JAIN DEEMED-TO-BE UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND IT

DEPARTMENT OF BCA

ANALYSIS AND DESIGN OF ALGORITHMS (16BCA5D11)

Module 2 : Brute Force Approaches

Prepared by: Prof. Bhavana Gowda D M

UNIT 2: BRUTE FORCE APPROACHES

Introduction

What is Brute Force Method?

The straightforward method of solving a given problem based on the problems statement and definition is called Brute Force method.

Ex : Selection Sort, Bubble Sort, Linear Search etc.

Pros and Cons of Brute Force Method :

Pros:

- The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.
- It is a generic method and not limited to any specific domain of problems.
- The brute force method is ideal for solving small and simpler problems.
- It is known for its simplicity and can serve as a comparison benchmark.

Cons:

- The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the O(N!) order of growth.
- This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.
- Brute force algorithms are slow.
- Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.

Exhaustive Search :

A Search problem involves some set of possibilities and we are looking for one or more of the possibilities and we are looking for one or more of the possibilities that satisfy some property.

Example : Travelling Salesperson Problem

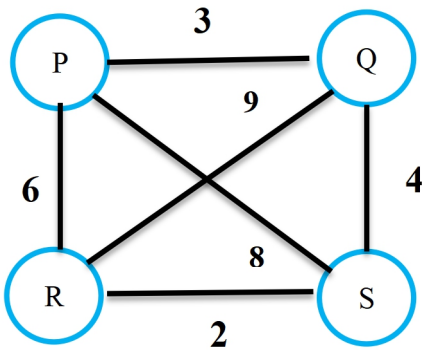
Method :

1. Generate a list of all potential solutions to the problem.
2. Evaluate potential solutions one by one disqualifying infeasible ones and for an optimization problem,keeping track of the best one found so far.

Travelling Salesman Problem :

- ✓ The Travelling Salesman Problem (TSP) is an optimization problem used to find the shortest path to travel through the given number of cities.
- ✓ Travelling salesman problem states that given a number of cities N and the distance between the cities, the traveler has to travel through all the given cities exactly once and return to the same city from where he started and also the length of the path is minimized.
- ✓ The Travelling Salesman Problem (TSP) can be formulated as follows: to choose a pathway optimal by the given criterion. In this, optimal criterion is usually the minimal distance between towns or minimal travel expenses. Travelling salesman should visit a certain number of towns and return to the place of departure, so that they visit each town only once.
- ✓ TSP can be solved as shown below:
    1. Get all the routes from one city to another city by taking various permutations.
    2. Compute the route length for each permutation and select the shortest among them.
- ✓ TSP can be modeled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once. Often, the model is a complete graph (i.e. each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

Example :



Here P Q R S are Cities.

- ✓ Distance between various cities are represented as numbers in each of the edge.
- ✓ Let assume, Salesperson starts from City P then the various routes using which he can visit each and every city exactly once and returns back to the start city P along with the cost is as shown below:

P ⟶ Q ⟶ R ⟶ S ⟶ P (COST = 22)
P ⟶ Q ⟶ S ⟶ R ⟶ P (COST = 15)
P ⟶ R ⟶ Q ⟶ S ⟶ P (COST = 27)
P ⟶ R ⟶ S ⟶ Q ⟶ P (COST = 15)
P ⟶ S ⟶ Q ⟶ R ⟶ P (COST = 27)
P ⟶ S ⟶ R ⟶ Q ⟶ P (COST = 22)

Finally considering routes with minimum cost to get the maximum profit, we will consider the below routes and cost:

P ⟶ Q ⟶ S ⟶ R ⟶ P (COST = 15)
P ⟶ R ⟶ S ⟶ Q ⟶ P (COST = 15)

Note :

In general, for n cities number of routes=(n-1)! I.e., f(n)=f(n-1)!

Hence the time complexity is given by ,

F(n)=O(n!)

Selection Sort and Bubble Sort

Selection Sort

- This is a sorting algorithm, which is very simple to understand and implement.
- The algorithm achieves its name from the fact that with each iteration the appropriate value for a key position is selected from the list of remaining elements and put in the required position of the array.
- However the algorithm is not efficient for large arrays.
- The method of selection sort relies heavily on a comparison mechanism to achieve its goals.

Procedure :

1. As the name indicates, we first find the smallest eement in the list and we exchange it with the first item.

2. Obtain the second smallest element in the list and exchange it with the second element and so on.

3. Finally all the items will be arranged in ascending order.

4. This technique is called Selection Sort.

```
Algorithm : selectionsort(a[ ], n)
//purpose : sort the given elements in ascending orderusing
selection sort.
//inputs : n - number of elements
           a - the items to be sorted are present in the array.
//outputs : a - contains the sorted list

for  i=0 to n-2 do
        pos = i;
        for j=i+1 to n-1 do
                if (a[j]<a[pos])
                pos <-- j;
        end for
        temp=a[pos]

        a[pos]=a[i]

        a[i]=temp

end for
```

Analysis of Selection Sort : $\theta (n^2)$

BUBBLE SORT (Sinking Sort)

- The algorithm of bubble sort functions as follows.
- The algorithm begins by comparing the element at the bottom of the array with the next element.
- If the first element is larger than the second element then they are swapped or interchanged.
- The process is then repeated for the next two elements. After n-1 comparisons the largest of all the items slowly ascends to the top of the array.
- The entire process till now forms one pass of comparisons.
- During the next pass the same steps are repeated from the beginning of the array, however this time the comparisons are only for n-1 elements.
- The second pass results in the second largest element ascending to its position. The process is repeated again and again until only two elements are left for comparison.
- The last iteration ensures that the first two elements of the array are placed in the correct order.

Algorithm :

Algorithm : Bubblesort (a[ ],n)

```
//Purpose : Arrange numbers in Ascending Order
//Inputs : n - number of items present in the table
        A - the items to be sorted are present in the table
//Output: a - contains the sorted list.
for j=1 to n-1 do
        for i=0 to n-j-1 do
                if(a[i] > a[j+1])
                        temp = a[i]
                        a[i] = a[i+1]
                        a[i+1] = a[temp]
                end if
        end for
end for
```

Time Complexity of BubbleSort : $O(n^2)$

Sequential Search (Linear Search) :

- The simplest of all forms of searching is the linear search.
- This search is applicable to a table organized either as an array or as a linked list.
- Let us assume that A is an array of N elements from A[0] through A[N - 1].
- Let us also assume that ele is the search element. The search starts by sequentially comparing the elements of the array one after the other from the beginning to the end with the element to be searched.
- If the element is found its position is identified otherwise an appropriate message is displayed.

Algorithm :

Algorithm : Linear Search (key,a[ ],n)

//Purpose : This algorithm searches for key element

//Inputs : n - number of elements present in the array

A - elements in the array where searching takes place

Key - item to be searched

//Outputs : The function returns the position of the key if found

Otherwise, function returns -1 indicating search is unsuccessful.

```
for i=0 to n-1 do
    if( key = a[i] )
    return i;
    end if
end for
return -1;
```

Time Complexity : O(N)

Sorting, Sets and Selection:

Sorting :

The process of rearranging the given elements in ascending order or descending order is called sorting.

Divide and Conquer Method :

- It is a top-down approach for designing algorithms which consist of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find.
- The solutions of all smaller problems are then combined to get a solution for the original problem.

Divide and Conquer method involves 3 steps:

A. **Divide :** Problem is divided into a number of sub problems.

B. **Conquer :** If the sub problem are smaller in size, the problem can be solved using straightforward method. If the sub problem size is large, then they are divided into number of sub problems of the same type and size. Each sub problem is solved recursively.

C. **Combine :** The solution od sub problems are combined to get the solution for the larger problem.

1. Merge Sort :

- This sorting method follows the technique of **divide-and-conquer.**
- The technique works on a principle where a given set of inputs are split into distinct subset and the required method is applied on each subset separately i.e., the sub problems are first solved and then the solutions are combined into a solution of the whole.
- Most of the times the sub problems generated will be of the same type as the original problem.
- In such situations re-application of the divide-and-conquer technique may be necessary on each sub problem.

- This is normally achieved through a recursive procedure.
- Thus smaller and smaller sub problems are generated until it is possible for us to solve each sub problem without splitting.

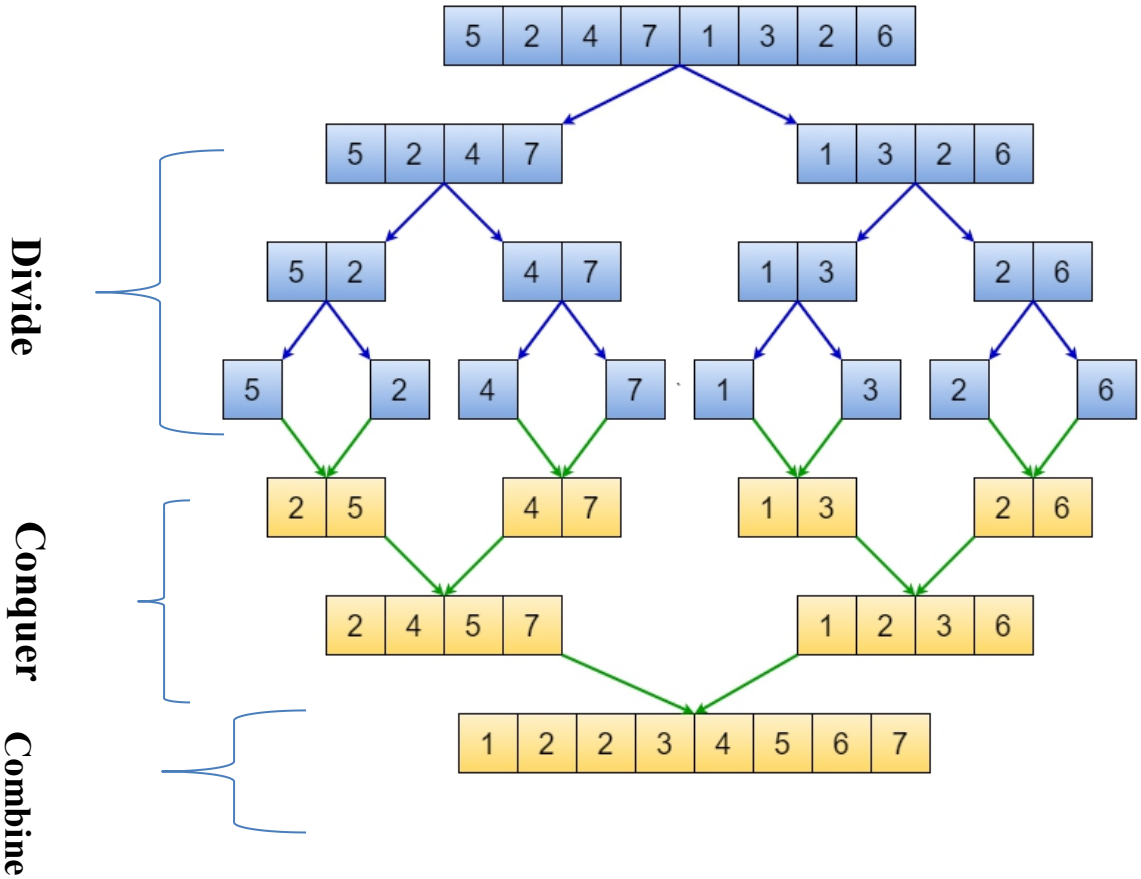
The Steps involved in Merge Sort :

1. **Divide :** Divide the given array consisting of n elements into two parts of n/2 elements each.
2. **Conquer :** Sort the left part of the array and right part of the array recursively using mergesort.
3. **Combine/Merge :** Merge the sorted left part and sorted right part to get a single sorted array.

Example :

N = 8

Elements : 5, 2, 4, 7, 1, 3, 2, 6



Algorithm :

Algorithm MergeSort(a,low,high)

//Purpose :Sort the given array between lower bound and upper bound

//Inputs: a is an array consisting of unsorted elements with low and high as lower bound and upper bound

//Output a:It is an array consisting of sorted elements

```
if (low>high) return //No elements to partition
mid<- (low+high)/2 //Divide the array into two parts
MergeSort (a,low,mid) //Sort the left part of the array
MergeSort (a,mid+1,high)//Sort the right part of the array
SimpleMerge (a,low,mid,high) // Merge the left part and right part
//End of the algorithm MergeSort
```

Simple Merge (a,low,mid,high)

//Purpose: Merge two sorted arrays where the first array starts from low to mid and the second

// starts from mid+1 to high

//Input : a is sorted from the index position low to mid

// a is sorted from index position mid+1 to high

//Output : a is sorted from index low to high

```
i<-low
J<- mid+1
k<-low
```

```
while(i<=mid and j<=high)
        if( a[i] < a[j] ) then
                c[k] <- a[i]
                i<-i+1; k<-k+1 ;
        else
                c[k]<-a[j]
                j<-j+1 ; k<-k+1 ;
        end if
end while
while(i<=mid)
        c[k]<-a[i]
        k <- k+1, i <- i+1
end while
while(j<=high)
        c[k]<-a[j]
        K <- k+1, j <- j+1;
end while
for i=low to high // Copy the elements from c to a a[i]<- c[i]
end for     //End of Algorithm Simple Merge
```

Analysis of Merge Sort :

$T(n) = \theta (n\log_2 n)$

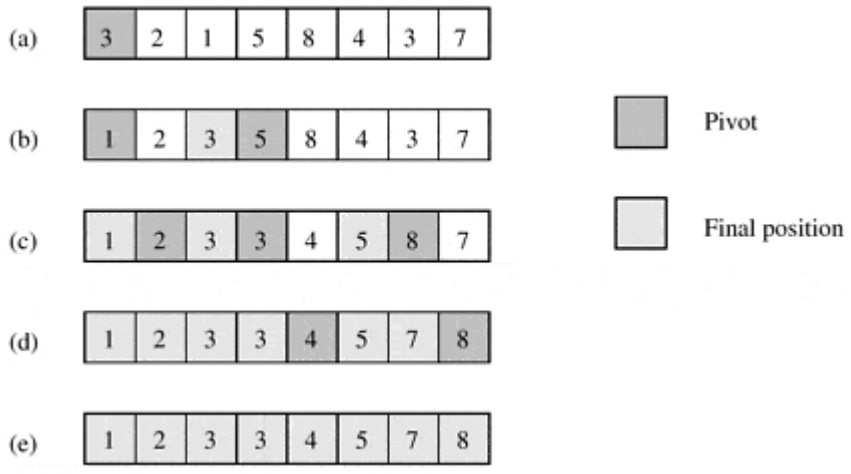
2. Quick Sort

This is one of the best techniques for a large set of data. This technique also works on the method of partitioning. The process of divide and conquer is again recursively applied. In the merge sort method of sorting, the array was partitioned exactly at the midpoint, however in this method of sorting the partition is created at a position such that the elements to the left of the partition is less than the elements to the right of the partition. The entire process functions as follows. First we choose an element from a specific position in the array to be sorted (for example let the element "ele" be chosen as the first element of the array "a" so that ele = a[0]). Suppose that the elements of the array a are partitioned so that ele is placed into the position "p" and the following conditions are satisfied.

- Each of the elements in the positions 0 through p-1 is less than or equal to ele.
- Each of the elements in the positions j+1 through n-1 is greater than or equal to ele.

The purpose of quick sort is to move the data item in the correct direction jus enough for it to reach its final place. Thus the amount of swapping is greatly reduced and the required item moves a greater distance in a shorter duration of time.

Example :



Algorithm :

Algorithm : QuickSort(a,low,high)

//Purpose :Sort the given array using quicksort

//Inputs: low: The position of first element in array a

high: The position of the last element of array a

a: It is an array consisting of unsorted elements

//Output a:It is an array consisting of sorted elements

if (low>high) return //No elements to partition

k<--partition(a,low,high) //Divide the array into two parts

QuickSort(a,low,k-1) //Sort the left part of the array

QuickSort(a,k+1,high) //Sort the right part of the array

"**algorithm QuickSort Ends here**"

Algorithm : partition(a,low,high)

key<-a[low]

i<-low

j<-high+1

while(i<=j)

do i<-i+1 **while**(key>=a[i])

do j<-j-1 **while**(key<a[j])

end while

If (i<j) exchange(a[low],a[j])

return j **//End of the algorithm Partition**

Analysis of Quick Sort :

Best Case : $T(n) = \theta (n\log_2 n)$

Worst Case : $T(n) = O (n^2)$

Average Case : $T(n) = O (n\log_2 n)$

3. Bucket sort

- **Bucket sort**, or **bin sort**, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.
- Several buckets are created. Each bucket is filled with a specific range of elements. The elements inside the bucket are sorted using any other algorithm. Finally, the elements of the bucket are gathered to get the sorted array.
- The process of bucket sort can be understood as a scatter-gather approach. The elements are first scattered into buckets then the elements of buckets are sorted. Finally, the elements are gathered in order.

Bucket sort works as follows:

- Set up an array of initially empty "buckets".
- Scatter: Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather: Visit the buckets in order and put all elements back into the original array.

```
BucketSort ( )
Create N buckets each of which can hold a range of values
    for all the buckets
      initialize each bucket with 0 values
      for all the buckets
      put elements into buckets matching the range
      for all the buckets
      sort elements in each bucket
      gather elements from each bucket
End bucket sort
```

Example :

Time Complexity :

Worst Case Complexity: O(n^2)

Best Case Complexity: O(n+k)

Average Case Complexity: O(n)

Radix Sort :

- Radix sort is a small method that many people intuitively use when alphabetizing a large list of names.
- Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.
- Intuitively, one might want to sort numbers on their most significant digit.
- However, Radix sort works counter-intuitively by sorting on the least significant digits first.
- On the first pass, all the numbers are sorted on the least significant digit and combined in an array.
- Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

Example :

Time Complexity :

n: number of elements

k: the range of the keys for each number. We will also repeat the operation for this amount.

All of the Time Complexities of Radix Sort is always **O(n*k)**

Algorithm :

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort
countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```