Assignment 2

Due: January 23, 2020

Kanyid, Bradon
bradon.kanyid@pdx.edu

Reimer, Daniel
daniel.reimer@pdx.edu

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.

Solution

Pseudocode

```
jarvisMarch(xs) {
  p = minOfXAxis(xs)
  convexHull = [p]

  forever:
    q = xs[0]
    for i in xs:
      if orientation(p, i, q) is counterclockwise:
         q = i
    p = q
    convexHull = cons(p, convexHull)

    if last(convexHull) == p:
      break

  return convexHull
}
```

Description

The Jarvis March algorithm works by first finding the element with the minimum x coordinate, since it is guarenteed to be on the convex hull. In an infinite loop, we select a random point, q, trying to find the next point on the convex hull. Then, iterate over the set of points, comparing the random point selected (q), the previous point that was found on the convex hull (p) and every other point within the set of points. During each iteration, update q is the most clockwise point. This will set q to the next greatest clockwise point from any other point in the set. We set p to the new found point on the hull, update the list of convex points, and repeat until we find the original leftmost point again.

Efficiency

The best-case efficiency is $O(hn)$ where h is the number of points on the hull. Since h is a constant for small numbers of h, the overall complexity is $O(n)$.

The worst-case complexity is $O(n^2)$-time, which occurs when all the points are on the hull. This is because in order to find the element on the convex hull, the points on the hull must be compared to every other element, which turns $O(nh)$ into $O(n^2)$, when $n = h$.

Citation

J. Erickson, "Computational Geometry," http://jeffe.cs.illinois.edu, 2008. [Online]. Available: http://jeffe.cs.illinois.edu/teaching/compgeom/notes/01-convexhull.pdf. [Accessed: 23-Jan-2020].

(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.

Solution

Jarvis March will perform significantly better than Graham's scan when most of the points are not on the convex hull. This is because the best case scenario for Graham's scan is $O(n \log n)$ due to needing to sort the list first regardless of the input. For Jarvis March, the best case scenario is when there is minimal points on the hull. With fewer points on the hull, Jarvis March will examine every point once for every point on the hull, effectively having a time complexity of $O(hn)$ where h is the number of points on the hull. Thus, with a small h value, the time complexity is essentially $O(n)$.

(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.

Solution

Graham's Scan will perform significantly better than Jarvis March when all the points are on the convex hull. When all the points are on the hull for Jarvis March, every point in the set must be compared with every point on the hull and when every point is on the hull, the time complexity is $O(hn) == O(n^2)$ when $h = n$, where h is number of points on the hull. For Graham's scan, the complexity of the sorting algorithm used is the complexity of Graham's scan therefore will always be $O(n \log n)$, no matter the size of the input.

Problem 2: Find the Missing Number You are given a list of $n-1$ integers A, in the range of 1 to n. 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 and $O(1)$-space, less efficient solutions will still get partial credit)

Solution

Pseudocode

```
findMissingNumberA(A) {
  lenA = len(A)
  //pre-compute sum of 1..N using closed-form solution
  sum = ((lenA)*(lenA+1))/2

  runningTotal = 0

  // loop through every element of A,
  // keeping a running total of everything in the array
  for(int i=0; i < aLen; i++) {
    runningTotal += A[i];
  }

  // The difference betweeen sum and runningTotal is
  // the size of our missing element
  return (sum - runningTotal)
}
```

Complexity

Because this algorithm has a single walk of the elements of A, it's complexity is $O(n)$-time, and because we know the closed-form solution is $(n)(n+1)/2$, we know this can be represented in $\lg(n) * \lg(n+1) - 1$ bits, so it has complexity of $O(\lg n)$-space.

Correctness

The algorithm is intuitively correct because we know what the sum of all elements $1..n$ would be, using the closed-form solution, we know that our list A contains every number in $1..n$ except one, and we know that there are no duplicates in A. Therefore, the difference between the closed-form solution and the sum of counting all the elements in A would have to be the size of the single, missing element.

(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 $O(logn)$-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 $29 = 11101$.

Solution

Pseudocode

```
findMissingNumberB(A) {
    // pre-compute some useful re-used values
    aLen = len(A)
    lgA = lg(aLen)

    // create array to hold intermediate results
    prevResult = new Array(lgA)

    // loop through every element of A
    for(int i=0; i < aLen; i++) {
      // check each bit for each element
      for(int j=0; j < lgA; j++) {
        // xor this element's bit w/ a running 'sum' of the previous xor'ed bits
        prevResult[j] ^= GetBinaryDigit(A[i], j)
      }
    }

    // now combine the stored bits back together into a result
    result=0;
    for (int j=lgA; j > 0; j--) {
      // build the resultant integer from it's bits
      result <<= prevResult[j];
    }
    // ... and return it
    return result;
  }
```

Complexity

Like in part 2a, this algorithm has a single walk of the elements of A, and it doesn't matter that we also walk across the bits of each element, so it's complexity is $O(n)$-time. We pre-allocate our result storage as $\lg(n)$elements, as this is the size to represent the largest binary number n, so it has complexity of $O(\lg n)$-space.

Correctness

Because we're assuming that $n = 2^m$, we can make further guarantees of what a full set of numbers from $1..n$ would look like, represented in binary. We know that any number in that range can be represented in $\lg(n)$ bits, and that every bit-pattern should be represented within those $\lg(n)$ bits.

For every binary digit, there should be an even number of occurances of 1's in a fully-represented bit-pattern that spans the space. Using xor allows us to track the even/odd status of each binary digit. In a fully-represented bit-space made up of n bits, xor'ing every number with the other will leave you with all zero bits in your reduction. In our array A, the only digits that won't have 0's in their position are digits that were not fully represented in the bit-pattern, i.e. were missing from the array.

To show this further, if someone were to xor the found missing number with the reduction variable, every digit would be 0, as at that point all bit-patterns within those $\lg(n)$ bits will have been represented.

It's important to leave the caveat that this is only true if $n = 2^n$ and we are working with binary digits.