CS 677: Assignment 1

Terence Henriod

February 18, 2014

Abstract

In this assignment, methods for solving recurrences are practiced. Methods considered include iteration, substitution, the recursion tree, and the "Master's" method.

1. Consider the following algorithm:

Algorithm *Enigma*$(A(0 \ldots n-1, 0 \ldots n-1))$
$(*$ // Input: a matrix A[0 ... n-1, 0 ... n-1] of integer numbers $*)$
1. **for** $i \leftarrow 0$ **to** $n-2$
2. **for** $j \leftarrow i+1$ **to** $n-1$
3. **if** $A[i,j] \neq A[j,i]$
4. **return false**
5. **return true**

(a) What does this algorithm do?

Solution: This algorithm checks to see if all of the elements below the main diagonal of the matrix are equal to the element that is transposed from the original element in a matrix, i.e. symmetric about, but not including, the diagonal. If the matrix is symmetric in this manner, this is indicated by the returning of true after checking all the elements. If any pair of elements are detected to be asymmetric, the check halts and false is returned.

(b) Compute the running time of this algorithm.

Solution: Outline the number of times each primitive operation (primitive math operations, comparisons, etc.) occurs, consider primitive operations to have constant running time. Note that t_x indicates the number of times a statement is executed for a given value of x. Then sum the cost multiples to get the running time of the algorithm.

Numbers refer to the steps of the algorithm above.

Step Number	Cost	Times Executed (Best Case)	Times Executed (Worst Case)
1.	c_1	1	$\sum_{i=0}^{n-2} 1 = n-1$
2.	c_2	1	$\sum_{i=0}^{n-2} t_i$
3.	c_3	1	$\sum_{i=0}^{n-2} t_i - 1$
4.	c_4	1	0
5.	c_5	0	1

In the "best" case scenario, the very first pair of elements A$[i,j]$ and A$[j,i]$ are not equal, thus the first for-loop check will only execute once, the inner for-loop check will only execute once, and the if statement will only execute once, and of course the return statement will return once. In this case, $t_i = 1$, but this will make little difference overall:

$$\begin{aligned} T(n) &= c_1(1) + c_2(1) + c_3(1) + c_4(1) + c_5(0) \\ &= c_1 + c_2 + c_3 + c_4 \\ &= c_a(\text{an arbitrary constant}) \\ &= \Theta(1) \end{aligned}$$

In the "worst" case, both loops will execute the maximum number of times because each pair of elements tested will be checked before true is finally returned. In this case the value for t_i will be:

$$\begin{aligned} t_i &= \sum_{j=i+1}^{n-1} 1 \\ &= ((n-1)-(i+1))+1 \\ &= (n-i-2)+1 \\ &= n-i-1 \end{aligned}$$

So the resulting time cost is:

$$\begin{aligned} T(n) &= c_1(\sum_{i=0}^{n-2} 1) + c_2(\sum_{i=0}^{n-2} t_i) + c_3(\sum_{i=0}^{n-2} t_i - 1) + c_4(0) + c_5(1) \\ &= c_1(n-1) + c_2(\sum_{i=0}^{n-2} n-i-1) + c_3(\sum_{i=0}^{n-2} (n-i-1)-1) + c_5 \\ &= c_1(n-1) + c_2(\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1) + c_3(\sum_{i=0}^{n-2} n-i-2) + c_5 \\ &= c_1(n-1) + c_2((n-1)(n) - (\frac{(n-1)n}{2}) - (n-1)) + c_3(\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 2) + c_5 \\ &= c_1(n-1) + c_2((n^2-n) - (\frac{n^2-n}{2}) - (n-1)) + c_3((n-1)(n) - (\frac{(n-1)n}{2}) - 2(n-1)) + c_5 \\ &= c_1(n-1) + c_2(\frac{n^2-n}{2} - (n-1)) + c_3(\frac{n^2-n}{2} - (2n-2)) + c_5 \\ &= c_1(n-1) + c_2(\frac{n^2-3n+2}{2}) + c_3(\frac{n^2-5n+4}{2}) + c_5 \\ &= an^2 + bn + c, \text{ for sufficient a, b, c} \\ &= \Theta(n^2) \end{aligned}$$

2. Solve the following recurrences using a method of your choice:

(a) $T(n) = 4T(\frac{n}{3}) + n^2$

Solution: One method to use is the "Master's Method." To use this method, we must put the expression into the form:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

and then, based on which of three cases $T(n)$ and $f(n)$ fall into, we can make our decision:

Case 1: if $f(n) = \mathrm{O}\big(n^{\log_b a - \epsilon}\big)$ for some $\epsilon > 0$, then $T(n) = \Theta\big(n^{\log_b a}\big)$

Case 2: if $f(n) = \Theta\big(n^{\log_b a}\big)$ then $T(n) = \Theta\big(n^{\log_b a} \lg n\big)$

Case 3: if $f(n) = \Omega\big(n^{\log_b a + \epsilon}\big)$ for some $\epsilon > 0$,
and if $a * f(\frac{n}{b}) \leq c * f(n)$ for some $c < 1$ and all sufficiently large n,
then $T(n) = \Theta\big(f(n)\big)$

By applying the Master's method to this recurrence, we have $a = 4$, $b = 3$, and $f(n) = n^2$, giving:

$$T(n) = 4T(\frac{n}{3}) + n^2 \rightarrow n^{\log_3 4} \approx n^{1.26} \rightarrow f(n) = \Omega\big(n^{\log_3 4}\big)$$

Thus, the recurrence falls under Case 3. Since it falls under case 3, we must ensure that the regularity condition is met:

$$\begin{aligned} a * f(\frac{n}{b}) &\leq c * f(n), \text{ for all } c > 1 \\ (4) f(\frac{n}{(3)}) &\leq c * f(n) \\ 4 * (\frac{n}{3})^2 &\leq c * (n)^2 \\ 4(\frac{n^2}{9}) &\leq cn^2 \\ \frac{4}{9}n^2 &\leq cn^2, \forall c \text{ s.t. } \frac{4}{9} \leq c < 1 \end{aligned}$$

The regularity condition holds, so: $T(n) = \Theta\big(n^2\big)$

(b) $T(n) = T(n-1) + 5$

Solution: Using the "tree" method, we find out how many times the recurrence will execute before reaching its base case (problem size of 1), and count the cost of each recursive execution.

In this problem, there isn't really much of a "tree" since the recurunce only takes one path, but we can still use the method.
The cost of one execution is $5 + T(n-1)$, which is a cost of 5 per recurrence execution plus a constant.
Next we must find how many times i the recurrence will execute. Note that each time the recurrence executes, the problem size decreases by one until the problem size is one. So the total number of times i the recurrence will execute is:

$$\begin{aligned} n - 1i &= 1 \\ -i &= 1 - n \\ i &= n - 1 \end{aligned}$$

Assuming that the running time of $T(1)/T(0)$ is constant, putting it all together, we get:

$$T(n) = 5i = 5(n-1) = \Theta\big(n\big)$$

3. Consider the following recursive algorithm for computing the sum of the first n cubes:

$$S(n) = 1^3 + 2^3 + \ldots + n^3$$

Algorithm $S(n)$
$(*$ // Input: A positive integer $n *)$
$(*$ // Output: The sum of the first n cubes $*)$
1. **if** $n = 1$
2. **return** 1
3. **else**
4. **return** $S(n-1) + (n * n * n)$

(a) Write and solve a recurrence relation for the number of multiplications made by this algorithm and solve it.

Solution: Assuming that multiplications are primitive operations, we have

$$S(n) = S(n-1) + 4$$

because the "problem size" decreases by one with each recursive call, and the cost incurred by each recursive call (other than the next recursive call) is one comparison, two multiplications and one addition.

(b) How does this algorithm compare with the straightforward non-recursive algorithm for computing this function?

Solution: Using the "tree" method, we can determine the running time for the recursive method. It clearly has a cost of 4 per recursive call, except for in the base case where it only has a cost of 1 comparison and the return of 1, i.e. a constant cost per recursice call. Since the "problem size" is only reduced by one with each call, and assuming that it will take i calls to solve the problem, we find that:

$$\begin{aligned} n - 1i &= 1 \\ -i &= 1 - n \\ i &= n - 1 \end{aligned}$$

$n-1$ calls are needed to resolve the problem. Thus, the recursive algorithm has running time

$$S(n) = ci = c(n-1) = \Theta\big(n\big)$$

As for the iterative algorithm, it would look like:

Algorithm $S(n)$
$(*$ // Input: A positive integer $n *)$
$(*$ // Output: The sum of the first n cubes $*)$
1. **for** $S \leftarrow 0, i \leftarrow 0$**to**$n$
2. **do** $S \leftarrow S + i^3$ **return** S

Simple cost analysis would indicate that the loop runs $n+1$ times, and the actions inside the loop run n times, giving

$$S(n) = c_1(n+1) + c_2(n) + c_3(1) = an + b = \Theta\big(n\big), \text{ for sufficient } a \text{ and } b$$

Thus the running times are very comparable. In practice, the iterative algorithm would actually run faster due to the lack of overhead associated with recursive calls, but the two algorithms do have the same order of growth.

4. Consider the following recursive algorithm:

Algorithm $Min(A, l, r)$
(∗ // Input: An array A[0 . . . n-1] of integer numbers ∗)
(∗ // The initial call is *Min(A, 0, n-1)* ∗)

```
1.  if l = r
2.          return A[l]
3.    else  temp1 ← Min(A, l, ⌊(l+r)/2⌋)
4.          temp2 ← Min(A, ⌊(l+r)/2⌋ + 1, r)
5.          if temp1 ≤ temp2
6.                  return temp1
7.            else
8.                  return temp2
```

(a) Write the recurrence relation for the above algorithm.

Solution: One might write the recurrence relation as follows:

$$M(n) = 2M(\frac{n}{2}) + c$$

because does make some comparisons and a return with each recursive call, but we can also see that when further recursive calls are made, the problem size is halved by moving one of the indices to the median value of the array given for the current recursive call and passing the smaller array to the next recursive call. Each half of the given array is given to one of the two recursive calls. It should be noted that all calls to the function/algorithm will generate sub-trees of equal height/size, but it can be reasonably said that the actual cost of the algorithm will be less than or equal to the running time that is found by assuming all recursive calls generate subtrees of precisely the same height/size. It should be noted that these "sub-tree" imbalances occur when integer division of an odd number creates a pair of unequal halves.

(b) Solve the recurrence obtained in part (a).

Solution: Using the tree method (appropriately this time), we can solve the running time of this recurrence. The cost associated with each recursive call is a constant plus two recursive calls (with the exception of base cases, which have only a constant cost). Then we need to compute the number of times the algorithm will run. This number, i, is found by solving the following equation:

$$\begin{aligned} \frac{n}{2^i} &= 1 \\ 2^i &= n \\ i &= \lg n \end{aligned}$$

So, to compute the total cost/running time of the algorithm, we have:

$$M(n) = ci = c(\lg n) = \Theta\big(\lg n\big)$$

5. Consider the following algorithm:

Algorithm *Mystery*(n)
($*$ // Input: A nonnegative integer n $*$)
1. $S \leftarrow 0$
2. **for** $i \leftarrow 1$ **to** n
3. **do**
4. $S \leftarrow S + i * i$
5. **return** S

(a) What does this algorithm compute?

Solution: This algorithm computes the sum of all the squares with roots 1 to n. Every number from 1 to n is visited (step 2), each of those numbers are squared (step 4), and the resulting square is added to the running total (step 4).

(b) Compute the running time of this algorithm.

Solution: We can begin a cost anaysis by laying out the cost and frequency of each step in a table:

Step Number	Cost	Times Executed
1.	c_1	1
2.	c_2	$n+1$
3.	—	—
4.	c_3	$n*3$
5.	c_4	1

Which, using the values in the table to compute the running time, gives:

$$\begin{aligned} T(n) &= c_1(1) + c_2(n+1) + c_3(3n) + c_4(1) \\ &= an + b \\ &= \Theta(n) \end{aligned}$$