Prepared by: G. Venusopal

Abstracted from:

Unit - II

Introduction to the Design & Analysis of Algorithms, Anany levitin, 2e,

Brute Force - The simplest design strategy.

- It is a straight forward approach to solving a problem usually directly based on the problem statement and definitions of the concepts involved.
- It is easiest to apply.

Ex: exponentiation problem, compute a^n for a given number a and a nonnegative integer n.

important points to be remembered

1. Unlike some of the other strategies, brute force is applicable to a very wide variety of problems.
2. For some important problems (ex., sorting, searching, matrix multiplication, string matching) the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.
3. The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of the problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
4. Even if too inefficient, a brute-force algorithm can still be useful for solving small-size instances of a problem.

what could be the most straightforward method for solving the sorting problem?

sorting: brute-force approach.

Given a list of n orderable items, rearrange them in nondecreasing order.

Selection sort start the selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list.

Then we scan the list, starting with the second element, to find the smallest among the last $n-1$ elements and exchange it with the second element, putting the second smallest element in its final position.

Generally, on the i^{th} pass through the list, which we number from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ elements and swaps it with A_i:

$$A_0 \le A_1 \le \cdots \le A_{i-1} \mid \underbrace{A_i, \ldots A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

in their final positions

After $n-1$ passes, the list is sorted.

Ex: (min=i; i; j)

	89	45	68	90	29	34	17
	17	45	68	90	29	34	89
	17	29	68	90	45	34	89
	17	29	34	90	45	68	89
	17	29	34	45	90	68	89
	17	29	34	45	68	90	89
	17	29	34	45	68	89	90

- unsorted elements list. sort them in nondecreasing order

Each line corresponds to one iteration

no. of comparisons: 21

no. of swaps: 6

17 29 34 45 68 89 90. – sorted list

Algorithm selectionsort(A[0...n-1])
// Input: An array A[0...n-1] of orderable elements.
// output: ASSay A[0...n-1] sorted in ascending order.

for i ← 0 to n-2 do
    min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[min]
            min ← j
    swap A[i] and A[min].

no. of key swaps = n-1 (one for each repetition of the i loop)

Time complexity = $O(n^2)$.

Bubblesort compare 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.

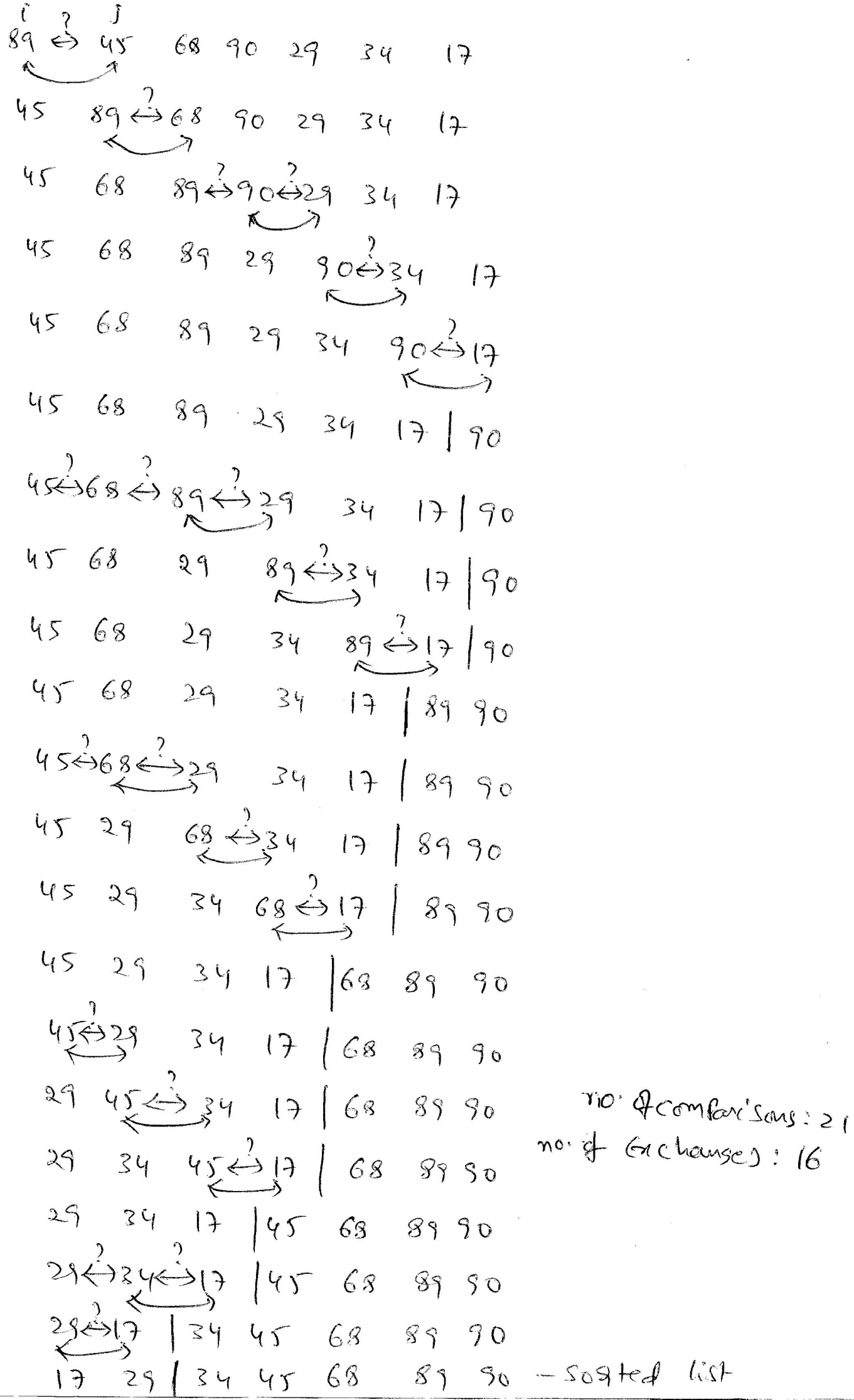
Pass i ($0 \le i \le n-2$) of bubble sort can be represented by the following diagram:

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

in their final positions

Ex: 89 45 68 90 29 34 17 – unsorted elements list.

sort them in nondecreasing order


i ? j
89 ↔ 45 68 90 29 34 17
45 89 ? ↔ 68 90 29 34 17
45 68 89 ? ↔ 90 ? ↔ 29 34 17
45 68 89 29 90 ? ↔ 34 17
45 68 89 29 34 90 ? ↔ 17
45 68 89 29 34 17 | 90
45 ? ↔ 68 ? ↔ 89 ? ↔ 29 34 17 | 90
45 68 29 89 ? ↔ 34 17 | 90
45 68 29 34 89 ? ↔ 17 | 90
45 68 29 34 17 | 89 90
45 ? ↔ 68 ? ↔ 29 34 17 | 89 90
45 29 68 ? ↔ 34 17 | 89 90
45 29 34 68 ? ↔ 17 | 89 90
45 29 34 17 | 68 89 90
45 ? ↔ 29 34 17 | 68 89 90
29 45 ? ↔ 34 17 | 68 89 90
29 34 45 ? ↔ 17 | 68 89 90
29 34 17 | 45 68 89 90
29 ? ↔ 34 ? ↔ 17 | 45 68 89 90
29 ? ↔ 17 | 34 45 68 89 90
17 29 | 34 45 68 89 90 - sorted list
no. of comparisons: 21
no. of exchanges: 16

Algorithm BubbleSort(A[0 ... n-1])
// 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 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j]
            swap A[j] and A[j+1].
```

The number of key swaps, depends on the input.

Time complexity = $O(n^2)$.

improvement for the above algorithm can be done by the following observation:

If a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm.

note: Choosing bubble sort to do sorting is not the right choice.

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).

Extra trick employed in implementing sequential search:

If we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate a check for the list's end on each iteration of the algorithm.

Algorithm sequentialsearch2(A[0...n], k) // enhanced version

// 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] ← K
i ← 0
while A[i] ≠ K do
    i ← i+1
if i < n return i
else return -1.
```

improvement in sequential search can be, if a given list is known to be sorted:

Searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

Advantage or strength: simplicity

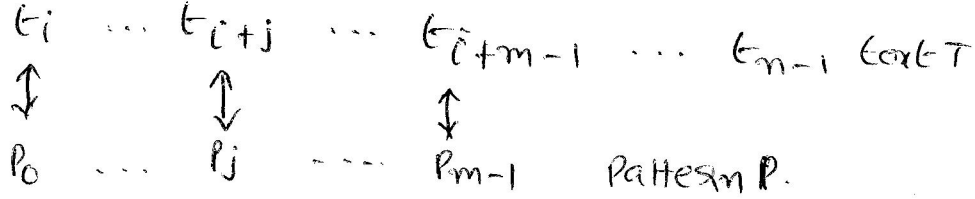
disadvantage or weakness: inferior efficiency.

Time complexity: linear in both worst & average cases

Brute-Force String matching

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.

more clearly, we want to find i — the index of the leftmost character of the first matching substring in the text — such that $t_i = p_0, \cdots t_{i+j} = p_j, \cdots, t_{i+m-1} = p_{m-1}$;



It matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

Brute-force approach:

Align the Pattern against the first m characters of the text and string matching the corresponding pairs of characters from left to right until either all m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

If a mismatching pair is encountered, shift the pattern one position to the right and resume character comparisons, starting again with the first character of the Pattern and its counterpart in the next.

Algorithm BruteForceStringMatch(T[0...n-1], P[0...m-1])
// Input: An array T[0...n-1] of n characters representing a text &
// 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 ← 0 to n-m do
    j ← 0
    while j<m and P[j] = T[i+j] do
        j ← j+1.
    if j = m return i.        // i is the position in 'T' where are found 'P'
return -1
```

Ex: NOBODY - NOTICED - HIM Text
NOT$_x$ Pattern
shift NOT$_x$
shift N$_x$O$_x$T$_x$
shift N$_x$O$_x$T$_x$
shift N$_x$O$_x$T$_x$
shift N$_x$O$_x$T$_x$
shift N O T

Note that, the algorithm shifts the pattern almost always after a single character comparison. In the 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. Thus, in the worst case, the algorithm is in O(nm).

Therefore, the average case efficiency should be considerably better than the worst-case efficiency. i.e., for searching in random texts, it has been shown to be linear O(n+m) = O(n).

Exhaustive search It is a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem's domain, selecting those of them that satisfy all the constraints, and then finding a desired element (ex, the one that optimizes some objective function).

Here we illustrate three important problems: The traveling salesman problem, the knapsack problem, and the assignment problem.

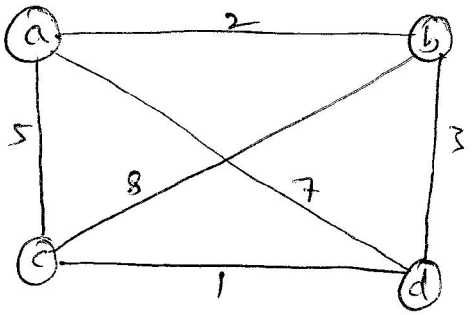
Traveling salesman problem (TSP)

find the shortest path tour through a given set of n cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.).

It is easy to see that a Hamiltonian circuit can be also defined as a sequence of n+1 adjacent vertices $v_{i0}, v_{i1}, v_{i2}, \ldots v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one while all the other n-1 vertices are distinct.

All circuits start and end at one particular vertex. Thus, we can get all the tours by generating all the permutations of n-1 intermediate cities, compute the tour lengths,

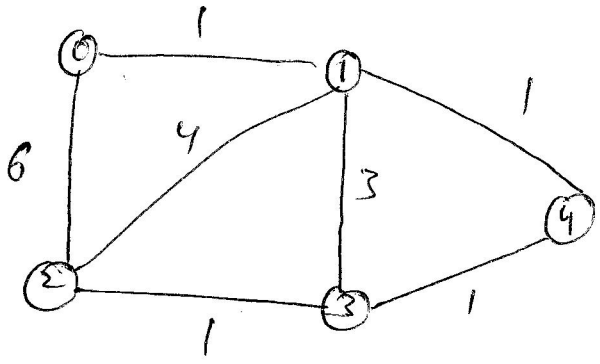
and find the shortest among them.



Tour	Length
$a \to b \to c \to d \to a$	$l = 2 + 8 + 1 + 7 = 18$
$a \to b \to d \to c \to a$	$l = 2 + 3 + 1 + 5 = 11$ optimal
$a \to c \to b \to d \to a$	$l = 5 + 8 + 3 + 7 = 23$
$a \to c \to d \to b \to a$	$l = 5 + 1 + 3 + 2 = 11$ optimal
$a \to d \to b \to c \to a$	$l = 7 + 3 + 8 + 5 = 23$
$a \to d \to c \to b \to a$	$l = 7 + 1 + 8 + 2 = 18$

Solution to a small instance of the travelling salesman problem by exhaustive search.

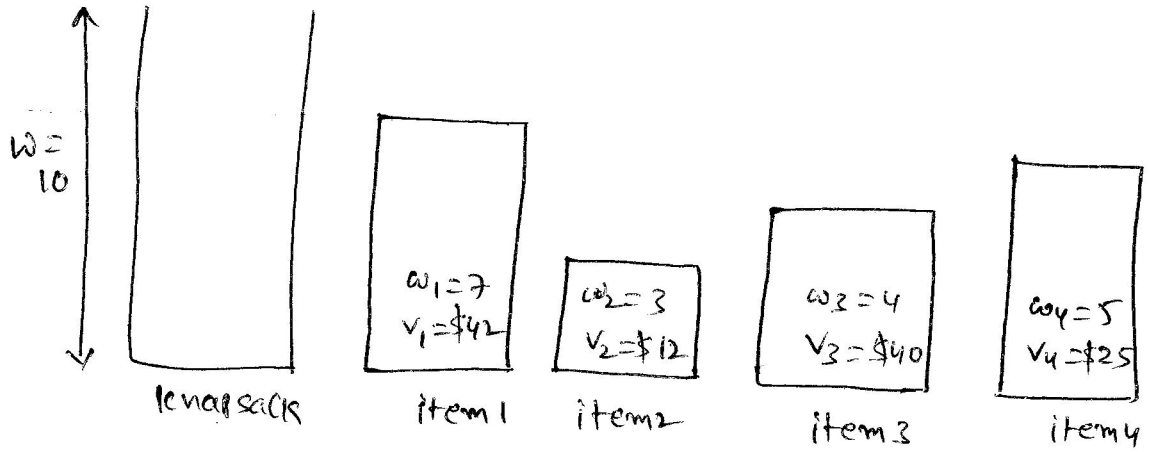
Q2:



Find the optimal cost of the tour for the given graph.

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

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 to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack's 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 $\Omega(2^n)$ algorithm no matter how efficiently individual subsets are generated.



Instance of the knapsack problem.

Subset	Total weight	Total value
ϕ	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1,2}	10	$54
{1,3}	11	not feasible
{1,4}	12	not feasible
{2,3}	7	$52
{2,4}	8	$37
{3,4}	**9**	**$65**
{1,2,3}	14	not feasible
{1,2,4}	15	not feasible
{1,3,4}	16	not feasible
{2,3,4}	12	not feasible
{1,2,3,4}	19	not feasible

Its solution by exhaustive search (optimal selection is in bold)

For both traveling salesman and knapsack problems, exhaustive search algorithms are extremely inefficient on every input. These two are the examples of NP-hard problems and cannot be solved in polynomial time.

Ex:2 Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$

Ex:3 Solve the following knapsack problem instance.

item	weight	value
1	4	$42
2	3	$12
3	1	$40
4	5	$25

capacity $W = 10$

Assignment Problem There are n people who need to be assigned to execute n jobs, one person per job. (i.e, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that could accrue if the ith person is assigned to the jth job is a known quantity C[i, j] for each pair i, j = 1, 2, ..., n. The problem is to find an assignment with the minimum total cost.

A small instance of this problem follows, with the table entries representing the assignment costs C[i, j]:

	Job1	Job2	Job3	Job4
Person1	9	2	7	8
Person2	6	4	3	7
Person3	5	8	1	8
Person4	7	6	9	4

It is easy to see that an instance of the assignment problem is completely specified by its cost matrix C. In terms of this matrix, the problem calls for a selection of one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

We can describe feasible solutions to the assignment problem as n-tuples $(j_1, \ldots, j_n)$ in which the ith component, i = 1, ..., n, indicates the column of the element selected in the ith row (i.e., the job number assigned to the ith person). For ex, for the cost matrix above, <2, 1, 3, 4> indicates a feasible assignment of Person1 to Job2, Person2 to Job1, Person3 to Job3, & Person4 to Job4. 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 could require generating all the permutations of integers 1, 2, ..., 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 no. 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.

C =

	Job1	Job2	Job3	Job4
Person1	9	2	7	8
Person2	6	4	3	7
Person3	5	8	1	8
Person4	7	6	9	4

<1, 2, 3, 4> cost = 9+4+1+4 = 18
<1, 2, 4, 3> cost = 9+4+8+9 = 30
<1, 3, 4, 2> cost = 9+3+8+6 = 26
<1, 3, 2, 4> cost = 9+3+8+4 = 24
<1, 4, 2, 3> cost = 9+7+8+9 = 33
<1, 4, 3, 2> cost = 9+7+1+6 = 23
<2, 1, 3, 4> cost = 2+6+1+4 = 13 optimum
<2, 1, 4, 3> cost = 2+6+8+9 = 25
<2, 3, 1, 4> cost = 2+3+5+4 = 14
<2, 3, 4, 1> cost = 2+3+8+7 = 20
<2, 4, 1, 3> cost = 2+7+5+9 = 23
<2, 4, 3, 1> cost = 2+7+1+7 = 17
<3, 1, 2, 4> cost = 7+6+8+4 = 25
<3, 1, 4, 2> cost = 7+6+8+6 = 27
<3, 2, 1, 4> cost = 7+4+5+4 = 20
<3, 2, 4, 1> cost = 7+4+8+7 = 26
<3, 4, 1, 2> cost = 7+7+5+6 = 25
<3, 4, 2, 1> cost = 7+7+8+7 = 29
<4, 1, 2, 3> cost = 8+6+8+9 = 31
<4, 1, 3, 2> cost = 8+6+1+6 = 21
<4, 2, 1, 3> cost = 8+4+5+9 = 26
<4, 2, 3, 1> cost = 8+4+1+7 = 20
<4, 3, 1, 2> cost = 8+3+5+6 = 22
<4, 3, 2, 1> cost = 8+3+8+7 = 26

feasible assignment
Person1 to Job2, Person2 to Job1, Person3 to Job3, Person4 to Job4

→ Find the optimal solution for the assignment problem given below.

	Job1	Job2	Job3	Job4
Person1	4	3	8	6
Person2	5	7	2	4
Person3	16	9	3	1
Person4	2	5	3	7