



# Algorithms Analysis and Design

## Chapter 3

### Brute Force and Exhaustive Search

# Introduction

After introducing the framework and methods for algorithm analysis in the preceding chapter, **we are ready to embark on a discussion of algorithm design strategies.**

- The subject of this chapter is **brute force** and its important special case, **exhaustive search**

- **Brute force**
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs



- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound



# Brute Force Strategy



# Brute Force

- ❑ **Brute force** is a straightforward technique of problem-solving, usually directly based on the problem statement and definitions of the concepts involved.
- ❑ “**Just do it!**” would be another way to describe the prescription of the brute-force approach.
- ❑ This method **relies more on compromising the power (force) of a computer system** for solving a problem than on a good algorithm design.
- ❑ And **often**, the brute-force strategy is indeed the one that is **easiest to apply**.
- ❑ **Rarely the most efficient** but can be applied to wide range of problems.
- ❑ Rarely a source of clever algorithms.
- ❑ For some elementary problems, almost as good as most efficient. May not be worth cost.
- ❑ Sometimes the first one that comes to mind.

A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.

# Examples

- ❑ Computing  $a^n$  ( $a > 0$ ,  $n$  a nonnegative integer)
- ❑ Computing  $n!$
- ❑ Multiplying two matrices
- ❑ Searching for a key of a given value in a list.
- ❑ Sorting algorithms (Selection sort , Bubble sort)
- ❑ Brute force string matching
- ❑ Brute force search (**Exhaustive search**)
  - Travelling salesman problem
  - 0/1 Knapsack problem
  - Assignment problem

# Importance of Brute Force approach

The brute-force approach should not be overlooked as an important algorithm design strategy

- ❑ Unlike some of the other strategies, brute force is **applicable to a very wide variety of problems**.
- ❑ The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a **brute-force algorithm can solve those instances with acceptable speed**.
- ❑ Even if too inefficient in general, a brute-force algorithm **can still be useful for solving small-size instances of a problem**.
- ❑ A brute-force algorithm can serve an important theoretical or educational purpose as a criterion with which **to judge more efficient alternatives for solving a problem**.

# Calculating powers of a number

- ❑ **Problem:** exponentiation problem [ Computing  $a^n$  (  $a > 0$ ,  $n$  a nonnegative integer ) ]

$$a^n = \underbrace{a * a * a \dots\dots\dots * a}_{n \text{ times}}$$

- ❑ Time complexity:  $\theta(n)$

- ❑ **Can we find a better (faster) algorithm?**

Naive algorithm

```
ALGORITHM pow ( a , n )
{
    result = 1
    for i = 1 to n
        result = result * a
    return result
}
```

## Exercise

❑ **Problem:** exponentiation problem [ Computing  $a^n$  (  $a > 0$ ,  $n$  a nonnegative integer ) ]

$$T(n) = T(n-1) + C$$

$$a^n = \underbrace{a * a * a \dots * a}_{n \text{ times}}$$

## Recursive algorithm

1. Design a recursive algorithm for computing  $a^n$
2. Derive a recurrence relation and solve it.
3. Find the time complexity.
4. Is it a good algorithm for solving this problem?

### ALGORITHM Rec\_pow ( a , n )

 $\{$ 

}



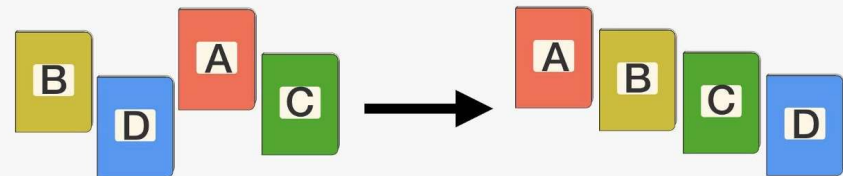


The application of the brute-force approach to the problem of  
**sorting**

# Sorting Problem

- ❑ **Sorting Problem:** given a list of  $n$  orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in **nondecreasing** order.
- ❑ “What would be the most straightforward method for solving the sorting problem?”
- ❑ The two prime candidates are:
  - **Selection sort**
  - **Bubble sort**

## Sorting Algorithms

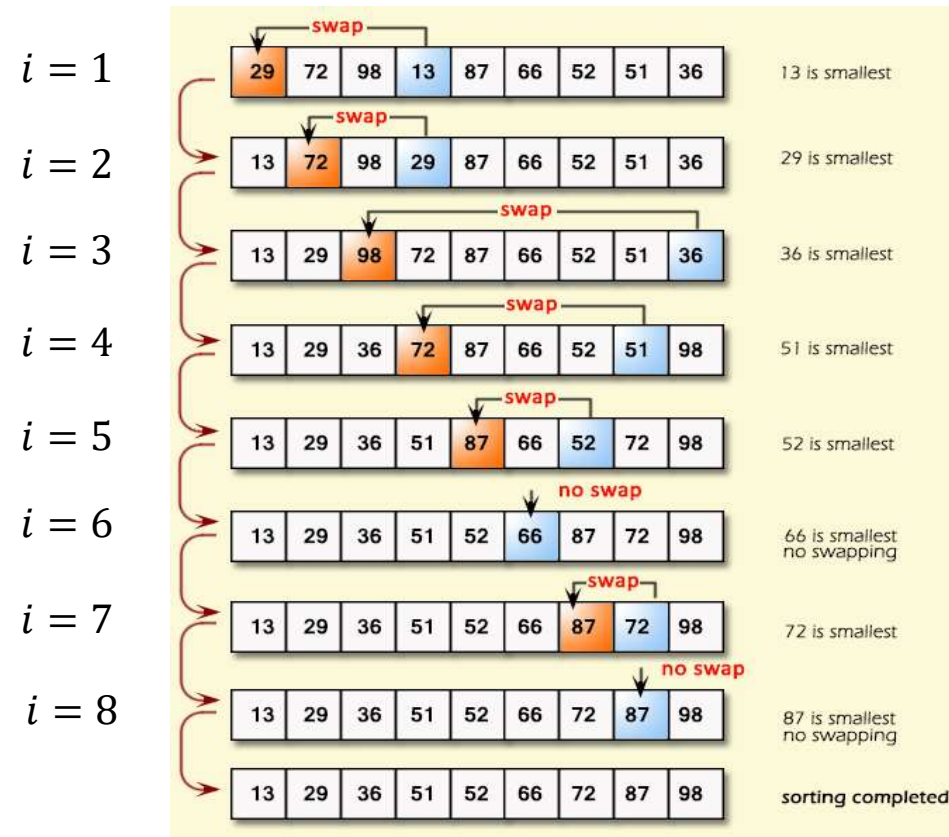


# Selection Sort

- ❑ Scan the entire list to find its smallest element.
- ❑ swap it with the first element. First element now in its in its final sorted position.
- ❑ Scan remaining  $n-1$  items, starting with second element, to find the smallest among them.
- ❑ Swap it with the second element. Second element now in its final sorted position.
- ❑ Repeat for the remaining elements.

## Example of selection sort

- Sort the list 29, 72, 98, 13, 87, 66, 52, 51, 36 in ascending order by **selection sort**



# Exact Analysis of selection sort

Sorts a given array by selection sort

**Input:** An array  $A[1 \dots n]$  of orderable elements

**Output:** Array  $A[1 \dots n]$  sorted in ascending order

**ALGORITHM** *SelectionSort* ( $A[1 \dots n]$ )

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

$\text{min} \leftarrow i$

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

**if**  $A[j] < A[\text{min}]$

$\text{min} \leftarrow j$

    swap  $A[i]$  and  $A[\text{min}]$



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

# Approximate Analysis of selection sort

**ALGORITHM** SelectionSort (A [1 .... n])

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

min  $\leftarrow i$

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

if  $A[j] < A[\text{min}]$

min  $\leftarrow j$

swap  $A[i]$  and  $A[\text{min}]$

The basic operation is the key comparison  $A[j] < A[\text{min}]$

The number of times it is executed depends on the array size and is given by the following sum:

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

Thus, selection sort is a  $\theta(n^2)$

Note, however, that the number of key swaps is only  $\theta(n)$ , or, more precisely,  $n - 1$  (one for each repetition of the  $i$  loop)

➡  $\sum_{i=m}^n c = c(n - m + 1) \quad \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

إذا ما بلشتت من ا بنطبق هاي  
القاعدة لفور لوب

## Exercise

Rewrite the ***SelectionSort*** algorithm to sort the elements of  $A$  in descending order.

**ALGORITHM** *SelectionSort* ( $A[1 \dots n]$ )

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

$\text{min} \leftarrow i$

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

**if**  $A[j] < A[\text{min}]$

$\text{min} \leftarrow j$

    swap  $A[i]$  and  $A[\text{min}]$



## Bubble sort

- ❑ The main concept of bubble sort is to compare adjacent elements of the list and exchange them if they are out of order. (**Compare and Swap**)
- ❑ 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.
- ❑ The same process goes for the remaining iterations (until after  $n - 1$  passes the list is sorted.)

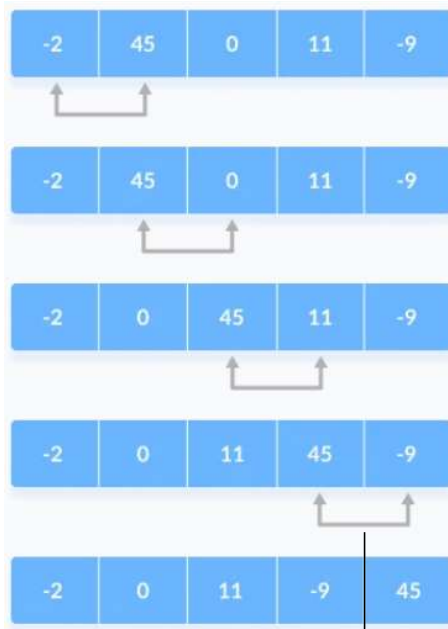
After each iteration, the largest element among the unsorted elements is placed at the end.



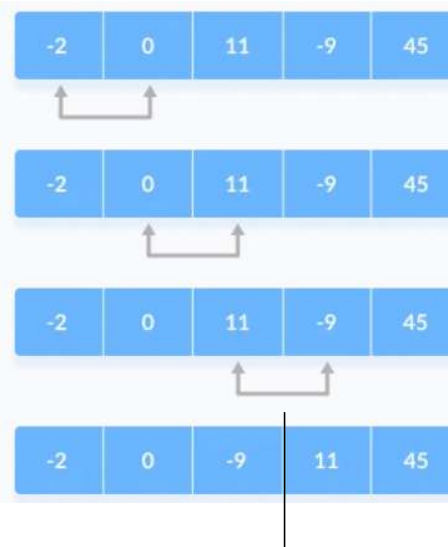
# Example of bubble sort

- Apply bubble sort on the list **-2, 45, 0, 11, -9**

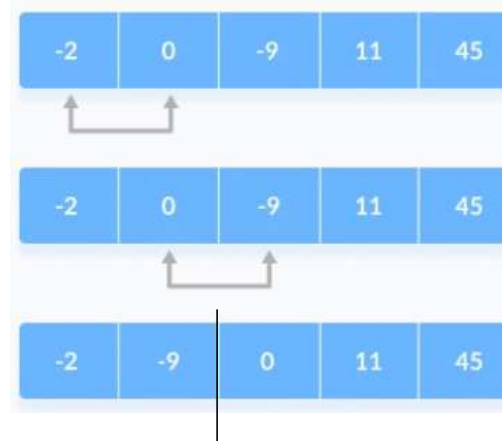
*first pass (i = 1)*



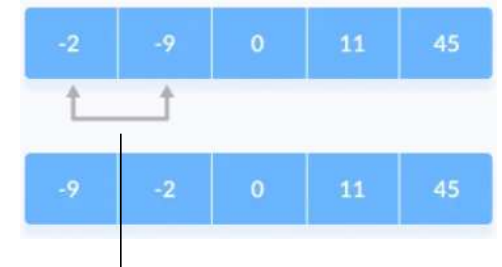
*second pass (i = 2)*



*third pass (i = 3)*



*forth pass (i = 4)*



After each iteration, the largest element among the unsorted elements is placed at the end.

The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

# Exact Analysis of Bubble sort

Sorts a given array by bubble sort

**Input:** An array  $A[1 \dots n]$  of orderable elements

**Output:** Array  $A[1 \dots n]$  sorted in ascending order

**ALGORITHM** *BubbleSort* ( $A[1 \dots n]$ )

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

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

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

        swap  $A[j]$  and  $A[j + 1]$



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

# Approximate Analysis of Bubble sort

**ALGORITHM** *BubbleSort* (A [1 .... n])

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

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

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

swap  $A[j]$  and  $A[j + 1]$

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n-1} [n - i]$$

Identical to the sum for selection sort

Thus, Bubble sort is a  $\theta(n^2)$



Can we improve the crude version of bubble sort??

**Hint:** What if a pass through the list makes no changes?

**Note:** A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.

## Selection sort vs Bubble sort

Algorithm	Time complexity		
	Best	Worst	Average
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$

العثماني

We will explore and analyze other sorting algorithms later in this course

# Matrix Multiplication

- If A is an  $m \times n$  matrix and B is an  $n \times p$  matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

- The *matrix product* C = AB is defined to be the  $m \times p$  matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

- Such that:

$$\underline{c_{ij}} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}, \quad \text{for } i = 1 \dots m \text{ and } j = 1 \dots p$$

That is  $c_{ij}$  is the dot product of the  $i^{\text{th}}$  row of A and the  $j^{\text{th}}$  column of B.

# Example

"Dot Product"

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix} \checkmark$$

# Analysis of matrix multiplication

SQUARE-MATRIX-MULTIPLY( $A, B$ )

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

□ Recall: We performed an exact analysis of matrix multiplication algorithm in chapter 2

the SQUARE-MATRIX-MULTIPLY procedure takes  $\theta(n^3)$  time

## □ Approximate analysis

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

□ Can we find a better algorithm? (Later in chapter 4)

# Sequential/Linear Search

- ❑ Searches for a given item (some search key  $X$ ) in a list of  $n$  elements by checking successive elements of the list until either a match with the search key is found (successful search) or the list is exhausted without finding a match (unsuccessful search)
- ❑ **Pseudocode** of Linear sequential search algorithm:

## ALGORITHM *SequentialSearch* ( $A, n, x$ )

// Input: An array of  $n$  elements and a search key  $X$   
// Output: The index of the first element in  $A[1..n]$  whose value is  
// equal to  $K$  or  $-1$  if no such element is found

$i \leftarrow 1$

**while**  $i \leq n$  **and**  $A[i] \neq X$

$i \leftarrow i + 1$

**if**  $i \leq n$  **return**  $i$

**else return**  $-1$

We analyzed the efficiency of sequential search in the worst, best, and average cases in chapter 2

We will discuss later in this course other searching algorithms with a better time efficiency



# Brute Force String Marching

- ❑ **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.

## ❑ Brute-force algorithm

- Step 1 Align the pattern at the beginning of the text
- Step 2 Moving from left to right, compare each character of the pattern to the corresponding character in text until
  - all characters are found to match (successful search); or
  - a mismatch is detected
- Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

# Brute Force String Marching

- ❑ **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.

## ❑ Brute-force algorithm

- Step 1 Align the pattern at the beginning of the text
- Step 2 Moving from left to right, compare each character of the pattern to the corresponding character in text until
  - all characters are found to match (successful search); or
  - a mismatch is detected
- Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

# Brute Force String Matching

□ Example: Text = HEY\_HELLO

Pattern = HELLO

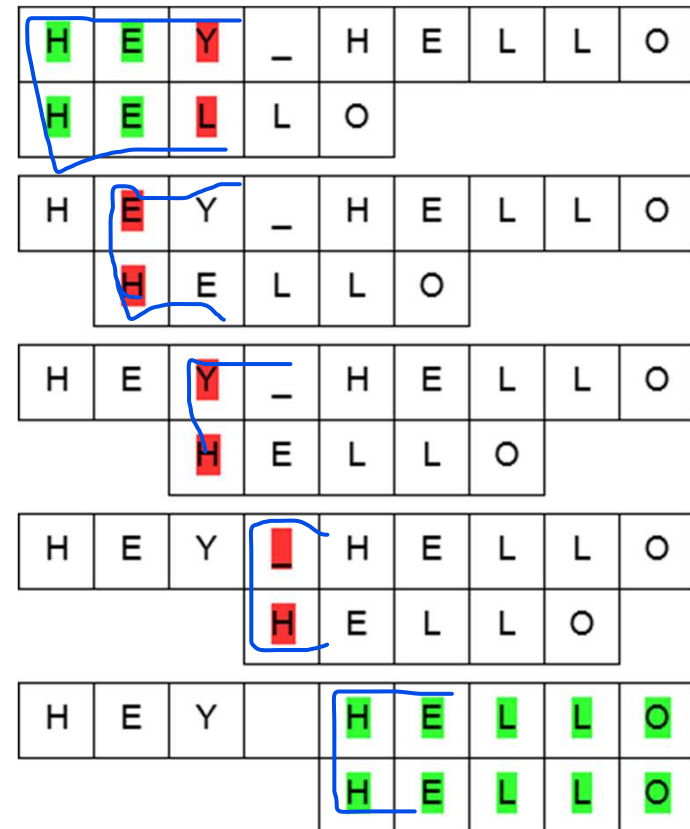
■ mismatch

■ match

$i = 1$

$i = 2$

⋮



# Brute Force String Marching

❑ **Pseudocode** (provided the text positions are indexed from 1)

**ALGORITHM** *BruteForceStringMatch* ( $T[1 \dots n], P[1 \dots m]$ )

// Input: An array  $T[1 \dots n]$  of  $n$  characters representing a **text**

          An array  $P[1 \dots m]$  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**  $i = 1$  **to**  $n - m + 1$  **do**

$j = 1$

**while**  $j \leq m$  **and**  $P[j] = T[i + j - 1]$  **do**

$j \leftarrow j + 1$

**if**  $j = m + 1$  **return**  $i$

**return**  $-1$

# Analysis of Brute Force String Matching

## Worst Case:

- 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.

- The algorithm makes  $m(n - m + 1)$  character comparisons

$O(n \cdot m)$

## Best Case

$$mn - m^2 + m \quad m \leq n$$

- The algorithm may have to make all  $m$  comparisons for just one try (the first comparison with the first pattern succeeds immediately)

- The algorithm makes  $m$  character comparisons

$O(m)$

Text: **a a b a** h n f g

Pattern: **a a b a**



also when  $n = m$   
لوب و حرة بنكضي

There are several more sophisticated and more efficient algorithms for string searching. The most widely known of them—by R. Boyer and J. Moore—(Boyer-Moore) is outlined in Section 7.2

## Exercise

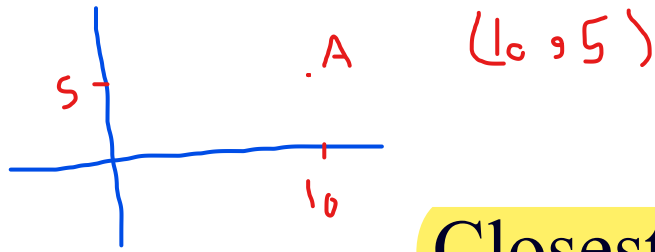
- Find the total number of character comparisons that will be made by the brute force algorithm in searching for a pattern of 4 characters in a text of 15 characters.

- 1) In the best case.
- 2) In the worst case.

$m$

$n$

$n \neq m$



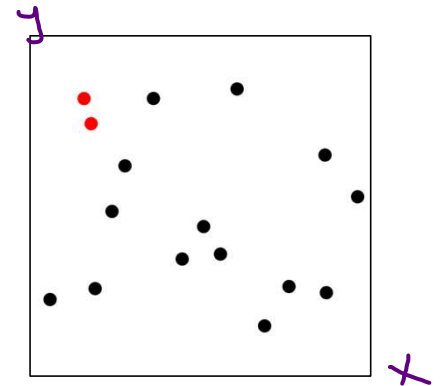
## Closest-pair problem

- Closest-pair problem: Find the two closest points in a set of  $n$  points (in the two-dimensional Cartesian plane).
- arise in two important applied areas: computational geometry and operations research

$$P_1 (x_1, y_1)$$

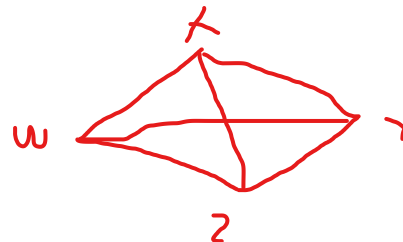
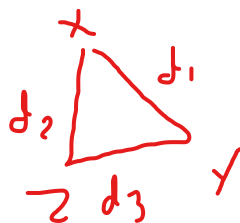
$$P_2 (x_2, y_2)$$

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



### Brute-force algorithm

- Compute the distance between every pair of distinct points.
- and return the indexes of the points for which the distance is the smallest.



# Closest-pair Brute Force Algorithm

## ALGORITHM *BruteForceClosestPoints(P)*

//Input: A list  $P$  of  $n$  ( $n \geq 2$ ) points  $P_1 = (x_1, y_1), \dots, P_n = (x_n, y_n)$

//Output: Indices  $index1$  and  $index2$  of the closest pair of points

$dmin \leftarrow \infty$

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

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

$d \leftarrow \text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2)$  //sqrt is the square root function

if  $d < dmin$

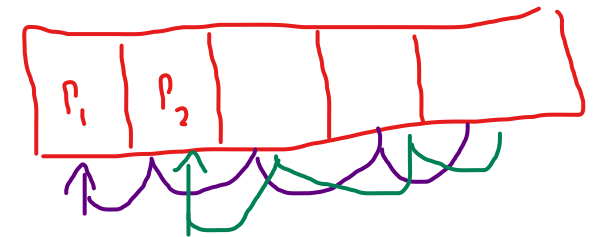
$dmin \leftarrow d; index1 \leftarrow i; index2 \leftarrow j$

return  $index1, index2$

Note: the distance between two points  $p_i(x_i, y_i)$  and  $p_j(x_j, y_j)$  is the standard

**Euclidean distance**

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$



The basic operation of the algorithm will be **squaring a number**. The number of times it will be executed can be computed as follows:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} 2 = 2 \sum_{i=1}^{n-1} [n - i] \quad \theta(n^2)$$





Exhaustive search

# Exhaustive search

- ❑ A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.
- ❑ It is known as **brute-force search**.
- ❑ It is also known as **generate and test**.

consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

- ❑ While a brute-force search is **simple to implement** and will always find a solution if it exists, **implementation costs are proportional to the number of candidate solutions** – which in many practical problems tends to **grow very quickly as the size of the problem increases**.

# Exhaustive search method

## Method:

- **Generate** a list of all potential solutions to the problem in a systematic manner.
- **Evaluate** potential solutions one by one, **disqualifying infeasible ones** and, for an optimization problem, keeping track of the best one found so far
- when search ends, **announce** the solution(s) found

# Three important problems

---

**To illustrate exhaustive search, we will apply it to three important problems:**

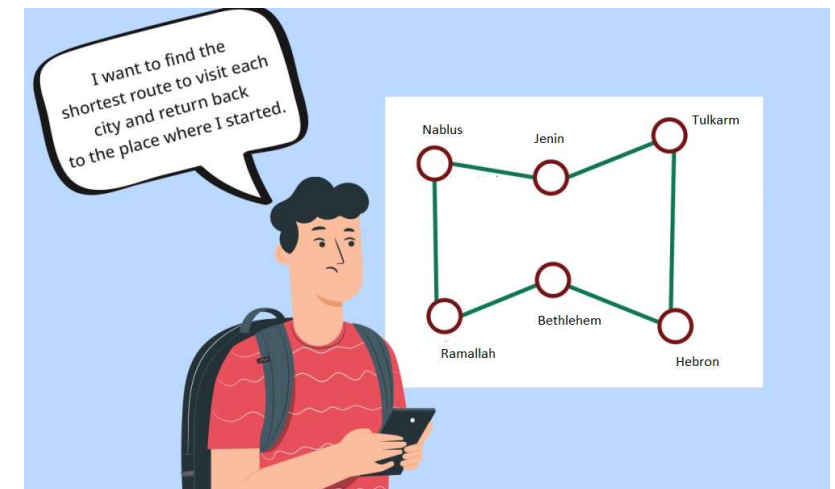
- Travelling salesman problem.
- Knapsack problem.
- Assignment problem.

# Travelling Salesman problem (TSP)

- ❑ **TSP problem:** Given  $n$  cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- ❑ TSP as **a graph problem:**
- ❑ The problem can be conveniently modeled by a **weighted graph**, with the graph's vertices representing the cities, graph's edges representing the paths, and the edge weights specifying the distances.

TSP problem	TSP as graph problem
Cities	Graph's vertices
paths	Graph's edges
distances	Edge weights

- ❑ Then the problem can be stated as the problem of finding the shortest **Hamiltonian circuit** of the graph



# TSP by Exhaustive Search

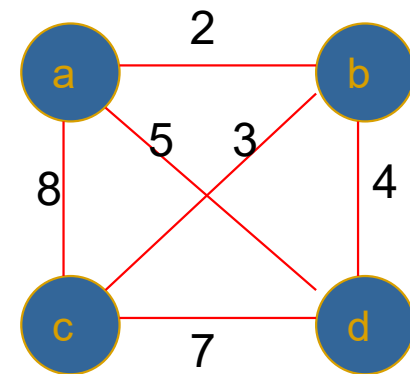
## ❑ Exhaustive search to solve the TSP:

- Get all tours by **generating all the permutations** of  $n - 1$  intermediate cities.
- **Evaluate** each tour length.
- Find the shortest among them

The total number of permutations needed is  **$(n-1)!$**

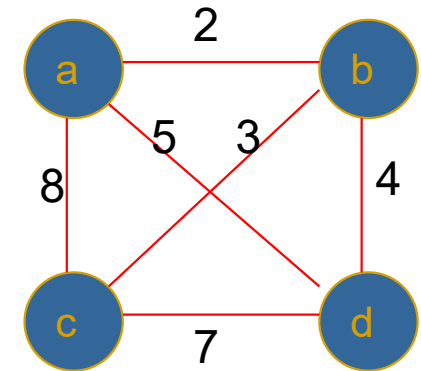
## ❑ Example:

- Consider city (a) as the starting and ending point.
- Generate all  $(n-1)!$  Permutations of cities.
- Calculate cost of every permutation and keep track of minimum cost permutation
- Return the permutation with minimum cost.



# TSP by Exhaustive Search

<u>Tour</u>	<u>Length (Cost)</u>	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$2+3+7+5 = 17$	optimal
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$2+4+7+8 = 21$	
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$8+3+4+5 = 20$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$8+7+4+2 = 21$	
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$5+4+3+8 = 20$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$5+7+3+2 = 17$	optimal



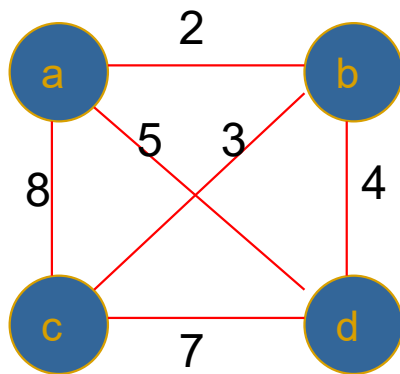
More tours????

Less tours????

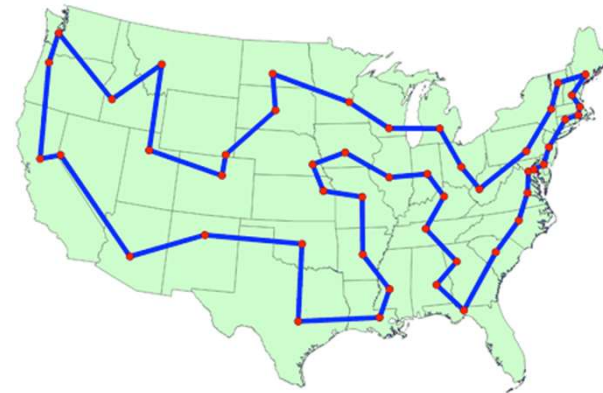
# TSP by Exhaustive Search

- ❑ What is the time efficiency of the exhaustive search?  $O(n!)$  where  $n$  is the number of cities (vertices)
- ❑ The exhaustive-search approach is **practical for very small values of  $n$** .

This solution becomes **impractical** even for only 20 cities



Small instance of TSP

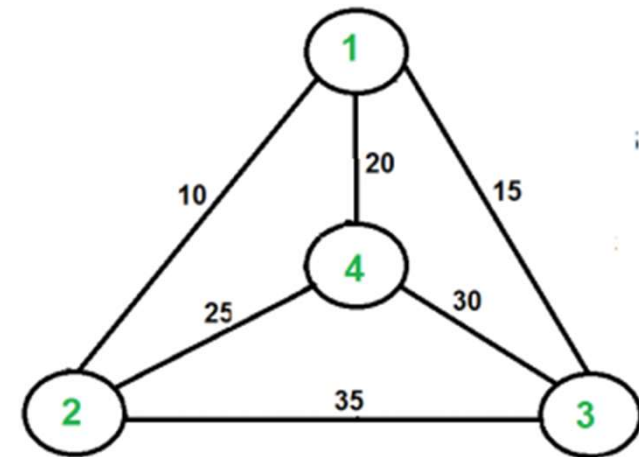


Large instance of TSP



# Exercise

- ❑ Consider the graph shown. Find the shortest possible route that visits every city exactly once before returning to the starting city. What is the cost of the tour?
- ❑ Note: Consider city 1 as the starting and ending point

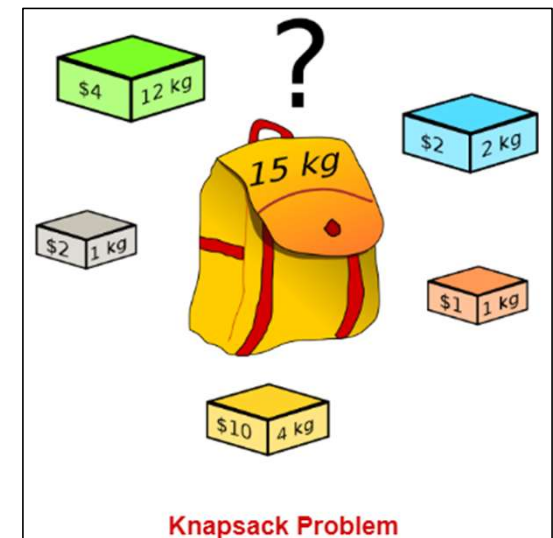


# Knapsack Problem

❑ **Knapsack problem:** Given  $n$  items  $I_1, I_2, \dots, I_n$  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.

- ❑ Which items should be placed into the knapsack such that:
  - The **value or profit** obtained by putting the items into the knapsack is **maximum**.
  - And the weight limit of the knapsack **does not exceed**.



❑ Application example:

- A transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity

# Knapsack by Exhaustive Search

## ❑ Exhaustive search to solve the 0/1 Knapsack problem:

- Generate **all the subsets** of the set of  $n$  items given.
- **Compute 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)
- Find a subset of the largest value among them Example

Since the number of subsets of an  $n$ -element set is  $2^n$ ,

The exhaustive search leads to a  $\Omega(2^n)$  algorithm, **no matter how efficiently individual subsets are generated**

# Knapsack by Exhaustive Search

## □ Example on small instance of Knapsack problem:

Subset	Total weight	Total value
$\emptyset$	0	\$0
{1}	2	\$20
{2}	5	\$30
{3}	10	\$50
{4}	5	\$10
{1,2}	7	\$50
{1,3}	12	\$70
{1,4}	7	\$30
<b>{2,3}</b>	<b>15</b>	<b>\$80</b>
{2,4}	10	\$40
{3,4}	15	\$60
{1,2,3}	17	not feasible
{1,2,4}	12	\$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

← optimal

Knapsack capacity  $W=16$

item	weight	value
1	2	\$20
2	5	\$30
3	10	\$50
4	5	\$10

Since number of items ( $n$ ) is small, so it is feasible to find the optimal solution by brute force (using a computer)

But if  $n$  is much bigger, the brute force would take too long.

# Exercise

- ❑ You are given 5 elements needed to be put inside a knapsack with limited weight capacity  $W = 50$ . The elements weights with their values are given below:

item	1	2	3	4	5
Weight	21	11	51	26	30
value	37	12	500	50	41

Apply brute force technique to find which subset of items should be put inside the knapsack such that the obtained profit is maximized and the weight limit of the knapsack does not exceed.

# Exercise

- ❑ You want to travel to someplace for a few days and have a single bag to carry which could fit 5 kgs. You have a list of items to take which could all not fit into that single bag. Assuming you cant take fractions of items, which items should be picked up to maximize the value of items according to your needs.

item	1	2	3	4
Weight	2	3	4	5
value	3	4	5	6

# Inefficient exhaustive search

For both the traveling salesman and knapsack problems considered before, **exhaustive search leads to algorithms that are extremely inefficient on every input.**

❑ In fact, these two problems are the best-known examples of so called **NP-hard** problems.

❑ We will study other techniques to solve **some but not all instances** of these problems in less than exponential and factorial times.