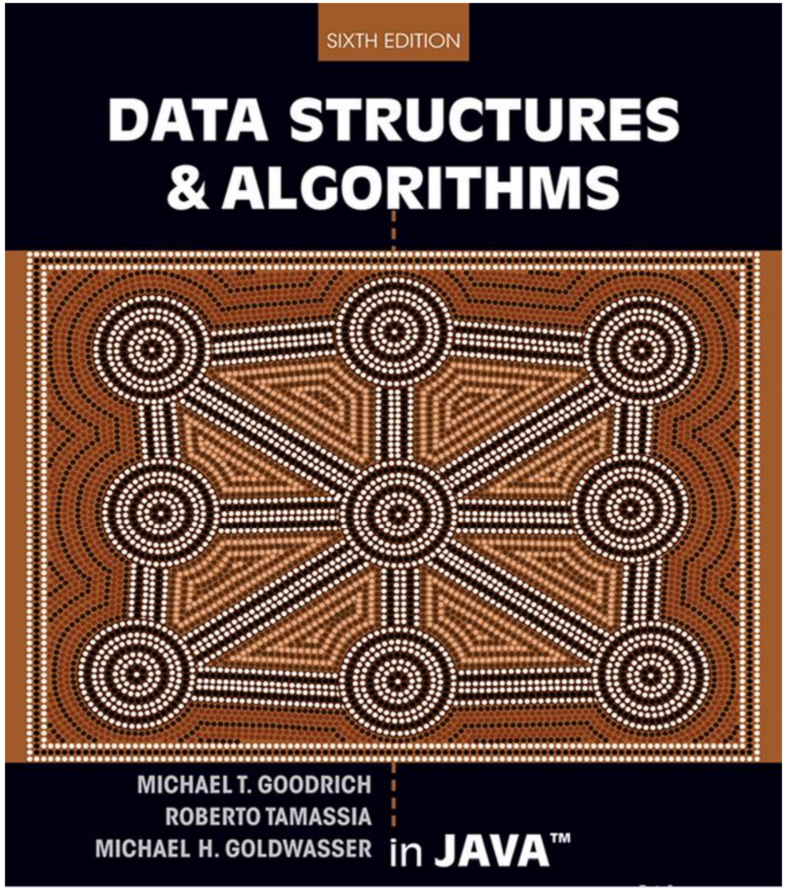
# DATA STRUCTURES & ALGORITHMS




Data Structures & Algorithms

## Lecture 3

Algorithm Analysis
And Recursion Algorithm

BTEC
Alliance with FPT Education

Pearson

# Topics

- Recursion Algorithm
- Illustrative Examples
- Review of Algorithm Concepts
- Algorithmic Performance
- Analysis of Algorithms
- General Rules and Case Analysis

# Review of Algorithm Concept

- An ***algorithm*** is a set of instructions to be followed to solve a problem.
  - There can be more than one solution (more than one algorithm) to solve a given problem.
  - An algorithm can be implemented using different programming languages on different platforms.
- An algorithm must be correct. It should correctly solve the problem.
  - e.g. For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.
- Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm.

# Algorithmic Performance

There are *two aspects* of algorithmic performance:

- Time
  - ✓ Instructions take time.
  - ✓ How fast does the algorithm perform?
  - ✓ What affects its runtime?
- Space
  - ✓ Data structures take space
  - ✓ What kind of data structures can be used?
  - ✓ How does choice of data structure affect the runtime?

➢ We will focus on time:

- How to estimate the time required for an algorithm
- How to reduce the time required

# Analysis of Algorithms

- ***Analysis of Algorithms*** is the area of computer science that provides tools to analyze the efficiency of different methods of solutions.
- How do we compare the time efficiency of two algorithms that solve the same problem?

  ***Naive Approach***: implement these algorithms in a programming language (Java), and run them to compare their time requirements. Comparing the programs (instead of algorithms) has difficulties.

  - *How are the algorithms coded?*
    - ✓ Comparing running times means comparing the implementations.
    - ✓ We should not compare implementations, because they are sensitive to programming style that may cloud the issue of which algorithm is inherently more efficient.
  - *What computer should we use?*
    - ✓ We should compare the efficiency of the algorithms independently of a particular computer.
  - *What data should the program use?*
    - ✓ Any analysis must be independent of specific data.

# Analysis of Algorithms

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

# The Execution Time of Algorithms (1/4)

- Each operation in an algorithm (or a program) has a cost.

    ➔ Each operation takes a certain of time.

count = count + 1; ➔ take a certain amount of time, but it is constant

***A sequence of operations:***

count = count + 1; Cost: $c_1$

sum = sum + count; Cost: $c_2$

➔ Total Cost = $c_1 + c_2$

# The Execution Time of Algorithms (2/4)

*Example: Simple If-Statement*

| | **Cost** | **Times** |
|---|---|---|
| `if (n < 0)` | c1 | 1 |
| `    absval = -n` | c2 | 1 |
| `else` | | |
| `    absval = n;` | c3 | 1 |

Total Cost <= c1 + max(c2,c3)

# The Execution Time of Algorithms (3/4)

*Example: Simple Loop*

| Code | Cost | Times |
|---|---|---|
| `i = 1;` | c1 | 1 |
| `sum = 0;` | c2 | 1 |
| `while (i <= n) {` | c3 | n+1 |
| `    i = i + 1;` | c4 | n |
| `    sum = sum + i;` | c5 | n |
| `}` | | |

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

# The Execution Time of Algorithms (4/4)

*Example: Nested Loop*

| Code | Cost | Times |
|---|---|---|
| `i=1;` | c1 | 1 |
| `sum = 0;` | c2 | 1 |
| `while (i <= n) {` | c3 | n+1 |
| `j=1;` | c4 | n |
| `while (j <= n) {` | c5 | n*(n+1) |
| `sum = sum + i;` | c6 | n*n |
| `j = j + 1;` | c7 | n*n |
| `}` | | |
| `i = i +1;` | c8 | n |
| `}` | | |

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to $n^2$

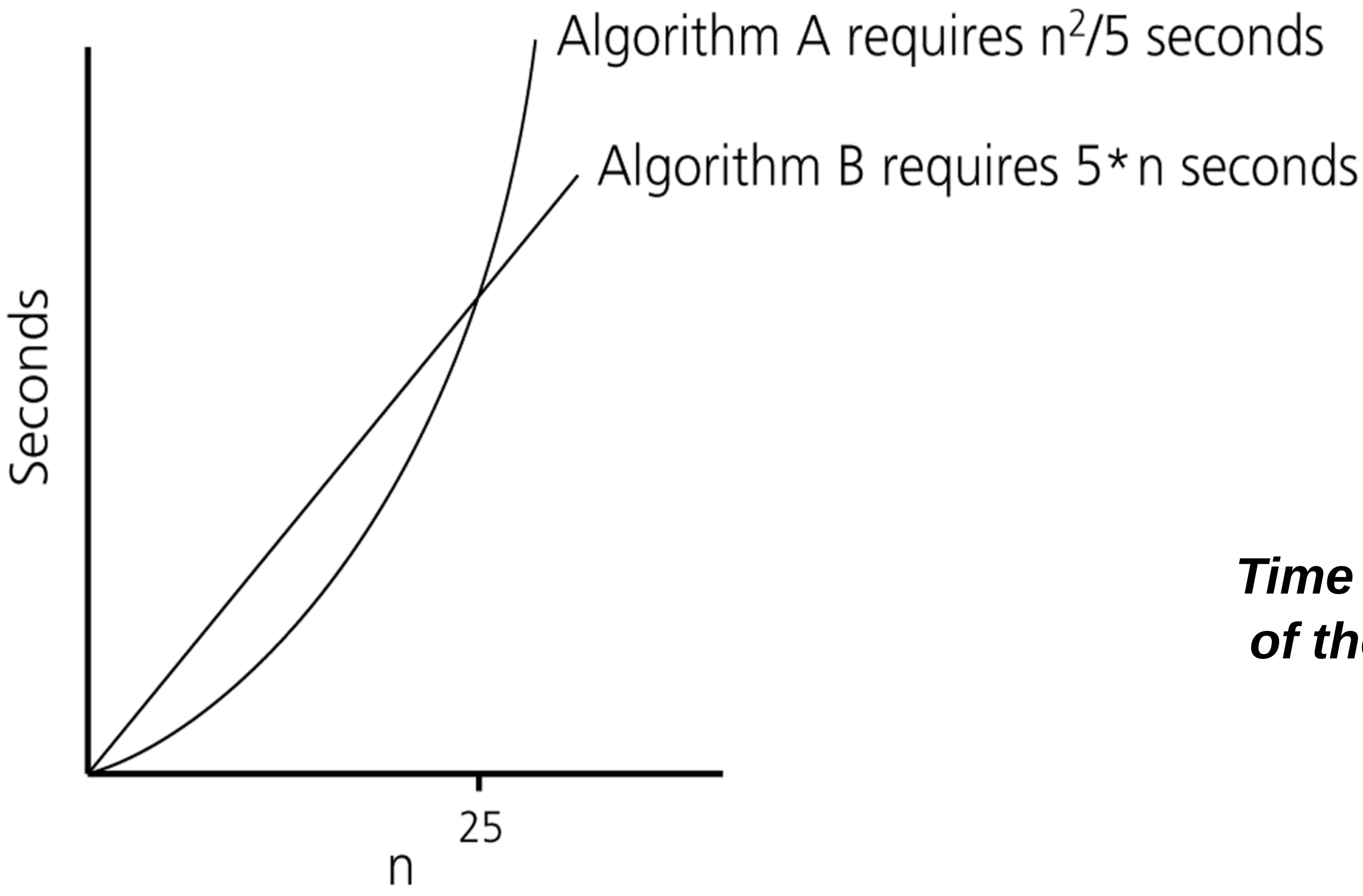
# General Rules for Estimation

- **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- **Nested Loops**: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.
- **Consecutive Statements:** Just add the running times of those consecutive statements.
- **If/Else**: Never more than the running time of the test plus the larger of running times of S1 and S2.

# Algorithm Growth Rates

- We measure an algorithm's time requirement as a function of the *problem size*.
  - Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number disks for towers of hanoi.
- So, for instance, we say that (if the problem size is n)
  - Algorithm A requires $\mathbf{5*n^2}$ time units to solve a problem of size n.
  - Algorithm B requires $\mathbf{7*n}$ time units to solve a problem of size n.
- The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.
  - Algorithm A requires time proportional to $\mathbf{n^2}$.
  - Algorithm B requires time proportional to **n**.
- An algorithm's proportional time requirement is known as ***growth rate***.
- We can compare the efficiency of two algorithms by comparing their growth rates.

# Algorithm Growth Rates




***Time requirements as a function of the problem size n***

# Common Growth Rates

| Function | Growth Rate Name |
| --- | --- |
| $c$ | Constant |
| $log\ N$ | Logarithmic |
| $log^2 N$ | Log-squared |
| $N$ | Linear |
| $N\ log\ N$ | |
| $N^2$ | Quadratic |
| $N^3$ | Cubic |
| $2^N$ | Exponential |

**Figure 6.1**

Running times for small inputs

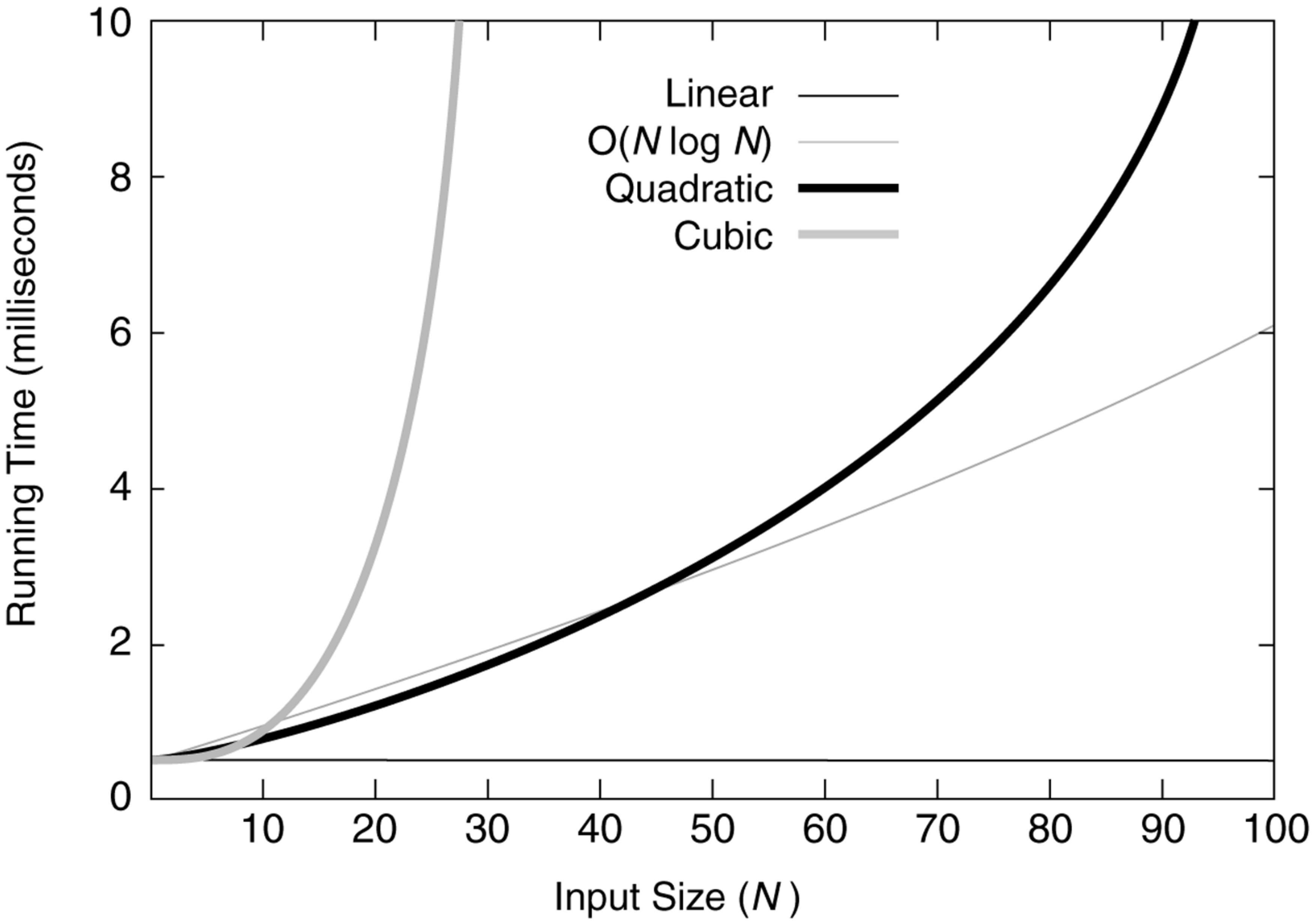
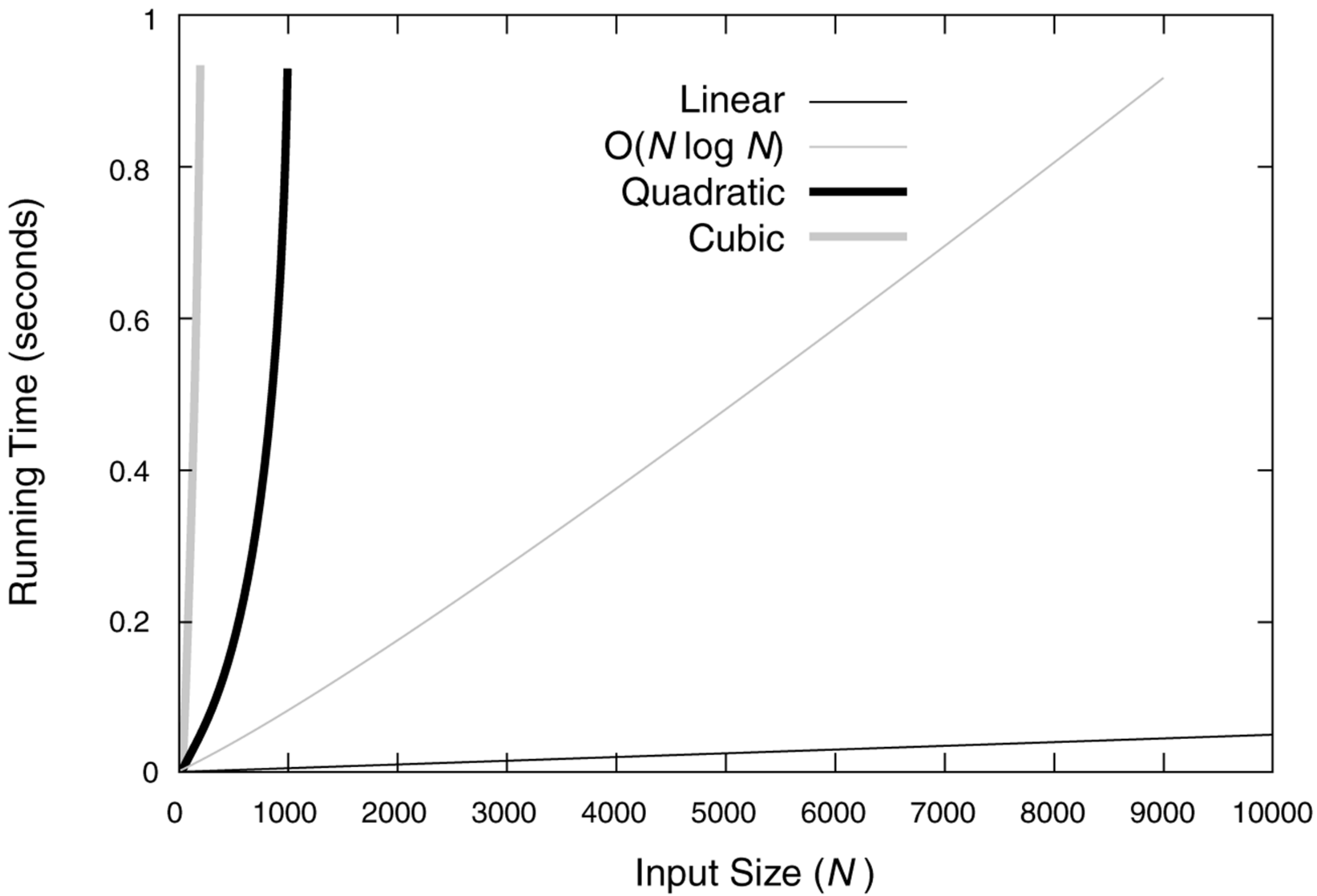


**Figure 6.2**

Running times for moderate inputs



# Order-of-Magnitude Analysis and Big O Notation

- If *Algorithm A requires time proportional to f(n),* Algorithm A is said to be **order f(n),** and it is denoted as **O(f(n)).**
- The **function f(n)** is called the algorithm's **growth-rate function**.
- Since the capital O is used in the notation, this notation is called the **Big O notation**.
- If Algorithm A requires time proportional to $\mathbf{n^2}$, it is $\mathbf{O(n^2)}$**.**
- If Algorithm A requires time proportional to **n**, it is **O(n).**

# Definition of the Order of an Algorithm

***Definition:***

Algorithm A is order f(n) – denoted as O(f(n)) – if constants k and $n_0$ exist such that A requires no more than k*f(n) time units to solve a problem of size $n \geq n_0$.

The requirement of $\mathbf{n \geq n_0}$ in the definition of O(f(n)) formalizes the notion of sufficiently large problems.

- In general, many values of k and n can satisfy this definition.

# Order of an Algorithm

- If an algorithm requires $n^2 - 3*n+10$ seconds to solve a problem size n. If constants k and $n_0$ exist such that

  $$k*n^2 > n^2 - 3*n+10 \qquad \text{for all } n \geq n_0 .$$

  the algorithm is order $n^2$ (In fact, k is 3 and $n_0$ is 2)

  $$3*n^2 > n^2 - 3*n+10 \qquad \text{for all } n \geq 2 .$$

  Thus, the algorithm requires no more than $k*n^2$ time units for $n \geq n_0$ ,

  So it is **$O(n^2)$**

# Order of an Algorithm

- If an algorithm requires $n^2 – 3*n+10$ seconds to solve a problem size n. If constants k and $n_0$ exist such that

  $$k*n^2 > n^2 – 3*n+10 \qquad \text{for all } n \geq n_0 .$$

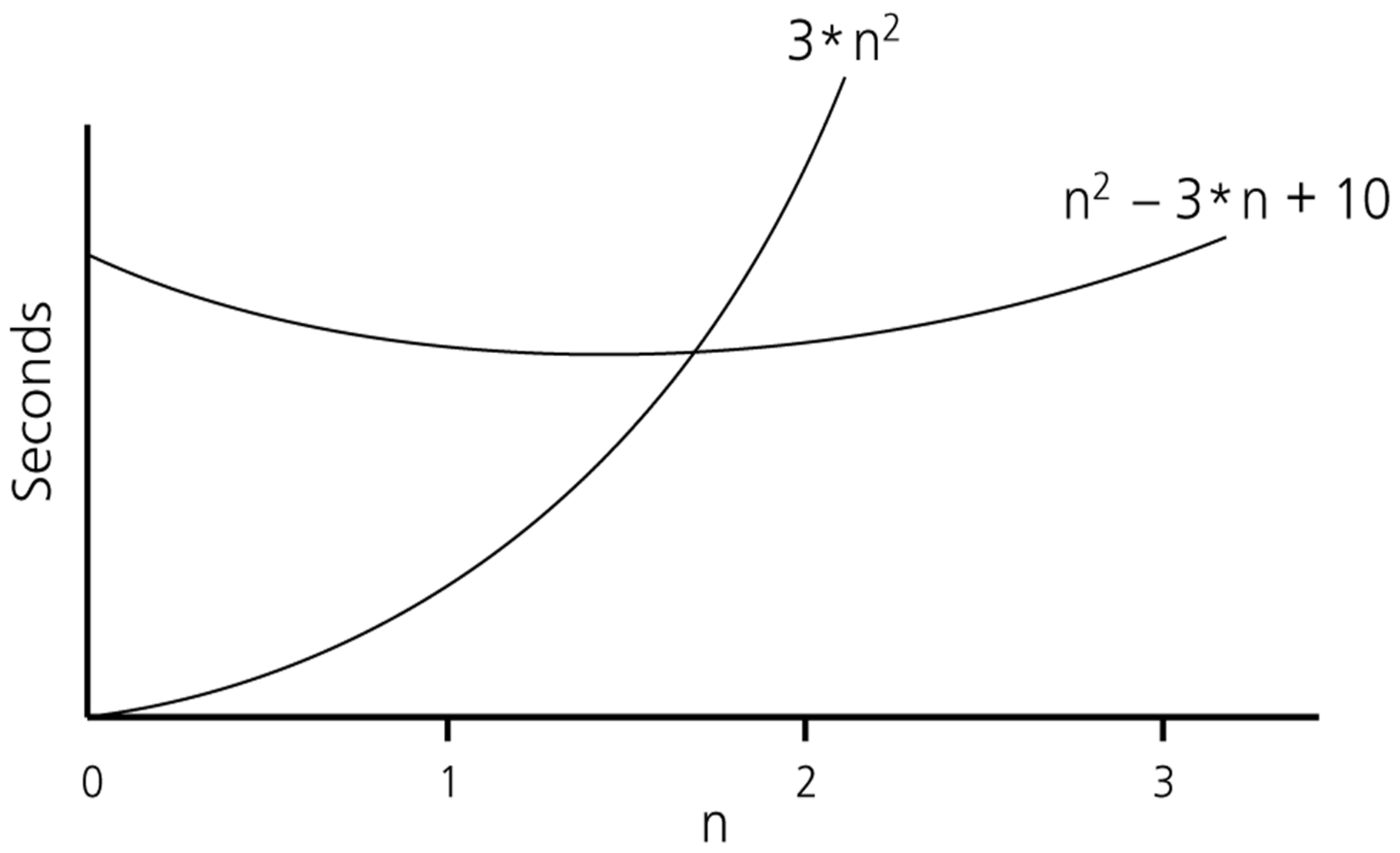
  the algorithm is order $n^2$ (In fact, k is 3 and $n_0$ is 2)

  $$3*n^2 > n^2 – 3*n+10 \qquad \text{for all } n \geq 2 .$$

  Thus, the algorithm requires no more than $k*n^2$ time units for $n \geq n_0$ ,

  So it is **O(n²)**

# Order of an Algorithm

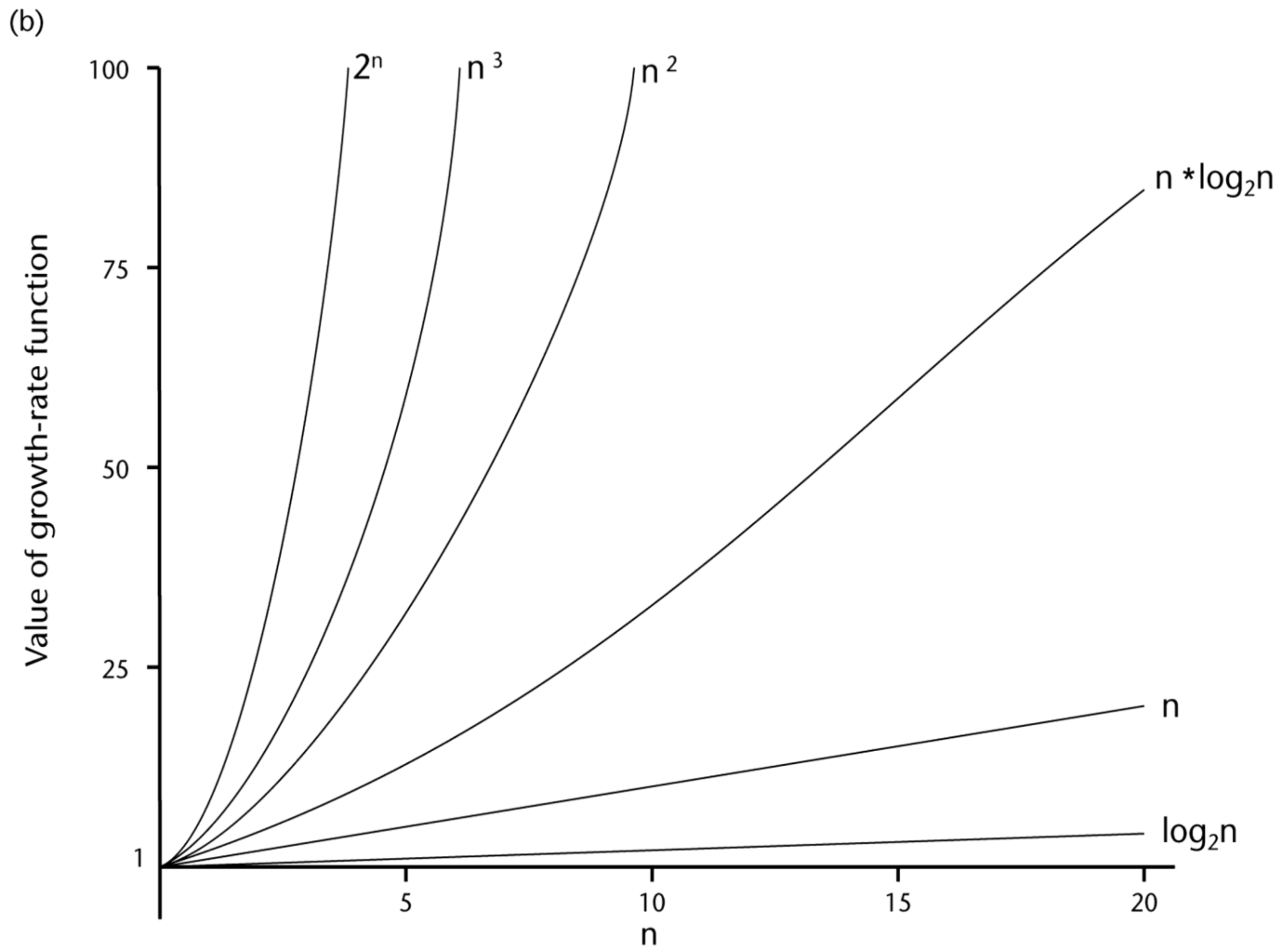


# A Comparison of Growth-Rate Functions

(a)

| Function | n = 10 | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $\log_2 n$ | 3 | 6 | 9 | 13 | 16 | 19 |
| n | 10 | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| $n * \log_2 n$ | 30 | 664 | 9,965 | $10^5$ | $10^6$ | $10^7$ |
| $n^2$ | $10^2$ | $10^4$ | $10^6$ | $10^8$ | $10^{10}$ | $10^{12}$ |
| $n^3$ | $10^3$ | $10^6$ | $10^9$ | $10^{12}$ | $10^{15}$ | $10^{18}$ |
| $2^n$ | $10^3$ | $10^{30}$ | $10^{301}$ | $10^{3,010}$ | $10^{30,103}$ | $10^{301,030}$ |

# A Comparison of Growth-Rate Functions



# Growth-Rate Functions

**$O(1)$** Time requirement is **constant**, and it is independent of the problem's size.

**$O(\log_2 n)$** Time requirement for a **logarithmic** algorithm increases increases slowly as the problem size increases.

**$O(n)$** Time requirement for a **linear** algorithm increases directly with the size of the problem.

**$O(n*\log_2 n)$** Time requirement for a **$n*\log_2 n$** algorithm increases more rapidly than a linear algorithm.

**$O(n^2)$** Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.

**$O(n^3)$** Time requirement for a c**ubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.

**$O(2^n)$** As the size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

# Growth-Rate Functions

- If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?
- If its order is:

**O(1)** → $T(n) = 1$ second

**O(log$_2$n)** → $T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds

**O(n)** → $T(n) = (1*16) / 8 = 2$ seconds

**O(n*log$_2$n)** → $T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds

**O(n$^2$)** → $T(n) = (1*16^2) / 8^2 = 4$ seconds

**O(n$^3$)** → $T(n) = (1*16^3) / 8^3 = 8$ seconds

**O(2$^n$)** → $T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds $= 256$ seconds

# Properties of Growth-Rate Functions

1. *We can ignore low-order terms in an algorithm's growth-rate function.*
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.

2. *We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.*
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

3. *$O(f(n)) + O(g(n)) = O(f(n)+g(n))$*
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n^2)$ ➔ So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

# Some Mathematical Facts

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + \ldots + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

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

# Growth-Rate Functions – Example1

| | Cost | Times |
|---|---|---|
| `i = 1;` | c1 | 1 |
| `sum = 0;` | c2 | 1 |
| `while (i <= n) {` | c3 | n+1 |
| `    i = i + 1;` | c4 | n |
| `    sum = sum + i;` | c5 | n |
| `}` | | |

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*c5
= (c3+c4+c5)*n + (c1+c2+c3)
= a*n + b

➔ So, the growth-rate function for this algorithm is **O(n)**

# Growth-Rate Functions – Example2

| | **Cost** | **Times** |
|---|---|---|
| `i=1;` | c1 | 1 |
| `sum = 0;` | c2 | 1 |
| `while (i <= n) {` | c3 | n+1 |
| `    j=1;` | c4 | n |
| `    while (j <= n) {` | c5 | n*(n+1) |
| `        sum = sum + i;` | c6 | n*n |
| `        j = j + 1;` | c7 | n*n |
| `  }` | | |
| `  i = i +1;` | c8 | n |
| `}` | | |

$$T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8$$

$$= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)$$

$$= a*n^2 + b*n + c$$

➔ So, the growth-rate function for this algorithm is **$O(n^2)$**

# Growth-Rate Functions – Example3

| | **Cost** | **Times** |
|---|---|---|
| `for (i=1; i<=n; i++)` | c1 | n+1 |
| `  for (j=1; j<=i; j++)` | c2 | $\sum_{j=1}^{n}(j+1)$ |
| `    for (k=1; k<=j; k++)` | c3 | $\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)$ |
| `      x=x+1;` | c4 | $\sum_{j=1}^{n}\sum_{k=1}^{j}k$ |

$$T(n) = c1*(n+1) + c2*\left(\sum_{j=1}^{n}(j+1)\right) + c3*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)\right) + c4*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}k\right)$$

$$= a*n^3 + b*n^2 + c*n + d$$

➔ So, the growth-rate function for this algorithm is **$O(n^3)$**

# Growth-Rate Functions – Recursive Algorithms

| | |
|---|---|
| `void hanoi(int n, char source, char dest, char spare) {` | **Cost** |
| `  if (n > 0) {` | c1 |
| `    hanoi(n-1, source, spare, dest);` | c2 |
| `     "Move top disk from pole " << source` | c3 |
| `            << " to pole " << dest << endl;` | |
| `    hanoi(n-1, spare, dest, source);` | c4 |
| `} }` | |

- The time-complexity function T(n) of a recursive algorithm is defined in terms of itself, and this is known as **recurrence equation** for T(n).
- To find the growth-rate function for a recursive algorithm, we have to solve its recurrence relation.

# Growth-Rate Functions – Hanoi Towers

- What is the cost of hanoi(n,'A','B','C')?

when n=0

$T(0) = c1$

when n>0

$$T(n) = c1 + c2 + T(n-1) + c3 + c4 + T(n-1)$$
$$= 2*T(n-1) + (c1+c2+c3+c4)$$
$$= \mathbf{2*T(n-1) + c}$$

🡐 recurrence equation for the growth-rate function of hanoi-towers algorithm

- Now, we have to solve this recurrence equation to find the growth-rate function of hanoi-towers algorithm

# Growth-Rate Functions – Hanoi Towers

- There are many methods to solve recurrence equations, but we will use a simple method known as *repeated substitutions*.

$T(n) = 2*T(n-1) + c$

$= 2 * (2*T(n-2)+c) + c$

$= 2 * (2* (2*T(n-3)+c) + c) + c$

$= 2^3 * T(n-3) + (2^2+2^1+2^0)*c$ (assuming n>2)

when substitution repeated i-1$^{th}$ times

$= 2^i * T(n-i) + (2^{i-1}+ ... +2^1+2^0)*c$

when i=n

$= 2^n * T(0) + (2^{n-1}+ ... +2^1+2^0)*c$

$= 2^n * c1 + (\quad)*c$

$= 2^n * c1 + ( 2^n-1 )*c = 2^n*(c1+c) – c$ ➔ So, the growth rate function is **O($2^n$)**

# What to Analyze

- An algorithm can require different times to solve different problems of the same size.
  - Eg. Searching an item in a list of n elements using sequential search. ➔ Cost: 1,2,...,n
- ***Worst-Case Analysis*** –The maximum amount of time that an algorithm require to solve a problem of size n.
  - This gives an upper bound for the time complexity of an algorithm.
  - Normally, we try to find worst-case behavior of an algorithm.
- ***Best-Case Analysis*** –The minimum amount of time that an algorithm require to solve a problem of size n.
  - The best case behavior of an algorithm is NOT so useful.
- ***Average-Case Analysis*** –The average amount of time that an algorithm require to solve a problem of size n.
  - Sometimes, it is difficult to find the average-case behavior of an algorithm.
  - We have to look at all possible data organizations of a given size n, and their distribution probabilities of these organizations.
  - ***Worst-case analysis is more common than average-case analysis.***

# What is Important?

- An array-based list retrieve operation is O(1), a linked-list-based list retrieve operation is O(n).
- But insert and delete operations are much easier on a linked-list-based list implementation.

  ➔ When selecting the implementation of an Abstract Data Type (ADT), we have to consider how frequently particular ADT operations occur in a given application.

- If the problem size is always small, we can probably ignore the algorithm’s efficiency.
  - In this case, we should choose the simplest algorithm.

# What is Important?

- We have to weigh the trade-offs between an algorithm’s time requirement and its memory requirements.
- We have to compare algorithms for both style and efficiency.
  - The analysis should focus on gross differences in efficiency and not reward coding tricks that save small amount of time.
  - That is, there is no need for coding tricks if the gain is not too much.
  - Easily understandable program is also important.
- Order-of-magnitude analysis focuses on large problems.

# Sequential Search

```
int sequentialSearch(const int a[], int item, int n){
  for (int i = 0; i < n && a[i]!= item; i++);
  if (i == n)
        return -1;
  return i;
}
```

***Unsuccessful Search:*** ➔ O(n)

***Successful Search:***

**Best-Case:** *item* is in the first location of the array ➔O(1)

**Worst-Case:** *item* is in the last location of the array ➔O(n)

**Average-Case**: The number of key comparisons 1, 2, ..., n

➔ O(n)

# Sequential Search

```
int binarySearch(int a[], int size, int x) {
    int low =0;
    int high = size -1;
    int mid;              // mid will be the index of
                          // target when it's found.
    while (low <= high) {
      mid = (low + high)/2;
      if (a[mid] < x)
          low = mid + 1;
      else if (a[mid] > x)
          high  = mid - 1;
      else
          return mid;
    }
    return -1;
}
```

# Binary Search – Analysis

- For an unsuccessful search:
  - The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$

    ➔ $O(\log_2 n)$

- For a successful search:
  - ***Best-Case:*** The number of iterations is 1. ➔ $O(1)$
  - ***Worst-Case:*** The number of iterations is $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$
  - ***Average-Case:*** The avg. # of iterations < $\log_2 n$ ➔ $O(\log_2 n)$

```
0   1   2   3   4   5   6   7
3   2   3   1   3   2   3   4
```

← an array with size 8

← # of iterations

The average # of iterations = $21/8 < \log_2 8$

# How much better is $O(log_2n)$?

| $n$ | $O(log_2n)$ |
|---|---|
| 16 | 4 |
| 64 | 6 |
| 256 | 8 |
| 1024 (1KB) | 10 |
| 16,384 | 14 |
| 131,072 | 17 |
| 262,144 | 18 |
| 524,288 | 19 |
| 1,048,576 (1MB) | 20 |
| 1,073,741,824 (1GB) | 30 |

# Recursive Thinking (1/2)

- ***Recursion*** is:
  - A *problem-solving* ***approach***, that can ...
  - Generate *simple solutions* to ...
  - *Certain kinds* of problems that ...
  - Would be *difficult to solve in other ways*
- Recursion *splits a problem*:
  - Into one or more *simpler versions of* ***itself***

# Recursive Thinking (2/2)

- An Example: *Strategy for processing nested dolls*

*1.if there is only one doll*

*2.do what it needed for it*

*else*

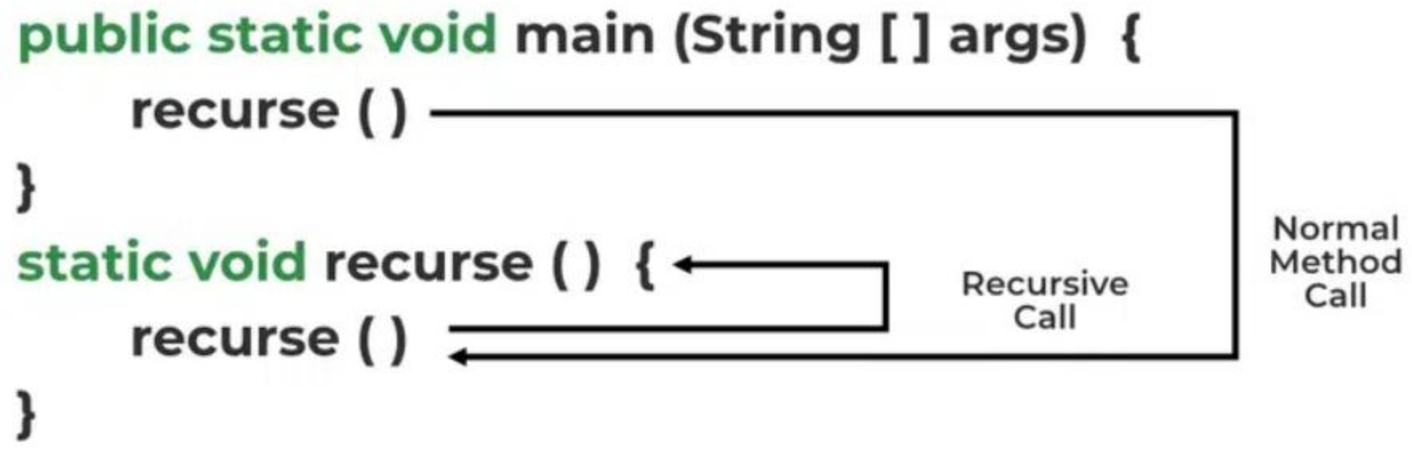
*3.do what is needed for the outer doll*

*4.Process the inner nest in the same way*

FIGURE 7.1
A Set of Nested Wooden Figures

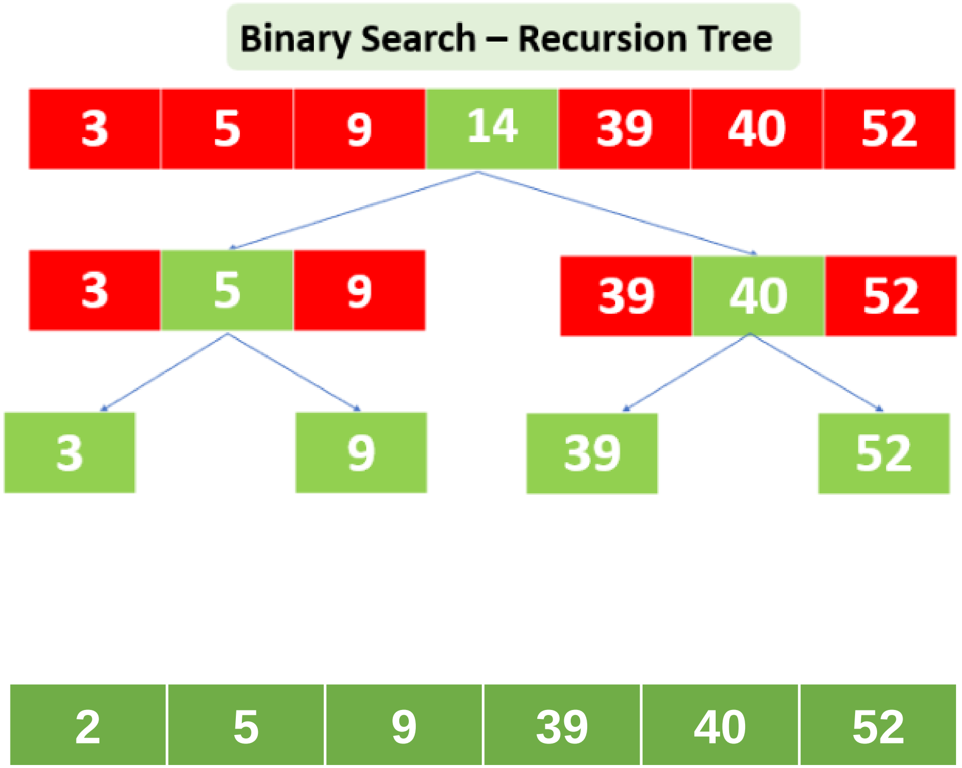
# Definitions

- At its core, recursion is a programming technique that involves a function calling itself until a specific condition is met.



# Definitions

- Recursive functions can be employed to address various problems: **searching, sorting**, and traversing data structures like **trees** and **graphs**. They are often utilized in algorithms such as **quicksort**, **binary search**, and **depth-first search**.



# Advantages of Recursion

- Clarity and simplicity
- Reducing code duplication
- Solving complex problems
- Memory efficiency
- Flexibility

# Disadvantages of Recursion

- Performance Overhead
- Difficult to Understand and Debug
- Memory Consumption
- Limited Scalability
- Tail Recursion Optimization

# Algorithm for Recursion (1/2)

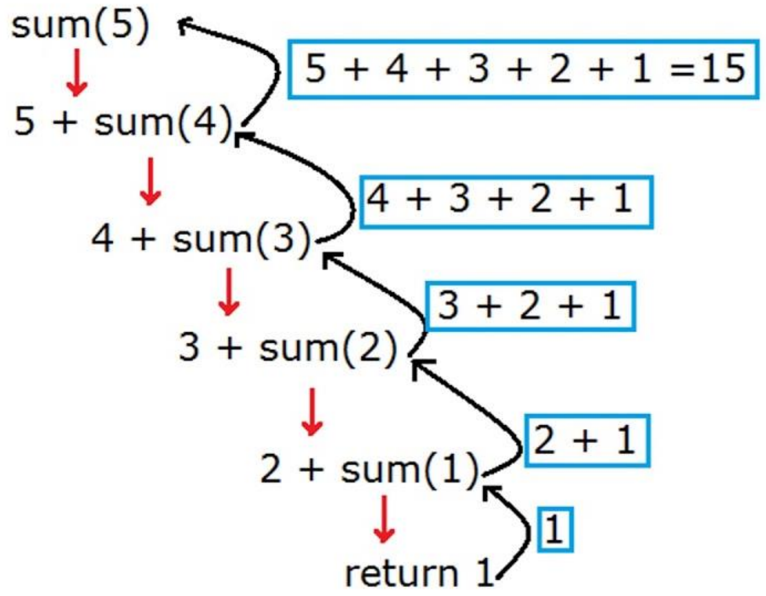
1. **Define the base case:** Identify the simplest problem that can be solved without recursion
2. **Define the recursive case:** Identify how to break the problem down into smaller sub-problems that can be solved recursively
3. **Call the function recursively:** Invoke the function again with the smaller sub-problem(s) as input

# Algorithm for Recursion (2/2)

4. **Combine the results:** Use the results of the recursive calls to solve the original problem
5. **Return the solution:** Return the final solution to the original problem

# Example

Using **recursion** to solve the problem of calculating the **sum** *from* **1** *to* **N**:




```
int sum(int n){
        if(n == 0)
                return 0;
         return n + sum (n-1);
}
```

# Algorithm for Recursion

**Define the base case:** Identify the simplest problem that can be solved without recursion.

```
int sum(int n){
        if(n == 0)          // The base case when N is equal to 0
                return 0; //  Returns 0 and dont call Recursion functions
        return n + sum (n-1);
}
```

# Algorithm for Recursion

**Define the recursive case:** Identify how to break the problem down into smaller sub-problems that can be solved recursively.

```
int sum(int n){
        if(n == 0)
                return 0;
        return n + sum (n-1);
}
```

For this exercise, we can identify the **recursive case** as the natural numbers preceding **N**. By iterating from **N** *down to* **1**, we obtain the **sum** of natural numbers from 1 to N.

# Algorithm for Recursion

**Call the function recursively:** Invoke the function again with the smaller sub-problem(s) as input.

```
int sum(int n){
        if(n == 0)
                return 0;
        return n + sum (n-1);
}
```

sum(5)

5 + sum(4)

4 + sum(3)

3 + sum(2)

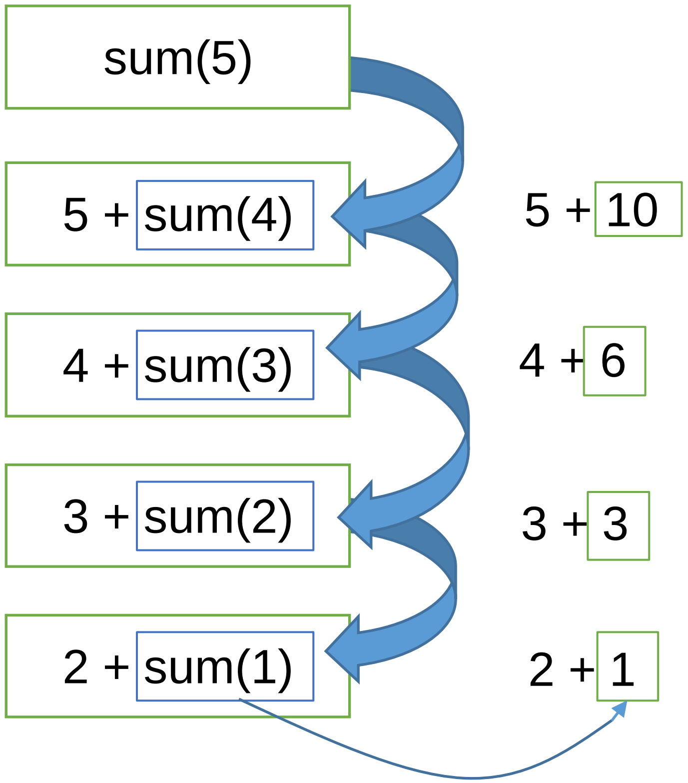
2 + sum(1)

Recursively call the function again and pass the parameter as the natural **number preceding N**, and the recursion will calculate automatically.

# Algorithm for Recursion

**Combine the results:** Use the results of the recursive calls to solve the original problem.

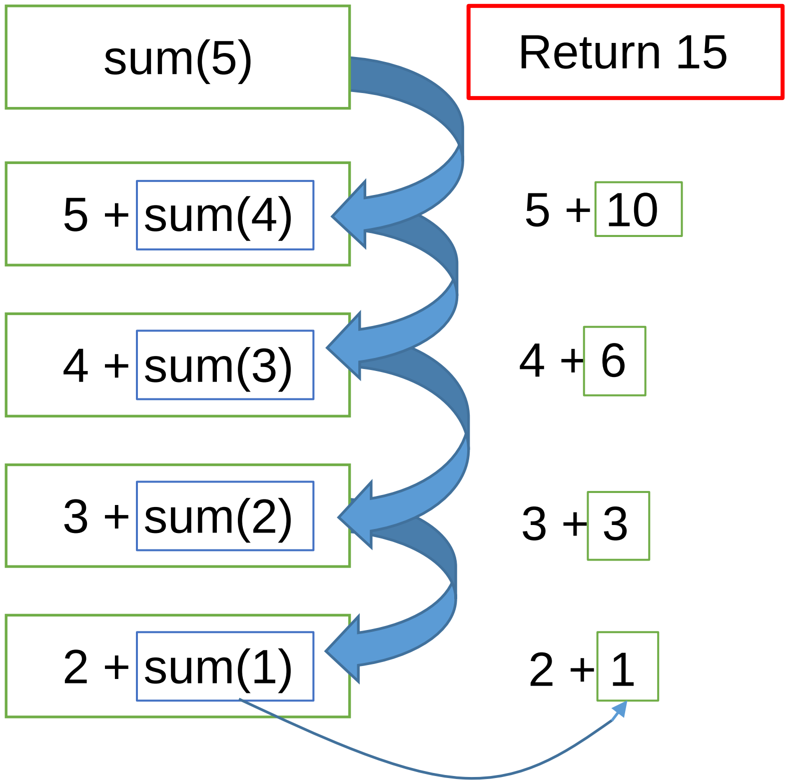
```
int sum(int n){
        if(n == 0)
                return 0;
        return n + sum (n-1);
}
```



# Algorithm for Recursion

**Return the solution:** Return the final solution to the original problem.

```
int sum(int n){
        if(n == 0)
                return 0;
         return n + sum (n-1);
}
```



# Difference between recursion and iteration

- There are some major differences between recursion and iteration, those are:

| SR No | Recursion | Iteration |
|---|---|---|
| 1 | It terminates when the base case becomes true | It particularly terminates when condition becomes false |
| 2 | Recursion is always used with functions. | Iteration is always used with loops |
| 3 | Every recursion call need extra space in the stack memory of that program | Every Iteration does not any extra space of that particular program |
| 4 | In recursion the code size is smaller | In iteration the code size is larger |

# Difference between recursion and iteration

- ***Direct recursion:***

```
public void derectRec() {
        // some codes
        derectRec();
        // some codes
}
```

*A function fun is called direct recursive if it calls the same function fun.*

# Difference between Recursion and Iteration

- ***Indirect recursion:***

```
public void derectRec1() {
        // some codes
         derectRec2(); // call method derectRec2()
        // some codes
}
public void derectRec2() {
        // some codes
         derectRec1(); // call method derectRec1()
        // some codes
}
```

*A function fun is called indirect recursive if it calls another function say funNew and funNew calls fun directly or indirectly.*

# Tail Recursion

- ***Tail recursion*** *is defined as a recursive function in which the recursive call is the last statement that is executed by the function. So basically nothing is left to execute after the recursion call.*

```
public void print(int n) {
        if(n < 0)
                return ;
        System.out.printf(" ",n);
        print(n - 1);
}
```

*The* ***tail recursive*** *functions are considered better than non-tail recursive functions as tail-recursion can be optimized by the compiler.*

**Time Complexity:** O(n)

**Auxiliary Space:** O(n)

# Time Complexity of Recursion

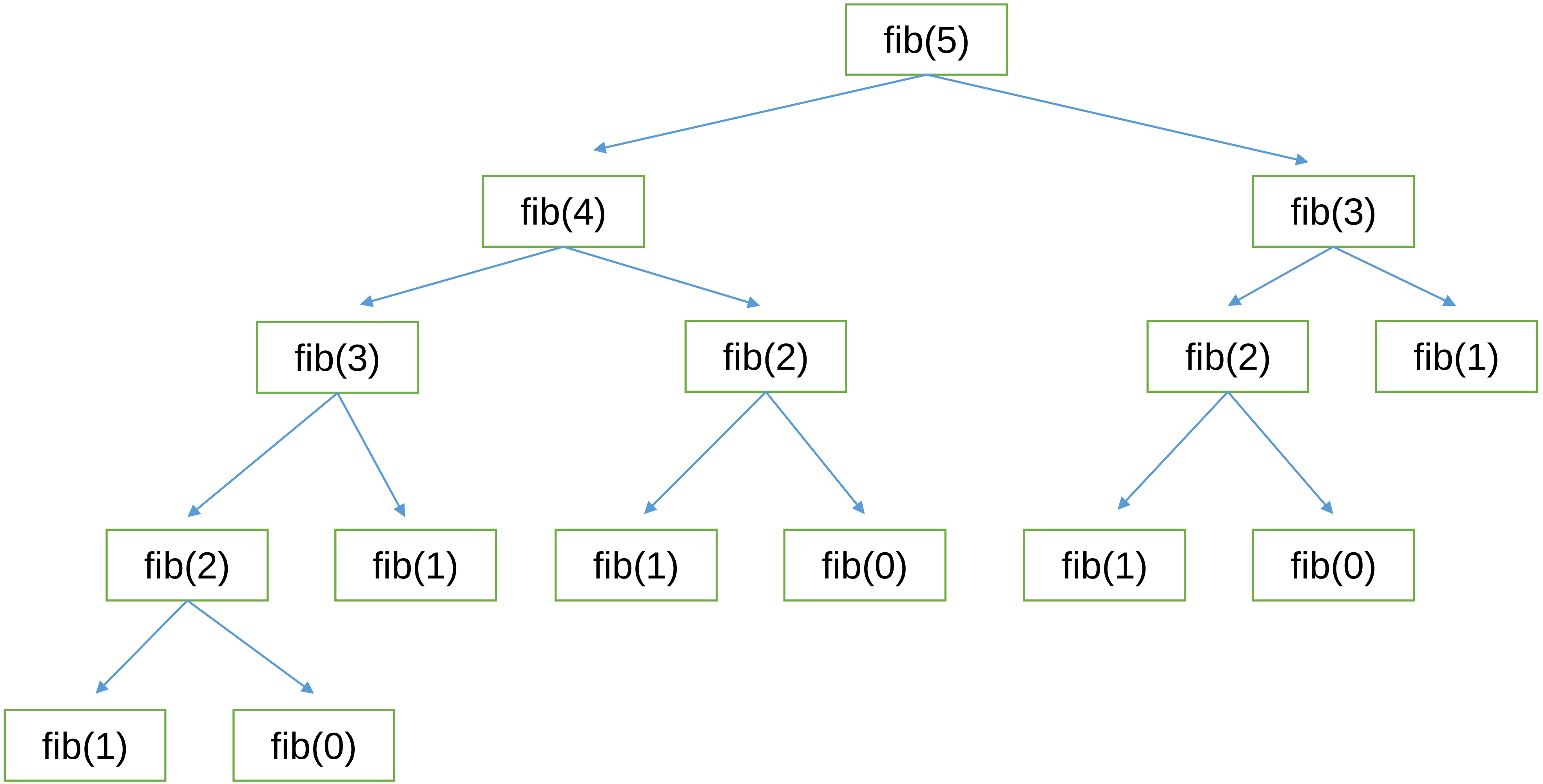
- **Recursive** algorithms in data structures and algorithms (DSA) have a time complexity that depends on the number of recursive function calls and the time complexity of each call.
- The time complexity of a recursive function can be computed by utilizing a recurrence equation, which characterizes the number of operations executed by the function as a function of the input size. This recursive equation can subsequently be solved through methods such as the **Master** theorem or the substitution method.

# Time Complexity of Recursion

- For example, consider the recursive function for computing the nth Fibonacci number:

```
public int fibonacci(int n) {
        if (n <= 1)
                return n;
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

# Simulation


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

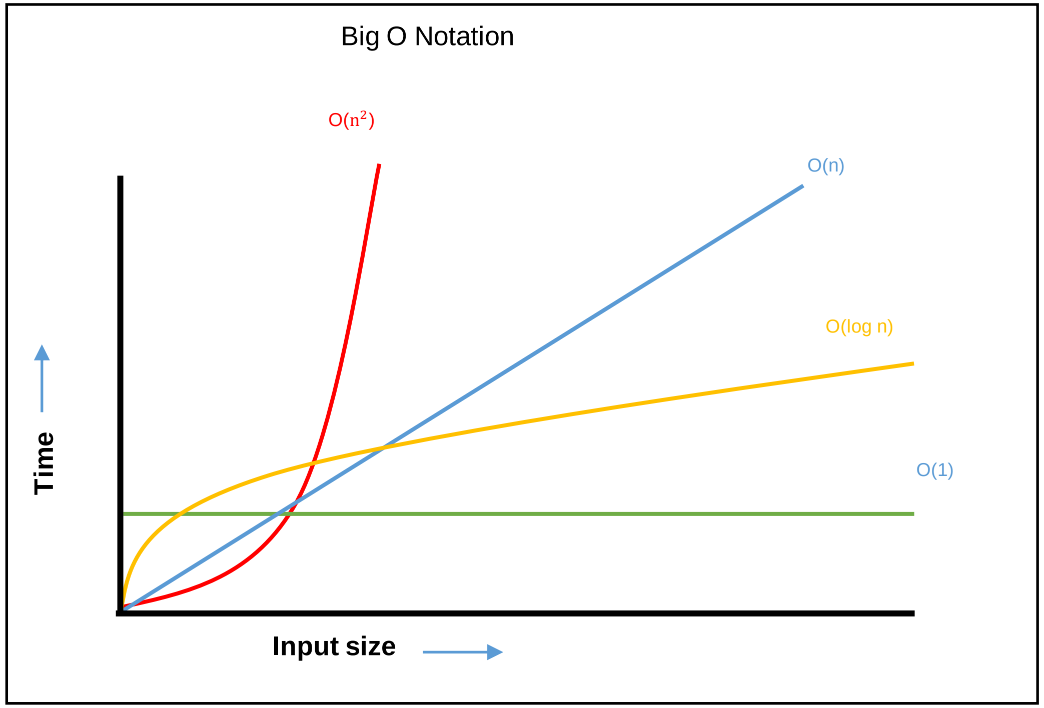
# Simulation

- fibonacci(5) calls fibonacci(4) and fibonacci(3)
- fibonacci(4) calls fibonacci(3) and fibonacci(2)
- fibonacci(3) calls fibonacci(2) and fibonacci(1)
- fibonacci(2) returns 1
- fibonacci(1) returns 1
- fibonacci(3) computes to 2 (1 + 1)
- fibonacci(2) returns 1
- fibonacci(4) computes to 3 (2 + 1)
- fibonacci(5) computes to 5 (3 + 2)

→ fib (5) = 5

# Space Complexity of Recursion function

- Recursion allocates memory for variables and frames. As **N** increases, more frames are created, possibly causing memory usage to exceed limits.
- The space complexity of the recursive Fibonacci function is O(n), where **n** is the number of recursive function calls.




*The execution time and memory space typically vary directly with the size of the input.*

# Simple Recursion

A procedure or function which calls itself is a recursive routine. Consider the following function, which computes $N! = 1 * 2 * \ldots * N$

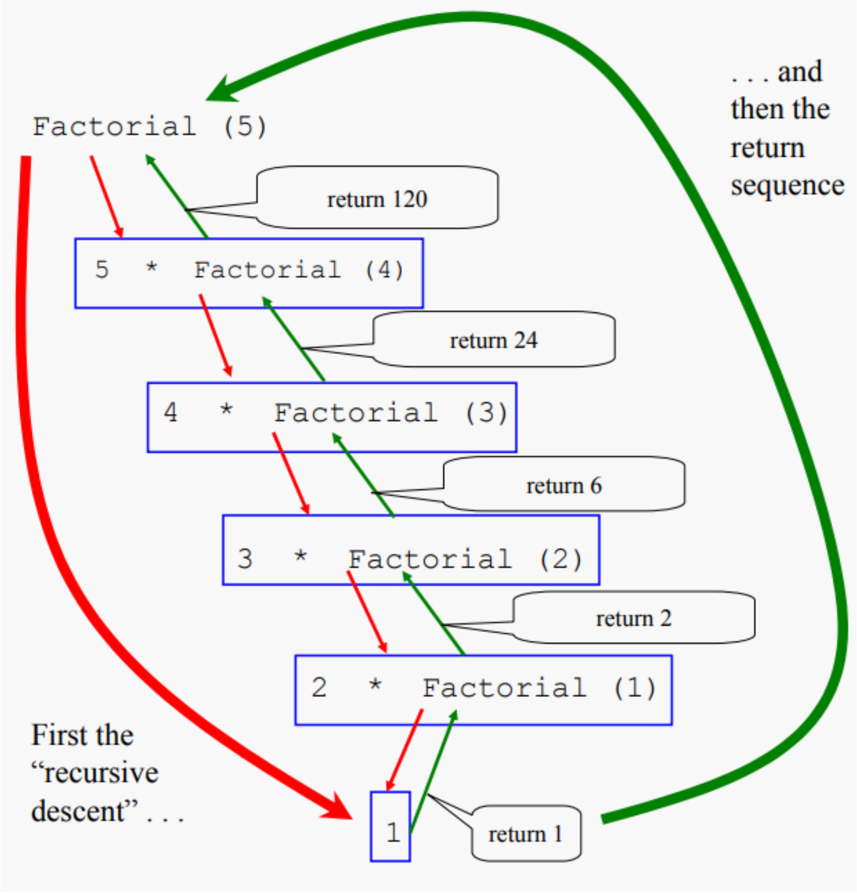
```
public static int factorial(int n) {
        int factorial = 1;
        for (int i = 1; i <= n; i++) {
                factorial *= i;
        }
        return factorial;
}
```

With this approach, code **management** becomes easier and more explicit. We can use recursion to make the code **appear more** concise and tidy.

# Simple Recursion

## Factorial using recursion

```
public static int factorial(int n) {
        if (n>0)
                return n * ( factorial (n-1) )
        return 1;
}
// factorial (5) = 120
```



# Recursive Array Summation

Here is a recursive function that takes an array of integers and computes the sum of the elements:

```
public static int sumArray(int[] X, int Start, int Stop) {
        // Error check
        if (Start > Stop || Start < 0 || Stop < 0) {
                return 0;
        } else if (Start == Stop) {
        // Base case
                return X[Stop];
        } else {
        // Recursion
                return (X[Start] + sumArray(X, Start + 1, Stop));
        }
}
```

```
                                              // return values:
SumArray(X, 0, 4)                             // == 134
    return(X[0]+SumArray(X,1,4))              // == 37 + 97
        return(X[1]+SumArray(X,2,4))          // == 14 + 83
            return(X[2]+SumArray(X,3,4) )      // == 22 + 61
                return(X[3]+SumArray(X,4,4))  // == 42 + 19
                    return X[4]               // == 19
```

*Method to traverse an array from the beginning to the end using recursion.*

# Summary

- Recursion Algorithm
- Illustrative Examples
- Review of Algorithm Concepts
- Algorithmic Performance
- Analysis of Algorithms
- General Rules and Case Analysis

# THANK YOU