PW SKILLS

Lesson Plan

Quick Sort Algorithm

PW SKILLS

Topic : Quicksort algorithm

This algorithm follows the divide and conquer approach. In the divide and conquer approach, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Divide: In Divide, firstly, a pivot element is chosen. After that subarrays are rearranged in such a manner that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Two subarrays are sorted recursively with Quicksort.

In Quicksort the pivot element is chosen , and then the given array is partitioned around the picked pivot element.Here, a large array is divided into two sub arrays in which one holds values that are lesser than the pivot value, and another sub array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned around the picked pivot element using the same approach.This process continues until the single element remains in the sub-array.

Choosing the pivot element

There are many different versions of quicksort where the pivot element is chosen in different ways.

- Pivot element can be chosen as the first element of the array.
- Pivot element can be chosen as the last element of the array.
- Any random element from the array can be chosen as a pivot.
- Median of the array can be chosen as the pivot.

Algorithm:

```
Quicksort(array Arr, first, last)
{
     if (first < last)
 {
     p = partition(Arr, first, last)
     Quicksort (Arr, first, p - 1)
               Quicksort (Arr, p + 1, last)
           }
}
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in place.

```
Partition (Arr, first, last)
{
 pivot = Arr[last]
 i = first-1
 for j = first to last -1 {
 do if (Arr[j] < pivot) {
 then i = i + 1
 swap Arr[i] with Arr[j]
      }
 }
 swap Arr[i+1] with Arr[last]
 return i+1
}
```

Topic : Working of quicksort algorithm

To understand the working of quick sort, let's take an unsorted array to make the concept more clear and understandable.

Let the elements of array be -

[18, 12, 30, 16, 35, 20]

In the given array, the rightmost element is considered as pivot. So, in this case, a[left] = 18, a[right] = 20 and pivot = 20. At each step, **right** will be incremented by 1, while **left** will increment only when we need to swap elements smaller than pivot with the element pointed by **left**. This ensures that whenever **right** points to an element smaller than pivot, then such elements will be placed one by one at the initial indices of the array, given by **left**.

[18, 12, 30, 16, 35, 20] **left = -1, right = 0, pivot = 20**

Now, **a[right] < pivot**, so the algorithm increments left and then swaps **a[left]** with **a[right]**, i.e. swapping a[0] with a[0] (No effective change in the array).

[18, 12, 30, 16, 35, 20] **left = 0, right = 1, pivot = 20**

Because, **a[right] < pivot**, so the algorithm increments left and then swaps **a[left]** with **a[right]**, i.e. swapping a[1] with a[1] (No effective change in the array).

[18, 12, 30, 16, 35, 20] **left = 1, right = 2, pivot = 20**

Now, **a[right] > pivot**, so the algorithm simply increments **right** without any change in the array.

[18, 12, 30, 16, 35, 20] **left = 1, right = 3, pivot = 20**

Now, **a[right] < pivot**, so the algorithm increments left and then swaps **a[left]** with **a[right]**, i.e. swapping a[2] with a[3].

[18, 12, 16, 30, 35, 20] **left = 2, right = 4, pivot = 20**

Now, **a[right] > pivot**, so the algorithm simply increments **right** without any change in the array.

[18, 12, 16, 30, 35, 20] **left = 2, right = 5, pivot = 20**

Now, as a final step, since right = 5 (size of the array - 1), We simply swap **a[left+1]** with **a[right] (pivot element)**

[18, 12, 16, 20, 35, 30] ← Array after partitioning about pivot = 20.

Elements that are on the right side of element 20 are greater than it, and the elements that are on the left side of element 20 are smaller than it.

Now, in a similar manner, the quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -

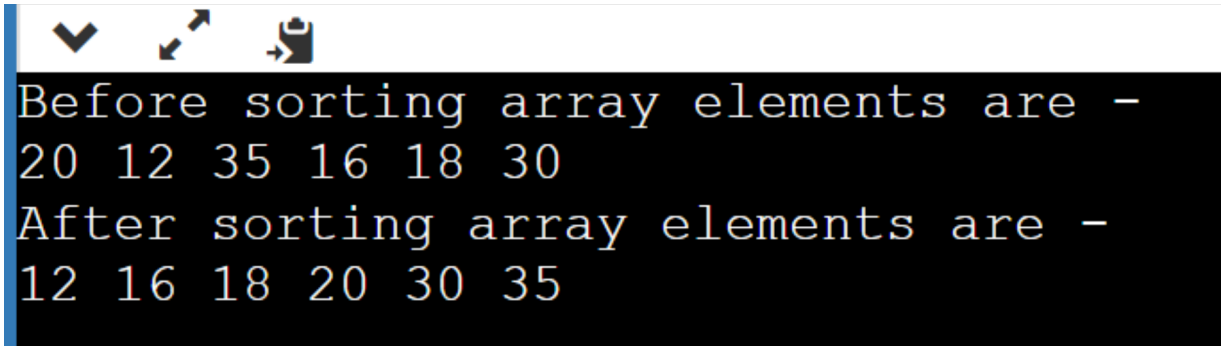
[12, 16, 18, 20, 30, 35]

Let's write the cpp program to sort the same array using the last element as pivot and see the result !!

Java program for Quicksort algorithm

https://pastebin.com/fnRjr9yR

Output:



Topic - Time and Space Complexity

Time Complexity	Time Complexity
Best	O(n*log n)
Worst	O(n2)
Average	O(n*log n)

Space Complexity: In the worst case scenario, partition is always unbalanced and there will be only 1 recursive call at each level of recursion.So space complexity will be O(n).

Topic : Need for new ways of partitioning and Randomised Quicksort algorithm

The worst case time complexity comes out to be O(n2) which happens because of wrongly chosen pivot elements in the worst case scenario, which fails to partition the array in a balanced way. If we somehow choose a pivot element that tends to keep the array partitioning balanced, then we can claim the worst case time complexity to be O(n * log n).

Let us consider an example, say we wish to partition the array **[5, 4, 3, 2, 1]**.

Choosing the last element as pivot everytime would break the array into two parts: one array contains x-1(x being size of the array to be partitioned at any recursive call) elements and another contains zero element.

[5, 4, 3, 2, 1] → [5, 4, 3, 2], [].
[5, 4, 3, 2] → [5, 4, 3], [].
[5, 4, 3] → [5, 4], []
[5, 4] → [5], [].
[5] → [], []

With such partitioning, we will be making n recursive calls (as we separate out only a single element in every call, which is the pivot chosen). So the time complexity comes out to be O(n^2)

Let's try to partition everytime on the basis of the median of the subarray.

[4, 6, 7, 5, 3, 2, 1] → [3, 2, 1], [6, 7, 5]
Since the median of [1, 2, 3, 4, 5, 6, 7] is 4.

[3, 2, 1] → [1], [3] [6, 7, 5] → [5], [7]
Since the median of [3, 2, 1] is 2 and the median of [6, 7, 5] is 6.

Now, this is the most optimal partitioning that we can have in any case. Changing a pivot element in any of the above cases would result in more recursive calls than the minimum number of calls required. However, for obvious reasons it would not be possible to implement this approach as for finding out the median we have to sort the subarray and we cannot use sorting to implement sorting! Hence the argument of partitioning about the median becomes senseless.

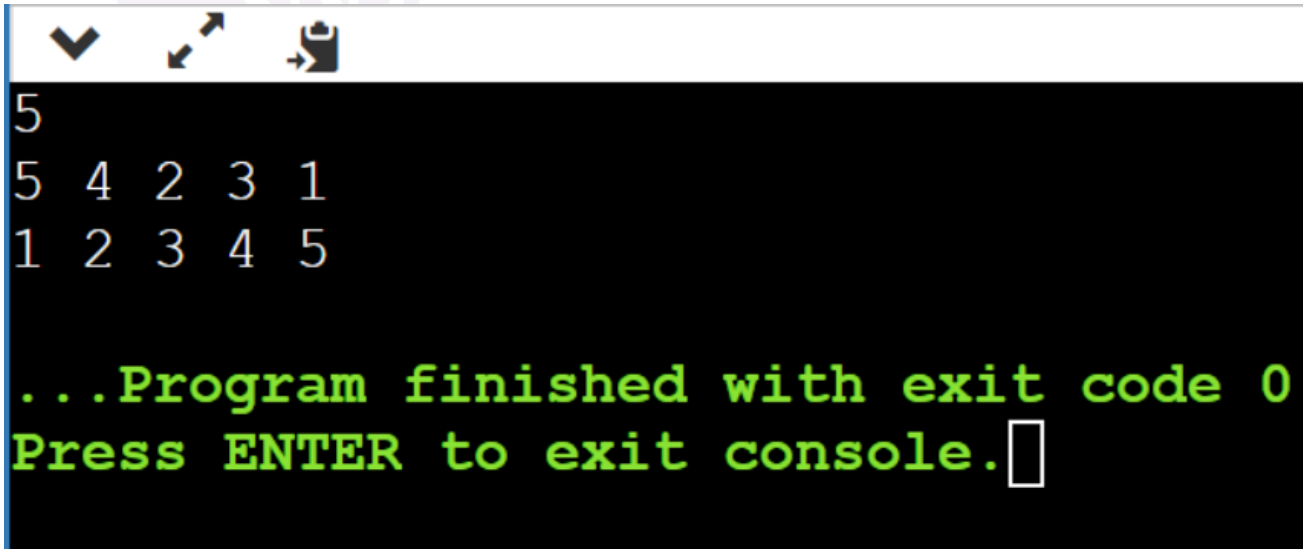
What intuitively comes to mind is to use a randomized approach. What we can do is, we can select a random pivot from the subarray that needs to be partitioned. We can probabilistically assume that the partitioning at each step will be done in a way that more or less tends to minimize the number of recursive calls.

From the implementation point of view, we can simply choose an index from 0 to subarray's size - 1 and then can simply swap it with the last element. The algorithm to partition using the last element of the subarray as pivot can then be used as it is. Addition of this simple step allows us to choose a random element of the subarray as pivot.

We have a built-in pseudo random number method in java which would help in selecting the random index, which gives us the element about which the partition is to be done.

```
// To introduce randomness in selecting pivot element.
   int randomIndex = first + (int) Math.random() % (last -
first + 1);
   swap(arr[randomIndex], arr[last]);
```

https://pastebin.com/u61ApSPg



Topic : Is Quicksort stable?

QuickSort is an unstable algorithm because elements are swapped according to pivot's position (without considering their original positions).

Topic : Applications of Quicksort:

- It is used in the place where the main concern is memory.
- It is used in sorting the element in excel application.
- Some programming languages inbuilt sort functions are built using quicksort algorithm only .
- Java sorting function is based on Quicksort only.

Topic : Difference between Quicksort and Mergesort.

1. Merge sort is more proficient and works speedier than quick sort in case of bigger array size or datasets. whereas Quick sort is more efficient and works speedier than merge sort in case of smaller array size or datasets.
2. Merge sort is preferred for linked lists since linked lists cannot be randomly accessed, and in merge sort, we don't require random access, while in quicksort, we need to randomly access elements so Quick sort is preferred for arrays.
3. Merge sort is stable but Quicksort is unstable.
4. If the cost of allocating new memory is very high, we should always prefer quicksort since it is an in-place sorting algorithm while merge sort requires additional memory

PW SKILLS

THANK YOU !

PW SKILLS