BRUTE FORCE

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

Selection Sort, Bubble Sort, Sequential Search, String Matching, Depth-First Search and Breadth-First Search

Advantages:

1. Brute force is applicable to a very wide variety of problems.

2. It is very useful for solving small size in stances of a problem, even though it is inefficient.

3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

Selection Sort

- First scan 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 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 in the sorted list.
- 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 <u>last n −i</u> elements and swaps it with A_i:

$$A_0 \le A_1 \le . .. \le A_{i-1} | A_i, ..., \overset{\downarrow}{A_{min}},. .. ,\overset{\downarrow}{A_{n-1}}$$

In their final positions **|** *the last n –i elements*

- Aftern −1 passes, the list is sorted.

ALGORITHM Selection Sort(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 non decreasing 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

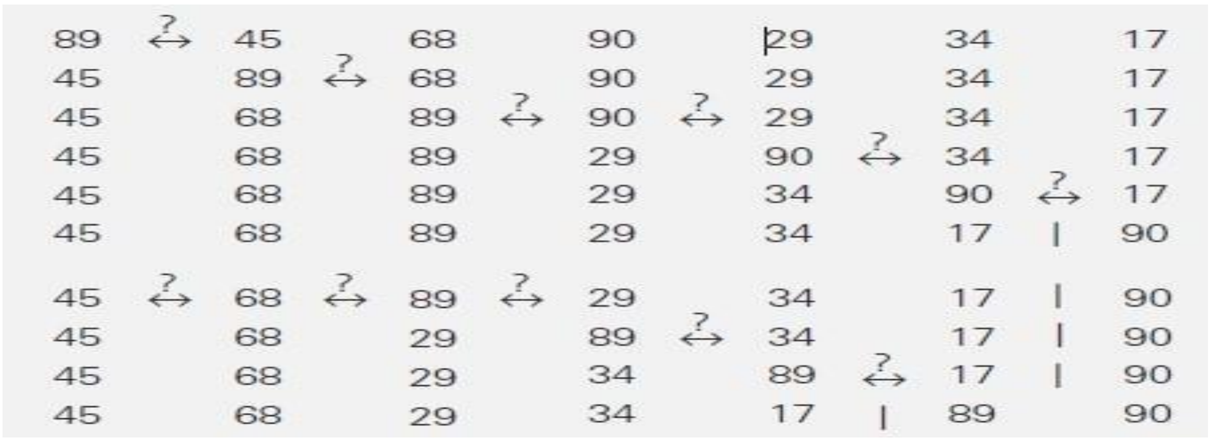
BubbleSort

The bubble sorting algorithm 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 aftern−1passes the list is sorted.Passi(0≤i≤n−2)of bubble sort can be represented by the following:$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} | A_{n-i} \le \ldots \le A_{n-1}$

ALGORITHMBubbleSort(A[0..n−1])

```
//Sortsa given array by bubble sort
//Input:AnarrayA[0..n−1]of orderable elements
//Output:ArrayA[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 algorithm on the list 89,45, 68,90, 29,34,17 is example.



89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17	|	90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17	|	90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17	|	90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17	|	90
45		68		29		34		17	|	89		90

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

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

EXHAUSTIVE SEARCH

For discrete problems in which no efficient solution method is known,it might be necessary to test each possibility sequentially in order to determine if it is the solution. Such ***exhaustive*** examination of all possibilities is known as ***exhaustive search, complete search or* direct *search.***

Exhaustive search is simply a brute force approach to combinatorial problems (Minimization or maximization of optimization problems and constraint satisfaction problems).

Reason to choose brute-force / ***exhaustive search*** approach as an important algorithm design strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact,it seems to be the only **general approach** for which it is more difficult to point out problems it *cannot* tackle.
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 **acceptable 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.

Exhaustive Search is applied to the important problems like

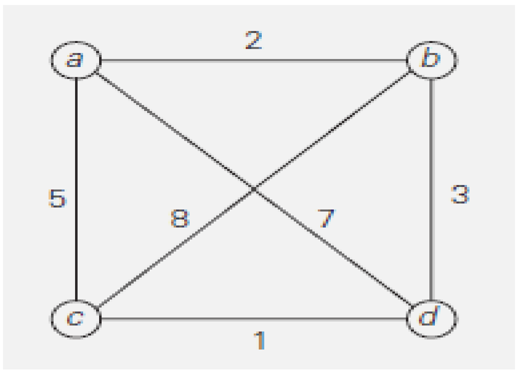
- Traveling Salesman Problem
- Knapsack Problem

TRAVELING SALES MAN PROBLEM

The ***traveling salesman problem (TSP)*** is one of the combinatorial problems. 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. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once).

AHamiltoniancircuitcanalsobedefinedasasequenceof$n$+1adjacentvertices vi_0, vi_1, . . . , vi_{n-1}, vi_0, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



Tour	Length
a--->b--->c --->d--->a	I=2 +8+1 +7 =18
a--->b--->d--->c--->a	**I= 2 + 3 + 1 + 5 = 11 optimal**
a--->c--->b--->d--->a	I=5 +8+3 +7 =23
a--->c--->d--->b--->a	**I= 5 + 1 + 3 + 2 = 11 optimal**
a--->d--->b--->c--->a	I=7 +3+8 +5 =23
a--->d--->c --->b--->a	I=7 +1+8 +2 =18

FIGURE2.4 Solution to a small instance of the traveling salesman problem by exhaustive search.

Time efficiency

- We can get all the to by generating all the permutations of n−1intermediate cities From a particular city..i.e.**(n-1)!**
- Consider two intermediate vertices,say,*b*and *c*,and then only permutations in which *b* Precedes *c*.(This trick implicitly defines tour's direction.)
- An inspection of Figure 2.4reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same.
- The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of *n*.

KNAPSACKPROBLEM

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.

Real time examples:

- A Thifwho wants to steal the most valuable loot that fits into his knapsack,
- A transport plane that has to deliver the most valuable set of items to are more location Without exceeding the plane's capacity.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given,computing the total weigh to each subset inorder to identify feasible subsets(i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

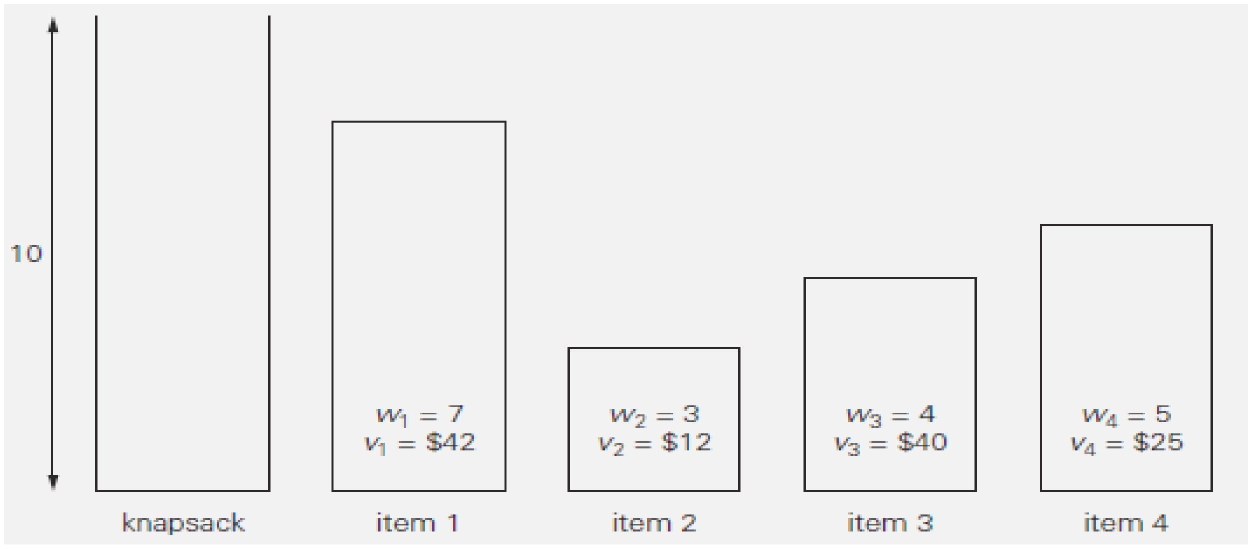


FIGURE2.5 Instance of the knapsack problem.

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(Maximum-Optimum)**
{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

FIGURE2. 6 knapsack problem's solution by exhaustive search. The information about the optimal
Selection is in bold.

Time efficiency: As given in the example, the solution to the instance of Figure 2.5 is given in Figure 2.6. Since the *number of subsets of an n-element set is* $\boldsymbol{2^n}$, the exhaustive search leads to a $\boldsymbol{\Omega(2^n)}$ algorithm, no matter how efficiently individual subsets are generated.

Note: Exhaustive search of both the traveling salesman and knapsack problems leads to extremely inefficient algorithms on every input. In fact, these two problems are the best-known examples of ***NP-hard problems***. **No polynomial-time** algorithm is known for any *NP*-hard problem. Moreover, most computer scientists believe that such algorithms do not exist. some sophisticated approaches like **backtracking** and **branch-and-bound** enable us to solve some instances but not all instances of these in less than exponential time. Alternatively, we can use one of many **approximation algorithms.**

Linear Search Algorithm(Sequential Search)

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the processof locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

- Linear search algorithm finds given element in a list of elements with O(n) timecomplexity where n is total number of elements in the list.
- This search process starts comparing of search element with the first element in the list.
- If both are matching then results with element found otherwise search element iscompared with next element in the list.
- If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list.
- if that last element also doesn't match, then the result is "Element not found in the list".That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

Step 1: Read the search element from the user

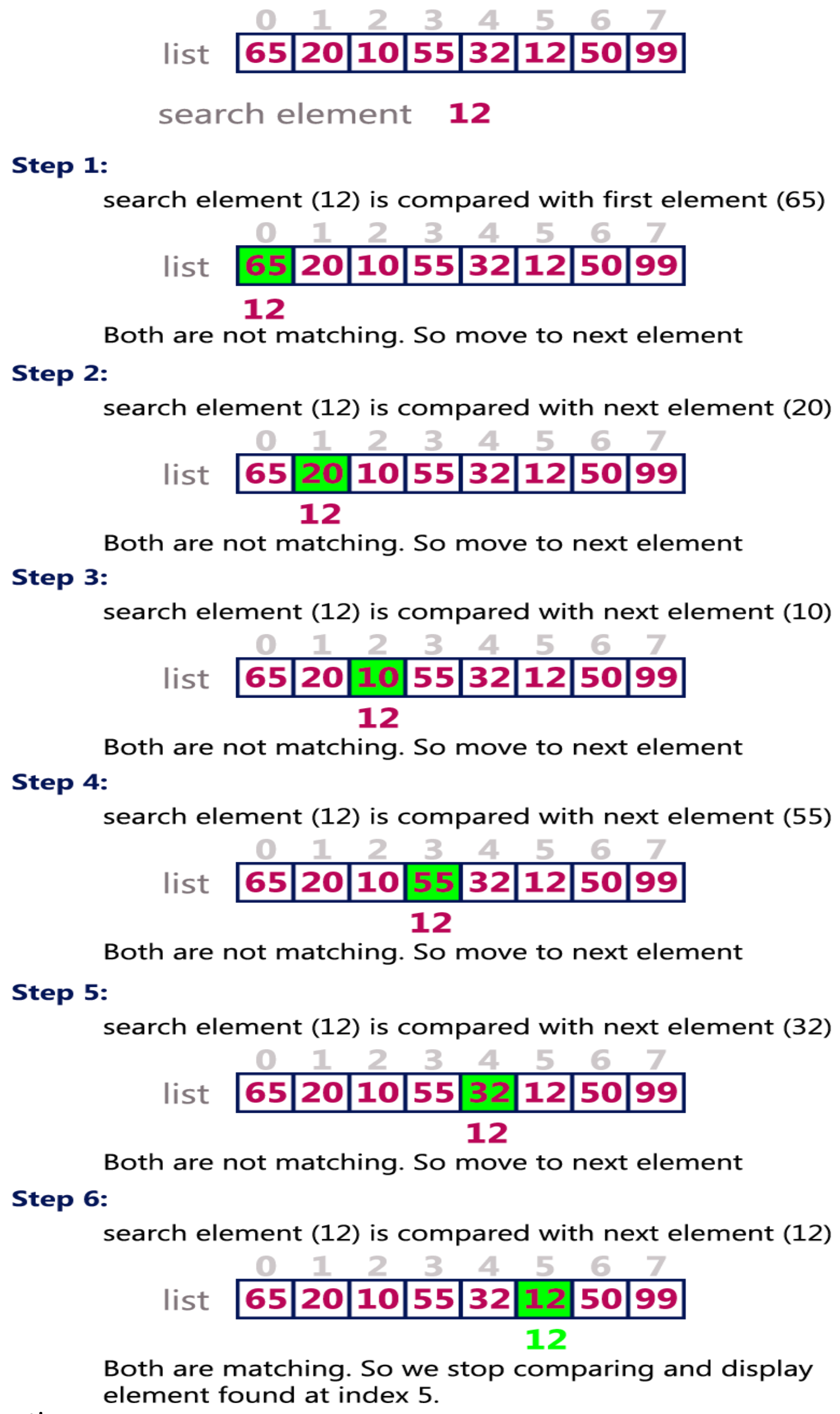
Step 2: Compare, the search element with the first element in the list.

Step 3: If both are matching, then display "Given element found!!!" and terminate the function

Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!"and



0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
search element 12
Step 1:
search element (12) is compared with first element (65)
12
Both are not matching. So move to next element
Step 2:
search element (12) is compared with next element (20)
Both are not matching. So move to next element
Step 3:
search element (12) is compared with next element (10)
Both are not matching. So move to next element
Step 4:
search element (12) is compared with next element (55)
Both are not matching. So move to next element
Step 5:
search element (12) is compared with next element (32)
Both are not matching. So move to next element
Step 6:
search element (12) is compared with next element (12)
Both are matching. So we stop comparing and display element found at index 5.

terminate the function.

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int target) {
    int i;
    for (i = 0; i< n; i++) {
        if (arr[i] == target) {
        return i; // Element found at index i8.
        }
    }
    return -1;  // Element not found
}

int main() {
    int arr[] = {10, 2, 8, 5, 17};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 8;
    int result = linearSearch(arr, n, target);
    if (result == -1) {
printf("Element not found in the array.\n");
    } else {
printf("Element found at index: %d\n", result);
    }
    return 0;
}
```

Assuming that the target element is ***8***, let's use the example array ***[10, 2, 8, 5, 17].*** Running the code yields the following result:

```
Element found at index: 2
```

The element ***'8'*** was found in this instance by the linear search technique at ***index 2*** of the array.

Let's dissect the implementation process in detail:

1. To use the ***printf function***, we include the ***h library***.
2. The array ***arr[]***, the array's ***size n***, and the element to be searched are the three arguments provided to the ***linear Search function***.
3. We create the integer ***variable i*** and use a ***for loop*** to iterate through the array.
4. We compare each element of the array with the target element inside the loop.
5. If a match is discovered, we return the element's location's index.

6. We return ***-1*** to denote that the element was not found if, after searching the full array, no matches were discovered.
7. We declare an array called ***arr[]*** with a few random members in the main function.
8. By dividing the array's overall size by the size of a single member, we may get the array's n-dimensional size.
9. The target element we wish to look for is specified (in this example, 8).
10. The array, size, and target are passed as arguments when we use the ***linear Search function***, and we store the result in the result variable.
11. Finally, we evaluate the value of the outcome. We print a message explaining that the element was not found if it is ***-1.*** If not, we display the index at which the element was located.