CSCI 3104 Spring 2018 Problem Set 2
BAILON Erika (09/28)

---

1. *Solve the following recurrence relations using any of the following methods: unrolling, tail recursion, recurrence tree (include tree diagram), or expansion.*

a) $T(n) = T(n-2) + C \cdot n$ *if* $n > 1$, *and* $T(n) = C$ *otherwise*

$T(n) = T(n-2) + C \cdot n$
$= T(n-4) + C \cdot (n-2) + C \cdot n$
$= T(n-6) + C \cdot (n-4) + C \cdot (n-2) + C \cdot n$
...
$= \sum_{i=0}^{\frac{n}{2}} C \cdot (n-2i)$
$= C \cdot \sum_{i=0}^{\frac{n}{2}} n - 2i$
$= 2 \cdot C \cdot \sum_{i=0}^{\frac{n}{2}} i = 2 \cdot C \frac{(\frac{n}{2})(\frac{n}{2}-1)}{2} = \frac{n^2}{8} - \frac{n}{4} \Rightarrow \mathbf{\Theta(n^2)}$

b) $T(n) = 3T(n-1) + 1$ *if* $n > 1$ *and* $T(1) = 3$

$T(n) = 3T(n-1) + 1$
$= 9T(n-2) + 3 \cdot 1 + 1$
$= 27T(n-3) + 3^2 \cdot 1 + 3 \cdot 1 + 1$
...
$= 3^i T(n-i) + \left(\sum_{k=0}^{i-1} 3^k\right) + 1$
We are looking for the base case when $n - i = 1$. We obtain from there that $i = n - 2$. We also know that by rules of summation that $\sum_{k=0}^{n-1} a \cdot r^k = a \cdot \frac{1-r^n}{1-r}$. Plugging the value of i and the solution for that type of summation, and rearranging, we obtain:
$2 \cdot 3^{n-1} + 1 + \frac{1-3^n}{1-3}$.

$\Rightarrow$Therefore we obtain $\mathbf{\Theta(3^n)}$

c) $T(n) = T(n-1) + 3^n$ *if* $n > 1$ *and* $T(1) = 3$

$T(n) = T(n-1) + 3^n$
$= T(n-2) + 3^{n-1} + 3^n$
$= T(n-3) + 3^{n-2} + 3^{n-1} + 3^n$
...
$= \sum_{i=0}^{i-1} 3^k$ (using the same solution for the geometric summation as in 1.b)
$\Rightarrow T(n) = \mathbf{\Theta(3^n)}$

d) $T(n) = T(n^{1/4} + 1)$ *if* $n > 2$, *and* $T(n) = 0$ *otherwise*

$T(n) = T(n^{1/4} + 1)$
$= T(n^{1/16} + 2)$
$= T(n^{1/64} + 3)$
...
$T(n) = n^{\frac{1}{4^k}} + k$
Using the base case $n^{\frac{1}{4^k}} = 4$
$= n^{\frac{1 \cdot 4^k}{4^k}} = 4^{4^k}$
$= n = 4^{4^k}$
$= log(n) = 4^k$
$= log(log(n)) = k$
$\Rightarrow T(n) = \mathbf{\Theta(log(log(n))}$

[1]

[1]Worked with: George Allison, Eric Oropeza, Michael Merola, Gustav Solis, Krish Dholakiyq, Selena Quintanilla

2. *Consider the following function:*

```
def foo(n) {
    if (n > 1) {
    print( ''hello'' )
    foo(n/3)
    foo(n/3)
}}
```

In terms of the input n, determine how many times is "hello" printed. Write down a recurrence and solve using the Master method.

Adding the number of operations from the code we obtain:
$T(n) = 2(T)(\frac{n}{3}) + 1$ (The +1 comes from *print* function)

The Master Theorem can be used here since our function has the form
$T(n) = a(T)(\frac{n}{b}) + f(n)$ where our $a = 2, b = 3$, and $f(n) = 1$.
To find which case we have, we compare $f(n)$ with $n^{log_3 2}$.
I$n^{log_3 2}$ is larger than $f(n)$.
We fall into the case number 1. of the $MasterTheorem$ as is cited in the CLRS book.

$\Rightarrow \mathbf{\Theta(n^{log_3 2})}$ is the answer.

[2]

[2]Worked with: George Allison, Eric Oropeza, Michael Merola, Gustav Solis, Krish Dholakiyq, Selena Quintanilla

3. *Professor McGonagall asks you to help her with some arrays that are **raludominular**. A raludominular array has the property that the subarray $A[1..i]$ has the property that $A[j] > A[j+1]$ for $1 \leq j < i$, and the subarray $A[i..n]$ has the property that $A[j] < A[j+1]$ for $i \leq j < n$. Using her wand, McGonagall writes the following raludominular array on the board $A = [7, 6, 4, -1, -2, -9, -5, -3, 10, 13]$, as an example.*

a) *Write a recursive algorithm that takes asymptotically sub-linear time to find the minimum element of A.*

```
initialStep(A)
{
    return RecursiveAlgo(A, 0, lengh(A) - 1)
}

RecursiveAlgo(A,L, R)
{
     if(L>R)  {return NIL}
     p = floor((L+R)/2)
     if(A[p] > A[p+1])
     {
         RecursiveAlgo(A,p,R)
     }
     else if (A[p]>A[p-1])
     {
         RecursiveAlgo(A, L, p)
     }
     else
     {
         return A[p]
     }
}
```

b) *Prove that your algorithm is correct. (Hint: prove that your algorithm's correctness follows from the correctness of another correct algorithm we already know.)*

To prove that the algorithm is correct:
Loop invariant: At the start of the ith iteration, for all $j < i$ it is true that $A[j] \neq p$
Initialization. Our base case, by the definition of the *raludominular* array, can be when the array contains 3 elements. This means $n = 3$. Since our $n = 3$, the value of our variable R will be $= 2$ since we are giving to R the value of $lengh(A) - 1$. When we enter the algorithm we see that $L > R$ so we go to our atomic operation of p which will give us 1. Then we enter the first if statement to check the second position of the array (because index 1 is the second position of the array). By definition, again, of the *raludominular* array, we will not find that $A[p] > A[p+1] or A[p] > A[p-1]$. Therefore we will fall into the *else* statement and we will return $A[p]$ which is our minimum value, therefore the algorithm works for the base case.
Maintenance. After our base case, when $n = 3$, we find that $n \geqslant 3. RecursiveAlgo$ will start and divide our array in half grabbing the value of p. Then is it going to check if the value in the array in position $A[p]$ is greater than the value that follows the position p, i.e. if $A[p] > A[p+1]$. If it is greater, it means that the *minimum* value of the array is NOT located in the left side of the array. Therefore, a new range of search is giving to the $RecursiveAlgo$ function to check the right side of the array and it will throw away the left side of the array, in which the *minimum* value is NOT located. This new range of search start from the point p and it ends on the last element of the array. Once we pass the new range of search, we are going to cut the array in half again to give a new value to p, we will check again if the value next to p is greater than the value $A[p]$ and if it is, we will throw away the left side again and set new range of search. The function will keep getting rid of the part of the array where for sure, the *minimum* value will NOT be found. The range of search that is changed each time, makes sure that it grabs the position where the array was cut, that way we have the guarantee that each element of the array will be checked. The same process occurs if the value $A[p]$ is greater than the value before $A[p]$, i.e. if $A[p] > A[p-1]$. The function will keep getting rid of the left half of the array and will continue to change the range and checking the point p with each partition to make sure it goes through the entire array and it gets rid of the partition of the array where the *minimum* is NOT.

Termination. The *RecursiveAlgo* will terminate under 2 circumstances.
1st The function called itself the maximum number of times so it will return NIL.
2nd When the *minimum* value is located before the function reaches its maximum times of recursion, which will return the correct *minimum* value.

c) *Now consider the* `multi-raludominular` *generalization, in which the array contains k local minima, i.e., it contains k subarrays, each of which is itself a raludominular array. Let $k = 2$ and prove that your algorithm can fail on such an input.*

If we were to have a $multi-raludominular$ array, where $k = 2$, it will mean that we will have two *minimum* values. One on each subarray, located in a way that will look something like this:
$A = [n_0...min_1...min_2...n_n]$
When our *RecursiveAlgo* starts, it will cut the array in two. This will create a value of p that will be compared to either the right or the left to decide which side of the array will be thrown away. No matter which side the array decided to throw away, either of the had each a *minimum* value in some position that will never be check because we will get rid of one side of the array since THE FIRST partition. Which will guarantee that we throw away one side where ONE of the *minimum* values was located. Therefore, depending on which side the algorithm chose to go, either way min_1 or min_2 will be thrown away. So the algorithm will fail!

d) *Suppose that $k = 2$ and we can guarantee that neither. local minimum is closer than $n = 3$ positions to the middle of the array, and that the "joining point" of the two singly-raludominular subarrays lays in the middle third of the array. Write an algorithm in sublinear time. Prove that your algorithm is correct, recurrence relation for its running time, and solve for its asymptotic behavior.*

```
ultimateAnsw(A)
{
    n = length(A)-1
    pp = floor(n/2)
    x = minFinder(A,0,pp)
    y = minFinder(A,pp,n)
    if x > y
    {
        return y
    }
    else
    {
        return x
    }
}
RecursiveAlgo(A,L, R)
{
     if(L>R)  {return NIL}
     p = floor((L+R)/2)
     if(A[p] > A[p+1])
     {
         RecursiveAlgo(A,p,R)
     }
     else if (A[p]>A[p-1])
     {
         RecursiveAlgo(A, L, p)
     }
     else
     {
         return A[p]
     }
}
```

To prove that the algorithm is correct:
Loop invariant: At the start of the *ith* iteration, for all $j < i$ it is true that $A[j] \neq p$
Initialization. Our base case, by the definition of the $multi - raludominular$ array, can be when the array contains 5 elements. This means $n = 4$ since our $n = lengh(A) - 1$. When we enter the algorithm we go to our atomic operation of p which will give us 2. We then assigned a variable x to find the minimum value on the left side of the array and then we give the variable y assigned to the function to find the minimum value of the array in the right side. Since we have proved the correctness of the $RecursiveAlgo$ we can guarantee that we will find the *minimum* value on each side of the array and we will not lose them since they are assigned to a variable. After we found the *minimum* value on each side, we compare them to each other and if x is geater then y then we will return y which is the smallest value. If x is not greater than y or is equal, then we will return x. So the algorithm works for the base case.
Maintenance. After our base case, when $n = 4$, we find that $n \geqslant 4.ultimateAnsw$ will start and divide our array in half grabbing the value of pp. Then we will assign one value to check one side of the array and one value to check the other side of the array. The function $RecursiveAlgo$ will keep be calling until a minimum value is find and will return variables to be compared. As we proved the correctness of $RecursiveAlgo$ the function is guarantee to return a minimum value for each side of the array. The parts of the array where the minimun value is not found, will be thrown away with $RecursiveAlgo$.
Termination. The $RecursiveAlgo$ will terminate under 2 circumstances.
1^{st} The function $RecursiveAlgo$ called itself the maximum number of times so it will return NIL. which will cause the $ultimateAnsw$ to return a NIL value and it will terminate.
2^{nd} When the *minimum* value is located,which will return the correct *minimum* value.

[3]

[3]Worked with: George Allison, Eric Oropeza, Michael Merola, Gustav Solis, Krish Dholakiyq, Selena Quintanilla

4. *Asymptotic relations like O, Ω, and Θ represent relationships between functions, and these relationships are transitive. That is, if some $f(n) = \Omega(g(n))$, and $g(n) = \Omega(h(n))$, then it is also true that $f(n) = \Omega(h(n))$. This means that we can sort functions by their asymptotic growth.*

Sort the following functions by order of asymptotic growth such that hte final arrangement of functions $g_1, g_2 \ldots, g_{12}$ satisfies the ordering constraint $g_1 = \Omega(g2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{11} = \Omega(g_{12})$.

n	$n^{1.5}$	$8^{\lg n}$	$4^{\lg *n}$	$n!$	$(\lg n)!$	$(\frac{5}{4})^n$	$n^{1/\lg n}$	$n \lg n$	$\lg(n!)$	e^n	42

Give the final sorted list and identify which pair(s) functions $f(n)$, $g(n)$, if any, are in the same equivalence class, i.e., $f(n) = \Theta(g(n))$. [4]

FASTES to SLOWEST
$n!$
e^n
$(\frac{5}{4})^n$
$(\lg n)!$
$8^{\lg n}$
$n^{1.5}$
$n \lg n = \lg(n!) = \Theta$
n
$4^{\lg *n}$. (It is important to mention that the function $log*$ reaches it maximun value of 5 which means that the maximum value that $4^{\lg *n}$ can have is 4^5)
$n^{1/\lg n} = 42 = \Theta$

[5]

[4] Worked with: George Allison, Eric Oropeza, Michael Merola, Gustav Solis, Krish Dholakiyq, Selena Quintanilla

[5] The notion of sorting is entirely general: so long as you can define a pairwise comparison operator for a set of objects S that is transitive, then you can sort the things in S. For instance, for strings, we use a comparison based on lexical ordering to sort them. Furthermore, we can use any sorting algorithm to sort S, by simply changing the comparison operators $>$, $<$, etc. to have a meaning appropriate for S. For instance, using Ω, O, and Θ, you could apply QuickSort or MergeSort to the functions here to obtain the sorted list.