**COMP 3270**

**Assignment 2**

**100 points**

**Due Friday, June 17th by 11:59PM**

Instructions:

1. This is an individual assignment. There are 10 problems.
2. Late submissions **will not** be accepted unless prior permission has been granted or there is a valid and verifiable excuse.
3. Think carefully; formulate your answers, and then write them out concisely using English, logic, mathematics and pseudocode (no programming language syntax).
4. Type your final answers in this Word document.
5. Don’t turn in handwritten answers with scribbling, cross-outs, erasures, etc. If an answer is unreadable, it will earn zero points.
6. **(6 points)** Prove that the following algorithm is correct by using the “Proof by Loop Invariants” method.

**Hint**: Loop Invariant **Si=x is not equal to any of the first i elements of the array**

Text, letter

Description automatically generated

**Loop Invariant: Si=x is not equal to any of the first i elements of the array.**

**Initialization: We know the loop invariant is true because i = 0 before the loop starts, so Si is the first element in the array. IfSi=x, then that means there aren’t any elements before the first that equals x.**

**Maintenance: Assume the loop invariant holds at the start of iteration i. Then it must be that Si=x is not equal to any of the first i elements of the array. In iteration i, the loop determines whether the element in that memory location is equal to x and continues iterating until it finds a value s.t. Si=x by incrementing the value of i. The loop invariant holds since incrementing i means that all the previous i elements are not equal to x. Thus, at the start of iteration i + 1, Si+1=x is not equal to any of the first i +1 elements of the array.**

**Termination: When the while loop terminates, the algorithm outputs the index i such that x = A[i]. The loop invariant states that Si=x is not equal to any of the first i elements of the array, which was proven since the loop iterates until it finds the index s.t. x = A[i]. Therefore, the algorithm is correct.**

**2. (5 points)** Order the following list of functions by the big-Oh notation. Group together (for example by underlining) those functions that are big-Theta to each other.

Text

Description automatically generated with medium confidence

**Increasing order:**

**3. (5 points)** Describe a method for finding both the minimum and maximum of n numbers using fewer than 3n/2 comparisons. ***Hint:*** First construct a group of candidate minimums and a group of candidate maximums.

**We would first construct a group of candidate minimums and group of candidate maximums by comparing pairs of numbers and separating each pair between the largest and smallest number. This process takes n/2 comparisons since we are incrementing the array by 2. From there, we can find the maximum number in the max array and the minimum number in the min array, which will take (n/2)-1 comparisons each. Therefore, the number of comparisons in total would be (3n/2)-2.**

**4. (18 points)**

Algorithm Mystery(A: Array [i..j] of integer) i & j are array starting and ending indexes

if i=j then return A[i]

else

k=i+floor((j-i)/2)

temp1= Mystery(A[i..k])

temp2= Mystery(A[(k+1)..j]

if temp1<temp2 then return temp1 else return temp2

(a) (1 points) What does the recursive algorithm above compute?

**The recursive algorithm computes the minimum number in the array.**

(b) (4 points) Develop and state the two recurrence relations exactly (i.e., determine all constants) of this algorithm by following the steps outlined in L7-Chapter4.ppt. Determine the values of constant costs of steps using directions provided in L5-Complexity.ppt. Show details of your work if you want to get partial credit.

|  |  |
| --- | --- |
| Steps | Cost |
| If I = j then return A[i] | O(1) |
| Else |  |
| k=i+floor((j-i)/2) | O(1) |
| temp1= Mystery(A[i..k]) | O(n/2) |
| temp2= Mystery(A[(k+1)..j] | O(n/2) |
| if temp1<temp2 | O(1) |

T(n) = 2T(n/2)+O(1)

Since O(1) is constant, T(n) = 2(T(n/2))+1

Text, letter

Description automatically generated

(c) (6 points) Use the Recursion Tree Method to determine the precise mathematical expression T(n) for this algorithm. First, simplify the recurrences from part (b) by substituting the constant “c” for all constant terms. Drawing the recursion tree may help but you do not have to show the tree in your answer; instead, fill the table below. Use the examples worked out in class for guidance. Show details of your work if you want to get partial credit.

You will need the following result:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Level | Level number | Total # of recursive executions at this level | Input size to each recursive execution | Work done by each recursive execution, excluding the recursive calls | Total work done by the algorithm at this level |
| Root | 0 | 1 | N | C | C |
| One level below root | 1 | 2 | n/2 | C | 2c |
| Two levels below root | 2 | 4 | n/4 | C | 4c |
| The level just above the base case level | Lg(n)-1 | 2^(log(n))-1 | 2 | c | C(2^log(n)-1) |
| Base case level | Lg(n) | N | 1 | c | nc |

A piece of paper with writing

Description automatically generated

(d) (1 points) Based on T(n) that you derived, state the order of complexity of this algorithm:

**T(n)= c + 2c + 4c +8c… nc = O(n)**

**5. (10 points)** T(n)=7T(n/8)+cn; T(1)=c. Determine the polynomial T(n) for the recursive algorithm characterized by these two recurrence relations, using the Recursion Tree Method. Drawing the recursion tree may help but you do not have to show the tree in your answer; instead, fill the table below. You will need to use the following results, where and b are constants and x<1:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| level | Level number | Total # of recursive executions at this level | Input size to each recursive execution | Work done by each recursive execution, excluding the recursive calls | Total work at this level |
| Root | 0 | 1 | n | c | c |
| 1 level below | 1 | 7 | n/8 | C | 7c |
| 2 levels below | 2 | 49 | n/64 | C | 49c |
| The level just above the base case level |  |  |  | c | C () |
| Base case level |  |  | 1 | C | cn |

**T(n) = c + 7c + 49c … cn**

**= O(n)**

**6. (11 points)** Use the substitution method to prove the guess that is indeed correct when T(n) is defined by the following recurrence relations: T(n)=3T(n/3)+5; T(1)=5. At the end of your proof state the value of constant c that is needed to make the proof work.

Statement of what you have to prove:

**T(n) = O(n) when T(n) is defined by the recurrence relations T(n)=3T(n/3) +5; T(1)=5. We want to show that *.***

Base Case proof:

**. This is true if .**

Inductive Hypotheses:

**Assume . We must show that .**

Inductive Step:

**.**

Value of c:

**7. (16 points)** Guess a plausible solution for the complexity of the recursive algorithm characterized by the recurrence relations T(n)=T(n/2) +T(n/4)+T(n/8)+T(n/8)+n; T(1)=c using the Substitution Method. (1) Draw the recursion tree to three levels (levels 0, 1 and 2) showing (a) all recursive executions at each level, (b) the input size to each recursive execution, (c) work done by each recursive execution other than recursive calls, and (d) the total work done at each level. (2) Pictorially show the shape of the overall tree. (3) Estimate the depth of the tree at its shallowest part. (4) Estimate the depth of the tree at its deepest part. (5) Based on these estimates, come up with a reasonable guess as to the Big-Oh complexity order of this recursive algorithm. Your answer must explicitly show every numbered part described above in order to get credit.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| level | Level number | Total # of recursive executions | Input size to each recursive execution | Work done by each recursive execution, excluding the recursive calls | Total work at this level |
| Root | 0 | 1 | n | c | c |
| 1 level below | 1 | 3 | n/2,n/4,n/8 | C | 3c |
| 2 levels below | 2 | 9 | n/4,n/8,n/16,n/8,n/16  n/32,n/16,n/32,n/64 | C | 9c |

1. Recursion Tree

Diagram

Description automatically generated

1. Depth of tree at shallowest part occurs when you take the heaviest branch.

The height will be:

1. Depth of tree at its deepest part occurs when you take the lightest branch.

The height will be:

1. The time complexity of this recursive algorithm would be roughly O(n). If you have this many recursive calls, then the sum of the total statements will be slightly less than n (which is n/2 + n/4 + n/8 < n). Since f(n) is a little less than n, the algorithm would be around O(n).

**8. (10 points)** Use the Substitution Method to prove that your guess for the previous problem is indeed correct.

Statement of what you have to prove:

**We must prove that the recurrence relations T(n)=T(n/2) +T(n/4)+T(n/8)+T(n/8)+n; T(1)=c has a time complexity of O(n).**

Base Case proof:

Inductive Hypotheses:

**Assume We must show .**

Inductive Step:

**9. (9 points)** Use the Master Method to solve the following three recurrence relations and state the complexity orders of the corresponding recursive algorithms.

1. **T(n)=2T(99n/100)+100n**

**A = 2, b = 100/99, f(n) = 100n**

**is growing much faster than 100n, so**

1. T(n)=16T(n/2)+n3lgn

**A = 16, b = 2, f(n) = (n^3)lgn**

**is larger than , so .**

1. T(n)=16T(n/4)+n2

**A = 16, b = 4, f(n) = n^2**

**Since .**

**10. (10 points)** Use Backward Substitution (10 points) and then Forward Substitution (10 points) to solve the recurrence relations T(n)=2T(n-1)+1;T(0)=1. In each case, do the following: (1) Show at least three expansions so that the emerging pattern is evident. (2) Then write out T(n) fully and simplify using equation (A.5) on Text p.1147. (3) Verify your solution by substituting it in the LHS and RHS of the recurrence relation and demonstrating that LHS=RHS. (4) Finally state the complexity order of T(n). You must show your work for parts (1)-(3) to receive credit.

**Backward substitution:**

**We see that**

**LHS:**

**RHS:**

**Forward substitution:**

**We see that**

**LHS:**

**RHS:**