Course: FA Name: Manish Parajuli

1.) What is the brute force algorithm? Explain with a suitable example.

Ans:

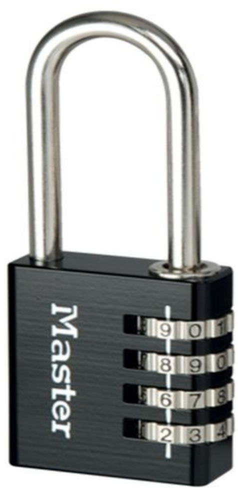
Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

For example, imagine you have a small padlock with 4 digits, each from 0-9. You forgot your combination, but you don't want to buy another padlock. Since you can't remember any of the digits, you have to use a brute force method to open the lock.

So you set all the numbers back to 0 and try them one by one: 0001, 0002, 0003, and so on until it opens. In the worst case scenario, it would take 104, or 10,000 tries to find your combination.

The time complexity of brute force is **O(mn)**, which is sometimes written as **O(n*m)** . So, if we were to search for a string of "n" characters in a string of "m" characters using brute force, it would take us n * m tries.

Example:



There are 4 digits in padlock, each in only 1 of 10 states (0,1,2,3,4,5,6,7,8,9) for a total of 10,000 combinations. Only one of these combinations unlocks the lock, but that's not a necessary constraint. The more important point is that a brute-force algorithm would iterate an entire domain of combinations in search of the correct answer. The search space can be reduced by knowing things like "*the combination starts with 9*" but this still leaves you with 1000 combinations to brute-force search. However, with a better understanding of your solution space you can often disambiguate to a single unambiguous solution in much faster time.

The combinations are like:
First Iteration: (9862), (9863), (9864), (9865), (9866), … , (9869), (9861)
Second Iteration: (9872), (9873), (9874), (9875), (9876), … , (9879), (9871)
.
.
.
(9892), (9893), (9894), (9895), (9896), … , (9899), (9891)
(9812), (9813), (9814), (9815), (9816), … , (9819), (9811)
.
.
.
(9852), (9853), (9854), (9855), (9856), … , (9859), (9851)

Second Step:
First Iteration: (9962), (9963), (9964), (9965), (9966), … , (9969), (9961)
(9872), (9873), (9874), (9875), (9876), … , (9879), (9871)
.
.
.
(9992), (9993), (9994), (9995), (9996), … , (9999), (9991)
(9912), (9913), (9914), (9915), (9916), … , (9919), (9911)
.
.
.
(9952), (9953), (9954), (9955), (9956), … , (9959), (9951)

The total complexity of the padlock brute force algorithm is O(n^4).

2.) Sort the given array {30,15,31,47,25,18} by using selection, merge and quick sort technique. Which technique is efficient and why?

Date
Page

Sort array using merge sort technique:

Like quick sort, merge sort is a Divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

arr[] = 30, 15, 31, 47, 25, 18

Divide equal into two sub array.

30, 15, 31   47, 25, 18

30, 15   31   47, 25   18

30   15   31   47   25   18

Merge the array, now we divide

~~30, 15 81~~

15, 30   31   25, 47   18

15, 30, 31   18, 25, 47

15, 18, 25, 30, 31, 47

Sorted list = 15, 18, 25, 30, 31, 47

~~Algorithm:~~

~~step 1: if it only one element in the~~

Average Time Complexity = $O(n * \log n)$

Date
Page

Sort array using Quick sort:
A large array is partitioned into arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

arr[] = 30, 15, 31, 47, 25, 18
Lets take, pivot = 30
pivot values compares with all arr[0--5] and divides the array into 3 parts:
left side is smaller than pivot and at the right side, greater than pivot and in the middle is pivot.

Ar => 15, 25, 18 30 31, 47
Again in sub array,
pivot = 15 pivot = 31
(nothing in left side) 15 25, 18 30 31 47
pivot = 25
15, 18 25 30 31 47
Now the array is sorted and merged again:
arr[] = 15, 18, 25, 30, 31, 47

Average time complexity = O(n*logn)

Date
Page

Array = {30, 15, 31, 47, 25, 18}

Sort array using selection sort technique:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

arr[] = 30, 15, 31, 47, 25, 18

Find the minimum element in ~~arr[0 ... 5]~~ ~~and~~ arr[0 ... 5] and place it at beginning.

⇒ arr[] = 15, 30, 31, 47, 25, 18

Find the minimum element in arr[1 ... 5] and place it at beginning ~~of~~ arr[1].

⇒ arr[] = 15, 18, 31, 47, 25, 30

Find the minimum element in arr[2 ... 5] and place it at arr[2]

⇒ arr[] = 15, 18, 25, 47, 31, 30

Find the minimum element in arr[3 ... 5] and place it at arr[3]

⇒ arr[] = 15, 18, 25, 30, 31, 47

Find the minimum element in arr[4 ... 5] and place it at arr[4]

⇒ arr[] = 15, 18, 25, 30, 31, 47

Sorted array is arr[] = 15, 18, 25, 30, 31, 47.

Time complexity = $O(n^2)$

The time complexity of Quicksort is O(n log n) in the best case, O(n log n) in the average case, and O(n^2) in the worst case. But because it has the best performance in the *average case* for most inputs, Quicksort is generally considered the “fastest” sorting algorithm. At the end of the day though, whatever the best sorting algorithm really is depends on the input (and who you ask). That’s why Quick sort is efficient.