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.

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

This algorithm begins with a call to $GiftWrap$, which takes as arguments a set of points Q (an array of structs which each contain an x and y coordinate for each point) and an integer n which is the the number of points in the set of points.

Algorithm 1 Jarvis March

```
1:  procedure GiftWrap(Q, n)
2:      if n < 3 then return -1
3:      CH = new array [n]
4:      initialize CH to all sentinel values (e.g. -1)
5:      p ← GetLeftMost(Q, n)
6:      for i = 0 → n − 1 do
7:          CH[i] ← p
8:          p_i ← Q[i]
9:          p_j ← Q[0]
10:         for j = 1 → n − 1 do
11:             p_k = Q[j]
12:             if (j == i)∨(MoreLeft(p_i, p_j, p_k)) then
13:                 p_j ← p_k
14:                 p ← j
15:         if p == CH[0] then return CH
        return CH
16: procedure GetLeftMost(Q, n)
17:     minx = Q[0].x
18:     index = 0
19:     for i = 1 → n − 1 do
20:         if Q[i].x < minx then
21:             minx = Q[i].x
22:             index = i
        return index
23: procedure MoreLeft(p_i, p_j, p_k)
24:     return [((p_j.x − p_i.x) · (p_k.y − p_i.y)) − ((p_k.x − p_i.x) · (p_j.y − p_i.y))] > 0
```

Brief Summary:
GiftWrap calls *GetLeftMost* (line 5) which returns the point with the smallest x-value, which is necessarily on the hull. This point becomes the first element in CH, which is an array of indices for the points of the Convex Hull of set Q. The frontier point is called p_i. An arbitrary point is selected ($Q[0]$) to be the potential next point on the hull, which we call p_j. Next, the doubly-nested for loop works by checking each of the remaining p_k points in Q against p_j and determining which points are on the hull by calculating the cross-product of the line $\overrightarrow{p_i\, p_j}$ against $\overrightarrow{p_i\, p_k}$ by calling *MoreLeft* to determine which is the leftmost point. If the cross-product is positive, p_j is updated to the value of p_k (line 13) and *GiftWrap* continues checking the remaining points. The leftmost point (from p_i's orientation) is added to CH until the index of said leftmost point is the same as the first index in CH, indicating that all points on the hull have been found.

Since the algorithm has two loops that run through n points, the worst case run-time would be $O(n^2)$. However, because the outer loop halts once the hull has returned to the initial point found, the algorithm is actually $O(n \cdot h)$ where h is the number of points on the hull.

Correctness:

Each time the loop iterates, the index p of Q is added to the list of indexes for points on the hull. This means that, at the beginning of each loop, the p must be the index for a point on the hull.

Initialization We assume no 3 colinear points. Initially, we can throw out all inputs less than three (line 2). This is correct because a hull must contain at least 3 points (a triangle). Next, we obtain the leftmost point by linearly scanning the x-values. We know this point must be on the hull because $\exists$ at least 3 points | each is on the hull and that $\exists$ up to 4 points with the maximum/minimum x/y value (if there are only 3 points then one point will share two of these extremes). Identifying the index of one of these points (minimum x-value) is where the loop begins, so the initialization is correct.

Maintenance: For successive iterations of the loop, we iterate through all the points unless it is the most recently discovered point on the hull (line 12, left side of $\vee$), comparing each line segment $\overrightarrow{p_i\, p_j}$ with $\overrightarrow{p_i\, p_k}$. This comparison is done by calculating the cross product of the two 'vectors' whose shared origin is the point p_i. If this cross product is positive, then we know that p_k is to the left, or counterclockwise to p_j. We want the leftmost point each time because we are wrapping CCW around the hull (it would be the rightmost if CW, we simply choose one or the other). We know a positive cross product means p_k is 'to the left' because $\exists$ two angles θ and ϕ, calculated with respect to p_i, | $\theta = tan^{-1}\frac{y_j}{x_j}$ and $\phi = tan^{-1}\frac{y_k}{p_k}$, where the sides of those triangles are the x and y distances from p_i to p_j and p_k, respectively, and the line segments $\overrightarrow{p_i\, p_j}$ and $\overrightarrow{p_i\, p_k}$ form the hypotenuse of each triangle. $MoreLeft$ first sets the origin of these two angles to p_i. The cross product allows us to compare these angles by recognizing that if the value of any $\frac{y}{x}$ is greater than another, the arctan of that result is also greater, because arctan(x) is monotonically increasing. p_j is updated (line 12) each time p_k is to the left until there are no more points left in the j loop. Therefore, when the loop ends, p is the index of the leftmost point for each iteration with respect to the last.

Termination The loop terminates when p is equal to the index of the first point selected on the hull (line 15). This means that we have gone all the way around the polygon and and found every point, returning to the first. Therefore, all the points in the hull have been found and CH contains the indices for all the points, in CCW order.

Sources:
formatting pseudo: tex.stackexchange.com/questions/163768/write-pseudo-code-in-latex
general/GiftWrap: wikipedia.org/wiki/Gift_wrapping_algorithm
general/analysis/MoreLeft: www2.cs.duke.edu/courses/fall14/compsci330/notes/scribe14.pdf
cross product/MoreLeft : CLRS 33.1 (pg. 1016-17)
cross product proof: sites.math.rutgers.edu/ ajl213/CLRS/Ch33.pdf

(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.

If we imagine a set of points Q which contains a subset of 3 points Q_k such that the triangle they form contains all the other points in $Q - Q_k$, then it would make sense to employ JM, because $\forall n \geq 16$ (or, better yet, > 8), we can see that $cn \cdot 3 \leq cn \lg n$

(c) [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.

Since JM's worst-case run time is $O(n^2)$, it is not too hard to see that if every single point in Q was on the hull (i.e., if all the points formed a convex polygon), then JM would check each of these points against each other, rather inefficiently. Since we know that Graham's Scan runs in $O(n \lg n)$ no matter the input, such a convexly polygonal set of points would better suit Graham's Scan.

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)

We remember from Gauss's discovery that the summation of any ordered set of numbers from 1 to n is equal to the first and last number added together n times, divided by two; or from our summation formulas:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

For an example, suppose we had the ordered set 1 to 9. We know that:

$$\sum_{i=1}^{9} i = \frac{9(9+1)}{2} = 45$$

If we were to be missing one of the numbers, this number would be the difference between the Gaussian summation and the summation of all $i = 1 \rightarrow n-1$ numbers (the n numbers minus the missing number). For example, if the above summation instead came out to be 37, we could

determine that 8 was the missing number. An algorithm which simply added up all the given numbers, compared it to the expected sum, and calculated the difference should be sufficient. Lets see:

Algorithm 2 Missing Number

```
1: procedure MissingNumber(A)
2:     n = A.length + 1
3:     for i = 1 → n − 1 do
4:         sum+ = A[i]
5:     total = n·(n+1)/2
6:     missing = total − sum
7: return missing
```

Complexity:

There is only one loop which adds the elements in A linearly, so this procedure is clearly $\in O(n)$. As for space complexity, I am under the impression that, since this algorithm is in place and doesn't use any auxiliary data structures (we only declare 4 auxiliary variables) that it uses constant space.

Correctness:

Since we are told that there are no duplicates, we know that there will be a unique difference between the total summation of all the numbers and the same summation without one of its numbers. In other words, there is no possible way that two different numbers, if they were missing in the ordered set 1..n could produce the same result when we compute the difference between the two summations.

(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 $A[i] = 13$ and $j = 2$, it would return a 0 since $13 = 1101$.

Discussion: It seems that we could solve this problem in a couple different ways.

Method 1: We could convert every number from binary to decimal, and then solve the problem in the same way that we solved the previous problem. We know that BinToDec converts a single binary number to decimal in $O(k)$ time, where k is the number of bits in the binary number. We would still have to convert all n binary numbers, so we've already reached $O(nk)$ and then it would take another n operations to sum, etc. This could certainly be achieved, but I'

Method 2: We could sort all the binary numbers and then find the first instance where the least significant bit does not alternate. This would be correct because bitstrings in order must have alternating least-significant bits, as in the series $1, 10, 11, 100, 101, 110, 111...$ (Note that we start

on 1 rather than 0 because the prompt does not include 0 in the set). The most efficient sorting algorithm runs in $O(n \lg n)$ time, so we've exceeded our goal again.

Method 3: We could count the number of 1s and 0s in all n bitstrings, and store each of these counts in a separate variable. We could then compare each count to find out how many even and odd numbers we had. This makes sense because an even binary number ends in 0, an odd in 1. We observe that for an uninterrupted (no missing number) series of bitstrings (importantly, starting with 1), $\forall n\ (count_1 > count_0) \vee (count_1 == count_0)$. The two cases correspond to cases where n is odd or even, respectively. If we are missing a number m, we can have the following outcomes:

If n is **odd**: $(count_1 == count_0) \vee (count_1 > count_0)$
If n is **even**: $(count_1 < count_0) \vee (count_1 > count_0)$

These possible outcomes correspond to cases where m is odd or even, respectively. Looking at this from a slightly different perspective, we observe that:

If m is **odd**, $count_1 \leq count_0$
If m is **even**, $count_1 > count_0$

This tells us that, if we count up all the least significant bits in our set, we can determine if n is even/odd, and ignore all the remaining odd/even numbers in the set. We can continue doing this with the next most significant bit because, assuming $n = 2^x$, there should be an even number of 1s and 0s at each bitplace, and when we discover an imbalance, we know the whether or not the bit of m in that bitplace is "on" or "off." We then simply keep track of these bits in an array and return this array.

Keep in mind when analyzing the following pseudocode that this algorithm assumes that GetBinaryDigit takes a second argument k such that $k = 0$ is the least significant bit of each number, rather than $k = 0$ being the most significant bit, which is kind of backward but we'll allow it here because we aren't sure how GetBinaryDigit (which, we should point out, is a confusing name for a procedure that gets a *bit*), is implemented under the hood. (The confusion arises because the prompt says j = 2 accesses the second bit from the right in a bitstring of 4 bits, which could be taken to mean that j = 1 accesses the least significant bit, or that j = 0 accesses the most significant bit. The same logic would work the other way, we would just start k at the bitlength of n and decrement k at the end of each while loop) Also, the lg n space constraint will only work if our missing number is not n (because we start at 1 instead of 0), so we assume $n > m \geq 1$.

Correctness:
The algorithm creates a bitstring array and the invariant is that at each iteration of the loop, a new bit from the missing number is added to this array, from right to left, until the missing numbers bits have filled this array and we are finished.

Initialization: At the beginning of the loop, *bitstring* is an array of size lg$n$. There is nothing in the array, so the invariant holds.

Maintenance: Because each iteration counts and compares the 1s and 0s of each number, we use the logic from above to discover the parity of our missing number in the first iteration. We then plug the corresponding bit into the bitstring. We also overwrite the beginning of the array with remaining relevant numbers, halve our set size, and continue. When the next loop begins, bitstring has the next bit of the missing number.

Algorithm 3 Missing Binary Number

```
1: procedure MissingBinNumber(A)
2:     n = A.length
3:     set = n
4:     k = 0
5:     bitstring = new array of size lg(n)
6:     z = bitstring.length − 1
7:     while set > 0 do
8:         keep = 0
9:         count_1 = 0
10:        count_0 = 0
11:        for i = 0 → set − 1 do
12:            if GetBinaryDigit(A[i], k) == 1 then
13:                count_1 + +
14:            else if GetBinaryDigit(A[i], k) == 0 then
15:                count_0 + +
16:        if (count_1 ≤ count_0) then
17:            bitstring[z] = 1
18:            for j = 0 → set − 1 do
19:                if GetBinaryDigit(A[j], k) == 1 then
20:                    A[keep] = A[j]
21:                    keep + +
22:        else
23:            bitstring[z] = 0
24:            for j = 0 → set − 1 do
25:                if GetBinaryDigit(A[j], k) == 0 then
26:                    A[keep] = A[j]
27:                    keep + +
28:        k + +
29:        z − −
30:        set/ = 2
       return bitstring
```

Termination: The while loop terminates when the set size becomes 0, meaning we have looked at all of the relevant subsets of numbers until we are finished.

Analysis: We are only using one auxiliary array of size $\lg n$, so the space constraint is met. Although this method uses a while loop, we can imagine a recurrence to think about the complexity of this solution. We begin with a problem of size n and we iterate over the n elements in the while loop at most twice (in each for loop). Then we halve the input size for each successive loop. As stated above, each call to GBD incurs a constant cost. So, a corresponding recurrence would be:

$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + 2cn \\
T(\frac{n}{2}) &= T(\frac{n}{4}) + 2c\frac{n}{2} \\
\therefore T(n) &= T(\frac{n}{4}) + 2c(n + \frac{n}{2}) \\
\implies T(n) &= T(\frac{n}{2^i}) + 2cn(1 + \frac{1}{2} + \frac{1}{4} ... + \frac{1}{2^{lgn}}) \\
&= T(\frac{n}{2^i}) + n(1 + \frac{1}{2} + \frac{1}{4} ... + \frac{1}{n}) \\
&= \sum_{i=0}^{\lg n} \frac{n}{2^i} \\
&= n \sum_{i=0}^{\lg n} \frac{1}{2^i} \\
&= n(1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{n}) \\
&= 2n - 1 \\
&= O(n)
\end{aligned}
$$