
KTU NOTES

# Basic Concepts of Data Structures



MODULE I

Jacob P Cherian
Asst.Professor
Dept.of CSE, Saintgits College of Engineering

# Contents

System Life Cycle

Algorithms

Performance Analysis

Space Complexity

Time Complexity

Asymptotic Notation

Complexity Calculation of Simple Algorithms

# Algorithms- The Definition

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.



An algorithm is thus a sequence of computational steps that transform the input into the output

An algorithm is independent of the programming language.

# Criteria for an Algorithm

(i) **input:** there are zero or more quantities which are externally supplied;



(ii) **output:** at least one quantity is produced;

(iii) **definiteness:** each instruction must be clear and unambiguous;

# Criteria for an Algorithm

(iv) **finiteness:** if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

(v) **effectiveness:** every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper.

# Real Life Example of an Algorithm

Preparing Tea

Preparing a Hamburger

Your Daily Routine

Going to School/College

# Tea Recipe

1. Put the teabag in a cup.
2. Fill the kettle with water.
3. Boil the water in the kettle.
4. Pour some of the boiled water into the cup.
5. Add milk to the cup.
6. Add sugar to the cup.
7. Stir the **tea**.
8. Drink the **tea**.

# Time Complexity

How much time an algorithm will take to solve a problem?

In order to compare algorithms, we need a way to measure the time required by an algorithm.

# A Real World Analysis

# The Basic Idea of Time Complexity

If we solve a problem that’s ten times as large, how does the running time change?

If we run **find_max()** on a list that’s a thousand elements long instead of a hundred elements, does it take the same amount of time?

Does it take 10 times as long to run, 100 times, or 5 times? This is called the algorithm’s *time complexity.*

# How to Analyze Time Complexity?

What are the factors which affect the time complexity of an algorithm?

input,

programming language and runtime,

coding skill,

compiler,

operating system, and hardware.

# How to Analyze Time Complexity?

We often want to reason about **execution time** in a way that depends only on the **algorithm** and its **input**.



This can be achieved by choosing an **elementary operation**, which the algorithm performs repeatedly, and define the **time complexity** T($n$)

# How to Analyze Time Complexity?

In general, an **elementary operation** must have two properties:

1. There can't be any other operations that are performed more frequently as the size of the input grows.
2. The time to execute an elementary operation must be constant: it mustn't increase as the size of the input grows. This is known as unit cost.

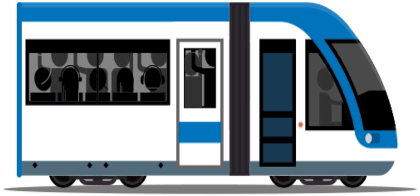
# Fairness-Performance Analysis

TRIVANDRUM TO CHENNAI VIA MADURAI    ROUTE A

ROUTE B    TRIVANDRUM TO CHENNAI VIA PALAKKAD



Mode of Travel



Time of Travel

Traffic Congestions

# Types of Analysis(Time Complexity)

Generally, we perform the following types of analysis −

**Worst-case** − The maximum number of steps taken on any instance of size a.



**Best-case** − The minimum number of steps taken on any instance of size a.

**Average case** − An average number of steps taken on any instance of size a

# Space Complexity

**Space complexity** is a measure of the amount of working storage an algorithm needs.

That means how much memory, in the worst case, is needed at any point in the algorithm.

# The Memory Requirement

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (include the constant values, temporary values)
2. Program Instruction
3. Execution

# Memory Requirements

**Auxiliary space** is extra space or temporary space used by the algorithms during its execution.

**Instruction Space** is used to save compiled instruction in the memory.

**Environmental Stack** is used to storing the addresses while a module calls another module or functions during execution.

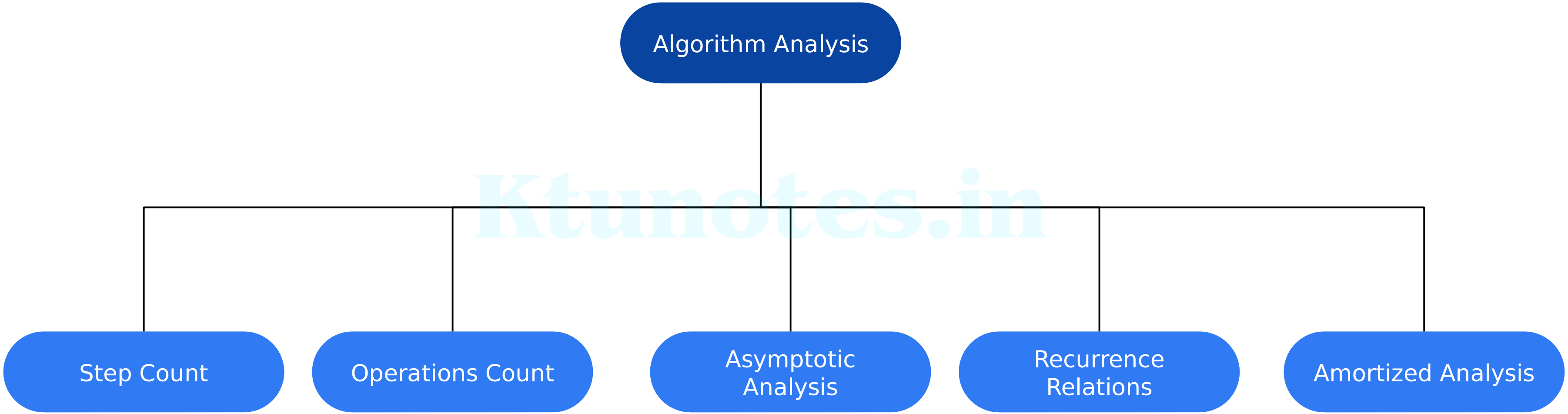
**Data space** is used to store data, variables, and constants which are stored by the program and it is updated during execution.

**Space Complexity = Auxiliary Space**

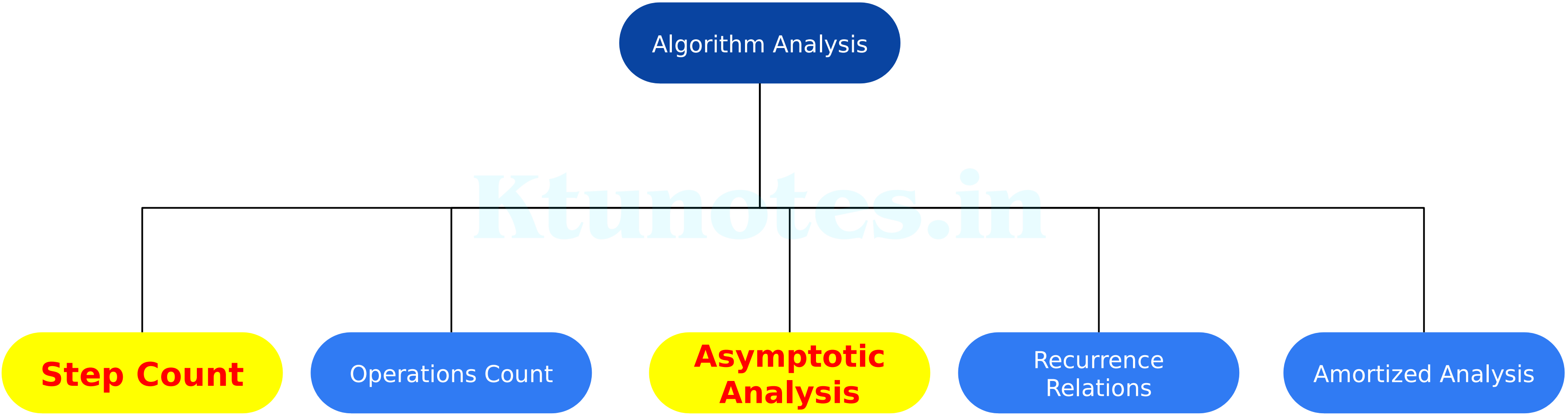
**+**

**Input space (Data Space)**

# Measuring Running Time



# Measuring Running Time



# Time Complexity:In-depth Analysis

Time complexity estimates the time to run an algorithm. It's calculated by counting elementary operations.

Best Case

Worst Case

Average Case

# Best Case Complexity

Let T1($n$), T2($n$), ... be the execution times for all possible inputs of size $n$.

The best-case time complexity W($n$) is then defined as

$$W(n) = \min(T1(n), T2(n), \ldots)$$

# Worst Case Complexity

Let T1($n$), T2($n$), … be the execution times for all possible inputs of size $n$.

The worst-case time complexity W($n$) is then defined as

$$W(n) = \max(T1(n), T2(n), \ldots)$$

Click to View Reference Article

# Average Case Complexity

Let T1($n$), T2($n$), … be the execution times for all possible inputs of size $n$, and let P1($n$), P2($n$), … be the probabilities of these inputs.

The average-case time complexity is then defined as

$$P1(n)T1(n) + P2(n)T2(n) + \ldots$$

Average-case time is often harder to compute, and it also requires knowledge of how the input is distributed.

# Understand the Loops

**Let n = 3**

| |
|---|
| for(i=0; i<n; i++) |
| { |
| sum++; |
| } |

i=0
i=1
i=2

# Understand the Loops

**n+1 times**

```
for(i=0; i<n; i++)
{
      sum++;
}
```

i=0
i=1
i=2

# Understand the Loops

**n+1 times** **n times**

```
for(i=0; i<n; i++)
{
      sum++;
}
```

i=0
i=1
i=2

# Understand the Loops

n+1 times    n times

```
for(i=0; i<n; i++)
{
        Sum++;
}
```

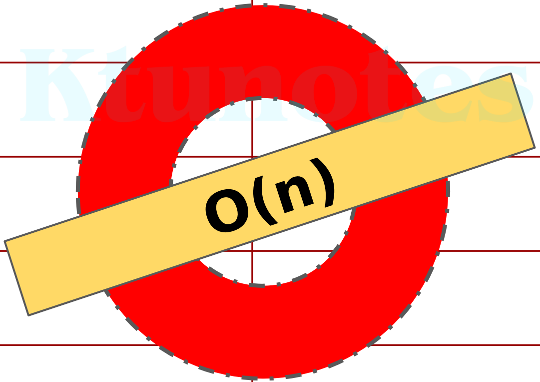
$\longrightarrow$ n times

i=0
i=1
i=2

# Example 1-Find Frequency Count

Ignore constants

| Code | Frequency |
|---|---|
| int sum=0; | 1 |
| for(i=0;i<n;i++) | n+1 |
| { | ----- |
| sum++; | n |
| } | ----- |
| **TOTAL** | **2n+2** |



# Example 2

| Algorithm sum(A,n) | Frequency |
|---|---|
| { | ----- |
| s=0 | 1 |
| for(i=0;i<n;i++) | n+1 |
| { | ----- |
| s=s+A[i] | n |
| } | ----- |
| return s | 1 |
| } | ----- |
| Frequency Count | 2n+3 |

**O(n)**

Ignore constants

## Example 3

| Algorithm Array_Sum(A,B,n) | Frequency |
|---|---|
| { | |
| for(i=0;i<n;i++) | |
| { | |
| for(j=0;j<n;j++) | |
| { | |
| c[i][j]=a[i][j]+b[i][j] | |
| } | |
| } | |
| return s | |
| } | |
| Frequency Count | |

Example 3

| Algorithm Array_Sum(A,B,n) | Frequency |
|---|---|
| { | ----- |
| for(i=0;i<n;i++) | n+1 |
| { | ----- |
| for(j=0;j<n;j++) | n |
| { | ----- |
| c[i][j]=a[i][j]+b[i][j] | n |
| } | ----- |
| } | ----- |
| return s | 1 |
| } | ----- |
| Frequency Count | |

| Algorithm Array_Sum(A,B,n) | Frequency |
|---|---|
| { | ----- |
| for(i=0;i<n;i++) | n+1 |
| { | ----- |
| for(j=0;j<n;j++) | n *(n+1) |
| { | ----- |
| c[i][j]=a[i][j]+b[i][j] | n*n |
| } | ----- |
| } | ----- |
| return s | 1 |
| } | ----- |
| Frequency Count | $2n^2+2n+3$ |

$O(n^2)$

Example 3

Ignore constants

# Homework Question-1

**Find the time complexity**

```
int fun(int n)
{
    int i = 0, j = 0, k = 0, m = 0;
    i = n;
    for (i = 0; i<n; i++)
        for (j = 0; j<n; j++)
            m += 1;
    for (i = 0; i<n; i++)
        for (k = 0; k<n; k++)
            m += 1;
    return m;
}
```

# Homework Question-2

**Find the time complexity**

```
int fun(int n)
{
    int i = 0, j = 0, k = 0, m = 0;
    i = n;
    for (i = 0; i<n; i++)
        for (j = 0; j<n; j++)
            for (k = 0; k<n; k++)
                m += 1;
    return m;
}
```

# Let's Explore Linear Search

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 7 | 8 | 9 | 12 | 67 | 21 | 0 | 2 | 91 |

Search for Element 5

Best Case

Search for Element 12

Average Case

Search for Element 91

Worst Case

# Linear Search Algorithm

```
Algorithm linear_search(A,key)
    for(i=0;i<n;i++)
  {
        If a[i]=key
            flag=1
            break
  }
return flag
```

# Frequency Count-Linear Search

| Algorithm linear_search(A,key) | Frequency |
|---|---|
| for(i=0;i<n;i++) | |
| If a[i]=key | |
| flag=1 | |
| break | |
| return flag | |
| **Frequency Count** | |

# Frequency Count-Linear Search

| Algorithm linear_search(A,key) | Frequency |
|---|---|
| for(i=0;i<n;i++) | n+1 |
| If a[i]=key | n |
| flag=1 | 1 |
| break | 1 |
| return flag | 1 |
| **Frequency Count** | 2n+4 |

# Best Case,Worst Case & Average Case Time Complexity-Linear Search

Best Case is when the first element of the array is the element to be searched



Worst Case is when the last element of the array is the element to be searched

Average Case is the average time required to search an element from the array

# Linear Search-Time Complexity

| | |
|---|---|
| BEST CASE | O(1) |
| WORST CASE | O(n) |
| AVERAGE CASE | O(n/2) |

# Let’s Play a Game

I’ll think of a number between 1 and 100 (inclusive).

You have to guess the number in the fewer number of possible steps.



Each time you guess a number,I’ll tell whether my number is lower or higher than your guess.

# Linear Search

1

Too Low!

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ~~1~~ | 2 | 3 | 4 | 5 | 6 | ... | 100 |



2

Too Low!

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~2~~ | 3 | 4 | 5 | 6 | ... | 100 |

# Linear Search


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~2~~ | ~~3~~ | ~~4~~ | ~~5~~ | 6 | ... | 100 |

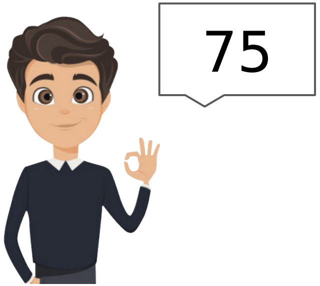



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~2~~ | ~~3~~ | ~~4~~ | ~~5~~ | ~~6~~ | ... | 100 |

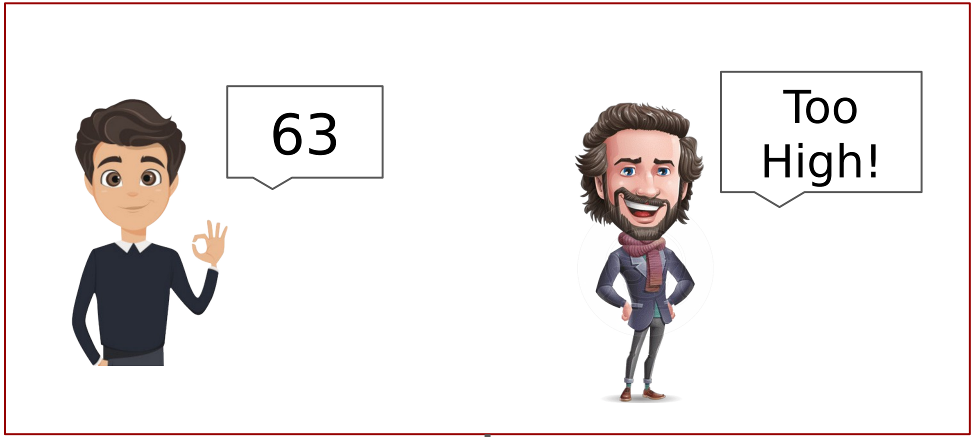
# A Better Way to Search

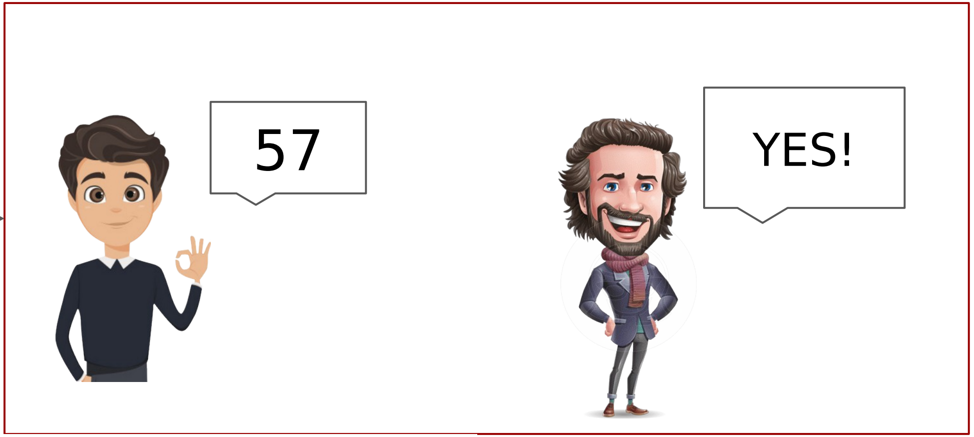


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ~~1~~ | ~~...~~ | ~~30~~ | ~~...~~ | ~~50~~ | 60 | ... | 100 |



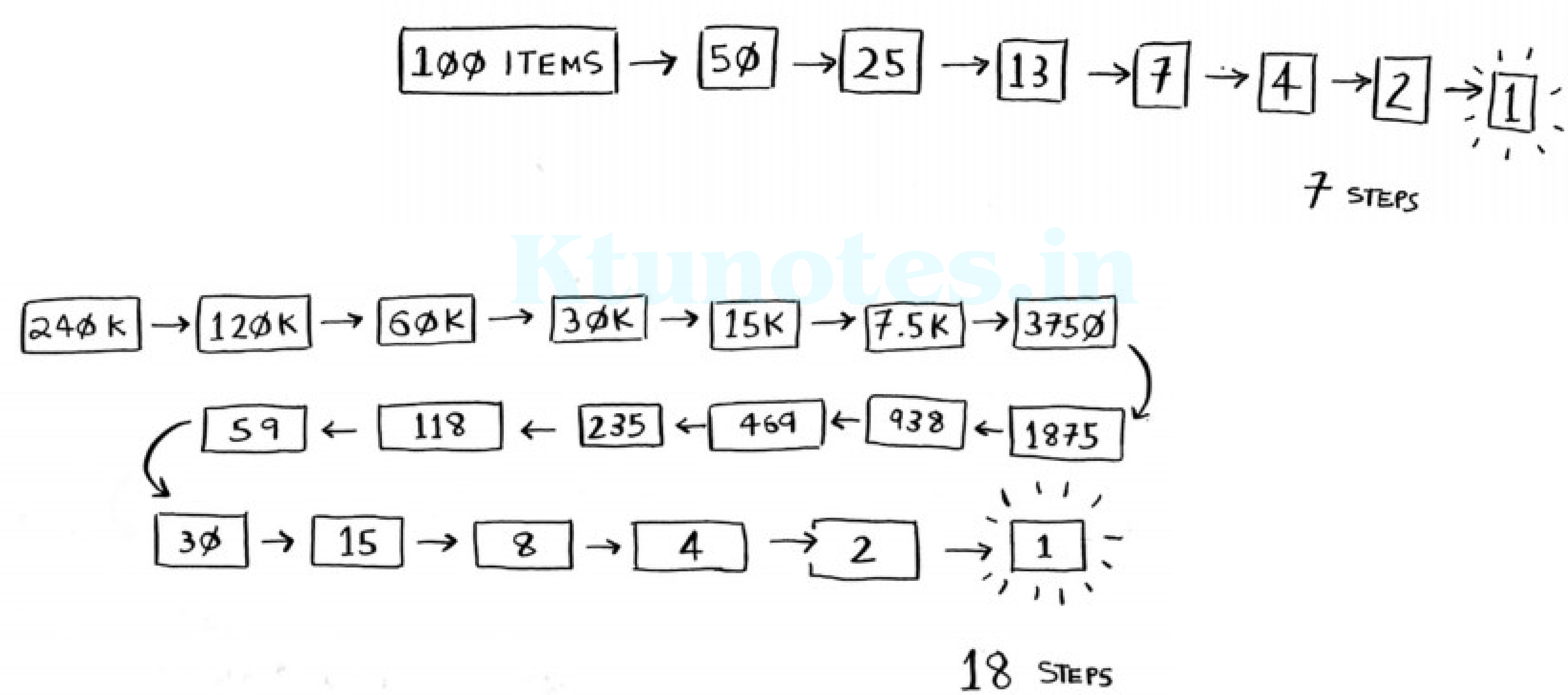

# A Better Way to Search



# A Better Way to Search

Here’s how many numbers you can eliminate every time.



# Asymptotic Notations

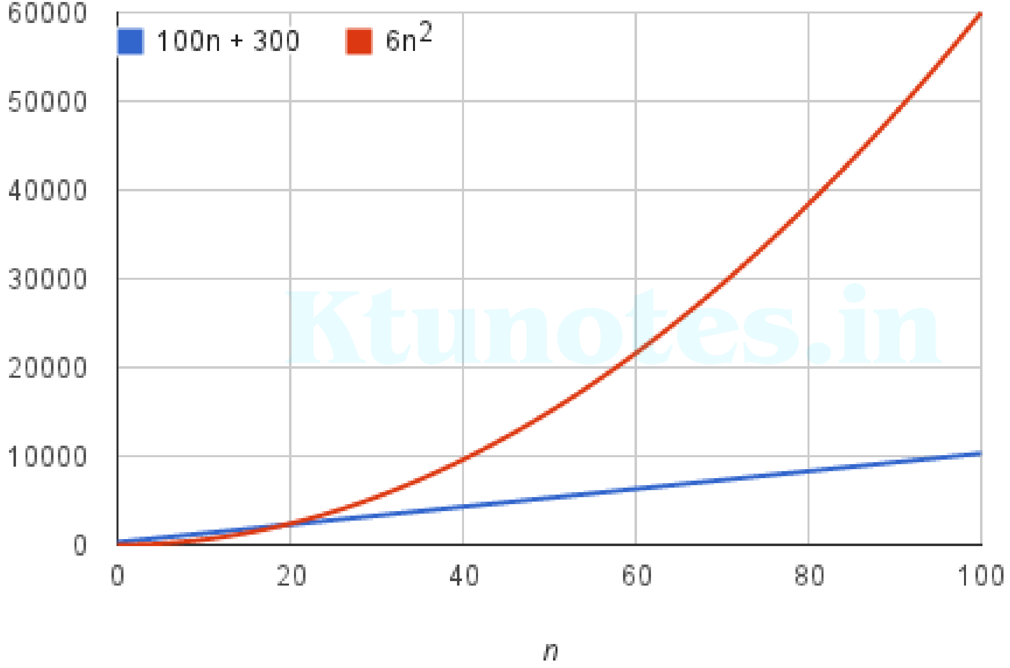
Reference

# Basic Idea

We must focus on how fast a function grows with the input size. We call this the **rate of growth** of the running time.

For example, suppose that an algorithm, running on an input of size n ,takes $6n^2+100n+300$ machine instructions. The 6n2 term becomes larger than 100n+300,once n becomes large enough, 20 in this case

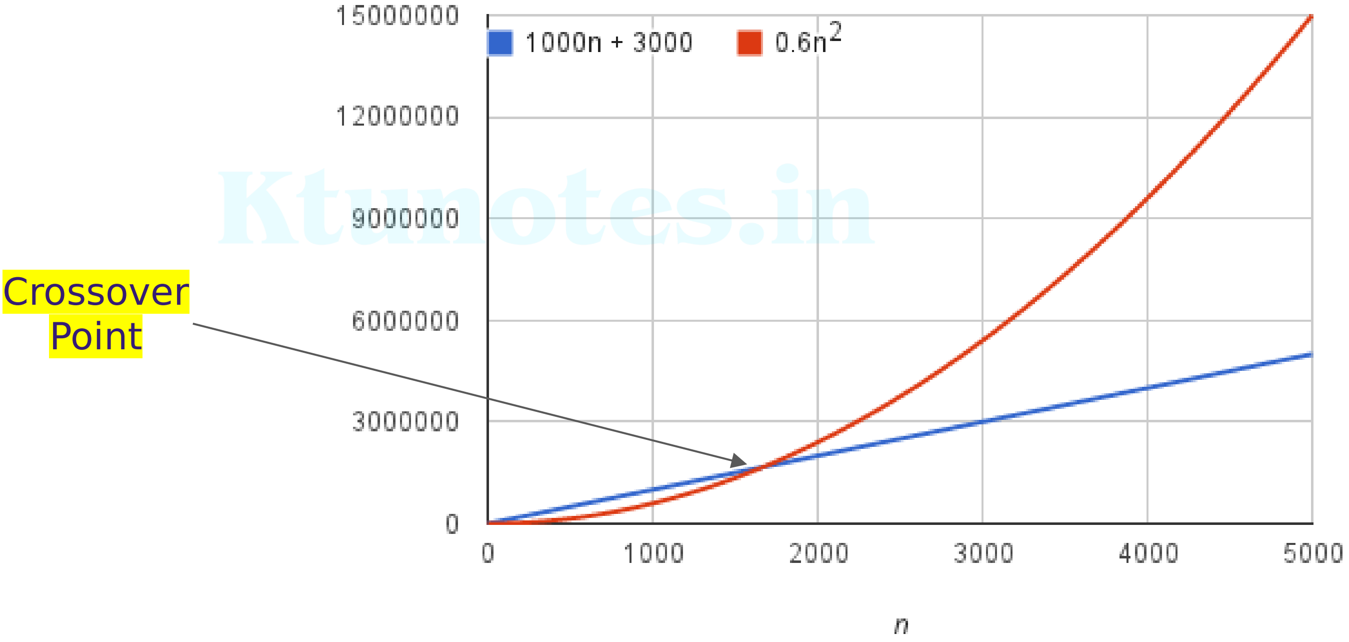
# Basic Idea




Source : https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

It doesn't really matter what coefficients we use; as long as the running time is $an^2+bn+c$ for some numbers $a>0$ ,b and c there will always be a value n (say n0) for which $an^2$ is greater than $bn+c$.

Basic Idea




Source : https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

# Types of Asymptotic Notations

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth.

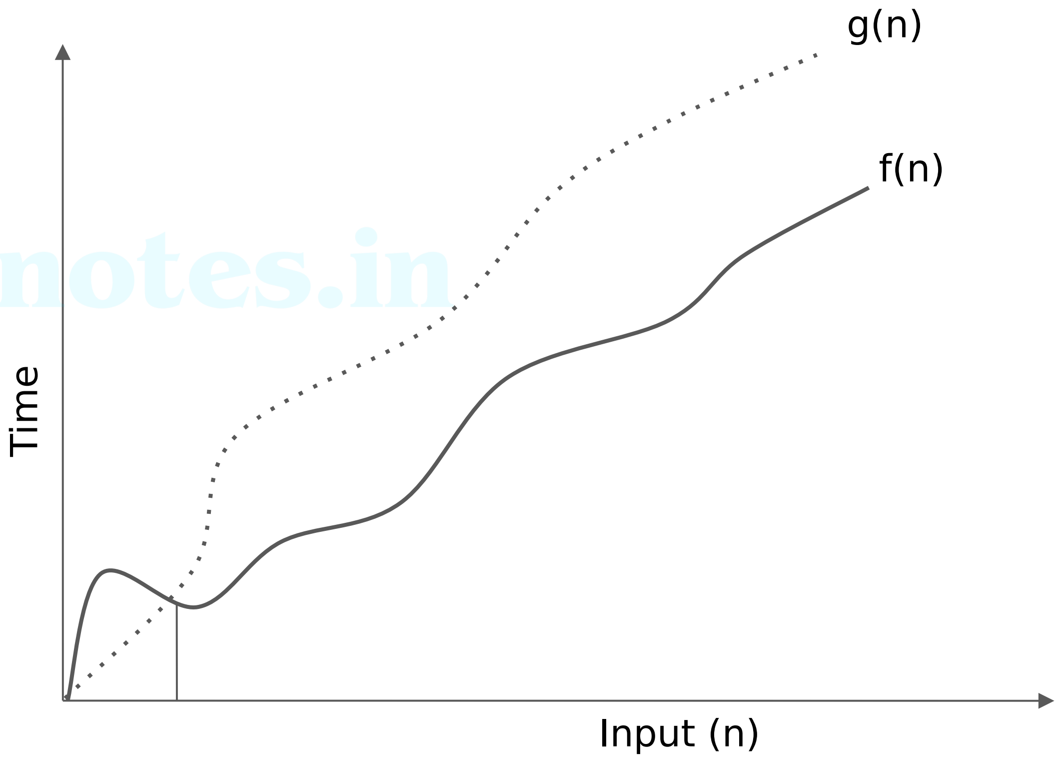


Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation ( Big-O Notation)
- Ω Notation (Omega Notation)
- θ Notation (Theta Notation)

# The Big-O Notation

We use Big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.

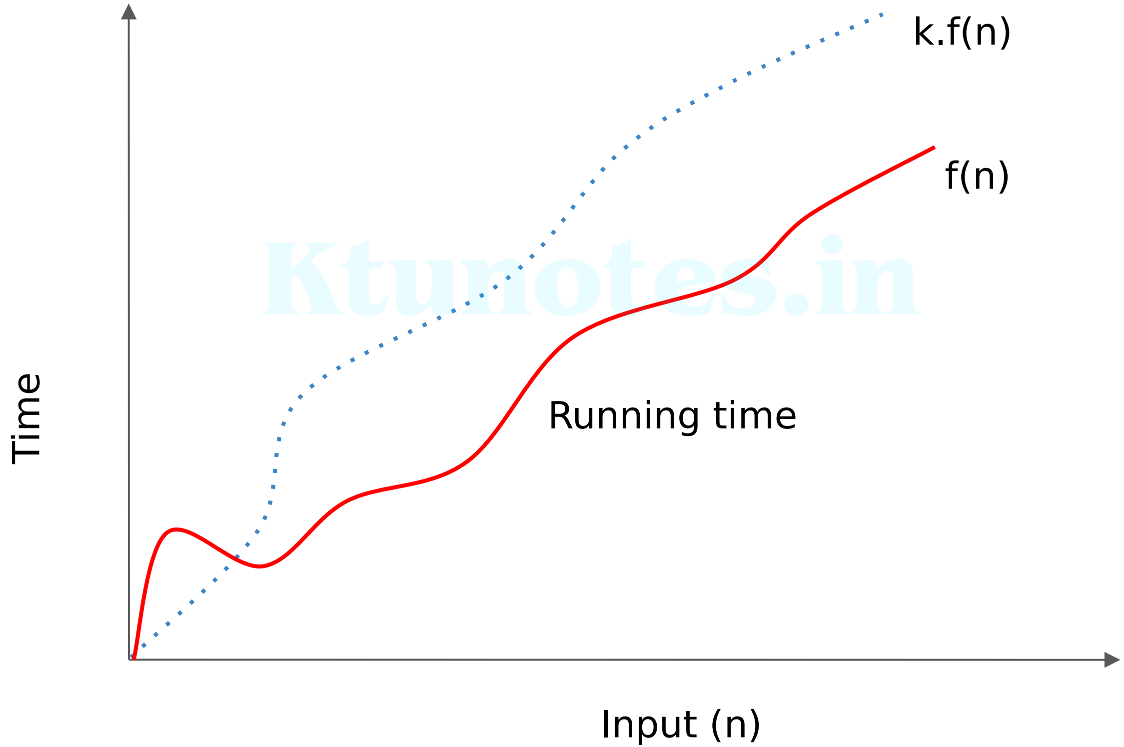


# Big O-The Definition

$f(n) = O(g(n))$ if there exists a positive integer $n0$ and a positive constant $k$, such that $0 <= f(n) \leq k.g(n)\ \forall\ n \geq n0$.

If a running time is O(f(n)),then for large enough n, the running time is at most k.f(n) for some constant k.

# Basic Idea



# What it tells?

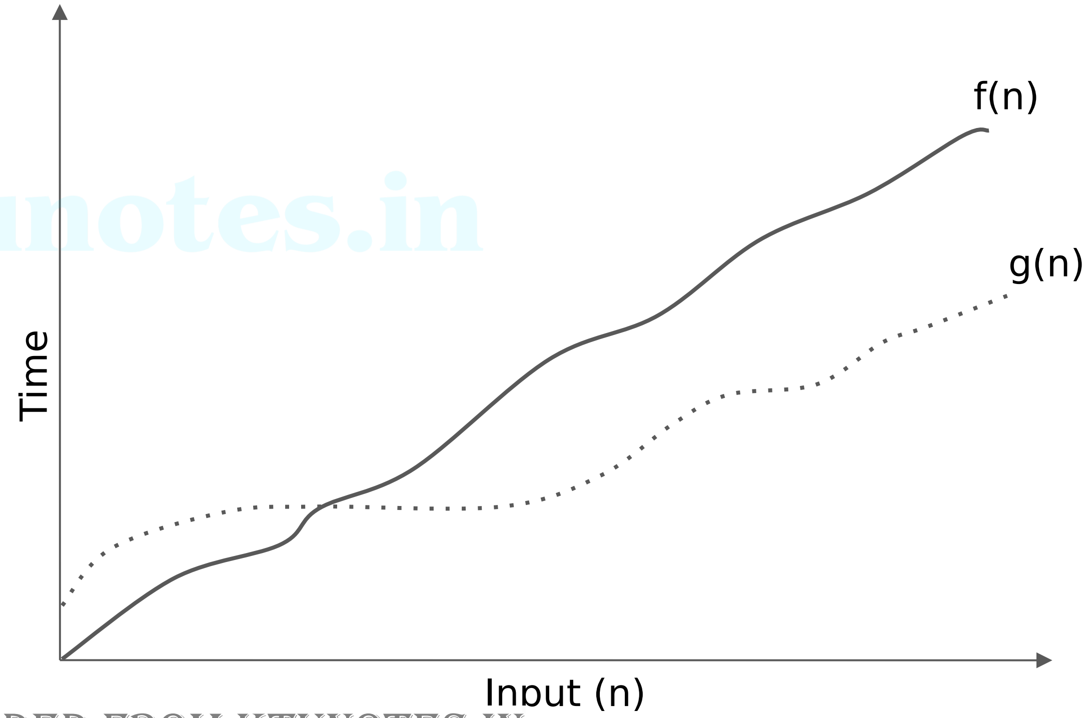
Big O notation tells you how fast an algorithm is. For example, suppose you have a list of size *n*. Simple search needs to check each element, so it will take *n* operations. The run time in Big O notation is O(*n*).



Big O doesn't tell you the speed in seconds. *Big O notation lets you compare the number of operations.* It tells you how fast the algorithm grows.

# The Big Omega Notation

We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.



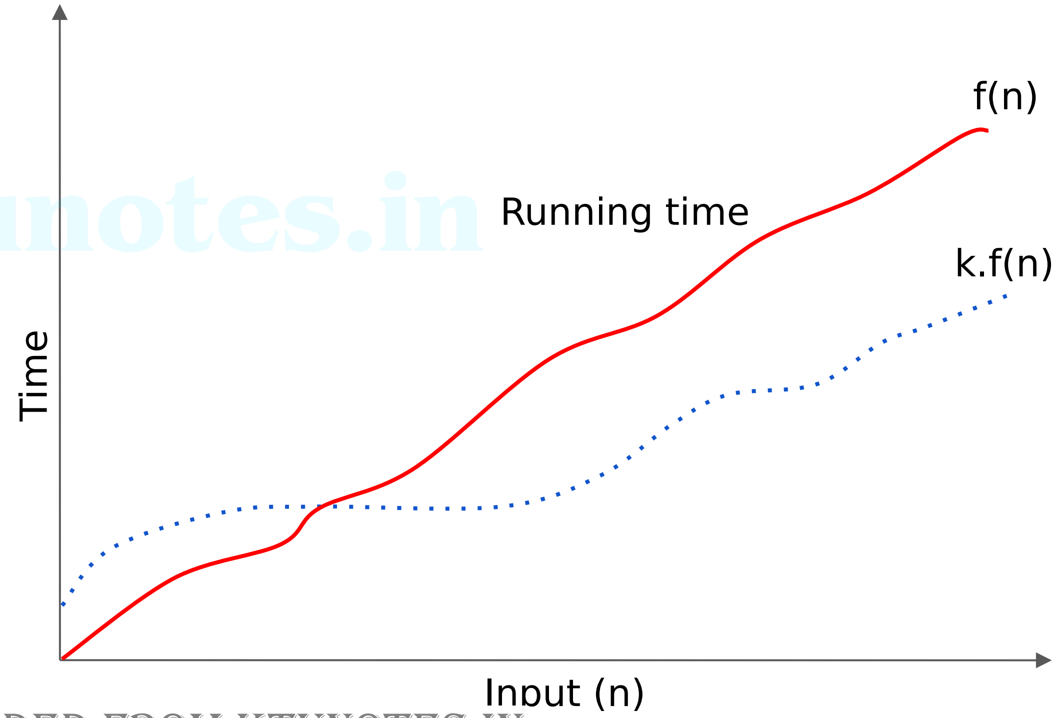
# The Big Omega Notation

**T**he Big Omega(Ω) provides us with the best case scenario of running an algorithm.

It would give us the minimum amount of resources (time or space) an algorithm would take to run.

# Basic Idea

If a running time is O(f(n)),then for large enough n, the running time is at least k.f(n) for some constant k.

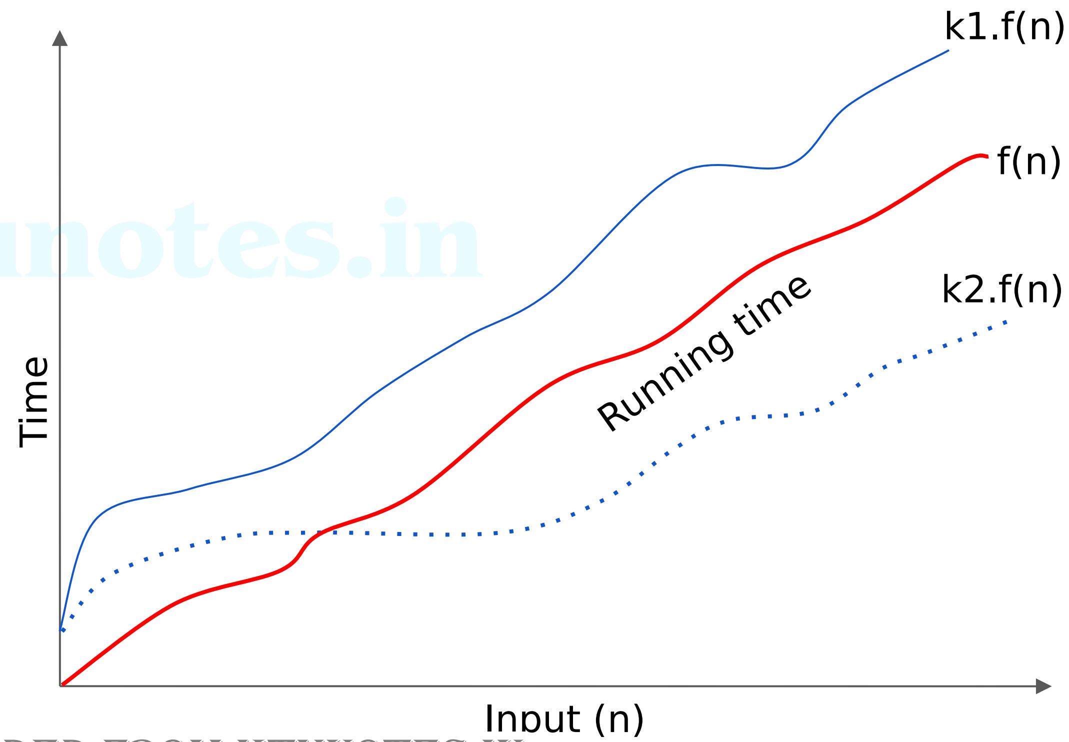


# Big Omega-The Definition

f(n) = **Ω**(g(n)) if there exists a positive integer n0 and a positive constant k, such that $0 \leq k.g(n) <= f(n)\ \forall\ n \geq n0$.

# Big Theta Notation

When we use big-Θ notation, we're saying that we have an **asymptotically tight bound** on the running time.



# Big Theta -The Definition

$f(n)=\Theta(g(n))$ if there exist positive constants $k1$, $k2$ and $n0$ such that $0 <= k1*g(n) <= f(n) <= k2*g(n)$ for all $n >= n0$

# Common Asymptotic Notations

| | |
|---|---|
| Constant | $O(1)$ |
| Logarithmic | $O(\log n)$ |
| Linear | $O(n)$ |
| nlogn | $O(n.\log n)$ |
| Quadratic | $O(n^2)$ |
| Cubic | $O(n^3)$ |
| Polynomial | $n^{O(1)}$ |
| Exponential | $2^{O(n)}$ |

# Practice Problem 1

```
int fun(int n)
{
    int m = 0;
    for (int i = 0; i<n; i++)
        m += 1;
    return m;
}
```

# Practice Problem 2

```
int fun(int n)
{
    int i=0, j=0, m = 0;
    for (i = 0; i<n; i++)
        for (j = 0; j<n; j++)
            m += 1;
    return m;
}
```

# Practice Problem 3

```
int fun(int n)
{
    int i=0, j=0, m = 0;
    for (i = 0; i<n; i++)
        for (j = 0; j<i; j++)
            m += 1;
    return m;
}
```

# Practice Problem 4

```
int i,p=0;
for(i=1;p<=n;i++)
      {
            p += i;
      }
      return m;
 }
```

# Practice Problem 5

```
int fun(int n)

{
    int i = 0, m = 0; i = n;
    while (i > 0)

    {

        m += 1;

        i = i / 2;

    }

    return m;
}
```

# Practice Problem 6

```
int fun(int n)
{
    int i = 0, j = 0, m = 0;
    for (i = 0; i<n; i++)
        for (j = 0; j< sqrt(n); j++)
            m += 1;
    return m;
}
```

# Practice Problem 7

```
int fun(int n)
{
    int i = 0, j = 0, m = 0;
    for (i = 0; i<n; i++)
        for (; j<n; j++)
            m += 1;
    return m;
}
```