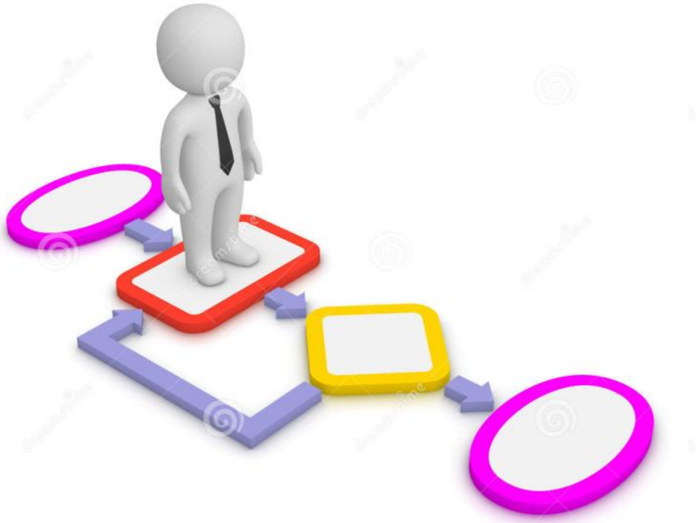
# ASYMPTOTIC ANALYSIS



Design & Analysis of Algorithms

**Saman Riaz (PhD)**
**Associate Professor**
**RSCI, Lahore**

**Lecture # 03**

# What’s Analysis of Algorithms

- The theoretical study of algorithm’s performance and resource usage.
- Other important features of an algorithm are
  - modularity
  - correctness
  - maintainability
  - functionality
  - robustness
  - user-friendliness
  - programmer time
  - Simplicity
  - extensibility
  - Reliability
- During this course our focus will be on **the performance and the storage requirements**.

# Types of Analysis

- Worst case
  - 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
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$\boxed{Lower\ Bound \le Running\ Time \le Upper\ Bound}$$

- Average case
  - Provides a prediction about the running time
  - Assumes that the input is random

# How do we compare algorithms?

- We need to define a number of objective measures.

(1) Compare execution times?

***Not good***: times are specific to a particular computer !!

(2) Count the number of statements executed?

***Not good***: number of statements vary with the programming language as well as the style of the individual programmer.

# Ideal Solution

- Express running time $t$ as a function of problem size $n$ (i.e., $t=f(n)$ ).
- Given two algorithms having running times $f(n)$ and $g(n)$, find which functions grows faster.
- Such an analysis is independent of machine time, programming style, etc.

# How do we find *f(n)*?

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

***Algorithm 1***

| | **Cost** |
|---|---|
| arr[0] = 0; | $c_1$ |
| arr[1] = 0; | $c_1$ |
| arr[2] = 0; | $c_1$ |
| ... | ... |
| arr[N-1] = 0; | $c_1$ |
| | ----------- |

***Algorithm 2***

| | **Cost** |
|---|---|
| for(i=0; i<N; i++) | $c_2$ |
| arr[i] = 0; | $c_1$ |
| | ------------- |

# How do we find *f(n)*?

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

***Algorithm 1***

| | **Cost** |
|---|---|
| arr[0] = 0; | $c_1$ |
| arr[1] = 0; | $c_1$ |
| arr[2] = 0; | $c_1$ |
| ... | ... |
| arr[N-1] = 0; | $c_1$ |
| | ----------- |

$c_1+c_1+...+c_1 = c_1 \times N$

***Algorithm 2***

| | **Cost** |
|---|---|
| for(i=0; i<N; i++) | $c_2$ |
| arr[i] = 0; | $c_1$ |
| | ------------- |

$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

# How do we find *f(n)*?

- ***Algorithm 3*** *Cost*

```
sum = 0;                          c1
for(i=0; i<N; i++)                c2
   for(j=0; j<N; j++)             c2
         sum += arr[i][j];        c3
                                  ------------
```

# How do we find *f(n)*?

- ***Algorithm 3*** *Cost*

| | |
|---|---|
| sum = 0; | $c_1$ |
| for(i=0; i<N; i++) | $c_2$ |
| for(j=0; j<N; j++) | $c_2$ |
| sum += arr[i][j]; | $c_3$ |
| | ------------ |

$$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$$

# Comparing algorithms

- Given two algorithms having running times $f(n)$ and $g(n)$, how do we decide which one is faster?

- Compare "rates of growth" of $f(n)$ and $g(n)$

# Understanding Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

**Cost**: (cost_of_elephants) + (cost_of_goldfish)

**Approximation:**

**Cost** ~ cost_of_elephants

# Understanding Rate of Growth (cont'd)

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

$$n^4 + 100n^2 + 10n + 50$$

**Approximation:**

$$n^4$$

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

- Highest order term determines rate of growth!

# Work done by an algorithm

Let

$T(n)$: running time

$c_{op}$ : execution time for basic operation

$C(n)$ : number of times basic operation is executed

- Then we have: $T(n) \approx c_{op}\, C(n)$

# Types of formulas for basic operation count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

- Formula indicating order of growth with unknown multiplicative constant

  e.g., $C(n) \approx cn^2$

# Example

Let $\mathbf{C(n) = 3n(n\text{-}1)} \approx \mathbf{3n^2}$

Suppose we double the input size.

How much longer the program will run?

# Orders of Growth

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

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

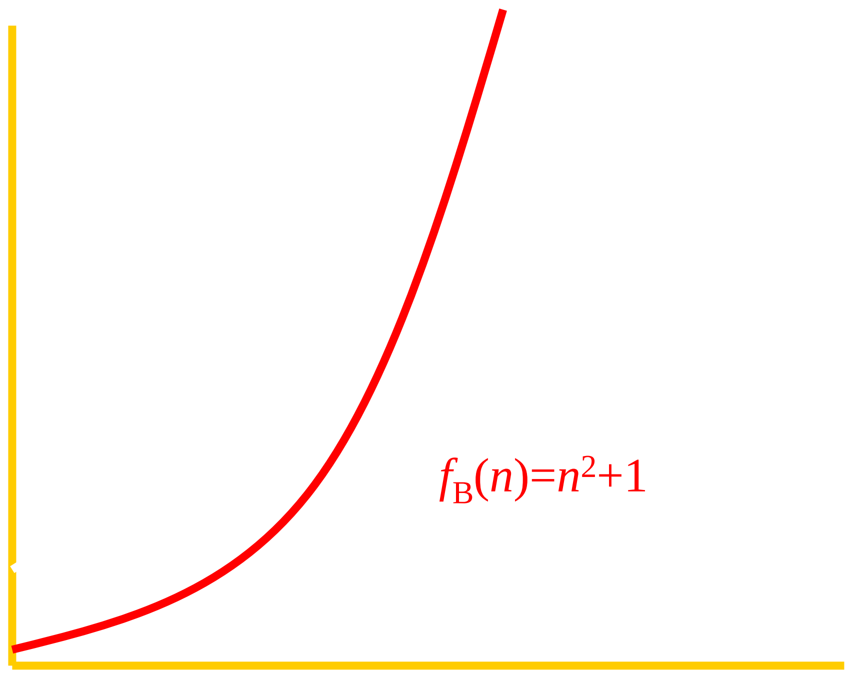
# Example

- Suppose you are designing a website to process user data (*e.g.*, financial records).
- Suppose program A takes $\boldsymbol{f_A(n)=30n+8}$ microseconds to process any $n$ records, while program B takes $\boldsymbol{f_B(n)=n^2+1}$ microseconds to process the $n$ records.
- Which program would you choose, knowing you'll want to support millions of users?

Compare rates of growth:

# Visualizing Orders of Growth

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



# Rate of Growth ≡ Asymptotic Analysis

- Using *rate of growth* as a measure to compare different functions implies comparing them **asymptotically** (i.e., as n $\rightarrow \infty$ )

- If *f*(*x*) is *faster growing* than *g*(*x*), then *f*(*x*) always eventually becomes larger than *g*(*x*) **in the limit** (i.e., for large enough values of *x*).

# Asymptotic Notation

- **O notation:** asymptotic "less than":

$f(n)=O(g(n))$ implies: $f(n)$ "$\leq$" $c\, g(n)$ in the limit*

(used in worst-case analysis)

# Asymptotic Notation

- **Ω notation:** asymptotic "greater than":

f(n)= Ω (g(n)) <u>implies:</u> f(n) "≥" c g(n) in the limit*

(used in best-case analysis)

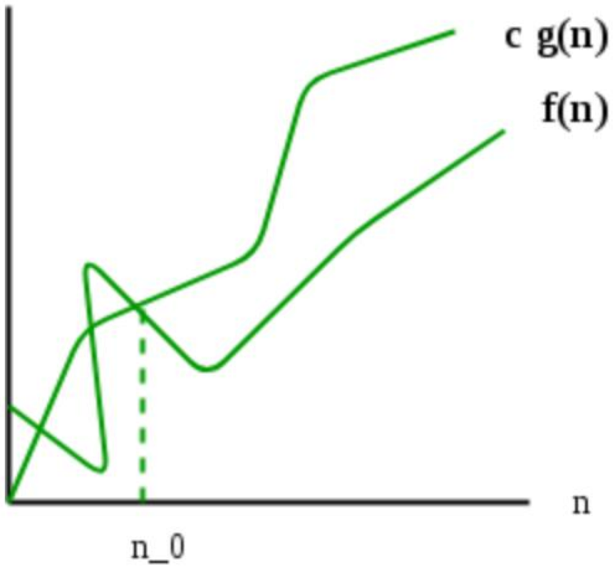
# Asymptotic Notation

- **Θ notation:** asymptotic "equality":

$f(n) = \Theta(g(n))$ <u>implies:</u> f(n) "=" c g(n) in the limit*

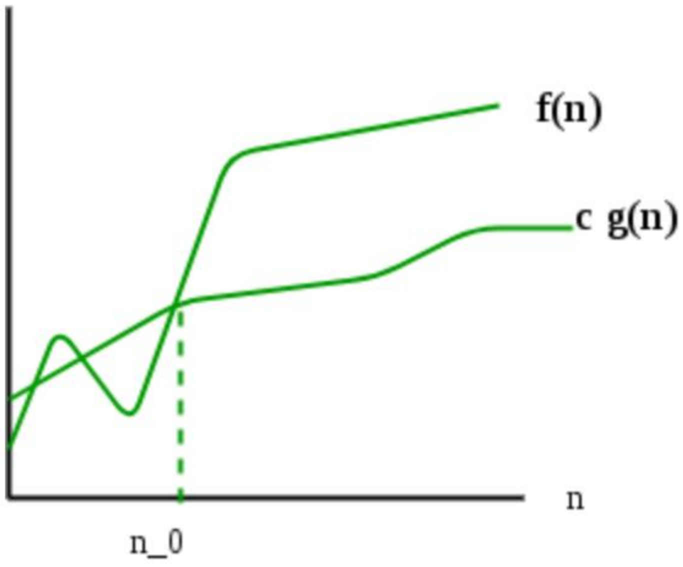
(provides a **tight bound** of running time)
(best and worst cases are same)

# Asymptotic notations (Big O, Omega, Theta)



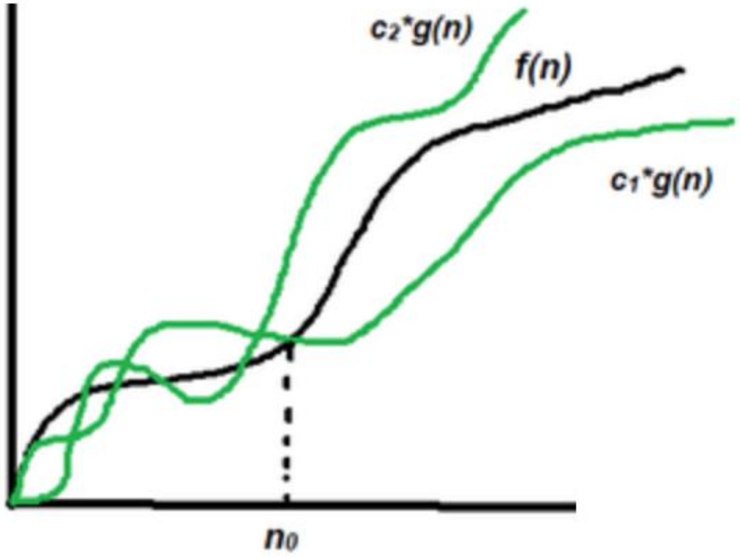

f(n) = O(g(n))

- Worst Case
- Upper Bound ( At most)




f(n) = Omega(g(n))

- Best Case
- Lower Bound(At least)




f(n) = Ω(g(n))

- Average Case
- Exact time

# Why study algorithms ?

- If you are given two brand new algorithms from two different companies to perform sorting. Which one you would go for ? Lets assume companies are not willing to install the software at your end for testing but are willing to share their pseudo-code with you.
  - You need an objective analysis of both the algorithms before you can choose one. Like:-
    - Scalability of algorithms
    - Real life constraints like time and storage
    - Behavior of the algorithms
    - Quickness (speed is fun)

# Machine Independent Time

- What is insertion sort's worst-case time?
- It depends on the speed of our computer:
  - relative speed (on the same machine),
  - absolute speed (on different machines).
- **<u>BIG IDEA</u>**
- Ignore machine-dependent constants.
- Look at the growth of running time $T(n)$ as $n \rightarrow \infty$
  - **"Asymptotic Analysis"**

# Θ-notation

- Commonly used notation
- Idea is to drop low-order terms, ignore leading constants
  - Example: $3n^3+90n^2-5n+6046 = \Theta(n^3)$
- Mathematically:
  - $\Theta(g(n)) = \{ f(n)$: there exist positive constants $c_1,c_2$, and $n_0$ such that $0 <= c_1\, g(n) <= f(n) <= c_2\, g(n)$ for all $n >= n_0\}$

# Asymptotic Performance

- As n gets large enough, a $\Theta(n^2)$ algorithm always beats a $\Theta(n^3)$ algorithm.
- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

# Example Sorting (Insertion Sort)

Insertion-Sort(A,n) → A[1 … n]

for j ← 2 to n

do key ← A[j]

i ← j-1

while i > 0 and A[i] > key

do A[i+1] ← A[i]

i ← i -1

A[i+1] = key

# Running time of Insertion Sort

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input,
  - short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

# Analysis of Insertion Sort

- Time to compute the **running time** as a function of the **input size**

| | cost | times |
|---|---|---|
| 1. **for** j=2 **to** *length*(A) | $c_1$ | n |
| 2. **do** key=A[j] | $c_2$ | n-1 |
| 3. "insert A[j] into the sorted sequence A[1..j-1]" | 0 | n-1 |
| 4. i=j-1 | $C_3$ | n-1 |
| 5. **while** i>0 **and** A[i]>key | $C_4$ | $\sum_{j=2}^{n} t_j$ |
| 6. **do** A[i+1]=A[i] | $C_5$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 7. i-- | $C_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 8. A[i+1]:=key | $C_7$ | n-1 |

# Insertion-Sort Running Time

$$T(n) = c_1 \bullet [n] + c_2 \bullet (n\text{-}1) + c_3 \bullet (n\text{-}1) + c_4 \bullet (n\text{-}1) + c_5 \bullet (\Sigma_{j=2,n}\, t_j) + c_6 \bullet (\, \Sigma_{j=2,n}\, (t_j - 1)\,) + c_7 \bullet (\, \Sigma_{j=2,n}\, (t_j - 1)\,) + c_8 \bullet (n\text{-}1)$$

*$c_3 = 0$, of course, since it's the comment*

# Best/Worst/Average Case

- **Best case**: elements already sorted ® $t_j=1$, running time = $f(n)$, i.e., *linear* time.

- **Worst case**: elements are sorted in inverse order ® $t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time

- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

# Best Case Result

Occurs when array is already sorted.

For each j = 2, 3,…..n we find that A[i]<=key in line 5 when I has its initial value of j-1.

T(n) =

$c_1 \bullet n + (c_2 + c_4) \bullet (n\text{-}1) + c_5 \bullet (n\text{-}1) + c_8 \bullet (n\text{-}1)$

$= n \bullet ( c_1 + c_2 + c_4 + c_5 + c_8 )$

$+ ( -c_2 – c_4 - c_5 – c_8 )$

$= c_9 n + c_{10}$

$= f_1(n^1) + f_2(n^0)$

# Worst Case T(n)

- Occurs when the loop of lines 5-7 is executed as many times as possible, which is when A[] is in reverse sorted order.
- key is A[j] from line 2
- i starts at j-1 from line 4
- i goes down to 0 due to line 7
- So, $t_j$ in lines 5-7 is [(j-1) – 0] + 1 = j

*The ‘1’ at the end is due to the test that fails, causing exit from the loop.*

# Worst Case T(n), ctd.

$T(n) = c_1 \bullet [n] + c_2 \bullet (n-1) + c_4 \bullet (n-1) + c_5 \bullet (\Sigma_{j=2,n}\, j) + c_6 \bullet [\, \Sigma_{j=2,n}\, (j-1)\,] + c_7 \bullet [\, \Sigma_{j=2,n}\, (j-1)\,] + c_8 \bullet (n-1)$

# Worst Case T(n), ctd.

$T(n) =$

$c_1 \bullet n + c_2 \bullet (n-1) + c_4 \bullet (n-1) + c_8 \bullet (n-1)$
$+ c_5 \bullet (\Sigma_{j=2,n}\ j)$
$+ c_6 \bullet [\ \Sigma_{j=2,n}\ (j-1)\ ] + c_7 \bullet [\ \Sigma_{j=2,n}\ (j-1)\ ]$

$= c_9 \bullet n + c_{10} + c_5 \bullet (\Sigma_{j=2,n}\ j) + c_{11} \bullet [\Sigma_{j=2,n}\ (j-1)\ ]$

# Worst Case T(n), ctd.

$T(n) = \quad c_9 \bullet n + c_{10} + c_5 \bullet (\Sigma_{j=2,n}\ j) + c_{11} \bullet [\ \Sigma_{j=2,n}\ (j-1)\ ]$

But

$\Sigma_{j=2,n}\ j = [n(n+1)/2] - 1$

so that

$\Sigma_{j=2,n}\ (j-1) = \Sigma_{j=2,n}\ j - \Sigma_{j=2,n}\ (1)$

$= [n(n+1)/2] - 1 - (n-2+1)$

$= [n(n+1)/2] - 1 - n + 1 = n(n+1)/2 - n$

$= [n(n+1)-2n]/2 = [n(n+1-2)]/2 = n(n-1)/2$

*Wasn't that fun?*

# Worst Case T(n), ctd.

In conclusion,

$T(n) =$

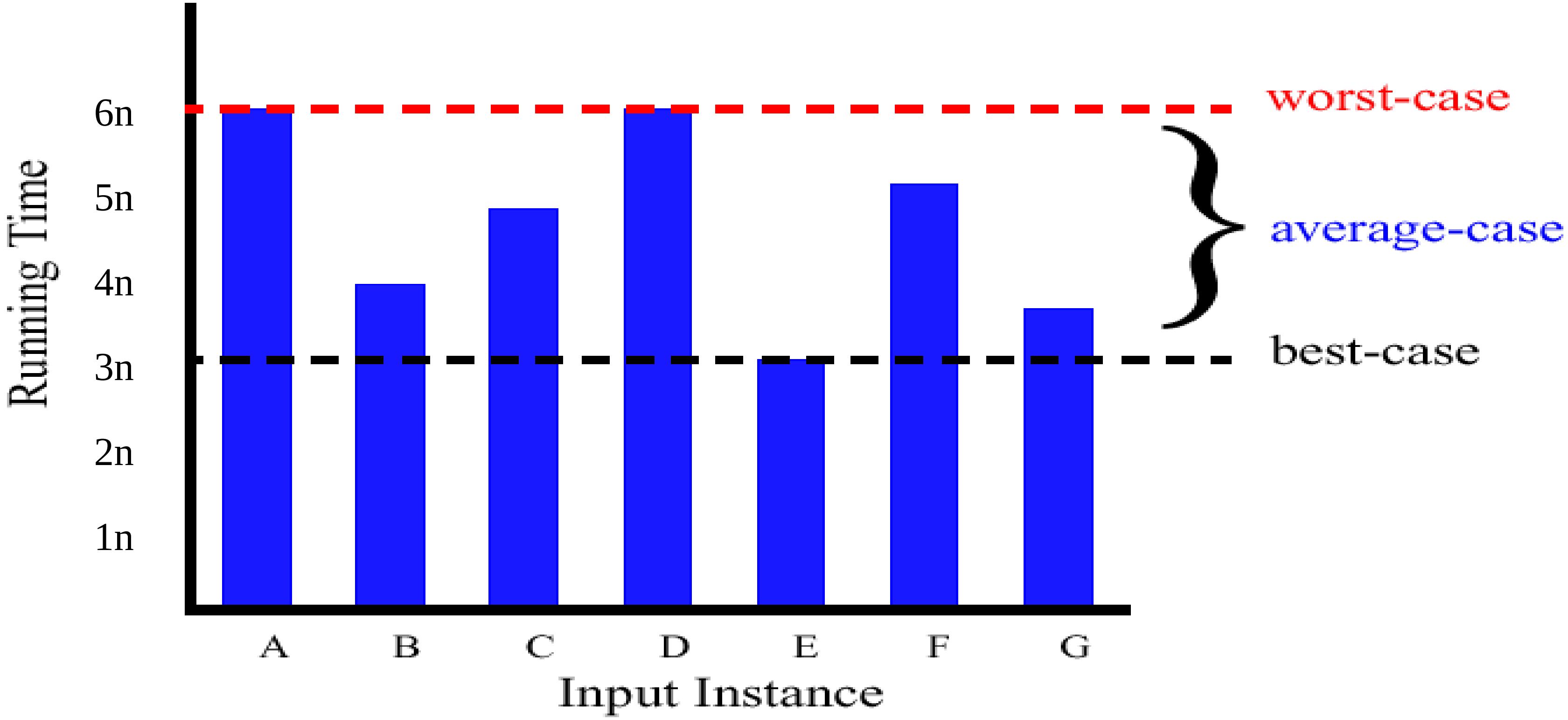
$c_9 \cdot n + c_{10} + c_5 \cdot [n(n+1)/2] - 1 + c_{11} \cdot n(n-1)/2$

$= c_{12} \cdot n^2 + c_{13} \cdot n + c_{14}$

$= f_1(n^2) + f_2(n^1) + f_3(n^0)$

# Best/Worst/Average Case (2)

- For a specific size of input *n*, investigate running times for different input instances:



# Insertion Sort Analysis

- Is insertion sort a fast sorting algorithm?
  - Moderately so, for small n
  - Not at all, for large n

# Merge Sort (Divide and Conquer)

**MERGE-SORT** $A[1 \,..\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$.
3. "*Merge*" the 2 sorted lists.

***Key subroutine:*** **MERGE**

# Merge Sort Example

20
13
7
2

12
11
9
1

1
2
7
9
11
12
13
20

# Analysis of Merge Sort

| | |
|---|---|
| $T(n)$ | MERGE-SORT $A[1\,..\,n]$ |
| $\Theta(1)$ | 1. If $n = 1$, done. |
| $2T(n/2)$ | 2. Recursively sort $A[\,1\,..\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil{+}1\,..\,n\,]$. |
| $\Theta(n)$ | 3. *"Merge"* the 2 sorted lists |

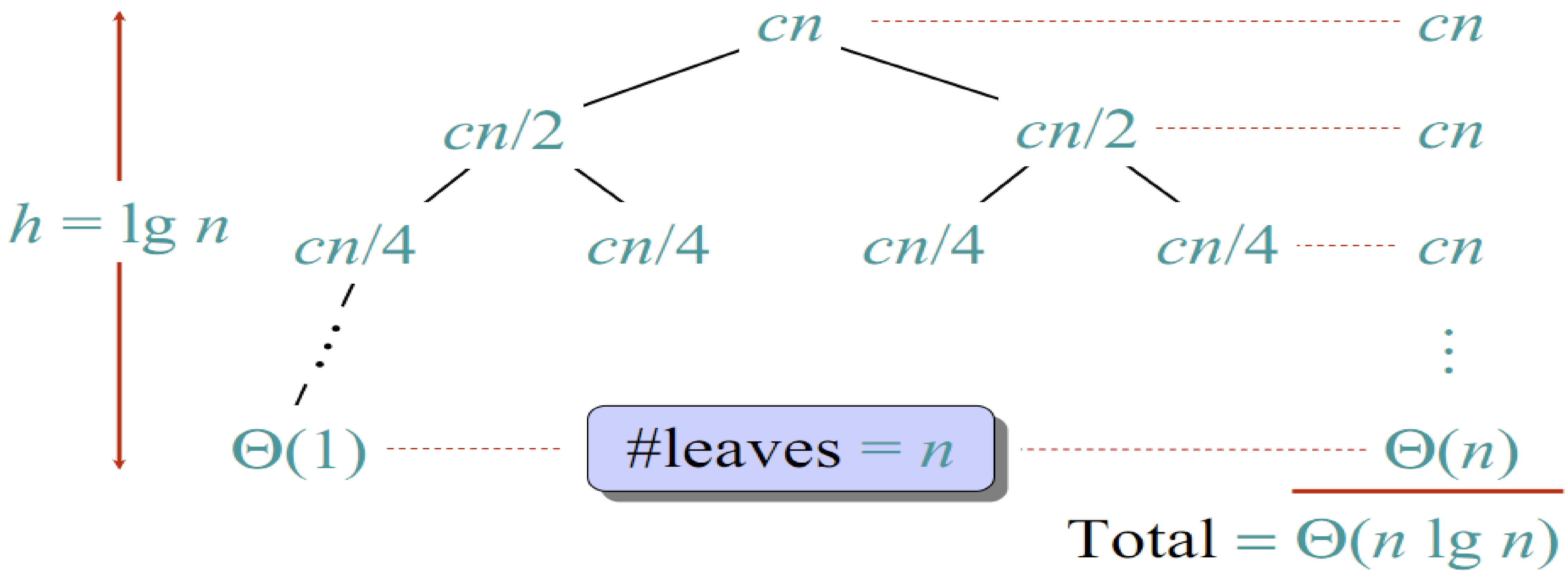
***Abuse***

***Sloppiness:*** Should be $T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor)$, but it turns out not to matter asymptotically.

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

# Visual Representation of the Recurrence for Merge Sort

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



# Conclusion

- $\Theta(n\lg n)$ grows more slowly than $\Theta(n^2)$
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n >= 3$

# Quiz 1

Marks : 10

Time :10 mins

## Question 1:

Design flowchart and calculate the running time of the given pseudo-code.

```
Start
    Total=0
    For i=1- n
          Input x
          If x>100
            then total=total+1
         end If
        ++i
     Print total
End
```