ASSIGNMENT COL351

Sourav(2019CS10404),Harikesh(2019CS10355)

October 2021

1 Problem

Convex Hull: Convex hull of a set of given points is a minimum subset of given points such that all given points can be generated by a convex combination of points in the selected subset.

OR

The points in the chosen subset are the corners of convex polygon of smallest area that encloses all points in the given set.

Algorithm intuition: We can clearly found out the convex hull by brute force by taking any segment by choosing any two points among the given points and then checking if that can be a segment or not. If not we try to join the two of segments in one to make it convex. But it is a brute force approach and will take $O(n^3)$ time. So, now we have to optimize our process of finding the convex hull and for that we have to see that we can first of all sort the data lets say according to x coordinate and then will try to divide the data into two halves (divide and Conqueror strategy) and for that we have to choose a good method which is having a low time complexity and always approximately divide the data into two parts. Now, we can use median of the dataset which is sorted according to x-coordinate (ties re broken by sorting y coordinate) and then find the convex hull of left data and right data and after that will try to merge the two of the convex hulls obtained to form a single hull by joining the lowest and highest common tangent points and finally we got what we need. Now, we have to basically find the base case of this recursion and which can be as small as 2 points, 3 points but we will have a problem if it is a dataset of just two points. So, we will always found out the convex hull of a dataset of size 3 by just brute force approach of taking segments and checking. Thus, after all the step of recursion we will get our convex hull of the whole dataset.

Input : P = {[x1,y1], [x2,y2], [x3,y3], ... [xn, yn]}

Expected Output: Q which is a subset of P and is our convex hull

Algorithm:

```
Def func convexhull( points[] P) do
    CustomSort(P) #according to xcoor if diff else according to y coordinate
    Return rec_hull(p)
End func
Def func rec_hull(points[] L) do
```

If $len(p) <= 3$ then # directly returning the set if no of points are this much small
Return L
End if
L1 = First $\lceil \frac{len}{2} \rceil$ points in L
L2 = L – L1
X_med = median(All x coordinates) # find this in O(n) time as told in lecture
P1 = rec_hull(L1)
P2 = rec_hull(L2)
Lst = Merging(P1,P2,X_med)
Return lst
End func
Def func Merging(P1, P2, X_med) do
List =[]
tan1, tan2 = Find_tangents(P1, P2,X_med) #here the tan1 and tan2 the set of points included in it
Add P1[tan1[1]] to list
First_ele = tan1[1]
Last_ele = tan1[len(tan1)]
Sec_Fir_ele = tan2[1]
Sec_last_ele = tan2[len(tan2)]
While (Last_ele not equals sec_last_ele of P2) do
Add P2[last_ele] to list
Last_ele = (last_ele + 1)mod (len(P2))
End while
While (sec_fir_ele not equals first_ele of P1) do
Add P1[sec_first_ele] to list
Sec_first_ele = (Sec_first_ele - 1)mod (len(P1))
End while
Return list
End func
Def func FindTangents(points[] p1, points[] p2, X_med) do
#here p1 and p2 are two convex hulls and X_med is the median of the x_coordinates
Lmost_point of p2 = p2[1]
Rmost_point of p1 = p1[len(p1)]
slope= (RMost_point[2] -Lmost_point[2])/ (RMost_point[1] -Lmost_point[1])
Line l =slope*x + c # Line with slope slope passing through p1 and p2
While(l crosses p1 or p2) do
While (l crosses p1) do
Rmost_point of p1 moves up
End while

While (l crosses p2) do
lMost_point of p2 moves up.

End while
End while
Tan1 = set (lmost_point, rmost_point)
Lmost_point2 of p2 = p2[1]
Rmost_point2 of p1 = p1[len(p1)]
While(l crosses p1 or p2) do
While (l crosses p1) do
Rmost_point2 of p1 moves down
End while

While (l crosses p2) do
lMost_point2 of p2 moves down
End while
End while
Tan2 = set(lmost_point2, rmost_point2)
Return tan1,tan2
End func

Note: here the meaning of the line crossing a polygon is that the line passes through any single edge of polygon which can be found out by checking that there exist some x_coordinate for which the y_coordinate of the corresponding points in polygon is more as well as less both. Also moving up and moving down pf points mean we chose another point with higher y_coordinate or lower y_coordinate.

Time Complexity: The Algorithm's time complexity is *O(nlogn)* because initially sorting takes time *O(nlogn)*. Now if we assume that time taken for the recursive function rechull is *T(n)* then we can easily see that

$$T(n) = T(n/2) + O(n)$$

because the set is divided into two parts and then the convex hull of two sets formed is merged using upper and lower tangent which takes *O(n)* time both, median calculation in each step also takes *O(n)* time and hence extra time is of *O(n)*. Now, this a very common recursion problem and we know that the time in this case comes out to be *O(nlogn)*. Hence, the overall time complexity of the procedure convexhull is *O(nlogn)*.

Correctness: We will start by proving that the given algorithm terminates which we can easily deduce by the fact that each time we are dividing the data set into two parts and then are joining the tangents of two convex hulls obtained above. Now, we know that on dividing the data each time into half it keeps on decreasing and when the size of the points is less than or equal to 3 we find the convex hull using brute force. So, the base case of the function rechull is when size of data provided is less than 3. So, this function will terminate there. After this is the process of merging the two data sets which can be done by finding the upper and lower tangents and then merging. This step is also going to terminate due to the fact that at last we will have our final set i.e. set of all given points in P and at that point we got our final convex hull and algorithm terminates.

The second part which we have to show is that this algorithm returns the convex

hull of the given set of points. The algorithm works by finding the convex hull of set of points with size less than 3 by brute force and it has to be convex hull After that we are finding the upper and lower tangent of left and right polygons and we know that convex hull is nothing but the union of the points of tangents and the edges in the two convex hulls apart from these tangents. Also, to show that it will be a convex hull we can give the argument that using less than 3 points we form a convex hull and then we are joining by using the tangents if above two convex hulls. We know that upper and lower tangents of two convex hulls can't form a concave polygon and hence the resulting polygon is convex too. We can also see that it will optimized set of points in convex hull which we are getting because if it is not and some extra point is there we would have removed that point while merging the two and hence our algorithm returns the convex hull of the set of points P.

2 Problem

In this problem we have to find the total force on each particle using Couloumb's law and this total force consists of the force due to all other particles. We can do this very easily in $O(n^2)$ time but that will take too much time and we will be probably out of time to calculate that force. Hence we need an optimized algorithm to find the total force on any single small charge due to all other charges in the surrounding and for this we will have to use Couloumb's law in an efficient manner to calculate it in time $O(nlogn)$.
Algorithm Sketch: We have to somehow manage to find the total force on a particle due to all other particles by formulating it as a problem of polynomial multiplication/convolution.
We are given that f_j

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

Which is the formula for finding the force on a single particle due to all other particles and it is giving force on a single particle due to all other particles. Now we are defining something like

$$\boxed{G_j = \frac{F_j}{Cq_j}} \tag{2}$$

and then using the above algorithm we can see that we can get the equation for G_j which will be like:

$$\boxed{G_j = \frac{F_j}{Cq_j} = \sum_{i<j} \frac{q_i}{(i-j)^2} - \sum_{j<i}^{i<=n} \frac{q_i}{(i-j)^2}} \tag{3}$$

Now to get to this kind of equation we have to define two polynomials which will be of the form lets say $p_0(x)$ and $p_1(x)$. Now here $p_0(x)$ can be assumed to

be of the form $\mathrm{p}_0(x) = \sum_{a=1}^{a=n} q_a x^a$ while $\mathrm{p}_1(x) = \sum_{b=1}^{b=2n-1} H_b x^b$ and this H_b is different for different set of values of b which varies as follows:

$$H_b = \begin{cases} \frac{1}{(n-b)^2} & \text{when b>n+1 and b<2n} \\ 0 & \text{when b = n} \\ \frac{-1}{(n-b)^2} & \text{when b>=1 and b<n} \end{cases}$$

Now we will have to just find our F_j in terms of multiplication of these two polynomials by somehow using the form G_j which can be easily seen to be deducted by this algorithm. We can see that our G_j is the coefficient of the $\mathrm{x}^{(n+j)}$ in the multiplication of two polynomials which are $p_0(x)$ and $p_1(x)$ that is $\sum_{a=1}^{a=n} q_a H_n + j - a$ where n is total number of particles. From this we can get our G_j's using Fast fourier transform multiplication and then we will get all the F_j's by just multiplying the corresponding F_j's by Cq_j (for the corresponding charge it will take *O(1)* time) and hence for all the charges it will take *O(n)* time.

Algorithm:

Def func findforce(n , int[] q) do
    # here n is total particles and q is charge array.
    we_going_to_obtain = $2^{(log2(2*n-1))}$
    Answer = []*$|we\_going\_to\_obtain|$
    $Poly\_1 = [0] * |we\_going\_to\_obtain|$
    $\#initializedthearrayswith0andsizewewillget$
    $Poly\_2 = [0] * |we\_going\_to\_obtain|$
    $For i < -1 to ndo$
        $Poly\_1[i] = q[i]$
    $Endfor$
    $For j < -1 to 2 * n - 1 do$
    $\#wearedirectlystoringourHbcoefficientshere$
        $If jislessthanorequalsn - 1do$
            $Poly\_2[j] = -1/(n-j)^2$
        End if
        Else if j equals n do
            Poly_2[j] = 0
        End else if
        Else do
            Poly_2[j] = 1/(n-j)^2
        $End\ else$
    $End\ for$
    $Fourier\_1 = Discrete\_Fourier\_Transform(Poly\_1)$
    $\#taught\ in\ lecture$
    $Fourier\_2 = Discrete\_Fourier\_Transform(Poly\_2)$
    $\#basically\ mult\ with\ Vandermore\ matrix$
    $For i < -1 to we\_going\_to\_obtain do$
        $Answer[i] = Fourier\_1[i] * Fourier\_2[i]$
    $End\ for$

$Final\_answer = Inverse\_Fourier\_Transform(Answer)$
$\#pre\ mult\ by\ inverse\ vandermore\ mat$
$\#now\ we\ have\ to\ find\ all\ the\ forces\ in\ O(n)\ by\ multiplying\ with\ C*$ *corresponding charge*
$Total\_Forses = []$
$For j < -1 to do$
$Add(Final\_Answer[i] * C*\mathrm{q}_j)$ to Total_Forses
End for
Return Total_Forses
this will give us force on a particular element due to others
End func

Note: we have not written the algorithm for fourier transform and inverse fourier transform as those are already given in lecture and are pretty basic. So, we have left that part. Anyways this function will return us the forces on each of our particles.

Time Complexity: The time taken by our algorithm is *O(nlogn)* because as we can see that initially we are finding our G_j's which are just founded by Discrete fourier transformation of the two arrays we want to convolve and after that converting the result back by applying inverse discrete fourier transform. This process of polynomial multiplication using fourier transform takes *O(nlogn)* time and hence we obtain our G_j's array containing all the G_j's for all particles. Now, finally we have to found the total forces on the particles for which we will be going to multiply each of the elements in the array with corresponding $C * q_i$ values and this again will take *O(n)* time. Hence, our upper bound of time complexity will be *O(nlogn)*.

Correctness: We will start by proving that the above algorithm terminates. As we are calculating the constants G_j's using polynomial multiplication using our Fourier transform convolution method and the first two for loops in our program will be running in range n and 2n respectively and hence they will terminate after completing n iteration. After that we are using two more for loops basically for fourier transformed multiplication of both polynomials and then one last for loop for just iterating through the array obtained and multiplying each one of with the value $C * q_i$. So, these will also going to terminate after *O(n)*, Fourier transform basically terminates as it is a standard algorithm and will take *O(nlogn)* time for multiplying two polynomials.

For correctness we can just conclude it by induction and see the similarties here and what we got during polynomial multiplication. We can say that the base base of having a single charge at a point always holds as there is only one charge. Now lets say this algorithm gives correct output for the case of having n-1 charges and we add a charge a nth place. Now we can see that on adding the charge at nth place we will get its influence on all the other charges and all the other charges influence it too and we can find that influence by adding that charge is nothing but addition of a new term to each of the functions which is just the convolution like $(q_a * H_n + j - a)$ and this will be the same case if we add a new term to one of the polynomials or both for multiplying. This, concludes that our algorithm is correct and has a time complexity of *O(nlogn)*.

3 Problem

3.1 Part A:

G(V,E) is given unweighted undirected graph and H(V,EH) is the undirected graph obtained from G that satisfy: (x,y) belong to EH only If (x,y) belongs to E or there exist some node w in G such that we have (x,w) belongs to E and (w,y) belong to E too. Also, we have DG is distance matrix of G and DH is the distance matrix of H.
(a) To prove that graph H (V,EH) can be computed from G in O(n^w) time, where w is exponent of matrix multiplication.
Lets take a n*n matrix (n is number of vertices) whose entries are m_{ij} where m_{ij}=1 when there is an edge between i and j, all other entries are zero. Now to compute the edges of the graph H we have to see that

$$\boxed{sum = \sum_{k=1}^{k=n} m_{ik} * m_{kj}} \tag{4}$$

So, from the value of k we can get whether there is a vertex in our Graph G such that both edges {I,k} and {k,j} are present in its vertex set. We can prove that sum will be 0 when there is no such vertex and vice versa.
Proof: We can try to prove it by contradiction by saying that assume there exists a vertex let's say p such that edges {i,p} and {p,j} exists in the graph G but the value of sum is zero. But this contradicts our basic assumption of matrix because if both m_{ip} =1 and m_{pj} =1 then the sum should be atleast 1 and can never be zero. So, this contradicts the fact that sum can be zero if there exists such a vertex.
Hence now, For finding the whole graph H we already know the vertices and have to find the edges which can be found out by calculating the matrix h where h_{ij} =1 if such vertex exists In original graph else h_{ij} =0.
Entries of h i.e.

$$\boxed{h_{ij} = \sum_{k=1}^{k=n} m_{ik} * m_{kj}} \tag{5}$$

So, there is a edge {I, j} in graph H only when h_{ij} = 1 else no such edge is there. This way we can calculate our graph just by using matrix multiplication of two n*n matrices. Hence, the time complexity of this process is O(n^w).

3.2 Part B:

We have to argue that for any x,y belonging to V, We have to show that

$$\boxed{DH(x,y) = \lceil \frac{DG(x,y)}{2} \rceil} \tag{6}$$

Proof: Lets take any two vertices x and y in the graph G. Let's assume the shortest path between these two vertices in the graph G is like x, g1, g2, ... gl-1, y which is of length l. Now, we know that all the edges of G are also present in the graph H as we know graph H is a supergraph of G.

Now we also know that if there is a vertex in G such that both {ik} and {kj} are there in the edges then we can have another edge in {ij} in our graph for such a vertex k. Hence, we can construct a path of length $\frac{l}{2}$ in H by taking every alternate vertex in the shortest path from x to y in G. This wat we will be having

$$\boxed{DH(x,y) <= \lceil \frac{DG(x,y)}{2} \rceil} \tag{7}$$

we will take ceiling because DH(x,y) will be an integer.
Again, let's assume the shortest path between x and y in graph H is l'. Now we know that for any edge {x,y} in H we can have either x,y in our graph G or for some vertex z, we have both edges {x,z} and {z,y} in our graph G. So, this gives l<=2*l' which in terms of matrix gives:

$$\boxed{\frac{DG(x,y)}{2} <= DH(x,y)} \tag{8}$$

and also DH(x,y) has to be a integer. So, for that we can say

$$\boxed{\lceil \frac{DG(x,y)}{2} \rceil <= DH(x,y)} \tag{9}$$

Hence from both of these conditions we can say that

$$\boxed{DH(x,y) = \lceil \frac{DG(x,y)}{2} \rceil} \tag{10}$$

3.3 Part C:

Now let's assume that AG is adjacency matrix of G, and matrix M = DH*AG. Now, we have to show that for any x,y belonging to V, the following holds:

$$DG(x,y) = \begin{cases} 2 * DH(x,y) & \text{when M(x,y) >=DegreeG(y)*DH(x,y)} \\ 2 * DH(x,y) - 1 & \text{when M(x,y) <=DegreeG(y)*DH(x,y)} \end{cases}$$

From part 2 we have already shown that there are only two possible values of DG(x,y) which are 2DH(x,y) and 2DH(x,y) -1. Hence, DG(x,y) is even only when it is equal to 2DH(x, y) and odd when it is equal to 2DH(x,y)-1.
Now lets simplify the condition i.e. M(x,y)-DegreeG(y)*DH(x,y) a little bit. We know that we have calculated the m by using the formula:

$$\boxed{M(i,j) = \sum_{allz} DH(x,z) * AG(z,y)}$$

where z Is the set of all vertices.
Now we know that in adjacency matrix representation we have only two values

which are 0 and 1 and for all vertices which are not neighbours of y i.e. let p , AG(p, y) = 0 which then reduces our expression to the form

$$\boxed{M(i,j) = \sum_{z=adj(y)} DH(x,z)}$$

for all z adjacent to y. Now subtracting DegreeG(y)*DH(x,y) from it gives us

$$\boxed{M(I,j) - DegreeG(y) * DH(x,y) = \sum_{z=adj(y)} (DH(x,z) - DH(x,y))}$$

all z adjacent to y in G. Now if DG(x,y) = 2a i.e. is even then DG(x,z)>=2a-1 for all neighbours z of y. On the other hand if DG(x,y) = 2a-1 i.e. is odd then DG(x,z)<2a for some of the neighbours of y and DG(I,k)<=2a for all the remaining neighbours.
Now from the part b we can say that

$$\boxed{DH(x,z) = \lceil DG(x,z) \rceil} \tag{14}$$

and which implies that DH(x,z)>=a which from above statement implies DH(x,z)>=DH(x,y) when our DG(x,y) is even and DH(x,z)<DH(x,y) when DG(x,y) is odd. This proves our result which we wanted to prove that is

$$DG(x,y) = \begin{cases} 2 * DH(x,y) & \text{when M(x,y) >=DegreeG(y)*DH(x,y)} \\ 2 * DH(x,y) - 1 & \text{when M(x,y) <=DegreeG(y)*DH(x,y)} \end{cases}$$

3.4 Part D:

Now, From (c) we have to argue that DG can be computed from DH in O(n^w) time.
From part c we can say that it is enough to compute the matrix M and have a look at all the entries of M. We have to compare all the entries of M with the value of DegreeG(y) *DH(x,y). So, for comparison we have to iterate the matrix M which will take O(n^2) and for calculating M initially using DH*AG took O(n^w) time (By matrix multiplication). Hence, total time taken to compute DG from DH is O(n^w).

3.5 Part E:

All pair distances in n vertex unweighted undirected graph can be computed in O($n^w logn$) time if w is larger than 2.
For calculating all pair distances in n vertex unweighted undirected graph we have to first calculate the distance matrix for the half of the matrix part which can be said by saying that

$$\boxed{DH(x,y) = \lceil \frac{DG(x,y)}{2} \rceil} \tag{15}$$

and hence the time recurrence relation for that can be expressed in the form

$$\boxed{T(n,d) <= T(n,d/2) + O(n^w)} \tag{16}$$

where n is number of vertices and d is the diameter.
Which then reduces to give the running time complexity as O($n^w logd$), and we know that the graph diameter is less than the total number of nodes. i.e. n>=d then it's time complexity will be O($n^w logn$). Where the n^w part comes due to matrix multiplication thing and logn comes due to a decrease by a constant factor.

4 Problem

4.1 Part A:

We have a random set initially and then we are choosing any random number in U and then are adding it to our set S.
To Prove: P(max chain length in map H()> $log_2 n$) <=1/n.
Proof: We know that if X is a non negative random variable p(x>= a) <= $\frac{E(x)}{a}$ using Markov's inequality Using this we can say that P(max chain in map H()> $log_2 n$) <= $\frac{E(x)}{log_2 n}$.
And to know the expectation of E(x) we can see that we will be getting all the values between 0 to n randomly as in H finally we are taking the modulus of n. We can also assume that the value of n is very very large and Hence, the log2n can be taken as a constant and hence Markov's inequality will almost hold true. Now, to find the expectation of the theorem we can say use the mapping we are doing there.
We can say that we are choosing a random number from the universal set and then are just adding it to the set and therefore,it is equally probable that the number after been taken the modulo n will lie in o to n-1 randomly and will have an probability of $\frac{1}{n}$. Which then gives P(required) <= $\frac{1}{log_2 n * n}$ and this is less than $\frac{1}{n}$as $log_2 n$ >0 we have n>>0.

4.2 Part B:

We have to prove that for any given r belonging to [1,p-1], there exists atleast $\binom{M/n}{n}$ subsets of U of size n in which maximum chain length in hash table corresponding to Hr(x) is $\Theta(n)$.
Lets say we are looking for value of r = r'. Let there is a set S in the universal set U that contains n elements and has maximum chain length as l. This implies that l of the elements in set S are having same mapped key in Hr(x) map and let it be k.
So, the equation

$$\boxed{(r'x \bmod p) \bmod n = k} \tag{17}$$

which can be transformed to

$$\boxed{r'x \ mod \ p = k + an} \tag{18}$$

where a is a constant, which can further be transformed to:

$$\boxed{x \ \textbf{mod} \ p = \text{r'}^{(p-2)} \ (\text{an+k}) \ \textbf{mod} \quad \text{p}} \tag{19}$$

Here, p is in range [M,2M] and k+an is between 0 to p-1, x < M and p>=M hence x mod p reduces to x. Now, we can say that l >=a+1 as the maximum value of a varies in 0 to l-1. So, $l = 1 + \frac{p-1-k}{n}$. From here we can say that there are atleast $\frac{M}{n}$ choices of such elements S for the maximum length to be $\Theta(n)$. And we have to form a set of n elements therefore there exists atleast $\binom{M/n}{n}$ choices of such subsets in the universal set S.

4.3 Part C:

```java
import java.util.*;
public class HelloWorld{
    int M = 10000; int setsize =100;
    public static void main(String []args){
        Scanner sc = new Scanner(System.in);
        HashSet<Integer> hash = new HashSet<Integer>();
        int[] arr = new int[100];
        int[] arr2 = new int[100];
        int maxh = Integer.MIN_VALUE;
        int maxhr = Integer.MIN_VALUE;
        int k = sc.nextInt();
        for (int i =0; i<k; i++){
            hash.add(i*100);
        }
        for (int j =0; j<100-k; j++){
            Random random = new Random();
            int adder = random.nextInt(10000);
            hash.add(adder);
        }
        for (int i : hash){
            arr[i%100]++;
            maxh = Math.max(arr[i%100], maxh);
        }
        for (int i : hash){
            Random random = new Random();
            int p = random.nextInt(10000) + 10000;
            Random rand = new Random();
            int r = rand.nextInt(p-1) + 1;
            arr2[((i*r)%p)%100]++;
            maxhr = Math.max(maxhr,arr2[((i*r)%p)%100]);


        }
        System.out.print(k + " " + maxh + " " + maxhr);
    }
}
```

import java.util.*;
public class HelloWorld

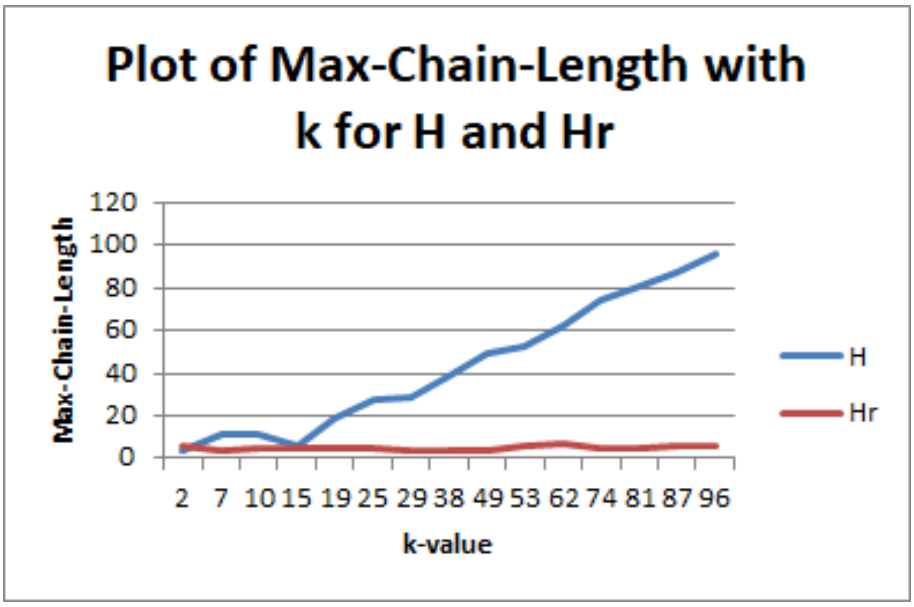
```
int M = 10000; int setsize =100;
public static void main(String []args)
    Scanner sc = new Scanner(System.in);
    HashSet¡Integer¿ hash = new HashSet¡Integer¿();
    int[] arr = new int[100];
    int[] arr2 = new int[100];
    int maxh = Integer.MIN_VALUE;
    int maxhr = Integer.MIN_VALUE;
    int k = sc.nextInt();
    for (int i =0; i<k; i++)
        hash.add(i*100);

    for (int j =0; j<100-k; j++)
        Random random = new Random();
        int adder = random.nextInt(10000);
        hash.add(adder);

    for (int i : hash)
        arr[i%100]++;
        maxh = Math.max(arr[i%100], maxh);

    for (int i : hash)
        Random random = new Random();
        int p = random.nextInt(10000) + 10000;
        Random rand = new Random();
        int r = rand.nextInt(p-1) + 1;
        arr2[((i*r)%p)%100]++;
        maxhr = Math.max(maxhr,arr2[((i*r)%p)%100]);


    System.out.print(k + " " + maxh + " " + maxhr);
```



From the above graph we can see that the maximum chain length value for function H() increase almost linearly with time this is because we are only choosing n-k numbers randomly from the universal set and the remaining k numbers are always having modulo 0 with respect to n which we are doing in our function H(). Regarding other n-k numbers we can have remainder contained from them in range of 0 to n-1 which is 99 in our case. Hence, on an average the mapped key of these n-1 numbers is $\frac{n-k}{n}$ and hence the maximum length which we are going to obtain on average is n + $\frac{n-k}{n}$ and the slight fluctuation from linearity in the graph is due to $\frac{n-k}{n}$ part. The max chain length finally when our k is equal to n i.e. 100 is n i.e. 100 as all the elements will be mapped to 0. On the other hand in case of universal map Hr(), here first of all we are calculating rx%p and as r is generated randomly between 1 and p-1 and then it will be multiplied with the elements in the set we have. Also to note that p is also generated randomly. So, the maximum length which we were getting previously due to the modulo n being 0 is not possible here as we are initially multiplying r*x which will range to our entire universal set and will give a random number in it. After taking modulo with p we will get any random number in 0 to p-1. And then n is very less and hence finally on taking modulo also gives us a random number in 0 to n-1 as p>>n. Hence, this mapping is taking all the numbers from 0 to n-1 by equal weightage and hence maximum chain length nearly remains constant and 1 most of the time (if total randomness).