CS 325 Assignment 3

Neale Ratzlaff

19 April 2015

1 Problem 1

1.1 Question:

Give the asymptotic bounds for T(n) in each of the following recurrences. Make your bounds as tight and justify your answers

a: $T(n) = T(n-2) + n$

b: $T(n) = T(n-1) + 3$

c: $T(n) = 2T(\frac{n}{4}) + n$

d: $T(n) = 4T(\frac{n}{4}) + n^2\sqrt{n}$

1.2 Solution:

a: a can be solved with the iteration method. The method iterates over the list and subtracts two for eah iteration. After iterating out the recurrence for some times it turns into

$$\begin{aligned} &T(n) = T(n-8) + (n-6) + (n-4) + (n-2) \\ &T(n) = T(n-(2i+2)) + (n-2i) \\ &\text{If we assume that } i = n \\ &T(n) = O(n^2) \end{aligned} \tag{1}$$

b: b can be solved with the iteration method as well. This one is easier once it is recognized that the dominating term in the recurrence is the $T(n-1)$ term, as the constant's effect is negligible. We can see that the recurrence runs for a total of $\frac{n}{2}$ times. Giving a complexity of cn, or

$$T(n) = \Theta(n)$$

c: c can solved with the master theorem

$$T(n) = \begin{cases} aT(\frac{n}{b}) + n^c & : n > 1 \\ d & : n = 1 \end{cases}$$

Where $a = 2, b = 4, c = 1$ giving a complexity of $\Theta(n^c)$. C therefore has a complexity

$$\Theta(n)$$

d: With a slight change to the master theorem, f(n) can be tested to see if it is the largest term asymptotically. c can solved with the master theorem

$$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n) & : n > 1 \\ d & : n = 1 \end{cases}$$

$f(n)$ is $\Omega(n^{\log_b a+e})$ so test f(n) to see that

$$n^2\sqrt(n) = \Omega(n^{1.1})$$

According to the master theorem, this allows us to test for the dominating term in the recurrence.

$$\begin{aligned}
&af(\frac{n}{b}) \leq cf(n) \\
&4(\frac{n}{b})^2 \leq cn^{2.5} \\
&4(\frac{n^{2.5}}{32}) \leq cn^{2.5} \\
&(\frac{n^{2.5}}{8}) \leq cn^{2.5} \\
&\text{with solution } c = \frac{1}{8} \leq 1 \\
&T(n) = \Theta(n^{2.5})
\end{aligned} \tag{2}$$

2 Problem 2

2.1 Question:

Suppose you are choosing between the following three algorithms:

a: Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

b: Algorithm B solves problems of size n by recursively solving two subproblems of size (n -1) and then combining the solutions in constant time.

c: Algorithm C solves problems of size n by dividing them into nine subproblems of size $\frac{n}{3}$, recursively solving each subproblem, and then combining solutions in $O(n^2)$ time.

2.2 Solution:

- Algorithm A can be solved with the master theorem where $a = 5, b = 2, c = 0$ giving
$$\log_b a \geq c \rightarrow \Theta(n^{\log_2 5})$$

- Algorithm B can be solved with iteration. Expanding out the recurrence leads to something resembling
$$T(n) = (2(2^i)T(n-i) + ic$$
If we assume that n here = 2^k, then the recurrence comes out have linear time, or
$$T(n) = O(n)$$
This should be expected because if we logically examine the recurrence it can be seen that the recurrence iterates simply generates two subproblems, n times. Which should intuitively mean a complexity of $2n$, or aysymptotically, $O(n)$

- Algorithm C can also be solved with the master theorem where $a = 9, b = 3, f(n) = n^2$ This requires solving a recurrence to see which term dominates, $f(n)$, or $9T(\frac{n}{3})$

$$\begin{aligned}
&f(n) = \Omega(n^{log_b a + e}) for e > 0 \\
&n^2 = \Omega(n^{1.1}) \\
&af(\frac{n}{b}) \leq cf(n) \\
&9T(\frac{n}{3} \leq cn^2 \\
&9(\frac{n}{3}) \leq cn^2 \\
&n^2 \leq cn^2 for c \geq 1
\end{aligned} \tag{3}$$

Because c here is equal to one to make the expressions equal, the equation holds and the complexity of algorithm C is

$$\Theta(n^2)$$

3 Problem 3

3.1 Question:

How many lines as a function of n (in Θform), does the following PHP function print? Write a recurrence and solve it. You may assume n is a power of 2.

```
function foo(n) {
  if($n > 1)
    echo "Still Printing <br>";
      foo(n/2);
        foo(n/2);
  } else {
      return 1;
  }
}
```

3.2 Solution:

The script makes two recursive calls with size $\frac{n}{n}$, it also makes two constant time comparison, with at most two more constant time operations following them. The recurrence is given by

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

If done with iteration, and assuming that $2^k = n$, it is found that

$$T(n) = (2^k - 1) + 2^k T(\frac{2}{2^k})$$
$$T(n) = (n - 1) + nT(1)$$
$$T(n) = \Theta(n)$$

This answer can be verified with the master theorem, where

$$T(n) = \Theta(n^{\lg 2}) = \Theta(n)$$

4 Problem 4

4.1 Question:

The ternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third. Write pseudo-code for the ternary search algorithm, give the recurrence for the ternary search algorithm and determine the asymptotic complexity of the algorithm. Compare the running time of the ternary search algorithm to that of the binary search algorithm.

4.2 Solution:

```
function tenery_search(list A, x):

  if length(A) <= 1
    if A = x return true
      else return false

  else
    if x > element at length(A) * 1/3
      if x > element at length(A) * 2/3
        b = top third of A
      else:
        b = middle third of A
    else:
      b = first third of A

  return tenery_search(b, x)
```

The binary search recurrence is given by

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

The tenery search is much the same, but requires a couple more comparisons each with complexity $O(1)$. Since tenery search differs significantly in the size of the fractioned list, because the list is broken up twice in each iteration in order to slice the original list into thirds. The recurrence is given here:

$$T(n) = T(\frac{2n}{3}) + c$$

The complexity can be solved with the master theorem, where for the tenery search, $a = 1$, $b = \frac{3}{2}$, and $c = 0$, by the master theorem tenery search has the complexity:

$$\begin{aligned} &T(n) = \Theta(\log_3(n)) \\ &T(n) = \Theta(\log(n)) \end{aligned} \tag{4}$$

The binary search alrorithm can be evaluated with the master theorem the same way instead with the recurrence $T(n) = T(\frac{n}{2}) + c$. The solution gives a similar complexity to the tenery search differing by the base of the log. This difference is negligible asymptotically. The complexity of the binary search algorithm is given by:

$$\begin{aligned} &T(n) = \Theta(\log_2(n)) \\ &T(n) = \Theta(\log(n)) \end{aligned} \tag{5}$$

5 Problem 5

5.1 Question:

A k-way merge operation. Suppose you have k sotred arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

a One strategey: Using a merge procedure, merge the first two arrays, then merge in the third array, then merge in the fourth, and so on. What is the time complexity of this algorithm in terms of k and n?

b Design a more efficient algorithm to solve this problem using divide and conquer.

5.2 Solution:

The procedure involves merging each list, and sorting each new list into the larger list. In order to do this, the new element to be merged has to be inserted into the combined list at the appropriate index if the k lists are not sorted with respect to each other. Putting two lists end to end from each other has takes time $O(1)$. Each individually sorted list is then sorted into the whole, each list is appended then sorted, forming a merge. If this is generalized to a mergesort among k arrays, the complexity of $O(n\log(n))$ becomes

$$O(kn\log(n))$$

First take each list and concatenate them, so that there is one large array of size kn, this operation is linear with k lists. Then perform mergesort on the kn list. Mergesort is a divide and conquer algorithm that reduces the merged array into indiviaual elements, then compares each element with the first element in the next list. Mergesort has a known complexity of

$$O(n\log(n))$$