Algorithm analysis

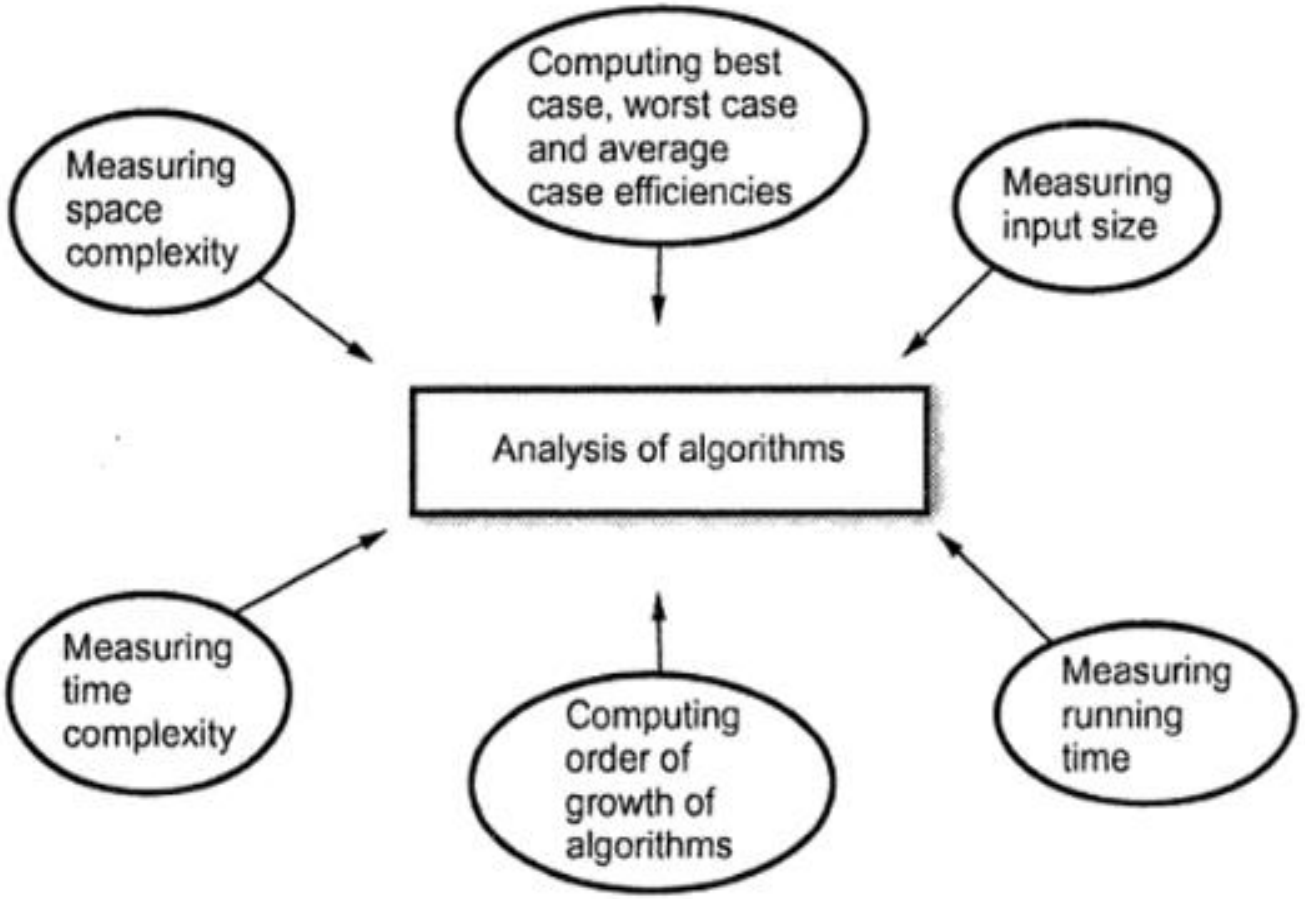
Course Objective

- Performance Analysis
  - Space complexity
  - Time complexity
  - Measuring an input size
  - Measuring Running Time
  - Order of Growth
- Asymptotic Notations
  - Big oh Notation
  - Omega Notation
  - θ Notation
  - Properties of Order of Growth

Analysis of Algorithms

- The efficiency of an algorithm can be decided by measuring the performance of an algorithm. We can measure the performance of an algorithm by computing two factors:
  - Amount of time required by an algorithm to execute
  - Amount of storage required by an algorithm
- This is popularly known as **time complexity and space complexity of an algoithm**

Analysis of Algorithms



How to analyze Programs?

- There are many criteria upon which we can judge program. For instance
  - Does my program work on what I want to do?
  - Does it work correctly according to original specifications?
  - Are procedures created in such a way that they perform logical functioning.
  - Is the code readable?

How to analyze Programs?

- If one asks the question that how much time it takes to execute this statement. It is impossible to determine exact timing taken by the statement to execute unless we have the following information.
  - The machine that is used to execute this statement,
  - Machine language instruction set.
  - The time required by each machine instruction.
  - The translation a compiler will make for this statement to machine language
  - And the kind of operating system (multi-programming or time sharing)

How to analyze Programs?

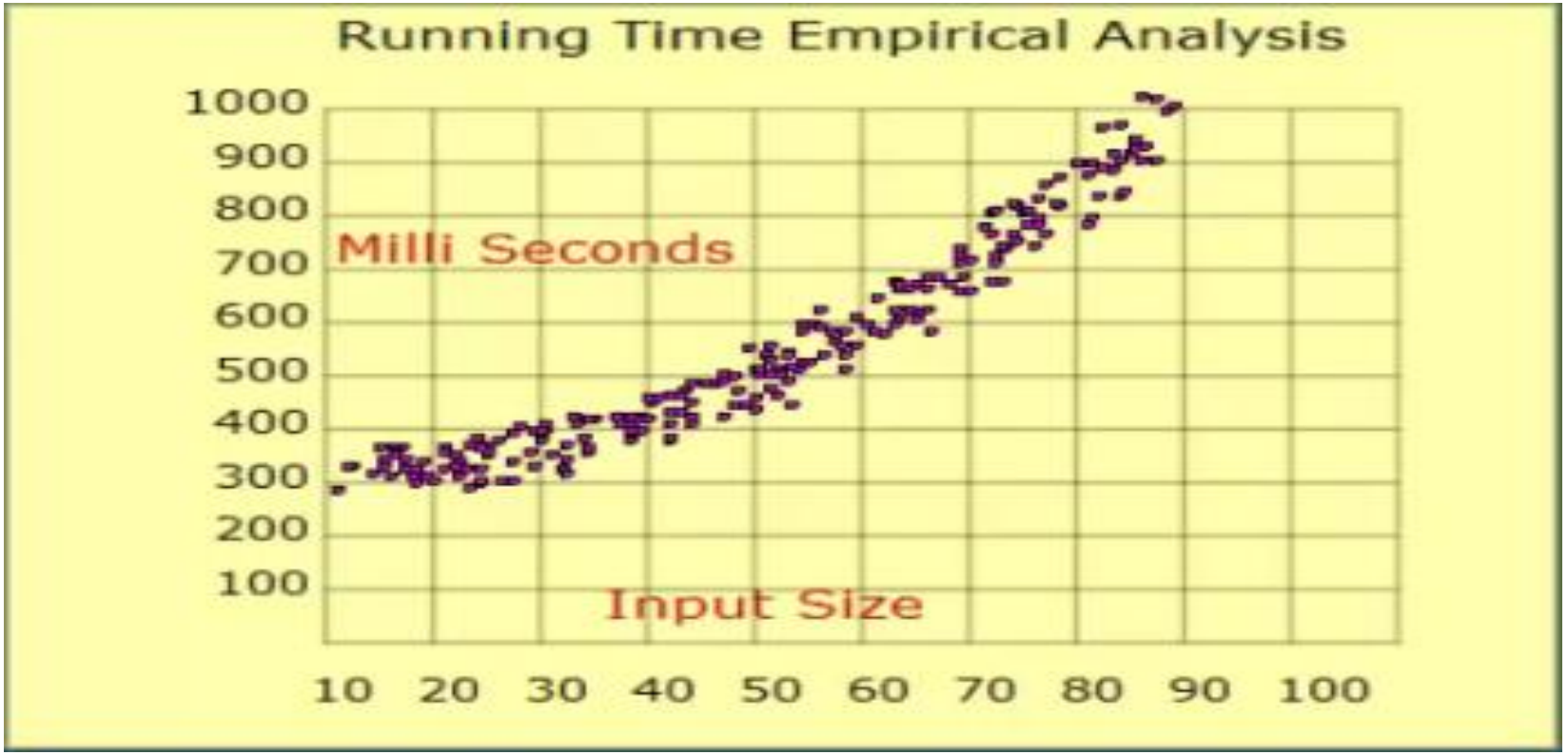
- The information may vary from machine to machine and perhaps we won't get the exact figures.
- So the better ideas is to determine the time taken by each instruction to execute called the '**frequency count**'.
- There are basically two approaches of analyzing algorithms
  - Empirical/ Experimental analysis
  - Analytical approach

Empirical/Experimental Analysis

- One way to study the efficiency of an algorithm is to implement it and experiment by running the program on various test inputs while recording the time spent during each execution
- The key is that if we record the time immediately before executing the algorithm and then immediately after, we can measure the *elapsed* time of an algorithm's execution by computing the difference of those times.
- Because we are interested in the general dependence of running time on the size and structure of the input, we should perform independent experiments on many different test inputs of various sizes.

Empirical/Experimental Analysis

- Running time is measured for different sets of input values.
- Actual time is plotted against the input size to analyze growth rate



Empirical/Experimental Analysis

- However, the measured times reported by any two methods will vary greatly from machine to machine, and may likely vary from trial to trial, even on the same machine. (CPU – Memory).

- Experiments are quite useful when comparing the efficiency of two or more algorithms, so long as they are gathered under similar circumstances.

Challenges of Experimental Analysis

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.

- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).

- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Moving Beyond Experimental Analysis

- Our goal is to develop an approach to analysing the efficiency of algorithms that:
  - Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
  - Is performed by studying a high-level description of the algorithm without need for implementation.
  - Takes into account all possible inputs.
  - Characterizes running time as a function of the input size, n.
- This calls for an analytical method that does not require implementation of an algorithm

Analytical Method

- This analysis uses mathematical techniques to estimate the running time.
- It identifies time consuming crucial operations in an algorithm.
- The number of key operations can be determined by analyzing the pseudo code.

Counting Primitive Operations

- To analyse the running time of an algorithm without performing experiments, we perform an analysis directly on a high-level description of the algorithm (either in the form of an actual code fragment, or language-independent pseudocode).
- Primitive operation is a low-level instruction with constant time.
- Counting the primitive operations can be used as measure for algorithm performance.
- We define a set of ***primitive operations*** such as the following:
  - Assigning a value to a variable
  - Following an object reference
  - Performing an arithmetic operation (for example, adding two numbers)
  - Comparing two numbers
  - Accessing a single element of an array by index
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.
- Frequency count in analytical method is highly recommended as it gives the efficiency of an algorithm without necessarily being machine dependent.

Frequency Count

```
Line 1: int sum_element (int a[10], int n)
Line 2:    {
Line 3:        int i, sum=0;
Line 4:              for (i=0; i<n; i++)
Line 5:    {
Line 6:               sum=sum+a[i];
Line 7:       }
Line 8:      return sum;
Line 9::     }
```

Frequency Count

Statement	Frequency count
i = 0	1
i < n	This statement executes for (n+1) times. When condition is true i.e. when i<n is true, the execution happens to be n times and the statement executes once more when i<n is false.
i ++	n times
sum=sum+ a[i]	n times
return sum	1
Total	**(3n+3)**

Frequency Count

- The efficiency of an algorithm can be measured by inserting a counter in the algorithm in order to count the number of times the basic operation is executed.
- This is a straightforward method of counting an efficiency of algorithm.
- For example: if we write a function for calculating sum of n number in an array then we find the efficiency of that function by inserting a frequency count. The frequency count is a count that denotes how many times that particular statement is executed.

Frequency count

```
for (int i=0; i<n; i++) // i=i+i
   {
          sum+=1; // sum=sum+i
    }
```

$T(n)= 1+(1+n)+n(1+1+1+1)$

$T(n) = 2+n+4n$

$T(n) = 5n+2$

Frequency count

```
For (int i=0; i<n; i++) //  1+(1+n)+2n
{
    for (int j=0; j<n*n; j++)   // n(1+(1+n)+2n)
    {
        sum +=i+j; //sum=sum+i+j  ---------n*n(1+1+1)
     }
  }
```

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

$T(n)=2+n+n(2+n+n(5))$

$T(n)=2+n+n(2+n+5n)$

$T(n)=6n^2+5n+2$

Frequency count

```
A()
{
  int i, j, k, n;
    for (i =1; i<=n, i++) //  1+(1+n)+2n
       {
          for (j=0; j<=i; j++)  //n (1+(1+n)+2n)
              {
                 for (k=1; k<=100, k++) // n^2(1+(1+n)+2n)
                  {
                       pf('rari');
                   }
              }
      }
```

Frequency count

```
A()
{
  int i, j, k, n;
    for (i =1; i<=n, i++) //  1+(1+n)+2n
       {
          for (j=0; j<=i; j++)  //n (1+(1+n)+2n)
              {
                 for (k=1; k<=n/2, k++) // n^2(1+(1+n)+2n)
                  {
                       pf('rari');
                   }
              }
       }
```

Common functions used in Analysis

- **The Constant Function *f*(*n*) = C**
  - For any argument *n*, the constant function *f*(*n*) assigns the value *C*. It doesn't matter what the input size *n* is, *f*(*n*) will always be equal to the constant value *C*.
  - The most fundamental constant function is *f*(*n*) = 1, and this is the typical constant function that is used
  - The constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers. Executing one instruction a fixed number of times also needs constant time only.
  - *Constant* algorithm does not depend on the input size.
  - **Examples**: arithmetic calculation, comparison, variable declaration, assignment statement, invoking a method or function.

Common functions used in Analysis

- **The Logarithm Function *f*(*n*) = log*n***
  - Running time varies as log N, where N is size of problem.
  - It grows slowly, and has the best performance.
  - If the problem size increases by factor of 1000, running time increases by 10
  - **Example:** Binary search algorithm
- **Linear Algorithm O(N)**
  - Execution time grows in direct proportion to the size of a problem.
  - Performance is rated as good
  - Algorithm, which are based on a single loop, shows linear growth rate.
  - **Examples are**: searching, deleting and inserting operations in an array

Common functions used in Analysis

- **The N-Log-N Function $f(n) = n\log n$**
  - This function grows a little faster than the linear function and a lot slower than the quadratic function (n^2)
  - **Example**: Divide – and – Conquer algorithms show this behavior like the merge sort, quick sort etc.
- **The Quadratic Function $f(n) = n^2$**
  - It appears a lot in the algorithm analysis, since there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. In such cases, the algorithm performs $n*n = n^2$ operations.
  - Quadratic algorithms are practical for relatively small problems. Whenever *n* doubles, the running time increases fourfold.
  - Example: some manipulations of the *n* by *n* array.

Common functions used in Analysis

- **The Cubic Function and Other Polynomials**
  - The cubic function $f(n) = n^3$. This function appears less frequently in the context of the algorithm analysis than the constant, linear, and quadratic functions. It's practical for use only on small problems.
  - Whenever *n* doubles, the running time increases eightfold.
  - Example: *n* by *n* matrix multiplication.
- **The Exponential Function $f(n) = b^n$**
  - In this function, *b* is a positive constant, called the **base**, and the argument *n* is the **exponent**.
  - Exponential algorithm is usually not appropriate for practical use.
  - **Example**: Towers of the Hanoi.

Common functions used in Analysis

- **The Factorial Function $f(n) = n!$**
  - Factorial function is even worse than the exponential function. Whenever n increases by 1, the running time increases by a factor of n.
  - For example, permutations of n elements.

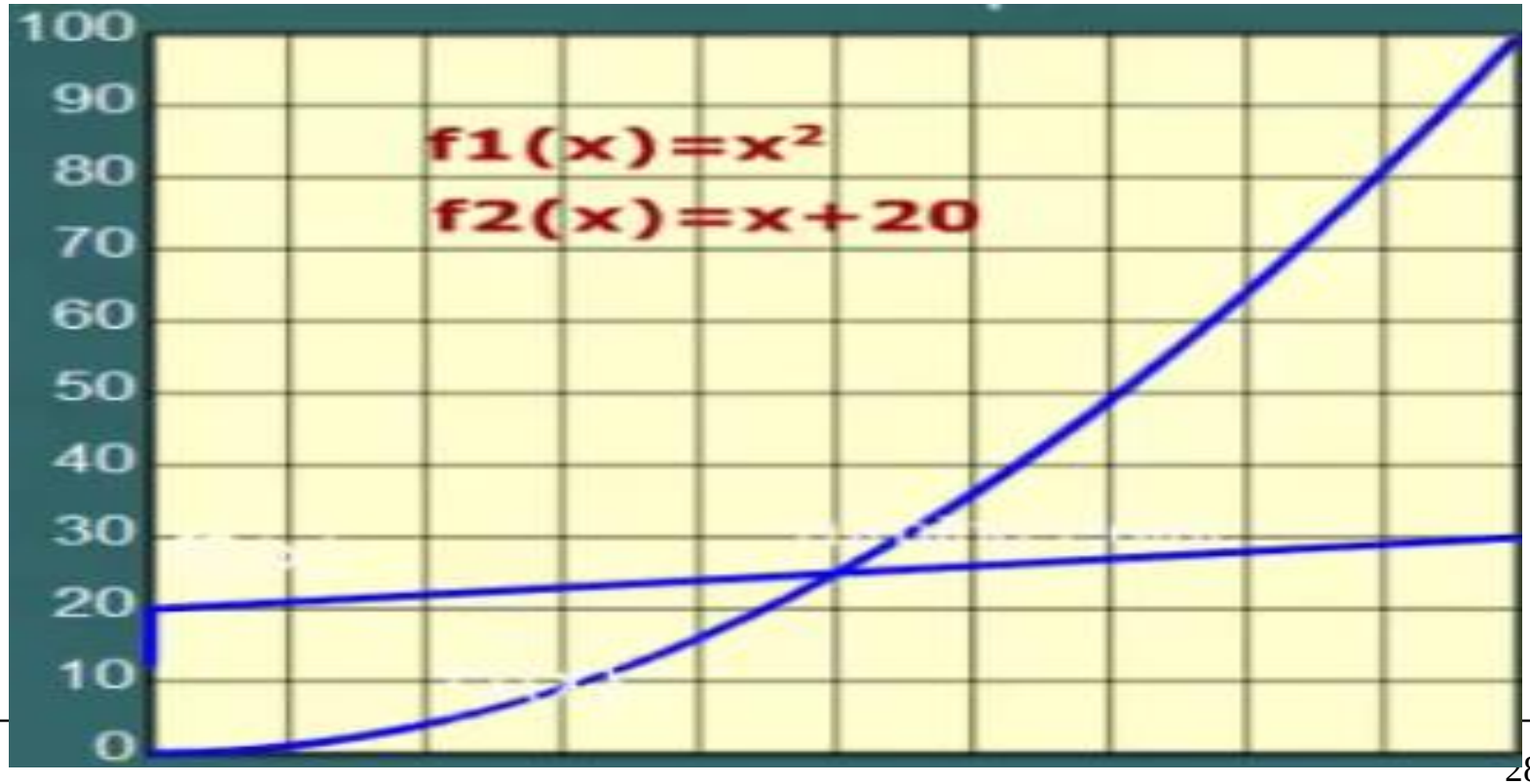
Order of Growth

- To capture the order of growth of an algorithm's running time, we will associate, with each algorithm, a function $f(n)$ that characterizes the number of primitive operations that are performed as a function of the input size n.
- Often we need to know how running time varies with the variation of input size.
- Measuring the performance of an algorithm in relation with the input size n is called **order of growth**
- Behavior of an algorithm for large input is called asymptotic growth

- If running time of a function grows as the size of input size N increases, we say that the algorithm asymptotically varies as N

Order of Growth

- Figure below shows the comparative behavior of two algorithms.
- Observe that for input size less than 5 second, algorithms have better performance.
- On the other hand the second algorithm shows much better performance for large values of inputs

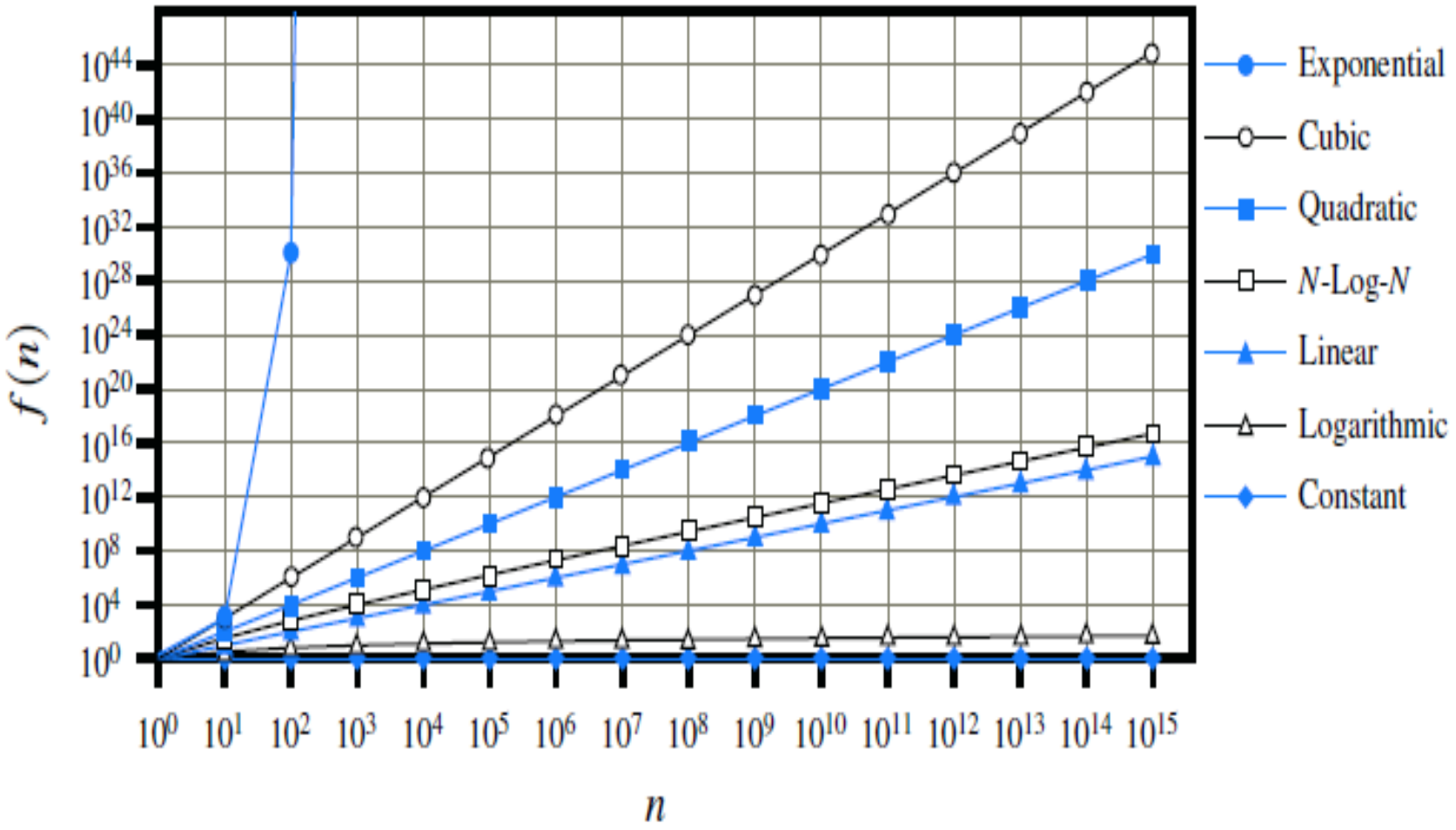


Comparing Growth Rates or Order of Growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- What distinguishes an efficient algorithm from inefficient algorithm is when we use larger input size that the difference in algorithm efficiencies becomes both clear and important

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Comparing Growth Rates or Order of Growth



Asymptotic Analysis

- Knowing the complexity of algorithms allows you to answer questions such as
  - How long will a program run on an input?
  - How much space will it take?
  - Is the problem solvable?
- An understanding of algorithmic complexity provides programmers with insight into the efficiency of their code. Complexity is also important to several theoretical areas in computer science, including algorithms, data structures, and complexity theory.

Asymptotic Analysis

- **Asymptotic analysis** refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is computed as **f(n)** and may be for another operation it is computed as **g(n^2).**

- Which means first operation's running time will increase linearly with the increase in **n** and running time of second operation will increase exponentially when **n** increases. Similarly the running time of both operations will be nearly same if **n** is significantly small.

Asymptotic Analysis

- Usually, time required by an algorithm falls under three types −
  - **Best Case time complexity** − Minimum time required for program execution.
  - **Average Case time complexity** − Average time required for program execution.
  - **Worst Case time complexity** − Maximum time required for program execution.

Running Time Analysis

- Generally we perform analysis to find maximum, average and minimum running times
- **Best case Analysis**
  - Best case analysis is used to determine the minimum running time of an algorithm.
  - Execution time depends on the nature of input data.
  - In Searching problems, for example, the most favorable input is the one which matches with the very first item in a data collection.
  - Likewise, in sorting problems, the best scenario would be when items are already in sorted order.
  - The best case analysis has little practical value.

Running Time Analysis

- **Worst Case Analysis**
  - Worst case analysis determines the running time for most unfavorable input.
  - It provides maximum running time for a given input.
  - In searching, for example, when an item matching with the search key happens to occur at the end of a data collection.
  - Thus worst case analysis yields the most pessimistic time estimate.
  - It, however, guarantees that in all circumstances, running time would never exceed the estimated time.
  - In almost all applications, algorithm analysis is done on a worst-case basis.

Running Time Analysis

- **Average Case Analysis**
  - Average case analysis is done to find out the average running time for an algorithm.
  - It lies somewhere between the optimistic and pessimistic times.
  - In searching problem, for example the average running time would be when a data item matching with the search key lies in the middle of a data collection.
  - The average case analysis often relies on probability theory. In most cases it is difficult to determine average running time of an algorithm.

Space Analysis

- Space analysis is concerned with determining the maximum storage requirement (memory) needed to implement a particular algorithm.
- It provides an estimate of number of bytes as function of input size
- Variables and constants occupy fixed storage, which does not depend on input size.
  - Therefore, the space utilization for constants and variables is O(1).
- For complex data structures, space utilization is estimated by counting the number of bytes.
- For example, an array of integers of size N, would require 4N bytes.
  - The space complexity would be O(N)
- a large data file can be stored more efficiency in shorter time by using large memory space.
  - For example, data of two arrays can be merged more efficiently by using a third array

Space complexity

- Space needed is sum of two components
- Fixed Part
  - Independent of input and output characteristics-
  - Typically include
    - Instruction
    - Space for variables
    - Space for constants and so on
- Variable Part
  - Dependent on particular problem instance
  - Typically include
    - Space needed by referenced variables
    - Recursion stack space and so on

Space complexity Calculation

- Let P be the algorithm
- Let S(P) be the space requirement for algorithm
- Then
  - S(P) = C + Sp
- Where
  - C is the fixed space
  - Sp is the variable space

- When analyzing space complexity of an algorithm we concentrate solely on estimating Sp (Variable space)

Space complexity Example

- Consider the following Algorithm

```
Algorithm Sum(a,n)
{
   S=0;
   For i=1 to n
      s=s+a[i];
   Return s;
}
```

- Space needed for the algorithm is as follows
  - Size variable ‘n’ – 1 word
  - Array ‘a’ values - n words
  - Loop variable ‘i’ - 1 word
  - Sum variable ‘s’ – 1 word
  - Total Size = (n+3) words

Asymptotic Notations

- To choose the best algorithm, we need to check efficiency of each algorithm.
- The efficiency can be measured by computing time complexity of each algorithm.
- A asymptotic notation is a shorthand way to represent the time complexity.
  - Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.
    - O Notation
    - Ω Notation
    - θ Notation

The "Big-Oh" Notation

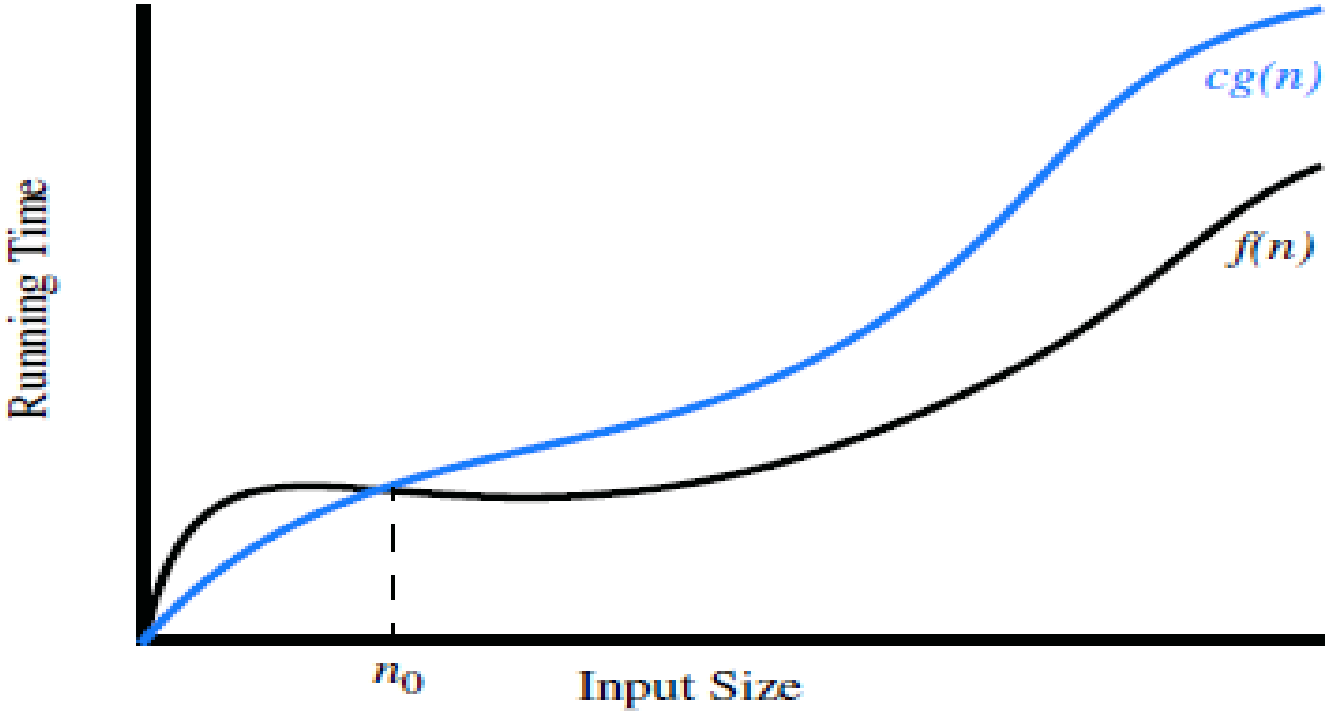
- The **O(n)** is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

- **Definition** :Let $f(n)$ and $g(n)$ be functions, where n is a positive integer. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for } n \geq n_0.$$

- This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is ***big-Oh*** of $g(n)$."

The "Big-Oh" Notation



- Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

The "Big-Oh" Notation

Examples 1: Show $3n^2 + 4n - 2 = O(n^2)$.

We need to find c and n_0 such that:
$3n^2 + 4n - 2 <= cn^2$ for all $n >= n_0$.

Divide both sides by n^2, getting:
$3 + 4/n - 2/n^2 <= c$ for all $n >= n_0$.

If we choose n_0 equal to 1, then we need a value of c such that:
$3 + 4 - 2 <= c$

We can set c equal to 5. Now we have:
$3n^2 + 4n - 2 <= 5n^2$ for all $n >= 1$.

The "Big-Oh" Notation

Examples 2: Show n^3 != $O(n^{2)}$.

Let's assume to the contrary that

n^3 != $O(n^2)$.

Then there must exist constants c and n_0 such that

n^3 != cn^2. for all n >= n_0.

Dividing by n^2, we get:

n <= c for all n >= n_0.

But this is not possible; we can never choose a constant c large enough that n will never exceed it, since n can grow without bound.

Thus, the original assumption, that n^3 = $O(n^2)$, must be wrong so n^3 != $O(n^2)$.

Some Properties of the Big-Oh Notation

- It provides a convenient method for expressing asymptotic behavior of an algorithm
- When input size becomes large, we simplify Big-Oh representation.
  - Any constants in an expression may be ignored.
    - $O(N^2 + 100) = O(N^2)$
  - Lower order terms may be dropped in favor of the highest order term in an expression
    - $O(N^3 + 2N^2 + N) = O(N^3)$
  - The multiplication constants may be ignored.
    - $O(20N^3) = O(N^3)$

More Examples

- Let us consider some further examples here, focusing on combinations of the seven fundamental functions used in algorithm design. We rely on the mathematical fact that $\log n \leq n$ for $n \geq 1$.

Example 4.9: $5n^2 + 3n\log n + 2n + 5$ *is* $O(n^2)$.

Justification: $5n^2 + 3n\log n + 2n + 5 \leq (5+3+2+5)n^2 = cn^2$, for $c = 15$, when $n \geq n_0 = 1$.

Example 4.10: $20n^3 + 10n\log n + 5$ *is* $O(n^3)$.

Justification: $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$.

Example 4.11: $3\log n + 2$ *is* $O(\log n)$.

Justification: $3\log n + 2 \leq 5\log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n_0 = 2$ in this case.

Example 4.12: 2^{n+2} *is* $O(2^n)$.

Justification: $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case.

Example 4.13: $2n + 100\log n$ *is* $O(n)$.

Justification: $2n + 100\log n \leq 102n$, for $n \geq n_0 = 1$; hence, we can take $c = 102$ in this case.

Big-Omega Notation, Ω

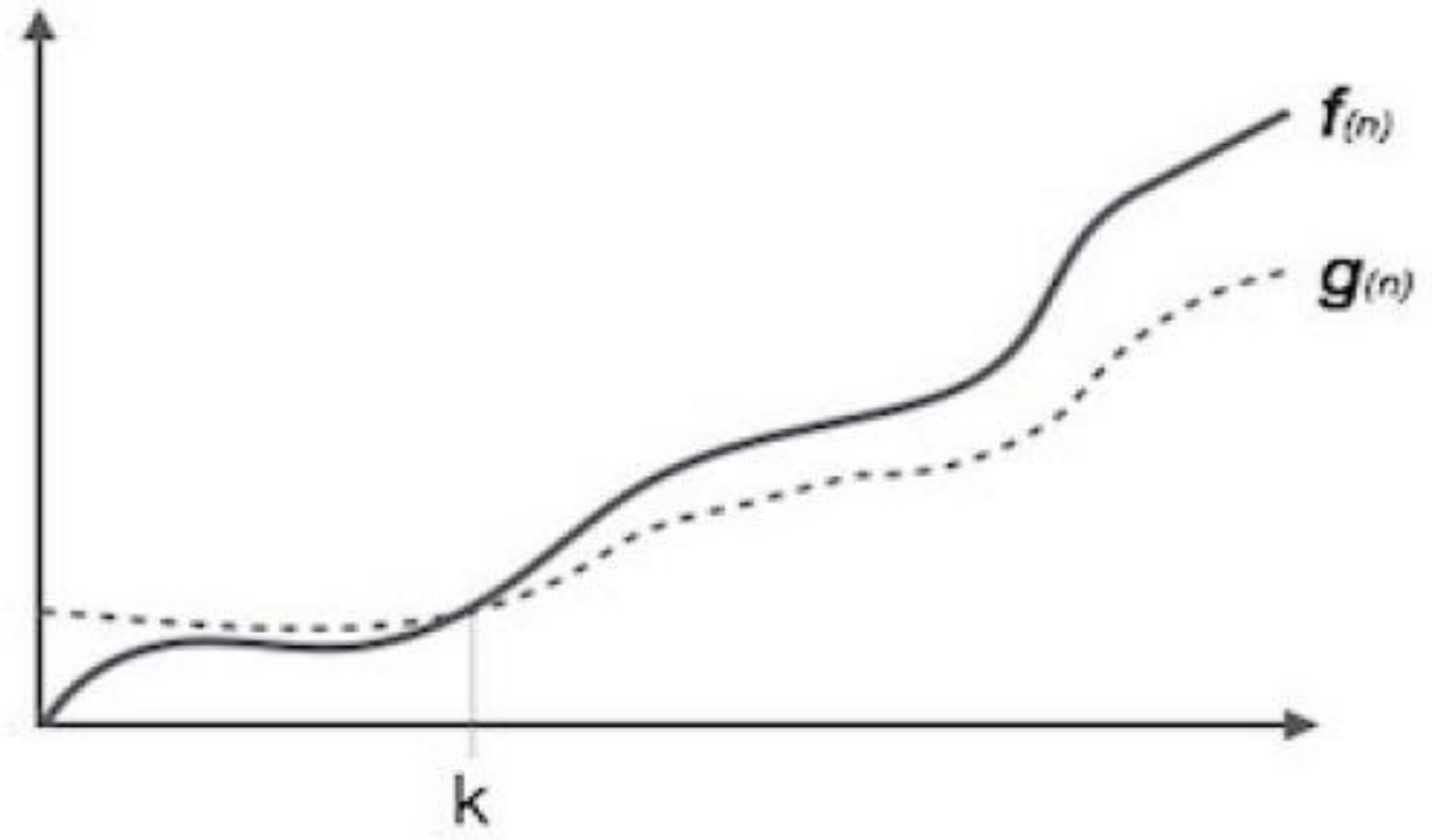
- The Ωn is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.

- Definition: Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $\Omega(g(n))$, pronounced "$f(n)$ is big-Omega of $g(n)$," if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq cg(n), \text{ for } n \geq n0.$$

- This definition allows us to say asymptotically that one function is greater than or equal to another, up to a constant factor.

Big-Omega Notation, Ω



Example : $3n\log n - 2n$ is $\Omega(n\log n)$.

Justification: $3n\log n - 2n = n\log n + 2n(\log n - 1) \geq n\log n$ for $n \geq 2$; hence,

we can take $c = 1$ and $n_0 = 2$ in this case.

Big-Theta Notation, θ

- In addition, there is a notation that allows us to say that two functions grow at the same rate, up to constant factors.

Definition: We say that $f(n)$ is $\theta(g(n))$, pronounced "$f(n)$ is big-Theta of $g(n)$," if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$, and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n), \text{ for } n \geq n_0.$$

- Example : $3n\log n+4n+5\log n$ is $\theta(n\log n)$.
- Justification: $3n\log n \leq 3n\log n+4n+5\log n \leq (3+4+5)n\log n$ for $n \geq 2$.

Big-Theta Notation, θ

- The θn is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following -