**COMP 3270**

**Assignment 2**

**100 points**

**Due Tuesday, September 20th 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: If Si = x is not equal to any of the first I elements of the array then return -1**

**Initialization: Before the first iteration of the loop, variable i is 0. So for A[0], the algorithm will return -1 if input element x is not equal to A[0].**

**Maintenance condition: Initial condition is satisfied then after the first iteration of the loop also condition should satisfy and also for the next iterations. The first iteration of loop i = 0, then the algorithm will return -1, if x is not equal to the first element of the array.**

**After the next iteration I = 1 (as I = I + 1), the algorithm will return -1, if x is not equal to the next element of the array.**

**Termination: At the termination of the given loop I = (n-1) + 1, algorithm will not return -1, if x is not equal to the last element of the array. Then this is the final output of the algorithm that executes the given while loop.**

**Therefore the algorithm is proven 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

1. 1/n
2. 2100
3. Loglogn
4. Sqrt(logn)
5. Log2n
6. N.01
7. Ceiling(sqrt(n)), 3n.5
8. 2logn, 5n
9. Nlog4n, 6nlogn
10. Floor(2nlog2n)
11. 4n3/2
12. 4logn
13. N2logn
14. N3
15. 2n
16. 4n
17. 22^n

**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.

Declare 2 variables max and min.

If n is odd, initialize max and min elements as A[0], otherwise we initialize max and min as the maximum and minimum of A[0] and A[1] respectively.

Beginning with the next index in each case, we iterate through the arrays with increments of two. So, we check in pairs. If A[i] is greater than A[i+1], then A[i] is a candidate for maximum and A[i+1] is a candidate for minimum. Hence we check for these respectively. This takes 3 comparisons whatever the case. Similar to the case when A[i] is less than A[i+1], here A[i] is a candidate for minimum and A[i+1] is a candidate for maximum.

As each step takes 3 comparisons and we iterate through array in increments of 2, the total number of comparisons is less than 3(n/2)

The exact number of comparisons are:

If n is odd: 3(n-1)/2

If n is even: (3n/2) - 2

**4. (6 points)** Consider the following “proof” that the Fibonacci function F(n), defined as F(1) = 1, F(2) = 2, F(n) = F(n-1) + F(n-2), is O(n):

* Base case (n<=2): F(1) = 1 which is O(1), and F(2) = 2, which is O(2).
* Inductive hypothesis (n>2): Assume the claim is true for n’ < n.
* Inductive step: F(n) = F(n-1) + F(n-2). By induction, F(n-1) is O(n-1) and F(n-2) is O(n-2). Then, F(n) is O((n-1)+(n-2)). Therefore, F(n) is O(n), since O((n-1)+(n-2)) is O(n).

What is wrong with this proof?

To determine the time complexity of this recursive Fibonacci function, you would have to develop the

recurrence relations to characterize its behavior and solve them. This proof claims that F(i) = O(i) for all i < n, which is not true.

**5. (12 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 given recursive algorithm compute the minimum element in array A[i…..j]

(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.

Recurrence relation

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

Substitution

T(n/2) = 2T(n/4) + 1

T(n) = 2(2T(n/4) + 1) + 1

T(n) = 22T(n/22) + 2 + 1

Similarly

T(n) = 23T(n/23) + 22 + 2 +1

T(n) = {1+2+22+23+….. } log2n terms

T(n) = 1[(2log2n – 1)/(2-1)] = (n-1)

T(n) = O (n)

(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 | 1 | 1\*1=1 |
| One level below root | 1 | 2 | n/2 | 1 | 1\*2=2 |
| Two levels below root | 2 | 4 | n/4 | 1 | 1\*4=4 |
| The level just above the base case level | Log2n - 1 | 2log2n ­- 1 |  | 1 | 1\*2log2n – 1 |
| Base case level | Log2n | n | 1 | 1 | n |

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

T(n) = (1+2+4+8+….) log2n times T(n) = O(n)

**6. (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 | Cn | cn |
| 1 level below | 1 | 7 | n/8 | C(n/8) | 7c(n/8) |
| 2 levels below | 2 | 72 | n/82 | C(n/82) | 72c(n/82) |
| The level just above the base case level | log8 (n) -  1 | (7(log 8 (n)))/7 | 8 | 8c | 8c(7(log 8 (n)))/7 |
| Base case level | log8 (n) | 7(log 8 (n)) | 1 | c | c(7(log 8 (n))) |

T(n) = ∑ i=0 log8(n) – 1 (7/8)i cn + O(7^(log 8 (n))) < ∑ i=0 ∞ (7/8)i cn + O(n^(log 8 (7))) = cn/(1-7/8) + O(n^(log 8 (7))) = 8cn + O(n^(log 8 (7))), i.e. T(n) < a linear polynomial, so T(n) = O(n)

7. (11 points) Use the substitution method to prove the guess that )()(nOnT 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) -> T(n) ≤ cn – 3

Base Case proof:

5 = T(1) ≤ c1 - 3 (if c ≥ 8)

Inductive Hypotheses:

Assume T(n/3) ≤ c(n/3) – 3

Inductive Step:

T(n) ≤ 3(c(n/3)-3) + 5 = cn – 4 ≤ cn - 3

Value of c: c = 8

**8. (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. A piece of paper with writing on it

Description automatically generated

(3) The shallowest part of the tree is when the 1/8 branch is taken every time.

Assuming a height of h: n(1/8)h ≤ 1 -> n ≤ 8h -> h ≥ log8 n

(4) The deepest part of the tree is when the 1/2 branch is taken every time.

Assuming a height of h: n(1/2)h ≤ 1 -> n ≤ 2h -> h ≥ log2n

(5) T(n) = n log n

**9. (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: T(n) = O(n log n) -> T(n) ≤ c(n log (n) + 1)

Base Case proof: c = T(1) ≤ c(1 log (1) + 1) = c

Inductive Hypotheses: Assume T(n/2) ≤ c(n/2 log (n/2) + 1), T(n/4) ≤ c(n/4 log (n/4) + 1), T(n/8) ≤ c(n/8

log (n/8) + 1)

Inductive Step: T(n) ≤ c(n/2 log (n/2) + 1) + c(n/4 log (n/4) + 1) + 2c(n/8 log (n/8) + 1) + n

= c(n/2(log n – log 2) + 1) + c(n/4(log n – log 4) + 1) + 2c(n/8(log n – log 8) + 1) + n

= cn log n – cn(1/2 log 2 + 1/4log 4 + 1/4log 8) + 4c + n ≤ c(n log (n) + 1)

**10. (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

nlog[100/99](2) = n69 , f(n) = O(nlog[100/99](2) – 68 ) = O(n69 – 68 ) = O(n)

Case 1 -> T(n) = Θ(nlog[100/99](2) ) = Θ(n69 )

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

a = 16, b = 2, f(n) = n3log n

nlog[2](16) = n4 , f(n) = O(n4 – ε )

Case 1 -> T(n) = Θ(n4 )

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

a = 16, b = 4, f(n) = n 2

nlog[4](16) = n2 = f(n)

Case 2 -> T(n) = Θ(n2log n)

**11. (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:

(1) T(n-1) = 2T(n-2) + 1 -> T(n) = 2(2T(n-2) + 1) + 1= 4T(n-2) + 3

T(n-2) = 2T(n-3) + 1 -> T(n) = 2(2(2T(n-3) + 1) + 1) + 1 = 8T(n-3) + 7

T(n-3) = 2T(n-4) + 1 -> T(n) = 2(2(2(2T(n-4) + 1) + 1) + 1) + 1 = 16T(n-4) + 15

(2) T(n) = 2n T(0) + (1 + 2 + 4 + … + 2n-1 ) = 2n \*1 + (2 n – 1) = 2n+1 – 1

(3) LHS: T(n) = 2n+1 – 1

RHS: 2T(n-1) + 1 = 2(2n-1+1 - 1) + 1 = 2n-1+2 - 2 + 1 = 2n+1 – 1

(4) 2n = O(n!)

Forward substitution:

(1) T(1) = 2T(0) + 1 = 2(1) + 1 = 3

T(2) = 2T(1) + 1 = 2(3) + 1 = 7

T(3) = 2T(2) + 1 = 2(7) + 1 = 15

(2) T(n) = 2(2n – 1) + 1 = 2n+1 - 2 + 1 = 2n+1 – 1

(3) LHS: T(n) = 2n+1 – 1

RHS: 2T(n-1) + 1 = 2(2n-1+1 - 1) + 1 = 2n-1+2 - 2 + 1 = 2n+1 – 1

(4) 2n = O(n!)