

# Design and Analysis of Algorithms (UE18CS251)

## Unit II - Brute Force

Mr. Channa Bankapur  
channabankapur@pes.edu

## **Brute Force:**

A straightforward approach, usually directly based on the problem statement and definitions of the concepts involved.

### Examples:

1. Sequentially searching for a key of a given value in a list
  2. Computing  $n!$  by finding the product of 1 thru  $n$ .
  3. Computing  $a^n$  ( $a > 0$ ,  $n \geq 0$ ) by multiplying 'a'  $n$  times.
  4. Hacking a password by matching all possible passwords.
- “Force” by the computer in terms of effort, but simple in strategy to implement.
  - Trial and Error method of trying out in some order.
  - Exhaustive effort rather than employing intellectual strategies.
  - Often one of the easiest way to solve it.

## Algorithm SequentialSearch(A[0..n-1], K)

//Outputs the index of the **first** element of A that  
// matches K or -1 if there are no matching elements.

**i**  $\leftarrow$  0

**while** (**i** < **n**) **and** (**A**[**i**]  $\neq$  **K**) **do**

**i**  $\leftarrow$  **i** + 1

**if** (**i** < **n**) **return** **i**

**return** -1

Input size: n.

Basic Operation: (**i** < **n**) **and** (**A**[**i**]  $\neq$  **K**)

$$C_{\text{worst}}(n) = n + 1$$

$$C_{\text{best}}(n) = 1$$

Let '**p**' be the probability of the search key present in the array.

$$C_{\text{avg}}(n) = p(n + 1) / 2 + (1 - p)(n + 1)$$

## Algorithm SequentialSearch( $A[0..n-1]$ , $K$ )

Input Size:  $n$

Basic Operation :  $(i < n)$  and  $(A[i] \neq K)$

$C_{\text{worst}}(n)$  = Count of the basic operation at the max  
 $= n + 1 \in \Theta(n)$

$C_{\text{best}}(n) = 1 \in \Theta(1)$

$C_{\text{avg}}(n)$  = from  $(n+1)/2$  to  $(n+1)$  depending on the probability of search key being present in the input array.

$C_{\text{avg}}(n) \in \Theta(n)$

## Brute-Force String Matching:

In an **n**-characters **text**, search for the first occurrence of **m**-character **pattern**.

There are **n-m+1** substrings of length **m** in the text of length **n**. Find the first such substring which matches with the pattern.

Find **i**, the index of the leftmost character of the first matching substring in the text such that

$$\begin{array}{ccccccccccc} t_0 & \dots & t_i & \dots & t_{i+j} & \dots & t_{i+m-1} & \dots & t_{n-1} \\ & & \updownarrow & & \updownarrow & & \updownarrow & & \\ & & p_0 & \dots & p_j & \dots & p_{m-1} & & \text{pattern } P \end{array}$$

$$t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$$

```
NaiveStringMatch(T[0..n-1], P[0..m-1])
  for i ← 0 to n-m
    j ← 0
    while (j < m and T[i+j] = P[j])
      j ← j + 1
    if (j = m) return i
  return -1
```

Analysis:

Input Size:  $n, m$

Basic Operation : ( $T[i+j] = P[j]$ )

...

Input Size:  $n, m$

Basic Operation :  $(T[i+j] = P[j])$

$$\begin{aligned}C_{\text{best}}(n) &= \mathbf{0}, \text{ when } n < m \\&= \mathbf{\min(m, n-m+1)} \text{ when } n \geq m \\&= \mathbf{1} \text{ when } n = m \\&= \mathbf{m}, \text{ when } n \geq 2m-1 \text{ (i.e., } n-m+1 \text{ is at least } m) \\&= \mathbf{\max(0, \min(m, n-m+1))}, \text{ for any } n, m\end{aligned}$$

$$C_{\text{best}}(n) \in \mathbf{\Theta(m)} \text{ with } n \geq 2m-1$$

$$C_{\text{best}}(n) \in \mathbf{O(m)} \text{ for any } n, m$$

$$C_{\text{worst}}(n) = (n - m + 1) * m \in \mathbf{\Theta(nm)}$$

$$C_{\text{avg}}(n) \in \mathbf{O(nm)}$$

**Write an algorithm to check if the array is sorted.**

```
boolean isSorted( A[0..n-1] )
```

```
//Checks if the array A is sorted.
```

```
//Input: An array A of orderable elements by  $\leq$ .
```

```
//Output: Return TRUE if array is sorted.
```

```
//          FALSE otherwise.
```

```
...
```



**Write an algorithm to check if the array is sorted.**

```
boolean isSorted( A[0..n-1] )  
//Checks if the array A is sorted.  
//Input: An array A of orderable elements by  $\leq$ .  
//Output: Return TRUE if array is sorted.  
//          FALSE otherwise.  
for i  $\leftarrow$  0 to n-2  
    if (A[i] > A[i+1]) //not in order  
        return FALSE  
return TRUE
```

**Sort by fixing the problems while checking for sortedness.**

**SortByCheckingSortedness ( A[0..n-1] )**

//Sorts by Checking sortedness.

//Input: An array **A** of orderable elements by  $\leq$ .

//Output: Sorted array A.

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

**if (A[i] > A[i+1])**

**Swap A[i] with A[i+1]**

Does it sort?

**Sort by fixing the problems while checking for sortedness.**

**SortByCheckingSortedness2( A[0..n-1] )**

//Sorts by Checking sortedness.

//Input: An array **A** of orderable elements by  $\leq$ .

//Output: Sorted array A.

**while (TRUE)**

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

**if (A[i] > A[i+1])**

**Swap A[i] with A[i+1]**

**if (isSorted( A[0..n-1] ))**

**return**

Does it sort and that too in a finite amount of time?

**Sort by fixing the problems while checking for sortedness.**

```
SortByCheckingSortedness3( A[0..n-1] )
```

```
//Sorts by Checking sortedness.
```

```
//Input: An array A of orderable elements by  $\leq$ .
```

```
//Output: Sorted array A.
```

```
    for i  $\leftarrow$  0 to n-2 //n-1 consecutive pairs
```

```
        if (A[i] > A[i+1])
```

```
            Swap A[i] with A[i+1]
```

```
    if (isSorted( A[0..n-1] ))
```

```
        return
```

```
    SortByCheckingSortedness3 (A[0..n-2])
```

It should sort.

Can we move `isSorted()` logic into the `for` loop?

**Algorithm SortByCheckingSortedness4 (A[0..n-1])**

//Sorts by an improved Bubble Sort algorithm.

//Input: An array **A** of orderable elements by  $\leq$ .

//Output: Sorted array **A**.

**isSorted**  $\leftarrow$  **TRUE**

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

**if** (A[i] > A[i+1])

**Swap** A[i] with A[i+1]

**isSorted**  $\leftarrow$  **FALSE**

**if** (isSorted = **TRUE**)

**return**

**SortByCheckingSortedness4** (A[0..n-2])

Base case for the recursion?

**Algorithm SortByCheckingSortedness5 (A[0..n-1])**

//Sorts by an improved Bubble Sort algorithm.

//Input: An array **A** of orderable elements by  $\leq$ .

//Output: Sorted array **A**.

**if** (n  $\leq$  1) **return**

**isSorted**  $\leftarrow$  **TRUE**

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

**if** (A[i] > A[i+1])

**Swap** A[i] with A[i+1]

**isSorted**  $\leftarrow$  **FALSE**

**if** (**isSorted** = **TRUE**)

**return**

**SortByCheckingSortedness5** (A[0..n-2])

Can we convert the recursive algorithm to iterative?

**SortByCheckingSortedness6( A[0..n-1] )**

//Sorts by Checking sortedness.

//Input: An array **A** of orderable elements by  $\leq$ .

//Output: Sorted array A.

**m**  $\leftarrow$  **n**

**while** (**m**>1)

**isSorted**  $\leftarrow$  **TRUE**

**for** **i**  $\leftarrow$  0 **to** **m-2** //m-1 consecutive pairs

**if** (**A**[**i**] > **A**[**i+1**])

**Swap** **A**[**i**] **with** **A**[**i+1**]

**isSorted**  $\leftarrow$  **FALSE**

**if** (**isSorted** = **TRUE**) **return**

**m**--

**return**

```
Algorithm BubbleSort( A[0..n-1] )  
//Sorts by an improved Bubble Sort algorithm.  
//Input: An array A of orderable elements by  $\leq$ .  
//Output: Sorted array A.  
for i  $\leftarrow$  n-1 downto 1 //n-1 passes  
    isSorted  $\leftarrow$  TRUE  
    for j  $\leftarrow$  0 to i-1 //last n-i-1 elements are sorted  
        if (A[j] > A[j+1])  
            Swap A[j] with A[j+1]  
            isSorted  $\leftarrow$  FALSE  
    if (isSorted = TRUE)  
        return  
return
```



$$A_0, \dots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \dots, A_{n-i-1} \mid A_{n-i} \leq \dots \leq A_{n-1}$$

in their final positions

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

//Sorts by Bubble Sort algorithm.

//Input: An array **A** of orderable elements by  $\leq$ .

//Output: Sorted array A.

**for** i  $\leftarrow$  n-1 **downto** 1 //n-1 passes

**for** j  $\leftarrow$  0 **to** i-1 //last n-i-1 elements are sorted

**if** (A[j] > A[j+1])

            Swap A[j] with A[j+1]

**return**

5	1	12	-5	16
---	---	----	----	----

unsorted

5	1	12	-5	16
---	---	----	----	----

5 > 1, swap

1	5	12	-5	16
---	---	----	----	----

5 < 12, ok

1	5	12	-5	16
---	---	----	----	----

12 > -5, swap

1	5	-5	12	16
---	---	----	----	----

12 < 16, ok

1	5	-5	12	16
---	---	----	----	----

1 < 5, ok

1	5	-5	12	16
---	---	----	----	----

5 > -5, swap

1	-5	5	12	16
---	----	---	----	----

5 < 12, ok

1	-5	5	12	16
---	----	---	----	----

1 > -5, swap

-5	1	5	12	16
----	---	---	----	----

1 < 5, ok

-5	1	5	12	16
----	---	---	----	----

-5 < 1, ok

-5	1	5	12	16
----	---	---	----	----

sorted

## Analysis of Bubble Sort:

```
Algorithm BubbleSort( A[0..n-1] )  
//Sorts by Bubble Sort algorithm.  
//Input: An array A of orderable elements by  $\leq$ .  
//Output: Sorted array A.  
for  $i \leftarrow 0$  to  $n-2$  //n-1 passes  
    for  $j \leftarrow 0$  to  $n-2-i$  //last i elements are sorted  
        if ( $A[j] > A[j+1]$ )  
            Swap  $A[j]$  with  $A[j+1]$   
return
```

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

Input Size:  $n$

Basic Operation:  $(A[j] > a[j+1])$

$$C_{\text{worst}}(n) = n * (n - 1) / 2 \in \Theta(n^2)$$

$$C_{\text{best}}(n) = n * (n - 1) / 2 \in \Theta(n^2)$$

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

## **Bubble Sort (improved version):**

**Algorithm BubbleSortImproved( A[0..n-1] )**

**//Sorts by an improved Bubble Sort algorithm.**

**//Input: An array **A** of orderable elements by  $\leq$ .**

**//Output: Sorted array A.**

**for i  $\leftarrow$  0 to n-2 //n-1 passes**

**anySwaps  $\leftarrow$  FALSE**

**for j  $\leftarrow$  0 to n-2-i //last i elements are sorted**

**if (A[j] > A[j+1])**

**Swap A[j] with A[j+1]**

**anySwaps  $\leftarrow$  TRUE**

**if (anySwaps = FALSE)**

**return**

**return**

**Algorithm BubbleSortImproved( A[0..n-1] )**

Input Size: **n**

Basic Operation : **(a[i] > a[i+1])**

$$C_{\text{worst}}(n) = n * (n - 1) / 2 \in \Theta(n^2)$$

$$C_{\text{best}}(n) = (n - 1) \in \Theta(n)$$

$$C_{\text{avg}}(n) \in O(n^2) \in \Theta(?)$$

Basic Operation : **Swap**

$$C_{\text{worst}}(n) = n * (n - 1) / 2 \in \Theta(n^2)$$

$$C_{\text{best}}(n) = 0 \in \Theta(1)$$

Yet another way of **sorting** by brute-force.

**Ex.:3.21 Arrange the following numbers in the ascending order :**

243   284   197   314   547

197 , 243 , 284 , 314 , 547

814   749   119   864   999

119 , 749 , 814 , 864 , 999

450   970   839   329   146

146 , \_\_\_\_\_ , \_\_\_\_\_ , \_\_\_\_\_ , \_\_\_\_\_

105   109   218   174   80

\_\_\_\_\_ , \_\_\_\_\_ , \_\_\_\_\_ , \_\_\_\_\_ , \_\_\_\_\_

## **Selection Sort:**

Find the smallest of the unsorted array and place it at the beginning of the unsorted array. Reduce the unsorted array by excluding the first one, which is already in its final position. Repeat sorting the unsorted array as long as there is only one element left in the unsorted array.

### **Algorithm SelectionSort\_Recursive(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 ascending order.

**if**( $n \leq 1$ ) **return**

**min**  $\leftarrow$  **minIndex**(A[0..n-1])

**Swap** A[0] **with** A[min]

**SelectionSort\_Recursive**(A[1..n-1])



## Selection Sort:

Find the smallest of the unsorted array and place it at the beginning of the unsorted array. Reduce the unsorted array by excluding the first one, which is already in its final position.

### Algorithm SelectionSort(A[0..n-1])

//Sorts a given array by Selection Sort.

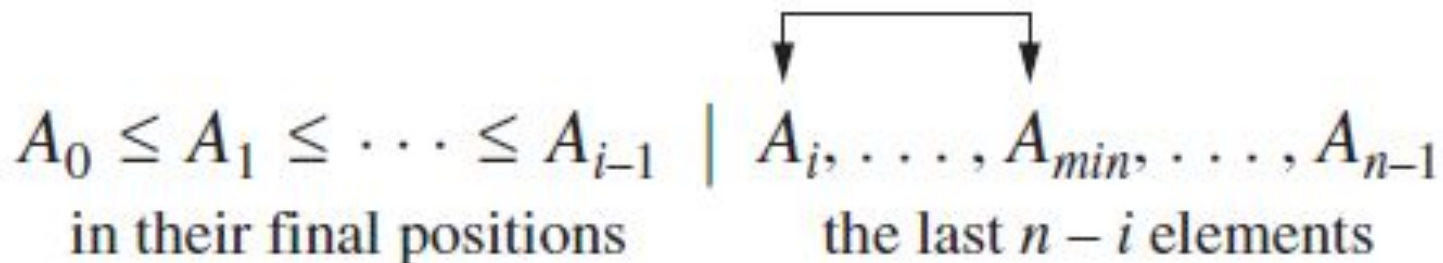
//Input: An array A[0..n-1] of orderable elements.

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

**for** i ← 0 to n-2

**min** ← minIndex(A[i..n-1])

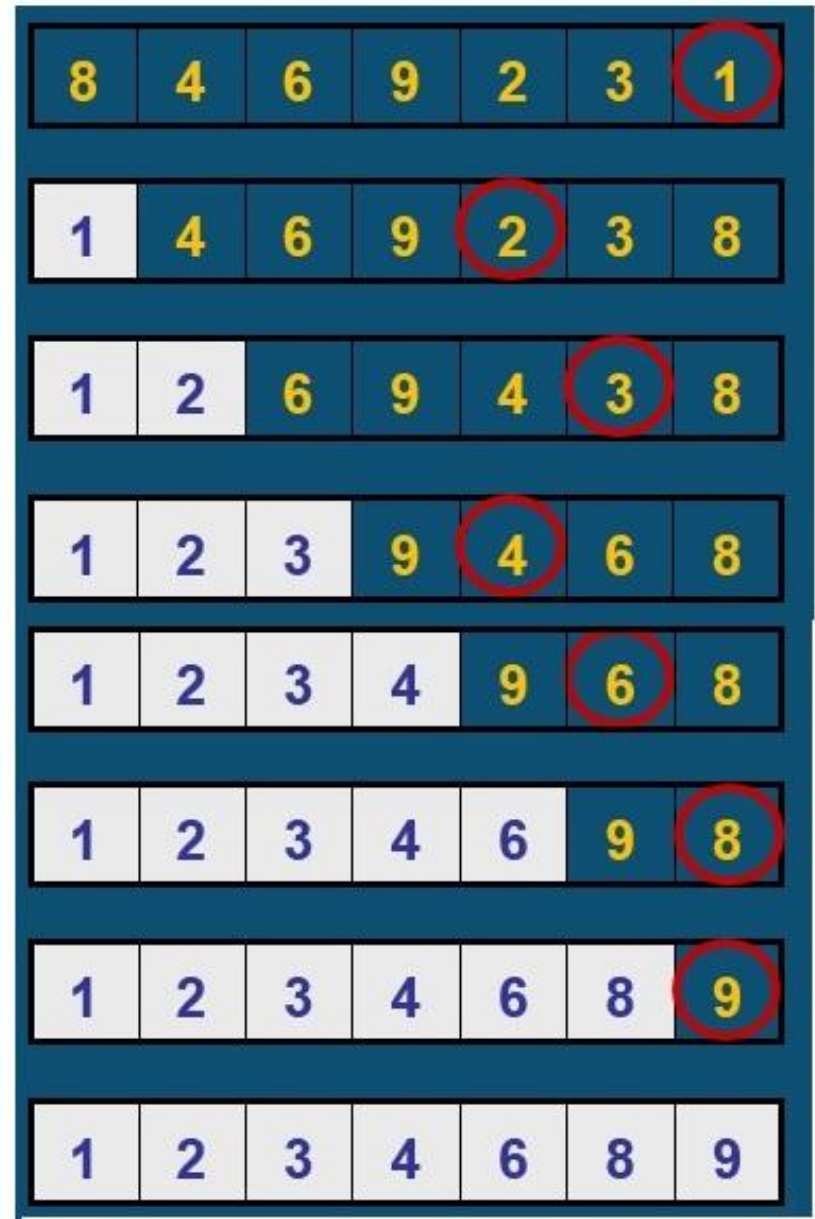
**Swap** A[i] with A[min]





## Selection Sort:

Example: 8 4 6 9 2 3 1



## Analysis of Selection Sort:

Input Size:  $n$  (size of the list)

Basic Operation: Comparison ( $A[j] < A[\min]$ )

$$C_{\text{best}}(n) = C_{\text{worst}}(n) = C_{\text{avg}}(n)$$

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

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

## Polynomial Evaluation:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 \quad \text{at } x=x_0$$

**Algorithm Polynomial(n,  $x_0$ , a[0..n])**

//Output: Value of the polynomial at  $x = x_0$ .

...

## Polynomial Evaluation:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 \quad \text{at } x=x_0$$

**Algorithm Polynomial( $n, x_0, a[0..n]$ )**

//Output: Value of the polynomial at  $x = x_0$ .

**p**  $\leftarrow$  0.0

**for** i  $\leftarrow$  n downto 0 **do**

**power**  $\leftarrow$  1

**for** j  $\leftarrow$  1 to i **do** //compute  $x^i$

**power**  $\leftarrow$  power \*  $x_0$

**p**  $\leftarrow$  p +  $a_i$  \* power

**return** p

Input Size:

Basic Operation :

$C(n) =$

## Polynomial Evaluation:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 \quad \text{at } x=x_0$$

**Algorithm Polynomial( $n, x_0, a[0..n]$ )**

//Output: Value of the polynomial at  $x = x_0$ .

**p**  $\leftarrow$  0.0

**for** i  $\leftarrow$  n downto 0 **do**

**power**  $\leftarrow$  1

**for** j  $\leftarrow$  1 to i **do** //compute  $x^i$

**power**  $\leftarrow$  **power** \*  $x_0$

**p**  $\leftarrow$  **p** +  $a_i$  \* **power**

**return** p

Input Size: n

Basic Operation : **power**  $\leftarrow$  **power** \*  $x_0$

$C(n) = n * (n + 1) / 2 \in \Theta(n^2)$

```
p ← 0.0
for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do //compute  $x^i$ 
        power ← power * x
    p ← p + a[i] * power
return p
```

-----

```
p ← a[0]
power ← 1
for i ← 1 to n do
    power ← power * x
    p ← p + a[i] * power
return p
```



## Polynomial Evaluation:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 \quad \text{at } x=x_0$$

**Algorithm Polynomial2**( $n, x_0, a[0..n]$ )

//Output: Value of the polynomial at  $x = x_0$ .

**$p \leftarrow a_0$**

**power  $\leftarrow 1$**

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

**$power \leftarrow power * x$**  //compute  $x^i$

**$p \leftarrow p + a_i * power$**

**return  $p$**

Input Size:  $n$

Basic Operation :  **$power \leftarrow power * x_0$**

**$C(n) = n \in \Theta(n)$**

## **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.

### **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.

## Sorting by exhaustive search:

Find an arrangement of elements of an array, which is sorted.

Strategy: For every possible arrangements of an array, check if it's sorted, and return the arrangement which is sorted.

```
boolean SortByBruteForce( a[0..n-1] )  
//Sorts array a by trying out all possible arrangements of  
//elements of the array.  
//Input: An array a of orderable elements by  $\leq$ .  
//Output: Sorted array a by  $\leq$ .  
for each permutation p[0..n-1] of array a  
    if (isSorted( p[0..n-1] ))  
        return p[0..n-1]
```

## Travelling Salesman Problem:

1. Given  $n$  cities and distances between each pair of cities, find the shortest tour that passes through all other cities and returns to the origin city.
2. In case of a weighted complete graph, it's about finding the shortest *Hamiltonian circuit*.

Eg: Driving time between some 10 cities of India (Cost Matrix).

000000	110189	050573	020948	109480	034435	028433	074836	091767	068406
109006	000000	079663	118195	079397	143304	083593	045792	037923	068146
051516	080265	000000	070149	121881	083636	044745	043763	042416	067450
021557	119539	069838	000000	095820	042397	037471	084186	111032	077756
110053	081231	121373	095977	000000	134475	085826	087690	100264	054016
034488	144238	082769	041728	134042	000000	062482	108885	123963	102455
028473	084770	045153	037117	085732	062772	000000	049417	078006	042987
075056	046162	044536	084245	086579	109354	049641	000000	031151	038399
092933	037994	042414	111566	099497	125053	078960	031010	000000	068113
068718	068844	068336	077907	055357	103016	043305	038648	068634	000000

# Travelling Salesman Problem:

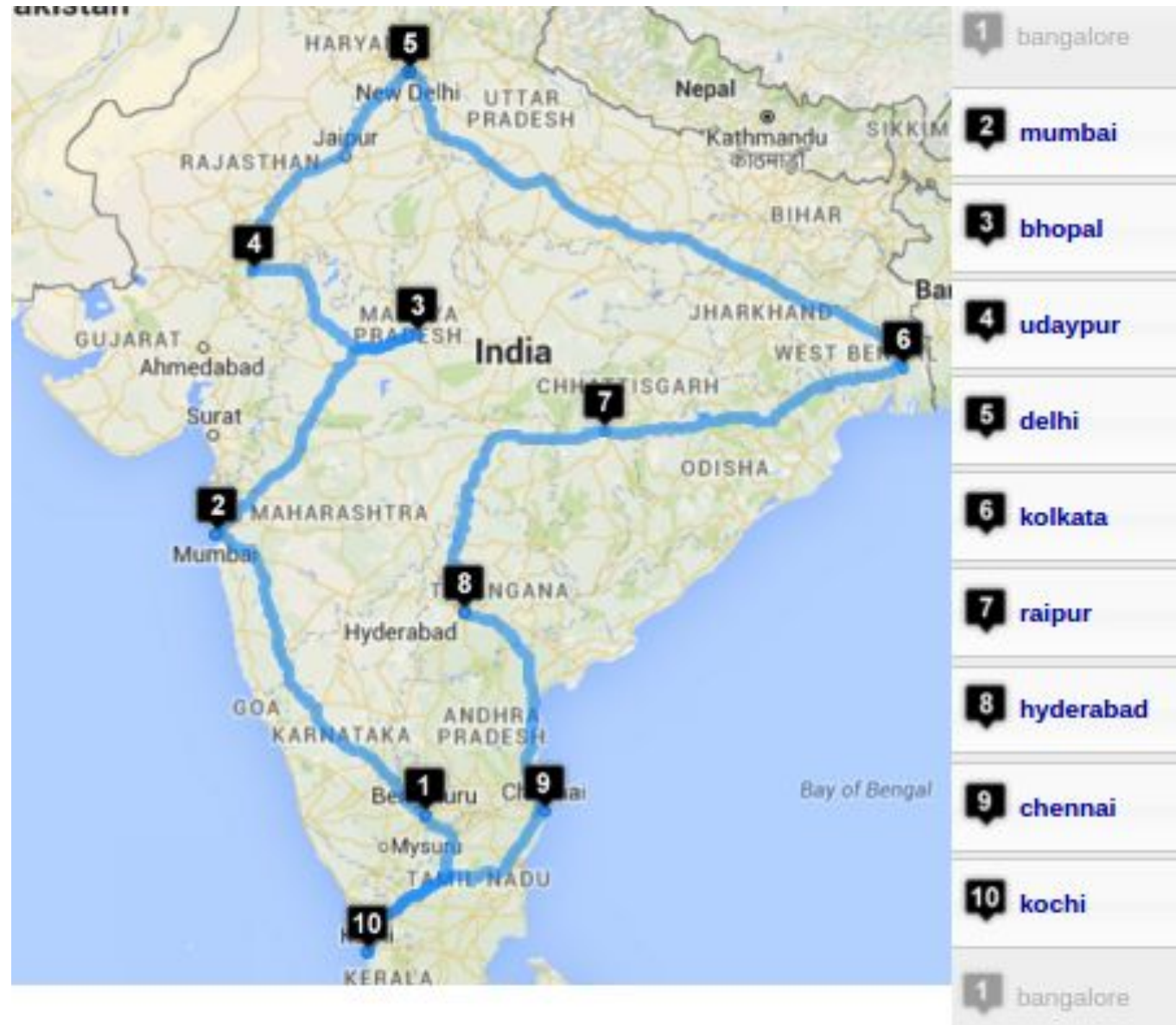
1. Bengaluru
2. New Delhi
3. Mumbai
4. Chennai
5. Kolkata
6. Kochi
7. Hyderabad
8. Bhopal
9. Udaipur
10. Raipur



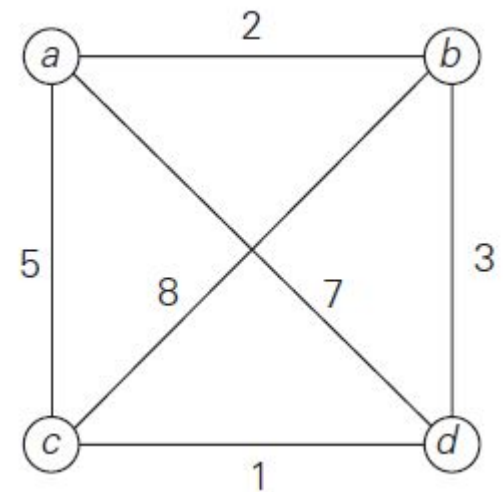
# Travelling Salesman Problem:

1. Bengaluru
2. Mumbai
3. Bhopal
4. Udaipur
5. New Delhi
6. Kolkata
7. Raipur
8. Hyderabad
9. Chennai
10. Kochi
11. Bengaluru

Shortest round  
trip takes  
**454201** sec.



# Travelling Salesman Problem:



Tour

Length

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

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

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

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

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

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

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

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

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

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

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

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

Algorithm **Travelling Salesperson Problem**

**mincost**  $\leftarrow$  Infinity

**for each** permutation of (n - 1) cities

**cost**  $\leftarrow$  0

**for each** edge in the Hamiltonian circuit

**cost**  $\leftarrow$  cost + cost of the edge

**if** (cost < mincost)

**mincost**  $\leftarrow$  cost

**return** mincost

Input Size: n

Basic Operation : addition of cost of an edge

$C(n) = n * (n - 1)! = n! \in \Theta(n!)$



# ALGORITHM **TravellingSalesmanProblem**

//Input:  $n \times n$  adjacency matrix  $A$ . Assumed  $n > 1$ .

//Output: Min Cost Hamiltonian circuit.

//getPermutation( $P[]$ ) returns true with next permutation in lexicographic

// order, if it exists. Returns false otherwise.

**mincost**  $\leftarrow$  **INFINITY**

**Permutation**[1.. $n-1$ ]  $\leftarrow$  [1, 2, 3, ...,  $n-1$ ] //1st permutation

**do**

**cost**  $\leftarrow$  **A**[0, **Permutation**[1]] //1st edge of the circuit

**for** **i**  $\leftarrow$  1 **to**  $n-2$

**cost**  $\leftarrow$  **cost** + **A**[**Permutation**[**i**], **Permutation**[**i**+1]]

**cost**  $\leftarrow$  **cost** + **A**[**Permutation**[ $n-1$ ], 0] //last edge

**if** (**cost** < **mincost**) **mincost**  $\leftarrow$  **cost**

**while** (**getNextPermutation**(**Permutation**[1.. $n-1$ ]))

**return** **mincost**

## The Assignment Problem:

Each one of  $n$  people are assigned one of  $n$  jobs, exactly one person per job. The cost of assigning person  $i$  to job  $j$  is  $C[i, j]$ . Find an assignment that minimizes the total cost.

	Job 0	Job 1	Job 2	Job 3
P0	9	2	7	8
P1	6	4	3	7
P2	5	8	1	8
P3	7	6	9	4

Exhaustive Search Strategy:

- 1.

## The Assignment Problem:

Each one of  $n$  people are assigned one of  $n$  jobs, exactly one person per job. The cost of assigning person  $i$  to job  $j$  is  $C[i, j]$ . Find an assignment that minimizes the total cost.

	Job 0	Job 1	Job 2	Job 3
P0	9	2	7	8
P1	6	4	3	7
P2	5	8	1	8
P3	7	6	9	4

### Exhaustive Search Strategy:

1. Generate all legitimate assignments,
2. Compute cost of each assignment, and
3. Select the cheapest one.

<b>Cost Matrix</b>	<b>Job 0</b>	<b>Job 1</b>	<b>Job 2</b>	<b>Job 3</b>
P0	9	2	7	8
P1	6	4	3	7
P2	5	8	1	8
P3	7	6	9	4

### Assignment (col.#s)

### Total Cost

1, 2, 3, 4

$$9+4+1+4 = 18$$

1, 2, 4, 3

$$9+4+8+9 = 30$$

1, 3, 2, 4

$$9+3+8+4 = 24$$

1, 3, 4, 2

$$9+3+8+6 = 26$$

1, 4, 2, 3

$$9+7+8+9 = 33$$

1, 4, 3, 2

$$9+7+1+6 = 23$$

2, 1, 3, 4

$$2+6+1+4 = 13$$

etc.

Algorithm **AssignmentProblem**( $C[0..n-1, 0..n-1]$ )

//Input:  $n \times n$  cost matrix  $C$ .

//Output: Cost of the cheapest assignment.

**mincost**  $\leftarrow$  Infinity

**for each** permutation of  $n$  jobs

**cost**  $\leftarrow 0$ , **person**  $\leftarrow 0$

**for each** job in the assignment

**cost**  $\leftarrow$  **cost** +  $C[\text{person}, \text{job}]$

**person**  $\leftarrow$  **person** + 1

**if** (**cost** < **mincost**) **mincost**  $\leftarrow$  **cost**

**return** **mincost**

**Input Size:**  $n$ , **Basic Operation :**  $\text{cost} + C[\text{person}, \text{job}]$

**$C(n) = n * n! \in \Theta(n * n!)$**

Algorithm **AssignmentProblem**(C[0..n-1, 0..n-1])

//Input: n x n cost matrix C.

//Output: Cost of the cheapest assignment.

**mincost**  $\leftarrow$  **Infinity**

**Permn**[0..n-1]  $\leftarrow$  [0,1,2,...,n-1] //1st permn.

**do**

**cost**  $\leftarrow$  0

**for** i  $\leftarrow$  0 to n-1

**cost**  $\leftarrow$  **cost** + C[i, **Permn**[i]]

**if**(**cost** < **mincost**) **mincost**  $\leftarrow$  **cost**

**while** (**getNextPermn** (**Permn**[0..n-1]))

**return mincost**

**Input Size:** n, **Basic Operation :** **cost** + C[i, **Permn**[i]]

**C(n)** = n \* n!  $\in \Theta(n*n!)$

# Knapsack Problem:

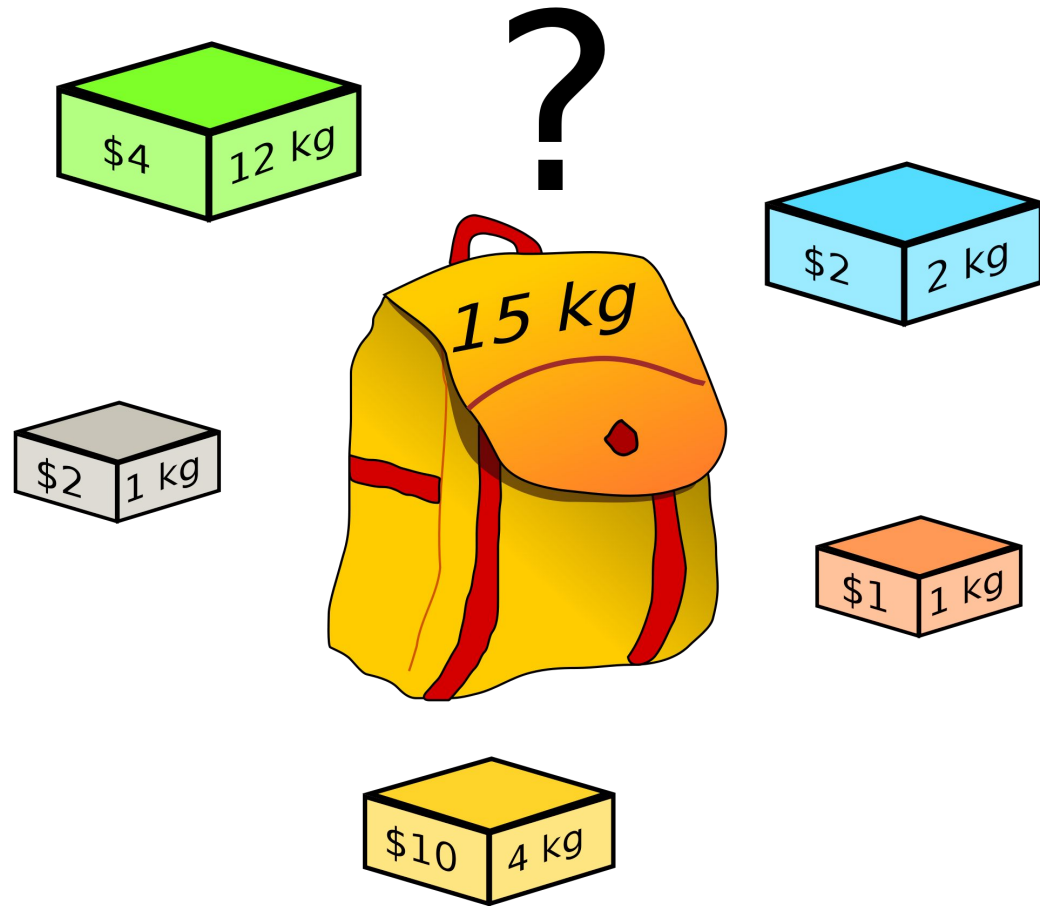
Given  $n$  items:

weights:  $w_1 \ w_2 \ \dots \ w_n$

values:  $v_1 \ v_2 \ \dots \ v_n$

a knapsack of capacity  $W$

Find most valuable subset  
of the items that fit into  
the knapsack.



## Knapsack Problem:

Given  $n$  items:

weights:  $w_1 \ w_2 \ \dots \ w_n$

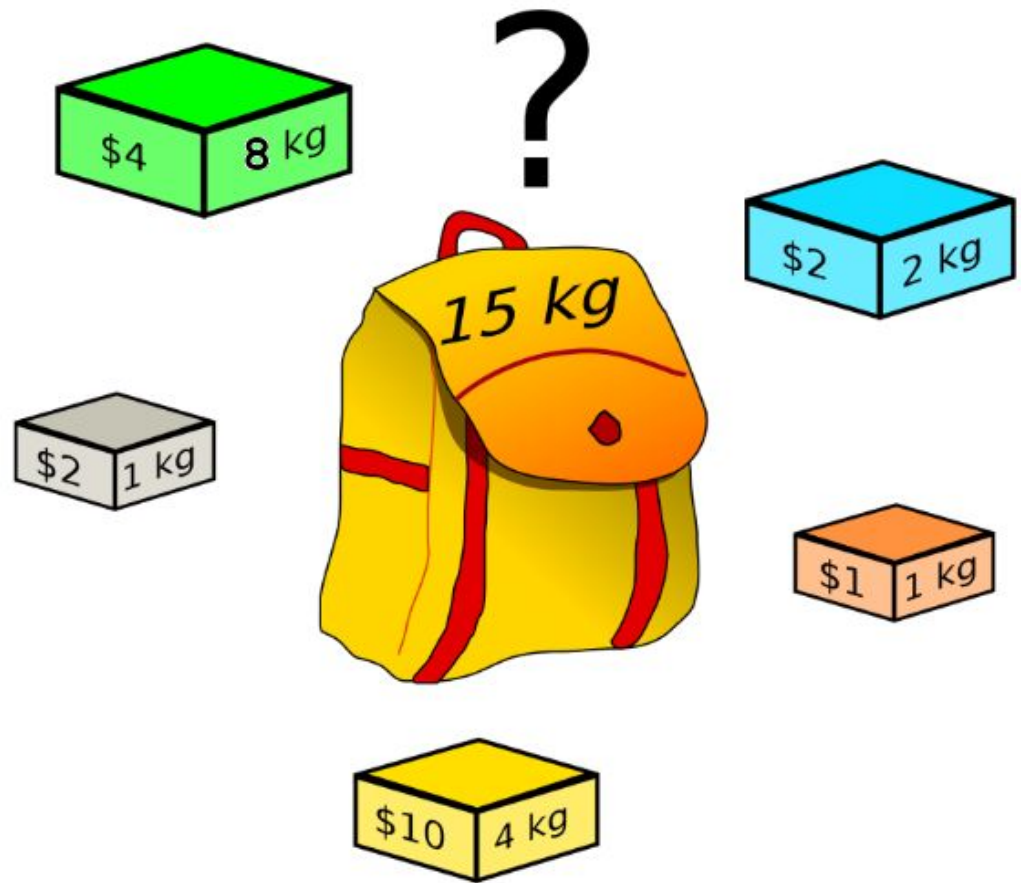
values:  $v_1 \ v_2 \ \dots \ v_n$

a knapsack of capacity  $W$

Find most valuable subset  
of the items that fit into the  
knapsack.

**Ans: {B,O,Y,W} 8kg, \$15**

What if the green object weighs  
8 kg instead of 12 kg?

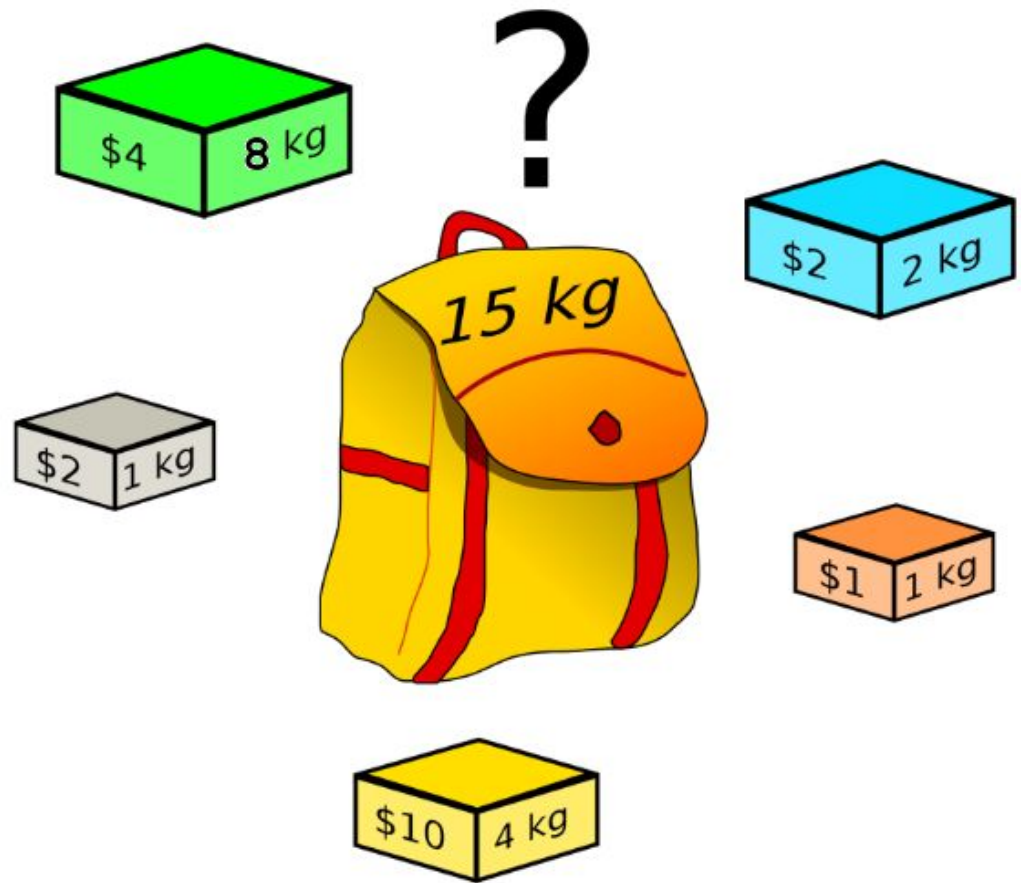




## Knapsack Problem:

What if the green object weighs 8 kg instead of 12 kg?

**Ans:**  
**{G,B,Y,W}**  
**15kg, \$18**



Example:

Knapsack capacity  $W=16$

<u>item</u>	<u>weight</u>	<u>value</u>
-------------	---------------	--------------

1	2	\$20
---	---	------

2	5	\$30
---	---	------

3	10	\$50
---	----	------

4	5	\$10
---	---	------

**{2,3}** with value **\$80** is optimal.

$C(n) \in \Omega(2^n)$

<u>Subset</u>	<u>Total weight</u>	<u>Total value</u>
{}	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
{2,3}	15	\$80
{2,4}	10	\$40
{3,4}	15	\$60
{1,2,3}	17	-
{1,2,4}	12	\$60
{1,3,4}	17	-
{2,3,4}	20	-
{1,2,3,4}	22	-

Consider the problem of multiplying two (large)  $n$ -digit integers represented by arrays of their digits such as:

## Brute-Force Strategy:

[illegible]

**Write a brute-force algorithm to multiply two arbitrarily large (of  $n$  digits) integers.**

```
      12345678 * 32165487
      86419746
      98765424+
      49382712++
      61728390+++
      74074068++++
      12345678+++++
      24691356++++++
      37037034+++++++
      -----
    397104745215186
```

Basic Operation:  
single-digit multiplication

$C(n) = n^2$  one-digit multiplications  
 $C(n) \in \Theta(n^2)$

May the **Force** be with you!

**</ Brute Force >**