COL351-A3

Pratyush Saini (2019CS10444)
Prakhar Aggarwal (2019CS50441)

24th October 2021

1 Convex Hull

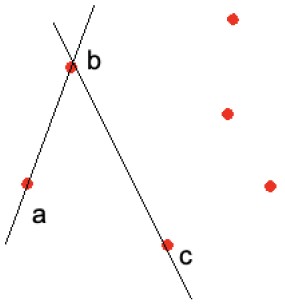
Given n points in plane

$$S = \{(x_i, y_i) | i \in \{1, 2, 3, ..., n\}\} \tag{1}$$

Assumption: We assume that no two points have same x coordinate or same y coordinate and no three points are collinear

Convex Hull CH(S) is the smallest polygon containing S. The boundary of convex hull is a simple closed curve with minimum perimeter containing S

Lemma 1: Let P_i denote the i^{th} point in S. Then the line segment P_iP_j is an edge of the convex hull if all the points in $S \backslash \{P_i, P_j\}$ lie on either side of line P_iP_j



(a,b) is an edge of CH(S) while (b,c) is not

Proof: The proof of the lemma comes from the fact than any edge of the Convex polygon never passes through the polygon.

Since all the points lie on either side of the line and no three points are collinear, we can claim the line does not pass through the interior of polygon and hence, forms it's edge.

1.1 Brute Force Algorithm

We can determine all the edges of CH(S) by using the above lemma. To say, We test each line segment if it makes up an edge of the convex hull.

- If the rest of the points are on one side of the segment, the segment is on the convex hull.
- else the segment is not.

Algorithm 1 Brute Convex Hull

```
S ← {(x_i, y_i)|i = 1, 2, ...n}
CH ← {}                                   ▷ Set containing vertices of Convex Hull
for P_i ∈ S do
    for P_j ∈ S\P_i do
        if ∀P_k, P_k lie on same side of line P_i, P_j then
            CH ← CH ∪ {P_i, P_j}
        end if
    end for
end for
return Sorted_Clockwise(CH)               ▷ Sorted vertices of Convex Hull
```

1.1.1 Time and Space Complexity

Total $O(n^2)$ edges, Testing time for each edge $O(n)$ (Since, we have to iterate over all points to check if they lie on same side of line), Hence overall time complexity is $O(n^3)$
Space required = $O(n)$ to store the vertices of Convex Hull

1.2 Divide and Conquer Algorithm

We initially perform a one time sort of S on the basis of x-coordinate. For input set of points,

- Divide S into two halves A and B such that $\forall P \in A$, x(P) $\leq$ Median(S[x]) and $\forall P \in B$, x(P) $>$ Median(S[x])
- Compute CH(A) and CH(B) recursively
- Merge CH(A) and CH(B) to compute CH(S)

Base Cases: It is easy to show that the Convex Hull for 3 or fewer points is the complete Set S itself. But in certain cases if the merge step creates issue for very few number of points, we can use brute force computation of CH whenever Size of S is less than 5

1.2.1 Algorithm Pseudo Code

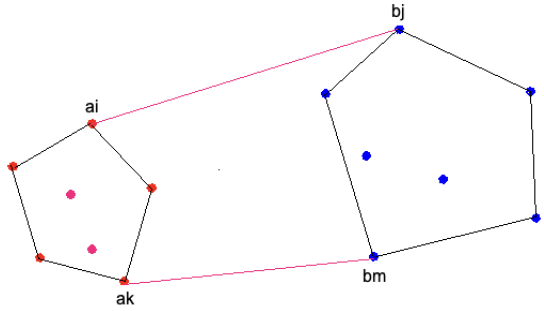
Algorithm 2 Convex Hull (S)

$S \leftarrow \{(x_i, y_i) | i = 1, 2, ...n\}$
$CH \leftarrow \{\}$ ▷ Set containing vertices of Convex Hull
m ← Median(S[x]) ▷ S is sorted, Takes O(1) time
$A, B \leftarrow \{\}, \{\}$
for $(x_i, y_i) \in S$ **do**
  if $x_i \leq m$ **then**
    $A \leftarrow A \cup \{(x_i, y_i)\}$
  else
    $B \leftarrow B \cup \{(x_i, y_i)\}$
  end if
end for
return $Merge(ConvexHull(A), ConvexHull(B))$

Now the problem is, how to merge CH(A) and CH(B) efficiently. We devise two algorithms for the Merge operation

1.2.2 Merge Step

To perform the merge operation on CH(A) and CH(B), we need to find Upper Tangent (a_i, b_j) and Lower Tangent (a_k, b_m) corresponding to left and right Convex Hulls.



With those tangents the convex hull of $A \cup B$ can be computed from the convex hulls of A and the convex hull of B in polynomial time.

Suppose:

$$CH(A) = \{a_1, a_2, a_3, ..., a_{|CH(A)|}\}$$
$$CH(B) = \{b_1, b_2, b_3, ..., b_{|CH(B)|}\}$$

First link a_i to b_j , go down b clockwise till we see b_m and link b_m to a_k, continue along the a anticlockwise until we return to a_i

We'll maintain sorted order of $CH(A)$ and $CH(B)$ in clockwise direction

1.2.3 Finding Tangents

Suppose:

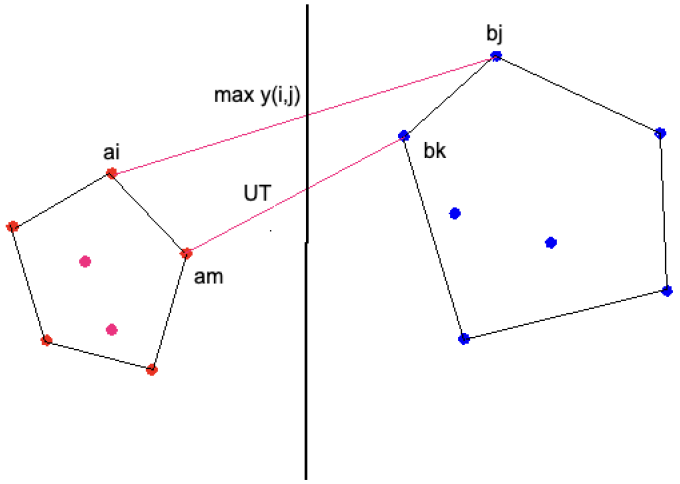
$$CH(A) = \{a_1, a_2, a_3, ..., a_{|CH(A)|}\}$$
$$CH(B) = \{b_1, b_2, b_3, ..., b_{|CH(B)|}\}$$

Let a_i be the point that maximizes x within $CH(A)$ and b_i which minimizes x within CH(B).
Suppose L is the vertical line separating points in CH(A) and CH(B), (can be taken as L = $median_x(CH(A) \cup CH(B))$).

Define, **y(i,j)** as the coordinate of intersection of Segment (a_i, b_j) with L.

Claim: (a_i, b_j) is the Upper Tangent if it maximizes y(i,j) and Lower Tangent if it minimizes y(i,j) for all $(a_i, b_j) \in CH(A) \times CH(B)$

Proof of claim:
We proceed ahead using Proof by Contradiction.
Suppose that (a_i, b_j) be the pair corresponding to **max y(i,j)** among all possible pairs and L = (a_m, b_k) be the Upper Tangent such that the ordered pair $(a_i, b_j) \neq (a_m, b_k)$



This boils down to the implication that either of a_i or b_i lies on opposite side of line UT, relative to other points. But using **lemma 1**, we know that all the points lie on either side of any edge of Convex polygon. Since UT is a part of polygon, Lemma 1 gets contradicted and therefore, our assumption is false.

Therefore, the ordered pair $(a_i, b_j) = (a_m, b_k)$
Similarly, we can show that (a_i, b_j) is lower tangent if it minimizes y(i,j) for all possible pairs.

1.2.4 Algorithm : Finding Tangents

Brute Force Approach

Algorithm 3 Finding Tangents (CH(A), CH(B)

```
UT ← {}
LT ← {}
maxY ← INT_MIN
minY ← INT_MAX
for (a_i) ∈ CH(A) do
    for (b_j) ∈ CH(B) do
        if y(i,j) > maxY then
            maxY = y(i,j)
            UT ← (a_i, b_j)
        end if
        if y(i,j) < minY then
            minY = y(i, j)
            LT ← (a_i, b_j)
        end if
    end for
end for
return LT, RT
```

Space Complexity: $O(1)$ to store UT and LT
Time Complexity: $O(n^2)$ to iterate over all pairs of points (a_i, b_j)

Efficient Algorithm: We have CH(A) and CH(B) as our input, which contains points sorted in clockwise order.

Algorithm 4 Upper Tangent (CH(A), CH(B))

```
i ← Index of rightmost point of A                          ▷ O(n) time
j ← Index of leftmost point of B                           ▷ O(n) time
p ← size(CH(A))
q ← size(CH(B))
while y(i, j + 1 mod q) > y(i, j) or y(i − 1 mod p, j) > y(i, j) do
    if y(i, j + 1 modq) > y(i, j) then
        j ← j + 1 mod q                                    ▷ Move Clockwise
    else if y(i − 1 mod p, j) > y(i, j) then
        i ← i − 1 mod p                                    ▷ Move Anti-Clockwise
    end if
end while
return (a_i, b_j)                                          ▷ Upper Tangent
```

Time Complexity: j can be incremented at most q times and i can be decremented at most p times. So Time complexity = $O(n)$.

Algorithm 5 Lower Tangent (CH(A), CH(B))

i ← Index of rightmost point of A ▷ O(n) time
j ← Index of leftmost point of B ▷ O(n) time
(p,q) ← $(size(CH(A)), size(CH(B)))$
while $y(i, j-1 \ mod \ q) < y(i,j)$ or $y(i+1 \ mod \ p, j) < y(i,j)$ **do**
  if $y(i, j-1 \ mod \ q) < y(i,j)$ **then**
    $j \leftarrow j-1 \ mod \ q$ ▷ Move Anti-Clockwise
  else if $y(i+1 \ mod \ p, j) < y(i,j)$ **then**
    $i \leftarrow i+1 \ mod \ p$ ▷ Move Clockwise
  end if
end while
return (a_i, b_j) ▷ Lower Tangent

Proof of Above Finding Tangent Algorithm

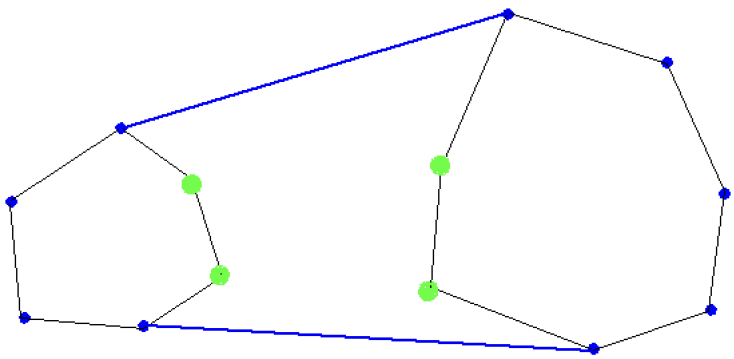
- a_i, b_j are right most and left most points in $CH(A)$ and $CH(B)$. We move anti clockwise from a_i, clockwise from b_j.
- CH(A) = $a_1, a_2, ..., a_p$ (where p = $\|CH(A)\|$) is a convex hull, as is CH(B) = $b_1, b_2, ..., b_q$ (where q = $\|CH(B)\|$).
- If a_i, b_j is such that moving from either a_i or b_j decreases y(i, j) there are no points above the (a_i, b_j) line, hence we break the while loop as we got the points with max y(i,j). Similarly, we can prove for Lower Tangent.

1.2.5 Merge Algorithm:

Algorithm 6 Merge (CH(A), CH(B))

$(a_i, b_j) \leftarrow UpperTangent(CH(A), CH(B))$
$(a_j, b_m) \leftarrow LowerTangent(CH(A), CH(B))$
$CH \leftarrow \{a_i, b_j\}$
$r \leftarrow j+1$
while $r \neq m$ **do** ▷ Move Clockwise till we see b_m
  $CH.append(b_r)$
  $r \leftarrow r+1$
end while
CH.append($\{b_m, a_k\}$)
$r \leftarrow k-1$
while $r \neq i$ **do** ▷ Move Anti-Clockwise till we see a_i
  $CH.append(a_r)$
  $r \leftarrow r-1$
end while
return CH ▷ Sorted in Clockwise order

1.2.6 Proof of Merge Operation:



Let's color the vertices as blue and green as shown in the figure, where the blue vertices represents the points taken it clockwise direction from top incidence of Tangent to the bottom with respect to A and anti-clockwise direction with respect to B. Colour the remaining vertices as green.
Now, all lines g having any green point as it's end would not satisfy Lemma 1, hence the green vertices can never be the part of convex hull. Further we know that the end points of both the tangents satisfy Lemma 1 (which was the basis of Finding Tangent Algorithm), hence they form a part of Convex Hull.
Now we are left with remaining blue vertices. Consider all the line segments between adjacent pair of blue points. If any of them is extended, they would not cause partitioning points of S into either of it's sides [Lemma 1] (since if they do so, they would have been considered as candidates for tangents instead of the current ones). So, all the remaining blue points satisfy Lemma 1, hence form the part of Convex Hull.

1.2.7 Time Complexity

- **Pre-processing:** One time sort of points in S by x-coordinate $O(nlogn)$
- Partitioning S into A and B based on Median $O(n)$
- Recursively computing Convex Hull (A) and Convex Hull (B) $T(n/2)$
- Merge Operation : $O(n)$

The Recurrence follows:

$$\mathrm{T(n)} = 2\ \mathrm{T(n/2)} + \mathrm{O(n)}$$

From Master's theorem, we have $\mathrm{T(n)} = O(nlogn)$, so final time complexity is $O(nlogn + nlogn) = O(nlogn)$

2 Particle Interaction

The total net force on particle j according to Coulomb's law is given by:

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

Taking inspiration from polynomial multiplication algorithm, we can devise an $O(nlogn)$ algorithm to compute net force acting on all particles.
We can achieve our goal if we find two polynomial such that coefficient of their product polynomial could give us required quantities.

2.1 Formulation of Problem as product of Polynomials

First, consider a polynomial $P(x)$ given by,

$$P(x) = q_n x^n + q_{n-1} x^{n-1} + ... + q_1 x \tag{3}$$

Secondly, a natural intuition inspires us to take the second polynomial as:

$$A(x) = \frac{x^n}{n^2} + \frac{x^{n-1}}{(n-1)^2} + ... + \frac{x^1}{1^2} \tag{4}$$

Now, consider the product polynomial $P(x)A(x)$. coefficient of x^k is given by,

$$[P(x)A(x)]_k = \frac{q_1}{(k-1)^2} + \frac{q_2}{(k-2)^2} + ... + \frac{q_{k-1}}{1^2} \tag{5}$$

or equivalently,

$$[P(x)A(x)]_k = \sum_{i<k} \frac{q_i}{(i-k)^2} \tag{6}$$

Similarly, Take the second polynomial B(x) as,

$$B(x) = \frac{x^{-n}}{n^2} + \frac{x^{-(n-1)}}{(n-1)^2} + ... + \frac{x^{-1}}{1^2} \tag{7}$$

The coefficient of x^k in the product polynomial $P(x)B(x)$ is given by,

$$[P(x)A(x)]_k = \frac{q_{k+1}}{1^2} + \frac{q_{k+2}}{2^2} + ... + \frac{q_{k+(n-k)}}{(n-k)^2} \tag{8}$$

or equivalently,

$$[P(x)B(x)]_k = \sum_{i>k} \frac{q_i}{(i-k)^2} \tag{9}$$

Combining (6) and (9), we get

$$[P(x)(A(x) - B(x))]_k = \sum_{i<j} \frac{q_i}{(i-j)^2} - \sum_{i>j} \frac{q_i}{(j-i)^2} \tag{10}$$

which, we can multiply by Cq_k to obtain F_k

As we see, A(x) - B(x) is not a proper polynomial since it contains negative powers of x as well. So, to construct a proper polynomial, we can multiple x^n with A(x) - B(x) and then, take the power of x^{n+k} instead of x^k.

Let R(x) = $x^n(A(x) - B(x))$ be our required polynomial.
Consider two vectors,

$$\mathbf{Q} = \left\langle \frac{1}{n^2}, \frac{1}{(n-1)^2}, ..., \frac{1}{4}, 1, 0, 1, \frac{-1}{4}, ..., \frac{-1}{(n-1)^2}, \frac{-1}{n^2} \right\rangle \tag{11}$$

$$\mathbf{X} = \{x^{2n}, x^{2n-1}, ..., x^2, x^1, x^0\} \tag{12}$$

Then the polynomial R(x) is given by $\mathbf{Q}.\mathbf{X}^T$. Therefore, we have

$$[P(x)R(x)]_{n+j} = \sum_{i<j} \frac{q_i}{(i-j)^2} - \sum_{i>j} \frac{q_i}{(j-i)^2} \tag{13}$$

Hence, we have **Proved** that the Force on particle j is given in terms of coefficient of x^{n+j} in polynomial P(x)R(x),

$$F_j = Cq_j \left([P(x)R(x)]_{n+j}\right) \tag{14}$$

2.2 Algorithm Sketch

We can use Polynomial Multiplication algorithm using DFT and Inverse DFT discussed in class to evaluate the product polynomial in O(nlogn) time.
We can take the two polynomials to be P(x) and R(x) with coefficients:

$$P(x) = \{q_{2n}, q_{2n-1}, ..., q_1, q_0\} \tag{15}$$

where $q_{2n} = q_{2n-1} = ... = q_{n+1} = q_0 = 1$ and rest value of $q_i \forall i \in \{1, 2, ..., n\}$ are input charge values

$$R(x) = \{\frac{1}{n^2}, \frac{1}{(n-1)^2}, ..., \frac{1}{4}, 1, 0, 1, \frac{-1}{4}, ..., \frac{-1}{(n-1)^2}, \frac{-1}{n^2}\} \tag{16}$$

Using the steps mentioned in class,

- Step1 (DFT): Evaluation of P(x) and R(x) at $\{1, \omega, \omega^2, ..., \omega^{2n}\}$: $O(2nlog2n)$
- Step2: Calculating value of C(x) = P(x)R(x) at $\{1, \omega, \omega^2, ..., \omega^{2n}\}$: $O(n)$
- Step3: Using Inverse DFT to get Polynomial C(x) : $O(2nlog2n)$
- Step4: Evaluating $F_j : \forall i \in \{1, 2, ..., n\}$ $F_j = Cq_j[C(x)]_{n+j}$: $O(n)$

2.3 Time and Space Complexity

Total time complexity for all steps = $O(nlogn + nlogn + n + n) = O(nlogn)$
Space complexity = $O(n^2)$ for storing Vandermonde matrix in Inverse DFT

3 Distance computation using Matrix Multiplication

Given G = (V,E) be an unweighted undirected graph, we need to compute H = (V,E_H) i.e. the edges of H. E_H can be computed as follows:

- if (x,y) ∈ E, then (x,y) ∈ E_H
- if ∃ w ∈ V such that (x,w), (w,y) ∈ E, then (x,y) ∈ E_H
- this implies (x,y) ∈ E_H if ∃ a walk of length at most 2 from x to y in G

3.1 Part a (Computing Adjacency Matrix of H)

Lemma 1: Let A_G be the adjacency matrix of G. Then $((I + A_G)^2)_{ij} > 0$ iff there is a walk of length at most 2 from (i) to (j)
Proof : Proved in class lecture 22.

Let us call $(I + A_G)^2$ to be A'. For computing A_H, iterate over A' and assign $(A_H)_{ij} = 1$ if $A'_{ij} > 0$ (using lemma 1, $A'_{ij} > 0$ implies there is a walk of length at most 2 from i to j which is exactly what we want to compute H's adjacency matrix).

Algorithm 7 Adjacency Matrix of H

```
A_G ← Adjacency matrix of G
A' ← I + A_G
A' ← A'^2                                   ▷ Matrix multiplication O(n^w)
A_H ← |V| * |V| matrix, all 0's

for i from 1 to |V| do
    for j from 1 to |V| do
        if A'_ij > 0 then
            (A_H)_ij = 1
        end if
    end for
end for
return A_H
```

Time Complexity: Matrix addition will take O(n) time (adding 1 to diagonal entries). Matrix multiplication will take O(n^ω) time (where ω is the exponent of matrix-multiplication). So, adjacency matrix i.e. graph H can be computed from G in $\mathbf{O(n^\omega)}$ time.

3.2 Part b (Computing D_H)

To prove: for any x, y ∈ V, D_H(x,y) = $\lceil D_G(x,y)/2 \rceil$

D_G denote the distance-matrix of G. Let the path from x to y be (x = v_i, v_{i+1}, ..., v_{i+k-1}, v_{i+k} = y) of **length k**. Now, as (v_i,v_{i+1}) ∈ E_G and (v_{i+1},v_{i+2}) ∈ E_G, this implies (v_i,v_{i+2}) ∈ E_H (the way H is computed).

So, a valid path from x to y in H will be (x = v_i, v_{i+2}, ..., v_{i+k-2}, v_{i+k} = y) of **length k/2** if k is even, else (x = v_i, v_{i+2}, ...,v_{i+k-3}, v_{i+k-1}, v_{i+k} = y) of **length (k+1)/2** if k is odd. We know $D_G(x,y)$ = k, so, D_H(x,y) ≤ $\lceil D_G(x,y)/2 \rceil$.

Now, let us take the shortest path P_H from x to y in H, let the path P_H be (x = u_i, u_{i+1}, ..., $u_{i+k'-1}$, $u_{i+k'}$ = y). By the definition of how H is constructed, for any $u_j, u_{j+1} \in H(E)$, implies either $u_j, u_{j+1} \in G(E)$ or $\exists\ v \in G(V)$ such that $u_j, v \in G(E)$ and $v, u_{j+1} \in G(E)$. This implies, if $(u_j, u_{j+1}) \in H(E)$, then $D_G(u_j, u_{j+1}) \leq 2$. So, using this, we can construct P_G (path between x,y in G) using P_H.

$$D_H(x,y) = D_H(x = u_i, u_{i+1}) + D_H(u_{i+1}, u_{i+2}) + ... + D_H(u_{i+k'-1}, u_{i+k'} = y) \tag{17}$$

where $D_H(u_j, u_{j+1}) = 1$ as $(u_j, u_{j+1}) \in H(E)$. Using this, we can have an estimate on $D_G(x,y)$

$$D_G(x,y) \leq D_G(x = u_i, u_{i+1}) + D_G(u_{i+1}, u_{i+2}) + ... + D_H(u_{i+k'-1}, u_{i+k'} = y) \tag{18}$$

We know $D_G(u_j, u_{j+1}) \leq 2\ \forall j \in \{1, 2, ..., k'-1\}$ (stated above), hence, $D_G(x,y) \leq 2{*}k'$ where k' = $D_H(x,y)$ i.e. $D_G(x,y) \leq 2{*}D_H(x,y)$. If $D_G(x,y)$ is odd, then the equation $D_G(x,y) \leq 2{*}D_H(x,y)$ can be written as $D_G(x,y)+1 \leq 2 * D_H(x,y)$. Combining for $D_G(x,y)$ even or odd case, we get $\lceil D_G(x,y)/2 \rceil \leq D_H(x,y)$
Hence, we get, for any x, y ∈ V, D_H(x,y) = $\lceil D_G(x,y)/2 \rceil$

3.3 Part c

We will prove this by cases.

Given $M = D_H$ * A_G, we can interpret M(x,y) as summation over D_H(x,j) where j ∈ $nbrs_G(y)$:

$$M = \sum_{j \in nbrs_G(y)} D_H(x,j) \tag{19}$$

Let $D_G(x,y)$ = k be the distance between x,y in G.

Claim: $\forall j \in nbrs_G(y), D_G(x,j)$ = k-1 or k or k+1

Proof: As j is a neighbour of y, $D_G(x,y) \leq D_G(x,j) + D_G(j,y)$ which is $D_G(x,y) \leq D_G(x,j) + 1$. Also, as y is a neighbour of j, $D_G(x,j) \leq D_G(x,y) + D_G(y,j)$ which is $D_G(x,j) \leq D_G(x,y) + 1$. This gives $D_G(x,y) - 1 \leq D_G(x,j) \leq D_G(x,y) + 1$ i.e. $k-1 \leq D_G(x,j) \leq k+1$. Hence, $D_G(x,j)$ can vary among k-1,k,k+1.

We know from part b that D_H(x,y) = $\lceil D_G(x,y)/2 \rceil$ holds. Let us assume that y has a_1 neighbours with $D_G(x,j)$ = k, a_2 neighbours with $D_G(x,j)$ = k-1 and a_3 neighbours with $D_G(x,j)$ = k+1.

So, M can be written as:

$$M = a_1 * \lceil k/2 \rceil + a_2 * \lceil (k-1)/2 \rceil + a_3 * \lceil (k+1)/2 \rceil \tag{20}$$

Case 1: k = even
Using basic algebra, $\lceil k/2 \rceil$ = (k/2), $\lceil (k-1)/2 \rceil$ = (k/2) and $\lceil (k+1)/2 \rceil$ = (k/2)+1 where k = $D_G(x,y)$, and $D_H(x,y) = \lceil D_G(x,y)/2 \rceil$, $D_H(x,y)$ = (k/2). So, M can be written as:

$$M = a_1 * (k/2) + a_2 * (k/2) + a_3 * ((k/2)+1) \tag{21}$$
$$M = (a_1 + a_2 + a_3) * (k/2) + a_3 \tag{22}$$

we know $a_1 + a_2 + a_3 = degree_G(y)$ and $D_H(x,y)$ = (k/2). So,

$$M = degree_G(y) * D_H(x,y) + a_3 \tag{23}$$

as $a_3 \geq 0$, we get $M \geq degree_G(y) * D_H(x,y)$. Also, by backtracking over these steps, we can argue that whenever $M \geq degree_G(y) * D_H(x,y)$ holds, $D_G(x,y)$ will be even.

Case 2: k = odd
Using basic algebra, $\lceil k/2 \rceil$ = ((k+1)/2), $\lceil (k-1)/2 \rceil$ = ((k-1)/2) and $\lceil (k+1)/2 \rceil$ = ((k+1)/2) where k = $D_G(x,y)$, and $D_H(x,y) = \lceil D_G(x,y)/2 \rceil$, $D_H(x,y)$ = ((k+1)/2). So, M can be written as:

$$M = a_1 * ((k+1)/2) + a_2 * ((k-1)/2) + a_3 * ((k+1)/2) \tag{24}$$
$$M = (a_1 + a_2 + a_3) * ((k+1)/2) + a_2(-1) \tag{25}$$

we know $a_1 + a_2 + a_3 = degree_G(y)$ and $D_H(x,y)$ = ((k+1)/2). So,

$$M = degree_G(y) * D_H(x,y) - a_2 \tag{26}$$

Now, a_2 is the number of neighbours which are closer to x than y itself. Count of such neighbours is greater than 0 i.e. $a_2 \geq 1$. This is because, let the shortest path from x to y be (x = v_i, v_{i+1}, ..., v_{i+k-1}, v_{i+k} = y), so, $D_G(x, v_{i+k-1})$ = k-1 (can't be less than k-1 as k is the shortest distance to v_{i+k}) and can't be greater than k-1 because we have an estimate of k-1. Note that there is no harm in v_{i+k-1} = x. So, as $a_2 > 0$, we get $M < degree_G(y) * D_H(x,y)$. Also, by backtracking over these steps, we can argue that whenever $M \geq$

$degree_G(y) * D_H(x,y)$ holds, $D_G(x,y)$ will be odd.

So, it has been shown that the condition obtained in both parts is if and only if condition i.e. whenever $D_G(x,y)$ will be even, $M \geq degree_G(y) * D_H(x,y)$ holds, and whenever $D_G(x,y)$ will be odd, $M \geq degree_G(y) * D_H(x,y)$ holds.

One might try to argue of a case where $M \geq degree_G(y) * D_H(x,y)$ holds but $D_G(x,y)$ will be odd, we can easily give a counter argument as, if $D_G(x,y)$ will be odd, then the condition $M \geq degree_G(y) * D_H(x,y)$ is bound to hold true. Similarly for the other side.

Hence, we have shown that the conditions obtained are exhaustive i.e. whenever $M \geq degree_G(y) * D_H(x,y)$ holds, $D_G(x,y)$ will be even and $D_G(x,y) = 2 * D_H(x,y)$ and whenever $M \geq degree_G(y) * D_H(x,y)$ holds, $D_G(x,y)$ will be odd and $D_G(x,y) = 2 * D_H(x,y) - 1$

3.4 Part d (Computing D_G from D_H)

Given we have the graph G, implies we have A_G (the adjacency matrix of G). So, to compute D_G, we will first compute M (as done is part c) using D_H and M. This is a matrix multiplication operation, hence, will take $\mathbf{O}(n^\omega)$ time. Also, $degree_G(x)$ is simply the sum of the x^{th} row or the x^{th} column, we can compute $degree_G(i)$ $\forall i \in \{1,2,...|V|\}$ in $\mathbf{O}(n^2)$ time. Then, we can simply iterate over the M matrix and compare $M(x,y)$ with $degree_G(y) * D_H(x,y)$ and assign $D_G(x,y)$ according to part c which is again an $\mathbf{O}(n^2)$ operation.
Hence, Time complexity: $\mathbf{O}(n^\omega)$

3.5 Part e (All pairs distances)

Taking motivation from part c, we will be approaching this problem using transitive closure. Let G_0 denote the original graph G, G_1 denote H. G_2 will be computed from G_1 in a similar fashion as G_1 i.e. H was computed from G_0. G_2's adjacency matrix can be computed using $(I + G_1)^2$. So, $((I + G_1)^2)_{ij} > 0$ implies there is a walk of length at most 2 from i to j in G_1 i.e. there is a walk of length at most 4 from i to j in G_0. (using lemma 1).

So, we will construct G_0, G_1, ..., G_{k-1}, G_k where G_i contains an edge between (x,y) if there exists a walk of length at most 2^i from x to y in original graph G (by extending lemma 1). As we have n vertices, maximum possible size of a walk in G is n. So, G_k (transitive closure) contains an edge between (x,y) if there exists a walk of length at most 2^k i.e. n from x to y in original graph G which gives k = $\lceil \log_2 n \rceil$. $A_{G_k}(x,y) = 1$ implies there is some path in G from x to y. Also, $D_{G_k}(x,y)$ is either 1 (if $A_{G_k}(x,y) = 1$) or 0 (if no path exists), this is because, if there exists a path from x to y, then we will have an edge from x to y in G_k (transitive closure).

Using D_{G_k} and $A_{G_{k-1}}$, we can compute $M_k = D_{G_k} * A_{G_{k-1}}$, using the M_k obtained, we can compute $D_{G_{k-1}}$ as explained in part d in $\mathbf{O}(n^\omega)$ time.

Using the $D_{G_{k-1}}$ obtained, we can compute $D_{G_{k-2}}$ in a similar fashion. Repeating this procedure, we can compute D_{G_0} which is the all-pairs-distances of the original graph G.

Algorithm 8 RecursiveAlgo(A_G, count)

if $count = \lceil \log_2 n \rceil$ **then**
  $D_H \leftarrow |V| * |V|$ matrix, all 0's ▷ Transitive closure
  for `i from 1 to |V|` **do**
    for `j from 1 to |V|` **do**
      if $A_G(i,j) = 1$ **then**
        $D_H(i,j) \leftarrow 1$
      end if
    end for
  end for
  return D_H
end if

$A_H \leftarrow$ AdjacencyMatrixOfH(A_G) ▷ part a's algorithm: O(n^ω) time
$D_H \leftarrow RecursiveAlgo(A_H, count + 1)$
$M \leftarrow D_H * A_G$ ▷ O(n^ω) time
$D_G \leftarrow |V| * |V|$ matrix, all 0's

for `i from 1 to |V|` **do**
  for `j from 1 to |V|` **do**
    if $M(i,j) \geq degree_G(j) * D_H(i,j)$ **then** ▷ degree: O(n) time
      $D_G(i,j) \leftarrow 2 * D_H(i,j)$
    else if $M(i,j) < degree_G(j) * D_H(i,j)$ **then**
      $D_G(i,j) \leftarrow 2 * D_H(i,j) - 1$
    end if
  end for
end for
return D_G

Algorithm 9 All Pair Distances of G

$A_G \leftarrow$ Adjacency matrix of G
return RecursiveAlgo(A_G, 1)

Time Complexity:

- Computing G_i from G_{i-1} (i.e. computation of the adjacency matrix) takes

$\mathbf{O}(n^\omega)$ time (proved in part a). We repeat this operation k times where k $= \lceil \log_2 n \rceil$, hence, computing all G_i's takes $\mathbf{O}(n^\omega * \log_2 n)$ time.

- Computing $D_{G_{i-1}}$ from D_{G_i} takes O($n^\omega + n^2$) i.e. $\mathbf{O}(n^\omega)$ time (proved in part d). We repeat this operation k times where k $= \lceil \log_2 n \rceil$, hence, computing all D_{G_i}'s takes $\mathbf{O}(n^\omega * \log_2 n)$ time.
- Hence, overall **Time Complexity:** $\mathbf{O}(n^\omega * \log_2 n)$

4 Universal Hashing

Given, $U = \{0, 1, 2, ..., M-1\}$, p = prime Integer$\in [M, 2M]$, Integer n << M

Hash functions

$$\text{H(x)} = \text{x mod n}$$
$$\text{Hr(x)} = ((\text{rx}) \text{ mod p}) \text{ mod n}$$

4.1 Part a

We are given a set S of n elements selected randomly from universal set U.
Let A_i denote the chain length for i^{th} position in the Hash table of S.
More formally,

$$A_i = |S_i| \quad S_i = \{x \in S : H(x) = i\} \tag{27}$$

Proof by Logical deduction

$$Pr(A_i = x) = \left(\sum\nolimits_{i=0}^{n-1} \frac{|\{i, i+n, i+2n, ..., i+\frac{M(n-1)}{n}\}|}{M}\right)^x \tag{28}$$

$$Pr(A_i = x) \approx \frac{1}{n^x} \tag{29}$$

Therefore,

$$Pr(A_i > x) \approx \sum_{k=x+1}^{n-1} \frac{1}{n^k} \tag{30}$$

$$Pr(A_i > x) \leq \frac{1}{n^{x+1}}(1 + \frac{1}{n} + ...) \approx \frac{1}{n^x} \tag{31}$$

$$Pr(max(A) > x) \equiv Pr(A_i > x) \forall i \in \{0, 1, 2, ..., n-1\} \tag{32}$$

$$Pr(max(A) > x) \leq \sum_{i=0}^{n-1} \frac{1}{n^x} = \frac{1}{n^{x-1}} \tag{33}$$

Taking x = $\log_2 n$ and considering n > 4, the inequality boils down to

$$Pr(max(A) > x) \leq \sum_{i=0}^{n-1} \frac{1}{n^2} = \frac{1}{n} \tag{34}$$

Therefore, we have

Pr (Max-chain-length in hash table of S under H($\cdot$) > $\log_2 n$) $\leq \frac{1}{n}$

4.2 Part b

Given $r \in [1, p-1]$, we need to show that $\exists$ at least $\binom{(M/n)}{n}$ subsets of U of size n in which max chain length corresponding to hash function $H_r(x)$ is $\theta(n)$.

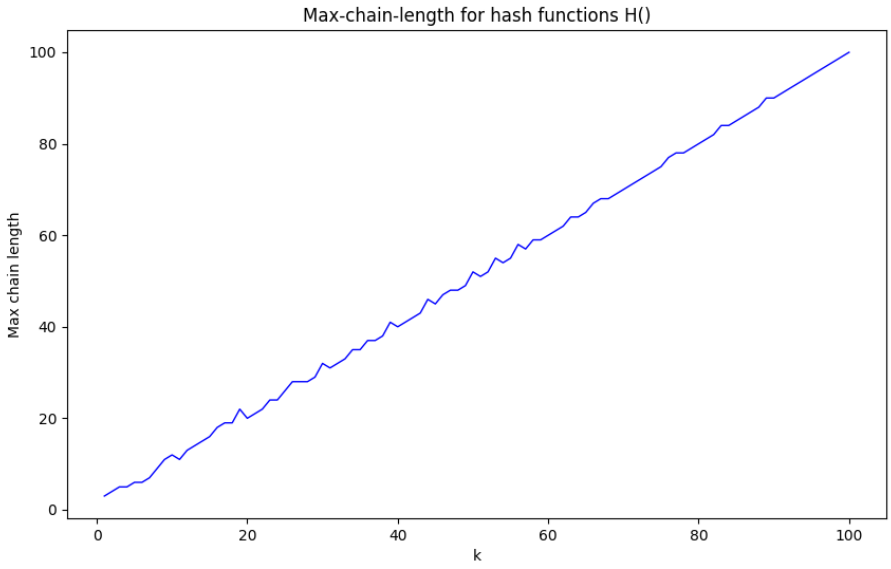
Now, consider set X_0, X_1, ..., $X_{\lfloor (M/n) \rfloor}$ such that X_i contains $\{(i-1)*n, (i-1)*n+1, ..., (i-1)*n+n-1\}$ i.e. size of X_i is n and X_i mod n = {0, 1, ..., n-2, n-1}. Now, we have proved in the lecture that the hash function H_r is invertible. So, let $X_i' = H_r(X_i)$
Due to invertible nature of $H_r(x)$, each X_i' is unique with size same as size of X_i. Also, we have

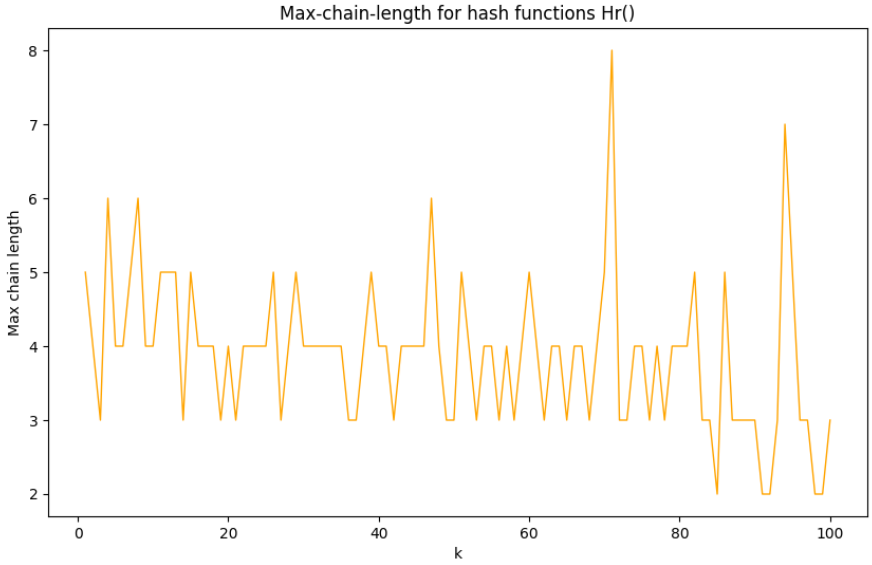
$$0 \leq H_r(i) \leq n-1 \tag{35}$$

Therefore, we can say that X_i' mod n = X_i mod n, i.e. $\{0, 1, 2, ..., n-1\}$ Now, out of these $\lfloor (M/n) \rfloor$ sets, we can choose n sets. From each of these n sets, we can select an element such that $H_r(x)$ is equal to k. All of these elements map to same slot in the hash table ie. k. The selected elements form a set of n elements, each of which satisfying the property that $H_r(x)$ = k. This sampling would make the max chain length to be $\theta(n)$. The selection of n sets from $\lfloor (M/n) \rfloor$ sets can be done in $\binom{(M/n)}{n}$ ways.
Therefore, there exists $\binom{(M/n)}{n}$ subsets of U of size n such that maximum chain length in hash-table corresponding to $H_r(x)$ is $\theta(n)$.

4.3 Part c

M = 10^4
n = 100





4.3.1 Observation

We observe that for Hash function H(), Max chain length increases almost linearly for k = 1 to n.
However, for Hash function H_r(x), Max chain length takes random values from y = 2 to 8.

A possible reason could be that the Hash function H() maps all elements in $\{0, n, 2n, ..., (k-1)n\}$ from S_k to 0
From lectures, the expected chain length for remaining values of S(chosen randomly from U) = $\frac{n-k}{M}$

Therefore, the expected chain length for 0^{th} index in Hash Table = k + $\frac{n-k}{M}$, while for other indices is $\frac{n-k}{M}$. Therefore, the expected max chain length increases linearly with k
For the Hash function H_r, we derived in class that the expected chain length for all slots in Hash table is constant. Therefore, we do not observe any specific trend for max chain length in this case. The average value of max chain length remains close to 4 and peak value reached is 8