CSCI 3104 Spring 2018 Problem Set 2
Cole Schlisner (01/30)

---

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

   Each each case, show your work.

   (a) $T(n) = T(n-2) + Cn$ *if* $n > 1$, *and* $T(n) = C$ *otherwise*
   Unrolling:

$$\begin{aligned} T(n) &= T(n-2) + Cn && (1)\\ &= (T(n-4) + Cn) + Cn && (2)\\ &= ((T(n-6) + Cn) + Cn) + Cn && (3)\\ &= (((T(n-8) + Cn) + Cn) + Cn) + Cn && (4)\\ &\dots && (5)\\ &= T(n-2i) + iCn && (6) \end{aligned}$$

   Base Case: $n - 2i = 1 \implies n = 2i + 1 \implies i = \frac{1-n}{2}$
   Given $i = \frac{1-n}{2}$:

$$\begin{aligned} T(n) &= T(n-2i) + iCn && (7)\\ &= T(1) + \frac{1-n}{2}Cn && (8)\\ &= C + \frac{1-n}{2}Cn && (9)\\ &= \Theta(n) && (10) \end{aligned}$$

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

Unrolling:

$$\begin{aligned} T(n) &= 3T(n-1) + 1 && (11)\\ &= 3(3T(n-2) + 1) + 1 && (12)\\ &= 3(3(3T(n-3) + 1) + 1) + 1 && (13)\\ &... && (14)\\ &= 3^i T(n-i) + i && (15) \end{aligned}$$

Base Case: $n - i = 1 \implies i = n - 1$
Given $i = n - 1$:

$$\begin{aligned} T(n) &= 3^i T(n-i) + i && (16)\\ &= 3^{n-1} T(1) + (n-1) && (17)\\ &= 3^{n-1} 3 + n - 1 && (18)\\ &= 3^n + n - 1 && (19)\\ &= \Theta(3^n) && (20) \end{aligned}$$

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

Unrolling:

$$\begin{aligned} T(n) &= T(n-1) + 3^n && (21)\\ &= T(n-2) + 3^n + 3^n && (22)\\ &= T(n-3) + 3^n + 3^n + 3^n && (23)\\ &... && (24)\\ &= T(n-i) + i3^n && (25) \end{aligned}$$

Base Case: $n - i = 1 \implies i = n - 1$
Given $i = n - 1$:

$$\begin{aligned} T(n) &= T(n-i) + i3^n && (26)\\ &= T(1) + (n-1)3^n && (27)\\ &= n3^n - 3^n + 3 && (28)\\ &= \Theta(n3^n) && (29) \end{aligned}$$

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

Unrolling:

$$T(n) = T(n^{(\frac{1}{4})}) + 1 \tag{30}$$
$$= T(n^{(\frac{1}{16})}) + 1 + 1 \tag{31}$$
$$= T(n^{(\frac{1}{64})}) + 1 + 1 + 1 \tag{32}$$
$$... \tag{33}$$
$$= T(n^{(\frac{1}{4^i})}) + i \tag{34}$$

Base Case:

$$n^{(\frac{1}{4^i})} = 2 \tag{35}$$
$$\frac{1}{4^i} lg(n) = lg(2) \tag{36}$$
$$lg(n) = 4^i lg(2) \tag{37}$$
$$4^i = lg(n) \tag{38}$$
$$ilg(4) = lg(lg(n)) \tag{39}$$
$$i = \frac{lg(lg(n))}{2} \tag{40}$$

Given $i = \frac{lg(lg(n))}{2}$:

$$T(n) = T(n^{\frac{1}{4^i}}) + i \tag{41}$$
$$= T(n^{\frac{1}{4^{(\frac{lg(lg(n))}{2})}}}) + \frac{lg(lg(n))}{2} \tag{42}$$
$$= T(2) + \frac{lg(lg(n))}{2} \tag{43}$$
$$= \frac{lg(lg(n))}{2} \tag{44}$$
$$= \Theta(lg(lg(n))) \tag{45}$$

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.

The recursion tree is a binary tree with each node having input $\frac{n}{3}$ of the parent. The recursion stops when the input $n \leq 1$.

Hence the base case is: $\frac{n}{3^i} = 1 \implies i = log_3(n) =$ depth of tree. "Hello" is printed for each node in this tree, and since it is a full binary tree, there are 2^i nodes on the i-th level. Following this, the total number of nodes in the tree excluding the bottom layer (where n = 1) is given by:

$$\sum_{i=0}^{log_3(n)-1} 2^i = 2^{log_3(n)} - 1$$

Given $\epsilon = 1 > 0$:

$$T(n) = 2T(n/3) + 1 \tag{46}$$
$$f(n) = 1 = O(n^{log_3(2)-\epsilon}) \tag{47}$$
$$= O(1) \tag{48}$$
$$\therefore T(n) = \Theta(n^{log_3(2)}) \tag{49}$$

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.*

```
def globmin(A, i):
    if (A[i] > A[i+1]):
        return globmin(A, i+1)
    else return A[i]
```

   (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.)*
   Assuming 0-based indexing:

   Invariant: $A[i] < \{x | x \in A[0..i-1]\} \vee A[0..i-1] = \varnothing$

   Initialization: i = 0, so A[0..-1] is the empty set and the loop invariant holds.

   Maintenance: if $(A[i] > A[i+1])$, the loop invariant holds and the function calls itself with the next index. Otherwise, it returns A[i], as $A[i-1] > A[i] < A[i+1]$.

   Termination: once min() returns: $A[i-1] > A[i] < A[i+1]$, and the loop invariant still holds because i was only incremented if $A[i] < A[i-1]$. This means A[i] is the minimum element in the subarray A[0..i]. Combining this fact with the definition of the 'raludominular' array, A[i] must also be the minimum element in the sub array A[i..n]. Thus A[i] must be the minimum element in A[0..n] = A.

(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.*

The algorithm will return as soon as it finds an index i such that

$$A[i-1] > A[i] < A[i+1]$$

This will be the first local minimum in the first subarray. This will not always be the global minimum, so the algorithm is incorrect for this type of input.

(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. Now write an algorithm that returns the minimum element of A in sublinear time. Prove that your algorithm is correct, give a recurrence relation for its running time, and solve for its asymptotic behavior.*

```
def globmin(A, i):
    while (A[i+1] < A[i]):
      ++i
    if (i < (A.length-(A.length/3))):
      return min(A[i], globmin(A, (A.length-(A.length/3)))
    return A[i]
```

For the while loop on the first two lines of globmin():
Let l = the initial value of i before entering the loop.
Let m = the end of the current raludominular subarray of A
Invariant: $A[i] < \{x | x \in A[0..i-1]\} \vee A[0..i-1] = \varnothing$
Initialization: i = 0, so A[0..-1] is the empty set and the loop invariant holds.
Maintenance: if $(A[i+1] < A[i])$, i is incremented preserving the loop invariant for the next iteration.
Termination: the loop exits when $A[i+1] \geq A[i]$, and the loop invariant still holds. Furthermore: $A[l..i-1] > A[i] < A[i+1]$. Combining this with the definition of 'raludominular' arrays means A[i] is the local minimum for the current subarray A[l..m].

Now, if we had just found the first local minimum, the if statement would return True since A[i] is necessarily not within the last third of A. From here, the minimum of A[i] (the first local minimum) and the result of a second call to globmin() starting at the last third of A is returned. We just proved that we will find the second minimum in this second call, and the if statement will now return False - simply returning A[i] (now our second minimum) to be compared with our first minimum. The result of this comparison is returned, meaning we have returned the minimum of our two local minima - the global minimum.

Running time:
Because the only costs of this algorithm are the while loop and the if statement evaluation (a constant +1 to each recursion), we really just need to calculate how many times the loop runs to find the cost. The loop only runs for each beginning section A[0..i] of the current raludominular subarray. For the first subarray, this is at most $(\frac{n}{2}) - (\frac{n}{3}) = \frac{n}{6}$ (the local minimum in this subarray cannot be closer than $\frac{n}{3}$ to the center of the array at $\frac{n}{2}$). For the second array, we begin iterating at $\frac{n}{3}$ from the end, so this is at most $\frac{n}{3}$.

Thus, the maximum number of times i can be incremented during a complete run is $\frac{n}{6} + \frac{n}{3} = \frac{n}{2}$.

A recurrance relation for this algorithm would be:

$$T(n) = \begin{cases} T(n/3) + \frac{n}{6} + 1, & \text{if } n < A.length - (A.length/3) \\ 1, & \text{otherwise} \end{cases}$$

However a relation is not needed. Since we know the maximum possible aggregate value $(\frac{n}{2})$ of the driving term, it follows that $T(n) = O(\frac{n}{2} + 2) = O(n)$.

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.*[1]

   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., $f(n) = \Theta(g(n))$.

   From highest order to lowest order:

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

$$42 = \Theta(x^{\frac{1}{lg(n)}}) \tag{50}$$

[1] 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.

References Used

---

1. CLRS
2. https://courses.engr.illinois.edu/cs173/fa2009/Lectures/lect_22.pdf
3. https://en.wikipedia.org/wiki/Iterated_logarithm