CS1102: Data Structure and Algorithms

Tutorial 6 - Analysis of Algorithms (Solutions)

Week of 08 March 2010

1. Rearrange the following functions in the increasing order of their Big Oh complexity:

   $4n^2$, $\log_3(n)$, $20n$, $n^{2.5}$, 2, n^n, 3^n, $n \log(n)$, $100n^{2/3}$, 2^n, 2^{n+1}, $n!$, $(n-1)!$, 2^{2n}

   Answer:

   The above sequence can be rearranged as follow:

   $O(2) < O(\log_3(n)) < O(100n^{2/3}) < O(20n) < O(n \log(n)) < O(4n^2) < O(n^{2.5}) < O(2^n) = O(2^{n+1}) < O(3^n) < O(2^{2n}) < O((n-1)!) < O(n!) < O(n^n)$.

2. Write the following different version of the exponential functions X^n

```
double Exponential(double base, int exp)
//Assumption: exp >= 0
```

   a. Iterative version
   b. Recursive version, utilizing the fact $X^n = X * X^{n-1}$
   c. Recursive version, utilizing the fact $X^{2n} = (X^n)^2$.

   Briefly comment on the time complexity of each of the version using Big-O notation.

   Answer:

a. Iterative Version

```
double Exponential(double base, int exp)
{
     double result = 1;

     for (int i = 0; i < exp; i++)
          result *= base;

     return result;
}
```

Complexity is O(n) if n represents the exponent factor.

b. Recursive Version One

```
double Exponential(double base, int exp)
{
     if (exp == 0)       //base case
          return 1;

     return base * Exponential(base, exp-1);
}
```

Complexity is O(n).

c. Recursive Version Two

```
double Exponential(double base, int exp)
{
     double result;

     if (exp == 0)       //base case
          return 1;

     result = Exponential(base, exp/2);

     if (exp % 2 == 0) {return result * result;
     }
     else return base * result * result;

}
```

The complexity is O(log n).

CS1102: Data Structure and Algorithms

3. Analyze the Big Oh complexity of each of the following code fragments.

a)
```
// loop 1
for(int i = 0; i < n; i++)
     // loop 2
     for(int j = 0; j < n; j++)
          System.out.println(“*”);
```

b)
```
// loop 1
for(int i = 0; i < n; i++)
     // loop 2
     for(int j = 0; j < i; j++)
          System.out.println(“*”);
```

c)
```
// loop 1
for(int i = 0; i < n; i++)
     // loop 2
     for(int j = i+1; j > i; j--)
          // loop 3
          for(int k = n; k > j; k--)
               Sytem.out.println(“*”);
```

d)
```
// loop 1
for (int i = 0; i < n; i++) {
     // loop 2
     for (int j = 0; j < n; j++) {

          if ((j % 2) == 0) {
               // loop 3
               for (int k = i; k < n; k++)
                    System.out.println(“*”);
               }
          else {
               // loop 4
               for (int l = 0; l < i; l++)
                    System.out.println(“*”);
          }
     }
}
```

Answer:

a) Big Oh complexity: $O(n^2)$
Loop1 runs n times, and loop 2 runs n times, so in total the print statement is run $n * n = n^2$.

b) Big Oh complexity: $O(n^2)$
For the first time around loop 1 (i=0), loop 2 runs 0 times, on the second time around loop 1 (i=1), loop 2 runs 1 time, and each time the number of times loop 1 runs is 1 longer than the last. Loop 1 runs n times, so the print statement is run $1 + 2 + ... + n - 1$ times is equal to n*(n-1)/2.

c) Big Oh complexity: $O(n^2)$
Loop 2 always runs exactly once and so can be ignored. Loop 1 runs n times. Loop 3 runs in the same fashion as loop 2 in (b), but in reverse order. First time it runs n-1 times, then n-2, then n-3, all the way down to 3, 2, 1 times. So the answer is exactly the same as in (b).

d) Big Oh complexity: $O(n^3)$
Loop 1 and loop 2 run through all n^2 values of i and j between 0 and $n - 1$, inclusive. For each i, j, loop 3 executes (n – i) times if j is even and loop 4 executes i times if j is odd. So, for each i, the total number of the print statement executed is $(n-i)*n/2 + i*n/2 = n^2/2$. So, the Big Oh complexity of the program is $O(n^3)$.

CS1102: Data Structure and Algorithms

4. We consider the problem of finding the maximum sum of a contiguous subsequence of integers from a given input data sequence. For example, if the input data sequence is {-1, 12, -7, 13, -5, 2, -3}, the answer is 18 (12-7+13). The maximum subsequence sum is defined as 0 if all the integers are negative. Your may use the following formulas:

$$\sum_{j=i}^{n-1} j = i+(i+1)+\ldots+(n-2)+(n-1) = (n+i-1)(n-i)/2$$
$$\sum_{j=0}^{n-1} j^2 = 0^2+1^2+\ldots+(n-2)^2+(n-1)^2 = (n-1)n(2n-1)/6$$

(a) The most obvious way to solve this problem is to compute the sum of each possible subsequence, and retain the largest sum as the result.

```
// a is an array which stores the input data sequence of length
//n
int maxSubSum1( int[]a )
{
      int max_sum = 0; // maximum sum of a contiguous
                       //subsequence
      int n = a.length;// the length of the array

      //loop 1:  i is the starting index of a subsequence
      for(int i=0; i< n; i++)
      {
            //loop 2: j is the ending index of a
            //subsequence
            for(int j=i; j< n; j++)
            {
                  int this_sum=0; // the sum of the current
                                  //subsequence
                  for(int k=i; k <= j; k++) //loop 3
                  {
                        this_sum += a[k];
                  }
                  if(this_sum > max_sum) max_sum = this_sum;
            }
      }
      return max_sum;
}
```

Each subsequence is identified by its starting index and ending index in the array. In the code, loop 1 and loop 2 are used to generate all possible pairs of starting and ending indices. For each subsequence, loop 3 is used to compute the sum of the elements in the subsequence.

What is the Big Oh complexity of this algorithm?

(b) The following code is an improved version of the solution in (a).

```
int maxSubSum2( int[]a )
{
      int max_sum = 0; // maximum sum of a contiguous
                       //subsequence
      int n = a.length;//length of the array

      //loop 1: i is the starting index of a subsequence
      for(int i=0; i< n; i++)
      {
            int this_sum= 0; // the current calculated sum
            //loop 2: j is the ending index of a subsequence
            for(j=i; j< n; j++)
            {
                  this_sum += a[j]; // the sum of the current
                                    //subsequence
                  if(this_sum > max_sum) max_sum = this_sum;
            }
      }
      return max_sum;
}
```

Explain why this solution correctly computes the maximum subsequence sum. Analyze its Big Oh complexity.

(c) Below is our final algorithm for computing the maximum subsequence sum. Explain why it is correct and analyze its Big Oh complexity.

```
int maxSubSum4( int[]a )
{
      int max_sum = 0;
      int this_sum = 0;
      int n = arr.length;
      for(int i=0; i< n; i++)
      {
            this_sum += a[i];
            if(this_sum > max_sum)
                  max_sum = this_sum;
            else if(this_sum < 0)
                  this_sum = 0;
      }
      return max_sum;
}
```

Answer:

(a) Intuitively, it is easy to see that the maximum number of iterations of each loop (the worst case) is governed by the size of the sequence, n. Thus, the worst case running time for the algorithm is $O(n^3)$.

More specifically,
Loop 3 executes $j-i+1$ times for a given i and j value.

For a given i value, loop 2 executes the statement "`this_sum += a[k];`"

$$\sum_{j=i}^{n-1} (j-i+1) = \sum_{j=i}^{n-1} j + \sum_{j=i}^{n-1} (-i+1)$$
$$= (n+i-1)(n-i)/2+(n-i)(-i+1)$$
$$= (n-i)(n-i+1)/2 \text{ times}$$

(Loop 1) So, the program executes the statement "`this_sum += a[k];`"

$$\sum_{i=0}^{n-1} (n-i)(n-i+1)/2$$
$$= \sum_{i=0}^{n-1} (n^2+n)/2 - (2n+1)\sum_{i=0}^{n-1} i/2 + \sum_{i=0}^{n-1} i^2/2$$
$$= (n^3+n^2)/2-(2n+1)n(n-1)/4+(n-1)n(2n-1)/12$$
$$= n^3/6+\ldots$$

(no need to expand all, just look at the leading terms)

Overall the Big Oh complexity is $O(n^3)$

(b) Observation: subsequence sums need not always be computed from scratch. The sum of interval $[i, j+1]$ can be computed based on the sum of interval $[i, j]$ by adding the integer at $j+1$. Thus, we can eliminate the innermost loop and rewrite the code with only two loops.

Loop 2 executes its body $(n-i)$ times for a given i value.

(Loop1) So the program executes the body of loop 2

$$\sum_{i=0}^{n-1} (n-i) = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i = n^2-n(n-1)/2=(n^2+n)/2 \text{ times}$$

Overall the Big Oh complexity is $O(n^2)$

(c) Compared with the solution in (b), this final solution further eliminates the outer loop based on the following observations.

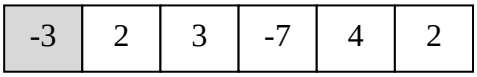
- If a[i] is negative, then any sequence that begins with it can be improved by starting with the next element.
  - e.g. MSS(-1,3,-2,3)= MSS(3,-2,3)
- Any subsequence that begins with a subsequence whose sum is negative can be improved by eliminating that subsequence.
  - e.g. MSS(1,-3,4,-2,3) = MSS(4,-2,3) because 1-3=-2<0

So, we just keep track of the index of the last element of the subsequence being examined. As soon as the sum of a subsequence becomes negative, we just set the sum (this_sum) back to zero, essentially eliminating it from the current subsequence's sum (this_sum).

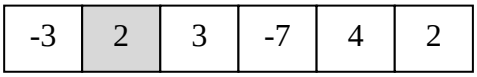
The loop executes its body *n* times. The complexity is *O(n).*

The following example illustrates the above algorithm:

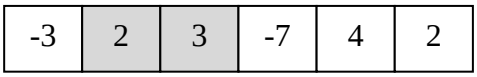
this_sum = -3, max_sum = 0. Apply observation (i) we conclude that max_sum doesn't contain a[0].



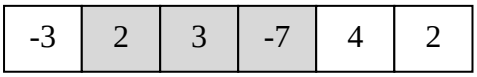
this_sum = 2, max_sum = 2



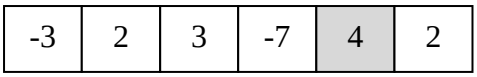
this_sum = 5, max_sum = 5



this_sum = -2, max_sum = 5 Apply observation (ii) we conclude that max_sum doesn't contain the subsequence a[1..3].



this_sum = 4, max_sum = 5



this_sum = 6, max_sum = 6