**COMP 7270 Assignment 1 200 points No late submissions accepted!**

**Due by 11:59 PM Sunday 02/19/2017**

**Upload your submission well before this deadline. Even if you are a few minutes late, as a result of which Canvas marks your submission late,** **your assignment may not be accepted**.

Instructions:

1. This is an individual assignment. You should do your own work. **Any evidence of copying will result in a zero grade and additional penalties/actions.**
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 answers in this Word document and submit it. If that is not possible, use a word processor to type your answers as much as possible (you may hand-write/draw equations and figures), turn it into a PDF document and upload**.

**1.** Solve the following three recurrences using the Master Method and state the order of complexity of the corresponding recursive algorithm. If you simply state the complexity order without showing your work, and the case that applies, you will not get any credit even if the complexity order is correct.

1. T(n)=3T(n/2)+n

According to this equation: a = 3, b = 2, f(n) = n

So: nlog of a to the base b=nlog of 3 to the base 2

So: f(n)=O(n(log of 3 to the base 2) -ε)(constant ε>0)

According to master method, T(n) = Θ(nlog of 3 to the base 2)

(b) T(n)=3T(n/2)+n(log of 3 to the base 2)

According to this equation: a =3 , b = 2, f(n)= n(log of 3 to the base 2)

So : nlog of a to the base b = n(log of 3 to the base 2)

So : f(n)=Θ( n(log of 3 to the base 2))

According to master method , T(n) = Θ( n(log of 3 to the base 2)lgn)

(c) T(n)=3T(n/2)+n3

According to this equation: a=3, b = 2, f(n)= n3

So : nlog of a to the base b = n(log of 3 to the base 2)

So : f(n)=Ω( n(log of 3 to the base 2) +ε )(constant ε>0)

Also:3f(n/2) = 3n3/8 , 3n3/8≤cn3(c=1/2)

According to master method, T(n) = Θ(n3)

Θ( n(log of 3 to the base 2)lgn)> Θ(nlog of 3 to the base 2)> Θ(n3)

**2.** Suppose a recursive algorithm is characterized by these recurrences: T(n)=3T(n/2)+n; T(1)=1. Let a guessed solution be T(n)=O(n2). Prove that this guess is correct using the Substitution Method. State values of the constants n0 and c that you determine as part of the proof. You must clearly show the three parts of the inductive proof to get credit. Hints: (i) The goal of simplification in the Inductive Step is to get to a form T(n)≤(the expression you are trying to prove)─(another term) so that you can show that T(n)≤(the expression you are trying to prove) when (another term)≥0. (ii) Note that the value of n in any expression is such that n≥n0.

**Since we guess solution be** T(n)=O(n2).

**so T(n)≤cn2  for an appropriate choice of the constant c > 0.**

**Base case: 1=T(1) )≤c**

**Now assume: T(n/2) ≤ c(n/2)2**

**We have to show T(n) ≤ cn2**

**T(n) ≤3c(n/2)2+n =(3/4)cn2+n = cn2-(1/4)cn2+n= cn2 -[(1/4)cn2-n]= cn2 –n[(1/4)cn2-1]**

**If n≥0 and [(1/4)cn2]-1≥0 we can make sure this solution, so n≥1 and c≥4**

**So T(n)=O(n2) (if n0 =1,c=4)**

**3.** Use the Recursion Tree Method to show that T(n)=2(2n)─1 for a recursive algorithm characterized by the recurrences T(n)=2T(n-1)+1; T(0)=1. You must fill in the table below for the first three levels of the Recursion Tree and for the base case level and the level above. Then write out the expression T(n) that you get by adding all values in the last column and simplifying using results from Appendix A to show the above. This simplification must be shown to get any credit.

Finally, state the most accurate complexity order of this algorithm:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Level # | # of recursive executions at this level as a function of level # | Input size to each execution | Additional work done by each execution | Total work done at this level as a function of level # |
| 0 | 20 | n | 1 | 1\*20 |
| 1 | 21 | n-1 | 1 | 1\*21 |
| 2 | 22 | n-2 | 1 | 1\*22 |
| Level just above base case level | 2n-1 | 1 | 1 | 1\*2n-1 |
| Base case level | 2n | 0 | 1 | 1\*2n |

Since T(n) by all recursive executions can be obtained by vertically adding the column "total work done” as the last line as the equation Sn = , a1 = 1, q=2,n=n+1

**T(n)=**=-1

**Complexity Order = O(2n)**

**4.** Solve the recurrences of Q.3 using the Backward Substitution Method. Show your work.

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

T(n-1) = 2T(n-2)+1⇒ T(n)=2[2T(n-2)+1]+1 = 4T(n-2)+2+1

T(n-2)=2T(n-3)+1⇒T(n)=4T(n-2)+2+1=8T(n-3)+4+2+1

T(n-3)=2T(n-4)+1⇒T(n)=8T(n-3)+4+2+1=16T(n-4)+8+4+2+1

So: T(n)=2n[T(n-n)]+ 2n-1 = 2n+1-1

Check:

LHS of recurrence T(n) = 2\*(2n)-1

RHS = 2T(n-1)+1=2\*[2\*2(2n-1)-2+1=2\*(2n)-1

LHS=RHS

So: T(n) = 2n+1-1

**5.** Solve the recurrences of Q.3 using the Forward Substitution Method. Show your work.

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

T(0)=1.

T(1)=2T(1-1)+1= 2+1 = 3 T(2)=2T(2-1)+1= 6+1 = 7 T(3)=2T(3-1)+1=14+1 = 15

T(4)=2T(4-1)+1=30+1=31

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

Check:

LHS : T(n)= 2\*(2n)-1

RHS =2T(n-1)+1 = 2\*(2n-1-1)+1=2\*(2n)-1

LHS=RHS

So: T(n) = 2n+1-1

**6.** Prove by contradiction that the following algorithm to multiply two integers without using multiplication is correct. Note that your proof must be detailed and that every step of the proof other than the initial assumption must be based on (and justified by) a mathematical or logical fact or a step of the algorithm. Several initial steps of the proof are given as hints. Complete the rest of the proof.

function multiply(y,z: non-negative integer)

1 if z==0 then return(0)

2 else if z==1 then return(y)

else

3 product=0

repeat

4 product=product+y

5 z=z–1

6 until z==0

7. return product

Proof by contradiction with numbered steps:

1. Suppose the algorithm is incorrect.

2. There is some value of z≥0 for which the answer returned, product ≠ z\*y.

3. This cannot be z=0 because when z=0 z\*y=0 and algorithm step 1 returns the correct answer.

4. This cannot be z= 1 because when z=1 z\*y=1 and algorithm step 2 returns the correct answer.

5. If z>1, the condition checks in steps 1 & 2 will fail so steps 3-7 will be executed.

<complete the rest of the proof>

6.That means for at least one valid input z>1, it will produce an incorrect answer.

7.That means for at least one valid input z>1, it will not produce the valid of z\*y

8.Step three firstly gives an value named “product” to z\*y which equals to 0 and then repeat the steps from step 4 to 6

9.Steps 4 to 6 divided z into z portions, every portion will add one y to the previous value “product”.

10. In other words, during the recursive steps from 4 to 6, there will be z times additions of y and this value will be added into “product”

11.so for all value z>1, this algorithm will print the the value of z\*y

12.Step 7of the proof contradicts step 2!

13.So our assumption on the proof step 1 must be wrong, i.e. the algorithm has to be correct

**7.** Consider the Selection Problem (SP) – selecting the k-th largest number from among n distinct numbers, 1≤k≤n. Prove that the algorithm below is correct (i.e., that it will print out the k-th largest number in A at termination) using the Loop Invariant:

“Before any execution of the outermost for loop with i=p, the (p-2) largest numbers in the original array A will be in cells A[1]...A[p-2] arranged in the descending order, and the remaining n-(p-2) numbers will be in cells A[p-1]...A[n].”

Note: Your proof must be written clearly, precisely, and at an appropriate level of detail.

**SelectK-thLargest** (A: Array [1…n] of distinct numbers; k: integer such that 1≤k≤n)

1 for i=2 to (k+1)

2 for j=n down to i

3 if A[j]>A[j-1] then

4 temp=A[j]

5 A[j]=A[j-1]

6 A[j-1]=temp

7 return A[k]

Initialization: prove that the LI holds true before the loop begins.

Before the start of the iteration of the loop with i=2, the subarray A[1…..1] consists of elements originally in A[1…1] but in sorted order, trivially true because A[1…1]is one element array which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: prove that if the LI holds true before an execution of the outermost for loop with i=p, it will be true before the next execution with i=p+1.

1. Suppose that before the starts of the loop’s iteration with value i, the subarray A[ 1…i-1] consists of the elements originally in A[1…i-1]but in sorted order.
2. During the i(th) iteration of the loop, the if loop(line 3~6) compares A[n]with A[i],A[i+1] ,A[i+2] etc.until
3. either a number A[k]>A[i] (k in i to n)
4. it turns out that A[i] is greater than all numbers A[i+1]… A[n]
5. Each number found to be greater than A[i] is moved one cell to the left, i.e, If A[i+1]>A[i]then A[i+1]moved to A[i] and so on.
6. If case(1)holds, then the numbers A[k] is moved to the left of A[i], if case (2)holds, then the numbers less than A[i] is on the right of A[i]
7. According to our assumption, A[1…i-1] in sorted order, also A[i-1]>A[i], which means any elements in A[1…i]are greater than A[i+1…n],meanwhile A[1…i]in descending order.

According to our assumption ,if i=p, the (p-2) largest number will be the number A[p-2], and the

remaining n-(p-2) numbers will be in cells A[p-1]...A[n].”

1. So in both cases, at the end of i(th) iteration of the loop, A[1]…A[i]will be the sorted order.
2. i.e., before the start of the next iteration with the loop variable having value i+1, the subarray A[1…i+1] will be in sorted order so the LI will be ture.

Termination: show that given Initialization and Maintenance proofs, the algorithm will produce the correct answer at termination.

1. initialization showed that the LI will be true before the loop begins (i.e. before the first iteration of the loop with i=2).
2. Maintenance showed that if the LI was true before an iteration, it would still be true before the next iteration.
3. So LI must be true before the second iteration of the loop with i=3 and the third i=4 and so on.
4. The loop end after the iteration i= n, i.e., before the iteration with i=n+1,the LI must be true at that time.
5. LI: before the start of the loop’s iteration with value i, 2<=i<=k+1 , the subarray A[1..i-1] consists of the elements in A[1…i-1] but in sorted array.
6. i.e., before the start start of the loop’s iteration with value i+1, the subarray A[1…i]consists of the elements in A[1…i] but in sorted order.
7. Thus , this algorithm sorts the array correctly and could find the p-largest element as A[p]

**8.** Explain in English, using precise language, why it is true that if f(n)=O(g(n)) then g(n) must be Ω(f(n)). Then explain why the converse is also true.

If f(n)=O(g(n)) means {∃c,n0 , 0≤ f(n) ≤c g(n),∀n≥n0,c>0}

So (1/c)f(n) ≤ g(n)

If we define “a”=(1/c),so there is equation: {∃a,n0 0≤af(n) ≤ g(n),∀n≥n0,a>0} ,which is the definition of g(n)= Ω(f(n))

**9.** Understand this recursive algorithm for computing (bn mod m). Then draw the Recursion Tree of it computing (25 mod 6).

Note: At each node of the tree you should show the inputs and outputs of that execution clearly to receive credit.

Modulo-exponent(b,n,m: integers such that m≥2, n≥0)

1. if n==0 then return 1

2. else

3. temp = Modulo-exponent(b, floor(n/2), m)

4. temp = temp\*temp

5. if n is even then

return temp mod m

6. else

7. return (temp mod m)\*(b mod m) mod m

recursion: bn mod m

Node 1 Input: b=2,n=5,m=6 Output: b=2,n=2,m=6 , return (1mod6)\*(2 mod 6) mod 6 =2

Node 2 Input: b=2,n=2,m=6 Output: b=2, n=1,m=6, temp=1, return 1 mod 6

Node 3 Input: b=2, n=1,m=6 Output b=2, n=0,m=6 ,temp=1,return (1 mod6)\*(2 mod6)mod 6

Node 4 Input: b=2, n=0,m=6 Output: temp=1

Node1(2,5,6),return :2 =25 mod 6=2

↓ ↑

Node2(2,2,6), return :4

↓ ↑

Node2(2,1,6), return :2

↓ ↑

Node3(2,0,6),return temp=1

**10.** Understand how this Bubble Sort algorithm works so that you can modify it to solve the Selection Problem: given an Array [1..n] of distinct numbers and an integer k, 1≤k≤n, return the k-th largest number in the array (the 1-st largest number is the largest number). State the modified algorithm as your answer. Make only the minimum needed modifications to obtain a correct and efficient algorithm. The smallest modification is to add a return statement at the end of the algorithm below "return A[the index of the array cell in which the k-th largest number will be after the entire array is sorted]" and while that gets you a correct algorithm, that is not the most efficient way to modify this sorting algorithm to make it do what you want, and therefore not acceptable.

Bubble-sort (A: Array [1..n] of numbers)

1 i=1

2 while i≤(n–1) modify into \*\*\*\*\*while i≤k

3 j=1

4 while j≤(n–i)

5 if A[j]>A[j+1] then

6 temp=A[j]

7 A[j]=A[j+1]

8 A[j+1]=temp

9 j=j+1

10 i=i+1

add return A[n-k+1]

**11.** A recursive algorithm to compute the exponent xy is given below.

power-recursive(x,y: non-negative integers)

if y == 0 then

return 1

else

if odd(y) then

return power-recursive(x,(y–1))\*x

else

return power-recursive((x\*x),floor(y/2))

end if

end if

Understand how this algorithm works and then convert it into a non-recursive algorithm. Part of the solution is given below. Fill in the blanks. Note: Your iterative algorithm should use the same strategy as the recursive one. I.e., an iterative algorithm that does the obvious and the inefficient - multiplying x with itself (y-1) times - will get 0 points.

power-iterative(x,y: non-negative integers)

result = 1

while\_\_ y!=0\_\_

if \_\_y is even\_ then

result = \_\_result\*x\_\_\_\_\_\_\_\_

y = \_\_\_y-1\_\_\_\_\_\_

end if

x =\_\_\_x\*x\_\_\_\_

y =\_\_\_ y/2\_\_\_\_\_

return result

**12.** What is the best estimate (tightest upper bound) on the number of times the while loop of power-iterative will execute?

power-iterative(x,y: non-negative integers) cost

result = 1 1

while\_\_ y!=0\_\_ 1

if \_\_y is odd\_ then 1

result = \_\_result\*x\_\_\_\_\_\_\_\_ 1

y = \_\_\_y-1\_\_\_\_\_\_ 1

end if 1

x =\_\_\_x\*x\_\_\_\_ lg y

y =\_\_\_ y/2\_\_\_\_\_ lg y

return result

What is the approximate complexity of power-iterative?

Because the times of the execution depends on y, this iterative will end when y==0.

So the times n= lgy

So T(n)=O(lgn)

**13.** Refer slides 27-29 of 7270-06-AnalyzingComplexity-PartI.pdf for this problem. Calculate the complexity of the Merge algorithm using (1) the approximate method and (2) the detailed method. You may assume that n1=n2=n/2. Fill in the table below appropriately. Note: step 3 of the algorithm is not executable.

Approximate analysis:

|  |  |
| --- | --- |
| Step # | Complexity stated as O(\_) |
| 1 | O(1) |
| 2 | O(1) |
| 4 | Complexity of #O(n/2) of executions: |
| 5 | O(n/2) |
| Loop 4-5 | Complexity of entire loop: O(n) |
| 6 | Complexity of # of executions: O (n/2+1) |
| 7 | O(n/2) |
| Loop 6-7 | Complexity of entire loop: O(n+1) |
| 8 | O(1) |
| 9 | O(1) |
| 10 | O(1) |
| 11 | O(1) |
| 12 | Complexity of # of executions: O(n+1) |
| 13 | O(n) |
| 14 | O(n) |
| 15 | O(n) |
| 16 | O(n) |
| 17 | O(n) |
| 13-17 | Complexity of single execution of loop body: O(3n) |
| 12-17 | Complexity of entire loop O((n) |
| 1-17 | Complexity of algorithm: O(n) |

Detailed analysis:

|  |  |  |
| --- | --- | --- |
| Step # | Cost of single execution | # of times executed |
| 1 | c1 | 1 |
| 2 | C2 | 1 |
| 4 | C4 | (n/2+1) |
| 5 | C5 | n/2 |
| 6 | C6 | n/2+1 |
| 7 | C7 | n/2 |
| 8 | C8 | 1 |
| 9 | C9 | 1 |
| 10 | C10 | 1 |
| 11 | C11 | 1 |
| 12 | C12 | n+1 |
| 13 | C13 | n |
| 14 | C14 | t |
| 15 | C15 | t |
| 16 | C16 | n-t |
| 17 | C17 | n-t |

**T(n)=** C1 **+** C2 **+** C4\*(n/2+1)+ C5\* n/2+ C6\* (n/2+1)+ C7\* n/2+ C8+ C9+ C10+ C11+ C12\* (n+1)+ C13\*n+ C14\*t+ C15\*t+ C16\*( n-t)+ C17\*( n-t)= (n/2+1)( C4+ C6)+ n/2(C5+ C7)+t(C14+ C15)+(n-t)( C16+ C17)+ C1 **+** C2+ C8+ C9+ C10+ C11+(n+1) C12+nC13

Simply this equation, then T(n)=O(n)

**14.** Consider the problem of reversing a string provided as an array A[p...r] of characters.

(a) Design an algorithm to this problem that uses the following strategy: iterate through the array from left to right and from right to left, each time swapping the characters in the current leftmost cell and the current rightmost cell, until the middle of the array is reached. State the algorithm precisely using numbered steps that follow the pseudocode conventions that we use.

Reversing-string(A, p, r, k: length of the array, k>0)

1. if k=1
2. return A
3. if k>1;
4. i=1
5. j=k
6. for (i<j)
7. temp=A[i]
8. A[i]=A[j]
9. A[j]=temp
10. i=i+1
11. j=j-1

(b) Design another algorithm to this problem that uses the following strategy: recursive divide and conquer that splits the array into two halves each time, reverses the two halves and then combines the two half-solutions correctly. State the algorithm precisely using numbered steps that follow the pseudocode conventions that we use.

Reversing (A,p,r)

1. n = r-p+1
2. B is a new array with the same length of A
3. k=0
4. while i= p to floor(n/2)
5. A[k]=B[i]
6. k++
7. while i= floor(n/2)+1 to r
8. A[k]=B[i]
9. k++

reversing-string(A,P,r):

1. if p<r
2. n=floor((r-p+1)/2)
3. reversing(A,p,r)
4. reversing-string(A,p,n)
5. reversing-string(A,n+1,r)

(c) Which algorithm is more efficient? Justify your answer by providing an appropriate efficiency analysis that clearly supports your claim.

The second algorithm is more efficient.

The first algorithm will has a loop with swap for (n/2)times ,which the complexity is O(n).

The second algorithm just divide the list averagely for lgn times, so the complexity is O(lg n)

Since O(n)>O(lg n), so the second algorithm is more efficient