2. Brute Force and Exhaustive Searching

2.1 What is a Brute force?

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

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

This suggests simply computing a^n by multiplying 1 by a n times.

2.2 List any five importance of Brute force.

1 First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems.

2 Second, for some important problems (e.g., 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 Third, 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 accept, able speed.

4 Fourth, even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.

5 Finally, a brute-force algorithm can serve an important theoretical or educational purpose, e.g., as a yardstick with which to judge more efficient alternatives for solving a problem.

2. 3. Selection 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 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 ith 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 Ai

$$A_0 \le A_1 \le \cdots \le A_{i-1} \mid \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}$$

in their final positions — the last $n - i$ elements

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

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 do
    min ← i
    for j ← i + 1 to n – 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

| 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

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

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

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

2. 4. Bubble Sort

Another brute-force application to the sorting problem 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<=i <=n - 2) of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \ldots \le A_{n-1}}_{\text{in their final positions}}$$

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 ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

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

the sum for selection sort:

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

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

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

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

2.5, Sequential Search and Brute-Force String Matching.

The first deals with the canonical problem of searching for an item of a given value in a given list. The second is different in that it deals with the string-matching problem.

Sequential Search

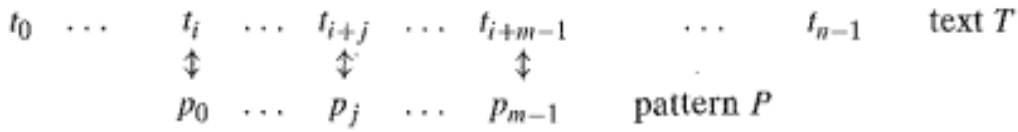
To repeat, the algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: 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. Here is a pseudocode for this enhanced version, with its input implemented as an array.

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

2. 6. Brute-Force String Matching

Recall the string-matching problem introduced in Section 1.3: given a string of n characters called the ***text*** and a string of m characters ($m \le n$) 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 $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:



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 ← 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
return −1
```



```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

FIGURE 3.3 Example of brute-force string matching. (The pattern's characters that are compared with their text counterparts are in bold type.)

Note that for this example, the algorithm shifts the pattern almost always after a single character comparison.

WORST CASE

However, 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**.

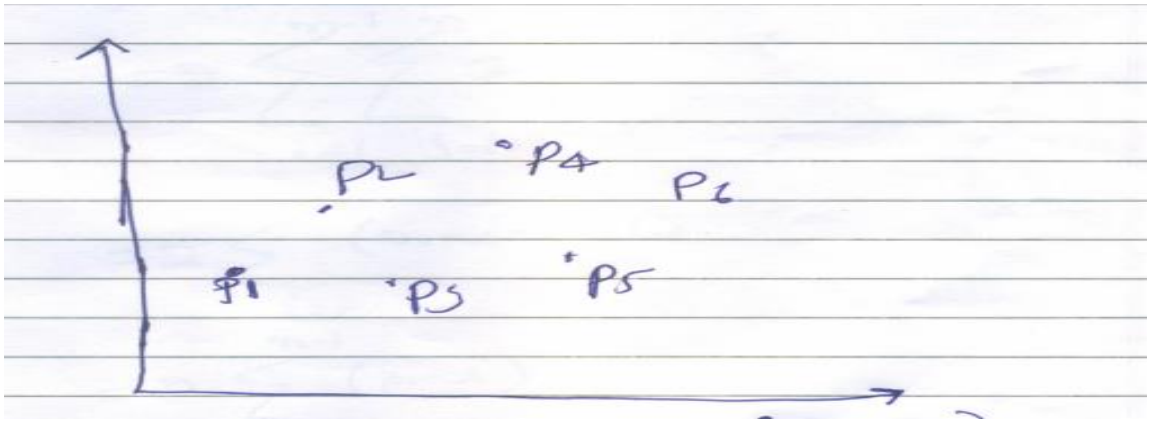
Thus, in the worst case, the algorithm is in O(nm).

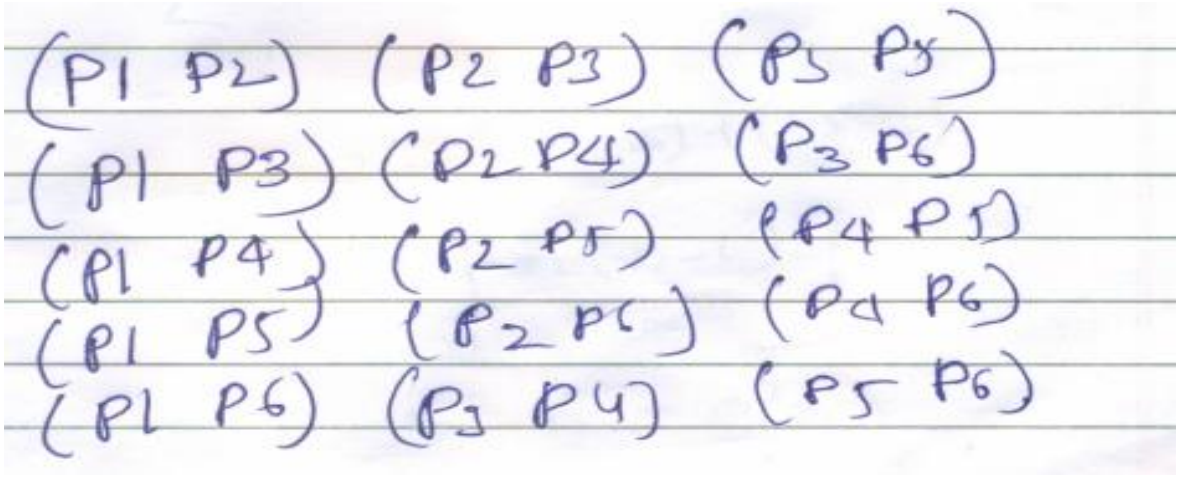
For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again).

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

analysis of algorithms. **A data structure can be defined as a particular scheme of organizing related data items.**

2.7. Closest-Pair and Convex-Hull Problems by Brute Force





2.7. Closest-Pair Problem

The closest-pair problem calls for finding two closest points in a set of n points. For simplicity, we consider the two-dimensional case, although the problem can be posed for points in higher-dimensional spaces as well. We assume that the points in question are specified in a standard fashion by their (x, y) Cartesian coordinates and that 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 brute-force approach to solving this problem leads to the following obvious algorithm: compute the distance between each pair of distinct points and find a pair with the smallest distance. Of course, we do not want to compute the distance between the same pair of points twice. To avoid doing so, we consider only the pairs of points (P_i, P_j) for which $i < j$.

ALGORITHM *BruteForceClosestPoints*(P)

```
//Finds two closest points in the plane by brute force
//Input: A list P of n (n ≥ 2) points P1 = (x1, y1), . . . , Pn = (xn, yn)
//Output: Indices index1 and index2 of the closest pair of points
dmin ← ∞
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        d ← sqrt((xi − xj)^2 + (yi − yj)^2) //sqrt is the square root function
        if d < dmin
            dmin ← d; index1 ← i; index2 ← j
return index1, index2
```

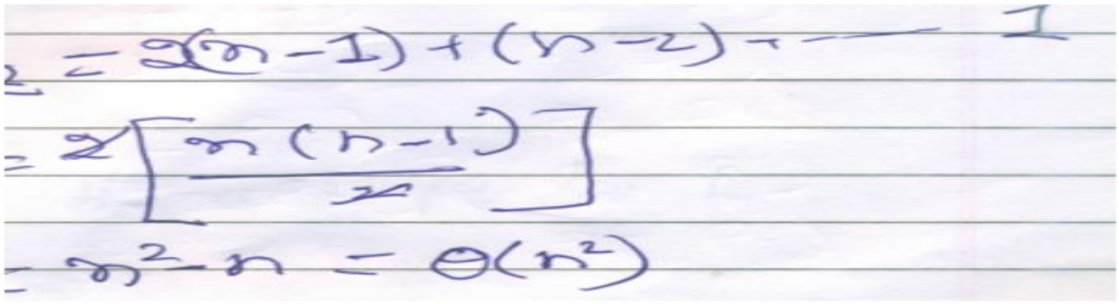
So, if we replace $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ by $dsqr \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$, the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

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

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

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

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



So, if we replace $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ by $dsqr \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$, the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

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

$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2).$$

2.8. Convex-Hull Problem

DEFINITION 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.

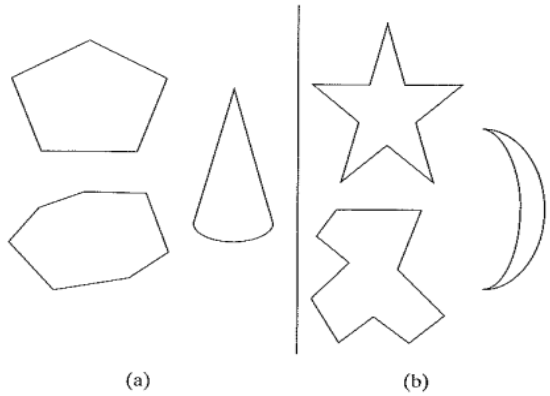


FIGURE 3.4 (a) Convex sets. (b) Sets that are not convex.

DEFINITION 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 Sis 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 line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see Figure 3.6.

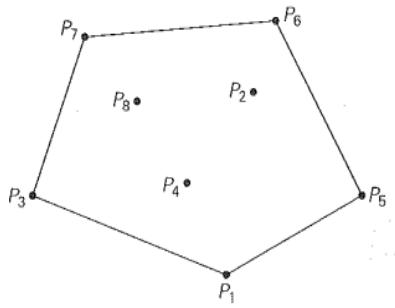
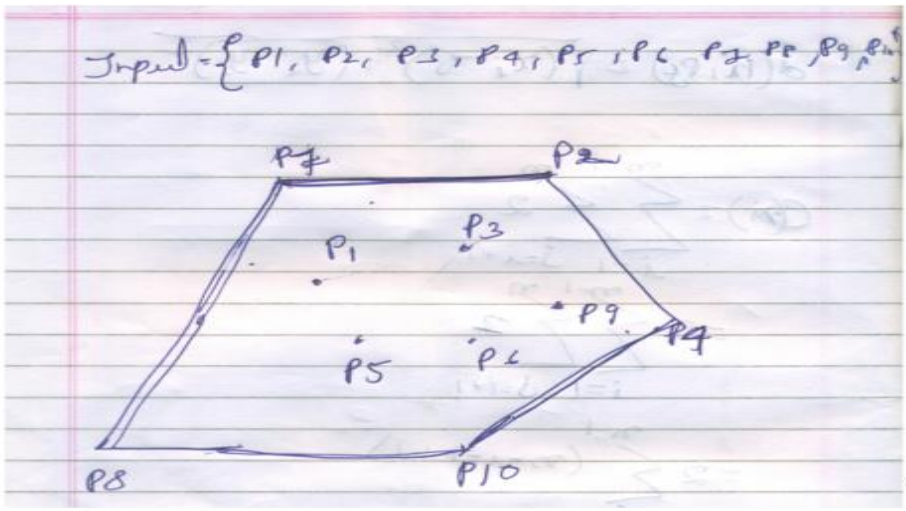
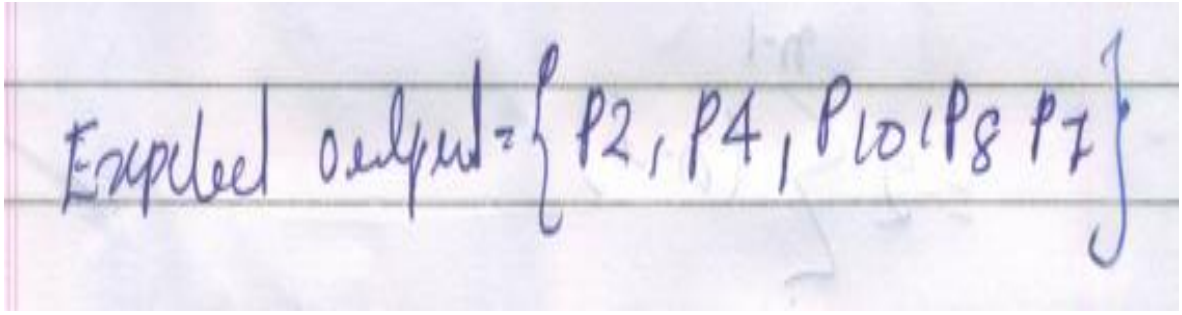


FIGURE 3.6 The convex hull for this set of eight points is the convex polygon with vertices at P_1, P_5, P_6, P_7, and P_3.

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 still with the endpoints at two points of S.)

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 Figure 3.6 are P1,P5, P6, P7, and P3.






```
BruleForce-ConvertHull (P)
{
        E=empty set
        For all ordered pairs(p,q) in PxP and p!=q do
{
        IsEdge=True
        For all points   r  in  P,r!=p and r!=q do
        {
                If r tries to the left of directed line from p to q
                {
                        IsEdge=false
                }
        }
        IF IsEdge the add pq to E
}
From the set of Edge in E, construct a list L values of CH (P) in clock
wise order
Return L
}
```

47

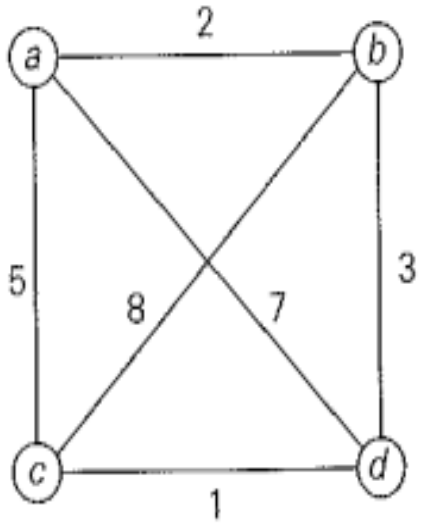
Time Complexity- **$O(n^3)$**

2.9 Exhaustive search

Exhaustive search is simply 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 We illustrate exhaustive search by applying it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem.

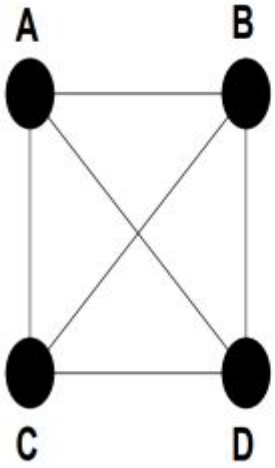
2.9.1 Traveling Salesman Problem

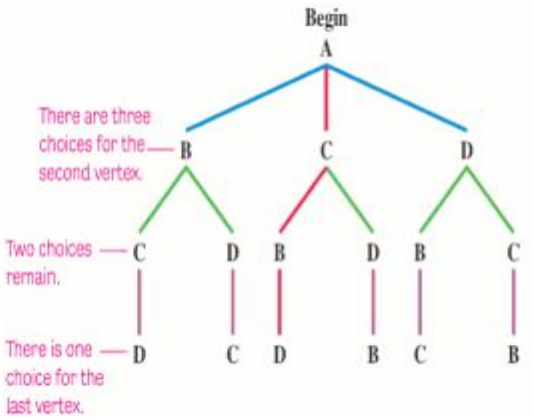
The traveling salesman problem (TSP) has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems. In layman's terms, the problem asks to find the shortest tour through a given set of n 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



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

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search





THE NUMBER OF HAMILTON CIRCUITS IN K_n K_n has $(n-1)(n-2)(n-3)(n-4)\cdots 3\times 2\times 1$ Hamilton circuits. This number is written $(n-1)!$ and is called $(n-1)$ *factorial.*

n	Number of Hamilton Circuits in K_n
3	$2! = 2 \cdot 1 = 2$
4	$3! = 3 \cdot 2 \cdot 1 = 6$
5	$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$
10	$9! = 362{,}880$
15	$14! = 87{,}178{,}291{,}200$
20	$19! = 121{,}645{,}100{,}408{,}832{,}000$

Time Complexity- O(n- 1)!

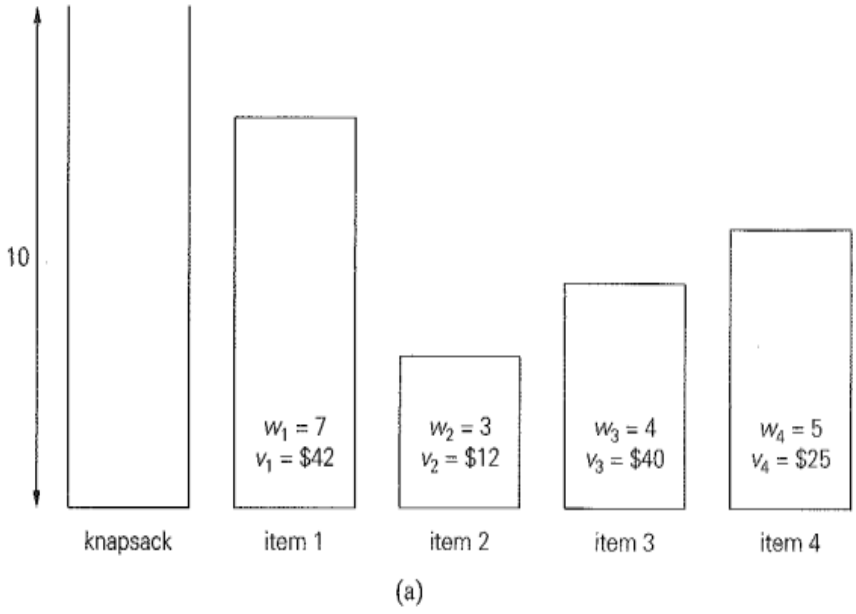
This improvement cannot brighten the efficiency picture much, however. The total number of permutations needed will still be (n- 1)!/2, which makes the exhaustive-search approach impractical for all but very small values of n. On the other hand, if you always see your glass as half-full, you can claim that cutting the work by half is nothing to sneeze at, even if you solve a small instance of the problem, especially by hand.

2.9.2. Knapsack Problem

Here is another well-known problem in algorithmic. Given n items of known weights w1, ... , wn and values v1, ... , vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. If you do not like the idea of putting yourself in the shoes of a thief who wants to steal the most valuable loot that fits into his knapsack, think about a transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity. Figure 3.8a presents a small instance of the knapsack problem.

Consider the Knapsack problem with the following inputs. Solve the problem using exhaustive search. Enumerate all possibilities and indicate unfeasible solutions and Optimal solution. Knapsack total capacity W=10 kg

Items	1	2	3	4
Weight(kg)	7	3	4	5
Value	42	12	40	25



(a)

Subset	Total weight	Total value
Ø	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{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

(b)

Time Complexity=$O(2^n)$

2.9.3 Assignment Problem

In our third example of a problem that can be solved by exhaustive search, **there are n people who need to be assigned to execute n jobs, one person per job.** (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) 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, ... , n. The problem is to find an assignment with the mimimum total cost.

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

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

JOB1	JOB2	JOB3	JOB4	COST-1	COST-2	COST-3	COST-4	TOTAL COST
1	2	3	4	9	4	1	4	18
1	2	4	3	9	4	9	8	30
1	3	2	4	9	8	3	4	24
1	3	4	2	9	8	9	7	33
1	4	2	3	9	6	3	8	26
1	4	3	2	9	6	1	7	23
2	1	3	4	6	2	1	4	13
2	1	4	3	6	2	9	8	25
2	3	1	4	6	8	7	4	25
2	3	4	1	6	8	9	8	31
2	4	1	3	6	6	7	8	27
2	4	3	1	6	6	1	8	21

3	1	2	4	5	2	3	4	14
3	1	4	2	5	2	9	7	23
3	2	1	4	5	4	7	4	20
3	2	4	1	5	4	9	8	26
3	4	1	2	5	6	7	7	25
3	4	2	1	5	6	3	8	22
4	1	2	3	7	2	3	7	19
4	1	3	2	7	2	1	7	17
4	2	1	3	7	4	7	8	26
4	2	3	1	7	4	1	8	20
4	3	1	2	7	8	7	7	29
4	3	2	1	7	8	3	8	26

Time Complexity=O(n!)

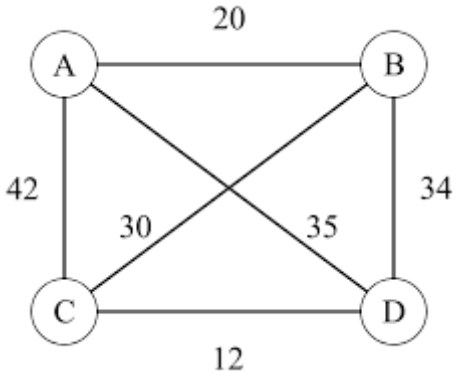
BHANDARKARS' ARTS & SCIENCE COLLEGE, KUNDAPURA
DEPT. OF COMPUTER SCIENCE
V SEMESTER BCA – DAA – UNIT-2

2 marks Questions

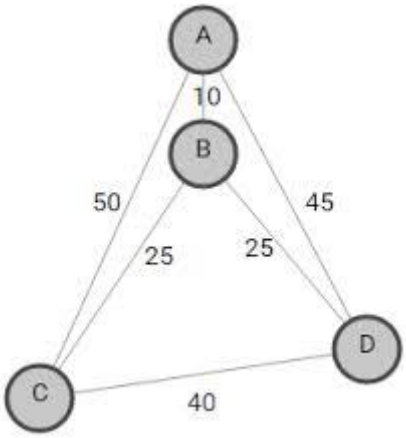
1. What is Brute force approach of problem solving? Give a example.
2. List any four importance of Brute force.
3. Write the number of operations and time complexity of selection sort.
4. Write the number of operations and time complexity of bubble sort.
5. What do you mean by Sequential Search? Write its time complexity
6. Write the worst case and average case complexity of Brute force String Matching.
7. Write the number of operations and time complexity of Closest-Pair Problem.
8. Define Convex and Convex hull.
9. List any two example algorithms of the brute force approach.
10. What do you mean by convex hull problem? Write its time complexity
11. Define Exhaustive search.mechansim.Give example.
12. What do you mean by Knapsack Problem? Write its time complexity
13. What do you mean by Travelling Salesman Problem? Write its time complexity.
14. What do you mean by Job assignment problem? Write its time complexity.

Long Answer

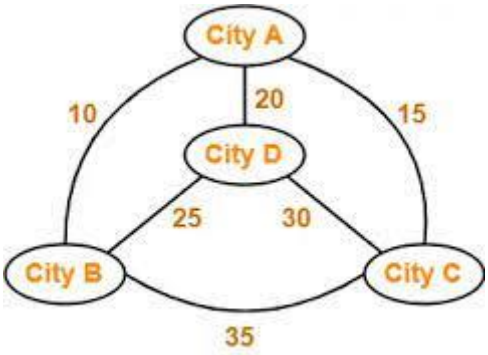
1. Write an algorithm to sort N numbers using Selection sort. Derive the number of operations and time complexity.
2. Write an algorithm to sort N numbers by applying Bubble sort. Derive the number of operations and time complexity.
3. Write and describe the Sequential search algorithm.
4. Write and describe Brute force String Matching Algorithm.
5. Write and explain the algorithm for Closest-Pair Problem. Derive its complexity.
6. Write and explain the algorithm for Convex Hull problem.
7. .Explain Travelling Salesman Problem by exhaustive search with an example.
8. Write a note on Knapsack Problem with an example.
9. Find the optimal solution for the Traveling Salesman problem using exhaustive search method by considering 'A' as the starting city.



10. Find the optimal solution for the Traveling Salesman problem using exhaustive search method by considering 'C' as the starting city.



11. Find the optimal solution for the Traveling Salesman problem using exhaustive search method by considering 'City D' as the starting city.



12. Consider the Knapsack problem with the following inputs. Solve the problem using exhaustive search. Enumerate all possibilities and indicate unfeasible solutions and Optimal solution. Knapsack total capacity W=15kg

Items	A	B	C	D
Weight(kg)	3	5	4	6
Value	36	25	41	34

13. Consider the Knapsack problem with the following inputs. Solve the problem using exhaustive search. Enumerate all possibilities and indicate unfeasible solutions and optimal solution. Knapsack total capacity W=20kg

Items	Item1	Item2	Item3	Item4
Weight	8	10	7	4
Value	40	45	65	30

14. Consider the Job Assignment problem with the following inputs. Solve the problem using exhaustive search.Calcualte Minimal total cost to complete to all jobs one job by each person

	JOB1	JOB2	JOB3	JOB4
Person-1	9	2	7	8
Person-2	6	4	3	7
Person-3	5	8	1	8
Person-4	7	6	9	4

15. Consider the Job Assignment problem with the following inputs. Solve the problem using exhaustive search.Calcualte Minimal total cost to complete to all jobs one job by each person

	JOB1	JOB2	JOB3
Person-1	9	2	7
Person-2	6	4	3
Person-3	5	8	1