Assignment 2
Submitted by: - Parth Parashar

Due: January 20, 2022

Your solutions must be typed (preferably typeset in LaTeX) and submitted as a PDF through Canvas before the beginning of class on the day its due.
When asked to provide an algorithm you need to give well formatted pseudocode, a description of how your code solves the problem, and a brief argument of its correctness.

Problem 1: Jarvis March (Gift Wrapping Algorithm) For this question you will need to research the *Jarvis March* convex hull algorithm. Be sure to cite your sources (ACM or IEEE formatted is preferred).

(a) [10 points] Give pseudocode describing the Jarvis March algorithm, a brief description of how it works, and explain its best and worst case efficiency.

Answer: - Now, we know from the definition of convex hull that given a set of points in a plane, a convex hull of the set is the smallest convex polygon that contains all the points of it.

Now for Jarvis march algorithm, the idea behind it is that we start from the leftmost point (or the point which has the minimum x-coordinate). Then we move in the counter clockwise direction.
To find the next point while moving in the counter clockwise direction, we will use the idea of slope.

For example:- Let there be three points p1,p2,p3

Slope of the line segment p1,p2: - (y2-y1)/(x2-x1) → (1)
Slope of the line segment p2,p3: - (y3-y2)/(x3-x2) → (2)

If (1) > (2), clockwise

Else, anti-clockwise,

Now, to select the next point, it is selected as the point that beats all other points at the counter-clockwise position.

The pseudo code can be given as follows: -

```
Jarvis(S)  //Here is the set of points
pointOnHull = leftmost point in S //This is because it is guaranteed to be a part of
```

the hull

i= 0
repeat
P[i] = pointOnHull
Endpoint = S[0] //This is the initial endpoint on the hull

For j from 1 to |s|
If (endpoint == pointOnHull) or (S[j] is on the left of the line from P[i] to endpoint)
endpoint = S[j] //found greater left turn, update the endpoint
i=i+1
pointOnHull = endpoint
until endpoint == P[0] //wrapped around to first hull point

Worst Case complexity: - ⊙(N ^ 2)
The worst case occurs when all the points are on the hull. This makes the worst case Because the inner loop checks every point in the set S and the outer loop repeats for each point on the hull.

Best case complexity: - ⊙(N log N)
The best case scenario occurs when the minimum number of points are present on the convex hull. This is because the loops repeat less number of times as we already Have the sense of counter clockwise direction.

(b) [5] points Give an example input on which Jarvis March will perform significantly better than Graham's scan and explain why it will perform better.

Answer: - Consider a plane which has 200 points and we have a convex hull which is present in that plane.
The convex hull present in the plane has only 15 points on it. Now when we run both the Jarvis march algorithm and graham scan algorithm in this scenario, then the Graham's scan will run better. This is because when less number of points are on the hull, then in Jarvis march, there is room for moving in the counter-clockwise direction and thus, find the best possible solution. This can be attributed to the fact that Jarvis March is based on the direction and outputs. Thus, the better the outputs, the better the performance. This cannot be said about graham's scan.

This also reflects in the time complexity of both the algorithms where the time complexity is O(nh) , h being the points on hull whereas the time complexity of graham's scan is O(n logn).

(c) [5] points Give an example input on which Graham's Scan will perform significantly better than Jarvis March and explain why it will perform better.

Answer: - Consider a plane which has 400 points and we have a convex hull which is present in that plane. This convex hull has all the 400 points on it. (or a

lot of points lie on it). In this case, the graham's scan will perform better than the Jarvis March.
This is because Graham's scan performs better with more and more points lying on the hull.

This can also be seen from the time complexities for both graham's scan and Jarvis March.

Problem 2: Find the Missing Number You are given a list of $n - 1$ integers A, in the range of 0 to $n-1$. There are no duplicates in the list. One of the integers is missing. (Feel free to assume that $n = 2^m$ for some integer m)

(a) [5 points] Give an efficient algorithm for finding the missing number, show its complexity, and argue its correctness. (You should try for $O(n)$-time, less efficient solutions will still get partial credit)

Answer: -

We can do this in O(n) time by iterating through the array and computing the sum of all numbers. Subtracting this from the sum of the n natural numbers will give us the value of the missing number.

The empty slot can be detected during the iteration in which the sum is calculated.

```
Sum = 0;
Index = -1;

For (I =0 ; i< arr.length; i++)
{
  If (arr[i] ==0)
{ index = 1;}
Else
{ sum += arr[i]; }
}
 total = arr.length + 1 * arr.length/2;

missing number = (total – sum) at index
```

(b) [10 points] For this question you are not allowed to access an entire integer with a single operation. The elements of the list are represented in binary, and the only operation you can use to access them is GetBinaryDigit(A[i],j) which returns the jth bit of element $A[i]$ which runs in constant time. Give an efficient algorithm for finding the missing number under these constraints, show its complexity, and argue its correctness. (You should try for $O(n)$-time and $O(\log n)$-space, less efficient solutions will still get partial credit)

Example: If we run GetBinaryDigit($A[i],j$) with $A[i] = 29$ and $j = 2$, it would return a 0 since 29 = 11101.

Answer: - Pseudo Code : -

```
I=1
totalNumbers = totalElements
return findmissingnumber(array[], I, totalNumbers)
findmissingnumber(arr[], I, totalNumbers)
if totalNumbers =1
even = 0
odd = 0
j=0

if getbinarydigit(array[j], i) =0
even++
else
odd++

for j in totalNumbers
i++
if(odd>even)
findmin(even[], I , even)
else
findmin(odd[], I, odd)
```

The time complexity is: - 0(n)