

Chapter 22 - Part 1

Big-O Notation

Big-O Notation

Big-O notation is used to express the **performance** of an algorithm (that is, the amount of effort/time the algorithm takes to process N number of elements).

Big-O notation helps us classify Algorithms based on **growth rate** (**scalability**). Algorithms with the same growth rate can be represented using the same O notation.

$O(n)$ is read "Order of N". The O function is known as the Order function, which deals with approximation (**order of magnitude**).

Big-O of an algorithm is counting number of operations the algorithm performs and then expressing that in order of magnitude (Big-O) notation.

Growth Rate

Since it difficult to compare algorithms by measuring their exact execution time, growth rate is used to do so, such that we analyze algorithms independent of computers and specific inputs.

Growth rate approximates the effect of change in performance relative to the size of the input. This way we can see how fast execution time increases as the input size increases. That is, comparing two algorithms by examining their *growth rates*.

Execution time to compare algorithms has 2 problems:

- Execution time is hardware-dependent since there may be many tasks running concurrently on a computer.
- Execution time is input-dependent. For example, a *linear search* for an element happens to be the first in the list would be faster than *binary search*.

Best, Worst, and Average Case Scenarios

For same input size, an algorithm's execution time may vary, depending on the input (e.g., order of inputs).

An input that results in the shortest execution time is called the *best case* input (best case scenario analysis).

A input that results in the longest execution time is called the *worst case* input (worst case scenario analysis).

Average case analysis determines the average amount of time among all possible input of the same size.

Ignoring Multiplicative Constants

With Big-O notation, multiplicative constants are ignored since the focus is on the growth rate.

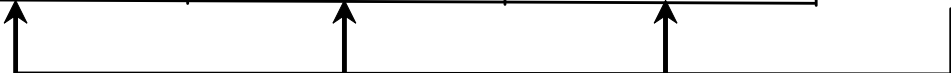
Example: In linear search of a list, we perform 1 comparison (best case), n comparisons (worst case), and $(n+1)/2$ comparisons (average case).

In Big-O notation, worst and average cases require $O(n)$ time. The multiplicative constant $(1/2)$ is omitted. It has no impact on the growth rate.

The growth rate for $n/2$ or $100n$ is the same as n .

Thus, $O(n) = O(n/2) = O(100n)$.

$f(n)$ n	n	$n/2$	$100n$	
100	100	50	1000	
200	200	100	2000	
	2	2	2	$f(200) / f(100)$



Ignoring Non-Dominating Terms

Consider finding the maximum number in an array of n elements.

If $n = 2$, it takes 1 comparison

If $n = 3$, it takes 2 comparisons

If $n = 4$, it takes 3 comparisons

So, for n , it takes $n-1$ comparisons $\implies f(n) = n-1$;

As n grows larger, the n part of the expression $n-1$ dominates the complexity.

Thus, Big-O notation ignores the non-dominating term (-1), and highlights the dominating term (n). So, the complexity of this algorithm is $O(n)$.

$$F(n) = 3n^2 - 10n + 50; \quad \implies$$

$$F(n) = 5n - 3n^3 - 10; \quad \implies$$

$$F(n) = 75 - 3^n + 10n; \quad \implies$$

$$F(n) = n + n \log n + 10; \quad \implies$$

Useful Mathematic Summations

Useful summation functions for algorithm analysis include:

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

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

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

Big-O Notation - Examples

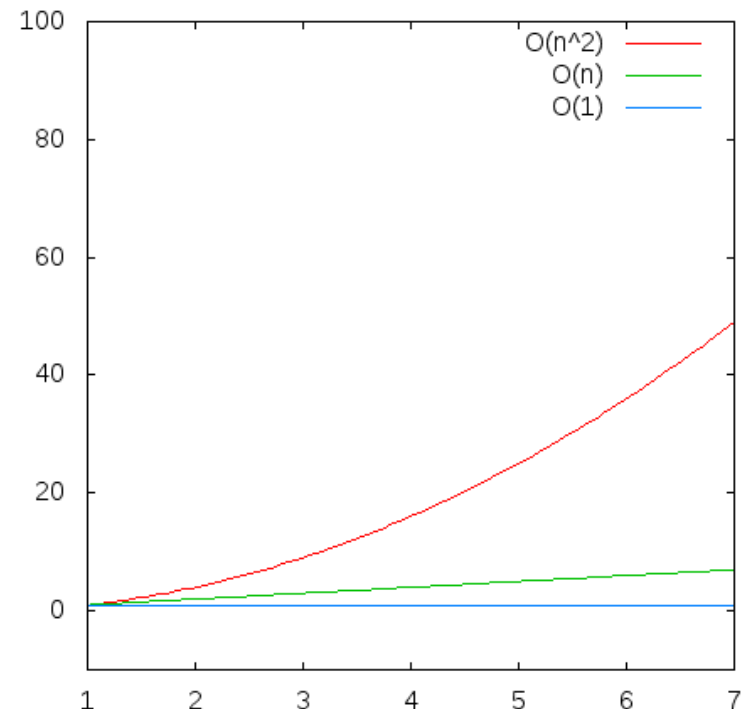
Examples:

For an algorithm with $O(1)$ (called **constant time**), it means that the algorithm always takes the same amount of time to process inputs regardless the number of input elements.

For an algorithm with $O(n)$, it means that for every input item the algorithm makes 1 operation. Thus, $n * 1 = n$.

For an algorithm with $O(n^2)$, it means that for every input item the algorithm makes n more operations.

Thus, $n * n = n^2$.



Big-O Calculation

```
function CountAGrades(list) //count number of A's
{
    total_A = 0; //set once
    for (each grade in the list) //loop n times
        if (grade == 'A') //apply best/worst case analysis
            total_A = total_A + 1;
    return total; //done once
}
```

Set total to zero.	→ 1 time
Loop through list.	→ n times
Check if grade equals A.	→ 1 time
Increment total by one as needed	→ 0 or 1 time // best/worst case scenario
Return statement	→ 1 time

We have: $F(n) = 1 + n * (1 + 1) + 1 = 2 + 2n$ OR $F(n) = 1 + n * (1 + 0) + 1 = 2 + n$

In Big-O notation, we only look at the “biggest term” in the expression. Thus, we have **$2n$** .

Since Big-O is an approximation, we ignore the 2 because **$2n$** and **n** are not fundamentally different (linear growth). So, we conclude that it is **$O(n)$** .

Big-O Calculation

```
function CreatArray(n) //matrix initialization
{
    int[][] myArray = new int[n][n]; //nxn matrix
    for i = 0 to n-1
        for j = 0 to n-1
            myArray[i][j] = i+j;
}
```

Create myArray → 1 time
Loop i → n times
Loop j → n times
Set myArray[i,j] → 1 time

We have: $F(n) = 1 + n * n * 1 = 1 + n^2$

Considering the biggest term in the expression, we have $O(n^2)$.

Big-O Calculation

```
function CompareArrays(A,B) //A and B are same length arrays
{
    for i = 0 to A.length-1 //assume A has n elements
    {
        int j = 0;
        while (A[i] != B[j]) //stop when A[i] is found in B
        {
            print A[j];
            j = j + 1;
        }
    }
}
```

Loop I	→ n times	While loop	→ 0 to n times	//best/worst case scenarios
Set j=0	→ 1 time	print A[j]	→ 1 time	
		Increment j	→ 1 time	

We have: $F1(n) = n * (1 + (0 * (1+1))) = n*(1) = n$ // best case scenario

$F2(n) = n * (1 + (n * (1+1))) = n*(1+2n) = n+2n^2$ // worst case scenario

On average: $F(n) = (F1+F2)/2 = (2n+2n^2)/2 = n+n^2$

Considering the biggest term in the expression, we have $O(n^2)$.

Big-O Calculation

```
function PrtintHalfN(n) //print half n at a time
{
    int i = n;
    while (i >= 1) {
        print i;
        i = i/2;
    }
}
```

$$\begin{aligned} 2^x &= N \\ \log(2^x) &= \log(N) \\ x \log(2) &= \log(N) \\ x &= \log(N) / \log(2) \\ x &= \log(N) \end{aligned}$$

Set i to n → 1 time

While loop → x times such that $2^x = n$ (solve for x)

Print i → 1 time

Divide i by 2 → 1 time

We have: $F(n) = 1 + (x * (1+1)) = 1 + 2x$

$$F(n) = 1 + 2 * \log(n)$$

Considering the biggest term in the expression, we have $O(\log(n))$.

Stack Operations Complexity

Array Implementation Operations	Average-Case Complexity
Constructor, <code>push()</code> , <code>pop()</code> , <code>top()</code> , <code>isEmpty()</code> , <code>isFull()</code> , <code>clear()</code> , <code>size()</code> (if size is stored in a variable)	$O(1)$ (no loop)
Printing, copying, comparison, <code>size()</code> (if size uses loop)	$O(N)$ (must loop n times)

Linked-List Implementation Operations	Average-Case Complexity
Constructor, <code>push()</code> , <code>pop()</code> , <code>top()</code> , <code>isEmpty()</code> , <code>size()</code> (if size stored in a variable)	$O(1)$ (no loop)
Printing, copying, comparison, <code>clear()</code> , <code>size()</code> (if size uses loop),	$O(N)$ (must loop n times)

Other Examples - Textbook

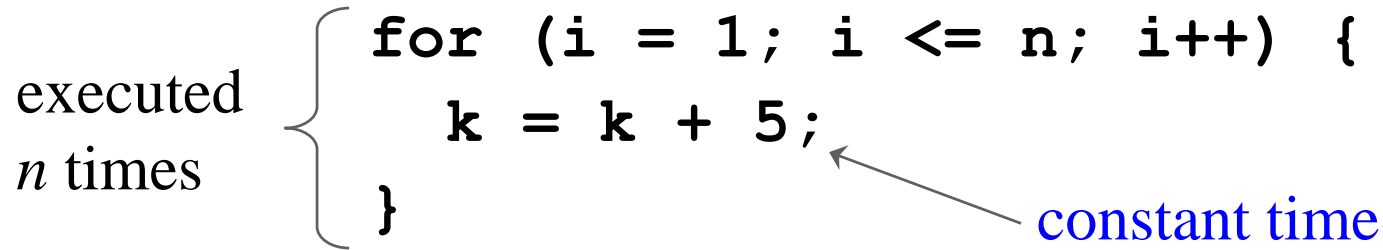
- Repetition
- Sequence
- Selection
- Logarithm

Repetition: Simple Loops

executed
 n times

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

constant time



Time Complexity

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

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

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

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

Repetition: Nested Loops

executed
n times

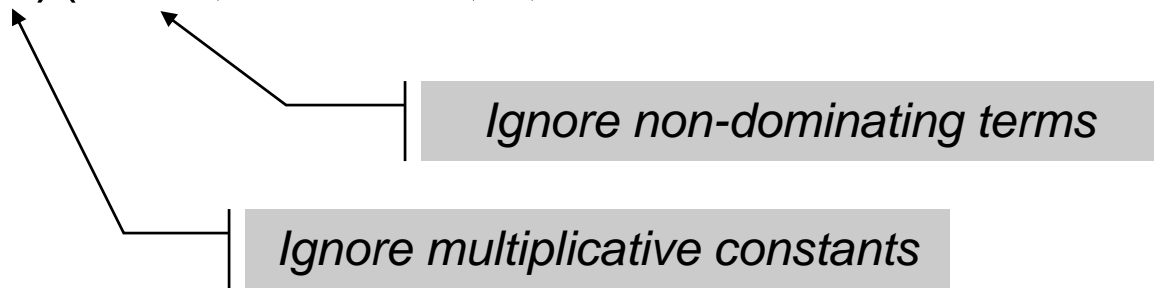
```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= i; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop
executed
i times

constant time *c*

Time Complexity

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



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;  
    }  
  }  
}
```

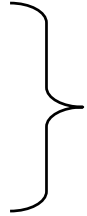
inner loop
executed
20 times

Time Complexity

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

Selection

$O(n)$  `if (list.contains(e)) { //this is a loop
 System.out.println(e);
}
else
 for (Object t: list) {
 System.out.println(t);
 }`

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

Time Complexity

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

Logarithmic

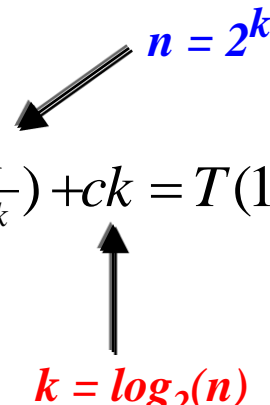
Binary Search:

Search a sorted list of n elements for a key value. Binary search eliminates half of the input after two comparisons.

Each iteration contains a fixed number of operations (denoted by c).

Assume $T(n)$ is the time complexity for searching n elements.

Assume $n = 2^k$, Thus $k = \log_2(n)$.

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = \dots = T\left(\frac{n}{2^k}\right) + ck = T(1) + c \log n = 1 + c \log n$$


$$T(n) = O(\log n)$$

Note: Logarithmic algorithms grow slowly as the input size increases. If you square the input size, you only double the time it takes to process the inputs.

Example: Performance Function

```
//Given array A of length n
FOR I = 2 to n
    FRO J = n to I
        IF (A[J] < A[J-1]) //comparison operation
            swap (A[J], A[J+1]) //swap operation
    End Loop
End Loop
```

Conduct best case and worst case performance analysis and derive the average performance function for this algorithm. Then determine the Big-O notation for it.

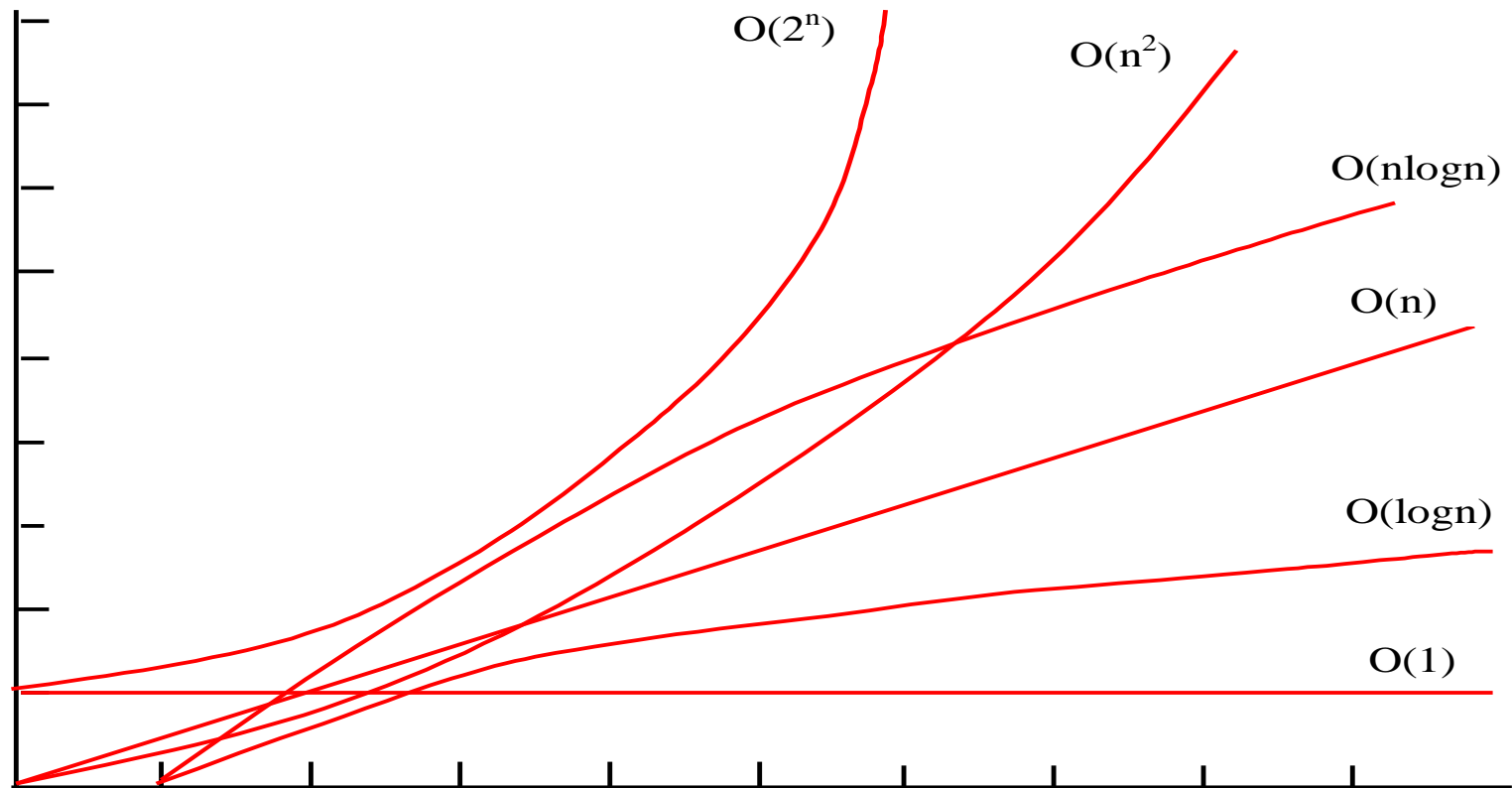
Comparing Growth Functions

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

$O(1)$	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 Growth Functions

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



End of Slides