Homework 1 – Answers to Questions

Jade Pearl

CMSC 451-7380

Professor Dennis Didulo

**Question 1**:

//Precondition n > 0  
int square(int n) {

if (n == 1)

return 1;

else

return (2 \* n – 1) + square(n – 1);  
}  
//Postcondition returns n^2

Proof by induction:

Base case (*n* = 1)

1 is returned when n = 1. 12 = 1. This is true for the base case.

Inductive case: Assume that this is true for *n = k –* 1. Show that this is true for *n = k* where *n* > 1

square(k) = square(k-1) + 2k – 1 From program

= (k-1)2 + 2k – 1 Inductive hypothesis

= k2 – 2k + 1 + 2k – 1 Algebra

= k2 Algebra

By proof of induction, we have proven that this recursive function is correct because the precondition and postcondition are both true.

**Question 2**:

To find at what point algorithm 2 becomes more efficient than algorithm 1, we first set them equal to each other:

3*n*2 + 9 = 51*n* + 17

Use rules of algebra to make the right-hand side zero:

3*n*2 - 51*n* – 8 = 0.

Plug this equation into the quadratic formula and solve it.

*n*1, *n*2 =

*n*1, *n*2 =

*n*1, *n*2 =

The square root of 2697 is approximately 51.933.

*n*1 =

*n*2 =

In order for algorithm 2 to be more efficient than algorithm 1, *n* must be greater than or equal to 18.

**Question 3**:

To count the number of swaps in the worst case as a function of *n* which is the length of the array, we shall use the summation evaluation. Since there are nested loops and statements, a nested summation may be of help in this scenario.

*f(n)* =

This shows that the number of swaps in the worst-case scenario can be represented as the sum of the first (n-1) natural numbers.

This is equivalent to

*f(n)* =

Once simplifying this summation, we get swaps.

**Question 4**:

Consider that *n* is the size of the array and the execution time will be dependent on the number of calls to the recursive sumEvenElements function (not to be confused with the single-line sumEvenElements function which only returns sumEvenElements(array, 0) that calls the recursive version).

First, we must find the initial condition. The initial condition is when index i reaches a value greater than or equal to array.length (or *n*). If this condition is met in the program, zero is returned and no further calculations are carried out.

t(0) = 1 is the initial condition because there is at least one call to the recursive sumEvenElements function and returns without any additional recursion.

After the initial condition comes the recurrence equation which is derived from the recursion in the sumEvenElements method. The function processes elements at each even subscript. It processes an element and recursively calls itself with the index i + 2. The recurrence equation will analyze the time of processing an array of length *n* using t(*n*):

t(*n*) = t(*n/2*) + O(1). t(*n*/2) is the time taken to process half of the array because we are analyzing at every even element. The O(1) represents the other actions: addition, comparison, and recursive call.

The last step is to solve this equation. In a previous course, I had learned that this can be done with the Master theorem. Specifically in this case, we can use case 2 of the master theorem.

Find a, b and f(*n*) in t(n) = at(*n/b*) + f(*n*)

a = 1, b = 2, f(*n*) = O(1)

Find log­b(a) 🡪 log21 = 0

Compare f(*n*) to nlogba

f(*n*) = O(1), nlogba = 1 so f(*n*) < nlogba

This applies case 2 of the master theorem since f(*n*) < nlogba

t(*n*) = Θ(nlogba \* log(k+1)n) where k is not specified but greater than or equal to 0.

This simplifies to t(*n*) = Θ(log n).

This concludes that the evaluation time of sumEvenElements in worst case is logarithmic in length *n* of the array.

Resources

*Advanced master theorem for divide and conquer recurrences*. (2018, April 17). GeeksforGeeks. https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/

*Bubble sort complexity calculation, unsure how it went from one step to another.* (n.d.). Mathematics Stack Exchange. Retrieved October 25, 2023, from https://math.stackexchange.com/questions/1642029/bubble-sort-complexity-calculation-unsure-how-it-went-from-one-step-to-another

Jarc, D. (n.d.). *Design and Analysis of Algorithms - Proof of Correctness* [Review of *Design and Analysis of Algorithms - Proof of Correctness*]. Retrieved October 20, 2023, from https://learn.umgc.edu/d2l/le/lessons/916484/topics/31312395  
UMGC course learning resources.

‌*Summation Formulas - What Are Summation Formulas? Examples*. (n.d.). Cuemath. <https://www.cuemath.com/summation-formulas/>