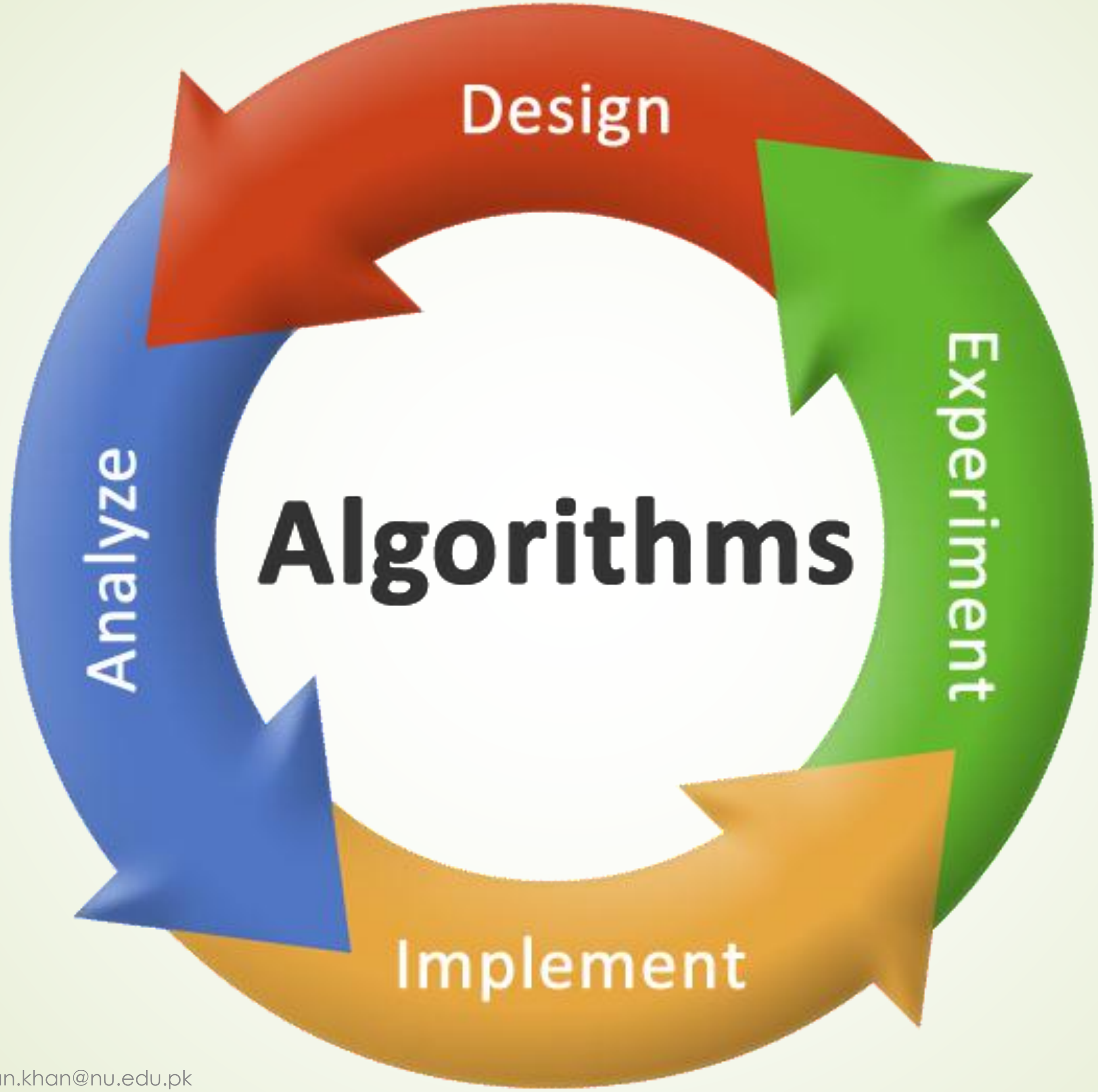





zeshan.khan@nu.edu.pk

# Algorithm

- Finite
- Set
- Unambiguous
- Instructions
- Specific task

# Why do we study algorithms?

# Business

- Google Business Growth
  - 25,762 Million USD in 2017-2.
  - 5,186 Million USD in 2008-1.

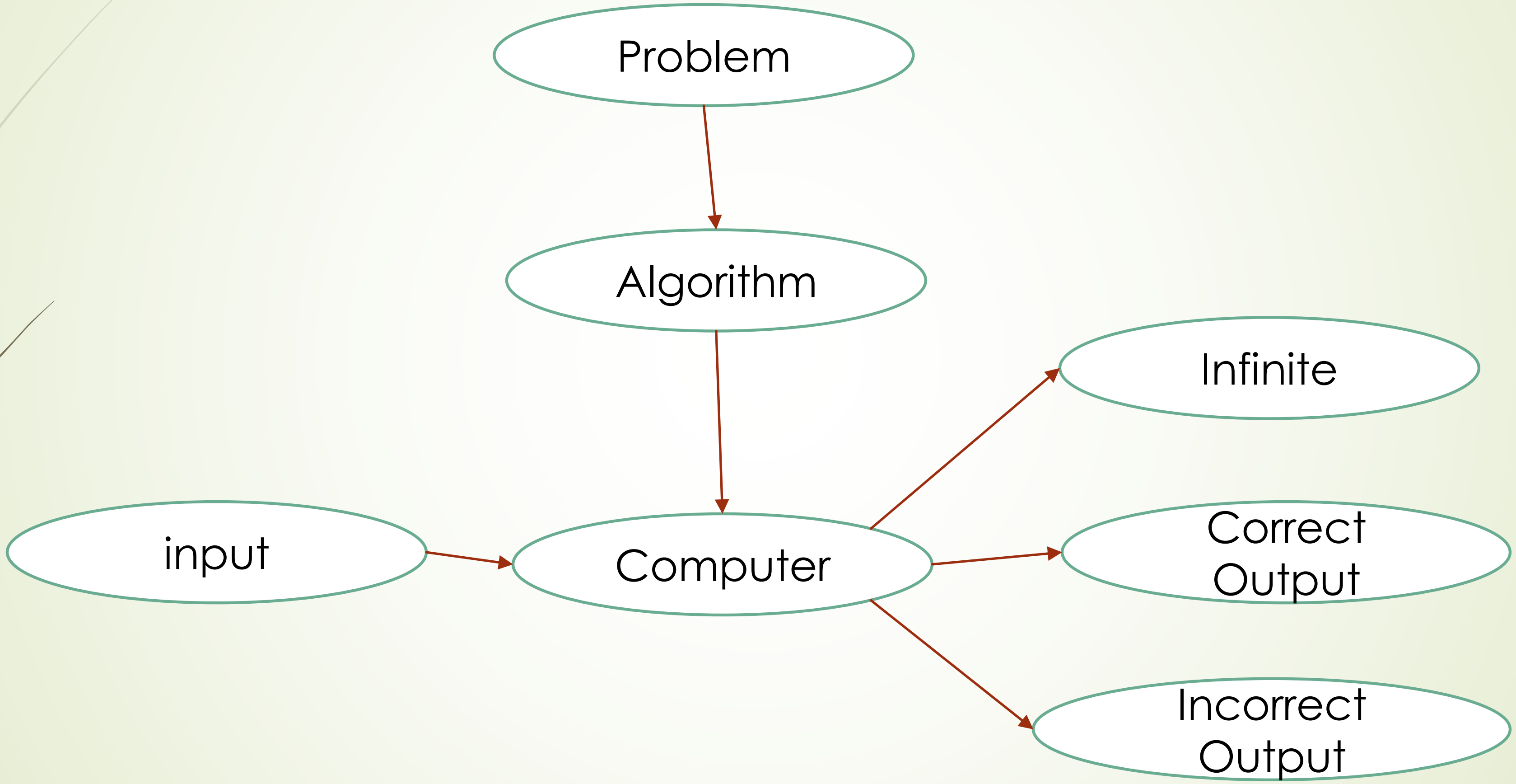
# Issues

- General Pervez Musharraf
  - There is a date and a country
- Prime Minister Nawaz Sharif
  - There is a date and a country
- All sonar animals not bat nor dolphin
  - Bat and dolphin must excluded
- American president not Trump
  - Trump must excluded

# Why do we study algorithms?

- To develop a computer based solution of a problem
- To know a standard set of important algorithms from different areas of computing
- To design new algorithms and analyze their efficiency
- To develop analytical skills

# Correct Algorithm???



# Correct Algorithm

- An algorithm is said to be correct if given input as described in the input specifications:
  - the algorithm terminates in a finite time
  - on termination the algorithm returns output as described in the output specifications

# Algorithm?

- A name
- Finite Input
- Finite Output

# Algorithm?

*Algorithm SumOfSquares*
*INPUT: a; b; where a and b are integers*
*OUTPUT: c; where c is a sum of the squares of input numbers.*
*start;*
    *c := a*a + b*b;*
    *return c;*
*end;*

- The name of this algorithm is *SumOfSquares*. Its input and output are integer sequences of length 2 and 1, respectively.

# Algorithm

- At least three important questions need to be answered for each algorithm
  - Is it correct?
  - **How much time does it take, as a function of n?**
  - And can we do better?

# Trivial Problem Solving

1. Understand the problem
2. Formulate a solution / algorithm
3. Analyze the algorithm
   - Design a program
   - Implement the program
   - Execute the code
   - Measure the time
4. See if the solution is ok
   - End The procedure
5. Otherwise go to step 1

# Algorithmic Problem Solving

1. Understand the problem
2. Formulate a solution / algorithm
   - Ascertaining the Capabilities of the Computational Device
   - Choosing between Exact and Approximate Problem Solving
   - Algorithm Design Techniques
   - Designing an Algorithm and Data Structures
3. Analyze the algorithm
   1. Methods of Specifying an Algorithm
   2. Proving an Algorithm's Correctness
4. See if the solution is ok
   - Coding an Algorithm
   - End The procedure
5. Else go to step 1

# Algorithmic Problem Solving

- **Trivial Approach**
- Analyze the algorithm
  - Design a program
  - Implement the program
  - Execute the code
  - Measure the time
- See if the solution is ok
  - End The procedure

- **Algorithmic Approach**
- Analyze the algorithm
  - Methods of Specifying an Algorithm
  - Proving an Algorithm's Correctness
- See if the solution is ok
  - Coding an Algorithm
  - End The procedure

zeshan.khan@nu.edu.pk

# Some Problems

- Knapsack problem
- Graph Visits Problems (Longest / Hamiltonian)
- Travelling salesman problem
- Subgraph isomorphism problem
- Vertex cover problem
- Graph coloring problem

# Formulas

- $\sum_{i=1}^{n} 1 = 1 + 1 + 1 + \cdots + 1 = n$
- $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \cong (\frac{1}{2})n^2$
- $\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \cong (\frac{1}{3})n^3$
- $\sum_{i=1}^{n} i^k = 1^k + 2^k + 3^k + \cdots + n^k \cong (\frac{1}{k+1})n^{k+1}$
- $\sum_{i=1}^{n} a^i = a^1 + a^2 + a^3 + \cdots + a^n = \frac{a^{n+1}-2}{a-1}$
- $\sum_{i=1}^{n} 2^i = 2^1 + 2^2 + 2^3 + \cdots + 2^n = 2^{n+1} - 2$
- $\sum_{i=1}^{n} i2^i = 1.2^1 + 2.2^2 + 3.2^3 + \cdots + n.2^n = (n-1)2^{n+1} - 2$
- $\sum_{i=1}^{n} 1/i = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \cong \ln n + 0.5772$
- $\sum_{i=1}^{n} \log i \cong n \log n$

# Formulas

- $\sum_{i=x}^{n} i = \sum_{i=1}^{n} i - \sum_{i=1}^{x-1} i$
- $\sum_{i=x}^{y} ca_i = c \sum_{i=x}^{y} a_i$
- $\sum_{i=x}^{y} (a_i \pm b_i) = \sum_{i=x}^{y} a_i \pm \sum_{i=x}^{y} b_i$
- $\sum_{i=x}^{y} (a_i - a_{i-1}) = \sum_{i=x}^{y} (a_y - a_{x-1})$

# Floor and Ceiling Formulas

- $x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$
- $\lfloor x + n \rfloor = \lfloor x \rfloor + n$
- $\lceil x + n \rceil = \lceil x \rceil + n$
- $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$
- $\lceil \log(n + 1) \rceil = \lfloor \log n \rfloor + 1$

# Modular Arithmetic

- Modular arithmetic (n, m are integers, p is a positive integer and % is for Mod)
- $(n + m)\,\%p = (n\,\%p + m\,\%p)\,\%p$
- $(nm)\,\%p = ((n\,\%p)(m\,\%p))\,\%p$

# Logarithm Formulas

- $\log_a 1 = 0$
- $\log_a a = 1$
- $\log_a x^y = y \log_a x$
- $\log_a xy = \log_a x + \log_a y$
- $\log_a x/y = \log_a x - \log_a y$

# Analysis of Algorithms

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size

input size

$$T(n) \approx c_{op} C(n)$$

running time

execution time
for basic operation

Number of times
basic operation
is executed

# Algorithm

- **How do we write algorithms?**
- **Pseudo Code:**
  - Similar construct / keywords as in a high level programming languages, e.g. in C, Pascal etc.
  - Structured semantics of the high level languages without caring about the syntactic errors / grammatical rules

# Sample Pseudo Code



```
Max-Subsequence-Sum(Array, N) //Where N is size of Array

{       int this-sum = 0, Max-sum = 0;
        for(int i = 0; i < N; i++)
        {   for(int j = i; j < N; j++)
            {
                this-sum = 0;
                for(int k = i; k <= j; k++)
                    this-sum = this-sum +  Array[k];
                if(this-sum > Max-sum)
                    Max-sum = this-sum;
            }
        }
            return(Max-sum);
}
```

# Analysis

- How much  time each construct / keyword of a pseudo code takes to execute.
- Assume it takes $t_i$ (the $i^{th}$ construct)
- Sum / Add up the execution time of all the constructs / keywords.
- if there are m constructs then total time for all the constructs is the function of the input size T(n)
- $T_n = \sum_{i=1}^{m} t_i$

# Analysis

- What are the constructs / Keywords.
- Time for each construct
- Total Time
- Total time as a function of input size

# Constructs

- Constructs:
  - Sequence
  - Selection
  - Iterations
  - Recursion

# Constructs

- **Sequence Statements**
  - Just add the running time of the statements (int x, x=5, x++, y-=x etc.)
- **Selection**
  - Add the time of maximum conditional code (if else, switch).
- **Iteration** is at most the running time of the statements inside the loop (including tests) times the number of iterations.

- **If-Then-Else:**
  - if (condition) $S_1$ else $S_2$
  - Running time of the test plus the larger of the running times of $S_1$ and $S_2$.
- **Nested Loops:** Analyze these inside out. The total Running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the size of all the loops.
- **Function Calls:** Analyzing from inside to out. If there are function calls, these must be analyzed first.

# Example 1

- Analyze Time for the following algorithm
  - Find the value of the largest element in a list of n numbers.

```
MaxElement(A[0..n-1])
  maxVal = A[0];
  for(i = 1; i < n; i++)
   if(A[i] > maxVal)
       maxVal = A[i];
  return maxVal
```

# Example 2

- Check whether all the elements in a given array are distinct.

```
UniqueElements(A[0..n-1])
    for(i= 0; i<n-1; i++)
     for(j = i+1; j<n; j++)
          if(A[i] = A[j])
          return false
    return true
```

# Analysis Example

```
Max-Subsequence-Sum(Array, N)      //Where N is size of Array
{
        int this-sum = 0, Max-sum = 0;
        for(int i = 0; i < N; i++)
        {   for(int j = i; j < N; j++)
            {
                this-sum = 0;
                for(int k = i; k <= j; k++)
                    this-sum = this-sum +  Array[k];
                if(this-sum > Max-sum)
                    Max-sum = this-sum;
            }
        }
            return(Max-sum);
}
```

# Analysis Example

```
bubbleSort()
{
        temp;
        for (long i = 0; i < length; i++)
        {
                for (long j = 0; j < length - i - 1; j++)
                {
                        if (list[j] > list[j + 1])
                        {
                                temp = list[j];
                                list[j] = list[j + 1];
                                list[j + 1] = temp;
                        }
                }
        }
}
```

# Analysis Example

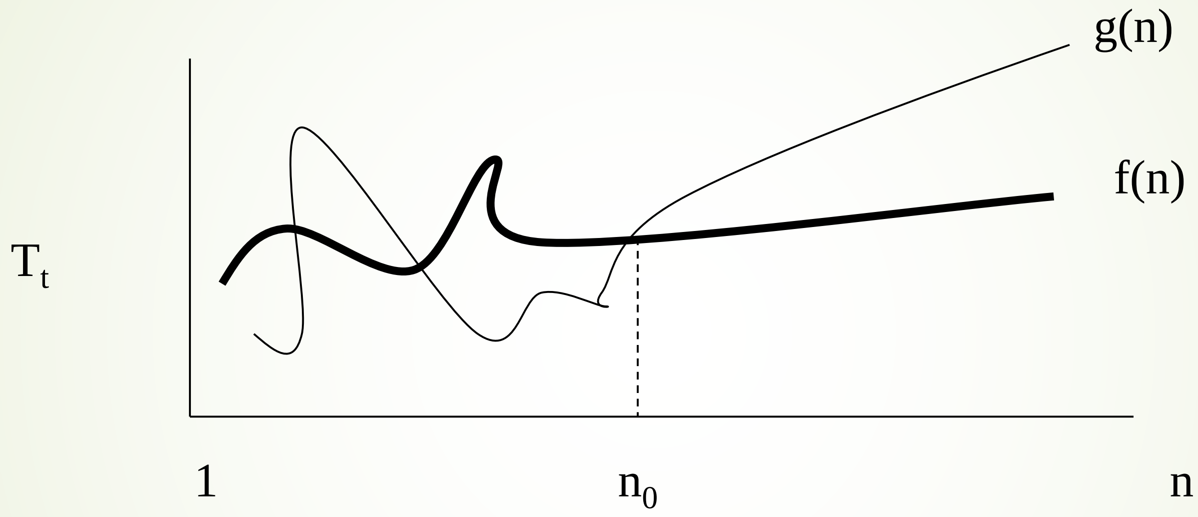
```
Max_Diff(S) {
    x=max(S)
    y=min(S)
    return x,y;
}
```

# Analysis Example

```
palindrom(s,x) {
    for(i=x,j=x;i>=1;i--,j++)
     if(s[i]!=s[j])
         return false;
    return true;
}
```

# Order Notation

There may be a situation, e.g.




$f(n) <= g(n)$ for all $n >= n_0$ Or

$f(n) <= cg(n)$ for all $n >= n_0$ and $c = 1$

g(n) is an **asymptotic upper bound** on f(n).

$f(n) = O(g(n))$ iff there exist two positive constants c and $n_0$ such that

$f(n) <= cg(n)$ for all $n >= n_0$

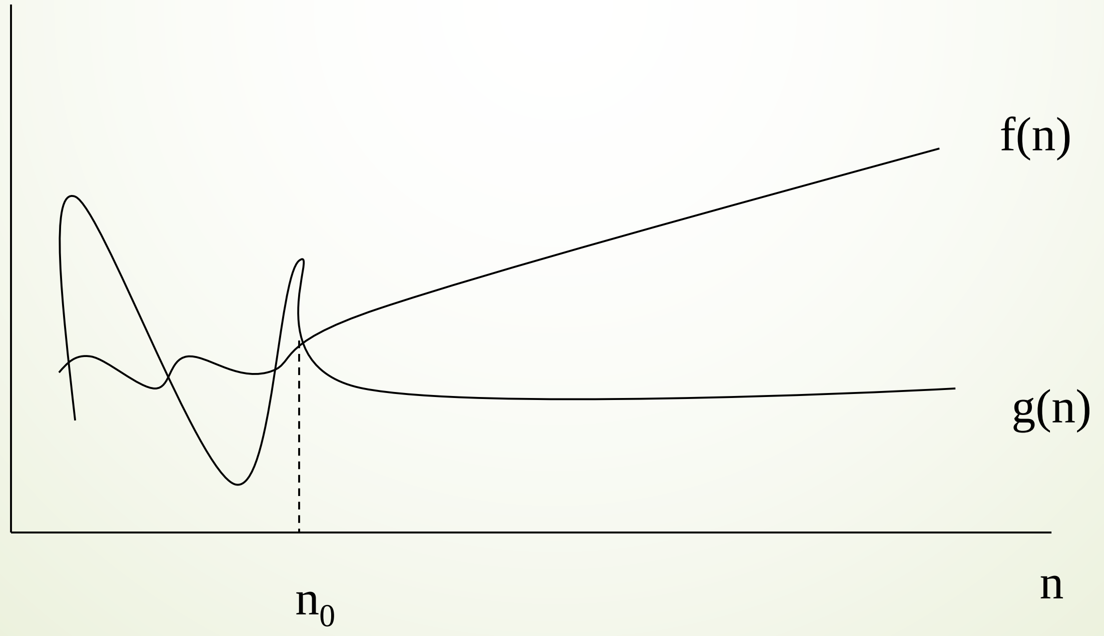
# Order Notation



**Asymptotic Lower Bound:** $f(n) = \Omega(g(n))$,

iff there exit positive constants c and $n_0$ such that

$f(n) >= cg(n)$ for all $n >= n_0$

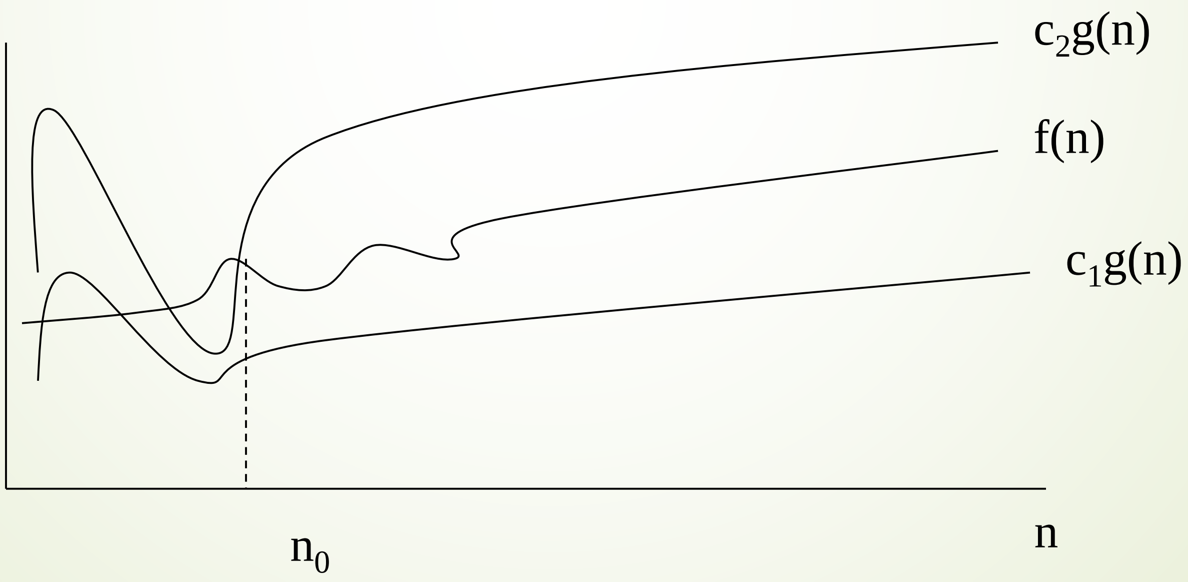


# Order Notation

**Asymptotically Tight Bound:** $f(n) = \theta(g(n))$,

iff there exit positive constants $c_1$ and $c_2$ and $n_0$ such that

$c_1\, g(n) <= f(n) <= c_2 g(n)$ for all $n >= n_0$




This means that the best and worst case requires the same amount of time to within a constant factor.

# Some Rules About Asymptotic Notation



1. If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$
   Then $T_1(n) + T_2(n) = Max(O(f(n)), O(g(n)))$
   $T_1(n) * T_2(n) = O(f(n) * g(n))$
2. If $T(x)$ is a polynomial of degree n, then
   $T(x) = \theta(x^n)$
3. $\log^k(n) = O(n)$ for any constant k. This tells that logarithms grow very slowly.
4. Do not include any constants or low order terms inside a big-Oh, e.g.,
   $T(n) = O(2n^2)$ -------- wrong
   $T(n) = O(n^2 + n)$ ----- wrong

# Order Notation



**Example:** **show that $(1/2)n^2 - 3n = \theta(n^2)$**

- To do so we must determine positive constants $c_1$, $c_2$ and $n_0$ such that $c_1n^2 <= (1/2)n^2 - 3n <= c_2n^2$, $n >= n_0$
- Dividing by $n^2$ $c_1 <= ½ - 3/n <= c_2$
- Right Hand Inequality $½ <= c_2 + 3/n$
- For positive n, if $c_2 >= ½$ then the inequality holds.
- Left Hand Inequality $c_1 + 3/n <= ½$
- For $n = 7$ and $c_1 <= 1/14$, the inequality holds.
- $c_1 <= 1/14$, $c_2 >= ½$ and $n = 7$

How to compare the efficiency of two algorithms?

# Standard Functions



| n | lgn | nlgn | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|
| 0 | | | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 2 |
| 2 | 1 | 2 | 4 | 8 | 4 |
| 4 | 2 | 8 | 16 | 64 | 16 |
| 8 | 3 | 24 | 64 | 512 | 256 |
| 16 | 4 | 64 | 256 | 4096 | 65536 |
| 32 | 5 | 160 | 1024 | 32768 | 4294967296 |
| 64 | 6 | 384 | 4096 | 262144 | 1.84467E+19 |
| 128 | 7 | 896 | 16384 | 2097152 | 3.40282E+38 |
| 256 | 8 | 2048 | 65536 | 16777216 | 1.15792E+77 |
| 512 | 9 | 4608 | 262144 | 134217728 | 1.3408E+154 |
| 1024 | 10 | 10240 | 1048576 | 1073741824 | |
| 2048 | 11 | 22528 | 4194304 | 8589934592 | |

# Execution Times



Execution time for algorithms with the given time complexities (one time constant is 1 nano seconds)

| n | f(n) = lgn | f(n) = n | f(n) = nlgn | f(n) = $n^2$ | f(n) = $n^3$ | f(n) = $2^n$ |
|---|---|---|---|---|---|---|
| 1 0 | 0.003 micro sec | 0.01 micro sec | 0.033 micro sec | 0.1 micro sec | 1 micro sec | 1 micro sec |
| 2 0 | 0.004 micro sec | 0.02 micro sec | 0.086 micro sec | 0.4micro sec | 8 micro sec | 1 milli sec |
| 3 0 | 0.005 micro sec | 0.03 micro sec | 0.147 micro sec | 0.9 micro sec | 27micro sec | 1 sec |
| 4 0 | 0.005 micro sec | 0.04 micro sec | 0.213 micro sec | 1.6 micro sec | 64 micro sec | 18.3 min |
| 5 0 | 0.006 micro sec | 0.05 micro sec | 0.282 micro sec | 2.5 micro sec | 125 micro sec | 13 days |
| $10^2$ | 0.007 micro sec | 0.10 micro sec | 0.664 micro sec | 10 micro sec | 1 milli sec | 4 exp 13 years |
| $10^3$ | 0.010 micro sec | 1.00 micro sec | 9.966 micro sec | 1 milli sec | 1 sec | |
| $10^4$ | 0.013 micro sec | 10 micro sec | 130 micro sec | 100 milli sec | 16.7 min | |
| $10^5$ | 0.017 micro sec | 0.10 milli sec | 1.67 milli sec | 10 s | 11.6 days | |
| $10^6$ | 0.020 micro sec | 1 milli sec | 19.93 milli sec | 16.7 min | 31.7 years | |
| $10^7$ | 0.023 micro sec | 0.01sec | 0.23 sec | 1.16 days | 31709 years | |
| $10^8$ | 0.027 micro sec | 0.10 sec | 2.66 sec | 115.7 days | 3.17 exp 7 years | |
| $10^9$ | 0.030 micro sec | 1 sec | 29.90 sec | 31.7 years | | |

# Using Limits for Comparing Orders of Growth

- A much more convenient method
- computing the limit of the ratio of two functions

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n),\\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n),\\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^{3}\end{cases}$$

- *the last two mean that t (n) ∈ (g(n)),*
- *and the second case means that t (n) ∈ (g(n)).*

$\frac{1}{2}n(n-1)$ and $n^2$

- Compare the orders of growth of

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

- Since the limit is equal to a positive constant, the functions have the same order of growth

# Basic asymptotic efficiency classes

| Class | Name | Comments |
|---|---|---|
| 1 | *constant* | 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. |
| $\log n$ | *logarithmic* | 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. |
| $n$ | *linear* | Algorithms that scan a list of size $n$ (e.g., sequential search) belong to this class. |

# Basic asymptotic efficiency classes

| | | |
|---|---|---|
| $n \log n$ | *linearithmic* | Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category. |
| $n^2$ | *quadratic* | 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. |
| $n^3$ | *cubic* | Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class. |

# Basic asymptotic efficiency classes

| | | |
|---|---|---|
| $2^n$ | *exponential* | 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. |
| $n!$ | *factorial* | Typical for algorithms that generate all permutations of an $n$-element set. |