CSC 413/513 Computer Algorithms

Spring 2021

Midterm Examination (Take-home)

Release Time: March 4

Due Time: 11:59 PM, March 28

Total points: 100

NAME Jackie Diep

STUDENT ID# W10076331

1. (20 points) **Filling the blanks** (4 points for each question).

   (1) Definition of an algorithm: ____.

Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as an output

   (2) There are three important aspects for each algorithm, which are design, _______, and, __________.

Design, Analysis, Proof

   (3) Typical algorithmic design strategies include:____________,____________, greedy approach, and backtracking.

Divide and conquer, Dynamic programming, Greedy approach, Backtracking

   (4) Give the definition of the asymptotic notation small w: $f(n) = w(g(n))$:____.

For any positive constant c > 0, there exists a constant $n_0 > 0$ s.t. $0 \le c * g(n) < f(n)$ for all $n \ge n_0$

$$\lim_{n \to \infty} f(n) / g(n) = \infty$$

   (5) The worst-case complexity of insertion sort algorithm is θ(_____); the worst-case complexity of merge sort algorithm isθ(_____).

Insertion sort:$\theta(n^2)$

Merge sort: $\theta(n * lg(n))$

***** where lg(n) stands for $\log_2(n)$**

2. (15 points). **Complexity Analysis.**
   Maximum Subarray problem: Given an array $A[1\ldots n]$ of numeric values (can be positive, zero, and negative) determine the subarray $A[i\ldots j]$ ($1\leq i \leq j \leq n$) whose sum of elements is maximum over all subvectors.
   Below is a brute-force algorithm. Analyze its best case, worst case and average case time complexity in terms of a polynomial of n and the asymptotic notation of θ. You need to show the steps of your analysis.

```
MAX-SUBARRAY-BRUTE-FORCE(A)
n = A.length
max-so-far = -∞
for l = 1 to n
    sum = 0
    for h = l to n
        sum = sum + A[h]
        if sum > max-so-far
            max-so-far = sum
            low = l
            high = h
return (low, high)
```

CSC 413 Midterm #2

```
n = A.length                     C1·1
max-so-far = -∞                  C2·1
 for l = 1 to n                  C3(n+1)
     sum = 0                     C4 n
     for h = l to n                      C5(n^2+1)
         sum = sum + A[h]                C6 n^2
         if sum > max-so-far             C7 n^2
             max-so-far = sum            C8 n
             low = l                     C9 n
             high = h                    C10 n
return (low, high)               C11 1
```

if loop; guaranteed to happen at least once; only once if only first # is pos

Best case: $n=1$

$C_1 + C_2 + C_3(n+1) + C_4 n + C_5(n^2+1) + C_6 n^2 + C_7 n^2 + C_8 n + C_9 n + C_{10} n + C_{11}$

$2c + cn + c + cn + cn^2 + \quad c + 2cn^2 + 3cn + c$

$5c + 5cn + 3cn^2$

$5 + 5n + 3n^2$

$5 + 5(1) + 3(1)^2 = 13$

est	$= 13$ & $\Omega(n)$
orst	$= 3n^2 + 5n + 5$ & $O(n^2)$
vg	$= \frac{3n^2+5n}{2} + 5$ & $\Theta(n^2)$

Originally had if-structure expressions as $c_i n^2$ and worst case as "$6n^2 + 2n + 5$", but I failed to see an array where the if structure could activate more than "n" times

3. (15 points). **Complexity Analysis:** Considering the following algorithm, analyze its best case, worst case and average case time complexity in terms of a polynomial of *n* and the asymptotic notation of θ. You need to show the steps of your analysis.

Consider the algorithm for the sorting problem that sorts an array by counting, for each of its elements, the number of smaller elements and then uses this information to put the element in its appropriate position in the sorted array:

ALGORITHM *ComparisonCountingSort*($A[0..n-1]$)
    //Sorts an array by comparison counting
    //Input: Array $A[0..n-1]$ of orderable values
    //Output: Array $S[0..n-1]$ of A's elements sorted
    //   in nondecreasing order
    for $i \leftarrow 0$ **to** $n-1$ **do**
        $Count[i] \leftarrow 0$
    for $i \leftarrow 0$ **to** $n-2$ **do**
        for $j \leftarrow i+1$ **to** $n-1$ **do**
            if $A[i] < A[j]$
                $Count[j] \leftarrow Count[j]+1$
            else $Count[i] \leftarrow Count[i]+1$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        $S[Count[i]] \leftarrow A[i]$
    return S

CSC 413 Midterm #3

```
for i ← 0 to n-1 do                          C1·(n+1)
    Count[i] ← 0                             C2·(n)
for i ← 0 to n-2 do                          C3·n
    for j ← i+1 to n-1 do                    C4·n^2
        if A[i] < A[j]                       C5·n^2
            Count[j] ← Count[j]+1            C6·n  > if (only one
        else Count[i] ← Count[i]+1           C7·n
for i ← 0 to n-1 do                          C8·(n+1)
    S[Count[i]] ← A[i]                       C9·n
return S                                     C10·1
```

$C_1\cdot(n+1) + C_2\cdot n + C_3\cdot n + C_4 n^2 + C_5 n^2 + C_6 n + C_8(n+1)$
$+ C_9 n + C_{10}$

$\cancel{c}n + \cancel{c} + \cancel{c}n + \cancel{c}n + cn^2 + cn^2 + \cancel{c}n + \quad + \cancel{c}n + \cancel{c} + \cancel{c}n + \cancel{c}$

$6cn + 3c + 2cn^2$

$2n^2 + 6n + 3$

$2(1)^2 + 6(1) + 3 = 11$

Let a = amount of possible values
Best case: $11 \rightarrow \Omega(n+a)$
Worst case: $2n^2 + 6n + 3 \rightarrow O(n^2+a)$
Avg case: $n^2 + 3n + 3 \rightarrow \Theta(n^2+a)$

- Research + textbook indicate that counting sort is usually $\Omega(n+a)$, $O(n+a)$, and $\Theta(n+a)$, but no examples used a nested for-loop

4. (20 points). **Correctness Proof of Bubble Sort:** Bubble Sort is a popular, but inefficient sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. Prove the correctness of following Bubble Sort algorithm based on Loop Invariant. Clearly state your loop invariant during your proof.

   ALGORITHM *BubbleSort*($A[0..n-1]$)
   //Sorts a given array by bubble sort
   //Input: An array $A[0..n-1]$ of orderable elements
   //Output: Array $A[0..n-1]$ sorted in nondecreasing order
   for $i \leftarrow 0$ **to** $n-2$ **do**
   for $j \leftarrow 0$ **to** $n-2-i$ **do**
   if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

   Prove: Bubble Sort will order the elements of array A, s.t. the elements are in ascending order.

   *****Array A' will be used to refer to a subarray of elements of A that have been sorted by the algorithm**

   Loop Invariant: At the start of each iteration of the loops, the altered form of A[0..n - 1] , A'[0..n - 1], holds the same elements as the original array that may be in a different order, and the element in the first index of A' will be the smallest element of A'.

   Initialization: No elements of array A have been sorted by the algorithm, s.t. A' is empty and trivially holds the smallest element of the subarray.

   Maintenance: Every iteration of the loop will compare index A[j] with the index to its right. If A[j] is less than its right index, they will stay ordered as they are. If A[j] is greater than its right index, the two values are swapped. Each iteration adds to the length of the subarray, and according to the loop invariant, the first value of A' is still the smallest element of A'.

   Termination: When i > n-2, the loop terminates, and array A' will hold all of the same elements of array A sorted in ascending order.

5. (7 points). **Proof by asymptotic definition:** Use the definition of the asymptotic notation small *o* to prove the following:

Let

$$p(n) = \sum_{i=0}^{d} a_i n^i ,$$

where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.

If $k > d$, then $p(n) = o(n^k)$.

o(g(n)) = {f(n) : for any positive constant c > 0, there exists a constant $n_0 > 0$ s.t. $0 \le f(n) < c * g(n)$ for all $n \ge n_0$}

Informally, this means that for some constant c, f(n) is less than c(g(n)). This also means that if c is positive, then g(n) and f(n) are positive, and the inequality must hold true.

We are given c > 0, $n_0 > 0$, and $n \ge n_0$. Along with $a_d > 0$ given in the prompt, then g(n) must be positive, and the inequality f(n) < c * g(n) must hold true.
In conclusion, p(n) must be equal to $o(n^k)$ within the given parameters.

6. (8 points). **Proof by asymptotic definition:** Use the definition of the asymptotic notation ϴ to prove the following:

   Show that for any real constants a and b, where $b > 0$,

   $$(n + a)^b = \Theta(n^b) .$$

Θ(g(n)) = {f(n) : there exist positive constants c_1, c_2, and n_0 s.t. $0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$ for all $n \ge n_0$}

$f(n) \in \Theta(g(n))$ which can be rewritten as $f(n) = \Theta(g(n))$

In this case, let $f(n) = (n + a)^b$

and let $g(n) = n^b$

s.t.

$0 \le c_1 * (n^b) \le (n + a)^b \le c_2 * (n^b)$ for all $n \ge n_0$

Starting from $(n + a)^b \le c_2 * (n^b)$,

$((n+a)/n)^b \le c_2$

$(n+a)/n \le c_2^{1/b}$

$1 + (a/n) \le c_2^{1/b}$

$|a/n| \le c_2$

$\mathbf{|a| \le n}$

Because n_0 must be a positive constant and $n \ge n_0$, $n > 0$,

$(n + a)^b \le (n + |a|)^b$

Because we can say that $|a| \le n$,

then $(n + a)^b \le (n + n)^b$

$(n + a)^b = (2n)^b$

$\mathbf{(n + a)^b = 2^b n^b}$

$0 \le c_1 * (n^b) \le \mathbf{2^b n^b} \le c_2 * (n^b)$ for all $n \ge n_0$

For any value of $c_1 \le 2^b$ and $c_1 > 0$, any value of $c_2 \ge 2^b$, and $n \ge |a|$, the inequality must hold true.

7. (15 points). **Divide and Conquer Algorithm Design:** Assume we want to sort an array *A*[1..*n*] based on merge sort algorithm and you have been already provided with an implementation of the merge procedure *Merge*(*A*, *p*, *q*, *r*) to combine two sorted arrays *A*[*p*..*q*] and *A*[*q*+1..*r*], that is you can use this procedure directly:
   (1) (4 points) Write the pseudocode for the merge sort algorithm.
   (2) (3 points) Draw a tree diagram to demonstrate the merge sorting process of the following sequence: 5 4 3 2 1.
   (3) (8 points) Similarly, if you are provided with a procedure *Find-Max-Crossing-Subarray* (*A*, *low*, *mid*, *high*), write the pseudocode for a divide-and-conquer algorithm to solve the maximum subarray problem, which has been described in Question 2.

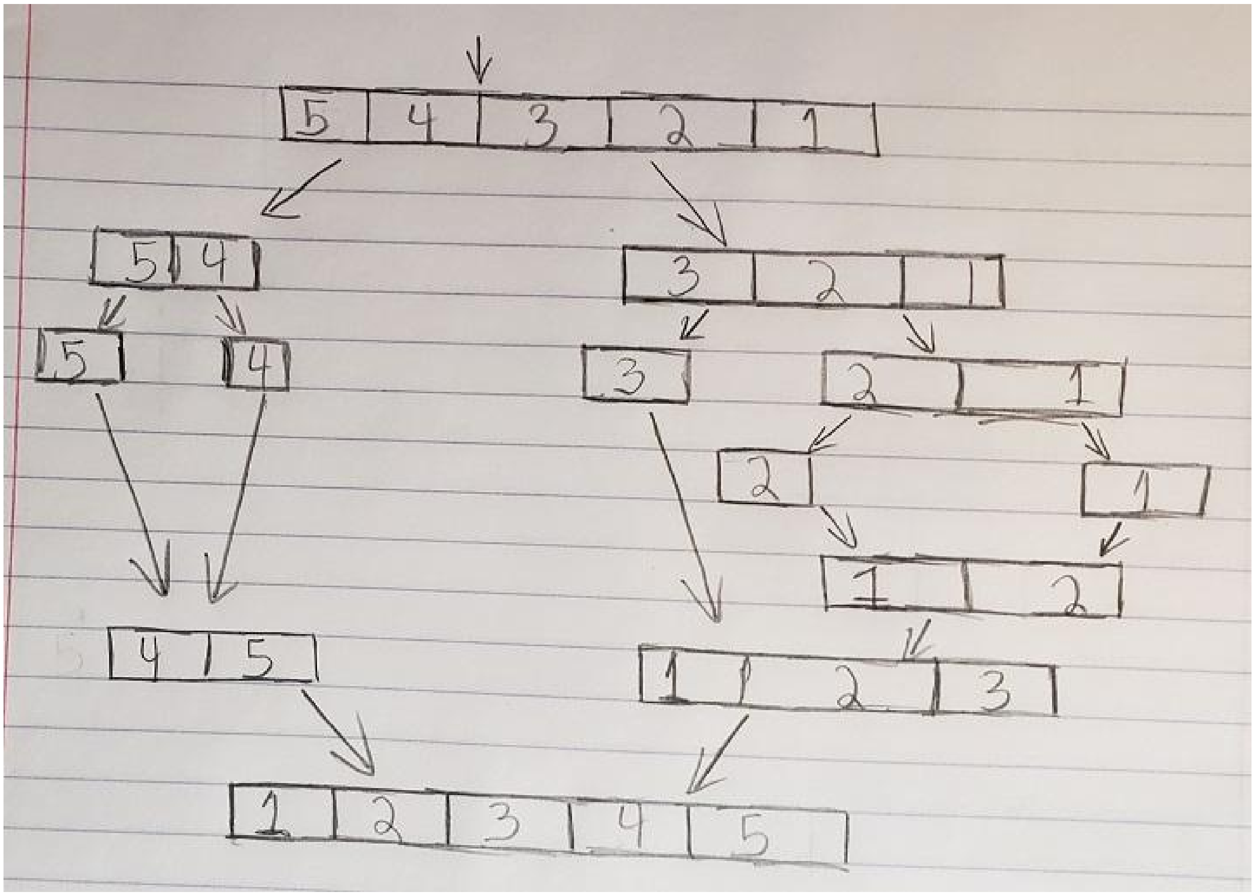
***Had trouble interpreting the question

"*Merge*(*A*, *p*, *q*, *r*) to combine two sorted arrays *A*[*p*..*q*] and *A*[*q*+1..*r*], that is you can use this procedure directly…"

I interpreted this as writing the Merge(...) procedure to be unnecessary.

1. MergeSort(A){

```
        r ← length(A)
        if ( r ≤ 1) {
                return; }
        p ← 0;
        q ← r/2;
        leftArray ← A[0...q];
        rightArray ← A[q+1…r];
        for i ← 0 to q - 1{
                leftArray[i] ← A[i]; }
        for i ← q to r - 1{
                rightArray[i - q] ← A[i]; }
        MergeSort[leftArray];
        MergeSort[rightArray];
        Merge(A, p, q, r);
};
```

2. 

(8 points) Similarly, if you are provided with a procedure *Find-Max-Crossing-Subarray* (*A*, *low*, *mid*, *high*), write the pseudocode for a divide-and-conquer algorithm to solve the maximum subarray problem, which has been described in Question 2.

*****This problem did not say that I could not use the procedure directly, so I wrote it to be safe**

3. divide-Max-Subarray(A, low, high){

```
divide-Max-Subarray(A, low, high){
        if low >= high{
                return array[low]; }
        middle = (low + high)/2;
        maxLeftSum = divide-Max-Subarray(A, low, mid - 1);
        maxRightSum = divide-Max-Subarray(A, mid + 1, high);
        maxCrossingSum = Find-Max-Crossing-Subarray(A, low, mid, high);

        return max(maxLeftSum, maxRightSum, maxCrossingSum);
        }

Find-Max-Crossing-Subarray(A, low, mid, high){
        maxLeftCrossing = -∞;
        currentSum = 0;
        for i = mid down to low - 1{
                currentSum = currentSum + A[i];
                if currentSum > maxLeftCrossing{
                        maxLeftCrossing = currentSum; }
        }
        maxRightCrossing = -∞;
        currentSum = 0;
        for i = mid + 1 to high + 1{
                currentSum = currentSum + A[i];
                if currentSum > maxRightCrossing{
                        maxRightCrossing = currentSum; }
        }
        return (maxLeftCrossing + maxRightCrossing );
```