Module 1: Introduction to Algorithms

Harvinod N
Dept. of Computer Science and Engineering
VCET Puttur
Course Outcome

• At the end of the course, you will be able to;
  1. Estimate the computational complexity of different algorithms along with its representation notations.
  2. Design and analyze problem solving using divide and conquer strategy
  3. Apply greedy method to solve problems.
  4. Apply dynamic programming to solve problems using the solutions of similar subproblems.
  5. Design and apply backtracking technique for problem solving
Algorithm

Input \rightarrow \text{Set of rules to obtain the expected output from the given input} \rightarrow \text{Output}

Algorithm
al-Khorezmi

- Muhammad ibn Musa al-Khwarizmi
- The Father of Algebra
- Persian Mathematician
How to prepare tea?

**How to Make a Tea**

**INGREDIENTS:**
- 1 kettle
- 1 tea bag
- 1 cup full of water
- As much sugar (or honey) as you want

**BOIL WATER**

**POUR THE WATER IN THE CUP AND ADD THE TEA BAG**

**ADD SUGAR OR HONEY**

**WAIT FOR FIVE MINUTES**

**ENJOY YOUR TEA!**
How to prepare Maggi noodles?

1. Take one and a half cup of Water in a pan.
2. Heat the pan on medium flame.
3. When the Water comes to boil, add the Maggi to the pan.
4. Cover it with a lid for a minute.
5. After a minute, uncover the lid and add the tastemaker to the pan.
6. Mix it well, Without breaking the Noodles.
7. Just when all of your Water is boiled, switch off the flame.
8. Enjoy the hot Maggi.
How Kingfisher catches a fish efficiently?
Shortest Distance between two cities
Text Book -1
Text Book -2
Course Website:
www.techjourney.in
Module 1 – Outline

Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. Fundamental Data Structures
Module 1 – Outline
Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. Fundamental Data Structures
What is an Algorithm?

• An algorithm is a finite sequence of unambiguous instructions to solve a particular problem.

• In addition, all algorithms must satisfy the following criteria:
  
  – *Input*: Zero or more quantities are externally supplied.
  
  – *Output*: At least one quantity is produced.
  
  – *Definiteness*: Each instruction is clear and unambiguous.
  
  – *Finiteness*: algorithm terminates after a finite number of steps.
  
  – *Correctness*
  
  – *Effectiveness*
Algorithm design and analysis process

1. Understand the problem
2. Decide on: 1) computational means  
   2) Exact vs. approximate solving  
   3) Algorithm design technique
3. Design the algorithm
4. Prove the correctness
5. Analyze the algorithm
6. Code the algorithm
Algorithm specification

- An algorithm can be specified in
  1. Simple English
  2. Graphical representation like flow chart
  3. Programming language like C++ / Java
  4. Combination of above methods.
void SelectionSort(Type a[], int n)
// Sort the array a[1:n] into nondecreasing order.
{
    for (int i=1; i<=n; i++) {
        int j = i;
        for (int k=i+1; k<=n; k++)
            if (a[k]<a[j]) j=k;
        Type t = a[i]; a[i] = a[j]; a[j] = t;
    }
}

for (i=1; i<=n; i++) {
    examine a[i] to a[n] and suppose the smallest element is at a[j];
    interchange a[i] and a[j];
}
Recursive Algorithm

- An algorithm is said to be **recursive** if the same algorithm is invoked in the body (direct recursive).
- Algorithm $A$ is said to be **indirect recursive** if it calls another algorithm which in turn calls $A$.
- Example 1: Factorial computation   $n! = n \times (n-1)!$
- Example 2: Binomial coefficient computation

\[
\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}
\]

- Example 3: Tower of Hanoi problem
- Example 4: Permutation Generator
Module 1 – Outline

Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. Fundamental Data Structures
Performance Analysis

• Space complexity
  – *Space Complexity* of an algorithm is total space taken by the algorithm with respect to the input size.
  – Space complexity includes both Auxiliary space and space used by input.

• Time complexity
Time Complexity

• Execution time or run-time of the program is refereed as its time complexity
• This is the sum of the time taken to execute all instructions in the program.

• But, We count only the number of steps in the program. Why?
• How to count?
  – Two ways
Method-1

- Introduce a count variable
- Increment count for every operation

```c
float Sum(float a[], int n)
{
    float s = 0.0;
    count++; // count is global
    for (int i=1; i<=n; i++) {
        count++; // For 'for'
        s += a[i]; count++; // For assignment
    }
    count++; // For last time of 'for'
    count++; // For the return
    return s;
}
```
Method-2

- Count the **steps per execution** for every line of code
- Note the frequency of execution of every line and find the total steps.

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>frequency</th>
<th>total steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>float Sum(float a[], int n)</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>{ float s = 0.0;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>for (int i=1; i&lt;=n; i++)</td>
<td>1</td>
<td>n + 1</td>
<td>n + 1</td>
</tr>
<tr>
<td>s += a[i];</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>return s;</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2n + 3</td>
</tr>
</tbody>
</table>
Method-2

```c
void Add(Type a[] [SIZE], Type b[] [SIZE],
         Type c[] [SIZE], int m, int n)
{
    for (int i=1; i<=m; i++)
        for (int j=1; j<=n; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>freq</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>void Add(Type a[], ...)</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>{ for (int i=1; i&lt;=m; i++)</td>
<td>1</td>
<td>m+1</td>
<td>m+1</td>
</tr>
<tr>
<td>for (int j=1; j&lt;=n; j++)</td>
<td>1</td>
<td>m(n+1)</td>
<td>mn+m</td>
</tr>
<tr>
<td>c[i][j] = a[i][j]</td>
<td>1</td>
<td>mn</td>
<td>mn</td>
</tr>
<tr>
<td>+ b[i][j];</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2mn+2m+1</td>
</tr>
</tbody>
</table>
Trade-off

• One has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on application.
Analysis Framework

• Space complexity
• Time complexity

• Measuring an Input’s Size
  – run longer on larger inputs
• Units for Measuring Running time
  – Basic operation
• Order of Growth
Orders of Growth

- **Order of growth** of an algorithm is a way of stating how execution time or memory occupied by it changes with the input size.
- Why emphasis on large input sizes?
- Because for large values of n, it is the function's order of growth that counts.

<table>
<thead>
<tr>
<th>n</th>
<th>$\log_2 n$</th>
<th>n</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10</td>
<td>3.3·10^1</td>
<td>10^2</td>
<td>10^3</td>
<td>10^3</td>
<td>3.6·10^6</td>
</tr>
<tr>
<td>$10^2$</td>
<td>6.6</td>
<td>$10^2$</td>
<td>6.6·10^2</td>
<td>10^4</td>
<td>10^6</td>
<td>1.3·10^30</td>
<td>9.3·10^{157}</td>
</tr>
<tr>
<td>$10^3$</td>
<td>10</td>
<td>$10^3$</td>
<td>1.0·10^4</td>
<td>10^6</td>
<td>10^9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^4$</td>
<td>13</td>
<td>$10^4$</td>
<td>1.3·10^5</td>
<td>10^8</td>
<td>10^{12}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td>17</td>
<td>$10^5$</td>
<td>1.7·10^6</td>
<td>10^{10}</td>
<td>10^{15}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^6$</td>
<td>20</td>
<td>$10^6$</td>
<td>2.0·10^7</td>
<td>10^{12}</td>
<td>10^{18}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast</td>
<td>Slow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>log $n$</td>
<td>log $n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>$n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>$n^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n^3$</td>
<td>$n^3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>$2^n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n!$</td>
<td>$n!$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### High time efficiency

### Low time efficiency

The graph on the right illustrates the comparison of different time complexities, with $O(1)$ being the fastest and $O(n!)$ being the slowest.
Analysis Framework

- Worst-Case
- Best-Case
- Average-Case Efficiencies
Worst Case

• **Definition:** The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size \( n \), for which the algorithm runs the longest among all possible inputs of that size.

\[
C_{\text{worst}}(n) = n.
\]
Best Case

- **Definition**: The best-case efficiency of an algorithm is its efficiency for the best-case input of size $n$, for which the algorithm runs the fastest among all possible inputs of that size.

\[
C_{\text{best}}(n) = 1.
\]

```
ALGORITHM  SequentialSearch(A[0..n-1], K)

// Searches for a given value in a given array by sequential search
// Input: An array A[0..n-1] and a search key K
// Output: The index of the first element in A that matches K or -1 if there are no matching elements

i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return -1
```

Average Case

- **Definition:** the **average-case complexity** of an algorithm is the amount of time used by the algorithm, **averaged over all possible inputs**.

**Algorithm**

```plaintext
ALGORITHM SequentialSearch(A[0..n − 1], K)
    // Searches for a given value in a given array by sequential search
    // Input: An array A[0..n − 1] and a search key K
    // Output: The index of the first element in A that matches K
    //         or −1 if there are no matching elements
    i ← 0
    while i < n and A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```

**Example Calculation**

\[
C_{\text{avg}}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1 - p)
\]

\[
= \frac{p}{n} \left[1 + 2 + \cdots + i + \cdots + n\right] + n(1 - p)
\]

\[
= \frac{p}{n} \frac{n(n + 1)}{2} + n(1 - p) = \frac{p(n + 1)}{2} + n(1 - p).
\]
Summary of analysis framework

• Both time and space efficiencies are measured as functions of the algorithm's input size.
  – Time efficiency - basic operation
  – Space efficiency - extra memory units

• The efficiencies of some algorithms may differ significantly for inputs of the same size.
  – For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.

• The primary interest lies in the order of growth of the algorithm's running time (or extra memory units consumed) as its input size goes to infinity.
Module 1 – Outline

Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. Fundamental Data Structures
Asymptotic Notations

• To compare orders of growth, computer scientists use three notations:
  – \( O(\text{big oh}) \),
  – \( \Omega(\text{big omega}) \),
  – \( \Theta(\text{big theta}) \) and
  – \( o(\text{little oh}) \)
Big-Oh notation

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large $n$, i.e., if there exist some positive constant $c$ and some nonnegative integer $n_0$ such that

$$t(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$
Big-oh notation: \( t(n) \in O(g(n)) \).
Strategies for Big-O

- Sometimes the easiest way to prove that $f(n) = O(g(n))$ is to take $c$ to be the sum of the positive coefficients of $f(n)$.

- **Example:** To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \geq n_0 = 1$,

  $$5n^2 + 3n + 20 \leq 5n^2 + 3n^2 + 20n^2 = 28n^2,$$

  thus $5n^2 + 3n + 20 = O(n^2)$.

- We can usually ignore the negative coefficients. Why?
Problems on Big O

• Prove $n^2 + n = O(n^3)$

• Prove $100n + 5 = O(n^2)$

• Prove $2n^2 + 5n - 3 = O(n^2)$
Omega notation

A function \( t(n) \) is said to be in \( \Omega(g(n)) \), denoted \( t(n) \in \Omega(g(n)) \), if \( t(n) \) is bounded below by some positive constant multiple of \( g(n) \) for all large \( n \), i.e., if there exist some positive constant \( c \) and some nonnegative integer \( n_0 \) such that
\[
t(n) \geq c \cdot g(n)
\]
for all \( n \geq n_0 \).
Big Omega \( t(n) = \Omega(g(n)) \)
Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

\[ n^3 \geq n^2 \quad \text{for all } n \geq 0, \]

i.e., we can select $c = 1$ and $n_0 = 0$. 
Example: $n^3 + 4n^2 = \Omega(n^2)$

- Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$
- It is not too hard to see that if $n \geq 0$,
  \[ n^3 \leq n^3 + 4n^2 \]
- We have already seen that if $n \geq 1$,
  \[ n^2 \leq n^3 \]
- Thus when $n \geq 1$,
  \[ n^2 \leq n^3 \leq n^3 + 4n^2 \]
- Therefore,
  \[ 1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1 \]
- Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$
  (by definition of Big-$\Omega$, with $n_0 = 1$, and $c = 1$.)
Theta Notation

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large $n$, i.e., if there exist some positive constants $c_1$ and $c_2$ and some nonnegative integer $n_0$ such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n)$$

for all $n \geq n_0$. 
Big-theta notation: $t(n) \in \Theta(g(n))$.
Graphical representation

• Which one best to represent order of growth
Strategies for $\Omega$ and $\Theta$

- Proving that a $f(n) = \Omega(g(n))$ often requires more thought.
  - Quite often, we have to pick $c < 1$.
  - A good strategy is to pick a value of $c$ which you think will work, and determine which value of $n_0$ is needed.
  - Being able to do a little algebra helps.
  - We can sometimes simplify by ignoring terms of $f(n)$ with the positive coefficients.
- The following theorem shows us that proving $f(n) = \Theta(g(n))$ is nothing new:
  - Theorem: $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - Thus, we just apply the previous two strategies.
**Example:** \( n^2 + 5n + 7 = \Theta(n^2) \)

**Proof:**

- When \( n \geq 1 \),
  \[
  n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2
  \]

- When \( n \geq 0 \),
  \[
  n^2 \leq n^2 + 5n + 7
  \]

- Thus, when \( n \geq 1 \)
  \[
  1n^2 \leq n^2 + 5n + 7 \leq 13n^2
  \]

Thus, we have shown that \( n^2 + 5n + 7 = \Theta(n^2) \) (by definition of Big-\( \Theta \), with \( n_0 = 1 \), \( c_1 = 1 \), and \( c_2 = 13 \).)
Show that $\frac{1}{2}n^2 + 3n = \Theta(n^2)$

Proof:

- Notice that if $n \geq 1$,
  
  $\frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n^2 = \frac{7}{2}n^2$

- Thus,
  
  $\frac{1}{2}n^2 + 3n = O(n^2)$

- Also, when $n \geq 0$,
  
  $\frac{1}{2}n^2 \leq \frac{1}{2}n^2 + 3n$

- So
  
  $\frac{1}{2}n^2 + 3n = \Omega(n^2)$

- Since $\frac{1}{2}n^2 + 3n = O(n^2)$ and $\frac{1}{2}n^2 + 3n = \Omega(n^2)$,

  $\frac{1}{2}n^2 + 3n = \Theta(n^2)$
Problem-3

For example, let us prove that $\frac{1}{2} n(n - 1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2} n(n - 1) = \frac{1}{2} n^2 - \frac{1}{2} n \leq \frac{1}{2} n^2 \quad \text{for all } n \geq 0.$$ 

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2} n(n - 1) = \frac{1}{2} n^2 - \frac{1}{2} n \geq \frac{1}{2} n^2 - \frac{1}{2} n \cdot \frac{1}{2} n \quad \text{(for all } n \geq 2) = \frac{1}{4} n^2.$$ 

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$. 

Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Proof:

- We need to find positive constants $c_1$, $c_2$, and $n_0$ such that
  
  $$0 \leq c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2 \text{ for all } n \geq n_0$$

- Dividing by $n^2$, we get
  
  $$0 \leq c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$
• $c_1 \leq \frac{1}{2} - \frac{3}{n}$ holds for $n \geq 10$ and $c_1 = 1/5$

• $\frac{1}{2} - \frac{3}{n} \leq c_2$ holds for $n \geq 10$ and $c_2 = 1$.

• Thus, if $c_1 = 1/5$, $c_2 = 1$, and $n_0 = 10$, then for all $n \geq n_0$,

\[ 0 \leq c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2 \text{ for all } n \geq n_0. \]

Thus we have shown that $\frac{1}{2} n^2 - 3n = \Theta(n^2)$. 
Little Oh

• The function \( f(n) = o(g(n)) \) \( \text{[i.e f of n is a little oh of g of n]} \) if and only if

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]

The function \( 3n + 2 = o(n^2) \) since \( \lim_{n \to \infty} \frac{3n+2}{n^2} = 0. \)
• For comparing the order of growth limit is used

\[
\lim_{{n \to \infty}} \frac{t(n)}{g(n)} = \begin{cases} 
0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\
c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\
\infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).
\end{cases}
\]
EXAMPLE 1  Compare the orders of growth of $\frac{1}{2}n(n - 1)$ and $n^2$

$$\lim_{n \to \infty} \frac{\frac{1}{2}n(n - 1)}{n^2} = \frac{1}{2} \lim_{n \to \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \to \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$  

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n - 1) \in \Theta(n^2)$.

EXAMPLE 2  Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$.

$$\lim_{n \to \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \to \infty} \frac{(\log_2 n)'}{\left(\sqrt{n}\right)'} = \lim_{n \to \infty} \frac{(\log_2 e) \frac{1}{n}}{2\sqrt{n}} = 2 \log_2 e \lim_{n \to \infty} \frac{1}{\sqrt{n}} = 0.$$  

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n \to \infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called little-oh notation: $\log_2 n \in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)
**Theorem:** If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

(The analogous assertions are true for the $\Omega$ and $\Theta$ notations as well.)

**Proof:**

The proof extends to orders of growth the following simple fact about four arbitrary real numbers $a_1, b_1, a_2, b_2$:

- If $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant $c_1$ and some nonnegative integer $n_1$ such that

$$t_1(n) \leq c_1 g_1(n) \text{ for all } n \geq n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$, $t_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$. 
Theorem: If \( t_1(n) \in O(g_1(n)) \) and \( t_2(n) \in O(g_2(n)) \), then \( t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}) \).
(The analogous assertions are true for the \( \Omega \) and \( \Theta \) notations as well.)

Proof: (continued):

Let us denote \( c_3 = \max\{c_1, c_2\} \) and consider \( n \geq \max\{n_1, n_2\} \) so that we can use both inequalities.

Adding them yields the following:

\[
t_1(n) + t_2(n) \leq c_1g_1(n) + c_2g_2(n) \\
\leq c_3 g_1(n) + c_3g_2(n) = c_3[g_1(n) + g_2(n)] \\
\leq c_32 \max\{g_1(n), g_2(n)\}.
\]

Hence, \( t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}) \), with the constants \( c \) and \( n_0 \) required by the \( O \) definition being \( 2c_3 = 2 \max\{c_1, c_2\} \) and \( \max\{n_1, n_2\} \), respectively.
# Basic Efficiency classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>Short of best-case efficiencies, very few reasonable examples can be given since an algorithm’s running time typically goes to infinity when its input size grows infinitely large.</td>
</tr>
<tr>
<td>( \log n )</td>
<td>logarithmic</td>
<td>Typically, a result of cutting a problem’s size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.</td>
</tr>
<tr>
<td>( n )</td>
<td>linear</td>
<td>Algorithms that scan a list of size ( n ) (e.g., sequential search) belong to this class.</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>linearithmic</td>
<td>Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.</td>
</tr>
<tr>
<td>Class</td>
<td>Name</td>
<td>Comments</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$n^2$</td>
<td>quadratic</td>
<td>Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.</td>
</tr>
<tr>
<td>$n^3$</td>
<td>cubic</td>
<td>Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.</td>
</tr>
<tr>
<td>$2^n$</td>
<td>exponential</td>
<td>Typical for algorithms that generate all subsets of an $n$-element set. Often, the term “exponential” is used in a broader sense to include this and larger orders of growth as well.</td>
</tr>
<tr>
<td>$n!$</td>
<td>factorial</td>
<td>Typical for algorithms that generate all permutations of an $n$-element set.</td>
</tr>
</tbody>
</table>
Quiz: Which kind of growth best characterizes each of these functions?

<table>
<thead>
<tr>
<th>Function</th>
<th>Constant</th>
<th>Linear</th>
<th>Polynomial</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$3n^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(3/2)^n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(3/2)n$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2n^3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Module 1 – Outline
Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. Fundamental Data Structures
Mathematical Analysis of Non-recursive Algorithms

1. Decide on a parameters indicating an input’s size.
2. Identify the algorithm’s basic operation.
3. Check whether the number of times the basic operation is executed depends only on the size of an input.
   If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm’s basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.
**ALGORITHM**  \( \text{MaxElement}(A[0..n-1]) \)

//Determines the value of the largest element in a given array
//Input: An array \( A[0..n-1] \) of real numbers
//Output: The value of the largest element in \( A \)

\[
\text{maxval} \leftarrow A[0] \\
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do} \\
    \text{if } A[i] > \text{maxval} \\
    \quad \text{maxval} \leftarrow A[i] \\
\text{return } \text{maxval}
\]

Best, Worst, Average case exist?

\[
C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).
\]
**ALGORITHM**  
*UniqueElements*(A[0..n − 1])

// Determines whether all the elements in a given array are distinct
// Input: An array A[0..n − 1]
// Output: Returns “true” if all the elements in A are distinct and “false” otherwise

for $i \leftarrow 0$ to $n - 2$ do
    for $j \leftarrow i + 1$ to $n - 1$ do

return true

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n - 1) - (i + 1) + 1] = \sum_{i=0}^{n-2} (n - 1 - i)$$

$$= \sum_{i=0}^{n-2} (n - 1) - \sum_{i=0}^{n-2} i = (n - 1) \sum_{i=0}^{n-2} 1 - \frac{(n - 2)(n - 1)}{2}$$

$$= (n - 1)^2 - \frac{(n - 2)(n - 1)}{2} = \frac{(n - 1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Best, Worst, Average case exist?
**ALGORITHM**  \( \text{MatrixMultiplication}(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1]) \)

//Multiplies two square matrices of order \( n \) by the definition-based algorithm
//Input: Two \( n \times n \) matrices \( A \) and \( B \)
//Output: Matrix \( C = AB \)
for \( i \leftarrow 0 \) to \( n - 1 \) do
  for \( j \leftarrow 0 \) to \( n - 1 \) do
    \( C[i, j] \leftarrow 0.0 \)
    for \( k \leftarrow 0 \) to \( n - 1 \) do
      \( C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j] \)
return \( C \)


Best, Worst, Average case exist?
Analysis

for $i \leftarrow 0$ to $n - 1$ do
  for $j \leftarrow 0$ to $n - 1$ do
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ to $n - 1$ do
      $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
  return $C$

$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$

$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3.$

Best, Worst, Average case exist?
**ALGORITHM**  

*Binary(n)*

//Input: A positive decimal integer *n*  
//Output: The number of binary digits in *n*’s binary representation

```
  count ← 1
  while *n* > 1 do
    count ← count + 1
    *n* ← ⌊*n*/2⌋
  return count
```

The basic operation is count=count + 1 repeats $\lceil \log_2 n \rceil + 1$ number of times

Best, Worst, Average case exist?
Module 1 – Outline

Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. Fundamental Data Structures
Analysis of Recursive Algorithms

1. Decide on a parameter indicating an input’s size.
2. Identify the algorithm’s basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.
EXAMPLE 1  Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer $n$. Since

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM  $F(n)$

// Computes $n!$ recursively
// Input: A nonnegative integer $n$
// Output: The value of $n!$
if $n = 0$ return 1
else return $F(n-1) \cdot n$

$M(n) = M(n-1) + 1$ for $n > 0$.

\[M(n) = M(n-1) + 1 \quad \text{for } n > 0,
M(0) = 0.\]
• We can use backward substitutions method to solve this

\[
M(n) = M(n - 1) + 1 \quad \text{substitute } M(n - 1) = M(n - 2) + 1 \\
= [M(n - 2) + 1] + 1 = M(n - 2) + 2 \quad \text{substitute } M(n - 2) = M(n - 3) + 1 \\
= [M(n - 3) + 1] + 2 = M(n - 3) + 3.
\]

\[
= M(n - i) + i = \cdots = M(n - n) + n = n.
\]
Tower of Hanoi puzzle.

- In this puzzle, there are \( n \) disks of different sizes that can slide onto any of three pegs.
- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
- The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary.
- We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.
Tower of Hanoi puzzle.

• The problem has an elegant recursive solution
• To move n>1 disks from peg 1 to peg 3 (with peg 2 as auxiliary),
  – we first move recursively n-1 disks from peg 1 to peg 2 (with peg 3 as auxiliary),
  – then move the largest disk directly from peg 1 to peg 3, and,
  – finally, move recursively n-1 disks from peg 2 to peg 3 (using peg 1 as auxiliary).
• If n = 1, we move the single disk directly from the source peg to the destination peg.
Algorithm

TowerOfHanoi(n, source, dest, aux)
   If n == 1, then
      move disk from source to dest
   else
      TowerOfHanoi (n - 1, source, aux, dest)
      move disk from source to dest
      TowerOfHanoi (n - 1, aux, dest, source)
   End if
Recurrence relation for total number of moves

The number of moves $M(n)$ depends only on $n$. The recurrence equation is

$$M(n) = M(n - 1) + 1 + M(n - 1) \quad \text{for } n > 1.$$ 

We have the following recurrence relation for the number of moves $M(n)$:

$$M(n) = 2M(n - 1) + 1 \quad \text{for } n > 1$$

$M(1) = 1.$
• We solve this recurrence by the same method of backward substitutions:

\[
M(n) = 2M(n - 1) + 1 \\
= 2[2M(n - 2) + 1] + 1 = 2^2M(n - 2) + 2 + 1 \\
= 2^2[2M(n - 3) + 1] + 2 + 1 = 2^3M(n - 3) + 2^2 + 2 + 1.
\]

• The pattern of the first three sums on the left suggests that the next one will be

\[
2^4M(n - 4) + 2^3 + 2^2 + 2 + 1,
\]

and generally, after \(i\) substitutions, we get

\[
M(n) = 2^i M(n - i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n - i) + 2^i - 1.
\]
• Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence

\[
M(n) = 2^{n-1} M(n - (n - 1)) + 2^{n-1} - 1 \\
= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1
\]
Example 3

**Algorithm** \( \text{BinRec}(n) \)

```plaintext
// Input: A positive decimal integer \( n \)
// Output: The number of binary digits in \( n \)'s binary representation
if \( n = 1 \) return 1
else return \( \text{BinRec}([n/2]) + 1 \)
```

- Basic operation is **Addition**
- The recurrence relation can be written as
  \[
  A(n) = A([n/2]) + 1 \quad \text{for } n > 1
  \]
- Assuming \( n = 2^k \)
  \[
  A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0, \\
  A(2^0) = 0.
  \]
Recurrence relation for basic operation

\[ A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0, \]
\[ A(2^0) = 0. \]

Now backward substitutions encounter no problems:

\[ A(2^k) = A(2^{k-1}) + 1 \]
\[ = [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 \quad \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \]
\[ = [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 \]
\[ \quad \ldots \]
\[ = A(2^{k-i}) + i \]
\[ \quad \ldots \]
\[ = A(2^{k-k}) + k. \]

Thus, we end up with

\[ A(2^k) = A(1) + k = k, \]

or, after returning to the original variable \( n = 2^k \) and hence \( k = \log_2 n \),

\[ A(n) = \log_2 n \in \Theta(\log n). \]
Module 1 – Outline

Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. Fundamental Data Structures
Important Problem Types

• **Sorting**
  – rearrange the items of a given list in **non-decreasing order**
  – there is no algorithm that would be the best solution in all situations

  **Two properties**
  – Algorithm is **stable** if it preserves the relative order of any two equal elements in its input
  – **in-place** - no extra memory

• **Searching**
  – Linear search
  – Binary search
Important Problem Types

• String Processing
• Graph Problems
  – Oldest problems
• Combinatorial Problems
  – grows extremely fast with a problem’s size
  – there are no known algorithms for solving such problems exactly in an acceptable amount of time
Module 1 – Outline

Introduction to Algorithms

1. Introduction
2. Performance Analysis
3. Asymptotic Notations
4. Mathematical analysis of Non-Recursive algorithms
5. Mathematical analysis of Recursive algorithms
6. Important Problem Types
7. **Fundamental Data Structures**
Fundamental Data structures

- Linear data structures
  - Array
  - Linked list
    - Singly linked list
    - Doubly linked list
  - List
    - Stack
    - Queue, Priority queue
Fundamental Data structures

• Graphs
  – Undirected
  – Directed (Digraph)
  – Weighted graph
  – cycle
Fundamental Data structures

- Graph Representations
  - Adjacency matrix
  - Adjacency list

**FIGURE 1.7** (a) Adjacency matrix and (b) adjacency lists of the graph in Figure
Fundamental Data structures

• Graph Representations
  – Weighted graph
  – cycle

![Graph Diagram]

**FIGURE 1.8** (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.
Fundamental Data structures

• Trees, Forests
  – Rooted tree
  – Depth of vertex v
  – Height of the tree

(a) Tree. (b) Forest.

1.11 (a) Free tree. (b) Its transformation into a rooted tree.
**FIGURE 1.14** (a) First child–next sibling representation of the tree in Figure 1.11b. (b) Its binary tree representation.
Fundamental Data structures

- Trees, Forests
  - Ordered tree
    - Binary tree
    - Binary search tree

2. (a) Binary tree, (b) Binary search tree.
Fundamental Data structures

• Sets
  – Set operations
    • Check membership, Union, Intersection
  – Implementations
    • Bit vector, List

\[
\begin{array}{ccccccccccc}
U &=& \{\text{banana, apple, pear, peach, guava, apricot, watermelon, tomato}\} \\
1 &1 &1 &1 &1 &1 &1 &1 &1 &1 \\
\hline
A &=& \{\text{apple, pear, peach}\} \\
0 &1 &1 &1 &0 &0 &0 &0 &0 &0 \\
\hline
B &=& \{\text{watermelon, apple, pear}\} \\
0 &1 &1 &0 &0 &0 &1 &0 &0 &0 \\
\hline
C &=& \{\text{tomato}\} \\
0 &0 &0 &0 &0 &0 &0 &0 &1 &0 \\
\hline
D &=& \{\text{banana, peach, apricot, guava}\} \\
1 &0 &0 &1 &1 &1 &1 &0 &0 &0 \\
\end{array}
\]
Fundamental Data structures

• Dictionaries
  – Searching
  – Adding
  – Deleting
Consider the following algorithm for finding the distance between the two closest elements in an array of numbers. Make as many improvements as you can in this algorithmic solution to the problem. If you need to, you may change the algorithm altogether; if not, improve the implementation given.

**ALGORITHM**  \(\text{MinDistance}(A[0..n-1])\)

```
//Input: Array A[0..n-1] of numbers
//Output: Minimum distance between two of its elements
\(d_{\text{min}} \leftarrow \infty\)

for \(i \leftarrow 0\) to \(n-1\) do
    for \(j \leftarrow 0\) to \(n-1\) do
        if \(i \neq j\) and \(|A[i] - A[j]| < d_{\text{min}}\)
            \(d_{\text{min}} \leftarrow |A[i] - A[j]|\)

return \(d_{\text{min}}\)
```
Consider the following algorithm.

```
ALGORITHM Secret(A[0..n − 1])
    //Input: An array A[0..n − 1] of n real numbers
    minval ← A[0]; maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] < minval
            minval ← A[i]
        if A[i] > maxval
            maxval ← A[i]
    return maxval − minval
```

a. What does this algorithm compute?
b. What is its basic operation?
c. How many times is the basic operation executed?
d. What is the efficiency class of this algorithm?
e. Suggest an improvement/better algorithm, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.
Extra Byte-1.3: Enigma

Consider the following algorithm.

```
ALGORITHM Enigma(A[0..n-1, 0..n-1])
    // Input: A matrix A[0..n-1, 0..n-1] of real numbers
    for i ← 0 to n-2 do
        for j ← i + 1 to n-1 do
            if A[i, j] ≠ A[j, i]
                return false
        return true
```

a. What does this algorithm compute?
b. What is its basic operation?
c. How many times is the basic operation executed?
d. What is the efficiency class of this algorithm?
e. Suggest an improvement/better algorithm, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.
Assignment-1

1. Prove the following
   a) $100n + 5 = O(n)$      b) $1000n^2 + 100n - 6 = O(n^2)$

2. Explain asymptotic notations with examples.

3. Consider the following algorithm.
   a. What does the algorithm compute?
   b. What is basic operation?
   c. What is the efficiency of this algorithm?

   ```
   Algorithm GUESS (A[ ][ ])
   for i ← 0 to n - 1
     for j ← 0 to i
       A[i][j] ← 0
   ```

4. Explain mathematical analysis of recursive algorithm for Towers of Hanoi. Give the algorithm

5. Explain two common ways to represent the graph with example
1. Consider the following algorithm (6M)
   a. What does this algorithm compute?
   b. What is its basic operation?
   c. How many times is the basic operation executed?
   d. What is the efficiency class of this algorithm?
   e. Suggest an better algorithm if any and indicate its efficiency class.

2. Solve the recurrence relation (6M)
   \[ M(n) = 2M(n-1) + 1 \] for \( n > 1 \); \( M(1) = 1 \)

3. Define best case, worst case and average case efficiency. Write the algorithm and give these efficiencies for sequential search. (8M)
End of Module-1