1. There exists a path from x to y in H using the edge $(v_{2k-2}, y)$ of length k. Again similar to the above case a path of length lesser than k in H implies a path of length less than 2k-1 in G which is a contradiction. Therefore $D_H(x,y) = k$.

## 3.3 (c)

From the condition proved in (b), $D_G(x,y)$ is either $2D_H(x,y)$ or $2D_H(x,y) - 1$. To decide which occurs when, we show that the condition $M(x,y) \geq degree_G(y).D_H(x,y)$ is true when $D_G(x,y)$ is even and its opposite is true when $D_G(x,y)$ is odd.
***Case 1***: Consider the case when $D_G(x,y)$ is even, say 2m. Then $D_H(x,y)$ is m from (b). Now for a neighbour of y in G, $D_G(x,y) - 1 \leq D_G(x,k) \leq D_G(x,y) + 1$. From this condition and result from (b) we can say, $\lceil D_G(x,k)/2 \rceil \geq m \implies D_H(x,k) \geq D_H(x,y)$.
$\sum_{k \in n_G(y)} D_H(x,k) \geq \sum_{k \in n_G(y)} D_H(x,y)$ ,
$\sum_{k \in n_G(y)} D_H(x,k) A_G(k,y) \geq degree_G(y).D_H(x,y)$.
$\sum D_H(x,k) A_G(k,y) \geq degree_G(y).D_H(x,y)$.
$M(x,y) \geq degree_G(y).D_H(x,y)$

***Case 2***: Now take the case when $D_G(x,y)$ is odd, say 2m-1. Again for a neighbour of y, $D_G(x,y) - 1 \leq D_G(x,k) \leq D_G(x,y) + 1$. $D_H(x,k) \leq m = D_H(x,y)$. Also, let k' be the neighbour of y that occurs just before y in the path from x to y. For k', $D_G(x,k') = 2m - 2$, thus $D_H(x,k') = m - 1$. Therefore, there exists at least one such neighbour of y k' for which

$D_H(x,k') < D_H(x,y)$ and $D_H(x,k') \leq D_H(x,y)$ for the rest. Summing over similar to case 1, $\sum D_H(x,k)A_G(k,y) < degree_G(y).D_H(x,y)$.
$M(x,y) < degree_G(y).D_H(x,y)$

### 3.4 (d)

The matrix M = $D_H * A_G$ can be computed using matrix multiplication in $O(n^w)$ time. The degree of each vertex can be computed and stored in an array from the adjacency matrix $A_G$ in $O(n^2)$ time. Using the conditions shown in (c) and $D_H$, $D_G$ can be found in $O(n^2)$ more time as $D_G(x,y) = 2D_H(x,y)$ when $M(x,y) \geq degree_G(y).D_H(x,y)$ and $2D_H(x,y) - 1$ when $M(x,y) < degree_G(y).D_H(x,y)$. Since $w > 2$, the total time complexity of the algorithm is $O(n^w)$.

### 3.5 (e)

Let d be the diameter of graph G. Consider Distance($A_G$) takes the adjacency matrix and returns the distance matrix $D_G$. $D_G$ is computed in the following way,

- $A_H$ is computed from $A_G$ in $O(n^w)$ time.
- $D_H$ is computed as a sub-problem from $A_H$, $D_H$ = Distance($A_H$). The diameter of the sub-problem is now reduced to $\lceil d/2 \rceil$. When the diameter reduces to 1, the squared graph becomes a complete graph with all entries $A_H(x,y)$ in adjacency matrix to be 1. For such a graph, the distance matrix will be the same as the adjacency matrix and this shall be our base case.
- $D_G$ is retrieved from $D_H$ as done in part (c) using the matrix M in $O(n^w)$ time.

The complexity of the algorithm can be written as T(n,d) = T(n,d/2) + $O(n^w)$, which is $O(n^w logd)$.In a n-vertex graph, the diameter can be at most n, hence the worst case complexity becomes $O(n^w logn)$.

# Universal Hashing

Given the universe U= [0, M-1] of M elements; a prime number $p$ in range [M, 2M], and n ($\ll$ M) be an integer. Considering the two hash functions given below:

H(x) := x mod n

$H_r$(x) := ( (rx) mod p) mod n

where r $\in$ [1, p-1]

## 4.1 (a) Prove that Probability(max-chain-length in hash table of S under hash-function H(·) > $log_2$n) ≤ 1/n

The set S is a randomly computed set with n elements, such that $\forall x \in S \implies x \in U$.

We have to find the probability that the maximum chain length in hash table of set S under the hash function H(.) is greater than $log_2$n.

Suppose the random variable $C_j$ denotes the chain length in the hash table when elements are mapped to $j$ ($\in$ [0, n-1]), under the hash function H(.). We need to find P($M_j > log_2$(n)), where $C$ = max($C_j$ $\forall$ j $\in$ [0, n-1].

$C_j$ can be alternatively written as, $C_j$=$\sum_{x\in S}$[1 if x%n=j, 0 else].
E[$C_j$]=$\sum_{x\in S}$E[x%n=j]=$\sum_{x\in S}$1/n (since, the x is chosen randomly, and the data uniformly hashes to each key, the expectation will be 1/n).

Consider the following,

P($C_j \geq log_2$(n))

$\implies$ $P(a\mathrm{C}_j \geq a log_2(\mathrm{n}))$, where a is some arbitrary positive constant

$\implies$ $P(2^{aC_j} \geq n^a)$

From the above expression obtained, we try to calculate the expectation of the same,

E[$2^{aC_j}$]

$\implies$ $E[2^{a\sum_{x\in S}[x\%n=j]}]$

$\implies$ $E[\Pi_{x\in S}2^{a[x\%n=j]}]$

$\implies$ $\Pi_{x\in S}E[2^{a[x\%n=j]}]$

$\implies$ $\Pi_{x\in S}(2^a/\mathrm{n})$

$\implies$ $(2^a/\mathrm{n})^n$ — eq(1)

Applying Markov Inequality on the obtained expression, we get,

P( $2^{aC_j} \geq n^a$) ≤ E[$2^{aC_j}$]/$n^a$

Put eq (1) in the above equation,
P( $2^{aC_j} \geq n^a$) ≤ E[$2^{aC_j}$]/$n^a$= $(2^a/\mathrm{n})^n/n^a$
Put a=2 in above equation,
P( $2^{aC_j} \geq n^a$) ≤ $(4/n)^n/n^2$
Since, $(4/n)^n$ is less than $(1+4/n)^n$, and $(1+4/n)^n$ is bounded from above. P( $2^{aC_j} \geq n^a$) ≤ $(4/n)^n/n^2 \leq 1/n^2$.
$\implies$ $P(\mathrm{C}_j \geq log_2(\mathrm{n})) \leq 1/n^2$.

Applying union bound, we get
P(C > $log_2$(n)) ≤ P(C ≥ $log_2$(n)) = P($\cup C_j \geq log_2$(n)) ≤ $\sum_{i=1}^{n}$ P($C_j \geq log_2$(n))
$\implies$ $P(C > \log_2(\mathrm{n})) \leq 1/\mathrm{n}$

Hence, proved.

### 4.2 (b) Prove that for any given r ∈ [1, p-1], there exists at least $C_n^{M/n}$ subsets of U of size n in which maximum chain length in hash-table corresponding to $H_r$(x) is $\theta$(n)

The hash function $H_r(.)$ maps the number ((rx) mod p, where r ∈ [1, p-1]) to the number obtained by applying modulo n to it, i.e., the hashed number belongs to the range [0, n-1]. As we know, that the universe U = [0, M-1], which contains total of M elements, and it is given that n ≪ M, thus, these M elements get almost uniformly mapped to a hash-table of size n.

By Pigeon Hole Principle, at least one element in the hashed table will have more than M/n elements. Thus, if we select any n elements out of these M/n elements which hash to the same element in hashed table, we will have the maximum chain length in hash table of order $\theta$(n).

Since, there is atleast one element in the hashed table with more than M/n elements, there exists at least $C_n^{M/n}$ subsets of U of size n with max-chain-length of order $\theta$(n). Hence, proved.

### 4.3 (c) Implementation of Hash Functions and Justification of Graph

The following graph was obtained for Hash functions H(.) and $H_r(.)$ for M= $10^4$ and n= 100, where $|S|$ = n. The set $S_k$ is a union of 0, n, 2n, 3n, ..., (k-1)n and n-k random elements from universe U. The value of k ranges from 1 to n. The value of p is chosen to be 12809, a prime in [M, 2M]. The set $S_k$ is different for both the hash functions, and a random different r is chosen for each k in $H_r(.)$.

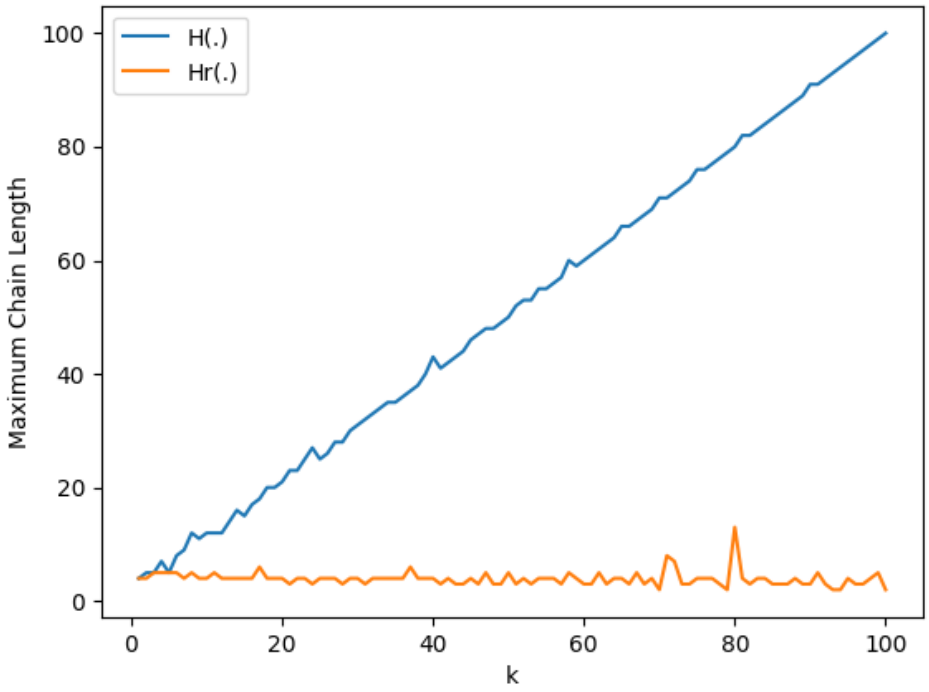


**Figure 1:** *Plot of Max-Chain-Length for Hash functions H(.) and $H_r(.)$*

***H*(.) Function:** The set $S_k$ contains 0, n, 2n, 3n, ..., (k-1)n, i.e., it has atleast k elements with same modulo (zero), for each k. Hence, for this hash function, the maximum chain length is atleast k for every choice of k. Thus, a linearly proportional graph is obtained. The randomly chosen n-k elements are almost equally distributed over the hash table and doesn't affect the nature of graph much.

$H_r$**(.) Function:** In this, the elements are first multiplied with a different random number, and their modulo is taken with a prime number. Thus, the numbers get distributed in a second universe and belong to [0, p-1]. Next, the modulo n ensures they are hashed between 0 to 99. Hence, the numbers are distributed more uniformly in the hash table as compared to the previous hash function, and thus, the maximum chain length doesn't vary or change much with the increasing size of set $S_k$.

The code used to generate the above graph is written below. We implemented both the hash functions in Python.

```
import matplotlib.pyplot as plt
import random

M=10000
n=100

x=[] #maintains the size of k to plot graph
graph_h=[]
graph_hr=[]

p=12809 #prime number in [M, 2M]

#implementation of H(.)
for k in range (1, n+1):
  x.append(k)
  S=[]
  for i in range (k):
    S.append(i*n)
  for j in range (n-k):
    S.append(random.randint(0, M-1))
  length_h={} #dictionaries that maintain the length in hash tables
  for element in S:
    hashed_element_h=element%n
    if hashed_element_h not in length_h:
      length_h[hashed_element_h]= 1
    else:
      length_h[hashed_element_h]+=1

  graph_h.append(max(list(length_h.values())))

#implementation of Hr(.)
for k in range (1, n+1):
  S=[]
  for i in range (k):
    S.append(i*n)
  for j in range (n-k):
    S.append(random.randint(0, M-1))
  length_hr={} #dictionaries that maintain the length in hash tables
  r=random.randint(1, p-1) #different random r for every set S
  for element in S:
    hashed_element_hr=((r*element)%p)%n
    if hashed_element_hr not in length_hr:
      length_hr[hashed_element_hr]= 1
    else:
      length_hr[hashed_element_hr]+=1

  graph_hr.append(max(list(length_hr.values())))

#plotting graph
plt.plot(x,graph_h,label="H(.)")
plt.plot(x,graph_hr, label="Hr(.)")
plt.xlabel("k")
plt.ylabel("Maximum Chain Length")
```

```
54 plt.legend()
55 plt.savefig("part4.png")
```