Time Complexity Analysis and Growth Functions

CSCD 300 – Data Structures

Eastern Washington University

Goal

The goal of today's lecture is to analyze the time usage of several example programs and show their time complexity in the big-oh notation. We will see their time complexities are represented by functions that have different growth rates.

Outline

1. Recall: How to measure the speed of an program ?
2. The time complexity analysis for several example programs
3. Compare the different growth functions

Recall: How to measure the speed of an program ?

- We just count how many constant-time operations (or steps) are executed.
- Constant-time operations include: add, sub, multi, div, modulo, assignment, comparison, ...
- A constant-time operation can also be a constant number of contant-time operations, because the time cost of those operations is the product of a contant (the time cost of one operations) and another contant (the number of operations), and the product is still a constant.
- We are often more interested in that count in the worst case. Namely, the worst-case time complexity.
- We are often more concerned with the most significant term in the expression of the time usage and neglect the leading constant, and then use the big-oh notation to give the asymptotic upper bound of the time usage of a program.

The time complexity analysis for several example programs

We will look at the several example programs with their time complexities being the following growth functions.

- The constant function
- The logarithmic function
- The linear function
- The N-Log-N function
- The quadratic function
- The cubic function

The constant function

```
/* return the bigger one between a and b */
int max(int a, int b)
{
   if (a >= b)          ---+
      return a;            | Obviously, all these together
   else                    | takes constant time: c
      return b;         ---+
}
```

So, by neglecting the leading constant c and using the big-oh notation, we get the time complexity of `max()` is $O(1)$, meaning its time usage is asymptotically upper bounded by a constant even in the worst case.

The logarithmic function

We have seen in the previous lecture that the binary search algorithm (either using the iteration structure or the recursion structure) has the time complexity of $O(\log n)$, where n is the input size. That means the binary search's time usage is asymptotically upper bounded by $\log n$ even in the worst case.

The linear function

```
//return the summation of all the numbers in A
int sum(int[] A)
{
   int s = 0, i = 0, n = A.length; -----> contant time: c1

   while(i < n){                   -----> n steps
      s = s + A[i]; ---+
                       | time cost for one step is constant: c
      i ++;         ---+
   }
   return s;        -------------------> contant time: c2
}
```

Thus, the time complexity is: $c_1 + cn + c_2 = O(n)$, by neglecting the constant terms c_1 and c_2 and neglecting the leading constant factor c.

The N-Log-N function

```
/* Search the location of every element of array X[]
   in the array A[], using binary search.
   Save the results in array Y[].
   Assume: A.length = X.length = Y.length = n
*/

void array_binary_search(int[] A, int[] X, int[] Y)
{
   For(int i = 0; i < n; i++)         ----> a total of n steps.
      Y[i] = BinarySearch(A, X[i]); ----> each binary search
                                          takes (log n) time.
}
```

Every `for` loop's step has some constant-time operation spending some constant amount of time c and has a binary search spending $O(\log n)$ time. So altogehter, the total time cost is: $O(n \log n)$.

The quadratic function

```
/* Decide whether array A and array B are disjoint
   Assume A.length = B.length = n
*/
boolean areDisjoint(int[] A, int[] B)
{
   for(int i = 0; i < n; i++)     ---+
                                     | n*n steps.
      for(int j = 0; j < n; j++)  ---+

         if(A[i] == B[j])  | constant time cost: c
            return false;  |

   return true; -----------------> constant time cost: c1
}
```

So the total time cost is: $cn^2 + c_1 = O(n^2)$.

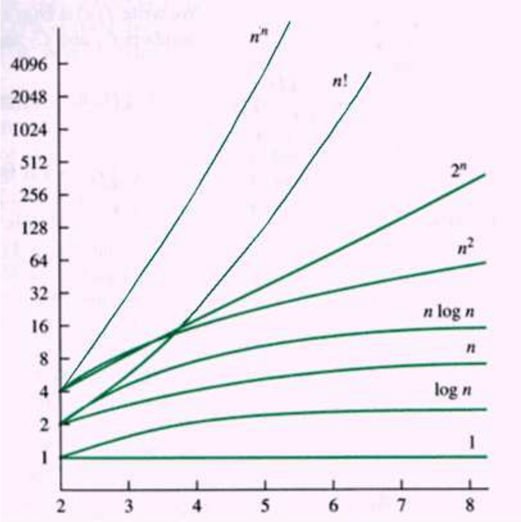
The cubic function

```
/* Decide whether there is an element that exists
   in Array A, B, and C.
   Assume A.length = B.length = C.length = n
*/
boolean areDisjoint(int[] A, int[] B, int[] C)
{
   for(int i = 0; i < n; i++)        ---+
      for(int j = 0; j < n; j++)        | n*n*n steps.
         for(int k = 0; k < n; k++) ---+
            if(A[i] == B[j] == C[k])    | constant time cost: c
               return false;            |

   return true;         ------------> constant time cost: c1
}
```

So the total time cost is: $cn^3 + c_1 = O(n^3)$.

Compare the different growth functions [1]



The picture on the left shows the different growth rate of different functions.

More discussions on the growth rate comparison of these functions will be given in a more formal way in CSCD320.

[1] Picture borrowed from:
http://blog.renatogama.com/2011/10/introducao-a-analise-de-algoritmos-parte-1