Unit - II

Brute Force and Divide-and-Conquer

Brute-Force:-

Brute force is a straightforward approach to solve a problem, usually directly based on the problem statement and definitions of the concepts involved.

Brute force algs can be:

* Optimizing: Finding the best solution. This may require findings all solutions, or if the best solution is known already, it stops when the best solution is found.

Ex: Finding the best path for a travelling salesman.

* Satisfying: Stop as soon as a solution is found that is good enough or till a condition is satisfied.

Ex: Finding a travelling salesman path that is within 10% of optimal solution.

Example:
Computing a^n based on the definition of exponentiation:

$a^n = a * a * a * \ldots * a$ $(a > 0$, n a nonnegative integer)

Adv:-

* Wide applicability
* Simplicity
* Yields reasonable algs for some important problems and standard algs. for simple computational tasks
* It is a good method for developing better algs.

Disadv:-

* Rarely produces efficient algs.
* Some brute-force algs. are extremely slow.
* Not as creative when compared with other design techniques.

Selection Sort :-

The Brute force approach can be used to perform selection sort.

* The problem is to sort a given a list of n orderable items and arrange them in nondecreasing order. (increasing order)

* Selection sort is a sorting alg, specifically an in-place comparison sort.

* We start 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 elements, 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 ith pass through the list, which we number from 0 to n-2, the alg. searches for the smallest item among the last n-i elements and swaps it with A_i:

$$\underbrace{A_0 \le A_1 \le \dots \le A_{i-1}}_{\text{in their final positions}} \mid \underbrace{A_i, \dots A_{min}, \dots A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

After n-1 passes, the list is sorted.

Here the pseudocode of this alg, the list is implemented as an array:

```
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 <- 0 to n-2 do
    min <- i
    for j <- i+1 to n-1 do
```

As an example, the action of the a/g on the list 89, 45, 68, 90, 29, 34, 17 is illustrated below:

Pass 1 :	$\boxed{89}$	45	68	90	29	34	17
" 2 :	17	$\boxed{45}$	68	90	29	34	89
3 :	17	29	$\boxed{68}$	90	45	34	89
4 :	17	29	34	$\boxed{90}$	45	68	89
5 :	17	29	34	45	$\boxed{90}$	68	89
6 :	17	29	34	45	68	$\boxed{90}$	89
7 :	17	29	34	45	68	89	90

Analysis:-

The analysis of selection sort is straightforward.

* The input size is given by the no. of elements n.
* The basic operation is key comparison $A[j] < A[min]$.
* The no. of times it is executed depends only on the array size and is given by the following sum:

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

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

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

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

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

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

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

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

Th. selection sort is a $\Theta(n^2)$ algorithm on all i/ps.

Bubble Sort :-

Another brute-force appln to the sorting pbm is to 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:

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

in their final positions.

```
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 <- 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 action of the alg on the list 89, 45, 68, 90, 29, 34, 17 is illustrated.

The number of key comparisons for the bubble sort version is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

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

$89 \overset{?}{\longleftrightarrow} 45 \quad 68 \quad 90 \quad 29 \quad 34 \quad 17$

$45 \quad 89 \overset{?}{\longleftrightarrow} 68 \quad 90 \quad 29 \quad 34 \quad 17$

$45 \quad 68 \quad 89 \overset{?}{\longleftrightarrow} 90 \overset{?}{\longleftrightarrow} 29 \quad 34 \quad 17$

$45 \quad 68 \quad 89 \quad 29 \quad 90 \overset{?}{\longleftrightarrow} 34 \quad 17$

$45 \quad 68 \quad 89 \quad 29 \quad 34 \quad 90 \overset{?}{\longleftrightarrow} 17$

$45 \quad 68 \quad 89 \quad 29 \quad 34 \quad 17 \quad \boxed{90} \rightarrow$ final position

$45 \quad 68 \quad 89 \quad 29 \quad 34 \quad 17 \quad \boxed{90}$

$45 \overset{?}{\longleftrightarrow} 68 \overset{?}{\longleftrightarrow} 89 \overset{?}{\longleftrightarrow} 29 \quad 34 \quad 17 \quad \boxed{90}$

$45 \quad 68 \quad 29 \quad 89 \overset{?}{\longleftrightarrow} 34 \quad 17 \quad \boxed{90}$

$45 \quad 68 \quad 29 \quad 34 \quad 89 \overset{?}{\longleftrightarrow} 17 \quad \boxed{90}$

$45 \quad 68 \quad 29 \quad 34 \quad 17 \quad 89 \quad 90$

$45 \quad 68 \quad 29$

etc...

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

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

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

The no. of key swaps, however depends on the input. In the worst case of decreasing arrays, it is the same as the no. of key comparisons:

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

Closest-Pair Problem by Brute Force:-

The Closest-pair pbm calls for finding the two closest points in a set of n points.

* It is the simplest of a variety of pbms in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.

* Points can represent physical objects such as airplanes or post offices as well as database records, statistical samples, DNA sequences, and so on.

* An air-traffic controller might be interested in two closest planes as the most probable collision candidates.

* A regional postal service manager might need a solution to the closest-pair pbm to find candidate post-office locations to be closed.

One of the important appln is cluster analysis in statistics. Based on n data points, hierarchical cluster analysis seeks to organize them in a hierarchy of clusters based on some similarity metric.

* For numerical data, this metric is usually the Euclidean distance; for text and other nonnumerical data, the metrics such as the Hamming distance are used

* A bottom-up alg begins with each elements as a separate cluster and merges them into successively larger clusters by combining the closest pair of clusters.

For simplicity, we consider the two-dimensional case of the closest-pair pbm.

* We assume that the points are specified in a standard fashion by their (x,y) Cartesian coordinates and that the distance b/n 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 brute-force approach to solve this pbm leads to the following obvious alg: compute the distance b/n each pair of distinct points and find a pair with the smallest distance.

Here, we consider only the pairs of points (p_i, p_j) for which $i<j$.

ALGORITHM BruteForce ClosestPair(P)
//Finds distance between two closest points in the plane
//Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points

```
d <- ∞
for i <- 1 to n-1 do
    for j <- i+1 to n do
        d <- min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))
return d.
```

The basic operation of the alg is computing square root.

The input size depends on the no. of points or Cartesian co-ordinates n.

The number of times, it will be executed can be computed as follows:

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

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

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

$$= 2[(n-1)+(n-2)+\cdots+1]$$

$$= (n-1)n \in \Theta(n^2).$$

Convex-Hull Problem by Brute Force:-

Finding the convex hull for a given set of points in the plane or a higher dimensional space is one of the most important problems in computational geometry.

* Several applns are based on the fact that convex hulls provide convenient approximations of object shapes and data sets given.
* For example, in computer animation, replacing objects by their convex hulls speeds up collision detection; the same idea is used in path planning for Mars mission rovers.
* Convex hulls are used in computing accessibility maps produced from satellite images by Geographics Information System.
* They are also used for detecting outliers by some statistical techniques.
* An efficient alg for computing a diameter of a set of points, which is the largest distance b/n two of the points, needs the set's convex hull to find the largest distance b/n two of its extreme points.
* Finally, convex hulls are important for solving many optimization pblm.

Definition of convex:-

A set of points (finite or infinite) in the plane is called convex if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.

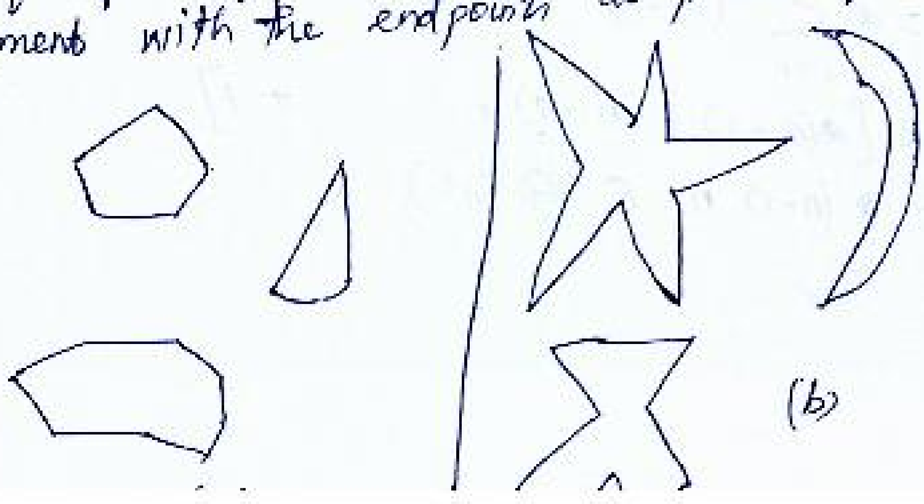


Fig. (a) Convex sets (b) Sets that are not convex.

All the sets depicted in Fig (a) are convex, and so are a straight line, a triangle, a rectangle, and convex polygon, a circle and the entire plane.

On the other hand, the sets depicted in Fig (b), any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.

Definition - Convex hull

The convex hull of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.

If S is convex, its convex hull is obviously S itself.

*If S is a set of two points, its convex hull is the line segment connecting these points.

*If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same ~~pt~~ line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For example, an example of the convex hull for a larger set, in Fig. ②

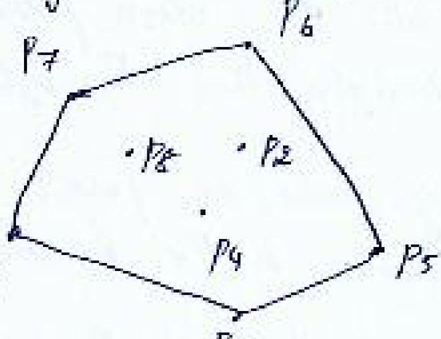


Fig. ②. The convex hull for this set of eight points is the convex polygon with vertices at P_1, P_5, P_6, P_7 and P_3.

Theorem:-

The convex hull of any set S of $n > 2$ points not all on the same line is a convex polygon with the vertices at some of the points of S. (If all the points do lie on the same line, the polygon degenerates to a line segment but

Convex-hull Problem:-

The convex-hull problem is the problem of constructing the convex hull for a given set S of n points.

* To solve it, we need to find the points that will serve as the vertices of the polygon.

* Mathematicians call the vertices of such a polygon "extreme points".

* By definition, an extreme point of a convex set is a point of this set that is not a middle point of any line segment with endpoints in the set.

* For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in Fig. ② are p_1, p_5, p_6, p_7, and p_3.

Identification of extreme points solves the convex-hull problem.

* The knowledge about pairs of points that need to be connected to form the boundary of the convex hull.

* The solution to convex hull problem by Brute Force technique is based on the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points p_i and p_j of a set of n points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points.

* Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

First, the straight line through two points (x_1, y_1), (x_2, y_2) in the coordinate plane can be defined by the equation:

$$ax + by = c,$$

where $a = y_2 - y_1$, $b = x_1 - x_2$

* Second, such a line divides the plane into two half-planes:
  ↳ for all the points in one of them, $ax+by>c$
  ↳ for all the points in another, $ax+by<c$.

* The time efficiency of this alg is $O(n^3)$:
* For each of $n(n-1)/2$ pairs of distinct points, we may need to find the sign of $ax+by-c$ for each of the other $n-2$ points.

Exhaustive Search:-

Exhaustive search is simply a brute-force approach to combinatorial problems.

* It suggests generating each and every element of the pbm domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g. the one that optimizes some objective function).

* It implements a state space search: Given an initial state and a goal state, and a set of operations to attain the goal state, our aim is to find a sequence of operations that transforms the initial state to the goal state.

* The solution process can be represented as a tree.

* The following are some of the important problem that could be solved by exhaustive search:
  (i) Travelling Salesman Problem
  (ii) Knapsack Problem
  (iii) Assignment Problem.

(i) Travelling Salesman Problem:-

The Travelling Salesman problem (TSP) has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applns, and interesting connections to other combinatorial problems.

* In layman's terms, the problem asks to find the shortest

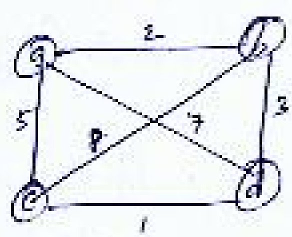
once before returning to the city where it started.

* The pbm 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 pbm can be stated as the pbm of finding the shortest Hamiltonian circuit of the graph.

* Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

Hamiltonian circuit can be defined as a sequence of n+1 adjacent vertices $v_{i_0}, v_{i_1}, \ldots v_{i_{n-1}}, v_{i_0}$ where the first vertex of the sequence is the same as the last one and all the other n-1 vertices are distinct.



Tour	Length	
1) a → b → c → d → a	l = 2+8+1+7 = 18	
2) a → b → d → c → a	l = 2+3+1+5 = 11	// Optimal
3) a → c → b → d → a	l = 5+8+3+7 = 23	
4) a → c → d → b → a	l = 5+1+3+2 = 11	// Optimal
5) a → d → b → c → a	l = 7+3+8+5 = 23	
6) a → d → c → b → a	l = 7+1+8+2 = 18	

Fig. Solution to a small instance of TSP by exhaustive search

* 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.

An inspection of fig. reveals three pairs of tours differs only by their direction.

* Hence, we could cut the no. of vertex permutations by half.

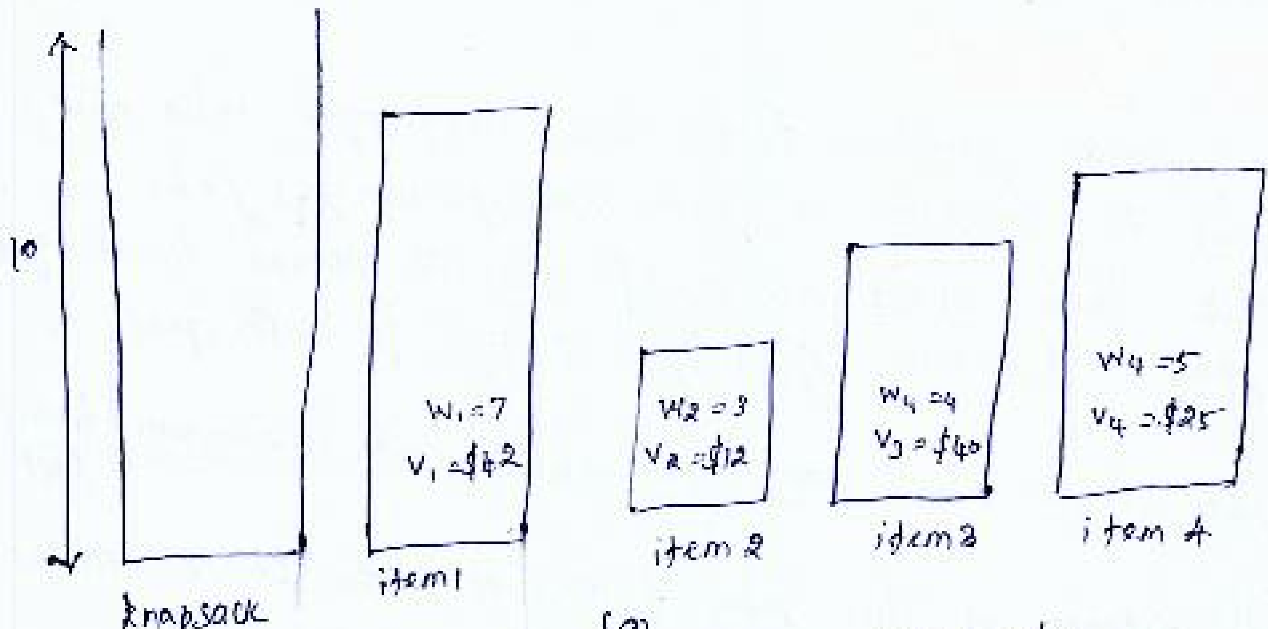
* The total no. of permutations in $\frac{1}{2}(n-1)!$, which makes exhaustive search approach impractical for all but very

(ii) Knapsack Problem:-

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

One such pbm is that a transport plane has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

Fig. represents a small instance of the knapsack pbm.



(a)

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	"
{1,3,4}	16	"

Fig. (a) Instance of the knapsack problem.
(b) Its solution by exhaustive search.

The exhaustive search approach to this pbm 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 (solution) subsets (i.e, the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

* As an example, the solution to the instance of Fig.(a) is given in Fig.(b).

* Since the no. of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ alg.

(iii) Assignment Problem:-

In assignment problem, there are n people who need to be assigned to execute n jobs, one person per job.

* The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i,j]$ for each pair $i, j = 1, 2, \ldots, n$.

* The pbm is to find an assignment with minimum total cost.

A small instance of this pbm 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 pbm is completely specified by its cost matrix C.

* In terms of this matrix, the pbm is to select one element in each row of the matrix so that all selected elements are in

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9+4+1+4 = 18
<1, 2, 4, 3> cost = 9+4+8+9 = 30
<1, 3, 2, 4> cost = 9+3+8+4 = 24
<1, 3, 4, 2> cost = 9+3+8+6 = 26
<1, 4, 2, 3> cost = 9+7+8+9 = 33
<1, 4, 3, 2> cost = 9+7+1+6 = 23
etc.,.

Fig. First few iterations of solving a small instance of the assignment pbm by exhaustive search.

We can describe feasible solutions to the assignment pbm as n-tuples $(j_1, \ldots j_n)$ in which the ith component, $i = 1, \ldots, n$, indicates the column of the element selected in the ith row (i.e, the job number assigned to the ith person).

* For example, for the cost matrix above, (2, 3, 4, 1) indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, Person 4 to Job 1.

* The requirements of the assignment pbm imply that there is a one-to-one correspondence b/n feasible assignments and permutations of the first n integers.

* Therefore, the exhaustive search approach to the assignment pbm would 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 pbm is n!, exhaustive search is impractical for all but very small instances of the problem.

* The Hungarian method is more efficient than the exhaustive search.