

# Searching and Algorithms Analysis

# 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

- Analyze algorithms independent of computers and specific input.
- This approach approximates the effect of a change on the size of the input.
- How fast an algorithm's execution time increases as the input size increases.
- Compare two algorithms by examining their *growth rates*.

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

| $f(n)$<br>$n$ | $n$ | $n/2$ | $100n$ |
|---------------|-----|-------|--------|
| 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

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

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

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

# Check Point

□ What is the order of each of the following:

$$- \frac{(n^2+1)^2}{n}$$

$$- \frac{n^2 + (\log 2)n^4}{n}$$

$$- n^3 + 100n^2 + n$$

$$- 2^n + 100n^2 + 45n$$

$$- n + 2^n + 2^n 2^n$$

# Examples: Determining Big-O

- Repetition
- Sequence
- Selection
- Logarithm

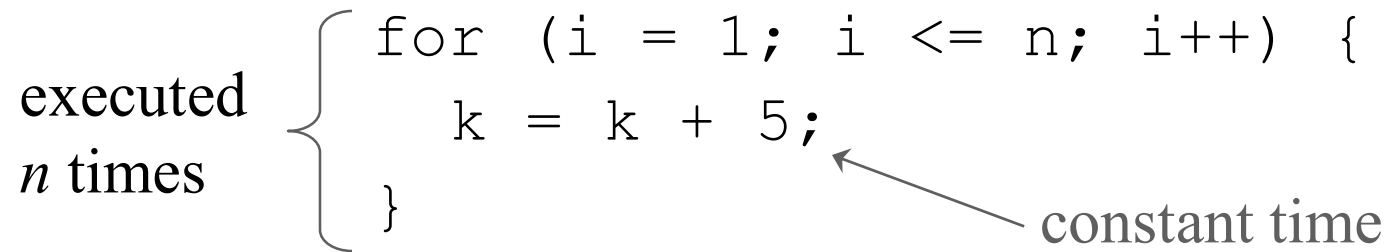


# Repetition: Simple Loops

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 <= 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;  
    }  
}
```

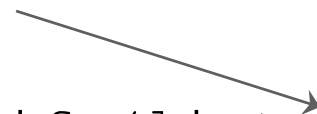
inner loop  
executed  
*20* times

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

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

# Logarithm: Analyzing Binary Search

$$n = 2^k$$

The binary search algorithm presented in Lecture, BinarySearch.java, searches a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by  $c$ . Let  $T(n)$  denote the time complexity for a binary search on a list of  $n$  elements.

Without loss of generality, assume  $n$  is a power of 2 and  $k = \log n$ .

Since binary search eliminates half of the input after two comparisons,

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

# Check Point

□ Count the number of iterations in the following loops:

a) `int cnt = 1;`  
    `while ( cnt < 30 )`  
        `cnt = cnt * 2;`

b) `int cnt = 15;`  
    `while ( cnt < 30 )`  
        `cnt = cnt * 3;`

c) `int cnt = 1;`  
    `while ( cnt < n )`  
        `cnt = cnt * 2;`

b) `int cnt = 15;`  
    `while ( cnt < n )`  
        `cnt = cnt * 3;`

# Check Point

- Use the Big O notation to estimate the time complexity of the following methods:

```
□ public static void m(int n)
{
    for( int i=0; i<n; ++i){
        System.out.println(Math.random());
    }
}

□ public static void m(int n)
{
    for( int i=0; i<n; ++i){
        for(int j=0; j<i; ++j)
            System.out.println(Math.random());
    }
}
```



# Check Point

```
(c) public static void mC(int[] m) {  
    for (int i = 0; i < m.length; i++) {  
        System.out.print(m[i]);  
    }  
    for (int i = m.length - 1; i >= 0; ) {  
        System.out.print(m[i]); i--; }  
}
```

```
(d) public static void mD(int[] m) {  
    for (int i = 0; i < m.length; i++) {  
        for (int j = 0; j < i; j++)  
            System.out.print(m[i] * m[j]);  
    }  
}
```

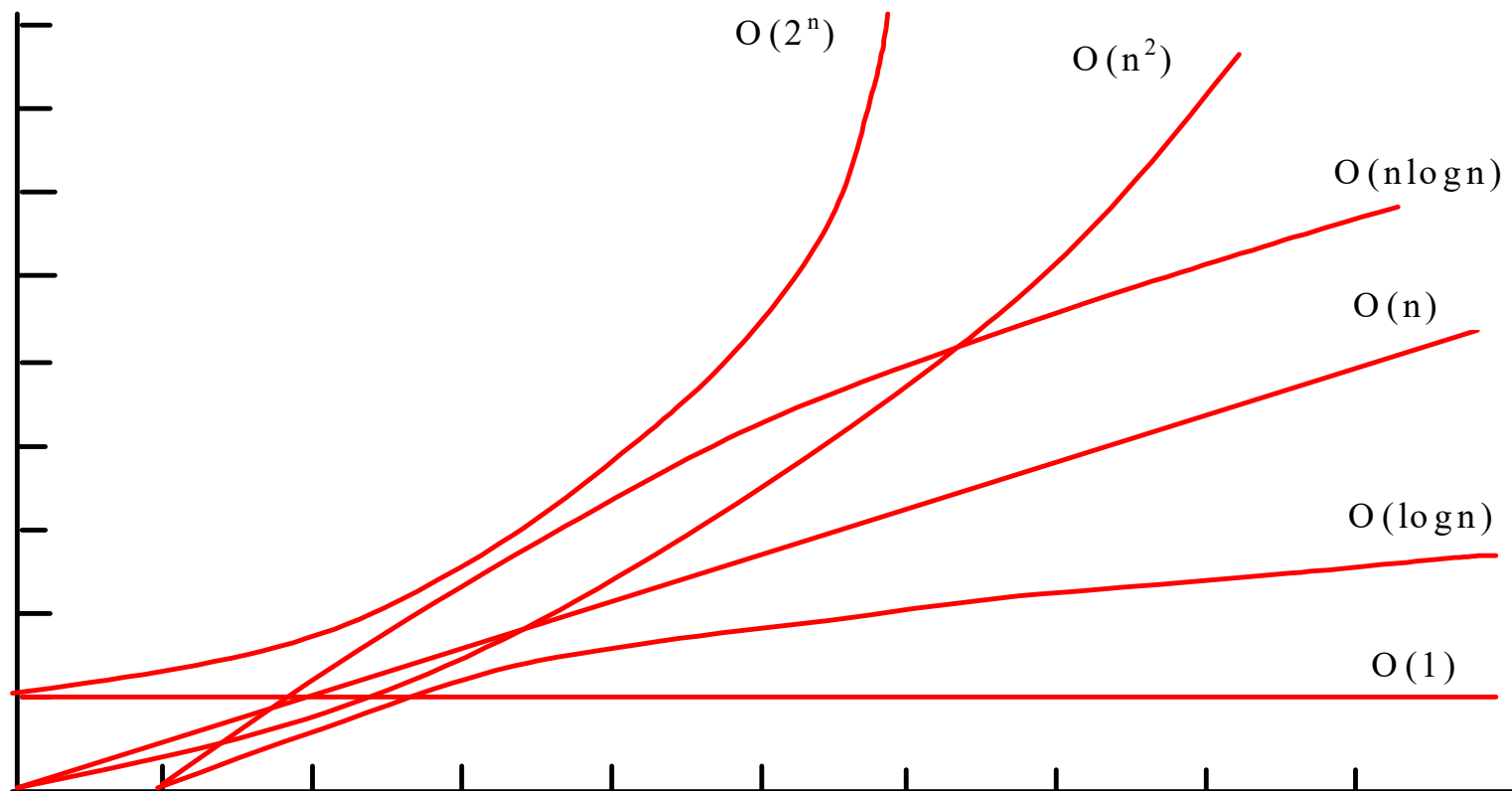
# Comparing Common Growth Functions

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

|               |                  |
|---------------|------------------|
| $O(1)$        | 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

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



# Analyzing Tower of Hanoi

The Tower of Hanoi problem presented in Listing 18.7, TowerOfHanoi.java, moves  $n$  disks from tower A to tower B with the assistance of tower C recursively as follows:

- Move the first  $n - 1$  disks from A to C with the assistance of tower B.
- Move disk  $n$  from A to B.
- Move  $n - 1$  disks from C to B with the assistance of tower A.

Let  $T(n)$  denote the complexity for the algorithm that moves disks and  $c$  denote the constant time to move one disk, i.e.,  $T(1)$  is  $c$ . So,

$$\begin{aligned} T(n) &= T(n-1) + c + T(n-1) = 2T(n-1) + c \\ &= 2(2(T(n-2) + c) + c) = 2^{n-1}T(1) + c2^{n-2} + \dots + c2 + c = \\ &= c2^{n-1} + c2^{n-2} + \dots + c2 + c = c(2^n - 1) = O(2^n) \end{aligned}$$

# Case Study: Fibonacci Numbers

```
/** The method for finding the Fibonacci number */  
public static long fib(long index) {  
    if (index == 0) // Base case  
        return 0;  
    else if (index == 1) // Base case  
        return 1;  
    else // Reduction and recursive calls  
        return fib(index - 1) + fib(index - 2);  
}
```

# Case Study: Non-recursive version of Fibonacci Numbers

```
public static long fib(long n) {  
    long f0 = 0; // For fib(0)  
    long f1 = 1; // For fib(1)  
    long f2 = 1; // For fib(2)  
  
    if (n == 0)  
        return f0;  
    else if (n == 1)  
        return f1;  
    else if (n == 2)  
        return f2;  
  
    for (int i = 3; i <= n; i++) {  
        f0 = f1;  
        f1 = f2;  
        f2 = f0 + f1;  
    }  
  
    return f2;  
}
```

# Case Study: GCD Algorithms

## Version 1

```
public static int gcd(int m, int n) {  
    int gcd = 1;  
    for (int k = 2; k <= m && k <= n; k++) {  
        if (m % k == 0 && n % k == 0)  
            gcd = k;  
    }  
    return gcd;  
}
```

$O(n)$

# Case Study: GCD Algorithms

## Version 2

```
for (int k = n; k >= 1; k--) {  
    if (m % k == 0 && n % k == 0) {  
        gcd = k;  
        break;  
    }  
}
```



# Euclid's Algorithm Implementation

```
public static int gcd(int m, int n) {  
    if (m % n == 0)  
        return n;  
    else  
        return gcd(n, m % n);  
}
```