

# Course – Analysis and Design of Algorithms

---

Course Instructor

Dr. Umadevi V



Department of CSE, BMSCE

Webpage: <https://sites.google.com/site/drvmadevi/>



# Unit 1:

---

- Mathematical analysis of Non-recursive and Recursive Algorithms
- Brute Force technique
- Exhaustive search

# Unit 1: Mathematical analysis of Non-recursive Algorithms

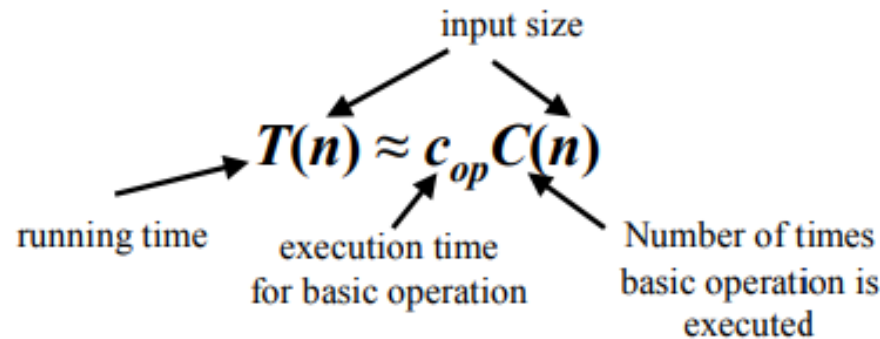
---

- General framework for analyzing time efficiency of Non-Recursive algorithms

# Theoretical analysis of time efficiency

---

- Time efficiency is analyzed by determining the number of repetitions of the **basic operation** as a function of **input size**.
- **Basic operation:** the operation that contributes most towards the running time of the algorithm.



- An algorithm to solve a particular task employs some set of basic operations. When we estimate the amount of work done by an algorithm we usually do not consider all the steps such as e.g. initializing certain variables. Generally, the total number of steps is roughly proportional to the number of the basic operations. Thus, we are concerned mainly with the basic operations - how many times the basic operations have to be performed depending on the size of input.

# Input size and Basic Operation examples

---

<b>Problem</b>	<b>Input Size</b>	<b>Basic Operation</b>
Search for key in list of n items	Number of items in list n	Comparison of Key element with Array element
Sort an array of numbers	The number of elements in the array	Comparison of two array entries
Multiply two matrices of floating point numbers	Dimensions of matrices	Floating point multiplication
Compute $a^n$	n	Floating point multiplication
Graph problem	Number of vertices and edges	Visiting a vertex or traversing an edge

# Example: Maximum element in an Array

---

**ALGORITHM** *MaxElement*( $A[0..n - 1]$ )

//Determines the value of the largest element in a given array

//Input: An array  $A[0..n - 1]$  of real numbers

//Output: The value of the largest element in  $A$

$maxval \leftarrow A[0]$

**for**  $i \leftarrow 1$  **to**  $n - 1$  **do**

**if**  $A[i] > maxval$

$maxval \leftarrow A[i]$

**return**  $maxval$

# Example: Maximum element in an Array

**ALGORITHM** *MaxElement*( $A[0..n - 1]$ )

//Determines the value of the largest element in a given array

//Input: An array  $A[0..n - 1]$  of real numbers

//Output: The value of the largest element in  $A$

$maxval \leftarrow A[0]$

**for**  $i \leftarrow 1$  **to**  $n - 1$  **do**

**if**  $A[i] > maxval$

$maxval \leftarrow A[i]$

**return**  $maxval$

## □ Basic Operation: Comparison operation $A[i] > maxval$

- Let us denote  $T(n)$  the number of times this comparison is executed and try to find a formula expressing it as a function of size  $n$ . The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable  $i$  within the bounds 1 and  $n - 1$ , inclusive. Therefore, we get the following sum for

$$T(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated  $n - 1$  times. Thus,

$$T(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

# Mathematical analysis of nonrecursive algorithms

---

Steps in mathematical analysis of nonrecursive algorithms:

- ❑ Decide on parameter  $n$  indicating input size
- ❑ Identify algorithm's basic operation
- ❑ Determine worst, average, and best case for input of size  $n$
- ❑ Set up summation for  $C(n)$  reflecting algorithm's loop structure
- ❑ Simplify summation using standard formulas



# Useful Formulas for the Analysis of Algorithms

## Important Summation Formulas

1.  $\sum_{i=l}^u 1 = \underbrace{1 + 1 + \dots + 1}_{u-l+1 \text{ times}} = u - l + 1$  ( $l, u$  are integer limits,  $l \leq u$ );  $\sum_{i=1}^n 1 = n$
2.  $\sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$
3.  $\sum_{i=1}^n i^2 = 1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$
4.  $\sum_{i=1}^n i^k = 1^k + 2^k + \dots + n^k \approx \frac{1}{k+1}n^{k+1}$
5.  $\sum_{i=0}^n a^i = 1 + a + \dots + a^n = \frac{a^{n+1} - 1}{a - 1}$  ( $a \neq 1$ );  $\sum_{i=0}^n 2^i = 2^{n+1} - 1$
6.  $\sum_{i=1}^n i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \dots + n2^n = (n-1)2^{n+1} + 2$
7.  $\sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \ln n + \gamma$ , where  $\gamma \approx 0.5772 \dots$  (Euler's constant)
8.  $\sum_{i=1}^n \lg i \approx n \lg n$

## Sum Manipulation Rules

1.  $\sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i$
2.  $\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i$
3.  $\sum_{i=l}^u a_i = \sum_{i=l}^m a_i + \sum_{i=m+1}^u a_i$ , where  $l \leq m < u$
4.  $\sum_{i=l}^u (a_i - a_{i-1}) = a_u - a_{l-1}$

# Commonly used sum manipulation rules and Summation formulas

---

- Two frequently used basic rules of sum manipulation

$$\sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i, \quad (\text{R1})$$

$$\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i, \quad (\text{R2})$$

- Summation Formulas

$$\sum_{i=l}^u 1 = u - l + 1 \quad \text{where } l \leq u \text{ are some lower and upper integer limits,} \quad (\text{S1})$$

$$\sum_{i=0}^n i = \sum_{i=1}^n i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \quad (\text{S2})$$

# Question

---

Consider the Algorithm

ALGORITHM Sum(n)

// Input: A nonnegative integer(n)

S = 0

for i =1 to n do

    S = S + i

return S

- What does this algorithm compute?
- Considering the, basic operation as " $S = S + i$ ", Find out how many times the basic operation will be executed?
- What is the efficiency class of this algorithm?

# Answer

---

Consider the Algorithm

ALGORITHM Sum(n)

// Input: A nonnegative integer(n)

S = 0

for i =1 to n do

    S = S + i

return S

- What does this algorithm compute?
- Considering the, basic operation as " $S = S + i$ ", Find out how many times the basic operation will be executed?
- What is the efficiency class of this algorithm?

Answer:

- Computes the sum of the first n numbers
- Number of executions of basic operation " $S = S + i$ " is n
- The basic operation is always (worst, average, best case) executed n times, so it's  $\Theta(n)$ .

## Example: Finding the Maximum and Minimum Element in an array

---

```
Algorithm Secret( $A[0..n-1]$ )  
//Input: An array  $A[0..n-1]$  of  $n$  real numbers  
 $minval \leftarrow A[0]$ ;  $maxval \leftarrow A[0]$   
for  $i \leftarrow 1$  to  $n-1$  do  
    if  $A[i] < minval$   
         $minval \leftarrow A[i]$   
    if  $A[i] > maxval$   
         $maxval \leftarrow A[i]$   
return  $maxval - minval$ 
```

- What does this algorithm compute?
- Considering the basic operation as " $A[i] < minval$  and  $A[i] > maxval$ ", find out how many times the basic operations will be executed?
- What is the efficiency class of this algorithm?

# Answer

---

```
Algorithm Secret( $A[0..n-1]$ )  
//Input: An array  $A[0..n-1]$  of  $n$  real numbers  
 $minval \leftarrow A[0]$ ;  $maxval \leftarrow A[0]$   
for  $i \leftarrow 1$  to  $n-1$  do  
    if  $A[i] < minval$   
         $minval \leftarrow A[i]$   
    if  $A[i] > maxval$   
         $maxval \leftarrow A[i]$   
return  $maxval - minval$ 
```

- a. What does this algorithm compute?
- b. Considering the basic operation as " $A[i] < minval$  and  $A[i] > maxval$ ", find out how many times the basic operations will be executed?
- c. What is the efficiency class of this algorithm?

- a. Finding maximum and minimum
- b. c. Basic operations:  $A[i] > minval$  and  $A[i] > maxval$

$$T(n) = \sum_{i=1}^{n-1} 2 = 2 \sum_{i=1}^{n-1} 1 = 2(n-1) = 2n - 2 = \Theta(n)$$

# Question

---

- In the following code find out how many times the **sum++** will be executed

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

# Answer

---

□ In the following code find out how many times the **sum++** will be executed

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

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n 1 = \Theta(n^2)$$



## Example: Checking all elements in an array are distinct

---

- **Element uniqueness problem:** check whether all the elements in a given array of  $n$  elements are distinct.

**ALGORITHM** *UniqueElements*( $A[0..n - 1]$ )

//Determines whether all the elements in a given array are distinct

//Input: An array  $A[0..n - 1]$

//Output: Returns “true” if all the elements in  $A$  are distinct

//           and “false” otherwise

**for**  $i \leftarrow 0$  **to**  $n - 2$  **do**

**for**  $j \leftarrow i + 1$  **to**  $n - 1$  **do**

**if**  $A[i] = A[j]$  **return false**

**return true**

# Example: Checking all elements in an array are distinct

- **Element uniqueness problem:** check whether all the elements in a given array of  $n$  elements are distinct.

**ALGORITHM** *UniqueElements*( $A[0..n - 1]$ )

//Determines whether all the elements in a given array are distinct

//Input: An array  $A[0..n - 1]$

//Output: Returns “true” if all the elements in  $A$  are distinct

// and “false” otherwise

**for**  $i \leftarrow 0$  **to**  $n - 2$  **do**

**for**  $j \leftarrow i + 1$  **to**  $n - 1$  **do**

**if**  $A[i] = A[j]$  **return false**

**return true**

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \end{aligned}$$

We also could have computed the sum  $\sum_{i=0}^{n-2} (n-1-i)$  faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

# Question

---

In a competition, four different functions are observed. All the functions use a single for loop and within the for loop, same set of statements are executed. Consider the following for loops:

- A) `for(i = 0; i < n; i++)`
- B) `for(i = 0; i < n; i += 2)`
- C) `for(i = 1; i < n; i *= 2)`
- D) `for(i = n; i > -1; i /= 2)`

If **n** is the size of input(positive), which function is most efficient(if the task to be performed is not an issue)?

# Question

---

In a competition, four different functions are observed. All the functions use a single for loop and within the for loop, same set of statements are executed. Consider the following for loops:

- A) `for(i = 0; i < n; i++)`
- B) `for(i = 0; i < n; i += 2)`
- C) `for(i = 1; i < n; i *= 2)`
- D) `for(i = n; i > -1; i /= 2)`

If **n** is the size of input(positive), which function is most efficient(if the task to be performed is not an issue)?

**Answer: (C)**

**Explanation:** The time complexity of first for loop is  $\Theta(n)$ .

The time complexity of second for loop is  $\Theta(n/2)$ , equivalent to  $\Theta(n)$  in asymptotic analysis.

The time complexity of third for loop is  $\Theta(\log n)$ .

The fourth for loop doesn't terminate.

# Question

---

- i. Find out how many times the statement `printf("*");` will be executed for the following code
- ii. What is the Time Complexity of this code ?

```
function(int n)
{
    if (n==1)
        return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

# Answer

---

- i. Find out how many times the statement "printf("\*");" will be executed for the following code
- ii. What is the Time Complexity of this code ?

```
function(int n)
{
    if (n==1)
        return;
    for (int i=1; i<=n; i++)
    {
        // Inner loop executes only one time due to break statement.
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Time Complexity of the above function  $\Theta(n)$ . Even though the inner loop is bounded by  $n$ , but due to break statement it is executing only once.

# Question

---

- In the following code find out how many times the **sum++** will be executed

```
for (i=1; i<=n*n; i++)  
    for (j=0; j<i; j++)  
        sum++;
```

# Answer

---

```
for (i=1; i<=n*n; i++)  
    for (j=0; j<i; j++)  
        sum++;
```

Exact # of times `sum++` is executed:

$$\begin{aligned}\sum_{i=1}^{n^2} i &= \frac{n^2(n^2 + 1)}{2} \\ &= \frac{n^4 + n^2}{2} \\ &\in \Theta(n^4)\end{aligned}$$



# Question: Study the given algorithm

---

## **ALGORITHM *Mystery*( $n$ )**

//Input: A nonnegative integer  $n$

$S \leftarrow 0$

**for**  $i \leftarrow 1$  **to**  $n$  **do**

$S \leftarrow S + i * i$

**return**  $S$

## ***Answer the following questions***

- a. What does this algorithm compute?
  - b. What is its basic operation?
  - c. How many times is the basic operation executed?
  - d. What is the efficiency class of this algorithm?
  - e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.
-

# Question

---

Consider the following function

```
Void unknown(int n) {  
    int i, j, k = 0;  
    for (i = n/2; i <= n; i++)  
        for (j = 2; j <= n; j = j * 2)  
            k = k + n/2;  
}
```

How many times the statement “ $k=k+n/2$ ” will be executed

## Answer

The outer loop runs  $n/2$  or  $\Theta(n)$  times.

The inner loop runs  $(\log n)$  times (Note that  $j$  is multiplied by 2 in every iteration).

So the statement “ $k = k + n/2$ ” runs  $\Theta(n \log n)$  times.

## Exercise 2: Study the given algorithm

---

### **ALGORITHM *Mystery*( $n$ )**

//Input: A nonnegative integer  $n$

$S \leftarrow 0$

**for**  $i \leftarrow 1$  **to**  $n$  **do**

$S \leftarrow S + i * i$

**return**  $S$

*Answer the following questions*

- a. What does this algorithm compute? **Computes the series  $1^2 + 2^2 + \dots + n^2$**
- b. What is its basic operation? **Multiplication ( $i * i$ )**
- c. How many times is the basic operation executed?  **$n$  times**
- d. What is the efficiency class of this algorithm?  **$\Theta(n)$**
- e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

**$S = n(n+1)(2n+1)/6$ , multiplication is done only three times**

---

# Homework Problem

---

- In the following code find out how many times the **sum++** will be executed

```
sum = 0;  
for( i = 1; i <= n; i++)  
  for( j = i; j <= n; j++)  
    sum++;
```

$$\sum_{i=1}^n \sum_{j=i}^n 1 = \sum_{i=1}^n (n-i+1) = \sum_{i=1}^n (n+1) - \sum_{i=1}^n i =$$

$$n(n+1) - \frac{n(n+1)}{2} = \frac{n(n+1)}{2} \approx n^2$$

# Homework Problem

---

- In the following code find out how many times the **sum++** will be executed

```
sum = 0;  
for( i = 1; i <= n; i++)  
for( j = 1; j <= 2n; j++)  
    sum++;
```

# Homework Problem

---

In the following code find out how many times the **sum++** will be executed

```
sum = 0;  
for( i = 0; i < n; i++)  
  for( j = 0; j < i*i; j++)  
    for( k = 0; k < j; k++)  
      sum++;
```

# Example: Two matrix Multiplication

---

- Given two  $n \times n$  matrices  $A$  and  $B$ , find the time efficiency of the definition-based algorithm for computing their product  $C = AB$

**ALGORITHM** *MatrixMultiplication*( $A[0..n-1, 0..n-1]$ ,  $B[0..n-1, 0..n-1]$ )  
//Multiplies two square matrices of order  $n$  by the definition-based algorithm  
//Input: Two  $n \times n$  matrices  $A$  and  $B$   
//Output: Matrix  $C = AB$   
**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  
    **for**  $j \leftarrow 0$  **to**  $n - 1$  **do**  
         $C[i, j] \leftarrow 0.0$   
        **for**  $k \leftarrow 0$  **to**  $n - 1$  **do**  
             $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$   
**return**  $C$

# Example: Two matrix Multiplication

- Given two  $n \times n$  matrices  $A$  and  $B$ , find the time efficiency of the definition-based algorithm for computing their product  $C = AB$

**ALGORITHM** *MatrixMultiplication*( $A[0..n-1, 0..n-1]$ ,  $B[0..n-1, 0..n-1]$ )  
//Multiplies two square matrices of order  $n$  by the definition-based algorithm  
//Input: Two  $n \times n$  matrices  $A$  and  $B$   
//Output: Matrix  $C = AB$   
**for**  $i \leftarrow 0$  **to**  $n - 1$  **do**  
    **for**  $j \leftarrow 0$  **to**  $n - 1$  **do**  
         $C[i, j] \leftarrow 0.0$   
        **for**  $k \leftarrow 0$  **to**  $n - 1$  **do**  
             $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$   
**return**  $C$

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = \Theta(n^3)$$



## Example: Counting binary digits in binary representation of a decimal number

---

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

### **ALGORITHM** *Binary*( $n$ )

*//Input: A positive decimal integer  $n$*

*//Output: The number of binary digits in  $n$ 's binary representation*

*count*  $\leftarrow 1$

**while**  $n > 1$  **do**

*count*  $\leftarrow$  *count* + 1

*n*  $\leftarrow \lfloor n/2 \rfloor$

**return** *count*

## Example: Counting binary digits in binary representation of a decimal number

---

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

### **ALGORITHM** *Binary(n)*

*//Input: A positive decimal integer  $n$*

*//Output: The number of binary digits in  $n$ 's binary representation*

*count*  $\leftarrow 1$

**while**  $n > 1$  **do**

*count*  $\leftarrow$  *count* + 1

*n*  $\leftarrow \lfloor n/2 \rfloor$

**return** *count*

$$C(n) = n \rightarrow (n/2) \rightarrow (n/4) \rightarrow \dots \rightarrow 1 = \lfloor \log_2 n \rfloor$$

$$\text{Number of times } \mathbf{n > 1} \text{ gets executed is } \lfloor \log_2 n \rfloor + 1.$$

## Example: Counting binary digits in binary representation of a decimal number

---

The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

### ALGORITHM *Binary(n)*

*//Input: A positive decimal integer  $n$*

*//Output: The number of binary digits in  $n$ 's binary representation*

*count*  $\leftarrow 1$

**while**  $n > 1$  **do**

*count*  $\leftarrow$  *count* + 1

$n \leftarrow \lfloor n/2 \rfloor$

**return** *count*

First, notice that the most frequently executed operation here is not inside the **while** loop but rather the comparison  $n > 1$  that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.

A more significant feature of this example is the fact that the loop variable takes on only a few values between its lower and upper limits; therefore, we have to use an alternative way of computing the number of times the loop is executed. Since the value of  $n$  is about halved on each repetition of the loop, the answer should be about  $\log_2 n$ . The exact formula for the number of times the comparison  $n > 1$  will be executed is actually  $\lfloor \log_2 n \rfloor + 1$ —the number of bits in the binary representation of  $n$ .

# Consecutive program fragments

---

- The total running time is the maximum of the running time of the individual fragments

```
sum = 0;
for( i = 0; i < n; i++)
    sum = sum + i;
sum = 0;
for( i = 0; i < n; i++)
    for( j = 0; j < 2n; j++)
        sum++;
```

The first loop runs in  $\Theta(n)$  time,  
the second  $\Theta(n^2)$  time, the maximum is  $\Theta(n^2)$

# Unit 1: Mathematical analysis of Recursive Algorithms

---

- General framework for analyzing time efficiency of Recursive algorithms

# Time efficiency of recursive algorithms

---

Steps in mathematical analysis of recursive algorithms:

- ❑ Decide on parameter  $n$  indicating input size
- ❑ Identify algorithm's basic operation
- ❑ Determine worst, average, and best case for input of size  $n$
- ❑ Set up a recurrence relation and initial condition(s) for  $T(n)$ -the number of times the basic operation will be executed for an input of size  $n$  (alternatively count recursive calls).
- ❑ Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution

# Example: Factorial

---

- Analysis of recursive algorithm to find factorial of a given number

# Example: Factorial

---

- Compute the factorial function  **$F(n) = n!$**  for an arbitrary nonneg-ative integer  **$n$** . Since
$$n! = 1 * 2 * ... *(n - 1) * n = (n - 1)! \cdot n \quad \text{for } n \geq 1$$
and  $0! = 1$  by definition, we can compute
- **$F(n) = F(n - 1) \cdot n$**  with the following recursive algorithm.



# Example: Factorial

---

- Compute the factorial function  $F(n) = n!$  for an arbitrary nonneg-ative integer  $n$ . Since
$$n! = 1 * 2 * \dots * (n - 1) * n = (n - 1)! \cdot n \quad \text{for } n \geq 1$$
and  $0! = 1$  by definition, we can compute
- $F(n) = F(n - 1) \cdot n$  with the following recursive algorithm.

## ALGORITHM $F(n)$

```
//Computes  $n!$  recursively  
//Input: A nonnegative integer  $n$   
//Output: The value of  $n!$   
if  $n = 0$  return 1  
else return  $F(n - 1) * n$ 
```

# Example: Factorial

## ALGORITHM $F(n)$

//Computes  $n!$  recursively

//Input: A nonnegative integer  $n$

//Output: The value of  $n!$

if  $n = 0$  return 1

else return  $F(n - 1) * n$

- The basic operation of the algorithm is multiplication, whose number of executions we denote  $M(n)$ .

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0.$$

to compute  $F(n-1)$       to multiply  $F(n-1)$  by  $n$

$M(0) = 0.$

the calls stop when  $n = 0$       no multiplications when  $n = 0$

# Example: Factorial

---

## ALGORITHM $F(n)$

```
//Computes  $n!$  recursively  
//Input: A nonnegative integer  $n$   
//Output: The value of  $n!$   
if  $n = 0$  return 1  
else return  $F(n - 1) * n$ 
```

Recurrence relation and initial condition for the algorithm's number of multiplications  $M(n)$ :

$$M(n) = M(n - 1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

# Example: Factorial

---

$$M(n) = M(n - 1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

Solving the above recurrence relation using the *method of **backward substitutions***.

$$\begin{aligned} M(n) &= M(n - 1) + 1 && \text{substitute } M(n - 1) = M(n - 2) + 1 \\ &= [M(n - 2) + 1] + 1 = M(n - 2) + 2 && \text{substitute } M(n - 2) = M(n - 3) + 1 \\ &= [M(n - 3) + 1] + 2 = M(n - 3) + 3. \end{aligned}$$

$$= M(n - i) + i$$

$$= M(n - n) + n$$

$M(n)=n$       Hence the time complexity of recursive factorial algorithm is  $T(n) = \Theta(n)$

# Question

---

□ Solve the following recurrence relations using Substitution method

a.  $x(n) = x(n - 1) + 5$  for  $n > 1$ ,  $x(1) = 0$

b.  $x(n) = 3x(n - 1)$  for  $n > 1$ ,  $x(1) = 4$

d.  $x(n) = x(n/2) + n$  for  $n > 1$ ,  $x(1) = 1$  (solve for  $n = 2^k$ )

# Question

---

- Solve the following recurrence relations using Substitution method

a.  $x(n) = x(n - 1) + 5$  for  $n > 1$ ,  $x(1) = 0$

# Answer

---

a.  $x(n) = x(n-1) + 5$  for  $n > 1$ ,  $x(1) = 0$

$$\begin{aligned}x(n) &= x(n-1) + 5 \\&= [x(n-2) + 5] + 5 = x(n-2) + 5 \cdot 2 \\&= [x(n-3) + 5] + 5 \cdot 2 = x(n-3) + 5 \cdot 3 \\&= \dots \\&= x(n-i) + 5 \cdot i \\&= \dots \\&= x(1) + 5 \cdot (n-1) = 5(n-1).\end{aligned}$$

Note: The solution can also be obtained by using the formula for the  $n$  term of the arithmetical progression:

$$x(n) = x(1) + d(n-1) = 0 + 5(n-1) = 5(n-1).$$

# Question

---

Solve the following recurrence relation

b.  $x(n) = 3x(n - 1)$  for  $n > 1$ ,  $x(1) = 4$



# Answer

---

b.  $x(n) = 3x(n-1)$  for  $n > 1$ ,  $x(1) = 4$

$$\begin{aligned}x(n) &= 3x(n-1) \\&= 3[3x(n-2)] = 3^2x(n-2) \\&= 3^2[3x(n-3)] = 3^3x(n-3) \\&= \dots \\&= 3^i x(n-i) \\&= \dots \\&= 3^{n-1}x(1) = 4 \cdot 3^{n-1}.\end{aligned}$$

Note: The solution can also be obtained by using the formula for the  $n$  term of the geometric progression:

$$x(n) = x(1)q^{n-1} = 4 \cdot 3^{n-1}.$$

# Question

---

Solve the following recurrence relations.

d.  $x(n) = x(n/2) + n$  for  $n > 1$ ,  $x(1) = 1$  (solve for  $n = 2^k$ )

# Answer

---

d.  $x(n) = x(n/2) + n$  for  $n > 1$ ,  $x(1) = 1$  (solve for  $n = 2^k$ )

$$\begin{aligned}x(2^k) &= x(2^{k-1}) + 2^k \\&= [x(2^{k-2}) + 2^{k-1}] + 2^k = x(2^{k-2}) + 2^{k-1} + 2^k \\&= [x(2^{k-3}) + 2^{k-2}] + 2^{k-1} + 2^k = x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\&= \dots \\&= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \dots + 2^k \\&= \dots \\&= x(2^{k-k}) + 2^1 + 2^2 + \dots + 2^k = 1 + 2^1 + 2^2 + \dots + 2^k \\&= 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2n - 1.\end{aligned}$$

# HomeWork Problem to Solve

---

Solve the following recurrence relations.

c.  $x(n) = x(n-1) + n$  for  $n > 0$ ,  $x(0) = 0$

e.  $x(n) = x(n/3) + 1$  for  $n > 1$ ,  $x(1) = 1$  (solve for  $n = 3^k$ )

# Answer

---

c.  $x(n) = x(n-1) + n$  for  $n > 0$ ,  $x(0) = 0$

$$\begin{aligned}x(n) &= x(n-1) + n \\&= [x(n-2) + (n-1)] + n = x(n-2) + (n-1) + n \\&= [x(n-3) + (n-2)] + (n-1) + n = x(n-3) + (n-2) + (n-1) + n \\&= \dots \\&= x(n-i) + (n-i+1) + (n-i+2) + \dots + n \\&= \dots \\&= x(0) + 1 + 2 + \dots + n = \frac{n(n+1)}{2}.\end{aligned}$$

e.  $x(n) = x(n/3) + 1$  for  $n > 1$ ,  $x(1) = 1$  (solve for  $n = 3^k$ )

$$\begin{aligned}x(3^k) &= x(3^{k-1}) + 1 \\&= [x(3^{k-2}) + 1] + 1 = x(3^{k-2}) + 2 \\&= [x(3^{k-3}) + 1] + 2 = x(3^{k-3}) + 3 \\&= \dots \\&= x(3^{k-i}) + i \\&= \dots \\&= x(3^{k-k}) + k = x(1) + k = 1 + \log_3 n.\end{aligned}$$

## Example 2:

---

- Towers of Hanoi

# Example: Tower of Hanoi

---

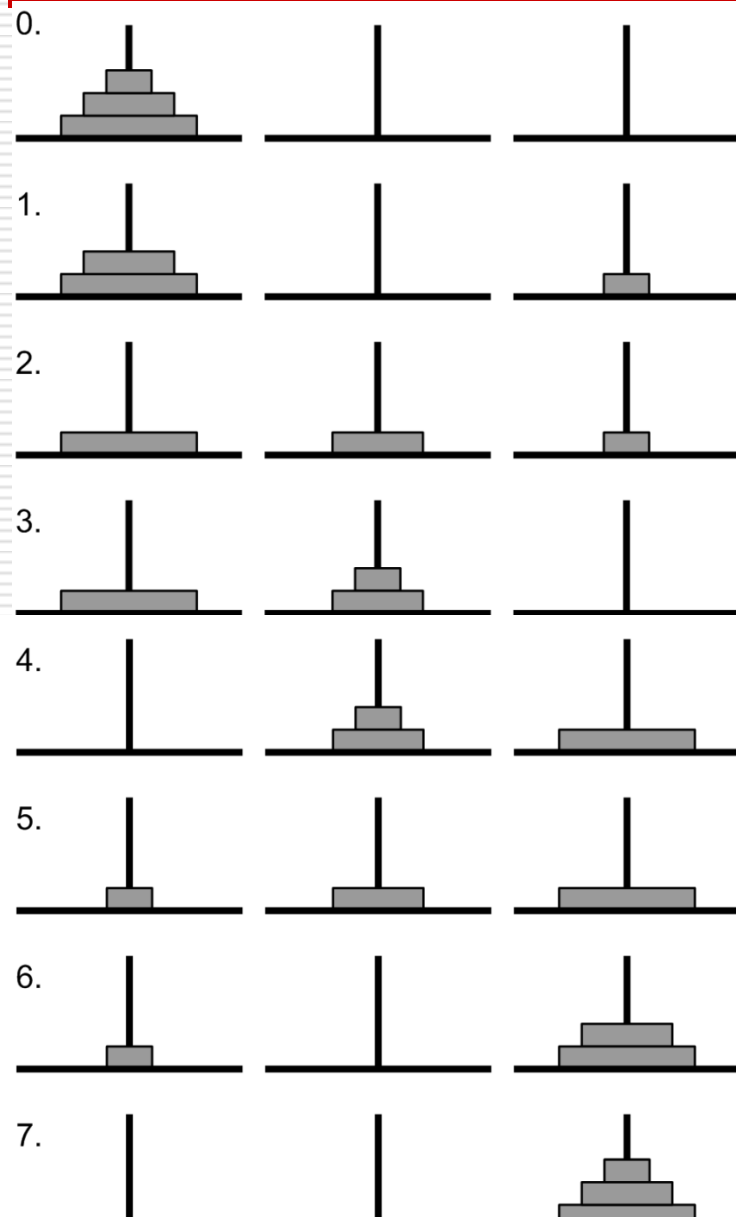
- There are three pegs, Source(A), Auxiliary(B) and Destination(C). Peg A contains a set of disks stacked to resemble a tower, with the largest disk at the bottom and the smallest disk at the top. The objective is to transfer the entire tower of disks in peg A to peg C maintaining the same order of the disks.

## **Obeying the following rules:**

- Only one disk can be transfer at a time.
- Each move consists of taking the upper disk from one of the peg and placing it on the top of another peg i.e. a disk can only be moved if it is the uppermost disk of the peg.
- Never a larger disk is placed on a smaller disk during the transfer.



# Tower of Hanoi: 3 Discs



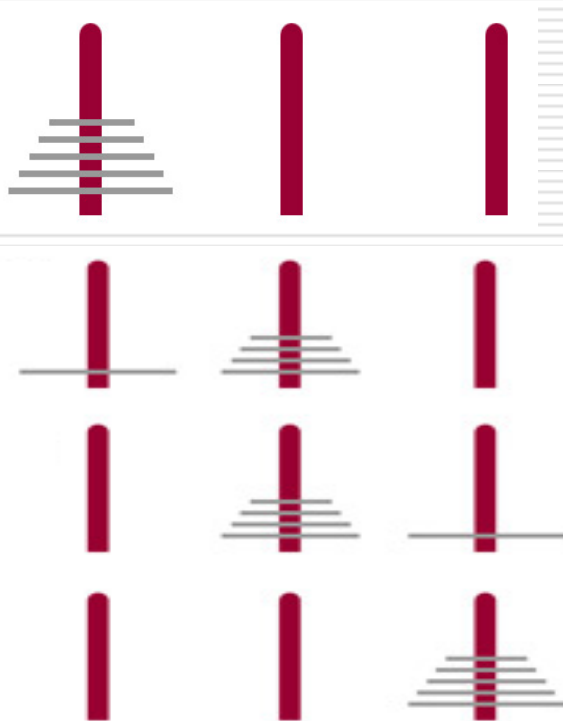


# Towers of Hanoi: Recursive

---

- Move **(n-1)** discs from the source post to the auxiliary post.
- Move the **last** disc to the destination post.
- Move **(n-1)** discs back from the auxiliary post to the destination post.

Source   Aux   Destination



# Towers of Hanoi: Recursive

---

Algorithm **TowerHanoi**(**n**, source, dest, aux)

IF  $n == 1$ , THEN

    move disk from source to dest

ELSE

**TowerHanoi**(**n - 1**, source, aux, dest)

    move disk from source to dest

**TowerHanoi**(**n - 1**, aux, dest, source)

END IF

END

# Towers of Hanoi: Recursive

---

Algorithm TowerHanoi( $n$ , source, dest, aux)

IF  $n == 1$ , THEN

    move disk from source to dest

ELSE

    TowerHanoi( $n - 1$ , source, aux, dest)

    move disk from source to dest

    TowerHanoi( $n - 1$ , aux, dest, source)

END IF

END

The number of disks  $n$  is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves  $M(n)$  depends on  $n$  only, and we get the following recurrence equation for it:

$$M(n) = M(n - 1) + 1 + M(n - 1) \text{ for } n > 1.$$

With the obvious initial condition  $M(1) = 1$ , we have the following recurrence relation for the number of moves  $M(n)$ :

$$M(n) = 2M(n - 1) + 1 \text{ for } n > 1$$

$$M(1) = 1 \quad \text{for } n=1$$

# Towers of Hanoi: Recursive

$$M(n) = 2M(n - 1) + 1 \text{ for } n > 1$$

$$M(1) = 1 \quad \text{for } n=1$$

Solving the above recurrence relation using the *method of backward substitutions*.

$$\begin{aligned} M(n) &= 2M(n - 1) + 1 && \text{sub. } M(n - 1) = 2M(n - 2) + 1 \\ &= 2[2M(n - 2) + 1] + 1 = 2^2M(n - 2) + 2 + 1 && \text{sub. } M(n - 2) = 2M(n - 3) + 1 \\ &= 2^2[2M(n - 3) + 1] + 2 + 1 = 2^3M(n - 3) + 2^2 + 2 + 1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be  $2^4M(n - 4) + 2^3 + 2^2 + 2 + 1$ , and generally, after  $i$  substitutions, we get

$$M(n) = 2^i M(n - i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n - i) + 2^i - 1.$$

Since the initial condition is specified for  $n = 1$ , which is achieved for  $i = n - 1$ , we get the following formula for the solution to recurrence

$$\begin{aligned} M(n) &= 2^{n-1} M(n - (n - 1)) + 2^{n-1} - 1 \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

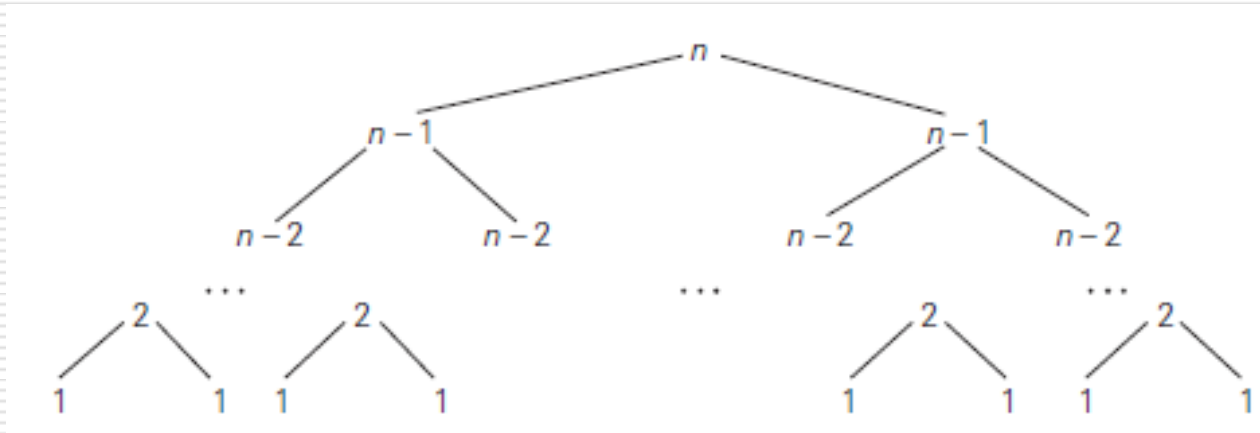
$$\sum_{i=0}^{i=n} 2^i = 2^{n+1} - 1$$

$$\begin{aligned} M(n) &= 2 * 2^{n-1} - 1 \\ &= 2 * (2^n / 2) - 1 \end{aligned}$$

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

# Towers of Hanoi: Recursive

- Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

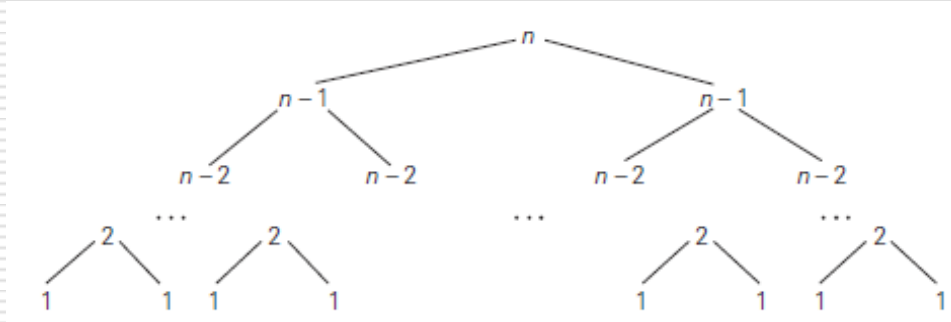


- When a recursive algorithm makes more than a single call to itself, it can be useful for analysis purposes to construct a tree of its recursive calls. In this tree, nodes correspond to recursive calls, and we can label them with the value of the parameter (or, more generally, parameters) of the calls. For the Tower of Hanoi example, the tree is given in Figure 2.5. By counting the number of nodes in the tree, we can get the total number of calls made by the Tower of Hanoi algorithm:

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1.$$

# Towers of Hanoi: Recursive

- Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.



The number of nodes at level 0 =  $2^0 = 1$

The number of nodes at level 1 =  $2^1 = 2$

The number of nodes at level 2 =  $2^2 = 4$

.....

The number of nodes at level  $(n-1) = 2^{(n-1)}$

Total number of calls made by Towers of Hanoi:

$$C(n) = 2^0 + 2^1 + 2^2 + \dots + 2^{(n-1)} = 2^n - 1$$

## Example 3:

---

- Counting bits in the binary representation of an decimal number

# Counting bits in the binary representation of an decimal number

---

## □ Recursive Algorithms

**ALGORITHM** *BinRec*( $n$ )

//Input: A positive decimal integer  $n$

//Output: The number of binary digits in  $n$ 's binary representation

**if**  $n = 1$  **return** 1

**else return** *BinRec*( $\lfloor n/2 \rfloor$ ) + 1



# Counting bits in the binary representation of an decimal number

---

## □ Recursive Algorithms

**ALGORITHM** *BinRec*( $n$ )

//Input: A positive decimal integer  $n$

//Output: The number of binary digits in  $n$ 's binary representation

**if**  $n = 1$  **return** 1

**else return** *BinRec*( $\lfloor n/2 \rfloor$ ) + 1

Let us set up a recurrence and an initial condition for the number of additions  $A(n)$  made by the algorithm. The number of additions made in computing *BinRec*( $\lfloor n/2 \rfloor$ ) is  $A(\lfloor n/2 \rfloor)$ , plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$

Since the recursive calls end when  $n$  is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0.$$

# Solving the Recurrence Relation

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\ &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\ &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\ &\dots && \\ &= A(2^{k-i}) + i && \\ &\dots && \\ &= A(2^{k-k}) + k. \end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable  $n = 2^k$  and hence  $k = \log_2 n$ ,

$$A(n) = \log_2 n \in \Theta(\log n).$$

# Homework Problem

---

Consider the following recursive algorithm for computing the sum of first  $n$  cubes  
 $S(n) = 1^3 + 2^3 + \dots + n^3$

```
Algorithm S(n)
  if ( n== 1)
    return 1
  else
    return s(n-1)+n*n*n
  end of if
end
```

Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

Consider the basic operation as Two multiplications and One Addition in the statement " $s(n-1)+n*n*n$ "

# Answer

Consider the following recursive algorithm for computing the sum of first n cubes  
 $S(n) = 1^3 + 2^3 + \dots + n^3$

```
Algorithm S(n)
  if ( n== 1)
    return 1
  else
    return s(n-1)+n*n*n
end of if
end
```

Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

Sol: The recurrence would be

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ T(n-1) + 3, & \text{if } n > 1 \end{cases}$$

When  $n=1$  then only a value is returned. For any value of  $n > 1$  there are two multiplication and one addition for a total of three operations in addition to recursive call which reduces the problem size by 1. Thus using back substitution

$$T(n) = T(n-1) + 3 = T(n-2) + 3 + 3 = T(n-3) + 3 + 3 + 3 = \dots = T(n-i) + 3 + 3 \dots i \text{ times.}$$

The recursion stops when  $n-i = 1$  or  $i = n-1$ . Thus substituting  $n-1$  for  $i$  in the equation above

$$T(n) = T(n-n+1) + 3*(n-1) = T(1) + 3(n-1) = 1 + 3(n-1) = 3n - 2 = \theta(n)$$

# Algorithm Design Techniques

---

Various design techniques exist:

- ❑ Classifying algorithms based on design ideas or commonality
- ❑ General problem solving strategies
  - Brute force
  - Divide-and-Conquer
  - Decrease-and-Conquer
  - Dynamic Programming
  - Greedy technique
  - Back tracking

# Brute Force Technique

---

## Sorting

- Selection Sort
- Bubble Sort

## Search

- Sequential Search
- String-Matching problem (or String pattern search)

# Brute Force Technique

---

- ❑ Brute Force – the simplest of the design strategies
- ❑ Is a straight forward approach to **solving a problem**, usually directly **based on the problem's statement** and definition of the concepts involved.
- ❑ Just do it – the brute-force strategy is easiest to apply
- ❑ Results in an algorithm that can be improved with a modest amount of time

# Brute Force Technique

---

**Selection sort:** repeatedly pick the smallest element to append to the result.

*Selection sort* is to repetitively pick up the smallest element and put it into the correct position:

- Find the smallest element, and put it to the first position.
- Find the next smallest element, and put it to the second position.
- Repeat until all elements are in the correct positions.



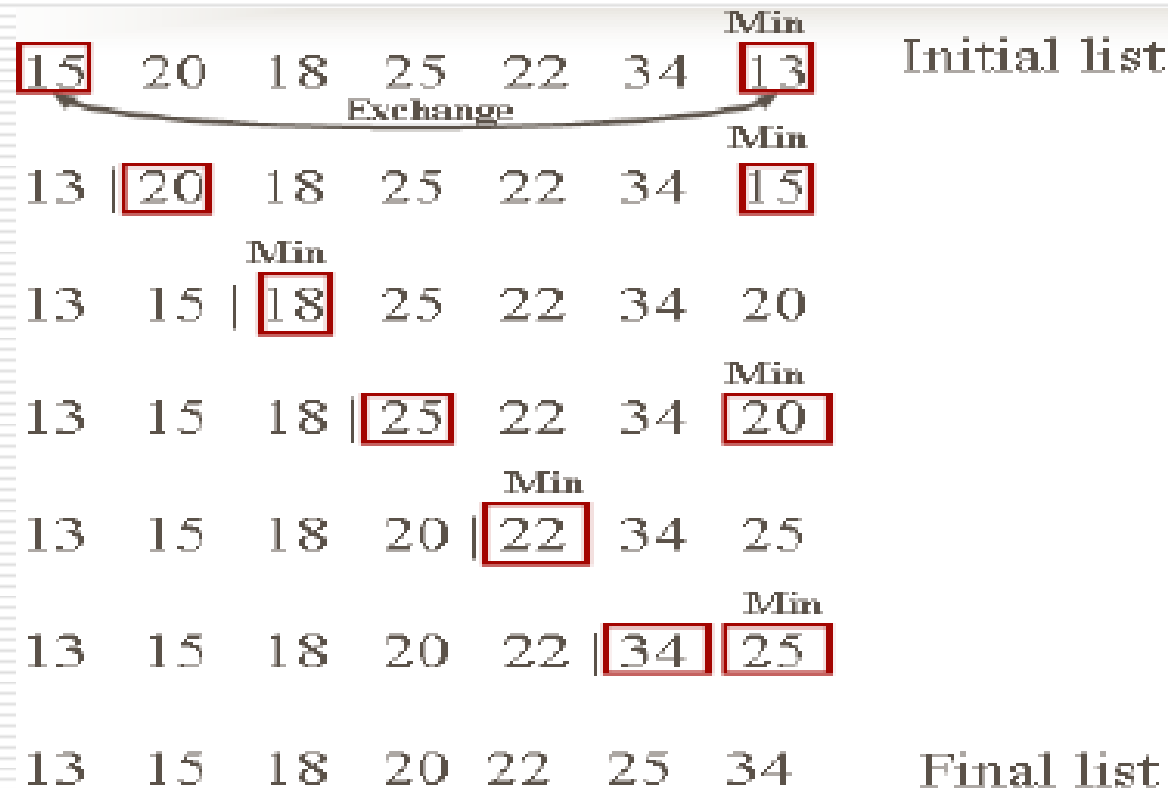
# Selection Sort

---

Scan the list repeatedly to find the elements, one at a time, in a nondecreasing order

- On the  $i^{\text{th}}$  pass through the list, search for the smallest item among the last  $(n-i)$  elements and swap it with  $A[i]$ . After  $(n-1)$  passes the list is sorted.
- Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped.

# Example: Selection Sort



# Example: Selection Sort

Given items	After pass 1	After pass 2	After pass 3	After pass 4
A[0] = 45	5	5	5	5
A[1] = 20	20	15	15	15
A[2] = 40	40	40	20	20
A[3] = 5	45	45	45	40
A[4] = 15	15	20	40	45
1 <sup>st</sup> smallest is 5. Exchange it with 1 <sup>st</sup> item	2 <sup>nd</sup> smallest is 15. Exchange it with 2 <sup>nd</sup> item	3 <sup>rd</sup> smallest is 20. Exchange it with 3 <sup>rd</sup> item	4 <sup>th</sup> smallest is 40. Exchange it with 4 <sup>th</sup> item	All elements are sorted

# Selection Sort Algorithm

---

**ALGORITHM** *SelectionSort*( $A[0..n - 1]$ )

//Sorts a given array by selection sort

//Input: An array  $A[0..n - 1]$  of orderable elements

//Output: Array  $A[0..n - 1]$  sorted in nondecreasing order

**for**  $i \leftarrow 0$  **to**  $n - 2$  {

$min \leftarrow i$

**for**  $j \leftarrow i + 1$  **to**  $n - 1$  {

**if**  $A[j] < A[min]$

$min \leftarrow j$

        }

    swap  $A[i]$  and  $A[min]$

}

# Question

---

- ❑ Sort the elements 89, 45, 68, 90, 29, 34, 17 using selection sort

# Answer

---

- Sort the elements 89, 45, 68, 90, 29, 34, 17 using selection sort

	89	45	68	90	29	34	<b>17</b>
17		45	68	90	<b>29</b>	34	89
17	29		68	90	45	<b>34</b>	89
17	29	34		90	<b>45</b>	68	89
17	29	34	45		90	<b>68</b>	89
17	29	34	45	68		90	<b>89</b>
17	29	34	45	68	89		90

# Question

---

- Sort the list “E X A M P L E” in alphabetical order by selection sort.

	<i>E</i>	<i>X</i>	<i>A</i>	<i>M</i>	<i>P</i>	<i>L</i>	<i>E</i>
<i>A</i>		<i>X</i>	<i>E</i>	<i>M</i>	<i>P</i>	<i>L</i>	<i>E</i>
<i>A</i>	<i>E</i>		<i>X</i>	<i>M</i>	<i>P</i>	<i>L</i>	<i>E</i>
<i>A</i>	<i>E</i>	<i>E</i>		<i>M</i>	<i>P</i>	<i>L</i>	<i>X</i>
<i>A</i>	<i>E</i>	<i>E</i>	<i>L</i>		<i>P</i>	<i>M</i>	<i>X</i>
<i>A</i>	<i>E</i>	<i>E</i>	<i>L</i>	<i>M</i>		<i>P</i>	<i>X</i>
<i>A</i>	<i>E</i>	<i>E</i>	<i>L</i>	<i>M</i>	<i>P</i>		<i>X</i>

# Analysis of Selection Sort

---

- The analysis of selection sort is straightforward. The input size is given by the number of elements  $n$ ; the basic operation is the key comparison  $A[j] < A[\text{min}]$ . The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

- Thus, selection sort is a  $\Theta(n^2)$  algorithm on all inputs.



# Question

---

If there are  $n$  elements to be sorted using Selection Sort then how many Swap operations will be carried out.

# Answer

---

If there are  $n$  elements to be sorted using Selection Sort then how many Swap operations will be carried out.

- Answer: The number of key swaps is only  $(n)$ , or, more precisely,  $n - 1$  (one for each repetition of the  $i$  loop). This property distinguishes selection sort positively from many other sorting algorithms.

# Brute Force Technique

---

## Sorting

- Selection Sort
- Bubble Sort

## Search

- Sequential Search
- String-Matching problem (or String pattern search)

# Brute Force Technique

---

- **Bubble sort:** repeatedly compare neighbor pairs and swap if necessary.

Bubble sort repetitively compares adjacent pairs of elements and swaps if necessary.

- Scan the array, swapping adjacent pair of elements if they are out of order. This bubbles up the largest element to the end.
- Scan the array again, bubbling up the second largest element.
- Repeat until all elements are in order.

# Bubble Sort

---

- ❑ Comparing adjacent elements of the list and exchange them if they are out of order.
- ❑ By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list.
- ❑ The next pass bubbles up the second largest element, and so on, until after  $n - 1$  passes the list is sorted.

# Bubble Sort Algorithm

---

**ALGORITHM** *BubbleSort*( $A[0..n - 1]$ )

//Sorts a given array by bubble sort

//Input: An array  $A[0..n - 1]$  of orderable elements

//Output: Array  $A[0..n - 1]$  sorted in nondecreasing order

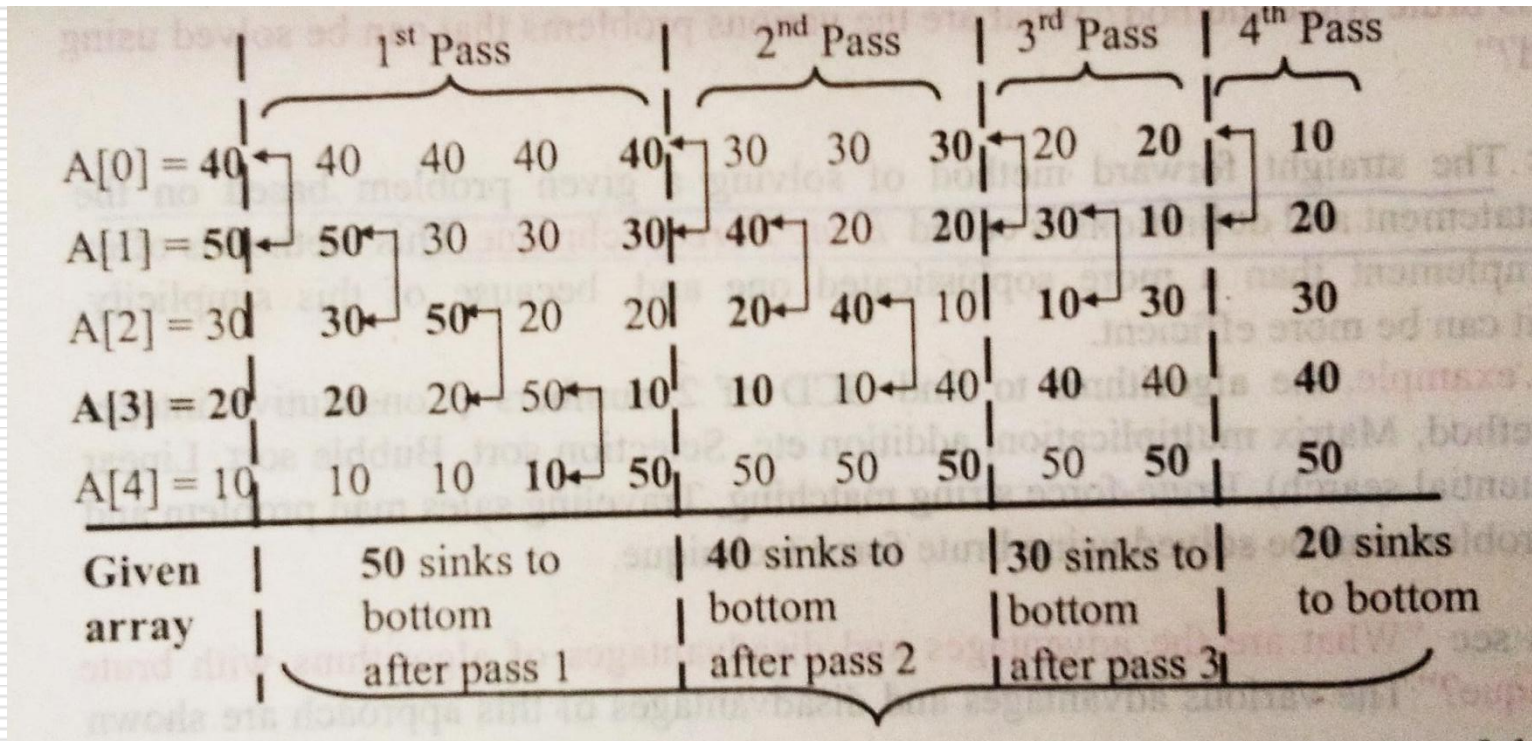
```
for  $i \leftarrow 0$  to  $n - 2$  {  
    for  $j \leftarrow 0$  to  $n - 2 - i$  {  
        if  $A[j + 1] < A[j]$   
            swap  $A[j]$  and  $A[j + 1]$   
        }  
    }  
}
```

# Example

- First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17.

89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17		90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17		90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17		90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17		90
45		68		29		34		17		89		90
etc.												

# Example: Bubble Sort





# Analysis of Bubble Sort Algorithm

---

- The number of key comparisons for the bubble-sort algorithm is the same for all arrays of size  $n$ ; it is obtained by a sum:

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] \\ &= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2). \end{aligned}$$

# Question

---

**ALGORITHM** *BubbleSort*( $A[0..n - 1]$ )

//Sorts a given array by bubble sort

//Input: An array  $A[0..n - 1]$  of orderable elements

//Output: Array  $A[0..n - 1]$  sorted in nondecreasing order

```
for  $i \leftarrow 0$  to  $n - 2$  {  
    for  $j \leftarrow 0$  to  $n - 2 - i$  {  
        if  $A[j + 1] < A[j]$   
            swap  $A[j]$  and  $A[j + 1]$   
    }  
}
```

**Question:**

In the worst case, if the input array given to the algorithm is in decreasing order then How many swap operations will be carried out ?

# Answer

---

- The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{\text{worst}}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

# Improving Bubble Sort Algorithm

---

- ❑ The first version of an Bubble Sort algorithm obtained can often be improved upon with a modest amount of effort.
- ❑ Observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm.
- ❑ Though the new version runs faster on some inputs, it is still in  $\Theta(n^2)$  in the worst and average cases.

# Improved Bubble Sort Algorithm

---

**ALGORITHM** *BubbleSort*( $A[0..n - 1]$ )

//Sorts a given array by bubble sort

//Input: An array  $A[0..n - 1]$  of orderable elements

//Output: Array  $A[0..n - 1]$  sorted in nondecreasing order

**for**  $i \leftarrow 0$  **to**  $n - 2$  {

    swapped = false;

**for**  $j \leftarrow 0$  **to**  $n - 2 - i$  {

**if**  $A[j + 1] < A[j]$

                { swap  $A[j]$  and  $A[j + 1]$  ; swapped = true; }

        }

    // IF no two elements were swapped by inner loop, then break

**if** (swapped == false)

**break;**

}

# Quiz

---

Assume that we use Improved Bubble Sort to sort  $n$  distinct elements in ascending order. When does the best case of Improved Bubble Sort occur?

- ☐ When elements are sorted in ascending order
- ☐ When elements are sorted in descending order
- ☐ When elements are not sorted by any order
- ☐ There is no best case for Improved Bubble Sort. It always takes  $O(n*n)$  time

# Answer

---

Assume that we use Improved Bubble Sort to sort  $n$  distinct elements in ascending order. When does the best case of Improved Bubble Sort occur?

- ☐ **When elements are sorted in ascending order**
- ☐ When elements are sorted in descending order
- ☐ When elements are not sorted by any order
- ☐ There is no best case for Improved Bubble Sort. It always takes  $O(n*n)$  time

# Question

---

- The given array is  $\text{arr} = \{1,2,3,4,5\}$ . (bubble sort is implemented with a flag variable). The number of iterations in selection sort and bubble sort respectively are,
- a) 5 and 4
  - b) 1 and 4
  - c) 0 and 4
  - d) 4 and 1



# Answer

---

- The given array is  $\text{arr} = \{1,2,3,4,5\}$ . (bubble sort is implemented with a flag variable i.e., improved Bubble sort). The number of iterations in selection sort and bubble sort respectively are,
- a) 5 and 4
  - b) 1 and 4
  - c) 0 and 4
  - d) 4 and 1

**Answer: d**

Explanation: Selection sort is insensitive to input, hence 4 iterations. Whereas bubble sort iterates only once to set the flag to 0 as the input is already sorted.

# Quiz

---

What is the best time complexity of Improved bubble sort?

- ☐  $N^2$
- ☐  $N \log N$
- ☐  $N$
- ☐  $N(\log N)^2$

# Answer

---

What is the best time complexity of Improved bubble sort?

- ☐  $N^2$
- ☐  $N \log N$
- ☐  **$N$**
- ☐  $N(\log N)^2$

# Question

---

Following process depicts which sorting algorithm:

First find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position and continue in this way until the entire array is sorted.

- ☐ Selection Sort
- ☐ Bubble Sort

# Answer

---

Following process depicts which sorting algorithm:

First find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position and continue in this way until the entire array is sorted.

- ☐ Selection Sort
- ☐ Bubble Sort

# Question

---

For each  $i$  from 1 to  $n-1$ , there are \_\_\_\_\_ exchanges for selection sort

☐ 1

☐  $n-1$

☐  $n$

# Answer

---

For each  $i$  from 1 to  $n-1$ , there are \_\_\_\_ exchanges for selection sort

☐ 1

☐  $n-1$

☐  $n$

# Question

---

A selection sort compares adjacent elements and swaps them if they are in the wrong order

- ☐ True
- ☐ False
- ☐ Depends on the Array Elements



# Answer

---

A selection sort compares adjacent elements and swaps them if they are in the wrong order

☐ True

☒ False

☐ Depends on the Array Elements

# Improved Bubble Sort

---

## Improved Bubble Sort

- **Worst and Average Case Time Complexity:**  $\Theta(n^2)$ . Worst case occurs when array is reverse sorted.
- **Best Case Time Complexity:**  $O(n)$ . Best case occurs when array is already sorted.

# To Do

---

Write C program to sort the elements using Selection sort and Improved Bubble sort for ascending order

- Provide the input array in ascending order
- Execute the program for n

=10,000

=15,000

=20,000

=25,000

- ☐ Plot the graph between **n** and **time taken** for both Selection Sort and Bubble Sort.

# Question

- Sort the list "E X A M P L E" in alphabetical order by bubble sort

E	↔ <sup>?</sup>	X	↔ <sup>?</sup>	A		M		P		L		E
E		A		X	↔ <sup>?</sup>	M		P		L		E
E		A		M		X	↔ <sup>?</sup>	P		L		E
E		A		M		P		X	↔ <sup>?</sup>	L		E
E		A		M		P		L		X	↔ <sup>?</sup>	E
E		A		M		P		L		E		X
E	↔ <sup>?</sup>	A		M		P		L		E		
A		E	↔ <sup>?</sup>	M	↔ <sup>?</sup>	P	↔ <sup>?</sup>	L		E		
A		E		M		L		P	↔ <sup>?</sup>	E		
A		E		M		L		E		P		
A	↔ <sup>?</sup>	E	↔ <sup>?</sup>	M	↔ <sup>?</sup>	L		E				
A		E		L		M	↔ <sup>?</sup>	E				
A		E		L		E		M				
A	↔ <sup>?</sup>	E	↔ <sup>?</sup>	L	↔ <sup>?</sup>	E						
A		E		E		L						
A	↔ <sup>?</sup>	E	↔ <sup>?</sup>	E	↔ <sup>?</sup>	L						

# Brute Force Technique

---

## Sorting

- Selection Sort
- Bubble Sort

## Search

- Sequential Search
- String-Matching problem (or String pattern search)

# Brute Force Technique

---

## Search

- Sequential Search

# Brute-Force Sequential Search

---

- The algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

**ALGORITHM** *SequentialSearch2*( $A[0..n]$ ,  $K$ )

//Implements sequential search with a search key as a sentinel

//Input: An array  $A$  of  $n$  elements and a search key  $K$

//Output: The index of the first element in  $A[0..n - 1]$  whose value is

// equal to  $K$  or  $-1$  if no such element is found

$A[n] \leftarrow K$

$i \leftarrow 0$

**while**  $A[i] \neq K$  **do**

$i \leftarrow i + 1$

**if**  $i < n$  **return**  $i$

**else return**  $-1$

# Brute Force Technique

---

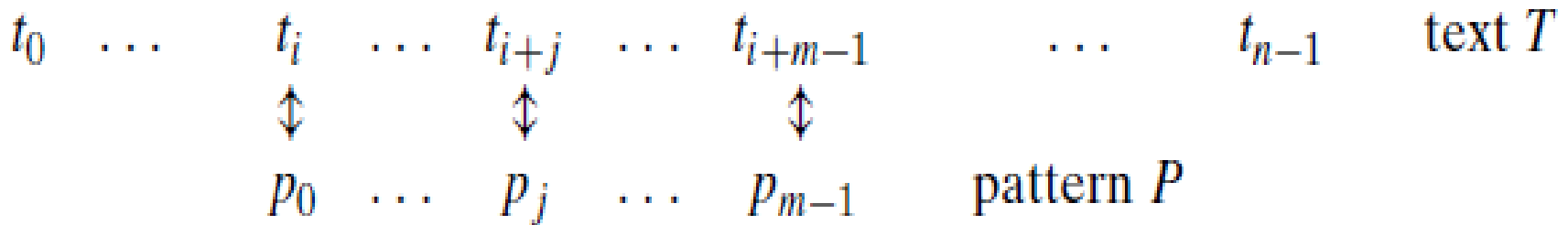
## Search

- String-Matching problem (or String pattern search)



# Brute Force: String Matching Problem

- Given a string of  $n$  characters called the **text** and a string of  $m$  characters ( $m \leq n$ ) called the **pattern**, find a substring of the text that matches the pattern. To put it more precisely, we want to find  $i$ —the index of the leftmost character of the first matching substring in the text—such that

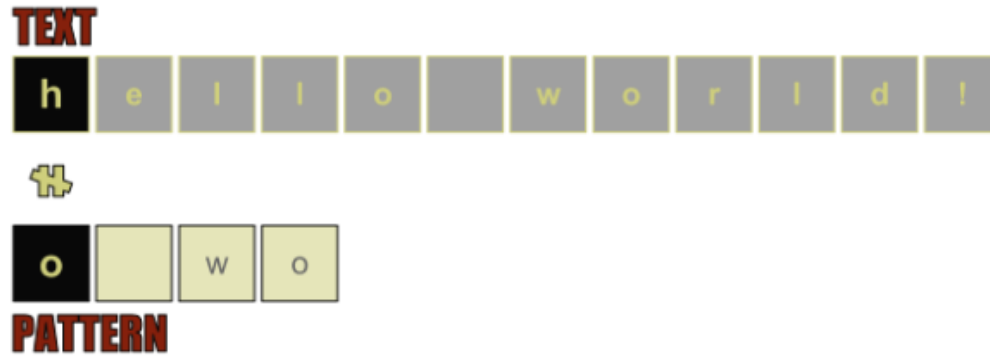


- A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first  $m$  characters of the text and start matching the corresponding pairs of characters from left to right until either all the  $m$  pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text that can still be a beginning of a matching substring is  $n - m$  (provided the text positions are indexed from 0 to  $n - 1$ ).

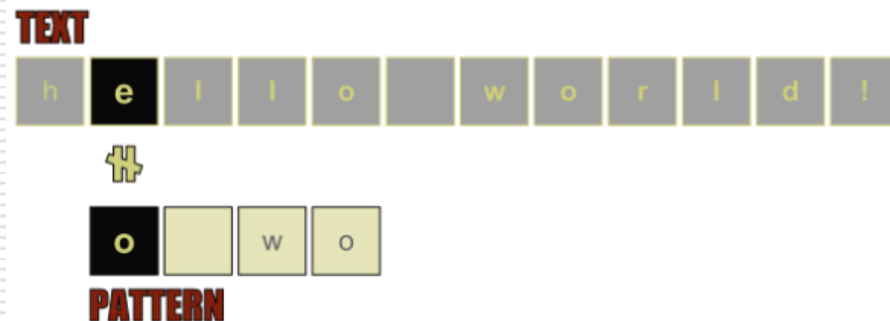
# Example: String Matching Problem

---

- We start by comparing the first characters of the text and the pattern!



- *Because the first character of the text and the pattern don't match, we move forward the second character of the text. Now we compare the second character of the text with the first character of the pattern!*



# Example: String Matching Problem

---

- *In case a character from the text match against the first character of the pattern we move forward to the second character of the pattern and the next character of the text!*

**TEXT**

h	e	l	l	o		w	o	r	l	d	!
---	---	---	---	---	--	---	---	---	---	---	---

||

o		w	o
---	--	---	---

**PATTERN**

**TEXT**

h	e	l	l	o		w	o	r	l	d	!
---	---	---	---	---	--	---	---	---	---	---	---

||

o		w	o
---	--	---	---

**PATTERN**

# String Matching Problem: Algorithm

---

**ALGORITHM** *BruteForceStringMatch*( $T[0..n - 1]$ ,  $P[0..m - 1]$ )

//Implements brute-force string matching

//Input: An array  $T[0..n - 1]$  of  $n$  characters representing a text and

// an array  $P[0..m - 1]$  of  $m$  characters representing a pattern

//Output: The index of the first character in the text that starts a

// matching substring or  $-1$  if the search is unsuccessful

**for**  $i \leftarrow 0$  **to**  $n - m$  **do**

$j \leftarrow 0$

**while**  $j < m$  **and**  $P[j] = T[i + j]$  **do**

$j \leftarrow j + 1$

**if**  $j = m$  **return**  $i$

**return**  $-1$

□ Trace algorithm for Text="COMPUTER" and Pattern="PUT"

# String Matching Problem: Algorithm

---

## Time Analysis

- ❑ The algorithm shifts the pattern almost always after a single character comparison.
- ❑ The worst case is much worse: the algorithm may have to make all  $m$  comparisons before shifting the pattern, and this can happen for each of the  $n - m + 1$  tries.

# Question

---

Determine the number of character comparisons made by the brute-force algorithm in searching for the pattern GANDHI in the text  
THERE\_IS\_MORE\_TO\_LIFE\_THAN\_INCREASING\_ITS\_SPEED  
(Assume that the length of the text is 47 characters long – is known before the search starts.)

# Question

---

Determine the number of character comparisons made by the brute-force algorithm in searching for the pattern GANDHI in the text  
THERE\_IS\_MORE\_TO\_LIFE\_THAN\_INCREASING\_ITS\_SPEED  
(Assume that the length of the text—it is 47 characters long—is known before the search starts.)

Answer:

- 43 comparisons. The algorithm will make  $47 - 6 + 1 = 42$  trials: In the first one, the G of the pattern will be aligned against the first T of the text; in the last one, it will be aligned against the last space. On each but one trial, the algorithm will make one unsuccessful comparison; on one trial—when the G of the pattern is aligned against the G of the text—it will make two comparisons. Thus, the total number of character comparisons will be  $41 * 1 + 1 * 2 = 43$ .

# Question

---

- How many comparisons (both successful and unsuccessful) are made by the brute-force string matching algorithm in searching for pattern “00001” in the binary text of 1000 zeros?



# Question

---

- How many comparisons (both successful and unsuccessful) are made by the brute-force string matching algorithm in searching for pattern “00001” in the binary text of 1000 zeros?

For the pattern 00001, the algorithm will make four successful and one unsuccessful comparison on each of its trials and then shift the pattern one position to the right:

```
0 0 0 0 0 0
0 0 0 0 1
  0 0 0 0 1
```

etc.

```
0 0 0 0 0
```

```
0 0 0 0 1
```

The total number of character comparisons will be  $C = 5 \cdot 996 = 4980$ .

# Question

---

- How many comparisons (both successful and unsuccessful) are made by the brute-force string-matching algorithm in searching for each of the following patterns in the binary text of 1000 zeros?  
b. 10000 c. 01010

# Answer

---

- How many comparisons (both successful and unsuccessful) are made by the brute-force string-matching algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

b. 10000 c. 01010

b) 10000 There will be a total of  $1000 - 5 + 1 = 996$  iterations. In each of these iterations, the first comparison would itself be unsuccessful. Hence, there will be  $996 * 1 = 996$  unsuccessful comparisons and there will not be any successful comparisons. Total comparisons = 996.

c) 01010 There will be a total of  $1000 - 5 + 1 = 996$  iterations. In each of these iterations, the first comparison would be successful and the second comparison would be unsuccessful. Hence, there will be  $996 * 1 = 996$  successful comparisons and another  $996 * 1 = 996$  unsuccessful comparisons. Total comparisons = 1992.

# Exhaustive Search

---

- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem

# Exhaustive Search

---

- ***Exhaustive search*** is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element

# Exhaustive Search

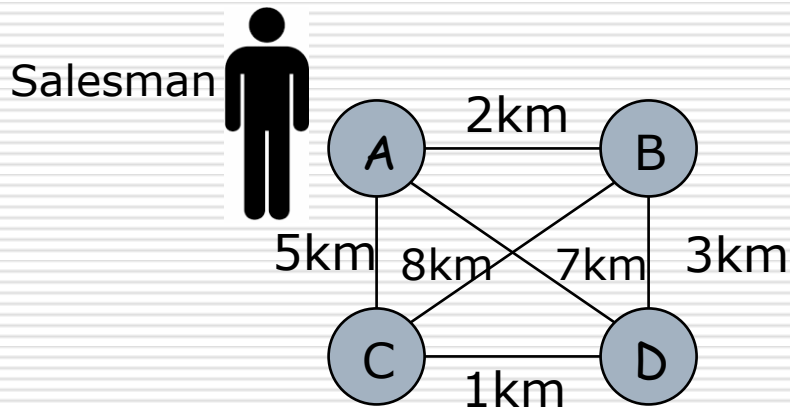
---

- Traveling Salesman Problem

# Traveling Salesman Problem (TSP)

---

- A salesman wants to visit all cities, A, B, C and D. What is the best way to do this (Minimal travel distance)?



# Traveling Salesman Problem (TSP)

---

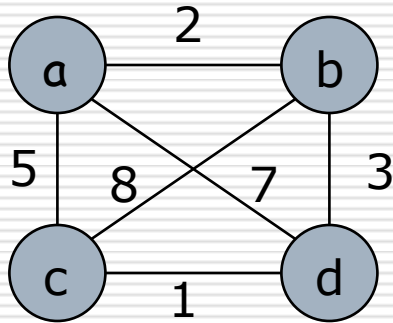
- Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
- How to Solve TSP ?
  - Get all tours by generating **all permutations of  $n-1$**  intermediate cities, compute the tour lengths, and find the shortest among them.



# Traveling Salesman Problem (TSP)

---

List all possible routes starting from city 'a' and find optimal route



$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$

$$2+8+1+7 = 18$$

$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$

$$2+3+1+5 = 11 \leftarrow \text{optimal}$$

$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$

$$5+8+3+7 = 23$$

$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$

$$5+1+3+2 = 11 \leftarrow \text{optimal}$$

$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$

$$7+3+8+5 = 23$$

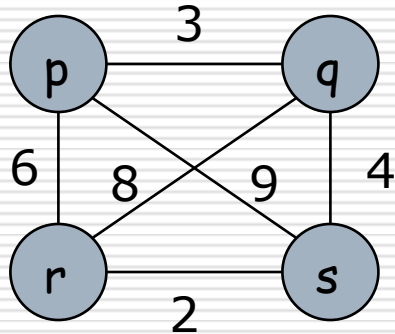
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$

$$7+1+8+2 = 18$$

# Question

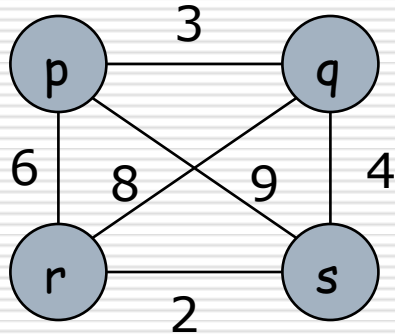
---

List all tours starting from city p and find the shortest among them



# Question

List all tours starting from city p and find the shortest among them



$p \xrightarrow{3} q \xrightarrow{9} r \xrightarrow{2} s \xrightarrow{8} p$  (Cost = 22)  
 $p \xrightarrow{3} q \xrightarrow{4} s \xrightarrow{2} r \xrightarrow{6} p$  (Cost = 15)  
 $p \xrightarrow{6} r \xrightarrow{9} q \xrightarrow{4} s \xrightarrow{8} p$  (Cost = 27)  
 $p \xrightarrow{6} r \xrightarrow{2} s \xrightarrow{4} q \xrightarrow{3} p$  (Cost = 15)  
 $p \xrightarrow{8} s \xrightarrow{4} q \xrightarrow{9} r \xrightarrow{6} p$  (Cost = 27)  
 $p \xrightarrow{8} s \xrightarrow{2} r \xrightarrow{9} q \xrightarrow{3} p$  (Cost = 22)

# Exhaustive Search: Traveling Salesman Problem

---

- We can get all the tours by generating all the permutations of  $n - 1$  intermediate cities, compute the tour lengths, and find the shortest among them.
- Exhaustive-search approach impractical for all but very small values of  $n$ .

# Exhaustive Search

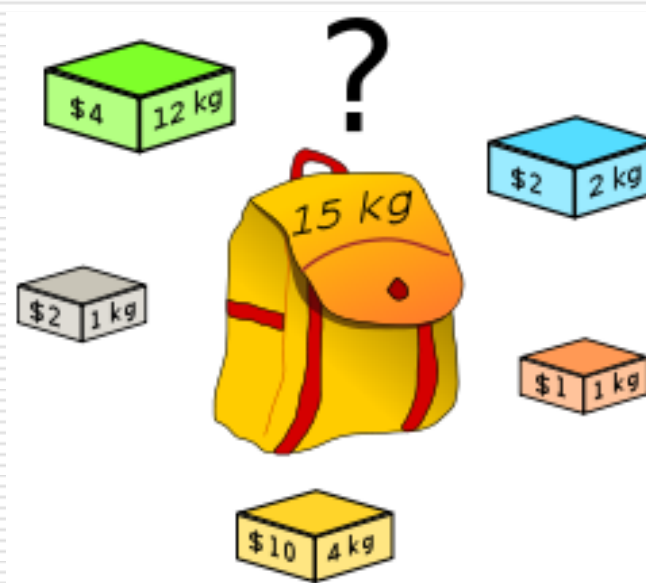
---

- Knapsack Problem

# Knapsack Problem

---

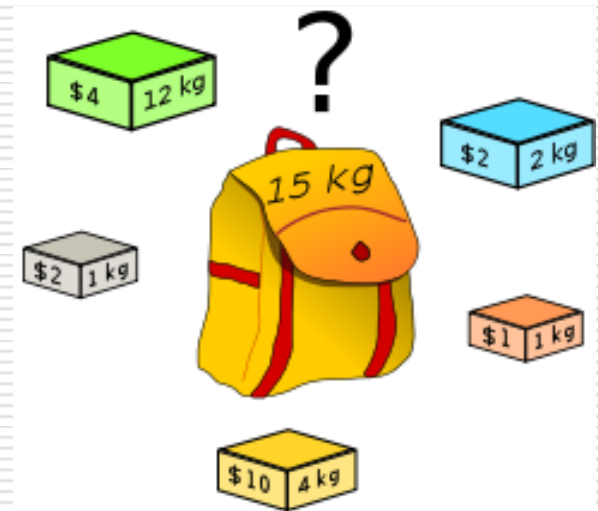
- Given  $n$  items of known weights  $w_1, w_2, \dots, w_n$  and values  $v_1, v_2, \dots, v_n$  and a knapsack of capacity  $W$ , find the most valuable subset of the items that fit into the knapsack.



# Knapsack Problem

---

- ❑ You have knapsack that has capacity (weight)  $W$
- ❑ You have several items  $i_1, \dots, i_n$
- ❑ Each item  $i_j$  has a weight  $w_j$  and a value  $v_j$
- ❑ You want to place a certain number of copies of each item  $i_j$  in the knapsack so that:
  - The knapsack weight **capacity** is **not exceeded** and
  - The **total value** is **maximal**



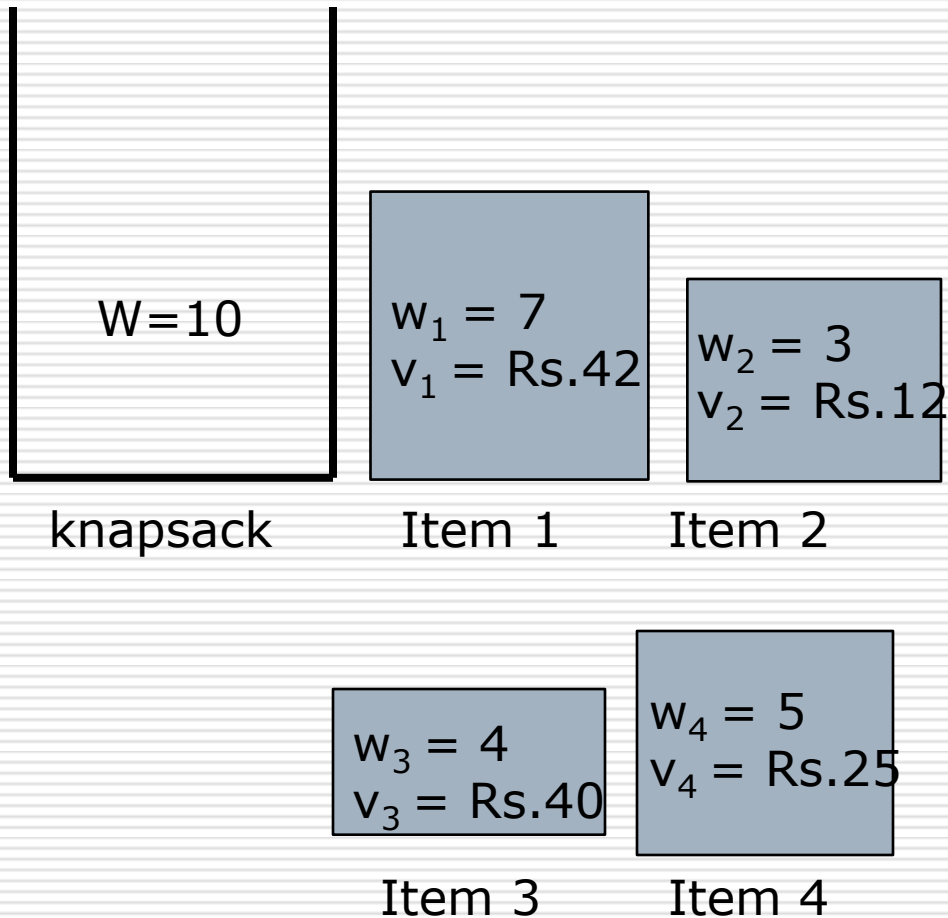
\_\_\_\_\_



\_\_\_\_\_

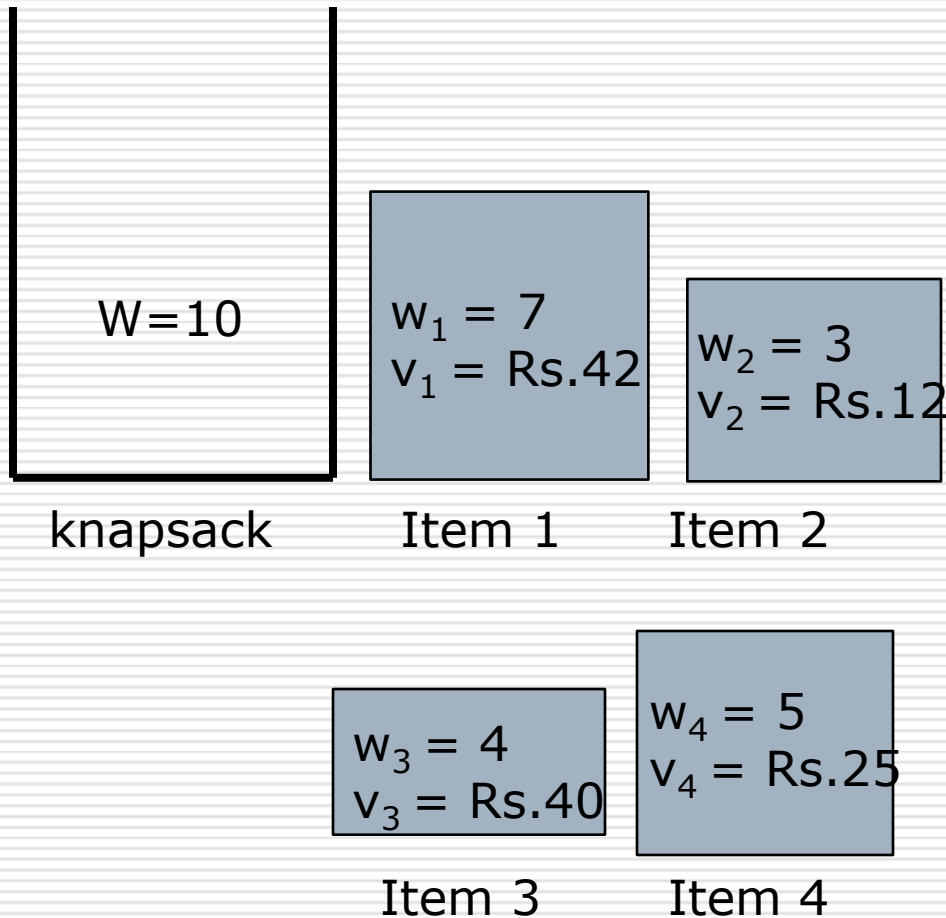


# Example



subset	weight	value
$\emptyset$	0	Rs.0
{1}	7	Rs.42
{2}	3	Rs.12
{3}	4	Rs.40
{4}	5	Rs.25
{1,2}	10	Rs.54
{1,3}	11	Not feasible
{1,4}	12	Not feasible

# Example



subset	weight	value
$\emptyset$	0	Rs.0
{1}	7	Rs.42
{2}	3	Rs.12
{3}	4	Rs.40
{4}	5	Rs.25
{1,2}	10	Rs.54
{1,3}	11	!feasible
{1,4}	12	!feasible
{2,3}	7	Rs.52
{2,4}	8	Rs.37
{3,4}	9	Rs.65
{1,2,3}	14	!feasible
{1,2,4}	15	!feasible
{1,3,4}	16	!feasible
{2,3,4}	12	!feasible
{1,2,3,4}	19	!feasible

# Question: Knapsack Problem

---

- Let  $W=40$ . Let the weight of three objects be 20, 25, 10 and value of corresponding objects be 30, 40 and 35. Find the optimal solution.

# Question: Knapsack Problem

---

- Let  $W=40$ . Let the weight of three objects be 20, 25, 10 and value of corresponding objects be 30, 40 and 35. Find the optimal solution.

Sl. Number	Objects selected	Total weight of objects selected	Feasible or not	Profit earned
1.	None i.e., $\{\}$	0	Feasible	0
2.	1 i.e., $\{1\}$	20	Feasible	30
3.	2 i.e., $\{2\}$	25	Feasible	40
4.	3 i.e., $\{3\}$	10	Feasible	35
5.	1,2 i.e., $\{1, 2\}$	$20 + 25 = 45$	Not feasible	
6.	1,3 i.e., $\{1, 3\}$	$20 + 10 = 30$	Feasible	65
7.	2,3 i.e., $\{2, 3\}$	$25 + 10 = 35$	Feasible	75
8.	1,2,3 i.e., $\{1, 2, 3\}$	$20 + 25 + 10 = 55$	Not feasible	

# Exhaustive Search: Knapsack Problem

---

- The exhaustive-search approach to this problem leads to generating all the subsets of the set of  $n$  items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.
- Since the number of subsets of an  $n$ -element set is  $2^n$ , the exhaustive search leads to a  $2^n$  algorithm, no matter how efficiently individual subsets are generated.

# Exhaustive Search

---

- Assignment Problem

# Assignment Problem

---

- There are  $n$  people who need to be assigned to execute  $n$  jobs, one person per job.
- $C[i, j]$  : cost that would occur if  $i^{\text{th}}$  person is assigned to  $j^{\text{th}}$  job.
- Find an assignment with the minimum total cost.

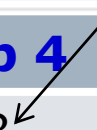
# Assignment Problem

---

- Let there be  $N$  workers and  $N$  jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

	Job 1	Job 2	Job 3	Job 4
Person A	Rs. 9	Rs. 2	Rs. 7	Rs. 8
Person B	Rs. 6	Rs. 4	Rs. 3	Rs. 7
Person C	Rs. 5	Rs. 8	Rs. 1	Rs. 8
Person D	Rs. 7	Rs. 6	Rs. 9	Rs. 4

Person A  
Takes 8 units  
To Complete Job





# Assignment Problem

- Let there be  $N$  workers and  $N$  jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

	Job 1	Job 2	Job 3	Job 4
Person A	9	2	7	8
Person B	6	4	3	7
Person C	5	8	1	8
Person D	7	6	9	4

Person A  
Takes 8 units  
Of time to  
Complete Job4

Green values show optimal job assignment that is A-Job2, B-Job1, C-Job3 and D-Job4

# Example: Assignment Problem

---

Find the optimal job assignment for the following

	<b>Job 1</b>	<b>Job 2</b>	<b>Job 3</b>	<b>Job 4</b>
<b>Person A</b>	10	3	8	9
<b>Person B</b>	7	5	4	8
<b>Person C</b>	6	9	2	9
<b>Person D</b>	8	7	10	5

# Example: Assignment Problem

	Job 1	Job 2	Job 3	Job 4
Person A	10	3	8	9
Person B	7	5	4	8
Person C	6	9	2	9
Person D	8	7	10	5

$\langle 1,2,3,4 \rangle$  Cost =  $10 + 5 + 2 + 5 = 22$   
 $\langle 1,2,4,3 \rangle$  Cost =  $10 + 5 + 9 + 10 = 34$   
 $\langle 1,3,2,4 \rangle$  Cost =  $10 + 4 + 9 + 5 = 28$   
 $\langle 1,3,4,2 \rangle$  Cost =  $10 + 4 + 9 + 7 = 30$   
 $\langle 1,4,2,3 \rangle$  Cost =  $10 + 8 + 9 + 10 = 37$   
 $\langle 1,4,3,2 \rangle$  Cost =  $10 + 8 + 2 + 7 = 27$

$\langle 3,1,2,4 \rangle$  Cost =  $8 + 7 + 9 + 5 = 29$   
 $\langle 3,1,4,2 \rangle$  Cost =  $8 + 7 + 9 + 7 = 31$   
 $\langle 3,2,1,4 \rangle$  Cost =  $8 + 5 + 6 + 5 = 24$   
 $\langle 3,2,4,1 \rangle$  Cost =  $8 + 5 + 9 + 8 = 30$   
 $\langle 3,4,1,2 \rangle$  Cost =  $8 + 8 + 6 + 7 = 29$   
 $\langle 3,4,2,1 \rangle$  Cost =  $8 + 8 + 9 + 8 = 33$

$\langle 2,1,3,4 \rangle$  Cost =  $3 + 7 + 2 + 5 = 17$   
 $\langle 2,1,4,3 \rangle$  Cost =  $3 + 7 + 9 + 10 = 29$   
 $\langle 2,3,1,4 \rangle$  Cost =  $3 + 5 + 6 + 5 = 19$   
 $\langle 2,3,4,1 \rangle$  Cost =  $3 + 5 + 9 + 8 = 25$   
 $\langle 2,4,1,3 \rangle$  Cost =  $3 + 8 + 6 + 10 = 27$   
 $\langle 2,4,3,1 \rangle$  Cost =  $3 + 8 + 2 + 8 = 21$

$\langle 4,1,2,3 \rangle$  Cost =  $9 + 7 + 9 + 10 = 35$   
 $\langle 4,1,3,2 \rangle$  Cost =  $9 + 7 + 2 + 7 = 25$   
 $\langle 4,2,1,3 \rangle$  Cost =  $9 + 5 + 6 + 10 = 30$   
 $\langle 4,2,3,1 \rangle$  Cost =  $9 + 5 + 2 + 8 = 24$   
 $\langle 4,3,1,2 \rangle$  Cost =  $9 + 4 + 6 + 7 = 26$   
 $\langle 4,3,2,1 \rangle$  Cost =  $9 + 4 + 9 + 8 = 30$

# Exhaustive Search: Assignment Problem

---

- We can describe feasible solutions to the assignment problem as  $n$ -tuples  $j_1, \dots, j_n$  in which the  $i$ th component,  $i = 1, \dots, n$ , indicates the column of the element selected in the  $i$ th row (i.e., the job number assigned to the  $i$ th person).
- The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first  $n$  integers.
- Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers  $1, 2, \dots, n$ , computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.
- Since the number of permutations to be considered for the general case of the assignment problem is  $n!$ , exhaustive search is impractical for all but very small instances of the problem.

# Thanks for Listening

---