CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 9 – Analysis of Algorithms

(Week 11, starting 28 March 2016)

1. Big-O Complexity

Remember, Big-O time complexity gives us an idea of the **growth rate** of a function. In other words, for a **large input size N**, **as N increases**, in what **order of magnitude** is the volume of statements executed expected to increase?

Rearrange the following functions in increasing order of their **big-O** complexity:

$4n^2$	$\log_3 n$	$20n$	$n^{2.5}$
$n^{0.00000001}$	$\log n!$	n^n	2^n
2^{n+1}	2^{2n}	3^n	$n \log n$
$100\ n^{2/3}$	$\log [(\log n)^2]$	$n!$	$(n-1)!$

Answer

Notice that some of the above terms have **equal Big-O complexity**. Remember that in Big-O notation, we only care about the **dominating term** of the function, **without** its constant **coefficient**. This can be proven mathematically. As N gets very large, the effect of the other terms on the volume of statements executed becomes insignificant.

	$O(\log [(\log n)^2]) = O(\log \log n)$			<				
Logarithmic time	$O(\log_3 n) = O(\log n)$			<				
(Sublinear time)	$O(n^{0.00000001})$			<	$O(100\ n^{2/3})$	<		
Linear time	$O(20n)$			<				
Linearithmic time	$O(n \log n)$	=	$O(\log n!)$ [Why?1]	<				
Quadratic time	$O(4n^2)$			<				
(Polynomial time)	$O(n^{2.5})$			<				
Exponential time	$O(2^n)$	=	$O(2^{n+1})$	<	$O(3^n)$	<	$O(2^{2n}) = O(4^n)$	<
Factorial time	$O((n-1)!)$			<	$O(n!)$	<		
!!!	$O(n^n)$							

[1] We can add coefficients to either function, so that one function bounds the other from above, i.e. both functions are in the same big-O complexity:

$\frac{n}{2} \log \frac{n}{2} < \log n! < n \log n < 2 \log (n!) = O(n \log n) = O(\log (n!))$

A **visualization** of this fact, though not a proof:
http://www.wolframalpha.com/input/?i=y%3D2log%28N%21%29%2C+y%3DN+log%28N%29%2C+y+%3D+log%28N%21%29%2C+y+%3D+%28N%2F2%29+log+%28N%2F2%29

Proof sketch: See accepted answer at http://stackoverflow.com/questions/2095395/is-logn-%CE%98n-logn

2. *Analyzing Time Complexity*

Find the big-O time complexity of each of the following code fragments:

(a)

```
int i = 1;
while (i <= n) {
     System.out.println("*");
     i = 2 * i;
}
```

(b)

```
int i = n;
while (i > 0) {
     for (int j = 0; j < n; j++)
          System.out.println("*");
     i = i / 2;
}
```

(c)

```
while (n > 0) {
     for (int j = 0; j < n; j++)
          System.out.println("*");
     n = n / 2;
}
```

(d)

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

(e)

```
void foo(int n){
     if (n <= 1)
          return;
     doOhOne(); // doOhOne() runs in O(1) time
     foo(n/2);
     foo(n/2);
}
```

(f)

```
void foo(int n){
     if (n <= 1)
          return;
     doOhN(); // doOhN() runs in O(n) time
     foo(n/2);
     foo(n/2);
}
```

Answer

Iterative Algorithms

(a) O(log n)

How many statements are executed, relative to input size n? Often, but **NOT always**, we can get an idea from the number of times a **loop iterates**.

There are no loops within the while loop, and the volume of statements executed within each iteration is a **constant**, i.e. not dependent on n. Therefore, we can just **sum up the number of iterations** to find out the relationship between n and the volume of statements executed.

The loop body executes for i = 2^0, 2^1, 2^2, 2^3, ..., $2^{\lfloor \log_2 n \rfloor}$, and this sequence has $1 + \lfloor \log_2 n \rfloor$ = O(log n) values.

When n is a power of 2:

i	# iterations
$1 = 2^0$	1
$2 = 2^1$	1
$4 = 2^2$	1
...	
$n = 2^{\log n}$	1

(b) O(n log n)

The two loops here are nested, but the number of iterations the inner loop runs is **independent** of the outer loop. Therefore, the total volume of statements can be taken by **multiplying** both values together.

The outer loop iterates O(log n) times. Within EACH iteration, the inner loop iterates n times, independent of the outer loop. Therefore, the time complexity of this code fragment is O(n log n).

When n is a power of 2:

i	# inner iter.
$n = 2^{\log n}$	n
...	
$4 = 2^2$	n
$2 = 2^1$	n
$1 = 2^0$	n

(c) O(n)

Here, we cannot examine each loop one at a time and then multiply the number of iterations together. Notice that n is used to control the outer loop, and it affects the number of iterations.

The inner loop is **dependent** on the control variable of the outer loop. Therefore, we have to fall back to **summing up the number of inner loop iterations** to find the volume of statements executed.

The geometric series[2] $1 + 2 + 4 + ... + \frac{n}{4} + \frac{n}{2} + \mathbf{n} = 2n - 1 = O(\mathbf{n})$

When n is a power of 2:

n, outer	# inner iter.
n	n
n/2	n/2
n/4	n/4
...	
4	4
2	2
1	1

(d) $O(n^2)$

Loop 2 runs a **constant number of times** (exactly once) every loop 1 iteration and **does not affect** the time complexity. We can carefully trace the values of j in each loop 1 / loop 2 iteration to find the number of times the innermost loop iterates.

The arithmetic series $1 + 2 + 3 + ... + (n-2) + (n-1) = \frac{(n-1)(n)}{2} = O(n^2)$

i	# k iter.
0	n-1
1	n-2
...	
n - 3	2
n - 2	1
n - 1	0

[2] You may observe that $\frac{n}{4} = 2^{(\log_2 n)-2}$, $\frac{n}{2} = 2^{(\log_2 n)-1}$, $n = 2^{\log_2 n}$

Recursive Algorithms

(e) O(n)

Draw the **recursive tree**. In many recursive problems, we can nicely add up the **volume of statements** executed within **each level** in the tree, and then nicely add up these subtotals to find the total volume of statements executed.

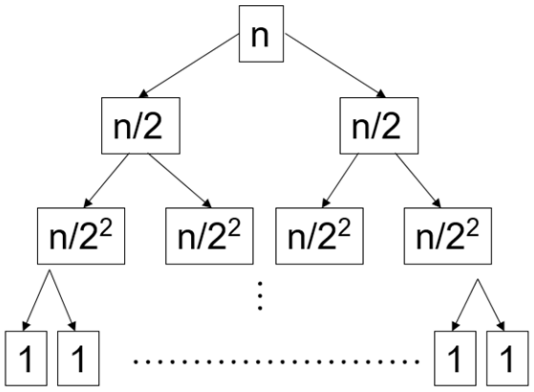
In this case, each call executes O(1) number of statements besides the 2 recursive calls, so we just need to find the total number of recursive calls made.

Level **1**:
1 call to foo

Level **2**:
2 calls to foo

Level **3**:
4 calls to foo

Level (**log n +1**):
n calls to foo



Adding up the level subtotals, the total volume of statements executed is the geometric series
$1 + 2 + 4 + \ldots + \frac{n}{4} + \frac{n}{2} + \mathbf{n} = 2n - 1 = O(\mathbf{n})$

(f) O(n log n)

Here, we cannot just add up the number of method calls. Do not forget that the **input size n changes** at each recursive call.

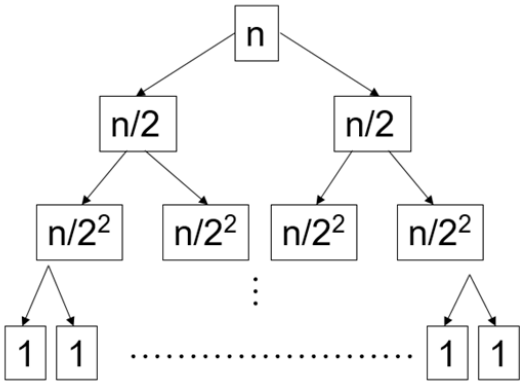
Each method call executes a different amount of statements, aside from invoking other recursive calls.

Level **1**:
1 call to foo, n

Level **2**:
2 calls to foo, 2 * n\2 = n

Level **3**:
4 calls to foo, 4* n\2² = n

Level (**log n +1**):
n calls to foo, n * 1 = n



Each level executes n statements, aside from the two recursive calls. There are O(log n) levels. Therefore, the time complexity is O(n log n).

3. *Efficient Algorithm Design*

Given an integer array A and an integer k, we want to find all pairs of elements in the array that add up to k. For each pair found, print the two numbers. If it is not possible to find any such combination, print "`Not possible.`"

The elements in A are unique. The array elements and k could be positive, zero, or negative. You do not need to print the other permutation that represents the same pair.

Sample Run 1

```
Enter size of array: 10
Enter the elements: 2 5 6 11 -1 0 25 -3 1 8
Enter the value of k: 22
-3 25
```

Sample Run 2

```
Enter size of array: 10
Enter the elements: 2 5 6 11 -1 0 25 -3 1 8
Enter the value of k: 50
Not possible.
```

There are two (to four) algorithms to solve this problem. One is a brute force algorithm, while the others are more efficient but require more code to implement. Design and analyze the big-O time complexity of two algorithms, then implement them.

```
public void findPairsBruteForce(int[] input, int target) {
     /* TODO */
}
public void findPairsImproved(int[] input, int target) {
     mergeSort(input); // O(n log n) sort given to you
     // Efficiency should be better than the brute force algorithm!
}
```

Answer

Besides the brute force algorithm, we can:	$O(n^2)$
Pre-sort the array and for each element, perform binary search	O(n log n) = O(n log n + log(n!))
Pre-sort the array and traverse it intelligently from both ends	O(n log n) = O(n log n + n)
Use a HashSet[3] to efficiently store and find one number	Average O(n)

Brute Force Algorithm

```
for each start index
     for each end index after start
          if A[start] + A[end] == k
               print
```

As the nested loops always run to completion and there is a constant amount of code within the inner loop, the time complexity can be determined by taking the sum of the number of inner loop iterations in the worst case. The worst case occurs when no pair is found.

For n numbers in A, the start index moves from 0 to n - 2 inclusive. When start index is 0, the end index moves from 1 to n-1 inclusive. Therefore, the time complexity of the brute force algorithm is (n-1) + (n-2) + ... + 3 + 2 + 1 = $\frac{(n-1)(n)}{2}$ = $O(n^2)$. If n is large, we can do much better.

[3] Coming soon

```java
public void findPairsBruteForce(int[] input, int target) {
   int n = input.length; // input size
   boolean possible = false;

   for (int start = 0; start < n - 1; start++) {
      for (int end = start + 1; end < n; end++) {
         if (input[start] + input[end] == target) {
            System.out.println(input[start] + " " + input[end]);
            possible = true;
            break;  // elements unique, no more matches for start
         }
      }
   }

   if (!possible)
      System.out.println("Not possible.");
}
```

Pre-sorting Followed by Efficient searching

```
sort A efficiently
for each start index
     if found (k - A[start]) in A[start + 1, n) using binary search
          print
```

Since we want to find an x where A[start] + x == k, we can instead use binary search to efficiently find the existence of x in the array. Binary search only works on a sorted array.

The time complexity of binary search over i elements is O(log i). For n numbers in A, the start index moves from 0 to n - 2 inclusive. When start index is 0, binary search needs to execute proportional to log (n-1) statements in the worst case. Therefore, the time complexity of this algorithm[4] is log (n-1) + log (n-2) + ... + log 3 + log 2 + 1 = 1 + log[(n-1)!] = O(n log n).

```java
public void findPairsImproved(int[] input, int target) {
   mergeSort(input); // O(n log n) sort given to you
   int n = input.length;
   boolean possible = false;

   for (int start = 0; start < n - 1; start ++) {
      int index = Arrays.binarySearch( // end index is exclusive
         input, start + 1, n, target - input[start]);
      if (index >= 0) { // match found
         System.out.println(input[start] + " " + input[index]);
         possible = true;
      }
   }

   if (!possible)
      System.out.println("Not possible.");
}
```

[4] We can conclude that log[(n-1)!] = O(n log n) because log[(n/2)!] < log[(n-1)!] < log(n!) and log[(n/2)!] = O($\frac{n}{2}$ log $\frac{n}{2}$) = O(n log n)

We can still improve the efficiency of searching for a matching pair on a sorted array, but the time complexity of the algorithm for this entire problem will not improve further because it is limited by the sorting algorithm.

Pre-sorting Followed by Efficient searching

```
sort A efficiently
start moves from the front, end moves from the end/back of A
while start < end
     if A[start] + A[end] < k
          start++
     else if A[start] + A[end] > k
          end--
     else
          print
          start++, end--
```

This algorithm allows us to eliminate many more pairs without even comparing them against k. Each index moves toward each other from the ends of the array. If the sum of the two array elements is too small, there will NOT be any solutions with the current first element and any second element after the first. Likewise, if the sum is too large, there will be no solutions with the given second element.

Each iteration in the while loop, either one or both indexes move toward each other. In the worst case, only one index moves toward each other at any time. The loop iterates n-1 times, but the time complexity of the entire algorithm is O(n log n) due to sorting.

You can (and should) easily implement this algorithm on your own!

^_^

O(1)

weeks left!