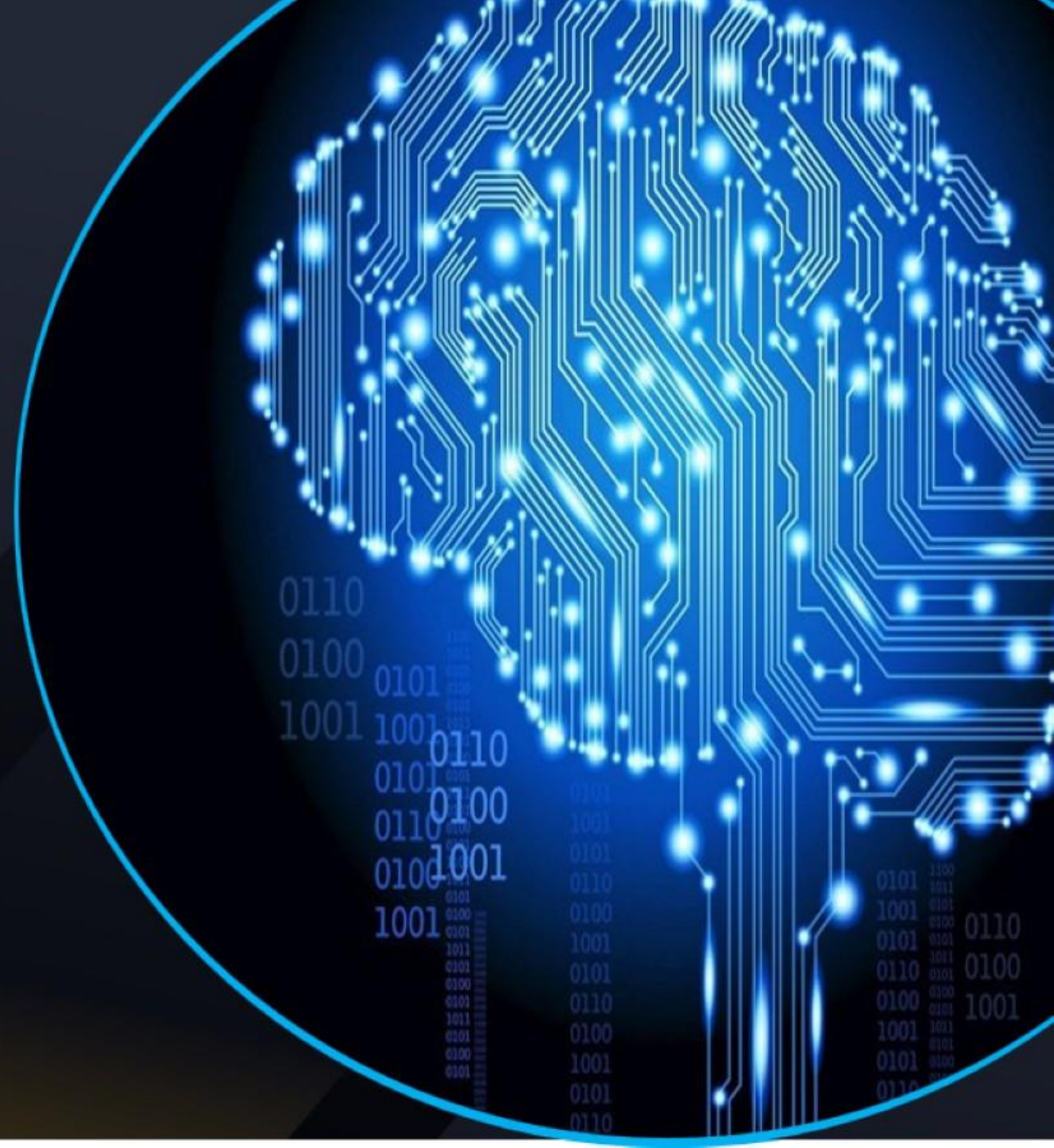


Discrete Mathematics

Dr. Shahira Ahmed

IT241

Lecture 10





Discrete Mathematics

Lecture 10

Time Complexity Analysis: Techniques for
Recursion, Recurrence, and Growth
Functions



LEARNING OUTCOMES

- ◉ Define and explain the concept of time complexity and its significance in algorithm design.
- ◉ Understand and apply asymptotic notation, such as big O notation, to express and analyze the growth rate of algorithms.
- ◉ Comprehend the principles and benefits of recursion in algorithmic problem-solving.
- ◉ Analyze the time complexity of recursive algorithms using appropriate techniques.



LEARNING OUTCOMES

- ◉ Identify and solve recurrence relations using the master theorem.
- ◉ Compare and contrast various growth functions, including constant, logarithmic, linear, and polynomial functions.
- ◉ Understand proof techniques.



LECTURE OUTLINE

- ◉ The Growth of Functions
- ◉ Time Complexity
- ◉ Recursive Function
- ◉ Recurrence using Master theorem

How Can I Measure Efficiency of An algorithm?

Time complexity analysis is a measure of:

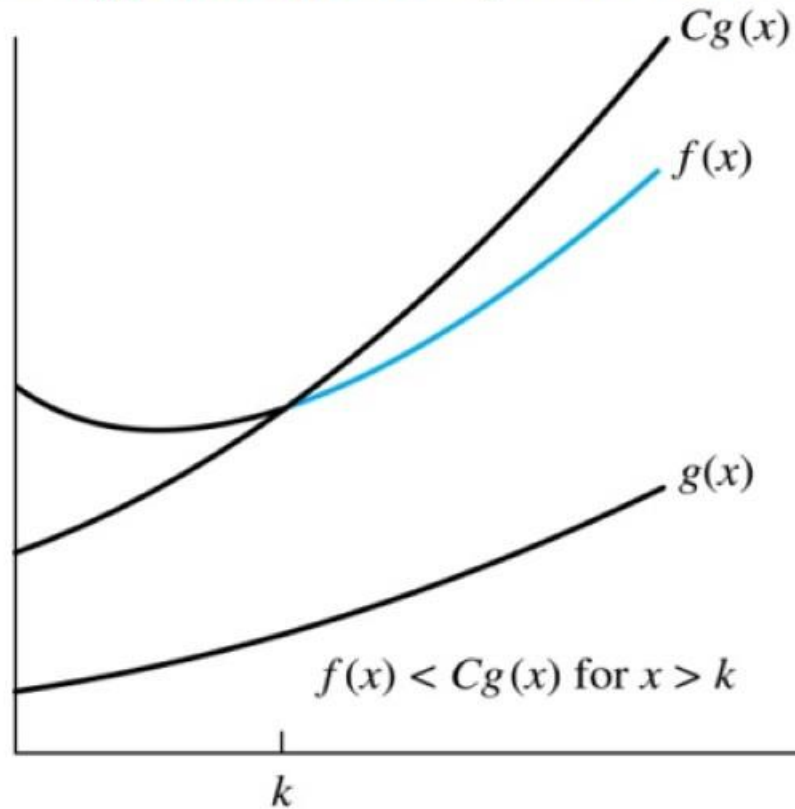
The **algorithm's growth** rate as the **input size increases**

The **number of operations or steps** the algorithm will take to complete its task

- In both computer science and in mathematics, there are many times when we care about how fast a function grows.
- In computer science, we want to understand how quickly an algorithm can solve a problem as the size of the input grows.
 - We can compare the efficiency of two different algorithms for solving the same problem.
 - We can also determine whether it is practical to use a particular algorithm as the input grows.

Illustration of Big- O Notation

Big- O gives an **upper bound** on the growth of a function



$$f(x) \text{ is } O(g(x))$$

The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

- **f and g be functions** from the set of integers or the set of real numbers to the set of real numbers. We say **that $f(x)$ is $O(g(x))$** if there are constants C and k such that whenever $x > k$

- This is read as “ **$f(x)$ is big- O of $g(x)$** ” or “ **g asymptotically dominates f .**”

$$|f(x)| \leq C|g(x)|$$

- The constants C and k are called *witnesses* to the relationship $f(x)$ is $O(g(x))$. **Only one** pair of witnesses is needed.

Using the Definition of Big- O Notation

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

Solution: Since when $x > 1$, $x < x^2$ and $1 < x^2$

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

- Can take $C = 4$ and $k = 1$ as witnesses to show that

$$f(x) \text{ is } O(x^2)$$

You may see “ $f(x) = O(g(x))$ ” instead of “ $f(x)$ **is** $O(g(x))$.”

- But this is an abuse of the equals sign since the meaning is that there is an inequality relating the values of f and g , for sufficiently large values of x .
- It is ok to write $f(x) \in O(g(x))$, because $O(g(x))$ represents the set of functions that are $O(g(x))$.

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$

- if there are constants C and k such that $|f(x)| \geq C|g(x)|$ when $x > k$. We say that “ $f(x)$ is big-Omega of $g(x)$.”
- Big- O gives an upper bound on the growth of a function, while Big-Omega gives a lower bound. Big-Omega tells us that a function grows at least as fast as another.

- Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. The function $f(x)$ is $\Theta(g(x))$
- if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.
- We say that “ f is big-Theta of $g(x)$ ” and also that “ $f(x)$ is of *order* $g(x)$ ” and also that “ $f(x)$ and $g(x)$ are of the *same order*.”
- $f(x)$ is $\Theta(g(x))$ if and only if there exists constants C_1, C_2 and k such that $C_1 g(x) < f(x) < C_2 g(x)$ if $x > k$. This follows from the definitions of big- O and big- Ω .

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size? To answer this question, we ask:
 - How much **time** does this algorithm use to solve a problem?
 - How much **computer memory** does this algorithm use to solve a problem?

- When we analyze the **time** the algorithm uses to solve the problem given **input of a particular size**, we are studying the *time complexity* of the algorithm.
- When we analyze the **computer memory** the algorithm uses to solve the problem given **input of a particular size**, we are studying the *space complexity* of the algorithm.

- We will measure time complexity in terms of the number of operations an algorithm uses and we will use **big- O** and **big-Theta** notation to estimate the time complexity.
- We can use this analysis to see whether it is **practical** to use this algorithm to solve problems with input of a particular size. We can also compare the **efficiency** of different algorithms for solving the same problem.

- To analyze the time complexity of algorithms, we determine **the number of operations**, such as comparisons and arithmetic operations (addition, multiplication, etc.). We **ignore minor details**, such as the “house keeping” aspects of the algorithm.
- We will focus on the *worst-case time* complexity of an algorithm. This provides an **upper bound on the number of operations an algorithm** uses to solve a problem with input of a particular size.
- It is usually much more difficult to determine the *average case time complexity* of an algorithm. This is the average number of operations an algorithm uses to solve a problem over all inputs of a particular size.

Example: Describe the time complexity of the algorithm for finding the maximum element in a finite sequence.

```
procedure max( $a_1, a_2, \dots, a_n$ : integers)
    max :=  $a_1$ 
    for  $i := 2$  to  $n$ 
        if  $max < a_i$  then  $max := a_i$ 
    return max{max is the largest element}
```

the time complexity of the algorithm is $\Theta(n)$.

Worst-Case Complexity of Linear Search

Example: Determine the time complexity of the linear search algorithm.

```
procedure linear search(x:integer,  
                        a1, a2, ..., an: distinct integers)  
  i := 1  
  while (i ≤ n and x ≠ ai)  
    i := i + 1  
  if i ≤ n then location := i  
  else location := 0  
  return location {location is the subscript of the term that  
                   equals x, or is 0 if x is not found}
```

the time complexity of the algorithm is $\Theta(n)$.

Worst-Case Complexity of Binary Search

Example: Describe the time complexity of binary search in terms of the number of comparisons used.

```
procedure binary search( $x$ : integer,  $a_1, a_2, \dots, a_n$ : increasing integers)
 $i := 1$  { $i$  is the left endpoint of interval}
 $j := n$  { $j$  is right endpoint of interval}
while  $i < j$ 
     $m := \lfloor (i + j)/2 \rfloor$ 
    if  $x > a_m$  then  $i := m + 1$ 
    else  $j := m$ 
if  $x = a_i$  then  $location := i$ 
else  $location := 0$ 
return  $location$  { $location$  is the subscript  $i$  of the term  $a_i$  equal to  $x$ , or 0 if  $x$  is not found}
```

the time complexity of the algorithm is $\Theta(\log n)$.

```
for i in range(n):  
    x++  
for x in range(n):  
    y++
```

$\Theta(n)$

```
for i in range(n):  
    for x in range(n):  
        y++
```

$\Theta(n^2)$

```
for i in range(n):  
    for x in range(n):  
        for s in range(n):  
            y++
```

$\Theta(n^3)$

```
for i in range(1000):  
    x++  
for x in range(n):  
    y++
```

$\Theta(n)$

```
for i in range(1000):  
    for x in range(n):  
        y++
```

$\Theta(n)$

```
for i in range(1000):  
    for x in range(n):  
        for s in range(n):  
            y++
```

$\Theta(n^2)$

```
for i in range(1000):  
    x++  
for x in range(1000):  
    y++
```

$\Theta(1)$

```
for i in range(1000):  
    for x in range(1000):  
        y++
```

$\Theta(1)$

```
for i in range(1000):  
    for x in range(1000):  
        for s in range(9):  
            y++
```

$\Theta(1)$

Worst-Case Complexity of Bubble Sort

What is the worst-case complexity of bubble sort in terms of the number of comparisons made?

```
procedure bubblesort( $a_1, \dots, a_n$ : real numbers with  $n \geq 2$ )  
  for  $i := 1$  to  $n - 1$   
    for  $j := 1$  to  $n - i$   
      if  $a_j > a_{j+1}$  then interchange  $a_j$  and  $a_{j+1}$   
  { $a_1, \dots, a_n$  is now in increasing order}
```

Solution: A sequence of $n-1$ passes is made through the list. On each pass $n-i$ comparisons are made.

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

The worst-case complexity of bubble sort is $\Theta(n^2)$

.

Worst-Case Complexity of Insertion Sort

What is the worst-case complexity of insertion sort in terms of the number of comparisons made?

Solution: The total number of comparisons are:

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

Therefore the complexity is $\Theta(n^2)$.

```
procedure insertion sort( $a_1, \dots, a_n$ :  
    real numbers with  $n \geq 2$ )  
  for  $j := 2$  to  $n$   
     $i := 1$   
    while  $a_j > a_i$   
       $i := i + 1$   
     $m := a_j$   
    for  $k := 0$  to  $j - i - 1$   
       $a_{j-k} := a_{j-k-1}$   
     $a_i := m$ 
```

Worst-Case Complexity of Selection Sort

What is the worst-case complexity of Selection sort in terms of the number of comparisons made?

the complexity is $\Theta(n^2)$.

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

<i>Complexity</i>	<i>Terminology</i>
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

<i>Complexity</i>	<i>Terminology</i>
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Understanding the Complexity of Algorithms

TABLE 2 The Computer Time Used by Algorithms.

<i>Problem Size</i>	<i>Bit Operations Used</i>					
<i>n</i>	$\log n$	<i>n</i>	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

Times of more than 10^{100} years are indicated with an *.

Definition: A *recursive or inductive definition* of a function consists of two steps.

A Recursive function can be defined as a routine that calls itself directly or indirectly.

- **BASIS STEP:** Specify the value of the function at zero.
- **RECURSIVE STEP:** Give a rule for finding its value at an integer from its values at smaller integers.

Example: Given a recursive definition of the factorial function $n!$:

$$f(0) = 1$$

$$f(n + 1) = (n + 1) \cdot f(n), \text{ Find } f(1), f(2), f(3)$$

Solution

- $f(1) = 1 * f(0) = 1$
- $f(2) = 2 * f(1) = 2$
- $f(3) = 3 * f(2) = 6$

Example: Suppose f is defined by:

$$f(0) = 3,$$

$$f(n + 1) = 2f(n) + 3$$

Find $f(1), f(2), f(3), f(4)$

Solution:

- $f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9$
- $f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21$
- $f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45$
- $f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93$

Divide-and-Conquer Recurrence Relations

- Suppose that a recursive algorithm divides a **problem of size n** into a subproblems.
- Assume each **subproblem** is of size n/b .
- Suppose $g(n)$ **extra operations are needed in the conquer step**.
- Then $f(n)$ represents the number of operations to solve a problem of size n satisfies the following recurrence relation:

$$f(n) = af(n/b) + g(n)$$

- This is called a *divide-and-conquer recurrence relation*.

Theorem 2. Master Theorem: Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Example 1

- $f(n) = af(n/b) + cn^d$
- $f(n) = 4f\left(\frac{n}{2}\right) + (cn^1)$
- $a=4$
- $b=2$
- $d=1$
- Since $a > b^d$
- $T(n) = O(n^{\log_b a}) = O(n^2)$

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Example 2

- $f(n) = af(n/b) + cn^d$
- $f(n) = f\left(\frac{n}{2}\right) + (cn^0)$

Solution

- $a=1$
- $b=2$
- $d=0$
- $b^d=1$
- Since $a = b$
- $T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Proof techniques refer to the methods employed in mathematics and logic to establish the validity or truth of a statement or proposition. **some** of the common proof techniques:

- 1. **Direct Proof**: This technique involves starting with the **given statements** and using logical reasoning to reach the desired conclusion.
- 2. **Proof by Contradiction**: In this technique, the proof assumes the **negation** of the desired conclusion and then shows that it leads to a contradiction or inconsistency. This establishes the original statement's validity.
- 3. **Proof by Contrapositive**: This technique involves proving the **contrapositive** of a statement instead of the original statement. The contrapositive is logically equivalent to the original statement, so proving it establishes the truth of the original statement.

- 4. **Proof by Mathematical Induction:** This technique is commonly used to prove statements that depend on a variable, typically natural numbers. It involves proving that a given statement holds for a base case (often $n = 1$) and then showing that if it holds for any arbitrary case ($n = k$), it also holds for the next case ($n = k + 1$).
- 5. **Proof by Exhaustion:** Also known as proof by cases, this technique involves considering all possible cases or scenarios and proving the statement for each specific case.
- 6. **Proof by Counterexample:** This technique aims to disprove a statement by providing a specific example or case that contradicts it. If a counterexample is found, it demonstrates that the statement is false.

Recap

1. Time complexity of **linear** search
2. Time complexity of **binary** search
3. Time complexity of **insertion** sort
4. Time complexity of **bubble** sort
5. Time complexity of **selection** sort
6. Time complexity of **merge** sort

Solutions

1. $\Theta(n)$ 2. $\Theta(\log n)$ 3. $\Theta(n^2)$ 4. $\Theta(n^2)$ 5. $\Theta(n^2)$ 6. $\Theta(n \log n)$

1. -----used to measure the **time** complexity of an algorithm?
2. -----used to measure the **space** complexity of an algorithm?
3. provides an **upper** bound on the time complexity of an algorithm.
4. provides a **lower** bound on the time complexity of an algorithm.
5. provides the **same order on** the time complexity of an algorithm.

Solution:

1. number of operations 2. Computer memory 3) Big-O 4) Omega 5) Theta

Three algorithms A, B, C with the following time complexity Which is the fastest algorithm

1. $A = \theta(1)$ $B = \theta(n)$ $C = \theta(n^2)$
2. $A = \theta(2^n)$ $B = \theta(n)$ $C = \theta(\log n)$
3. $A = \theta(1)$ $B = \theta(5^n)$ $C = \theta(n \log n)$
4. $A = \theta(2^n)$ $B = \theta(n^2)$ $C = \theta(n \log n)$
5. $A = \theta(n^2)$ $B = \theta(n^3)$ $C = \theta(n^4)$
6. $A = \theta(2^n)$ $B = \theta(n)$ $C = \theta(n \log n)$

Solution

1. A 2. C 3. A 4. C 5. A 6. B

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	Terminology
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Example : The Fibonacci numbers are defined as follows:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Find f_2, f_3, f_4, f_5 , and f_6

- $f_2 = f_1 + f_0 = 1 + 0 = 1,$
- $f_3 = f_2 + f_1 = 1 + 1 = 2,$
- $f_4 = f_3 + f_2 = 2 + 1 = 3,$
- $f_5 = f_4 + f_3 = 3 + 2 = 5,$
- $f_6 = f_5 + f_4 = 5 + 3 = 8.$

- 1) Given an algorithm, how **efficient** is this algorithm for solving a problem given input of a particular size? We use-----,-----
- 2) If there exists constants C_1 , C_2 and k such that $C_1 < g(x) < f(x) < C_2 g(x)$ if $x > k$
- 3) if there are constants C and k such that $|f(x)| \leq C|g(x)|$ when $x > k$.
- 4) if there are constants C and k such that $|f(x)| \geq C|g(x)|$ when $x > k$.

Solution : 1) Time complexity – Space complexity 2) Theta 3) Big-O 4) Omega

- $f(n) = af(n/b) + cn^d$
- $f(n) = f\left(\frac{n}{2}\right) + c$

Solution

- $a=1$
- $b=2$
- $d=0$
- $b^d=1$
- Since $a = b^d$
- $T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Binary Search

- $f(n) = af\left(\frac{n}{b}\right) + cn^d$
- $f(n) = 2f\left(\frac{n}{2}\right) + cn$
- $a=2$
- $b=2$
- $d=1$
- $b^d=2$
- Since $a = b^d$
- $O(n^d \log n) = O(n^1 \log n)$

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Merge Sort

- $f(n) = af(n/b) + cn^d$
- $f(n) = 2f(n/2) + cn^2$
- $a = 2$
- $b = 2$
- $d = 2$
- $b^d = 4$
- Since $a < b^d$
- $T(n) = O(n^d) = O(n^2)$

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

- $f(n) = af(n/b) + cn^d$
- $f(n) = 3f\left(\frac{n}{2}\right) + (cn^1)$
- $a = 3$
- $b = 2$
- $d = 1$
- $b^d = 2$
- *Since $a > b^d$*
- $T(n) = O(n^{\log_b a}) = O(n^{\log_2 3})$

$$f(n) \text{ is } \begin{cases} O(n^d) & \text{if } a < b^d, \\ O(n^d \log n) & \text{if } a = b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

- Give big O

1)

```
t := 0
for i := 1 to 3
  for j := 1 to 4
    t := t + ij
```

2)

```
m := 0
for i := 1 to n
  for j := i + 1 to n
    m := max(aiaj, m)
```

- Suppose that an element is known to be among the first four elements in a list of 32 elements. Would a linear search or a binary search locate this element in a sorted list more rapidly?
- Solution 1.O(1) 2.O(n²) 3.Linear

- 1. This technique involves starting with the **given statements** and using logical reasoning to reach conclusion.
- 2. the proof assumes the **negation** of the desired conclusion and then shows that it leads to an inconsistency.
- 3. proving the **contrapositive** of a statement instead of the original statement
- Solution:
- 1.Direct Proof 2. Proof by Contradiction 3. Proof by Contrapositive

- 4. prove statements that depend on a variable, typically natural numbers.
 - 5. consider all possible cases and proving the statement for each case.
 - 6. disprove a statement by providing a specific example that contradicts it
 - Solution
1. Proof by Mathematical Induction 2. Proof by Exhaustion 3. Proof by Counterexample

*Thank
you*

