
# Algorithms Analysis and Design

## Chapter 2

## Analysis of Algorithms – Part 2

# Analysis of recursive algorithms

# Example 1

| Function RecFunc(n): | cost | time |
|---|---|---|
| // Base case | | |
| if n <= 0 | c1 | 1 |
| return | c2 | 1 x t |
| | | |
| RecFunc(n - 1) | T(n-1) | 1 x (1-t) |
| | | |
| for i = 1 to n | c4 | (n + 1) x (1-t) |
| print(i) | c5 | n x (1-t) |

If t=1 (n<=0) : T(n) = c1 + c2 = C [constant]

If t=0 (n>0): T(n) = c1 + T(n-1) + c3 (n+1) + c5(n)

= T(n-1) + an + b [a and b are constants]

$$\mathbf{T}(\boldsymbol{n}) = \begin{cases} \boldsymbol{C} & , \quad \boldsymbol{n} <= \mathbf{0} \\ \boldsymbol{T(n-1)} + \boldsymbol{n}, & \boldsymbol{n} > \mathbf{0} \end{cases}$$

Remember that we **care about the worst-case** so that we will **take the maximum possible runtime**.

T(n) = T(n-1) + n n>0

# Example 1

```
Function RecFunc(n):                                cost          time
    // Base case
    if n <= 0
        return

    RecFunc(n - 1)
    for i = 1 to n
        print(i)
```

# Example 1

- Compute the factorial function F (n) = n! for an arbitrary nonnegative integer n

  n!= n . (n-1) . (n-2) . ..... . 2 . 1= n . (n − 1)!  for n ≥ 1

  and 0!= 1  (base case)

  we can compute Fact (n) = n. Fact (n − 1) with the following recursive algorithm.

```
ALGORITHM Fact ( n )
{                                          cost          time
 1   If (n == 0)                           c1            1
 2     return 1                            c2            t
       Else
 3    return n * Fact (n – 1)              T(n-1) + c3   (1-t)
}
```

To compute Fact(n-1)

To multiply Fact(n-1) by n

If t=1 (n=0) : T(n) = c1 + c2 = d [constant]
If t=0 (n>0): T(n) = c1 + T(n-1) + c3
= T(n-1) + C where C is constant

$$\mathbf{T}(\boldsymbol{n}) = \begin{cases} \boldsymbol{d}, & \boldsymbol{n} = \mathbf{0} \\ \boldsymbol{T}(\boldsymbol{n}-\mathbf{1}) + \boldsymbol{c}, & \boldsymbol{n} > \mathbf{0} \end{cases}$$

Remember that we **care about the worst-case** so that we will **take the maximum possible runtime**.

T(n) = T(n-1) + c  n>0

Recurrence relation

# Example 1

- Compute the factorial function F (n) = n! for an arbitrary nonnegative integer n

  n!= n . (n-1) . (n-2) . ..... . 2 . 1= n . (n − 1)!  for n ≥ 1

  and 0!= 1  (base case)

  we can compute Fact (n) = n. Fact (n − 1) with the following recursive algorithm.

```
ALGORITHM Fact ( n )
{                                          cost        time
 1   If (n == 0)
 2     return 1
     Else
 3    return n * Fact (n – 1)
}
```

# Example 2

- Fibonacci Numbers Sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

```
ALGORITHM Fib ( n )
{                                          cost                    time
 1   If (n == 1 || n==2)                   c1                      1
 2     return 1                            c2                      t
       Else
 3    return Fib (n – 1) + Fib (n-2)       T(n-1) + T(n-2) + c3    (1-t)
}
```

If t=1 (n=1or 2) : T(n) = c1 + c2 = d [constant]
If t=0 (n>2): T(n) = c1 + T(n-1) + T(n-2) + c3
= T(n-1) + T(n-2) + C where C is constant

$$\mathbf{T}(\boldsymbol{n}) = \begin{cases} \boldsymbol{d}, & \boldsymbol{n} = \mathbf{1}, \mathbf{2} \\ \boldsymbol{T}(\boldsymbol{n}-\mathbf{1}) + \boldsymbol{T}(\boldsymbol{n}-\mathbf{2}) + \boldsymbol{c}, & \boldsymbol{n} > \mathbf{2} \end{cases}$$

T(n) = T(n-1) + T(n-2) + c n>2

Recurrence relation

# Example 2

- Fibonacci Numbers Sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

```
ALGORITHM Fib ( n )
{                                          cost          time
 1   If (n == 1 || n==2)
 2     return 1
       Else
 3    return Fib (n – 1) + Fib (n-2)
}
```

# Example 3

- **Multiply is a recursive function to perform multiplication of two positive integers (*m* and *n*)**

6 x 1 = 6
6 x 2 = 6 + (6 x 1)
6 x 3 = 6 + (6 x 2) = 6 + [6 + (6 x 1)]

m * n = m + m * (n-1)  base case is m * 1 = m

```
multiply (m , n )
{                                          cost        time
  if ( n == 1  )
    return m
  else
    return m + multiply (m , n – 1);
}
```

$$\mathbf{T}(\boldsymbol{n}) = \begin{cases} \boldsymbol{d}, & \boldsymbol{n} = \mathbf{1} \\ \boldsymbol{T}(\boldsymbol{n}-\mathbf{1}) + \boldsymbol{c}, & \boldsymbol{n} > \mathbf{1} \end{cases}$$

T(n) = T(n-1) + c  n>1

Recurrence relation

# Example 4

- Derive the recurrence relation for the following algorithm. **Assume T(1) = 1**

**Algorithm** ***A*** **(*n*)**

```
{                                   cost        time
1       A(n-1)                      T(n-1)      1
2       for i=1 to n                c2          n+1
3           statement               c3          n
}
```

$$T(n) = T(n-1) + c_2(n+1) + c_3 n$$

$$T(n) = T(n-1) + cn + d$$

**Recurrence relation**

Note: For Algorithm A to function properly, it is crucial to include a defined stopping point.

# Example 5

- **Recursive binary search algorithm**

```
Algorithm BinarySearch (A , start , end , key )
{
    if (start > end)
        return -1
    else
        mid = (start + end)/2
        if key == A[mid]
            return mid
        else if key < A[mid]
            return BinarySearch (A , start, mid-1 , key)
        else
            return BinarySearch (A, mid+1, end, key
}
```

$$T(n) = \begin{cases} d, & n = 1 \\ T\left(\frac{n}{2}\right) + c, & n > 1 \end{cases}$$

$T(n) = T(\frac{n}{2}) + c \qquad n>1$

- Cost of solving one subproblem of size n/2
- Cost of dividing the problem

# Example 5

- **Recursive binary search algorithm**

```
Algorithm BinarySearch (A , start , end , key )
{
    if (start > end)
        return -1
    else
        mid = (start + end)/2
        if key == A[mid]
            return mid
        else if key < A[mid]
            return BinarySearch (A , start, mid-1 , key)
        else
            return BinarySearch (A, mid+1, end, key
}
```

# Exercise

Consider the following recurrence relation:

$$T(n) = 2T(\frac{n}{2}) + n^2$$

Write a pseudocode for an algorithm that solves a problem with the given recurrence relation.

$ALGORITHM\ MyAlg\ (A,\ n)$
{

}

# Solving recurrence equations

# Recurrence relation

- A ***recurrence relation*** is an equation that recursively defines a sequence where the next term is a function of the previous terms.
  - Example: Fibonacci series Fib(n) = Fib(n-1) + Fib(n-2)
- When an algorithm contains a recursive call to itself, we can often describe its running time by a **recurrence equation** or **recurrence**, which **describes the overall running time on a problem of size *n* in terms of the running time on smaller inputs.**
- Recall the recursive algorithms that were explained in the lecture

| | Recurrence equation of running time |
|---|---|
| factorial(n) | $T(n) = T(n-1) + c \quad n>0$ |
| Fibonacci Sequence | $T(n) = T(n-1) + T(n-2) + c \quad n>2$ |
| Multiply(m , n) | $T(n) = T(n-1) + c \quad n>1$ |
| Binary search | $T(n) = T(\frac{n}{2}) + c \quad n>1$ |

# Solving Recurrence relation

- Solving recurrence relation means that we want to **convert the recursive definition to closed formula**.
- There are different techniques for solving recurrence:
  - **Substitution method:** Guess the Solution, and then Use the mathematical induction to find the boundary condition and shows that the guess is correct.
  - **Iteration method**: It means to **expand the recurrence and express it as a summation** of terms of n and initial condition.
  - **Recursion tree method**: A pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.
  - **Characteristic equation**
    - Master method

# Iteration method

## Example 1

- $T(n) = T(n-1) + c$ , $n>1$ $\quad\quad$ $T(0) = 1$ (base case)

**Repeat the procedure for k times**

**Then derive the general form**

| k | T(n) |
|---|---|
| 1 | T(n) = T(n-1) + c |
| 2 | = [ T(n-2) + c ] + c = T(n-2) + 2c |
| 3 | = [T(n-3) + c] + 2c = T(n-3) + 3c |
| 4 | = [T(n-4) + c] + 3c = T(n-4) + 4c |
| …… | …… |
| k | T(n) = T(n-k) + kc |

General form

$T(n-1) = T(n-1-1) + c = \mathbf{T(n-2) + c}$

$T(n-2) = T(n-2-1) + c = \mathbf{T(n-3) + c}$

$T(n-3) = T(n-3-1) + c = \mathbf{T(n-4) + c}$

$T(0) = 1$

$T(n-k) = 1 \quad \rightarrow \quad n - k = 0 \quad \rightarrow \quad \boldsymbol{n = k}$

Now substitute k = n: $T(n) = T(n - n) + n\,c$

$T(n) = T(0) + c\,n$

$= \mathbf{1 + cn}$

Order of growth is **O(n)**

# Example 2

- $T(n) = T(\frac{n}{2}) + c$ , $n>1$ $T(1) = 1$ (base case)

| k | T(n) |
|---|---|
| 1 | $T(n) = T(\frac{n}{2}) + c$ |
| 2 | $= [\ T(\frac{n}{4}) + c\ ] + c = T(\frac{n}{4}) + 2c$ |
| 3 | $= [T(\frac{n}{8}) + c] + 2c = T(\frac{n}{8}) + 3c$ |
| 4 | $= [T(\frac{n}{16}) + c] + 3c = T(\frac{n}{16}) + 4c$ |
| …… | …… |
| k | $T(n) = T(\frac{n}{2^k}) + kc$ |

General form

$T(\frac{n}{2}) = T(\frac{\frac{n}{2}}{2}) + c = \mathbf{T(\frac{n}{4}) + c}$

$T(\frac{n}{4}) = T(\frac{\frac{n}{4}}{2}) + c = \mathbf{T(\frac{n}{8}) + c}$

$T(\frac{n}{8}) = T(\frac{\frac{n}{8}}{2}) + c = \mathbf{T(\frac{n}{16}) + c}$

$T(1) = 1$

$T(\frac{n}{2^k}) = 1$

$\frac{n}{2^k} = 1 \rightarrow n = 2^k \rightarrow \boldsymbol{lgn = lg2^k}$

$lgn = k\ lg2 \rightarrow \boldsymbol{k = lgn}$

Now substitute $k = lgn$: $T(n) = T(1) + c\ lgn$

**$T(n) = 1 + c\ lgn$**

Order of growth is **O(lgn)**

# Iteration method

## Example 3 (Solution 1)

- T(n) = 2 T(n-1) + 1 , n>1 T(0) = 1 (base case)

| k | T(n) |
|---|---|
| 1 | T(n) = 2 T(n-1) + 1 |
| 2 | =2 [2T(n-2) + 1 ] + 1 = **4 T(n-2) + 2 + 1** |
| 3 | = 4 [2T(n-3) + 1] + 2 + 1 = **8 T(n-3) + 4 + 2 + 1** |
| 4 | = 8 [2T(n-4) + 1] + 4 + 2 + 1 = **16 T(n-4) + 8+4 + 2 + 1** |
| …… | …… |
| k | T(n) = $2^k$T(n-k) + [ 1 + 2 + 4 + 8 + …… $2^{(k-1)}$] |

T(n-1) = 2T (n-2) + 1

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

T(n-3) = ٢T (n-٤) + ١

1 + 2 + 4 + 8 + 16 + ….. $ar^{(n-1)}$

Is the summation of geometric sequence

$S_n = a\left(\frac{r^n - 1}{r-1}\right)$

a: the first term in the sequence

r: common ratio

T(0) = 1

T(n-k) = 1 → $n - k = 0$ → $\boldsymbol{n = k}$

Now substitute T(n) = $2^n$ T(n - n) + [$2^n$ - 1]

T(n) = $2^n$ T(0) + [$2^n$ - 1]

= $2^n$ + [$2^n$ - 1]

Order of growth is **O($2^n$ ) exponential**

# Iteration method

## Example 3 (Solution 2)

- T(n) = 2 T(n-1) + 1 , n>1 T(0) = 1 (base case)

| k | T(n) |
|---|---|
| 1 | T(n) = 2 T(n-1) + 1 |
| 2 | =2 [2T(n-2) + 1 ] + 1 = **4 T(n-2) + 3** |
| 3 | = 4 [2T(n-3) + 1] + 2 + 1 = **8 T(n-3) + 7** |
| 4 | = 8 [2T(n-4) + 1] + 4 + 2 + 1 = **16 T(n-4) + 15** |
| …….. | …….. |
| k | T(n) = $2^k$T(n-k) + $2^k$ - 1 |

T(n-1) = 2T (n-2) + 1

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

T(n-3) = ٢T (n-٤) + ١

T(0) = 1

T(n-k) = 1 → $n - k = 0$ → $\boldsymbol{n = k}$

Now substitute T(n) = $2^n$ T(n - n) + [$2^n$ - 1]

T(n) = $2^n$ T(0) + [$2^n$ - 1]

= $2^n$ + [$2^n$ - 1]

Order of growth is **O($2^n$ ) exponential**

# Example 4 (solution 1)

- $T(n) = T(\frac{n}{2}) + n$ , $n>1$ $\quad\quad$ $T(1) = 1$ (base case)

| k | T(n) |
|---|---|
| 1 | $T(n) = T(\frac{n}{2}) + n$ |
| 2 | $= [\, T(\frac{n}{4}) + \frac{n}{2}\,] + n = \mathbf{T(\frac{n}{4}) + \frac{n}{2} + n}$ |
| 3 | $= [T(\frac{n}{8}) + \frac{n}{4}] + \frac{n}{2} + n = \mathbf{T(\frac{n}{8}) + \frac{n}{4} + \frac{n}{2} + n}$ |
| 4 | $= [T(\frac{n}{16}) + \frac{n}{8}\,] + \frac{n}{4} + \frac{n}{2} + n = \mathbf{T(\frac{n}{16}) + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n}$ |
| …… | …… |
| k | $T(n) = T(\frac{n}{2^k}) + [\frac{n}{2^k} + \frac{n}{2^{k-1}} + \ldots\ldots \mathbf{\frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n}\,]$ |

$$T(\tfrac{n}{2}) = \mathbf{T(\tfrac{n}{4}) + \tfrac{n}{2}}$$

$$T(\tfrac{n}{4}) = \mathbf{T(\tfrac{n}{8}) + \tfrac{n}{4}}$$

$$T(\tfrac{n}{8}) = \mathbf{T(\tfrac{n}{16}) + \tfrac{n}{8}}$$

$$\mathbf{n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots\ldots 1 = 2(n-1)}$$

$T(\frac{n}{2^k}) = 1$

$\frac{n}{2^k} = 1 \;\rightarrow\; n = 2^k \;\rightarrow\; \boldsymbol{k = logn}$

**T(n) ≈ T(1) + 2(n-1)**
**≈ 1 + 2 (n – 1)**

Order of growth is **O(n)**

# Example 4 (solution 2)

- $T(n) = T(\frac{n}{2}) + n$ , $n>1$ $\quad\quad$ $T(1) = 1$ (base case)

| k | T(n) |
|---|---|
| 1 | $T(n) = T(\frac{n}{2}) + n$ |
| 2 | $= [\, T(\frac{n}{4}) + \frac{n}{2}\,] + n = \mathbf{T(\frac{n}{4}) + \frac{3n}{2}}$ |
| 3 | $= [T(\frac{n}{8}) + \frac{n}{4}] + \frac{n}{2} + n = \mathbf{T(\frac{n}{8}) + \frac{7n}{4}}$ |
| 4 | $= [T(\frac{n}{16}) + \frac{n}{8}\,] + \frac{n}{4} + \frac{n}{2} + n = \mathbf{T(\frac{n}{16}) + \frac{15n}{8}}$ |
| …… | …… |
| k | $T(n) = T(\frac{n}{2^k}) + [\frac{2^k - 1 * n}{2^{k-1}}]$ |

$T(\frac{n}{2^k}) = 1$

$\frac{n}{2^k} = 1 \quad \rightarrow \quad n = 2^k \quad \rightarrow \quad \boldsymbol{k = logn}$

$$T(\frac{n}{2}) = \mathbf{T(\frac{n}{4}) + \frac{n}{2}}$$

$$T(\frac{n}{4}) = \mathbf{T(\frac{n}{8}) + \frac{n}{4}}$$

$$T(\frac{n}{8}) = \mathbf{T(\frac{n}{16}) + \frac{n}{8}}$$

$$\mathbf{T(n) \approx T(1) + \frac{(n-1) * n}{\frac{n}{2}}}$$

$$\mathbf{\approx 1 + 2\,(n - 1)}$$

Order of growth is **O(n)**

# Exercise:

- Solve the following recurrence relations using iteration method.
- $T(n) = T(n-1) + n$ $\quad T(1) = 1$ ⟹ $O(n^2)$
- $T(n) = T(n-1) + 2n$ $\quad T(0) = 0$ ⟹ $O(n^2)$
- $T(n) = 2\,T(\frac{n}{2}) + n$ $\quad T(1) = 1$ ⟹ $O(n \log n)$

# Master Method

- Master method provides bound for recurrences of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

$$T(n) = a\,T\left(\frac{n}{b}\right) + c.n^{d}$$

Where $a \geq 1$, $b > 1$ *and* $f(n)$ *is a give function which is asymptotically positive.*

- This recurrence characterizes a divide-and-conquer algorithm that divides a problem of size *n* into subproblems, each of size $\frac{n}{b}$, and solves them recursively.

We will explain the parts of this formula in detail later

# Master Method

- There are three cases:
- if $a = b^d$ $T(n) = O(n^d \log_b n)$
- if $a > b^d$ $T(n) = O(n^{\log_b a})$
- if $a < b^d$ $T(n) = O(n^d)$

# Examples

- Apply master method to solve the following recurrences:

- $T(n) = T(\frac{n}{2}) + c$

  $T(n) = T(\frac{n}{2}) + c.n^0$

  **a = 1  b = 2  d = 0**

  a ??  $b^d$  $1 = 2^0$ , so we will follow case 1

  $T(n) = O(n^d \log_b n)$  $T(n) = O(\log_2 n)$

- $T(n) = T(\frac{n}{2}) + n$

  **a = 1  b = 2  d = 1**

  a ??  $b^d$  $1 < 2^1$ , so we will follow case 3

  $T(n) = O(n^d)$  $T(n) = O(n)$

# Examples

- $T(n) = 2\,T(\frac{n}{2}) + n$

  **a = 2  b = 2  d = 1**

  a ?? $b^d$  $2 = 2^1$ , so we will follow case 1

  $T(n) = O(n^d \log_b n)$  T(n) = **$O(n \log_2 n)$**

- $T(n) = 3\,T(\frac{n}{2}) + n^2$

  **a = 3  b = 2  d = 2**

  a ?? $b^d$  $3 < 2^2$ , so we will follow case 3

  $T(n) = O(n^d)$  T(n) = **$O(n^2)$**

- $T(n) = 8\,T(\frac{n}{2}) + n^2$

  **a = 8  b = 2  d = 2**

  a ?? $b^d$  $8 > 2^2$ , so we will follow case 2

  $T(n) = O(n^{\log_b a})$  $T(n) = O(n^{\log_2 8})$ = **$O(n^3)$**

# Exercises

- Solve the following recurrence relations using Master's theorem

1) $T(n) = 8\,T(n) - n^2$

   The given recurrence relation does not correspond to the general form of Master's theorem.

   So, it can not be solved using Master's theorem.

2) $T(n) = 8\,T(\frac{n}{2}) + 1000\,n^2$

3) $T(n) = 16\,T(\frac{n}{4}) + n$

4) $T(n) = 3\,T(\frac{n}{3}) + \frac{n}{2}$

5) $T(n) = 7\,T(\frac{n}{2}) + n^2$

6) $T(n) = 64\,T(\frac{n}{8}) - n^2$ (Does not apply because f(n) is not positive)

# Types of Analysis

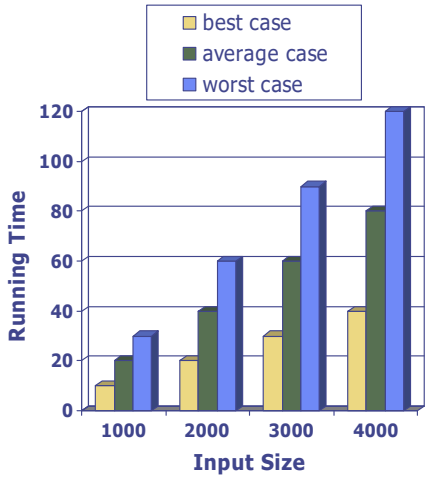
# Types of Analysis

- **Three cases of analysis: Best, worst, and average case running time**

- **Worst case running time:**
  - constraints on the input, rather than size, resulting in the slowest possible running time.
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are.
- **Best case running time:**
  - constraints on the input, rather than size, resulting in the fastest possible running time
  - Provides a lower bound on running time

$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

- **Average case running time:**
  - average running time over every possible type of input
  - usually involve probabilities of different types of input
  - Provides a prediction about the running time
  - Assumes that the input is random



# Types of Analysis

**The worst case is usually fairly:**

- Because it is usually very hard to compute the average running time (the expected performance averaged over all possible inputs!)
- The worst case is usually fairly easy to analyze and often close to the average or real running time.

# Example: Linear Search Algorithm

**search for x in array A of n items.**

**If x = 33** → $7\ comparisons$

If x = 10 → 1 comparison


**Best case**: x present at the first element

Time is constant → $T_B(n) = O(1)$

**Worst case**: x does not present in the array

Time is linear → $T_w(n) = O(n)$

**Average case**: $\frac{All\ possible\ case\ time}{no\ of\ cases}$

$$= \frac{1+2+3+4+ \ldots\ldots\ldots n}{n} = \frac{\frac{n\,(n+1)}{2}}{n} = \frac{n+1}{2}$$

Time is linear → $T_{avg}(n) = O(n)$

Recall $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

# Example: Linear Search Algorithm

**search for x in array A of n items.**

Your algorithm should return the index of found item and **n+1** if not found (assuming that the first index is 1)

```
Seq_search (A, n , x)
{                                   cost        time
1.   i=1                          c1          1
2.   while(i<=n && A[i] != x)     c2          ?
3.        i++                     c3          ?
4.   return i                     c4          1
}
```

The # of times lines 2, 3 are executed depends on the data in the array and not just *n*

# Example: Linear Search Algorithm

**Worst case: x does not present in the original array**

```
Seq_search (A, n , x)
{                                   cost        time
1.   i=1                          c1          1
2.   while(i<=n && A[i] != x)     c2          n+1
3.        i++                     c3          n
4.   return I                     c4          1
}
```

$T(n) = c1 + c2(n+1) + c3\ n + c4$

$= c1 + c2 + c4 + (c2 + c3)\ n$

$T(n) = a\ n + b$ where a and b are constants → **O(n)**

# Example: Linear Search Algorithm

**Best case: x is found at the first element A[1]**

```
Seq_search (A, n , x)
{                                   cost        time
1.   i=1                          c1          1
2.   while(i<=n && A[i] != x)     c2          1
3.        i++                     c3          0
4.   return I                     c4          1
}
```

T(n) = c1 + c2 + c4

= C [constant time] → O(1)

# Example: Linear Search Algorithm

**What about average case analysis???**

الموضوع معقد

```
Seq_search (A, n , x)
{                                   cost      time
1.   i=1                          c1          1
2.   while(i<=n && A[i] != x)     c2          ?
3.        i++                     c3          ?
4.   return I                     c4          1
}
```

It is hard to compute the average running time (the expected performance averaged over all possible inputs!)

# Exercise

**Consider the following algorithm that finds the largest element of an array**

**(Assume the first index is 1)**

```
Find_Max(A, n)
{                                  cost     time
1.  Max=A[1]
2.  for i=2 to n
3.       If(A[i]>Max)
4.            Max=A[i]
5.  return Max
}
```

1) What is the worst, best, and average cases?
2) Do a line by line analysis for the worst and best cases and derive the time complexity (use Big O notation)

[2, 3, 4, 5, 6, 7, 8, 9] Max = 9
The **worst case** is to input a **sorted array** from smallest to largest

What is the best case?
9, 8, 7, 6, 5, 4, 3, 2 Max = 9

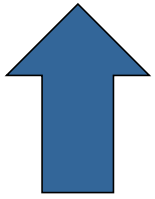
What is the average case?

# Asymptotic Notations

# Asymptotic Analysis

- To compare two algorithms with running times ***f(n)*** and ***g(n)***, we need a **rough measure** that characterizes **how fast each function grows.**
- *Hint:* use *rate of growth*
- Compare functions in the limit, that is, **asymptotically!**

  (i.e., **for large values of *n***)



# Order of Growth

- The low order terms in a function are relatively insignificant for **large** $n$

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

*i.e.*, we say that $n^4 + 100n^2 + 10n + 50$ and $n^4$ have the same **rate of growth**

# Order of growth

Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input:

- **Algorithm A:** takes 100 $n$ + 1 steps to solve a problem with size $n$;
- **Algorithm B:** takes $n^2$ + $n$ + 1 steps.

The **leading term** is the term with the highest exponent.

The following table shows the run time of these algorithms for different problem sizes:

| Input size<br>***n*** | Run time of Algorithm A<br>**100 *n* + 1 steps** | Run time of Algorithm B<br>**$n^2$ + *n* + 1 steps** |
|---|---|---|
| 10 | 1 001 | 111 |
| 100 | 10 001 | 10 101 |
| 1 000 | 100 001 | 1 001 001 |
| 10 000 | 1 000 001 | > $10^9$ |

Which algorithm is better????

# Order of growth

| Input size<br>$n$ | Run time of Algorithm A<br>**$100n + 1$ steps** | Run time of Algorithm B<br>**$n^2 + n + 1$ steps** |
|---|---|---|
| 10 | 1 001 | 111 |
| 100 | 10 001 | 10 101 |
| 1 000 | 100 001 | 1 001 001 |
| 10 000 | 1 000 001 | $> 10^9$ |

- At $n$ =10, Algorithm A looks bad
  - For Algorithm A, the leading term has a **large coefficient**, 100, which is why B does better than A for small $n$.
- But for $n$=100 they are about the same,
- **for larger values of n, A is much better.**
  - **any function that contains an $n^2$ term will grow faster than a function whose leading term is n**.

Algorithm A is better than Algorithm B for sufficiently large n.

# Values of some important functions as $n$ grows

| $n$ | $\log_2 n$ | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| 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 | $10^6$ | $2.0 \cdot 10^7$ | $10^{12}$ | $10^{18}$ | | |

**Table 2.1** Values (some approximate) of several functions important for analysis of algorithms

ألجوريثم مقبول

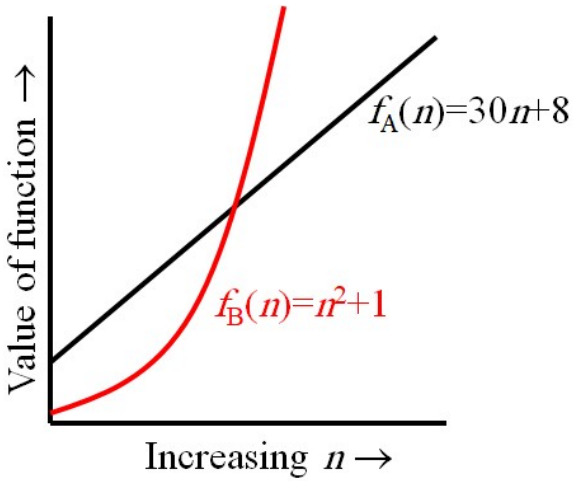
ممكن يوخذ ملايين من السنوات حتى يعطي نتيجة بزيادة n.

# How to compare algorithms?

- for large problems, we expect an algorithm with a smaller leading term to be a better algorithm
- but for smaller problems, there may be a **crossover point** where another algorithm is better.
  - The location of the crossover point depends on the details of the algorithms, the inputs and the hardware,

On the graph, as you go to the right, a faster growing function eventually becomes larger...

ممكن الألجوريثم الأسوأ يكون أفضل




Visualizing Orders of Growth

# How to compare algorithms?

- If two algorithms have the same leading order term, it is hard to say which is better; the answer will depend on the details.
  - **they are considered equivalent, even if they have different coefficients.**

**Algorithm A : T(n) = 1000 n + 100**

**Algorithm B: T(n) = 2n + 3**

Which algorithm is better????

# Order of growth

An **order of growth** is a set of functions whose growth is considered equivalent.

Examples:

- $2n$, $100n$ and $n + 1$ belong to the same order of growth, which is written $O(n)$ in "Big-Oh notation"
- All functions with the leading term $n^2$ belong to $O(n^2)$;

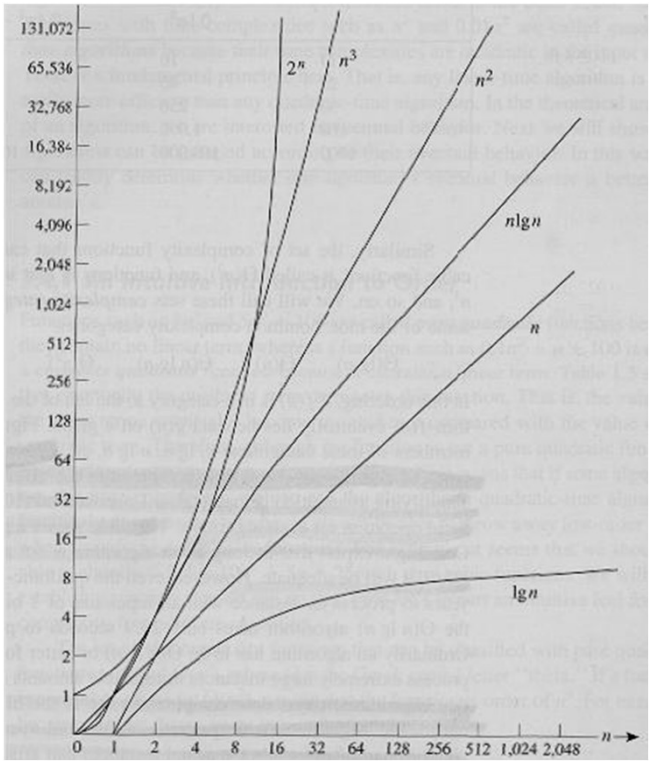
- What is the order of growth of $n^3 + n^2$?
- What about $1000000\ n^3 + n^2$. What about $n^3 + 1000000\ n^2$?
- What is the order of growth of $(n^2 + n) * (n + 1)$?

# Basic asymptotic efficiency classes

order of growth

| | |
|---|---|
| 1 | constant |
| log $n$ | logarithmic |
| $n$ | linear |
| $n$ log $n$ | $n$-log-$n$ or linearithmic |
| $n^2$ | quadratic |
| $n^3$ | cubic |
| $2^n$ | exponential |
| $n!$ | factorial |

سرعة نمو الدالة الجورثم بتتم بزيادة n



# Execution times for algorithms with the given time complexities

**Table 1.4** Execution times for algorithms with the given time complexities

| $n$ | $f(n) = \lg n$ | $f(n) = n$ | $f(n) = n \lg n$ | $f(n) = n^2$ | $f(n) = n^3$ | $f(n) = 2^n$ |
|---|---|---|---|---|---|---|
| 10 | 0.003 $\mu$s* | 0.01 $\mu$s | 0.033 $\mu$s | 0.1 $\mu$s | 1 $\mu$s | $\mu$s |
| 20 | 0.004 $\mu$s | 0.02 $\mu$s | 0.086 $\mu$s | 0.4 $\mu$s | 8 $\mu$s | ms† |
| 30 | 0.005 $\mu$s | 0.03 $\mu$s | 0.147 $\mu$s | 0.9 $\mu$s | 27 $\mu$s | s |
| 40 | 0.005 $\mu$s | 0.04 $\mu$s | 0.213 $\mu$s | 1.6 $\mu$s | 64 $\mu$s | 18.3 mir |
| 50 | 0.005 $\mu$s | 0.05 $\mu$s | 0.282 $\mu$s | 2.5 $\mu$s | 25 $\mu$s | 13 days |
| $10^2$ | 0.007 $\mu$s | 0.10 $\mu$s | 0.664 $\mu$s | 10 $\mu$s | 1 ms | $4 \times 10^{15}$ years |
| $10^3$ | 0.010 $\mu$s | 1.00 $\mu$s | 9.966 $\mu$s | 1 ms | 1 s | |
| $10^4$ | 0.013 $\mu$s | 0 $\mu$s | 130 $\mu$s | 100 ms | 16.7 min | |
| $10^5$ | 0.017 $\mu$s | 0.10 ms | 1.67 ms | 10 s | 11.6 days | |
| $10^6$ | 0.020 $\mu$s | 1 ms | 19.93 ms | 16.7 min | 31.7 years | |
| $10^7$ | 0.023 $\mu$s | 0.01 s | 0.23 s | 1.16 days | 31,709 years | |
| $10^8$ | 0.027 $\mu$s | 0.10 s | 2.66 s | 115.7 days | $3.17 \times 10^7$ years | |
| $10^9$ | 0.030 $\mu$s | 1 s | 29.90 s | 31.7 years | | |

*1 $\mu$s = $10^{-6}$ second.
†1 ms = $10^{-3}$ second.

# Asymptotic Notations

- Notations used for representing the simple form of a function or showing the class of a function.
- It is a simple method for representing the time complexity.
- A way of comparing functions that ignores constant factors and small input sizes.
- Asymptotic Notations:
  - O Big-O notation : upper bound of a function.
  - Ω **omega** notation: Lower bound of a function.
  - Θ **theta** notation : Average bound

Recall ➡ $1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots\ldots < 2^n < 3^n < \ldots\ldots\ldots < n^n < n!$

# Big-O notation

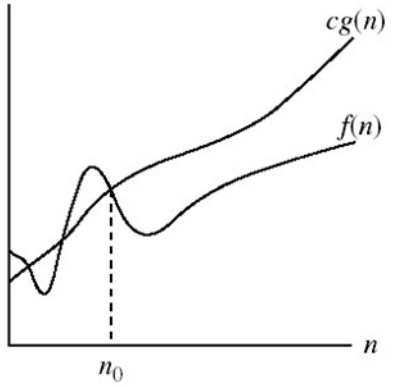
Let **f** and **g** be nonnegative functions on the positive integers

- We write f(n) = O (g(n)) and say that:
  - f(n) is big oh of g(n) or,
  - f(n) is of order at most g(n) or,
  - g is an asymptotic upper bound for f
- O($g(n)$): class of functions $f(n)$ that grow no faster than $g(n)$

"Asymptotically" because it matters for only large values of n

**Definition:**

The function $f(n) = O(g(n))$ iff $\exists$ $+ve$ constants $c$ and $n_0$ such that $f(n) \leq c\, g(n)\ \forall\ n \geq n_0$




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

# Big-O notation

Example: f(n) = 3n + 2

$$3n + 2 \quad \leq \quad 10\,n \qquad n \geq 1$$

f(n) c g(n) → f(n) = O(n)

We can also say that $3n + 2 \leq 5\ n^2$ → f(n) = $O(n^2)$

$$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots\ldots < 2^n < 3^n < \ldots\ldots\ldots < n^n$$

Upper bound

**But when writing Big-O notation we try to find the closest function, So 3n + 2 is O(n)**

# More Examples …

Drop constants and lower order terms. E.g. O(3*n^2 + 10n + 10) becomes O(n^2).

- $n^4 + 100n^2 + 10n + 50$ is $O(n^4)$
- $10n^3 + 2n^2$ is $O(n^3)$
- $n^3 - n^2$ is $O(n^3)$
- $n\log n + n$ $O(\log n)$
- $2^n + n^2$ → lower term $O(2^n)$
- constants
  - 10 is $O(1)$
  - 1273 is $O(1)$

$1 < \log n < \sqrt{n} < n\log n < n^2 < n^3 \cdots < 2^n < 3^n$

$\cdots < n^n < n!$

للأسوأ يلي
بكلف أكثر

# Example

- Show that $30n + 8$ is $O(n)$.

**Solution**: Use Big-O definition

$g(n) = n$

Show that ∃ **(there exist)** $c, n_0$ *such that* $30n + 8 \leq cn, \ \forall n \geq n_0$ .

- Let $c = 31$, $n_0 = 8$.
- $30n + 8 \leq 31n \qquad \forall n \geq 8$

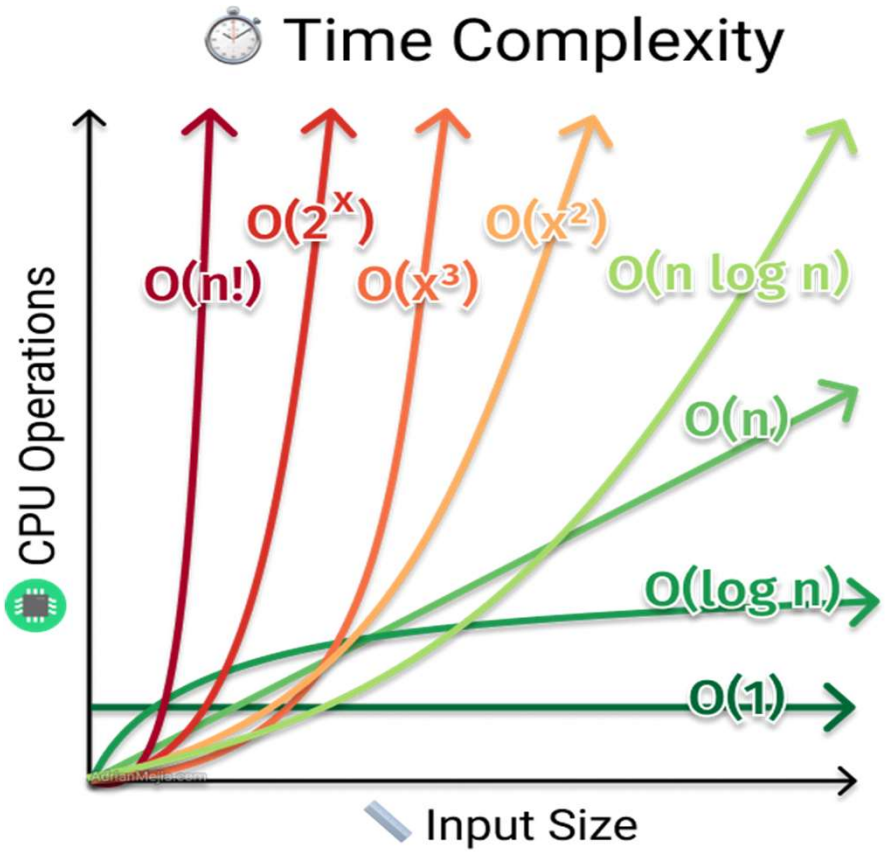
The following table shows some of the orders of growth (using Big-O notation) that appear most commonly in algorithmic analysis, in increasing order of badness.

| **Order of growth** | **Name** |
|---|---|
| O(1) | Constant. |
| O(log $n$) | Logarithmic. |
| O($n$) | Linear. |
| O($n$ log $n$) | Log-linear or linearithmic |
| O($n^2$) | Quadratic. |
| O($n^3$) | Cubic. |
| O(n$^c$), $c > 1$ | Polynomial, sometimes called algebraic.<br>Examples: O($n^2$), O($n^3$), O($n^4$). |
| O($c^n$) | Exponential, sometimes called geometric.<br>Examples: O($2^n$), O($3^n$). |
| O($n$!) | Factorial, sometimes called combinatorial. |

*badness*

*n* is the problem size.

# Plot of the most common Big-O notation



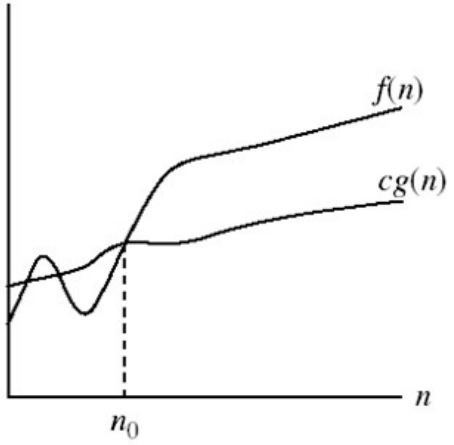
# Ω notation

Let **f** and **g** be nonnegative functions on the positive integers

- We write **f(n) = Ω (g(n))** and say that:
  - f(n) is omega of g(n) or,
  - f(n) is of order at least g(n) or,
  - g is an asymptotic lower bound for f
- Ω(g(n)): class of functions f(n) that grow at least as fast as g(n)

**Definition:**

The function $f(n) = \Omega(g(n))$ iff $\exists$ $+ve$ $constants$ $c$ $and$ $n_0$ $such$ $that$ $f(n) \geq c\ g(n)\ \forall\ n \geq n_0$




$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

# Ω notation

Example: f(n) = 3n + 2

$3n + 2 \geq 1\,n \qquad n \geq 1$

f(n) c g(n) → f(n) = Ω(n)

We can also say that 3n + 2≥ logn → f(n) = Ω(logn)

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

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots\ldots < 2^n < 3^n < \ldots\ldots\ldots < n^n$

Lower bound

**But when writing Ω notation we try to find the closest function, So 3n + 2 is Ω(n)**

# Θ notation

Let **f** and **g** be nonnegative functions on the positive integers

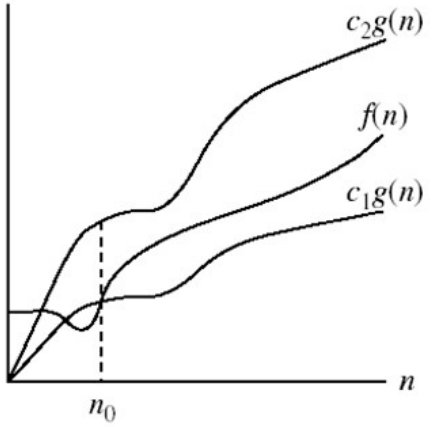
- We write **f(n) = Θ (g(n))** and say that:
  - f(n) is theta of g(n) or,
  - f(n) is of order of g(n) or,
  - g is an asymptotic tight bound for f
- Θ(g(n)): class of functions f(n) that grow at same rate as g(n)

A tight bound implies that both the lower and the upper bound for the computational complexity of an algorithm are the same.

**Definition:**

The function $f(n) = \Theta(g(n))$ iff $\exists$ $+ve$ $constants$ $c1, c2$ $and$ $n_0$ $such$ $that$

$$c1\ g(n) \leq f(n) \leq c2\ g(n) \quad \forall\ n \geq n_0$$




$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

# Θ notation

Example: f(n) = 3n + 2

$$1\,n \leq 3n + 2 \leq 5\,n \qquad n \geq 1$$

c1 g(n) f(n) c2 g(n) → f(n) = Θ(n)

Is f(n) = Θ($n^2$) (wrong)

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots\ldots < 2^n < 3^n < \ldots\ldots\ldots < n^n$

# Θ notation

Example: f(n) = n  g(n) = $2^n$

**Is f(n)= Θ(g(n))?  Is n is Θ($2^n$)?**

1 < logn < $\sqrt{n}$ < **n** < nlogn < $n^2$ < $n^3$ < ……. < **$2^n$** < $3^n$ < …… < $n^n$

Solution:

→ n<=$c_1$* $2^n$  $\forall\, n \geq n_0$  **yes** always  → n is O($2^n$)

→ n>=$c_2$* $2^n$  $\forall\, n \geq n_0$  **NO**  → n is not Ω($2^n$)

**f(n) is not Θ($2^n$)**

# Example

**f(n) = $60n^2+5n+1$** **g(n) =$n^2$** **Is f(n)= $\Theta(g(n))$?**

$1 < \log n < \sqrt{n} < n < n\log n < \mathbf{n^2} < n^3 < \ldots\ldots < 2^n < 3^n < \ldots\ldots\ldots < n^n$

→**Is $60n^2+5n+1 = O(n^2)$?**

$f(n) <= c_1 g(n)$ for all $n >= n_o$ and some constant $c_1 > 0$

→ $60n^2+5n+1 <= c_1 * n^2$

yes for $c_1 = 66$ and $n >= 1$

→**Is $60n^2+5n+1 = \Omega(n^2)$?**

$f(n) >= c_2 g(n)$ for all $n >= n_o$ and some constant $c_2 > 0$

→ $60n^2+5n+1 >= c_2 * n^2$

yes for $c_2 = 60$ and $n >= 1$

**f(n) is $\Theta(n^2)$**

# Example

**$T(n) = 32n^2 + 17n + 32$**

**Represent T(n) using asymptotic notations**

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

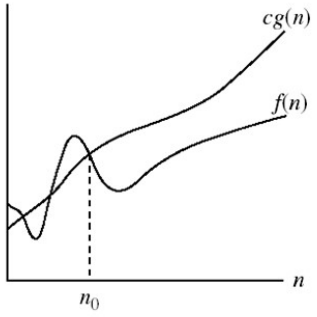
$T(n)= \Omega(n^2)$

$T(n)= \Theta(n^2)$

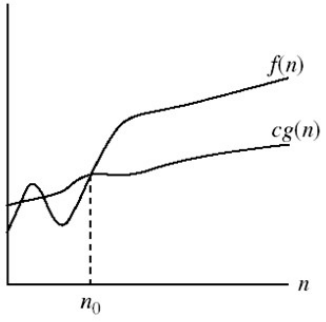
ignore the multiplicative constants and the lower order terms

# Summary

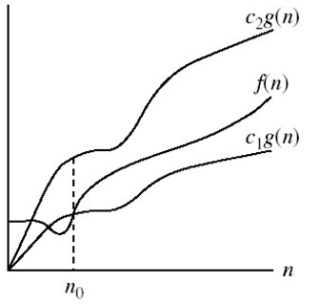
| O notation | Ω notation | Θ notation |
|---|---|---|
| The function $f(n) = O(g(n))$ iff $+ve\ constants\ c\ and\ n_0$ $such\ that\ f(n) \leq c\ g(n)\ \forall\ n \geq n_0$ | The function $f(n) = \Omega(g(n))$ iff $\exists$ $+ve\ constants\ c\ and\ n_0$ $such\ that\ f(n) \geq c\ g(n)\ \forall\ n \geq n_0$ | The function $f(n) = \Theta(g(n))$ iff $\exists$ $+ve\ constants\ c1, c2\ and\ n_0$ $such\ that$ $c1\ g(n) \leq f(n) \leq c2\ g(n) \quad \forall\ n \geq n_0$ |




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




$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.




$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

# Asymptotic Analysis of Algorithms (Asymptotic → for large n)

**big oh** expressions greatly simplify the analysis of the running time of algorithms

- **all that we get is an upper bound on the running time of the algorithm**

- the result does **not** depend upon the values of the constants
- the result does **not** depend upon the characteristics of the computer and compiler actually used to execute the program!

## Conventions for Writing Big Oh Expressions (Tight bound)

- Ignore the multiplicative constants
  - Instead of writing $O(3n^2)$, we simply write $O(n^2)$
  - If the function is constant (e.g. O(1024) we write O(1))
- Ignore the lower order terms
  - Instead of writing $O(n\log n+n+n^2)$, we simply write $O(n^2)$

# Examples

**$T(n) = 32n^2 + 17n + 32$**

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

- *n, n+1, n+80, 40n, n+log n* *is O(n)*
- $n^2$ + ***10000000000n*** is $O(n^2)$
- ***$3n^2$ + 6n + log n + 24.5*** is $O(n^2)$

# NOTES

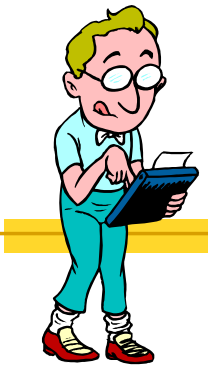
- A problem that has a worst case **Polynomial time** algorithm is considered to have a **good** algorithm.
  - Such problems are called **feasible** or **tractable**

- A problem that does not have a worst case **Polynomial time** algorithm is said to be **intractable**

- **NP-complete problems and NP-hard problems**

# Math you need to review

# Math you need to Review



- Summations
- Logarithms and Exponents
- Proof techniques

- **properties of logarithms:**
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b ax = a\log_b x$
  $\log_b a = \log_x a/\log_x b$
- **properties of exponentials**:
  $a^{(b+c)} = a^b a^c$
  $a^{bc} = (a^b)^c$
  $a^b / a^c = a^{(b-c)}$
  $b = a^{\log_a b}$
  $b^c = a^{c*\log_a b}$

# Common summation formulas

1. $\sum_{i=1}^{n} c = cn$

2. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

3. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

4. $\sum_{i=1}^{n} i^3 = \left[\frac{n(n+1)}{2}\right]^2$

$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

c + c + c + ..... $c_n$

1 + 2 + 3 + 4 + 5 … + n

$1^2 + 2^2 + 3^2 + 4^2 + 5^2 \ldots + n^2$

$1^3 + 2^3 + 3^3 + 4^3 + 5^3 \ldots + n^3$

$1 + 2 + 4 + 8 + 16 + \ldots + 2^n$

# Logarithms and properties

- In algorithm analysis we often use the notation "log n" without specifying the base

Binary logarithm $\lg n = \log_2 n$

Natural logarithm $\ln n = \log_e n$

$\lg^k n = (\lg n)^k$

$\lg \lg n = \lg(\lg n)$

$\log x^y = y \log x$

$\log xy = \log x + \log y$

$\log \frac{x}{y} = \log x - \log y$

$a^{\log_b x} = x^{\log_b a}$

$\log_b x = \frac{\log_a x}{\log_a b}$