

Chapter 22 Developing Efficient Algorithms

Executing Time

Suppose two algorithms perform the same task such as search (linear search vs. binary search). Which one is better? One possible approach to answer this question is to implement these algorithms in Java and run the programs to get execution time. But there are two problems for this approach:

- First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
- Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

Growth Rate

It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their *growth rates*.

Big O Notation

Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires n comparisons for an array of size n . If the key is in the array, it requires $n/2$ comparisons on average. The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of n . Computer scientists use the Big O notation to abbreviate for “order of magnitude.” Using this notation, the complexity of the linear search algorithm is $O(n)$, pronounced as “*order of n*.”

Best, Worst, and Average Cases

For the same input size, an algorithm's execution time may vary, depending on the input. An input that results in the shortest execution time is called the *best-case* input and an input that results in the longest execution time is called the *worst-case* input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case. An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

Ignoring Multiplicative Constants

The linear search algorithm requires n comparisons in the worst-case and $n/2$ comparisons in the average-case. Using the Big O notation, both cases require $O(n)$ time. The multiplicative constant ($1/2$) can be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for $n/2$ or $100n$ is the same as n , i.e., $O(n) = O(n/2) = O(100n)$.

$f(n)$ n	n	$n/2$	$100n$
100	100	50	10000
200	200	100	20000
	2	2	2

$f(200) / f(100)$

Ignoring Non-Dominating Terms

Consider the algorithm for finding the maximum number in an array of n elements. If n is 2, it takes one comparison to find the maximum number. If n is 3, it takes two comparisons to find the maximum number. In general, it takes $n-1$ times of comparisons to find maximum number in a list of n elements. Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency. As n grows larger, the n part in the expression $n-1$ dominates the complexity. The Big O notation allows you to ignore the non-dominating part (e.g., -1 in the expression $n-1$) and highlight the important part (e.g., n in the expression $n-1$). So, the complexity of this algorithm is $O(n)$.

Useful Mathematic Summations

$$1 + 2 + 3 + \dots + (n - 1) + n = \frac{n(n + 1)}{2}$$

$$a^0 + a^1 + a^2 + a^3 + \dots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1}$$

Examples: Determining Big-O

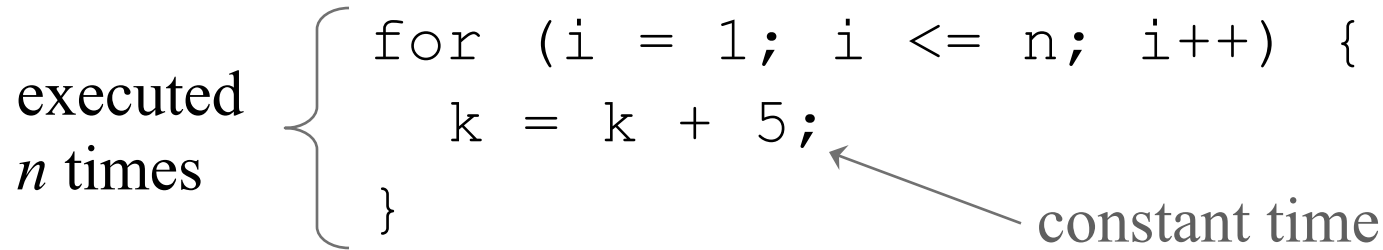
- Repetition
- Sequence
- Selection
- Logarithm

Repetition: Simple Loops

executed
 n times

```
{ for (i = 1; i <= n; i++) {  
    k = k + 5;  
}
```

constant time



Time Complexity

$$T(n) = (\text{a constant } c) * n = cn = \mathbf{O(n)}$$

Ignore multiplicative constants (e.g., “c”).



Repetition: Nested Loops

executed n times

```
{  
  for (i = 1; i <= n; i++) {  
    for (j = 1; j <= n; j++) {  
      k = k + i + j;  
    }  
  }  
}
```

inner loop executed n times

constant time

Time Complexity

$$T(n) = (\text{a constant } c) * n * n = cn^2 = O(n^2)$$

Ignore multiplicative constants (e.g., “c”).

Repetition: Nested Loops

executed n times

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= i; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop executed i times

constant time

Time Complexity

$$T(n) = c + 2c + 3c + 4c + \dots + nc = cn(n+1)/2 = (c/2)n^2 + (c/2)n = O(n^2)$$

Ignore non-dominating terms

Ignore multiplicative constants

Repetition: Nested Loops

executed n times

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= 20; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop executed 20 times

constant time

Time Complexity

$$T(n) = 20 * c * n = O(n)$$

*Ignore multiplicative constants (e.g., $20*c$)*

Sequence

executed
10 times

```
{  
    for (j = 1; j <= 10; j++) {  
        k = k + 4;  
    }  
}
```

executed
n times

```
{  
    for (i = 1; i <= n; i++) {  
        for (j = 1; j <= 20; j++) {  
            k = k + i + j;  
        }  
    }  
}
```


inner loop
executed
20 times

Time Complexity

$$T(n) = c * 10 + 20 * c * n = O(n)$$

Selection

$O(n)$



```
if (list.contains(e)) {  
    System.out.println(e);  
}  
else  
    for (Object t: list) {  
        System.out.println(t);  
    }
```

} Let n be
list.size().
Executed
 n times.

Time Complexity

$$\begin{aligned} T(n) &= \text{test time} + \text{worst-case (if, else)} \\ &= O(n) + O(n) \\ &= O(n) \end{aligned}$$

Constant Time

The Big O notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take *constant time* with the notation $O(1)$. For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.

Common Recurrence Relations

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 24)
$T(n) = 2T(n/2) + O(n \log n)$	$T(n) = O(n \log^2 n)$	
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

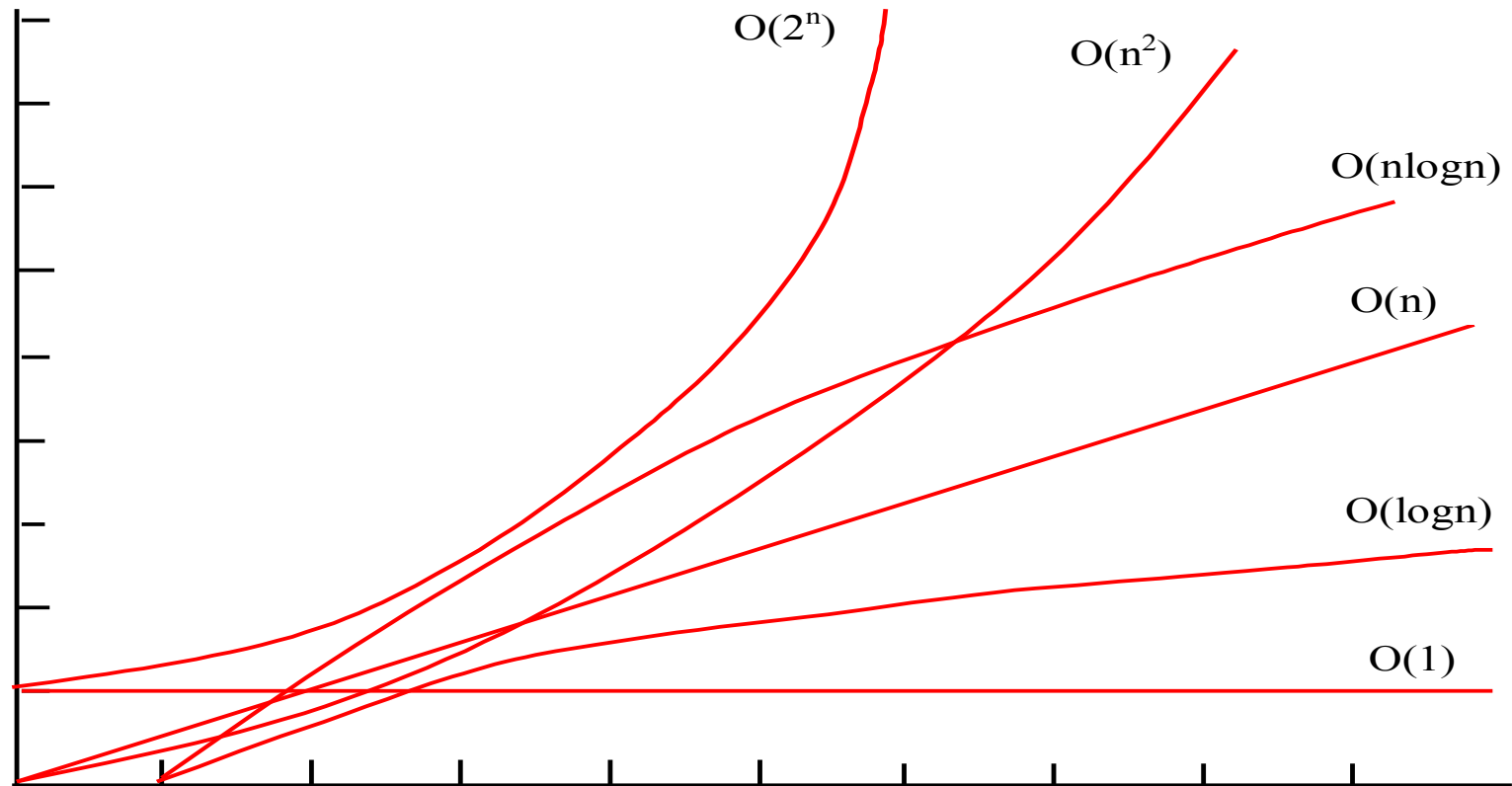
Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



Practical Considerations

The big O notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient.