DESIGN AND ANALYSIS OF ALGORITHMS

Introduction

- What is Algorithm?

Introduction

- What is Algorithm?
  - An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

Introduction…

- Characteristics of an Algorithm

Introduction…

- Characteristics of an Algorithm
  - **Input: ?**
  - **Output: ?**
  - **Definiteness: ?**
  - **Finiteness: ?**
  - **Effectiveness: ?**

Introduction…

- Characteristics of an Algorithm
  - **Input: zero or more inputs**, taken from a specified set of objects
  - **Output: At least one quantity** is produced relation to the inputs
  - **Definiteness:** Each instruction must be **precisely defined**
  - **Finiteness:** It terminates after a **finite number of steps**
  - **Effectiveness:** All operations to be performed must be **sufficiently basic** that they can be done exactly and in finite length.

Algorithms vs Programs

Algorithms	**Programs**
– Design Level	Implementation Level
– Domain Knowledge	Programmer
– Any Language	Programming Language
– H/W & OS	H/W & OS
– Analyse	Testing

Analysis of algorithms

Analysis of algorithms

- Measuring efficiency of an algorithm
  - Time : How long the algorithm takes (running time)
  - Space : Memory requirement

Time and space

Time and space

- Time depends on processing speed
  - Not possible to change for given hardware
- Space is a function of available memory
  - Easier to reconfigure
- Typically, we will focus on time, not space

Time and space complexity

- Algorithm swap(a,b)

```
{
        Temp=a;
        a=b;
        b=Temp;
}
```

Time and space complexity

- Algorithm swap(a,b)

```
{
        Temp=a; ………………………………………….(1)
        a=b; ……………………………………………………(2)
        b=Temp; …………………………………………..(3)
}
```

Total Time= 3 Units

Constant=O(1)

Time and space complexity

- Algorithm swap(a,b)

```
{
        Temp=a;
        a=b;
        b=Temp;
}
```

Total variables= Temp,a,b

Constant =O(1)

Measuring running time

Measuring running time

- Analysis independent of underlying hardware
  - Don’t use actual time
  - Measure in term of “Basic operations”

Input size

Input size

- Running time depends on input size
- Measure time efficiency as function of input size
  - Input size n
  - Running time $t(n)$

Worst-case analysis

Worst-case analysis

- Why do we usually focus on the worst case analysis?
  - Being upper bound, the worst case guarantees that the algorithm will not take any longer.
  - Average case is often roughly as bad as the worst case.

Example 1: Sorting

Example 1: Sorting

- Sorting as array with n elements

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations
  - 10^8 operation per second => 10^{10} seconds

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations
  - 10^8 operation per second => 10^{10} seconds
  - 2778000 hours

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations
  - 10^8 operation per second => 10^{10} seconds
  - 2778000 hours
  - 115700 days

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations
  - 10^8 operation per second => 10^{10} seconds
  - 2778000 hours
  - 115700 days
  - 300 years!!!

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations
  - 10^8 operation per second => 10^{10} seconds
  - 2778000 hours
  - 115700 days
  - 300 years!!!

- Best $n \log n$ algorithm takes only about 3×10^{10} operations

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations
  - 10^8 operation per second => 10^{10} seconds
  - 2778000 hours
  - 115700 days
  - 300 years!!!

- Best $n \log n$ algorithm takes only about 3×10^{10} operations
- About 300 seconds

Example 1: Sorting

- Sorting as array with n elements
  - Basic algorithms : time proportional to n^2
  - Best algorithms : time proportional to $n \log n$
  - Typical CPUs process up to 10^8 operation per second (for approximate calculation)
  - Telephone directory for mobile phone users in India
  - India has about 1 billion = 10^9 phones
  - Basic n^2 algorithm requires 10^{18} operations
  - 10^8 operation per second => 10^{10} seconds
  - 2778000 hours
  - 115700 days
  - 300 years!!!

- Best $n \log n$ algorithm takes only about 3×10^{10} operations
- About 300 seconds
- About 5 minutes

Typical functions

Problem #1

- Problem: print "$Hello\ NIT\ Surat$" for n times

Problem #1

- Problem: print “$Hello\ NIT\ Surat$” for n times
- Algorithm: Print

Problem #1

- Problem: print “$Hello\ NIT\ Surat$” for n times
- Algorithm: Print

- What could be the running time of the solution?

Analyzing Problem #1

- Prepare a table as shown below

Statement	cost	times_executed
1		
2		
3		
4		
5		

Analyzing Problem #1

- Problem: print "$Hello\ NIT\ Surat$" for n times
- Algorithm/ pseudo code: Print (n)

1. $i = 1$
2. While $i \leq n$
3.   Print "$Hello\ NIT\ Surat$"
4. $i = i + 1$
5. Exit

Analyzing Problem #1

- Problem: print "$Hello\ NIT\ Surat$" for n times
- Algorithm/ pseudo code: Print (n)

1. $i = 1$……………………………………………C_1
2. While $i \leq n$…………………………………C_2
3. Print "$Hello\ NIT\ Surat$"……….C_3
4. $i = i + 1$………………………………C_4
5. Exit……………………………………………C_5

Analyzing Problem #1

- Problem: print “$Hello\ NIT\ Surat$” for n times
- Algorithm/ pseudo code: Print (n)

1. $i = 1$………………………………………C_1
2. While $i \leq n$…………………………………C_2
3. Print “$Hello\ NIT\ Surat$”……….C_3
4. $i = i + 1$………………………………C_4
5. Exit……………………………………………C_5

Total steps = Total time = $\boldsymbol{C_1 + (n+1)C_2 + nC_3 + nC_4 + C_5}$

Problem #2

- Consider the code snippet

For $i = 1$ to n

  For $j = 1$ to n

    Print "$DAA\ 2021$"

- What is the cost of execution?

Analyzing Problem #2

- Consider the code snippet

For $i = 1$ to n..C_1

  For $j = 1$ to n....................................C_2

    Print "DAA 2021"..................C_3

Analyzing Problem #2

- Consider the code snippet

For $i = 1$ to n......................................C_1

For $j = 1$ to n..................................C_2

Print "DAA 2021"..................C_3

Total time = $\boldsymbol{C_1(n+1) + C_2 n(n+1) + C_3 n^2}$

Analyzing Problem #2

- Consider the code snippet

For $i = 1$ to n……………………………………C_1

For $j = 1$ to n…………………………………C_2

Print "DAA 2021"………………C_3

Total time = $\boldsymbol{C_1(n+1) + C_2 n(n+1) + C_3 n^2}$

= $\boldsymbol{C_1(n+1) + C_2(n^2+n) + C_3 n^2}$

Problem #3

- Consider the code snippet

```
For i = 1 to n
    For j = 1 to i
        Print "DAA 2021"
```

- What is the cost of execution?

Analyzing Problem #3

- Consider the code snippet

For $i = 1$ to n…………………………………….C_1

    For $j = 1$ to i………………………………C_2

        Print "$DAA\ 2021$"……………….C_3

Analyzing Problem #3

- Consider the code snippet

1 For $i = 1$ to n..................................C_1

2 For $j = 1$ to i................................C_2

3 Print "DAA 2021"................C_3

Total time = $\boldsymbol{C_1(n+1) + C_2\left(\frac{(n+1)(n+2)}{2} - 1\right) + C_3\,\frac{n(n+1)}{2}}$

Analyzing Problem #3

- Consider the code snippet

For $i = 1$ to n…………………………………C_1

  For $j = 1$ to i……………………………C_2

    Print "DAA 2021"………………C_3

Total time = $\boldsymbol{C_1(n+1) + C_2\left(\frac{(n+1)(n+2)}{2} - 1\right) + C_3\,\frac{n(n+1)}{2}}$

$$= \boldsymbol{C_1(n+1) + C_2\left(\frac{n^2+3n}{2}\right) + C_3\left(\frac{n^2+n}{2}\right)}$$

Problem #4

- Consider the code snippet

For $i = 1$ to n

    For $j = i$ to n

        Print "$DAA\ 2021$"

- What is the cost of execution?

Analyzing Problem #4

- Consider the code snippet

For $i = 1$ to n..C_1

    For $j = i$ to n...C_2

        Print "$DAA\ 2021$"...................C_3

Analyzing Problem #4

- Consider the code snippet

For $i = 1$ to n……………………………………C_1

For $j = i$ to n…………………………………C_2

Print "$DAA\ 2021$"………………C_3

Total time = $\boldsymbol{C_1(n+1) + C_2\left(\frac{(n+1)(n+2)}{2} - 1\right) + C_3\,\frac{n(n+1)}{2}}$

Analyzing Problem #4

- Consider the code snippet

For $i = 1$ to n……………………………………C_1

    For $j = i$ to n………………………………C_2

        Print "$DAA\ 2021$"………………C_3

Total time = $\boldsymbol{C_1(n+1) + C_2\left(\frac{(n+1)(n+2)}{2} - 1\right) + C_3\,\frac{n(n+1)}{2}}$

$$= \boldsymbol{C_1(n+1) + C_2\left(\frac{n^2+3n}{2}\right) + C_3\left(\frac{n^2+n}{2}\right)}$$

Problem #6

- Problem: Insertion sort

Analyzing Problem #6

- Algorithm Insertion-Sort $(A[], n)$

For $j = 2$ to n
    $key = A[j]$
    $i = j - 1$
    While $(i > 0)$ and $(A[i] > key)$
    $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$

Analyzing Problem #6

- Algorithm Insertion-Sort $(A[], n)$

For $j = 2$ to n

$key = A[j]$

$i = j - 1$

While $(i > 0)$ and $(A[i] > key)$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

- How do we analyze the time complexity?

Analyzing Problem #6

- Algorithm Insertion-Sort $(A[], n)$

For $j = 2$ to n
    $key = A[j]$
    $i = j - 1$
    While $(i > 0)$ and $(A[i] > key)$
    $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$

- How do we analyze the time complexity?
- We need to analyze **how many times** the while loop is executed?

Analyzing Problem #6

- Algorithm Insertion-Sort $(A[], n)$

For $j = 2$ to n

  $key = A[j]$

  $i = j - 1$

  While $(i > 0)$ and $(A[i] > key)$

  $A[i + 1] = A[i]$

    $i = i - 1$

  $A[i + 1] = key$

- How do we analyze the time complexity?
- We need to analyze **how many times** the while loop is executed?
  - Assume while loop is executed $\boldsymbol{t_j}$ times…

Analyzing Problem #6

- Then the running time is given by the expression

```
For j = 2 to n
        key = A[j]
        i = j − 1
        While (i > 0) and (A[i] > key)
        A[i + 1] = A[i]
                i = i − 1
        A[i + 1] = key
```

Analyzing Problem #6

- Then the running time is given by the expression

$$C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} t_j + C_5 \sum_{j=2}^{n} (t_j - 1) + C_6 \sum_{j=2}^{n} (t_j - 1) + C_7(n-1)$$

For $j = 2$ to n
    $key = A[j]$
    $i = j - 1$
    While $(i > 0)$ and $(A[i] > key)$
    $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Analyzing Problem #6

- When does the **best case** occur?

For $j = 2$ to n
    $key = A[j]$
    $i = j - 1$
    While $(i > 0)$ and $(A[i] > key)$
    $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$

Analyzing Problem #6

- When does the **best case** occur?
  - $t_j = 1$ in every case
  - i.e. when the array is sorted

```
For j = 2 to n
        key = A[j]
        i = j − 1
        While (i > 0) and (A[i] > key)
        A[i + 1] = A[i]
                i = i − 1
        A[i + 1] = key
```

Analyzing Problem #6

- When does the **best case** occur?
  - $t_j = 1$ in every case
  - i.e. when the array is sorted
- Then the best case running time is
- $C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} 1 + C_5 \sum_{j=2}^{n} 0 + C_6 \sum_{j=2}^{n} 0 + C_7(n-1)$

```
For j = 2 to n
        key = A[j]
        i = j − 1
        While (i > 0) and (A[i] > key)
        A[i + 1] = A[i]
                i = i − 1
        A[i + 1] = key
```

Analyzing Problem #6

- When does the **best case** occur?
  - $t_j = 1$ in every case
  - i.e. when the array is sorted
- Then the best case running time is

$$C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} 1 + C_5 \sum_{j=2}^{n} 0 + C_6 \sum_{j=2}^{n} 0 + C_7(n-1)$$

$$= C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1)$$

Analyzing Problem #6

- When does the **worst case** occur?

```
For j = 2 to n
        key = A[j]
        i = j − 1
        While (i > 0) and (A[j] > key)
        A[i + 1] = A[i]
                i = i − 1
        A[i + 1] = key
```

Analyzing Problem #6

- When does the **worst case** occur?
  - At least when the while loop is executed for all the values of i……
  - i.e. when the array is reverse sorted

For $j = 2$ to n

  $key = A[j]$

  $i = j - 1$

  While $(i > 0)$ and $(A[j] > key)$

  $A[i + 1] = A[i]$

    $i = i - 1$

  $A[i + 1] = key$

Analyzing Problem #6

- When does the **worst case** occur?
  - At least when the while loop is executed for all the values of i……
  - i.e. when the array is reverse sorted
  - Thus, $t_j = j$.

For $j = 2$ to n
  $key = A[j]$
  $i = j - 1$
  While $(i > 0)$ and $(A[j] > key)$
  $A[i + 1] = A[i]$
    $i = i - 1$
  $A[i + 1] = key$

Analyzing Problem #6

- When does the **worst case** occur?
  - At least when the while loop is executed for all the values of i……
  - i.e. when the array is reverse sorted
  - Thus, $t_j = j$. Therefore the expression is
  - $C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} j + C_5 \sum_{j=2}^{n}(j-1) + C_6 \sum_{j=2}^{n}(j-1) + C_7(n-1)$

```
For j = 2 to n
        key = A[j]
        i = j − 1
        While (i > 0) and (A[j] > key)
        A[i + 1] = A[i]
                i = i − 1
        A[i + 1] = key
```

Analyzing Problem #6

- When does the **worst case** occur?
  - At least when the while loop is executed for all the values of i......
  - i.e. when the array is reverse sorted
  - Thus, $t_j = j$. Therefore the expression is

$$C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} j + C_5 \sum_{j=2}^{n} (j-1) + C_6 \sum_{j=2}^{n} (j-1) + C_7(n-1)$$

$$= C_1\, n + C_2(n-1) + C_3(n-1) + C_4\left(\frac{n(n+1)}{2} - 1\right) + C_5\left(\frac{n(n-1)}{2}\right) + C_6\left(\frac{n(n-1)}{2}\right)$$
$$+ C_7(n-1)$$

Analyzing Problem #6

- When does the **worst case** occur?
  - At least when the while loop is executed for all the values of i……
  - i.e. when the array is reverse sorted
  - Thus, $t_j = j$. Therefore the expression is

$$C_1 n + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} j + C_5 \sum_{j=2}^{n} (j-1) + C_6 \sum_{j=2}^{n} (j-1) + C_7(n-1)$$

$$= C_1\, n + C_2(n-1) + C_3(n-1) + C_4 \left(\frac{n(n+1)}{2} - 1\right) + C_5 \left(\frac{n(n-1)}{2}\right) + C_6 \left(\frac{n(n-1)}{2}\right)$$
$$+ C_7(n-1)$$

$$= C_1 n + C_2(n-1) + C_3(n-1) + C_4 \left(\frac{n^2+n-2}{2}\right) + C_5 \left(\frac{n^2-n}{2}\right) + C_6 \left(\frac{n^2-n}{2}\right)$$
$$+ C_7\,(n-1)$$

Growth of functions

- Consider
  - An algorithm **A** which for a problem, does $\mathbf{2n}$ **basic** operation & $\mathbf{2C_1n}$ **total** operations, while some other algorithm **B** does $\mathbf{4.5n}$ **basic** operations & $\mathbf{4.5C_2n}$ **total** operations.

Growth of functions

- Consider
  - An algorithm **A** which for a problem, does $\mathbf{2n}$ **basic** operation & $\mathbf{2C_1n}$ **total** operations, while some other algorithm **B** does $\mathbf{4.5n}$ **basic** operations & $\mathbf{4.5C_2n}$ **total** operations.
    - Consider constant of proportionality representing overhead operations.

Growth of functions

- Consider
  - An algorithm **A** which for a problem, does $\mathbf{2n}$ **basic** operation & $\mathbf{2C_1n}$ **total** operations, while some other algorithm **B** does $\mathbf{4.5n}$ **basic** operations & $\mathbf{4.5C_2n}$ **total** operations.
    - Consider constant of proportionality representing overhead operations.

  - Which algorithm of the two do you think is better?

Growth of functions …

n	$2n$	$4.5n$
5	10	22
10	20	45
100	200	450
1000	2000	4500
10000	20000	45000
100000	$2.0 * 10^5$	$4.5 * 10^5$
$1000000 = 10^6$	$2.0 * 10^6$	$4.5 * 10^6$

Growth of functions

- Consider
  - Another such example with **algo1** taking $\boldsymbol{\frac{n^3}{2}}$ multiplicative steps while **algo2** taking $\boldsymbol{5n^2}$ steps.

Growth of functions

- Consider
  - Another such example with **algo1** taking $\frac{n^3}{2}$ multiplicative steps while **algo2** taking $5n^2$ steps.
    - Consider constant of proportionality representing overhead operations.
  - Which algorithm of the two do you think is better?

Growth of functions …

n	$2n$	$4.5n$	$n^3/2$	$5n^2$
5	10	22	45	125
10	20	45	500	500
100	200	450	$5 * 10^5$	$5 * 10^4$
1000	2000	4500	$5 * 10^8$	$5 * 10^6$
10000	20000	45000	$5 * 10^{11}$	$5 * 10^8$
100000	$2.0 * 10^5$	$4.5 * 10^5$	$5 * 10^{14}$	$5 * 10^{10}$
$1000000 = 10^6$	$2.0 * 10^6$	$4.5 * 10^6$	$5 * 10^{17}$	$5 * 10^{12}$

Growth of functions …

- A relook at costs of insertion sort with $\boldsymbol{C_i's = 1}$

Growth of functions …

- A relook at costs of insertion sort with $\boldsymbol{C_i's = 1}$
- Best case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1)$$

Growth of functions …

- A relook at costs of insertion sort with $\boldsymbol{C_i's = 1}$
- Best case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1)$$
$$= 5n - 4$$

Growth of functions …

- A relook at costs of insertion sort with $\boldsymbol{C_i's = 1}$
- Best case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1)$$
$$= 5n - 4$$

- Worst Case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4\left(\frac{n^2+n-2}{2}\right) + C_5\left(\frac{n^2-n}{2}\right)$$
$$+C_6\left(\frac{n^2-n}{2}\right) + C_7\,(n-1)$$

Growth of functions …

- A relook at costs of insertion sort with $\boldsymbol{C_i' s = 1}$
- Best case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1)$$
$$= 5n - 4$$

- Worst Case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4\left(\frac{n^2+n-2}{2}\right) + C_5\left(\frac{n^2-n}{2}\right)$$
$$+C_6\left(\frac{n^2-n}{2}\right) + C_7(n-1)$$
$$= \frac{1}{2}(3n^2 + 7n - 8)$$

Growth of functions …

- A relook at costs of insertion sort with $\boldsymbol{C_i's = 1}$
- Best case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1)$$
$$= 5n - 4$$

- Worst Case

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4\left(\frac{n^2+n-2}{2}\right) + C_5\left(\frac{n^2-n}{2}\right)$$
$$+C_6\left(\frac{n^2-n}{2}\right) + C_7(n-1)$$
$$= \frac{1}{2}(3n^2 + 7n - 8)$$

- Which term dominates the overall result in the above expression, especially at large values of n ?

Growth of functions …

- Which term dominates the overall result in the above expression, especially at large values of n ?

$\boldsymbol{n}$	$\boldsymbol{T(n) = 3n^2 + 7n - 8}$	$\boldsymbol{T(n) = 3n^2}$
10	362	300
100	30692	30000
1000	$3.006992 * 10^6$	$3.00 * 10^6$
10000	$3.0000699992 * 10^{10}$	$3.00 * 10^{10}$

Growth of functions …

- Does constant of proportionality matter when n gets very large?

Growth of functions …

- Does constant of proportionality matter when n gets very large?
- Then, what is asymptotic growth rate, asymptotic order or order of functions?

Growth of functions …

- Does constant of proportionality matter when n gets very large?
- Then, what is asymptotic growth rate, asymptotic order or order of functions?
- Is it reasonable to ignore smaller values and constants?

Growth of functions …

- Hence, we shall now also drop the all the terms

Growth of functions …

- Hence, we shall now also drop the all the terms
  - **Except the highest degree of the polynomial** for the running time of the algorithm

Growth of functions …

- Hence, we shall now also drop the all the terms
  - **Except the highest degree of the polynomial** for the running time of the algorithm
- Example

Growth of functions …

- Hence, we shall now also drop the all the terms
  - **Except the highest degree of the polynomial** for the running time of the algorithm
- Example
- Insertion sort Best case complexity …
  $\boldsymbol{T(n) = 5n - 4}$
  - So, we will say that **complexity is of the order of $\boldsymbol{n}$**

Growth of functions …

- Hence, we shall now also drop the all the terms
  - **Except the highest degree of the polynomial** for the running time of the algorithm
- Example
- Insertion sort Best case complexity …

  $\boldsymbol{T(n) = 5n - 4}$
  - So, we will say that **complexity is of the order of $\boldsymbol{n}$**
- Insertion sort Best case complexity …

  $$\boldsymbol{T(n) = \frac{1}{2}(3n^2 + 7n - 8)}$$
  - So, we will say that **complexity is of the order of $\boldsymbol{n^2}$**

Order of growth and abstractions

- So, now we have assumed/abstracted at three different levels viz.

Order of growth and abstractions

- So, now we have assumed/abstracted at three different levels viz.
  - Level 1 – ignored the actual cost of execution of each statement.

Order of growth and abstractions

- So, now we have assumed/abstracted at three different levels viz.
  - Level 1 – ignored the actual cost of execution of each statement.
  - Level 2 – ignored even the abstract cost (C_i) of each statement.

Order of growth and abstractions

- So, now we have assumed/abstracted at three different levels viz.
  - Level 1 – ignored the actual cost of execution of each statement.
  - Level 2 – ignored even the abstract cost (C_i) of each statement.
  - Level 3 – ignore all the terms except for the one with the highest degree in the expression of time complexity

Order of growth and abstractions

- So, now we have assumed/abstracted at three different levels viz.
  - Level 1 – ignored the actual cost of execution of each statement.
  - Level 2 – ignored even the abstract cost (C_i) of each statement.
  - Level 3 – ignore all the terms except for the one with the highest degree in the expression of time complexity

- Such analysis is based on the asymptotic growth rate,

Order of growth and abstractions

- So, now we have assumed/abstracted at three different levels viz.
  - Level 1 – ignored the actual cost of execution of each statement.
  - Level 2 – ignored even the abstract cost (C_i) of each statement.
  - Level 3 – ignore all the terms except for the one with the highest degree in the expression of time complexity

- Such analysis is based on the asymptotic growth rate,
  - Asymptotic order or order or functions and called asymptotic analysis

Typical functions

- We are interested in order of magnitude
- $t(n)$ may proportional to $\log n, \dots, n^2, n^3, \dots, 2^n$
- Logarithmic, polynomial, exponential …

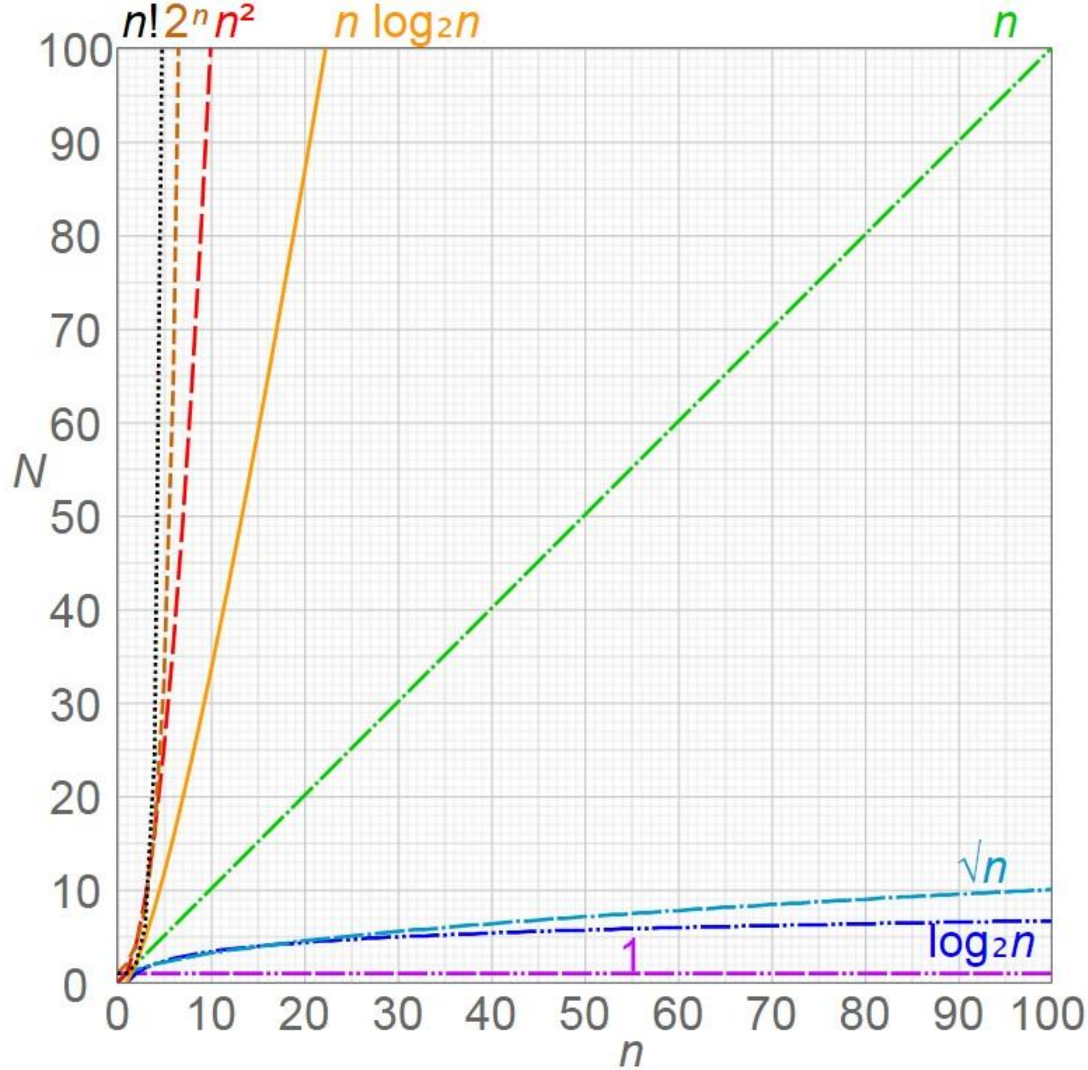
Basic Asymptotic Efficiency classes

1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	$n \log n$
n^2	Quadratic
n^3	Cubic
2^n	Exponential
$n!$	Factorial

Typical functions $t(n)$

Input	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
$\mathbf{10^4}$	13	10^4	10^5	10^8	10^{12}		
$\mathbf{10^5}$	17	10^5	10^6	10^{10}			
$\mathbf{10^6}$	20	10^6	10^7				
$\mathbf{10^7}$	23	10^7	10^8				
$\mathbf{10^8}$	27	10^8	10^9				
$\mathbf{10^9}$	30	10^9	10^{10}				
$\mathbf{10^{10}}$	33	10^{10}					

Typical functions $t(n)$



An interesting “seconds” conversion

10^2	1.7 min
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries

Asymptotic Notations

- In this approach, the running time of an algorithm is describes as Asymptotic Notations.

- Computing the running time of **algorithm's operations in mathematical units of computation and defining the mathematical formula of its run-time performance** is referred to as Asymptotic Analysis.

- An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

- Asymptotic analysis accomplishes the study of change in performance of the algorithm with the change in the order of the input size.

Asymptotic Notations

- Asymptotic notations are mathematical notations used to represent the time complexity of algorithms for Asymptotic analysis.

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

  1. Ο Notation
  2. Ω Notation
  3. θ Notation

- Asymptotic Notations are used,
  1. To characterize the complexity of an algorithm.
  2. To compare the performance of two or more algorithms solving the same problem.

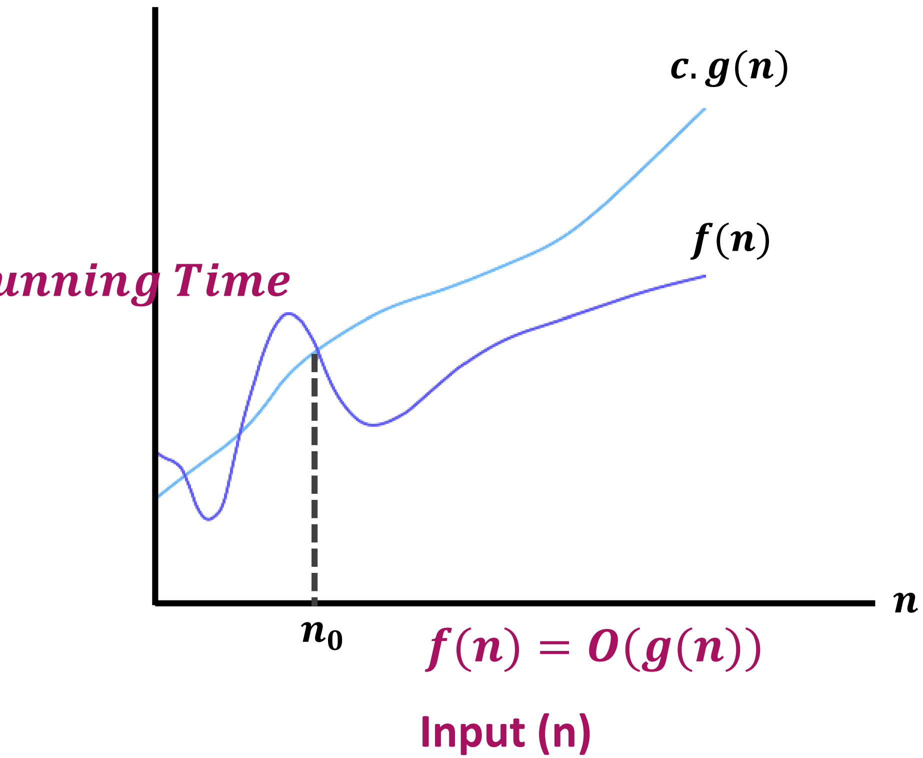
1. O-Notation (Big O notation) (Upper Bound)

- The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time.

- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions,

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n_0 \leq n\}$

Big(O) Notation



- $g(n)$ is an asymptotically **upper bound** for $f(n)$.
- $f(n) = O(g(n))$ implies:
  $$f(n) \text{“} \leq \text{”} c.g(n)$$

2. Ω-Notation (Omega notation) (Lower Bound)

- Big Omega notation (Ω) is used to define the lower bound of any algorithm.

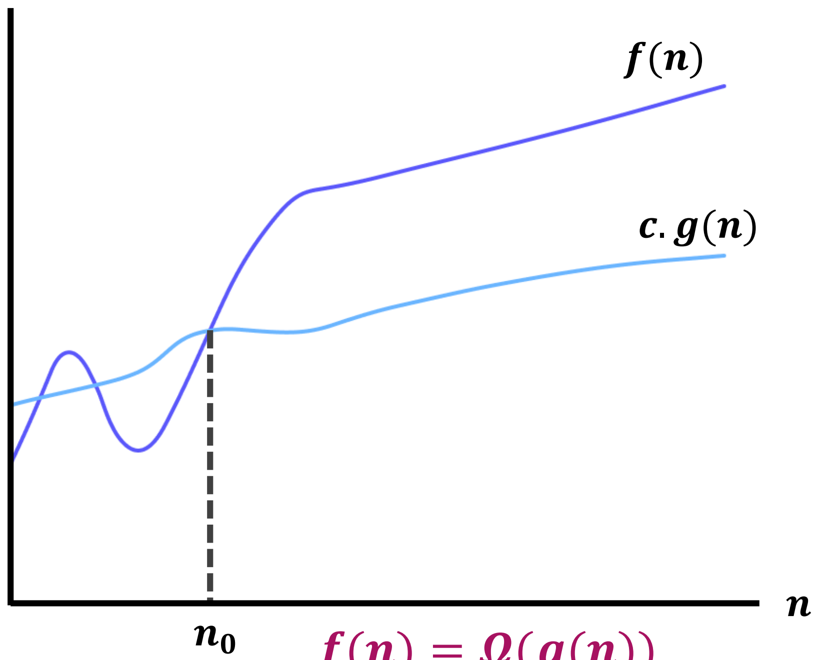
- This always indicates the minimum time required for any algorithm for all input values.

- When a time complexity for any algorithm is represented in the form of big-Ω, it means that the algorithm will take at least this much time to complete it's execution. It can definitely take more time than this too.

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions,

$\Omega(\mathrm{g(n)}) = \{\mathrm{f(n)}$:there exist positive constants c and n_0 such that $0 \leq \mathrm{cg(n)} \leq \mathrm{f(n)}$ for all $n_0 \leq \mathrm{n}\}$

Big(Ω) Notation



$$\boldsymbol{f(n) = \Omega(g(n))}$$

- $g(n)$ is an asymptotically **lower bound** for $f(n)$.
- $f(n) = \Omega(g(n))$ implies:
  $$\boldsymbol{f(n)\text{“} \geq \text{”}\, c.\,g(n)}$$

3. θ-Notation (Theta notation) (Same order)

- The notation $\theta(n)$ is the formal way to enclose both the lower bound and the upper bound of an algorithm's running time.

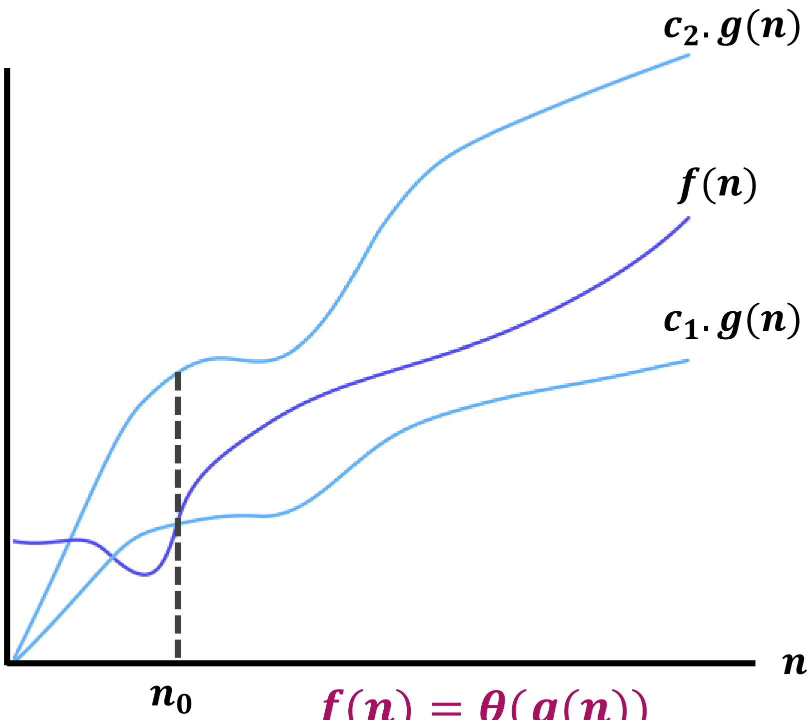
- The time complexity represented by the Big-θ notation is the range within which the actual running time of the algorithm will be.

- So, it defines the exact Asymptotic behavior of an algorithm.

- For a given function $g(n)$, we denote by $\theta(g(n))$ the set of functions,

$\theta(g(n))$ = {$f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n_0 \le n$}

θ-Notation



$f(n) = \theta(g(n))$

- $\theta(g(n))$ is a set, we can write $f(n) \epsilon \theta(g(n))$ to indicate that $f(n)$ is a member of $\theta(g(n))$.
- $g(n)$ is an asymptotically tight bound for $f(n)$.
- $f(n) = \theta(g(n))$ implies:

$$f(n) \text{“} = \text{''} c.g(n)$$

Asymptotic Notations

Asymptotic Notations

1. O-Notation (Big O notation) (Upper Bound)

$O(g(n))$ = $\{f(n)$: there exist positive constants c and n_0 such that $\mathbf{0 \le f(n) \le g(n)}$ for all $n_0 \le n\}$

$\mathbf{f(n) = O(g(n))}$

2. Ω-Notation (Omega notation) (Lower Bound)

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $\mathbf{0 \le cg(n) \le f(n)}$ for all $n_0 \le n\}$

$\mathbf{f(n) = \Omega(g(n))}$

3. θ-Notation (Theta notation) (Same order)

$\theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and $n_0$such that $\mathbf{0 \le c_1 g(n) \le f(n) \le c_2 g(n)}$ for all $n_0 \le n\}$

$\mathbf{f(n) = \theta(g(n))}$

Asymptotic Notations – Examples

- Example 1:

$f(n) = n^2$ and $g(n) = n$

Algo. 1 running time

Algo. 2 running time

$f(n) \geq g(n) \Longrightarrow f(n) = \Omega(g(n))$

n	$f(n) = n^2$	$g(n) = n$
1	1	1
2	4	2
3	9	3
4	16	4
5	25	5

- Example 2:

$f(n) = n$ and $g(n) = n^2$

Algo. 1 running time

Algo. 2 running time

$f(n) \leq g(n) \Longrightarrow f(n) = O(g(n))$

n	$f(n) = n$	$g(n) = n^2$
1	1	1
2	2	4
3	3	9
4	4	16
5	5	25

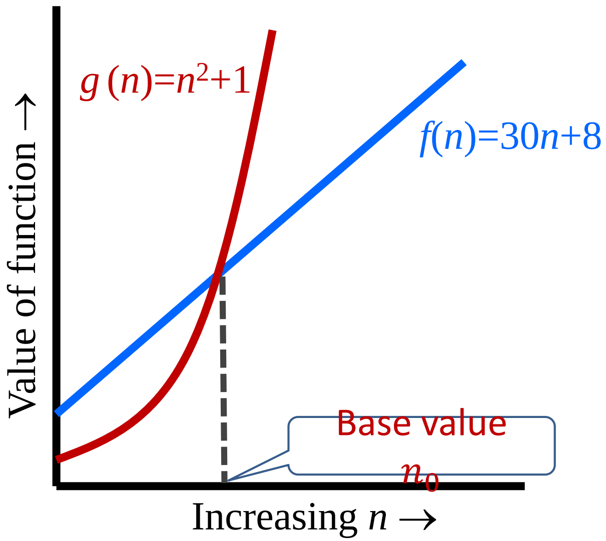
Asymptotic Notations – Examples

- Example 3: $f(n) = n^2$ and $g(n) = 2^n$

$f(n) \leq g(n) \Longrightarrow f(n) = O(g(n))$

n	$f(n) = n^2$	$g(n) = 2^n$		
1	1	2		
2	4	4		
3	9	8		
4	16	16		Here for $n \geq 4$, $f(n) \leq g(n)$ so, $n_0 = 4$
5	25	32		
6	36	64		
7	49	128		

Asymptotic Notations – Examples



- Example 4:
  $\mathbf{f(n) = 30n + 8}$ is in the order of n, or $\mathrm{O(n)}$
  $\mathbf{g(n)} = \boldsymbol{n^2} + \mathbf{1}$ is order n^2, or $\mathrm{O(n^2)}$

  $$\boxed{\boldsymbol{f(n) = O(g(n))}}$$

- In general, any $O(n^2)$ function is faster-growing than any $O(n)$ function.

Asymptotic notations: The Big-O notation

Prove that f(n) =2n+3 is O(n)

$2n+3 \leq 7n$ **[Upgrade to higher order term]**

n	L.H.S.	R.H.S.
1	5	7
2	7	14
3	9	21

f(n)=O(n)   for n>=1 && c=7

Asymptotic notations: The Big-O notation

Prove that $f(n) = 2n+5$ is $O(n)$

$2n+5 \leq 3n$ **[Upgrade to higher order term]**

n	L.H.S.	R.H.S.
1	7	3
2	9	6
3	11	9
4	13	12
5	15	15
6	17	18

$f(n)=O(n)$ for $n>=5$ && $c=3$

- **Prove that $f(n) = 2n+5$ is $O(n^2)$**

Asymptotic notations: The Big-O notation

$f(n) = 2n^2+3n+4$

$2n^2+3n+4 \leq 2n^2+3n^2+4n^2$

$\leq 9n^2$

$f(n) = 2n^2+3n+4$

$c=9$

$g(n)=n^2$

$f(n)=O(n^2)$

Asymptotic notations: The Big- Ω notation

Asymptotic notations: The Big- Ω notation

f(n) =2n+3

$2n+3 \geq 1 * n \quad n \geq 1$

f(n) =2n+3

c=1

g(n)=n

f(n)= Ω(n)

Asymptotic notations: The Big- Ω notation

$f(n) = 2n^2+3n+4$

$2n^2+3n+4n \geq 1 * n^2$

$f(n) = 2n^2+3n+4$

$c=1$

$g(n)=n^2$

$f(n)= \Omega(n^2)$

Asymptotic notations: The Big- θ notation

Asymptotic notations: The Big-θ notation

f(n) =2n+3

f(n)=O(n)

f(n)= Ω(n)

f(n)= θ(n)

Asymptotic notations: The Big-θ notation

$f(n) = 2n^2+3n+4$

$f(n)=O(n^2)$

$f(n)= \Omega(n^2)$

$f(n)= \theta(n^2)$

The Big-theta notation

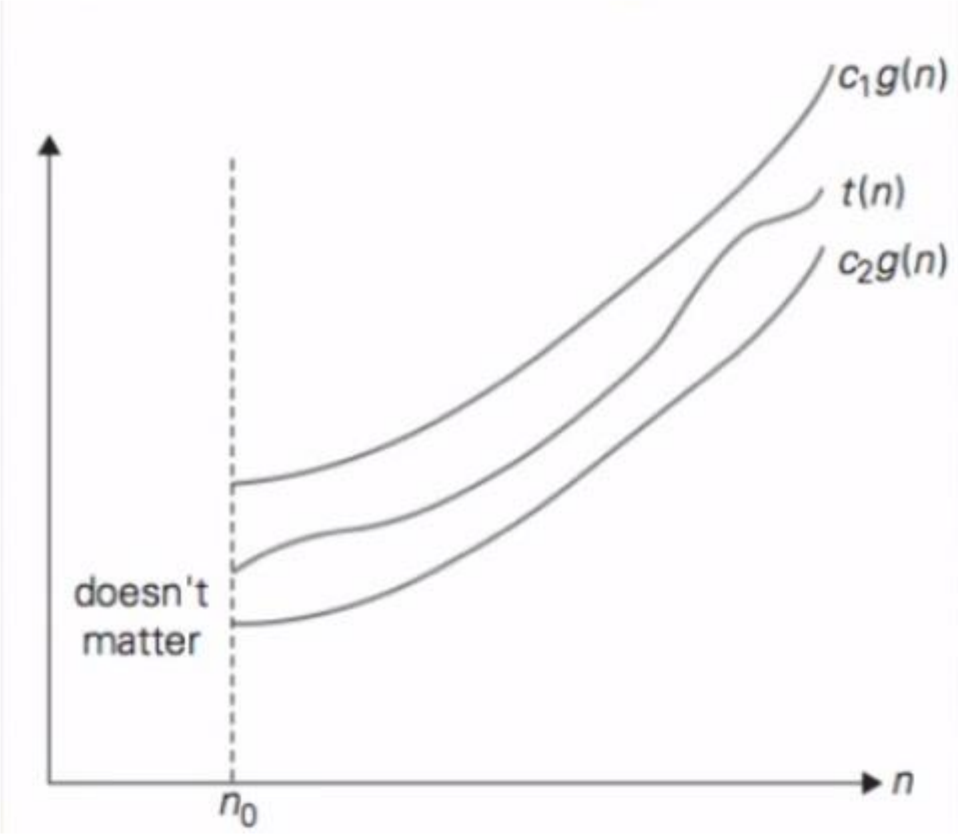


Figure: Big-theta notation: $t(n) \in \theta(g(n))$

- Prove that 10 log n + 4 = θ(log n)

Calculating complexity

- Iterative programs
- Recursive programs

Example 1

- Problem: Maximum value in an array

Example 1

- Problem: Maximum value in an array
- Solution:

Function $MaxElement(A, n)$

$maxval = A[0]$
    For $i = 1$ to $n - 1$
        If $A[i] > maxval$
            $maxval = A[i]$
Return $maxval$

Example 1

- Problem: Maximum value in an array
- Solution:

Function $MaxElement(A, n)$

$maxval = A[0]$

For $i = 1$ to $n - 1$ $\quad ----- -(n-1)\ steps$

    If $A[i] > maxval$

        $maxval = A[i]$

Return $maxval$

Example 2

- Problem: Check if all element in an array are distinct

Example 2

- Problem: Check if all element in an array are distinct
- Solution:

Function $NoDuplicates(A, n)$

For $i = 1$ to n

  For $j = i + 1$ to n

    If $A[i] == A[j]$

      Return $False$

Return $True$

Example 3

- Problem: Matrix multiplication

Example 3

- Problem: Matrix multiplication
- Solution:

Function $MatrixMultiply(A, B)$

1. For $i = 1$ to n
2. For $j = 1$ to n
3. $C[i][j] = 0$
4. For $k = 1$ to n
5. $C[i][j] = c[i][j] + A[i][k] \times B[k][j]$
6. Return C

Properties of Asymptotic Notations:

1) General Properties:

If f(n) is O(g(n)) then a*f(n) is also O(g(n)) ; where a is a constant.

Example:

$f(n) = 2n^2+5$ is $O(n^2)$
then $7*f(n) = 7(2n^2+5)$
$= 14n^2+35$ is also $O(n^2)$

Properties of Asymptotic Notations:

Similarly this property satisfies for both Θ and Ω notation. We can say,

If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)); where a is a constant.

If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)); where a is a constant.

Properties of Asymptotic Notations:

Reflexive Properties:

- If f(n) is given then f(n) is O(f(n)).
  Example: $f(n) = n^2$; $O(n^2)$ i.e O(f(n))
- **Function is Upper Bound for itself.**
- Similarly, this property satisfies both Θ and Ω notation. We can say
  If f(n) is given then f(n) is Θ(f(n)).
  If f(n) is given then f(n) is Ω (f(n)).

Properties of Asymptotic Notations:

- **Transitive Properties :**
- If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

Example: if f(n) = n , g(n) = n^2 and h(n)=n^3
n is O(n^2) and n^2 is O(n^3) then n is O(n^3)

- Similarly this property satisfies for both Θ and Ω notation. We can say
  If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
  If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

Properties of Asymptotic Notations:

Symmetric Properties :

- If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .
  Example: f(n) = n^2 and g(n) = n^2 then f(n) = Θ(n^2) and g(n) = Θ(n^2)

- This property only satisfies for Θ notation.

Properties of Asymptotic Notations:

- **Transpose Symmetric Properties :**
- If f(n) is O(g(n)) then g(n) is Ω (f(n)).
  Example: f(n) = n , g(n) = n^2 then n is O(n^2) and n^2 is Ω (n)
- This property only satisfies for O and Ω notations.

Properties of Asymptotic Notations:

- **Some More Properties :**
- 1. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$

- 2. If $f(n) = O(g(n))$ and $d(n)=O(e(n))$
  then $f(n) + d(n)$ **$= O(\max(g(n), e(n)))$**