COMP26120
Academic Session: 2022-23

Worksheet 3: Complexity of Recursive Programs
Model Solution

This worksheet is assessed.

Learning Objectives

At the end of this worksheet, you should be able to:

1. Apply standard manipulations related to exponentials, logarithms, factorials, and sums (this is revised material from the first year).
2. Explain the divide-and-conquer paradigm and write a recurrence equation from a given algorithm.
3. Solve recurrences using the substitution, iteration, and master methods.
4. Describe various examples to analyse divide-and-conquer algorithms and how to solve their recurrence equations.
5. Compute loop invariants for recursive functions.

Introduction

This worksheet is about analysing the correctness and complexity of recursive algorithms. In particular, this coursework covers questions related to the substitution method, changing variables, derive recursive equations, analysis of recursive algorithms, the master method, and loop invariants.

Suggested Reading. This worksheet is based on Sections 2.1, 2.3, 4, 4.3, and 4.5 of "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein, ISBN 978-0-262-03384-8, MIT Press 2009 (22 pages). Section 2.1 introduces loop invariants to understand why an algorithm is correct. Section 2.3 describes the divide-and-conquer approach applied to the merge sort algorithm and discusses loop invariants. The introduction of Section 4 describes the general equation of the divide-and-conquer strategy, while Sections 4.3 and 4.5 present the substitution method and master method used to solve recurrence equations.

Exercises

Exercise 1 (Recursion Invariant). Computer science has many algorithms that require writing recursive functions to achieve efficiency. However, recursive algorithms are hard to understand, which thus requires the computation of a recursion invariant to understand its correctness. So, recursion invariants represent another application of proof by induction. In this respect, consider the following recursive function to find 3^n for some nonnegative integer n. What is the recursion invariant of this algorithm? Note that you should give at least some informal discussion concerning that invariant computation, e.g., considering the initialisation, maintenance, and termination.

```
function EXP((n))
    if $n = 0$ then return 1
    else if $n \mod 2 = 0$ then
        x = exp(n/2) return x * x
    else $n$ is (odd)
        x = exp((n-1)/2) return 3 * x * x
    end if
end function
```

We compute the following recursion invariant to prove the correctness of this recursive algorithm:

Invariant: $exp(k)$ returns 3^k at each call. In particular, this recursion invariant holds at each if-statement of the above algorithm, as described below.

Base case: When $k = 0$, $exp(k)$ returns $1 = 3^0$.

Maintenance: We split the maintenance case into two sub-cases, which include k is even, or k is odd. If k is even, then the algorithm sets x to $exp(k/2)$; the recursion invariant becomes $3^{k/2}$. The algorithm returns $x * x$, which is $3^{k/2} * 3^{k/2} = 3^k$. Similarly, if k is odd, then the algorithm sets x to $exp((k-1)/2)$; the recursion invariant becomes $3^{(k-1)/2}$. The algorithm returns $3 * x * x$, which is $3 * 3^{(k-1)/2} * 3^{(k-1)/2} = 3^k$.

Termination: $exp(n)$ gives 3^n at the top level of the recursive call, which is our recursion invariant.

Exercise 2 (Substitution Method). Use the substitution method to show that the solution of $T(n) = T(\lceil \frac{n}{2} \rceil) + 1$ is $O(log_2 n)$. Note that when $n = 1$, we have $T(1) = 1$. Note further that, the ceiling function $\lceil x \rceil$, when applied to the argument x, computes the least integer that is greater than or equal to x.

Solution:
Base case: show that the inequality holds for some n sufficiently small.
If $n = 1$, then $T(1) \leq c \times log_2 1 = 0$.
If $n = 2$, then $T(2) \leq c \times log_2 2 \leq c$ (base case holds when $c \geq 2$).
Step case: we will show $T(n) \leq c\, log_2 n$, which will imply $T(n) = O(log_2 n)$

$$\begin{aligned} T(n) &= T(\left\lceil \frac{n}{2} \right\rceil) + 1 \\ &\leq c(log_2(\left\lceil \frac{n}{2} \right\rceil)) + 1 \end{aligned}$$

Case 1: Assume n is even, then we obtain:

$$\begin{aligned} T(n) &\leq c(log_2(\frac{n}{2})) + 1 \\ &= c(log_2 n - log_2 2) + 1 \\ &= c(log_2 n - 1) + 1 \\ &= c\, log_2 n\ - c + 1 \\ &\leq c\, log_2 n \end{aligned}$$

Case 2: Assume n is odd, then we obtain:

$$\begin{aligned} T(n) &\leq c(log_2(\frac{n}{2} + 1) + 1) \\ &= c(log_2(\frac{n+2}{2})) + 1 \\ &= c(log_2 (n+2) - log_2 2) + 1 \\ &= c(log_2 (n+2) - 1) + 1 \\ &= c\, log_2 (n+2)\ - c + 1 \\ &\leq c\, log_2 (n+2) \end{aligned}$$

For case 2, our inductive assumption turns out not to be strong enough to prove the detailed bound. Even if we revise the guess by subtracting or adding a lower-order term, the math does not go through.

Exercise 3 (Changing variables). Solve the following recurrence by changing variables and then applying the master method.

$$T(n) = \begin{cases} 2T(n^{0.25}) + 1, & \text{if } n > 1 \\ 1, & \text{if } n = 1. \end{cases}$$

Solution:

This recurrence needs to be manipulated in order to apply the master method. In particular, the following substitution should be done $m = log_4\, n$, which yields $4^m = n$ and $T(4^m) = S(m)$. Applying the transformation $T(4^m) = S(m)$ yields: $T(4^m) = 2T(4^{m/4}) + 1$ and $S(m) = 2S(m/4) + 1$. The original recurrence can thus be rewritten as:

$$S(m) = \begin{cases} 2S(m/4) + 1, & \text{if } m > 0 \\ 1, & \text{if } m = 0. \end{cases}$$

Applying the master method to the above equation yields $T(n) = \Theta(\sqrt{log_4 n})$. In particular, we apply the first case of the master method, where $a = 2$, $b = 4$, $f(n) = 1$. Thus, we obtain $T(n) = \Theta(\sqrt{m})$. If we replace m by $log_4\, n$, then we get $T(n) = \Theta(\sqrt{log_4 n})$. Note that, we could generalise this solution and make $m = log_k\, n$, then $n = k^m$. Thus, we would obtain $T(k^m) = 2T(k^{(m/4)}) + 1$. The base of the logarithm disappears in $S(m) = 2S(m/4) + 1$ and has no impact on the application of the master method. Thus, we would obtain $\Theta((log_k(n)^{1/2})$, where $k > 1$.

Exercise 4 (Complexity Analysis). Design a *recursive* binary-search algorithm to find an arbitrary number k in a list of n integers, which breaks the instance into two parts: one with 1/3 of the elements and another one with 2/3. You must consider that the list is sorted and present your *recursive* algorithm using pseudocode. Compute the complexity and make a comparative analysis with the traditional binary-search algorithm (which breaks the instance in half). You should use one of the methods introduced in this course to compute the complexity but it is up to you to select which one.

Solution:

The difference between the traditional and proposed binary-search algorithms is the way they break their instances. In the traditional binary-search algorithm, the instances given to the recursive calls always have approximately half of the size of the original instance. In the proposed binary-search algorithm, due to the division factor being 3, two instances will be generated: one with n/3 and another one with 2n/3. The traditional binary search can be represented as:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ T(n/2) + 1, & \text{if } n > 1. \end{cases}$$

which has complexity $T(n) = O(lg\ n)$.
The proposed binary-search algorithm can be represented as:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ T(n - (n/3)) + 1 = T(2n/3) + 1, & \text{if } n > 1. \end{cases}$$

which has complexity $T(n) = O(lg_{3/2}\ n)$.

The student may argue that $O(lg\ n) = O(lg_{3/2}\ n)$ as the base is a constant scaling factor; they must present this argument, simply stating the complexity of the proposed binary-search algorithm is $O(lg\ n)$ is insufficient. Note that the substitution, iteration or master method can be used to compute the complexity of these recurrences.

Exercise 5 (Recurrence). John has developed three different recursive algorithms to solve a particular problem at BoostCode UK Ltd. John needs to know which recursive algorithm performs faster on the same machine for any input value n. The recurrence equation of each algorithm developed by John is given below:

(A) $T(n) = 2T(n/2) + n^4$.

(B) $T(n) = T(7n/10) + n$.

(C) $T(n) = 2T(n/4) + \sqrt{n}$.

Which solution is the fastest one? You can use the Θ-notation to express asymptotic bounds for $T(n)$ for each recurrence above. You can also assume that $T(n)$ is constant for $n \leq 2$. In summary, your task is:

1. Solve the recurrence equations using your preferred method (e.g., master method).
2. Express your solution using the Θ notation, i.e., make your bounds as tight as possible, and justify your answers.
3. Compare the running time of each solution to determine which one is the fastest.

Solution:

(A) By Master Method, $T(n) \in \Theta(n^4)$.
(B) By Master Method, $T(n) \in \Theta(n)$.
(C) By Master Method, $T(n) \in \Theta(n^{1/2}\, log_2\, n)$.
Solution C is the fastest one for sufficiently large n.