CIS121-204–Fall 2007

How to Determine Big-O in a Given Algorithm?

Tuesday, October 2

We already know how to count the exact number of steps, given an algorithm, in Lab 2. Counting a number of steps is a precise way to determine the complexity of the algorithm, but, as we have experienced, it is a tedious process. In fact, in order to obtain the complexity of an algorithm, a nicer, better way can be done using an approximation of the exact running time. That is, we take the advantage of Big-O in analyzing the running time.

Since all we care about the complexity is the most dominant term in the running time function, we can just keep track of that term and discard the rest, including the lower-order terms and the coefficient of the most dominant term.

Consider the following examples.

1. This algorithm is from Lecture 5, Slide 10:

```
01 public static int maxSubSum1(int [] a) {
02  int maxSum = 0;                              O(1)
03  for (int i=0; i < a.length; i++) {           n times; this line O(1)
04   for (int j=i; j < a.length; j++) {          n-i times; this line O(1)
05    int thisSum = 0;                           O(1)
06    for (int k=i; k <= j; k++)                 j-i+1 times; this line O(1)
07     thisSum = thisSum + a[k];                 O(1)
08    if (thisSum > maxSum)                      O(1)
09     maxSum = thisSum;                         O(1)
10   }
11  }
12  return maxSum;                               O(1)
13 }
```

First of all we write the complexity of each line, shown above. Then we calculate the overall complexity of the algorithm. The best way to do this is to consider the innermost loop first.

Let `n=a.length`. For Lines 6-7, the innermost `for` loop has two $O(1)$ statements. The loop executes $j - i + 1$ times. Hence, the running time of this loop is $O(j - i + 1)$ or simply $O(j - i)$.

For Lines 4-10, the intermediate `for` loop has four $O(1)$ statements and one $O(j-i)$ block. Hence, the complexity of the body of the loop is $O(j - i)$. The loop executes $n - i$ times, so the running time of this loop is

$$\begin{aligned} O\left(\sum_{j=i}^{n-1} j - i\right) &= O\left(\frac{n(n-1)}{2} - \frac{i(i-1)}{2} - i(n-i)\right) \\ &= O(n^2 + i^2 - ni). \end{aligned}$$

To obtain the first summation, we observe that the running time of the loop is the summation of each iteration's running time, which is a function of $O(j - i)$. By the Big-O theorem in the lecture notes, we know that the summation equals $O\left(\sum_{j=i}^{n-1} j - i\right)$.

For Lines 3-11, the outermost **for** loop has an $O(n^2 + i^2 - ni)$ block. The loop executes n times, so the running time of this loop is

$$\begin{aligned} O\left(\sum_{i=0}^{n-1} n^2 + i^2 - ni\right) &= O\left(n^3 + \frac{(n-1)(n)(2n-1)}{6} - n\frac{n(n-1)}{2}\right) \\ &= O\left(n^3 + n(n-1)\frac{2n-1-3n}{6}\right) \\ &= O\left(n^3 - \frac{(n-1)n(n+1)}{6}\right) \\ &= O(n^3). \end{aligned}$$

Finally, the whole algorithm consists of two statements of O(1) and an $O(n^3)$ block. Therefore, **maxSubSum1** runs in $O(n^3)$ time. □

2. This algorithm is from Lecture 5, Slide 12:

```
01 public static int maxSubSum2(int[] a) {
02  int maxSum = 0;                          O(1)
03  for (int i=0; i < a.length; i++) {       n times; this line O(1)
04   int thisSum = 0;                        O(1)
05   for (int j=i; j < a.length; j++) {      n-i times; this line O(1)
06    thisSum = thisSum + a[j];              O(1)
07    if (thisSum > maxSum)                  O(1)
08     maxSum = thisSum;                     O(1)
09   }
10  }
11  return maxSum;                           O(1)
12 }
```

Let **n=a.length**. For Lines 5-9, the inner **for** loop has four $O(1)$ statements. Hence, the complexity of the body of the loop is $O(1)$. The loop executes $n - i$ times, so the running time of this loop is $O(n - i)$.

For Lines 3-10, the outer **for** loop has an $O(1)$ statement and an $O(n - i)$ block. The loop executes n times, so the running time of this loop is

$$\begin{aligned} O\left(\sum_{i=0}^{n-1} n - i\right) &= O\left(n^2 - \frac{n(n-1)}{2}\right) \\ &= O(n^2). \end{aligned}$$

Finally, the whole algorithm consists of two statements of O(1) and an $O(n^2)$ block. Therefore, **maxSubSum2** runs in $O(n^2)$ time. □