Spring 2021

Name ______________________________

CS360 - Exam 1 - 100 Points
DUE WEDNESDAY 2/24 by 9:00am/10:00am

Use scratch paper to work out solutions and then **neatly** write your answers in the space provided, or use a *separate* document (written or electronic) for *ALL* your answers. Upload a **SINGLE** pdf file to Canvas. **BE SURE** to proofread your submission so that the pages are **in order** and **clearly legible**. Use a **scanner** app for any handwritten work.

You may use your textbook, the course website including lecture notes, Zoom lecture recordings, my posted homework solutions, and your homework solutions along with any notes you've taken from class - but in the words of Porky Pig "THAT'S ALL FOLKS!"

YOU MAY NOT WORK WITH OR HELP OTHER STUDENTS, OR REFER TO OUTSIDE (e.g. Internet) SOURCES.
But you **may** ask me questions regarding the exam prior to the due date.

1.(40 points) **Maximum Matching**

Given an array of n values stored in an array A, determine the value with the maximum number of duplicates in the array.

(a) *Brute Force*

One simple approach is to select each element of the array as the key, loop through the elements following the key counting if a match is found, then updating the maximum count and maximum value if the current count is larger. The pseudocode for this algorithm is given by:

```
BRUTE_MAX_MATCH(A)
1   n = A.length
2   max_count = 0
3   max_value = A[1]
4   for i = 1 to (n-1)
5      count = 1
6      for j = (i+1) to n
7         if A[j] == A[i]
8            count = count + 1
9      if count > max_count
10        max_count = count
11        max_value = A[i]
12  return max_value
```

(i) Give the *worst-case* run time using an asymptotic analysis and give a **brief intuitive** rationale, i.e. 1 or 2 sentences, for your answer. **DO NOT** give an exact analysis or formally prove correctness, but explain which lines contribute to the bound and which ones will not execute.

(ii) What are the best and worst case inputs for this algorithm. Briefly explain intuitively, but *do not* give an asympotic analysis justification. Hint: Consider how the input may or may not cause certain lines to execute.

(iii) Complete the table below and determine the *exact* run time $T(n)$ of BRUTE_MAX_MATCH(). (Note: there may be additional unknowns for conditional statements). **DO NOT perform any algebraic simplifications for this part.**

Line	**Cost**	**# Iterations**
1 `n = A.length`		
2 `max_count = 0`		
3 `max_value = A[1]`		
4 `for i = 1 to n-1`		
5 `   count = 1`		
6 `   for j = i+1 to n`		
7 `      if A[j] == A[i]`		
8 `         count = count + 1`		
9 `   if count > max_count`		
10 `      max_count = count`		
11 `      max_value = A[i]`		
12 `return max_value`		

(iv) Determine the *exact* runtime formula for $T(n)$ in the *worst case* as a function of n. (Note: There should **not** be any additional unknowns). Be sure to simplify as much as possible.

(v) Using the *exact* runtime formula from part (iv), determine the *asymptotic* run time of BRUTE_MAX_MATCH() in the *worst case* and *briefly* justify your answer.

(b) *Smarter Approach:* **Use only topics and data structures covered to this point in class** (Lectures 1-5, Assignments 1-2). You may use algorithms covered in class by simply calling the routine and stating their asymptotic runtimes, i.e. you do **not** need to give the pseudocode for them or justify the asymptotic runtimes.

(i) Give a pseudocode strategy for an asymptotically more efficient algorithm for finding the value with the maximum duplicates and *intuitively* **argue that it is correct**, i.e. you do **not** need to formally prove correctness using a loop invariant proof. Hint: Consider rearranging the values.

```
SMART_MAX_MATCH(A)
```

(ii) Determine the *asymptotic* run time of your SMART_MAX_MATCH() algorithm in the *worst case* and compare it to BRUTE_MAX_MATCH(). Note: Use an asymptotic analysis, you do **NOT** need to find the *exact* run time.

2.(25 points) **Natural Sorting**

Consider a rather intutitive sorting algorithm that sorts n numbers stored in an array A by first using a linear search to find the smallest element of $A[1..n]$ and exchanging it with the element in $A[1]$. Then repeats the linear search to find the second smallest element of $A[2..n]$, exchanges it with $A[2]$, and continues this process for the first $n-1$ elements of A. This sorting strategy is given by the following pseudocode algorithm:

```
EXAM01-SORT(A)
n = A.length
for i = 1 to (n-1)
   smallest = i
   for j = (i+1) to n
      if A[j] < A[smallest]
         smallest = j
   exchange A[i] <-> A[smallest]
```

(a) State a loop invariant for the *outer loop.*

(b) Prove correctness for this sort using the loop invariant from part (a). Hint: You do not need to prove an invariant for the inner loop, but just describe the behavior of the inner loop as part of the maintenance step. Also, consider what needs to be true at termination.

(c) Similar to INSERTION_SORT(), the asymptotic runtime of EXAM01_SORT() can easily be shown to be $O(n^2)$. Download the skeleton file from

http://faculty.ycp.edu/˜dbabcock/spring2021/cs360/CS360_Exam01.zip

and implement the pseudocode above adding counts for each line of *pseudocode* in the same manner as the implementations from assignments 1 and 2. Attach a hardcopy of your source code, a table of empirical values, and a graph showing the data points and trend curve with appropriate legend indicating the approximated constant.

3. (15 points) **Master Theorem**

Use the Master Theorem to solve the following recursive equations by **showing each of the steps described in class**.

(a) $T(n) = 4T(n/2) + 7n^2 - 2n$

(b) $T(n) = 2T(n/3) + \Theta(n^2)$

4.(20 points) **Local Minima**

Consider the problem of finding a local minimum in an array A containing n **distinct** integers. We define a local minimum of A to be a value x such that $A[i] = Ax$, for non-boundary elements (e.g. $1 < i < n$) as

$$A[i-1] > A[i] < A[i+1]$$

In other words, a local minimum x is less than its neighbors in A. For boundary elements ($A[1]$ and $A[n]$), there is only one neighbor to consider, so only one of the inequalities needs to be met to consider that element a local minimum. There may be *many* local minima, but consider locating any *one* of them.

One algorithm to solve this problem uses divide and conquer. The proposed algorithm starts by selecting the middle element in the array, i.e. the element at location $m = n/2$ (we assume an n element array where n is odd for simplicity). Then one of three cases holds:

Case 1: $A[m-1] > A[m] < A[m+1]$. Then $A[m]$ is a local minimum, so return it.

Case 2: $A[m-1] < A[m]$. If this is the case, the left half of the array must contain a local minimum somewhere (as $A[m-1]$ is either a local minimum itself or will also be case 2). Therefore we recursively look for a local minimum in the left half.

Case 3: $A[m] > A[m+1]$. Then the right half of the array must contain a local minimum. Therefore we recursively look for a local minimum in the right half.

As an example, suppose $A = \{10, 6, 3, 7, 4, 5, 12, 15, 11, 19, 18\}$, then since $m = (p+q)/2 = (1+11)/2$ the algorithm will examine $A[6] = 5$. Since $A[5] = 4 < A[6] = 5$, we have case 2 giving a recursive call **FIND_LOCAL_MINIMA(A,1,5)**. Then examining $A[3] = 3$ gives case 1 since $A[2] = 6 > A[3] = 3$ and $A[3] = 3 < A[4] = 7$, thus the routine returns $A[3] = 3$ as a local minima.

The algorithm is given by

```
FIND_LOCAL_MINIMA(A,p,q)
1.   n = A.length
1.   A[0] = A[1] + 1
2.   A[n+1] = A[n] + 1
3.   m = (p+q)/2
4.   if A[m-1] > A[m] and A[m] < A[m+1]  //Case 1
5.      return A[m]
6.   if A[m-1] < A[m]                    // Case 2
7.      FIND_LOCAL_MINIMA(A,p,m-1)
8.   if A[m] > A[m+1]                    // Case 3
9.      FIND_LOCAL_MINIMA(A,m+1,q)
10.  return error
```

(a) Derive the recursive equation for FIND_LOCAL_MINIMA(). Show all work and explain how each term in the final equation is derived from the pseudocode.

(b) Determine the *worst-case* asymptotic runtime of FIND_LOCAL_MINIMA(). Explain the input that produces worst-case behavior.

(c) Determine the *best-case* asymptotic runtime of FIND_LOCAL_MINIMA(). Explain the input that produces best-case behavior.