upGrad

Module Name - Searching & Sorting algorithms

Topic Name: Merge Sort & Quick Sort

Instructor: Arun Kudiyal

Time Allocation Summary

Element	Slide numbers	Maximum time(min)
Today's Agenda		
Spot Test		
Merge Sort – Java Implementation & Algorithm Analysis		
Quick Sort – Java Implementation & Algorithm Analysis		
Sample Problem Statement		
Total Time		120

Today’s Agenda

- Introduction to Merge Sort
- Java Implementation of Merge Sort
- Time Complexity of Merge Sort
- Introduction to Quick Sort
- Time Complexity of Quick Sort

Spot Test

Let’s take a quick revision assessment of previously taught topics :-

- Bubble Sort
- Selection Sort
- Insertion Sort

Merge Sort

- So, we have learnt 3 algorithms so far but all of them have the time complexity of O(n^2). But can we further reduce this time complexity by using some other algorithms?

- What if we try to divide our problems into subproblems and then try??? Yes, once again, it's **Divide and Conquer** to the rescue!!!

- We'll be looking at Merge Sort algorithms today.

Merge Sort

upGrad

- Let’s start by dividing our main array into two subarrays:

8	6	3	1	4	2	7	5

8	6	3	1

4	2	7	5

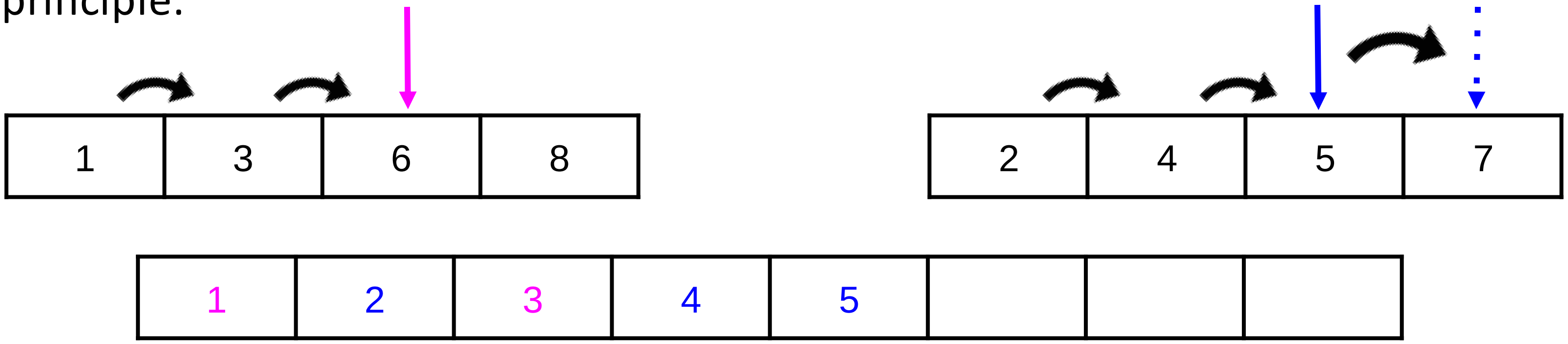
- If we somehow manage to sort these subarrays individually, can you think of how to combine(or MERGE) them?

1	3	6	8

2	4	5	7

Merge Sort

- Let’s start with **two pointers at the starting of each subarray** and start filling our original sized array with the sorted elements from these subarrays one at a time.
- Just simply increment the pointer of the subarray whose pointer element is smaller than the pointer element of the other subarray after writing it to the main array.
- The arrays and subarray will look like this somewhen during the process using this principle:



Merge Sort

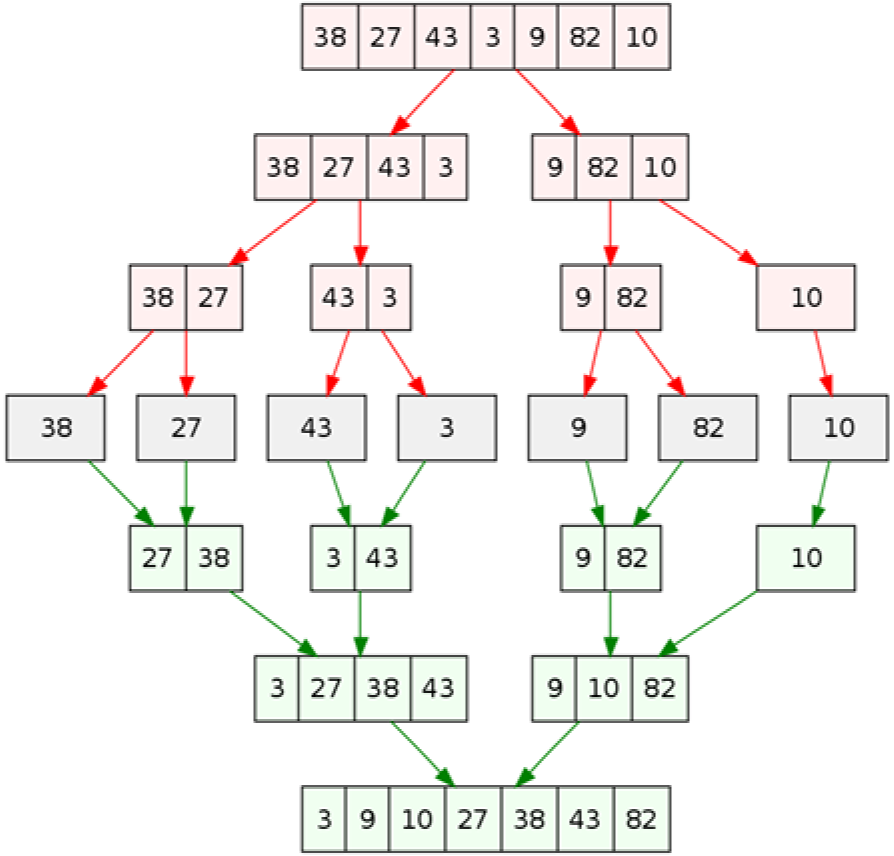
- Now that you know the process of merging two sorted subarrays, it’s time to find out how to sort these two subarrays.

- We divided our main array into two subarrays, we can also divide our subarrays into two subarrays each and merge those sub-subarrays to obtain our sorted subarrays.

- We can keep splitting each subarray in the same way into smaller and smaller subarrays. What do you think will be the terminating condition for this recursion?

Merge Sort

- We will continue this division of subarrays until there is only one element in the subarray, which will make it sorted by default.
- And then we can start merging from that point and finally obtain our sorted array!
- This example towards the right will clarify the entire process to you furthermore.



Java Implementation of Merge Sort

upGrad

- Well, that was the logic behind the Merge Sort algorithm. Let’s start with the code now.

- You know that by now, that the algorithms has two main functions; to split the array into subarrays and sub-subarrays, and to merge the sorted subarrays till we get the full sized sorted array. Let’s take a look at the first recursive function for dividing the subarrays:

```
        public static int[] mergeSort(int[] numbers, int first, int
last){
         if (first < last) {
               int mid = (first + last) / 2;
               mergeSort(numbers, first, mid);
               mergeSort(numbers, mid + 1, last);
               merge(numbers, first, mid, last);            }
         return numbers;
      }
```

Java Implementation of Merge Sort

upGrad

- Now, let's look at the main function, `merge` which was called in our recursive function, `mergeSort` after every division of a subarray. The objective of this function, as the name suggests, is to merge the two sorted subarrays into a bigger sorted array/subarray.

```
private static void merge(int[] numbers, int i, int m, int j) {
int l = i;                          // initial index of first subarray
int r = m + 1;      // initial index of second subarray
int k = 0;                          // initial index of merged array
int[] t = new int[numbers.length];

while (l <= m && r <= j) {
    if (numbers[l] <= numbers[r]) {
        t[k] = numbers[l];
        k++;
        l++;
    } else {
        t[k] = numbers[r];
        k++;
```

Do you think the function is complete?
Or is there anything that should be added?

Java Implementation of Merge Sort

- The given code segment did not cover the case when one subarray is completed and other one still has some/all elements remaining. To cover it, check the following code:

```
     // Copy the remaining elements on left half , if there are any
while (l <= m) {
    t[k] = numbers[l];
    k++;
    l++;
}
// Copy the remaining elements on right half , if there are any
while (r <= j) {
    t[k] = numbers[r];
    k++;
    r++;
}
```

- The only part now left, is to update the original array with the sorted elements...

Java Implementation of Merge Sort

upGrad

- The given code segment updates all the elements of our sorted array into the original array:

```
l = i;
k = 0;
while (l <= j) {
    numbers[l] = t[k];
    l++;
    k++;
}
```

- This completes our code of merge sort. Do you have any queries regarding it?

Time Complexity of Merge Sort

upGrad

So, we have seen the complete code of Merge Sort. Let’s analyze its time complexity now:

- **Step 1: Divide**
  For dividing, we will compute the middle of the given ‘n’ number of elements. This would take a constant time, let’s say $D(n)= \Theta(1)$

- **Step 2: Conquer**
  The sequence of ‘n’ elements is divided into n/2 elements each and those, further into n/2 each. Therefore, we are recursively solving two sequences of n/2 elements that give us time complexity, let’s say $2T(n/2)$

- **Step 3: Merge**
  Combining the subarrays of ‘n’ elements would take time $C(n)= \Theta(n)$.

Time Complexity of Merge Sort

upGrad

We can therefore conclude that:

$T(n) = 0$, if $n<=1$
$T(n) = T(n/2) + T(n/2) + D(n)+ C(n)$, otherwise

OR

$$\mathbf{T(n)= 2*T(n/2) + \Theta(n) + \Theta(1)}$$

For ease of calculations, let's ignore $\Theta(1)$ and replace $\Theta(n)$ with n.

i.e **$T(n)= 2*T(n/2) + n$** and using Substitution,

$T(n/2) = 2*(2*T(n/4) + n/2)$ $=2^2 T(n/2^2)+n$

Time Complexity of Merge Sort

upGrad

Using the substitution, we can rewrite the T(n) equation as:

$$T(n) = 2^2 * T(n/2) + 2n$$ and using subsequent substitutions,

.

.

.

$$= 2k(2 * T(n/2) + kn$$

Now, assuming $n=2^k$ and therefore $k=\log_2 n$:

$$T(2^k) = nT(n/n) + \log n * n$$

$$= n * T(1) + n\log n$$

And since T(1) = O(1),

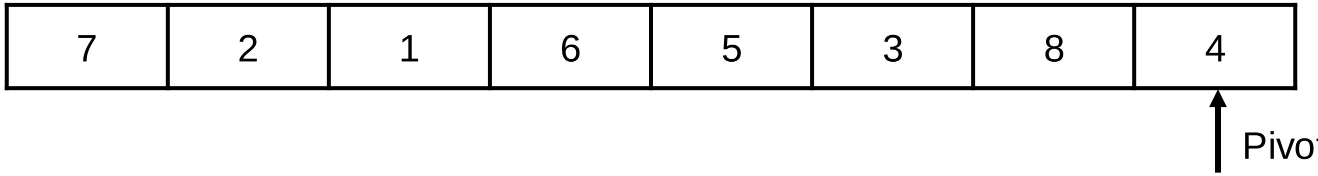
T(n) = O(nlogn)

Quick Sort

upGrad

Finally, we got to see our first **nlogn** sorting algorithm. Now let's QUICKLY check the next algorithm as well. And yes, this one follows the divide and conquer technique too.

- Let's once again start by dividing our main array into two subarrays. BUT, instead of splitting through centre, let's choose a pivot element, say LAST element i.e. **4** and sort the array into elements smaller and larger than 4.



- But our approach will be to rearrange the original array instead of creating a new array of the same size. This will be done with the help of a partition pointer other than start and end pointers. Let's take a closer look step by step:

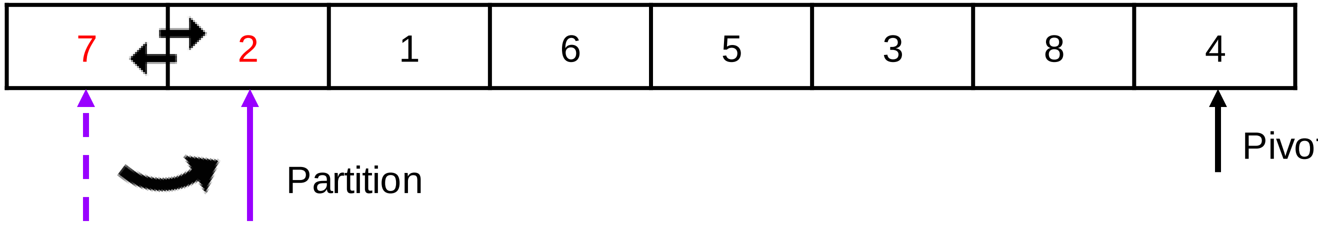
Quick Sort

upGrad

- We'll start with the partition pointer at the first element i.e. 7 and iterate throughout the array. If an element is lesser than pivot element i.e. 4 here, then:
  - The element is swapped with the element at the partition pointer
  - The partition pointer is incremented after swap

- So, when the loop runs, the first step is comparing the 1st element(7) with pivot(4). Since 7 is not smaller than 4, nothing happens and we move to next element i.e. 2.

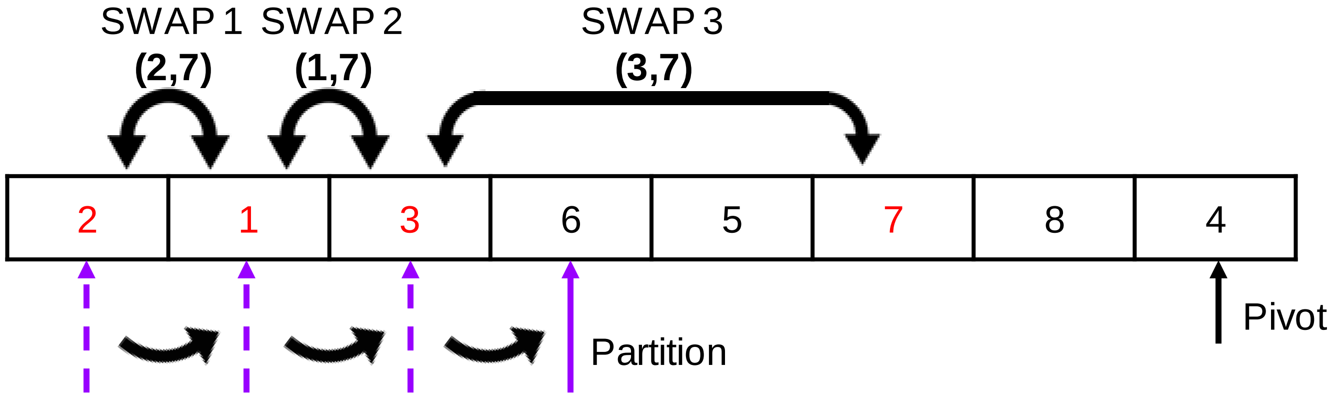
- Now, since 2 is smaller than 4, 2 and 7 are swapped and partition pointer is incremented and again points to 7.



Quick Sort

upGrad

- Now, let’s do the same procedure for the remaining array. The resultant array will be:

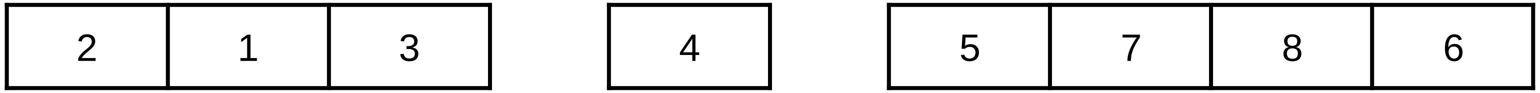


- Now, we should place our pivot(4) at its correct position in sorted array. Can you tell the approach to complete our objective?

Quick Sort

upGrad

- Yes, you guessed it right! We simply need to make a final swap of our pivot element with the element pointed by the partition pointer. The final thus becomes:



- Now, we can use the same algorithm to sort the sub arrays to the left and right of our pivot element i.e. 4. And using this partition function recursively, we can sort the whole array. Let’s take a look at the final code.

Java Implementation of Quick Sort

upGrad

- Like merge sort, this algorithm also has two main functions:
  - The recursive calling function, `quickSort`, to divide the arrays into subarrays to be sorted
  - The `partition` function, which puts the pivot element into its correct position and partitions the array into elements smaller(and equal), and larger than the pivot element.

- Let's see both these functions one by one:

```
public static void quickSort(int[] ar, int start, int end) {
 if (start < end) {
      int p = partition(ar, start, end);
      quickSort(ar, 0, p - 1);
      quickSort(ar, p + 1, end);                     }}
```

Java Implementation of Quick Sort

```
public static int partition(int[] ar, int start, int end) {
    int pivot = ar[end];
    int i = start;
    for (int j = start; j < end; j++) {
        if (ar[j] <= pivot) {
            int temp1 = ar[j];
            ar[j] = ar[i];
            ar[i] = temp1;
            I++;          }}

    int temp2 = ar[i];
    ar[i] = ar[end];
    ar[end] = temp2;

    return i;                   }
```

Time Complexity of Quick Sort

upGrad

Now that we have seen the code, you know the next step. Yes, it’s analyzing the time complexity. But this time we will find out the time complexity for both the worst and best case scenarios.

- What do you think will be the worst case scenario for quicksort?

- What do you think will be the best case?

Time Complexity of Quick Sort

The worst case will be when the array is sorted already. Quite ironic, isn't it? But why do you think it is so?

Let's take a look at the quick sort function once again to understand it better:

```
public static void quickSort(int[] ar, int start, int end) {
        if (start < end) {
            int p = partition(ar, start, end);
            quickSort(ar, 0, p - 1);
            quickSort(ar, p + 1, end);          }}
```

Majority of the `partition` function comprises of declaration and assignment statements which follow O(1). The main loop in the function will always run from start to end pointer each time, which means it has the time complexity of O(n).

Time Complexity of Quick Sort

upGrad

Now, if the array is already sorted, partition function will always return **p** as **end**. Therefore, only the the first `quickSort` function is executed every time(total **n** times); the second `quickSort` is never executed.

Since, only the first quickSort sub-function is executed every time, the time complexity equation of quickSort becomes:

T(n) = n+T(n-1)+0
(Corresponding to the 3 statements in `quickSort`)

and using Substitution, **T(n) = [T(n-2)+(n-1)] + n**
=T(n-2)+ 2n-1

Similarly, for kth substitution: **T(n) =T(n-k)+ kn + ∑(k-1)**
=T(n-k)+

Time Complexity of Quick Sort

upGrad

$$T(n) = T(n-k) + kn + k(k-1)/2$$

Now, assuming **n-k=1** and therefore **k=n-1**:

$$T(n) = T(1) + n(n-1) + (n-1)(n-2)/2$$

$$= an^2 + bn + c$$ *(a,b,c are some constant values)*

$$\therefore T(n) = O(n^2)$$

Time Complexity of Quick Sort

upGrad

Now let's find out the best case time complexity. The best case scenario is when the pivot is correctly placed in the exact middle of each array/subarray. Thus, the sorting will be equally divided between both the `quickSort` functions(which obviously is complete opposite of worst case scenario).

Hence, the time complexity equation of the function comes out to be:
T(n) = n+T(n/2) +T(n/2) (Corresponding to the 3 statements in `quickSort)` comes out to be **T(n) = 2*T(n/2)+ n**

Does this equation ring any bells? Have you ever solved this equation before?

YES!!! This is the exact same equation for the analysis for the time complexity of MERGE SORT. So, can you solve this equation on your own now?

Time Complexity of Quick Sort

upGrad

Using the substitution, we can rewrite the T(n) equation as:

T(n) **$= 2^2 * T(n/2) + 2n$** and using subsequent substitutions,

.

.

.

$= 2k(2 * T(n/2) + kn$

Now, assuming **$n=2^k$** and therefore **$k=\log_2 n$** :

$T(2^k)$ **$=nT(n/n)+\log n * n$**

$= n * T(1) + n\log n$

And since T(1) = O(1),

T(n) = O(nlogn)

Quick Sort Vs Merge Sort

You've learnt both Merge Sort and Quick Sort today. But which one is better? Before we get to the answer of this question, let's learn about two new terms first:

- **Stability:** If a sorting algorithm outputs two elements with same key but different values in the same order as the one prior to sorting, it is said to be sorted.
  For eg. Given X-Y coordinate pairs *(11,8), (5,5), (2,18), (9,0), (5,1)* and sorting on the basis of increasing X coordinates, if *(5,1)* comes after *(5,5)*, it is said be stable.

- **In-Place:** If a sorting algorithm does not require extra space of similar size to the input array, i.e does not have space complexity of O(n) or higher, it is said to be in-place.

Now can you tell which of the algorithm is/are stable and in-place?

Quick Sort Vs Merge Sort

Let’s see the comparison with the help of a table:

Quick Sort	Merge Sort
Not stable	Stable
In-Place	Not in-place

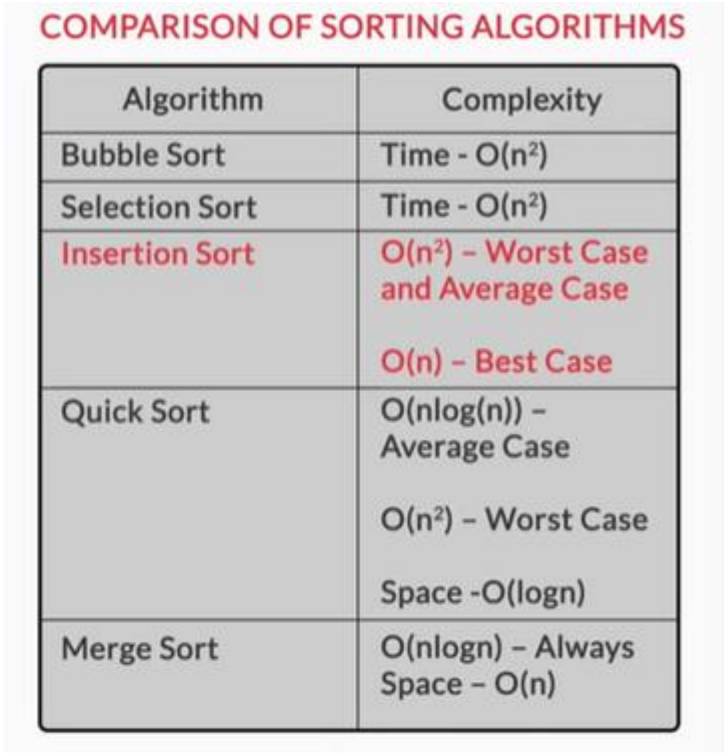
Now you know that if you need to preserve the order of the input elements after sorting, or if you have constraint on memory usage which sorting algorithms to prefer.

Which one will you prefer in the former case?

Comparison of Sorting Algorithms

upGrad

Congratulations!!! You’ve learnt 5 sorting algorithms. The following table summarizes the analysis of all the algorithms:



COMPARISON OF SORTING ALGORITHMS

Algorithm	Complexity
Bubble Sort	Time - $O(n^2)$
Selection Sort	Time - $O(n^2)$
Insertion Sort	$O(n^2)$ – Worst Case and Average Case $O(n)$ – Best Case
Quick Sort	$O(n\log(n))$ – Average Case $O(n^2)$ – Worst Case Space -$O(\log n)$
Merge Sort	$O(n\log n)$ – Always Space – $O(n)$

upGrad
#LifeKoKaroLift

Thank You!

Happy learning!