BRUTE FORCE AND EXHAUSITIVE SEARCH

A brute force algorithm solves a problem through exhaustion: it goes through all possible choices until a solution is found. The time complexity of a brute force algorithm is often proportional to the input size. Brute force algorithms are simple and consistent, but very slow.

Brute force is a non-uniform search that uses a trial and error to guess login info, encryption keys, or find a hidden web page. Exhaustive search is a type of brute force search used to solve problems related to permutation and combination. It is a uniform search and takes less time.

SELECTION SORT

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

- Selection sort is an effective and efficient sort algorithm based on comparison operations.
- It adds one element in each iteration.
- You need to select the smallest element in the array and move it to the beginning of the array by swapping with the front element.
- You can also accomplish this by selecting the most potent element and positioning it at the back end.
- In each iteration, selection sort selects an element and places it in the appropriate position

ALGORITHM SelectionSort(A[0….n-1])

//Sorts a given array by selection sort

//input: an array A[0…n-1] of ordered elements

//output: array A[0…n-1] sorted in increasing order

for ← 0 to n-2 do

min ← i

for j←i+1 to n-1 do

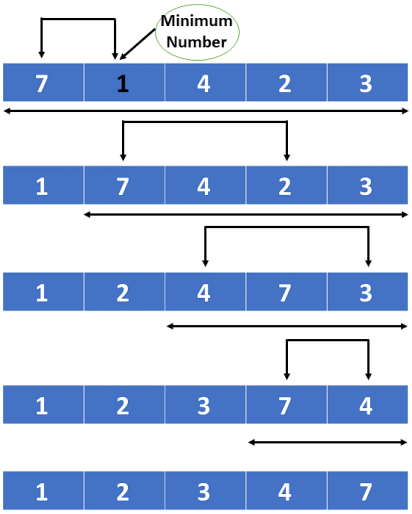
if A[j]<A[min]

min ←j

swap A[j] and A[min]

Analysis of Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted



The analysis of selection sort is straight forward.

Time complexity

The input size is given by the number of elements n

The basic operation is the key comparison A[j]<A[min].

The number of times it is executed depends only on the array size and given by the following sum:

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

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

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

i=0

Substitute i=0,1,2,3…….n

So, n-1-i

n-1-0, n-1-1, n-1-2, n-1-3…….

n-1, n-2, n-3…..n-1-n+2

(n-1)+(n-2)+(n-3)+…..1

=n(n-1)/2 // ½ is considered as constant

=n2-n // as n2 is large so consider only that

=n2

Therefore, the selection sort algorithm encompasses a time complexity of $O(n^2)$ and a space complexity of O(1) because it necessitates some extra memory space for temp variable for swapping.

- Best Case Complexity: The selection sort algorithm has a best-case time complexity of $O(n^2)$ for the already sorted array.
- Average Case Complexity: The average-case time complexity for the selection sort algorithm is $O(n^2)$, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- Worst Case Complexity: The worst-case time complexity is also $O(n^2)$, which occurs when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is $O(n^2)$ in all three cases. This is because, in each step, we are required to find minimum elements so that it can be placed in the correct position. Once we trace the complete array, we will get our minimum element.

BUBBLE SORT

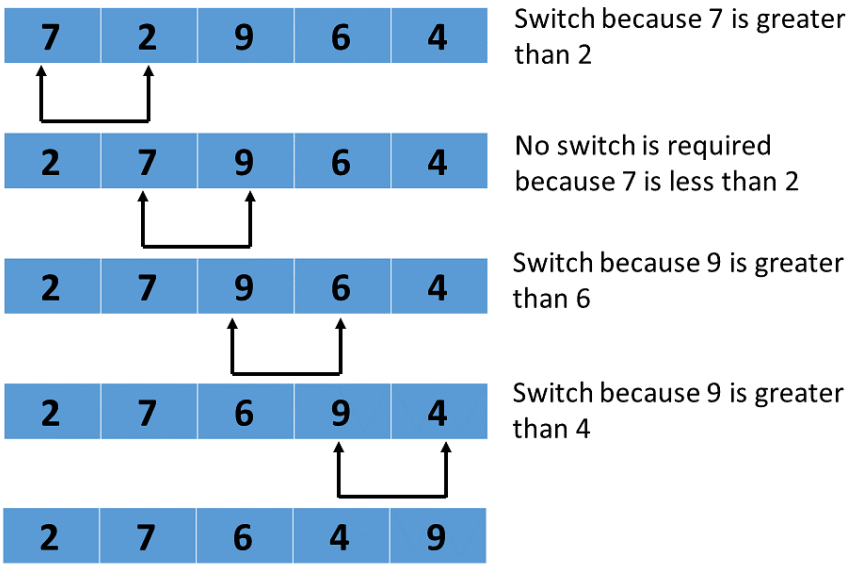
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

Example:

First Pass:

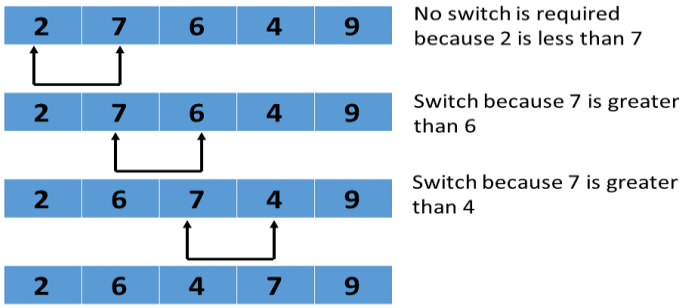
- Compare the first and second elements, starting with the first index.
- They are swapped if the first element is greater than the second.

- Compare the second and third elements now. If they are not in the correct order, swap them.
- The preceding procedure is repeated until it reaches the final element.



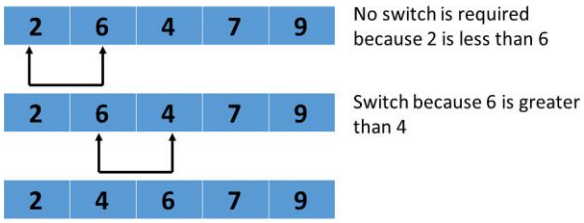
Second Pass

- The process is repeated for the remaining iterations.
- The most significant element among the unsorted elements is placed at the end of each iteration



Third Pass

The comparison is performed up to the last unsorted element in each iteration.



Fourth Pass

When all of the unsorted elements are placed in their correct positions, the array is sorted.

Algorithm: bubblesort(A[0…n-1])

//sorts a given array by bubble sort

//input: an array A[0…n-1]) of ordered elemets

//output:array A[0…n-1] sorted in ascending order

for i←0 to n-2 do // number of iterations

    for j←0 to n-2-i do // bubble down the largest element

        if A[j+1]<A[j] // if its true then go to next step

            Swap A[j] and A[j+1]

Time complexity

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

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

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

Substitute i=0,1,2,3…….n

So, n-1-i

n-1-0, n-1-1, n-1-2, n-1-3…….

n-1, n-2, n-3…..n-1-n+2

(n-1)+(n-2)+(n-3)+…..1

=n(n-1)/2 // ½ is considered as constant

=n2-n // as n2 is large so consider only that

=n2

Therefore, the bubble sort algorithm encompasses a time complexity of O(n2) and a space complexity of O(1) because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- Best Case Complexity: The bubble sort algorithm has a best-case time complexity of O(n) for the already sorted array.
- Average Case Complexity: The average-case time complexity for the bubble sort algorithm is $O(n^2)$, which happens when 2 or more elements are in jumbled, i.e., neither in the ascending order nor in the descending order.
- Worst Case Complexity: The worst-case time complexity is also $O(n^2)$, which occurs when we sort the descending order of an array into the ascending order.

SEQUENTIAL OR LINEAR SEARCH

Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

First,

- we have to traverse the array elements using a for loop.
- In each iteration of for loop, compare the search element with the current array element, and -
- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return -1.

Algorithm: sequentialSearch(A[0…n-1],k)

//implements sequential search with a search key

// input: a array 'A' to 'n' elements and search key 'K'

//output: the index of the first element in A[0…n-1] whose value is equal to K or n-1 if no element is found.

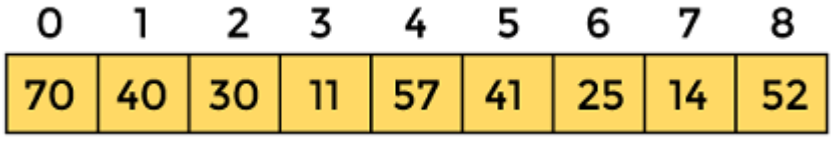
```
A[n]←K
I←0
        While A[i]!=K do      // if element not found then go to next step
        I←i+1
If  i<n returns I             // so element found successfully
Else
Return -1                     // value not found
```

Working of Linear search

Now, let's see the working of the linear search Algorithm.

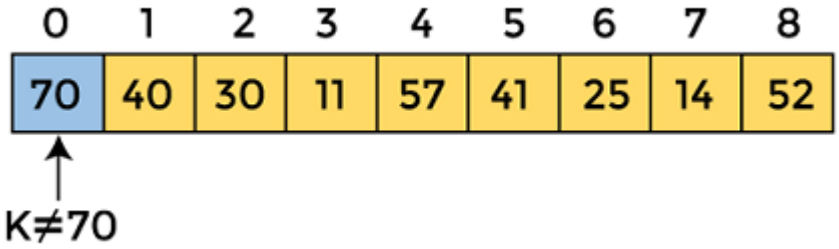
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

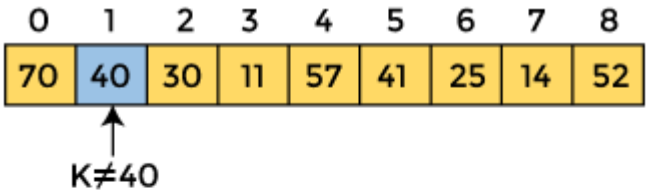


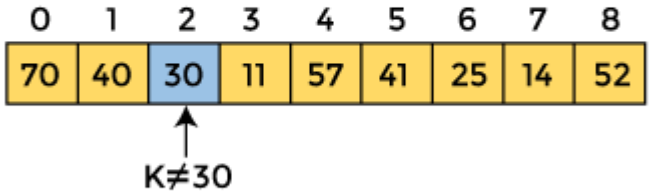
Let the element to be searched is **K = 41**

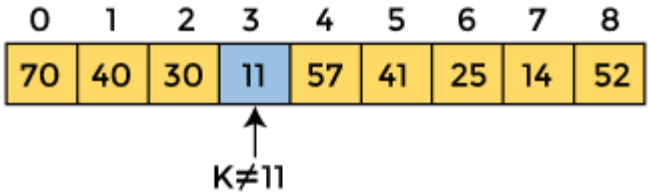
Now, start from the first element and compare **K** with each element of the array.

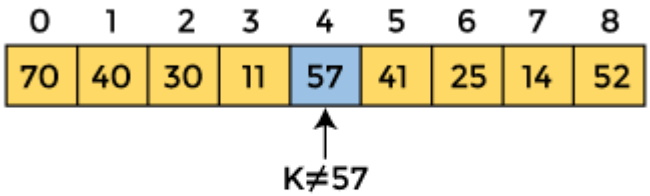


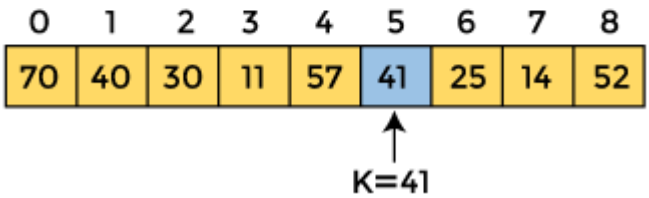
The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.











Now, the element to be searched is found. So algorithm will return the index of the element matched.

Linear Search complexity

Now, let's see the time complexity of linear search in the best case, average case, and worst case. We will also see the space complexity of linear search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

- **Best Case Complexity -** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
- **Average Case Complexity -** The average case time complexity of linear search is **O(n).**
- **Worst Case Complexity -** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**

BRUTE FORCE STRING MATCHING

The simplest algorithm for string matching is a brute force algorithm, where we simply try to match the first character of the pattern with the first character of the text, and if we succeed, try to match the second character, and so on; if we hit a failure point, slide the pattern over one character and try again.

The string matching problem is to find if a pattern P[1...m] occurs within the text T[1….n].

1. It is also known as substring search.

2. Given a text T and a pattern P.

● Is the pattern a substring of the text

● Is there a position iwhere the entire pattern occurs in the given text.

3. In every position of the given text T, do the next m elements of the array, match the M elements of the pattern.

NAIVE-STRING-MATCHER (T, P)

1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s

Time Complexity: O(N2)
Auxiliary Space: O(1)

Complexity Analysis of Naive algorithm for Pattern Searching:

Best Case: O(n)

- When the **pattern** is found at the very beginning of the **text** (or very early on).
- The algorithm will perform a constant number of comparisons, typically on the order of **O(n)** comparisons, where n is the length of the **pattern**.

Worst Case: O(n2)

- When the **pattern** doesn't appear in the **text** at all or appears only at the very end.
- The algorithm will perform **O((n-m+1)*m)** comparisons, where **n** is the length of the **text** and **m** is the length of the **pattern**.
- In the worst case, for each position in the **text**, the algorithm may need to compare the entire **pattern** against the text.

Example:

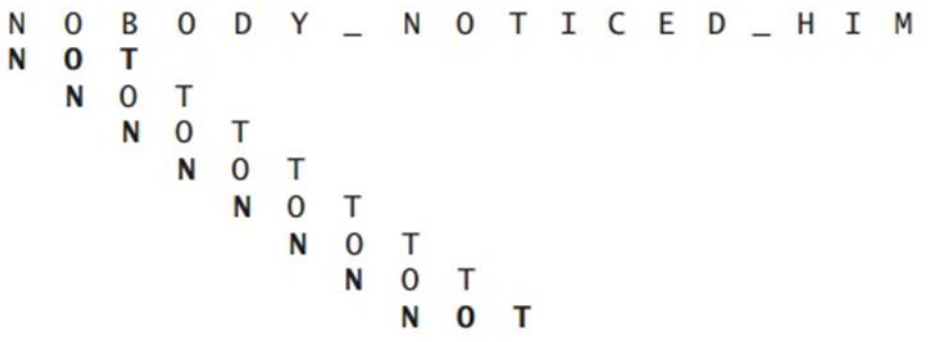
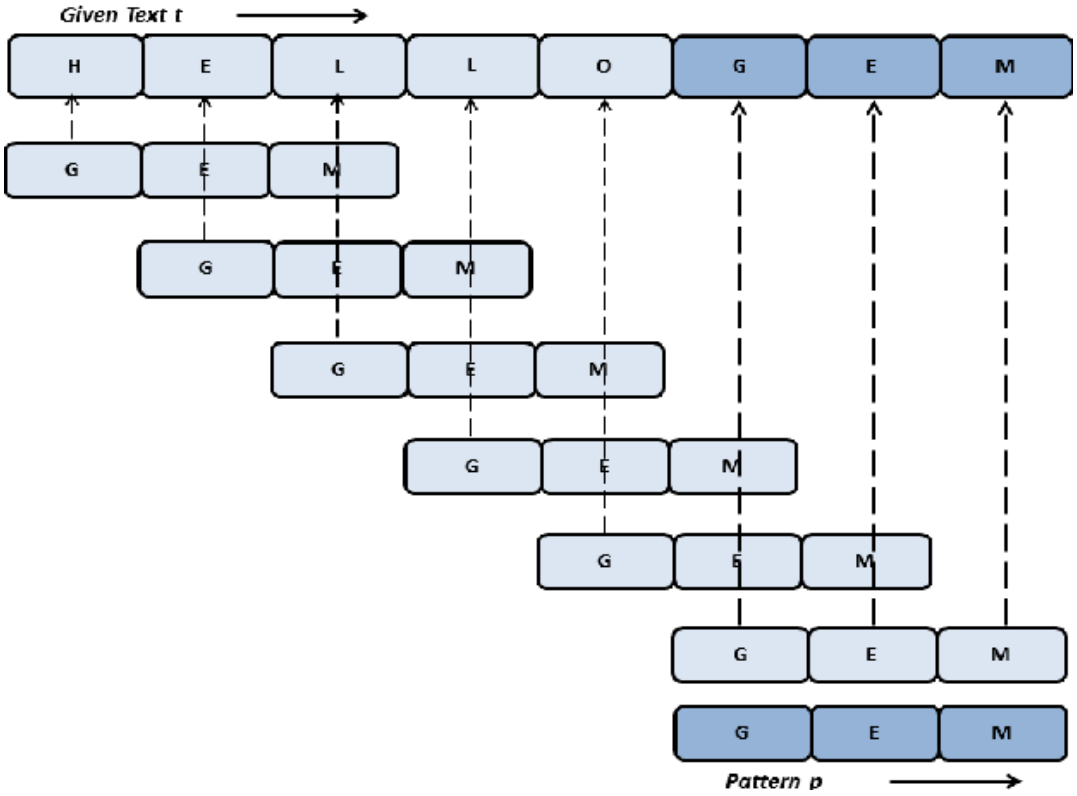


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

Example 2



Travelling salesman problem

In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j.

The time complexity of an algorithm depends on the number of nodes. If the number of nodes is 'n' then the time complexity will be proportional to 'n!' is O(n!).

Examples: Solved in notes

Knapsack problem

A Knapsack problem models a situation similar to the filling of a Knapsack that cannot support more than a certain weight with all or part of a given set of objects each having a weight and a value.

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Time complexity

The number of subsets of an 'n' elements is 2 power n. so the exhaustive search leads to O(2 power n).

Examples: solved in notes

DEPTH FIRST SEARCH AND BREADTH FIRST SEARCH PROBLEMS ARE SOLVED IN THE NOTES- REFER IT