CMPU 241 - Analysis of Algorithms

Spring 2019
Assignment 3
Due: Tuesday, February 26th, by 5pm

NAME: Jonah Tuckman

1. Give tight asymptotic bounds for $T(n)$ in each of the following recurrences. Use backward substitution, either version of the Master Theorem, or make a good guess and show that it holds. Assume that $T(n)$ is constant for $n \leq 2$. State your method of solution and if you use either version of the Master Theorem, state the case the recurrence falls into (Case 1, 2, or 3, etc.). Show all your work.

   a. $T(n) = T(\frac{9n}{10}) + n$
   Based on the master theorem:
   $a = 1\ b = \frac{10}{9}\ k = 1\ p = 0\ f(n) = n$

   Case 3: if $f(n) = \theta(n^c) such that (c > \log_a b)$
   Then: $T(n) = \theta(f(n))$

   $1 < \frac{10}{9}^1$
   Thus, $T(n) = \Theta(n)$

   b. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$
   Based on the master theorem:
   $a = 2\ b = 4\ k = \frac{1}{2}\ p = 0\ f(n) = \sqrt{n}$

   Case 2: If it is true, for some constant k 0, that: $f(n) = \Theta(n^c \log^k n) where (c = \log_b a)$
   Then: $T(n) = \Theta(n^c \log^{k+1} n)$

   $a = b^k => 2 = \sqrt{4}$
   thus, $T(n) = \Theta(\sqrt{n} \log n)$

   c. $T(n) = T(n-1) + n$

   Will need to use backward substitution, master theorem does not apply to the $n - 1$.

   $T(n) = T(n-1) + n$
   $= T(n-2) + n - 1 + n$
   $= T(n-3) + n - 2 + n - 1 + n$
   Pattern: $T(n-k) = T(n-k) + kn + \frac{k(k-1}{2}$
   Base Case: $T(1) = T(n-1) + n + \frac{1(0)}{2} = T(n-1) + n$
   Because base case brings us to $k = n - 1$, we will now evaluate $T(1) + T(k-1)$.

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

d. $T(n) = 3T(\frac{n}{2}) + n$
Based on the Master Theorem:
$a = 3\ b = 2\ k = 1\ p = 0\ f(n) = n$

Case 1: if $f(n) = \Theta(n^c)$ where $c < \log_b a$
then: $T(n) = \Theta(n^{\log_b a})$

$a > b^k => 3 > 2^1$
thus, $T(n) = \Theta(n^{\log_2 3})$

e. $T(n) = 6T(\frac{n}{4}) + n^2$
$a = 6\ b = 4\ k = 2\ p = 0\ f(n) = n^2$

Case 3 is applicable here because:
$a < b^k => 6 < 4^2$
Thus:
$T(n) = \Theta(n^2)$

f. $T(n) = 3T(\frac{n}{4}) + nlgn$

$a = 3\ b = 4\ k = 1\ p = 1\ f(n) = n \log n$
As above, case 3 is applicable for similar reasoning:
$a < b^k => 3 < 4^1$
Thus: $T(n) = \Theta(n \log n)$

2. For each of the following arrays of keys, give the heapsize. That is, if the array is a max-heap at the root and at every other node in the tree, the heapsize is the number of items in the array. If the entire array is not a max-heap, give the node at which the max-heap property fails and give the size of the max-heap up until the node where the property fails.

   The first two examples are given for you.

24 12 16 13 10 8 14	Node 2 is not the root of a max-heap. The heapsize is 1
25 14 16 13 10 8 12	Each node is the root of a max-heap. The heapsize is 7
25 14 13 16 10 8 12	Fails at node 2 (14 ! > 16). The heapsize is 1 before failure.
25 14 12 13 10 8 16	Fails at node 3 (12 ! > 16). The heapsize before this failure is 6.
14 13 12 10 8	Each node is the root of a max-heap. The heapsize is 5
14 12 8 10 13	Fails at node 2, is not the root of a max heap. heapsize is 1
89 19 40 17 12 10 2 5 7 11 6 9 70	Node 6 is not the root of a max heap. Size is 5 at this point.

3. What are the minimum and maximum numbers of nodes in a heap of height h?

   Minimum = 2^h
   Maximum = $2^{h+1} - 1$ (bottom row is fully filled)

4. Order the following functions in terms of their order of growth (i.e., growth rate), from lowest to highest:

   $n\lg n$, $n+\lg n$, $n!$, n, n^2, 4^n, $n^{\frac{1}{2}}$, 4^n
   Lowest to Highest: $n^{\frac{1}{2}}$, n, $n+\lg n$, $n\lg n$, 4^n, n^2,2^n, $n!$

   Provide a justification for your answers and indicate which of the functions, if any, grow at the same rate.
   $n^{\frac{1}{2}} = \sqrt{n}$ which runs the fastest, followed by the linear function n. We consider $4^n = 4^n$ as the constants attached to the raise to the n power are of less importance than the data size itself. $n!$ is the highest run time as its computation is by far the highest as our datasizes get larger

5. Consider the SELECTION-SORT algorithm, given below. Assume the input and output are as specified for the sorting problem on page 16 of our textbook.

```
SELECTION-SORT(A):
1.  n = A.length
2.  for j = 1 to n − 1
3.      smallest = j
4.      for i = j + 1 to n
5.          if A[i] < A[smallest]
6.              smallest = i
7.      exchange A[j] with A[smallest]
```

Consider the following loop invariant of the outer for loop (lines 2 through 7).

> The algorithm maintains the loop invariant that at the start of each iteration of the outer for loop, the subarray $A[1..j-1]$ consists of the $j-1$ smallest elements in the array $A[1..n]$ and the subarray $A[1..j-1]$ is in sorted order.

Show that the loop invariant is true using a formal proof like that shown in class for INSERTION-SORT (prove the invariant holds initially, at every subsequent iteration, and at termination, according to the value of j).

Base Case: If the length of the array is one, prior to entering the first loop the statement of looping 1 to n - -1 would not enter and would return. This loop is ordered because there is one sole element in the input array and thus it is sorted.

Inductive Hypothesis: Assume that the loop invariant holds up to iteration k. Thus, at the beginning of iteration k, the array from element 1 to element k - 1 is in sorted order.

Inductive Step: When entering the iteration j = k. By our inductive hypothesis we understand that the array from 1 to k is already in sorted order. The algorithm then loops through the remaining array and finds the smallest element in the array from range k to n. Once the smallest element in this section is found, it is swapped into position k. Based on our assumption in the inductive hypothesis that our array was sorted from 1 to k - 1, once the smallest remaining element is swapped into index k, we can understand that at the beginning of iteration k + 1, our array is sorted in the range 1 to k.

Termination: When the loop has gone through n - 1 elements, the array is sorted based on the assumption above. We do not need to do n full loops in the initial loop because once we have gone through n - 1 iterations, the last element is also sorted as the largest based on a process of elimination. Once we have gone through n - 1 iterations, we fail the outer for loop check and terminate with a sorted array.

6. Come up with running time solutions for the following peudocode samples using either the unrolling method or by multiplying the time taken by each case with nested loops as shown in class. If an inner loop counter j is dependent on an outer loop counter i, for each value of i, determine how many times the internal loop is repeated and derive a closed-form expression from the summation. If all loops are independent, find the number of times each is executed and multiply the result.

```
Alg1(n)
{
   for (i = 1;  i ≤ n;  i++)

      for (j = 1;  j ≤ i;  j++)

         for (k = 1;  k ≤ 100;  k++)

            printf (i + j + k)

}
```

The first loop runs n times. The second inner loop runs an incrementing amount of time, beginning at 1 alongside i and continuing until reaching n. This loop is thus equal to a runtime of $\frac{n+1}{2}$. The third loop runs 100 times. Thus the multiplication of the loops runs $100n\frac{n+1}{2}$. The constant 100 is eliminated as does the $\frac{1}{2}$. The runtime for this algorithm is thus $T(n) = O(n^2)$.

```
Alg2(n)
{
   for (i = n/2;  i ≤ n;  i++)

      for (j = 1;  j ≤ n;  j = j * 2)

         for (k = 1;  k ≤ n;  k = k * 2)

            printf (i + j)

}
```

The first loop goes from $\frac{n}{2}$ to n, thus runs $\frac{n}{2}$ times. This can be simplified to n times. The second and third loops run a maximum of $\log n$ times. There is an n runtime, and two logn runtimes for our three loops. Thus the overall runtime is $T(n) = O(n \log^2 n)$