EECE.3220: Data Structures

Spring 2017

Homework 1 Solution

1. *(25 points) Assume each expression listed below represents the execution time of a program. Express the order of magnitude for each time using big O notation.*

Solutions: In each case, the fastest growing term (which determines order of magnitude) is in **bold**.

a. $T(n) = \boldsymbol{n^3} + 100n \cdot log_2 n + 5000 = \boldsymbol{O(n^3)}$

b. $T(n) = \boldsymbol{2^n} + n^{99} = \boldsymbol{O(2^n)}$

c. $T(n) = \boldsymbol{\frac{n^2-1}{n+1}} + 8\, log_2 n = \boldsymbol{O(n)}$

Note: $n^2 - 1 = (n + 1) * (n - 1)$, so that first term is simply $(n - 1)$.

d. $T(n) = 1 + 2 + 4 + \cdots + \boldsymbol{2^{n-1}} = \boldsymbol{O(2^n)}$

Note: The answer here is $O(2^n)$ and not $O(2^{n-1})$ because $2^{n-1} = 2^n / 2 = 0.5 * 2^n$.

2. *(75 points) For each of the code segments below, determine an equation for the worst-case computing time T(n) (expressed as a function of n, i.e. 2n + 4) and the order of magnitude (expressed using big O notation, i.e. O(n)).*

Solutions: In each case, the number of times each line is executed is written to the right in red. Also, for simplicity's sake, a for loop is treated as a single statement, despite the fact that a for loop is really a collection of three statements.

a.
```
// Calculate mean
n = 0;                  1
sum = 0;                1
cin >> x;               1
while (x != -999)       n + 1
{
   n++;                 n
   sum += x;            n
   cin >> x;            n
}
mean = sum / n;         1
```

$T(n) = 1 + 1 + 1 + (n+1) + n + n + n + 1 = \mathbf{4n + 5 = O(n)}$

Note: While the value of x controls the number of loop iterations, n counts the number of iterations, as it's incremented in every loop iteration. You can therefore express the execution time as a function of n.

b.
```
// Matrix addition
for (int i = 0; i < n; i++) {             n + 1
   for (int j = 0; j < n; j++) {          n + 1
      c[i][j] = a[i][j] + b[i][j];        n        n
   }
}
```

$T(n) = (n + 1) + n * ((n + 1) + n) = \mathbf{2n^2 + 2n + 1 = O(n^2)}$

c.
```
// Matrix multiplication
for (int i = 0; i < n; i++) {                    n + 1
   for (int j = 0; j < n; j++) {                 n + 1
      c[i][j] = 0;                               n
      for (int k = 0; k < n; k++) {              n + 1   } n   } n
         c[i][j] += a[i][k] * b[k][j];           n
      }
   }
}
```

$$
\begin{aligned}
T(n) &= (n + 1) + n * [\ (n + 1) + n + n * ((n + 1) + n)\ ] \\
&= (n + 1) + n * [\ (n + 1) + n + 2n^2 + n\ ] \\
&= (n + 1) + 2n^3 + 3n^2 + n = \mathbf{2n^3 + 3n^2 + 2n + 1 = O(n^3)}
\end{aligned}
$$

d.
```
// Bubble sort
for (int i = 0; i < n - 1; i++) {            n
   for (int j = 0; j < n - 1; j++) {         n^2
      if (x[j] > x[j + 1]) {                 n^2-n
         temp = x[j];                        (n-1) + (n-2) + (n-3) …
         x[j] = x[j + 1];                    (n-1) + (n-2) + (n-3) …
         x[j + 1] = temp;                    (n-1) + (n-2) + (n-3) …
      }
   }
}
```

$$T(n) = n + n^2 + (n^2 - n) + n*(n-1) / 2 = \mathbf{2.5n^2 - 0.5n = O(n^2)}$$

<u>**Note:**</u> The worst case for a bubble sort is that the array is initially sorted from largest to smallest value. So, the body of the if statement executes (n-1) times the first time through the inner loop, (n-2) times the second time, and so on, executing just 1 time in the last inner loop iteration. As shown in our discussion of selection sort, that sum is equal to n*(n-1)/2.

e.
```
while (n >= 1)        log2n + 2
   n /= 2;            log2n + 1
```

$$T(n) = (\log_2 n + 2) + (\log_2 n + 1) = \mathbf{2 \log_2 n + 3 = O(\log_2 n)}$$

<u>**Note:**</u> The analysis here is similar to the analysis of binary search, in which the loop test executes ($\log_2 n + 2$) times. A few examples will show this analysis to be true—for example, say $n = 8 = 2^3$. It takes 4 (which is $\log_2 n + 1$) iterations for n /= 2 to produce the value 0, and the loop condition must therefore be tested a 5th time for the loop to end.

f.
```
x = 1;                                      1
for (int i = 1; i <= n - 1; i++) {          n
   for (int j = 1; j <= x; j++)             2 + 3 + 5 + … + (2^(n-2)+1)
      cout << j << endl;                    1 + 2 + 4 + … + 2^(n-2)
   x *= 2;                                  n-1
}
```

$$T(n) = 1 + n + (2^{n-1} + n - 2) + (2^{n-1} - 1) + (n - 1)$$
$$= 2 \cdot 2^{n-1} + 3n - 3 = \mathbf{2^n + 3n - 3 = O(2^n)}$$

Note: To derive the formula for T(n) shown above, I went through the following analysis:

The number of inner loop iterations is based on the value of x—the for loop condition is always tested (x + 1) times, and the body of the loop executes x times. x doubles every time you go through the outer loop, and the body of that loop executes n-1 times. So, the last time you execute the inner loop, $x = 2^{n-2}$, then x is doubled one last time to 2^{n-1}.

After about 10 minutes of Google searching (I wish that was a joke), I was able to find the following formula that helped me determine a simple value for the sum $1 + 2 + 4 + \ldots + 2^{n-2}$:

$$\sum_{k=0}^{N-1} r^k = \frac{1 - r^N}{1 - r}$$

Therefore, the body of the inner loop executes $2^n - 1$ times, as shown by evaluating that formula for r = 2 and N = (n-1):

$$\sum_{k=0}^{n-2} 2^k = \frac{1 - 2^{n-1}}{1 - 2} = 2^{n-1} - 1$$

The inner loop condition is tested 1 more time than the loop body executes. The number of terms in the sum $1 + 2 + 4 + \ldots + 2^{n-2}$ is n-1. So, the third line executes $2^{n-1} - 1 + (n-1) = 2^{n-1} + n - 2$ times.