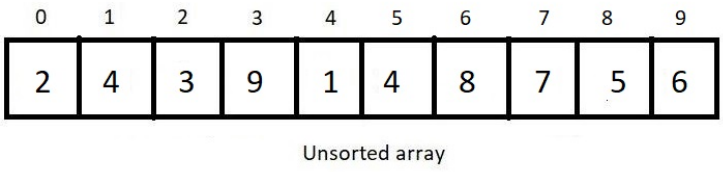
QuickSort Algorithm in Hindi (With Code in C)

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-56

Having finished three of the sorting algorithms, our next concern would be to learn the QuickSort algorithm. We have already finished the bubble sort algorithm, the insertion sort algorithm, and the selection sort algorithm. If you have missed any, please check out the previous videos first. Today we are interested in learning a new sorting algorithm called the **QuickSort Algorithm.**

The QuickSort algorithm is quite different from the ones we have studied so far. Here, we use the divide and conquer method to sort our array in pieces reducing our effort and space complexity of the algorithm. There are two new concepts you must know before you jump into the core. First is the **divide and conquer** method. As the name suggests, Divide and Conquer divides a problem into subproblems and solves them at their levels, giving the output as a result of all these subproblems. The second is the **partition** method in sorting. In the partition method, we choose an element as a pivot and try pushing all the elements smaller than the pivot element to its left and all the greater elements to its right. We thus finalize the position of the pivot element. QuickSort is implemented using both these concepts. And I'll help you master them very soon.

Suppose we are given an array of integers, and we are asked to sort them using the quicksort algorithm, then the very first task you would do is to choose a pivot. Pivots are chosen in various ways, but for now, we'll consider the first element of every unsorted subarray as the pivot. Remember this while we proceed.



Unsorted array

In the quicksort algorithm, every time you get a fresh unsorted subarray, you do a partition on it. Partition asks you to first choose an element as a *pivot*. And as already decided, we would choose the first element of the unsorted subarray as the *pivot*. We would need two more index variables, *i and j*. Below enlisted is the flow of our partition algorithm we must adhere to. We always start from step 1 with each fresh partition call.

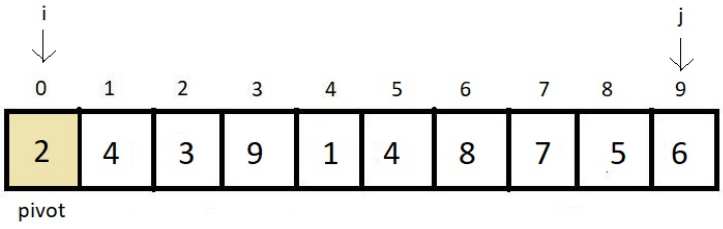
1. Define i as the *low* index, which is the index of the first element of the subarray, and j as the *high* index, which is the index of the last element of the subarray.
2. Set the *pivot* as the element at the *low* index *i* since that is the first index of the unsorted subarray.
3. Increase *i* by 1 until you reach an element greater than the pivot element.
4. Decrease *j* by 1 until you reach an element smaller than or equal to the pivot element.
5. Having fixed the values of *i and j*, interchange the elements at indices *i and j*.
6. Repeat steps 3, 4, and 5 until j becomes less than or equal to i.
7. Finally, swap the pivot element and the element at the index.

This was the partitioning algorithm. Every time you call a partition, the pivot element gets its final position. A partition never guarantees a sorted array, but it does guarantee that all the smaller elements are located to the pivot's left, and all the greater elements are located to the pivot's right.

Now let's look at how the array we received at the beginning gets sorted using partitioning and divide and conquer recursively for smaller subarrays.

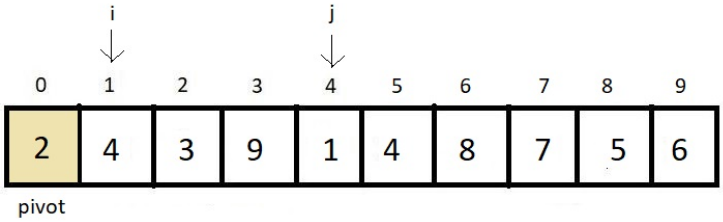
Firstly, the whole array is unsorted, and hence we apply quicksort on the whole array.

Now, we apply a partition in this array. Applying partition asks you to follow all the above steps we discussed.



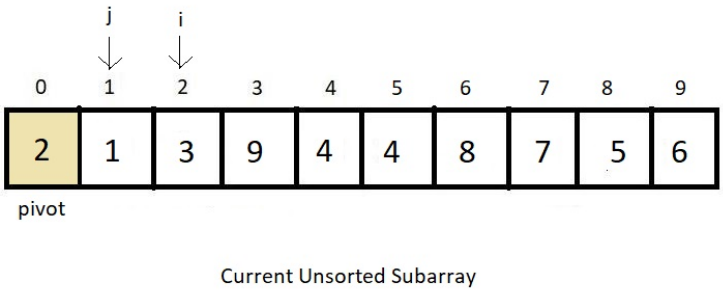
Current Unsorted Subarray

Keep increasing i until we reach an element greater than the pivot, and keep decreasing j until we reach an element smaller or equal to the pivot.

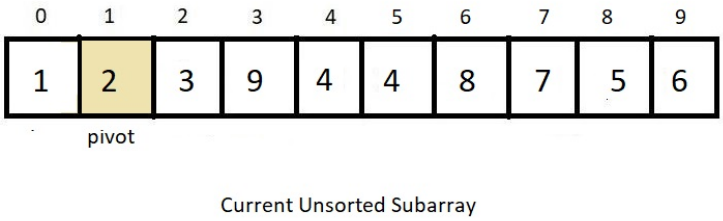


Current Unsorted Subarray

Swap the two elements and continue the search further until j crosses i or becomes equal to i.

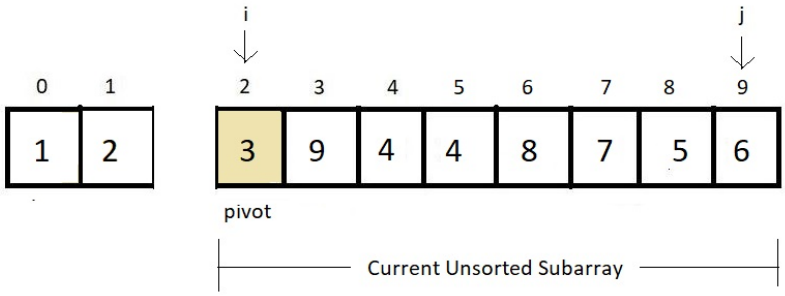


As j crossed i while searching, we followed the final step of swapping the pivot element and the element at j.

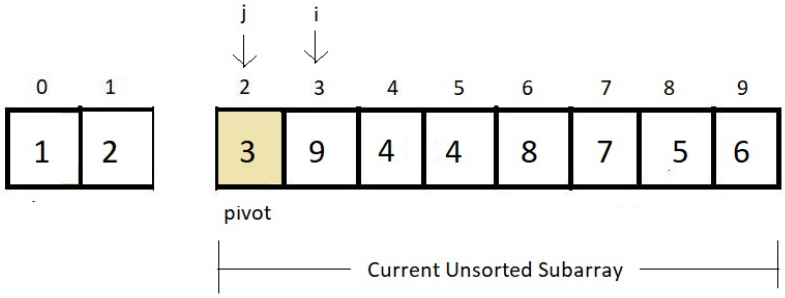


And this would be the final position of the current pivot even in our sorted array. As you can see, all the elements smaller than 2 are on the left, and the rest greater than 2 are on the right. Here comes the role of divide and conquer. We separate our focus from the whole array to just the subarrays, which are not sorted yet. Here, we have subarrays {1} and {3, 9, 4, 4, 8, 7, 5, 6} unsorted. So, we make a call to quicksort on these two subarrays.

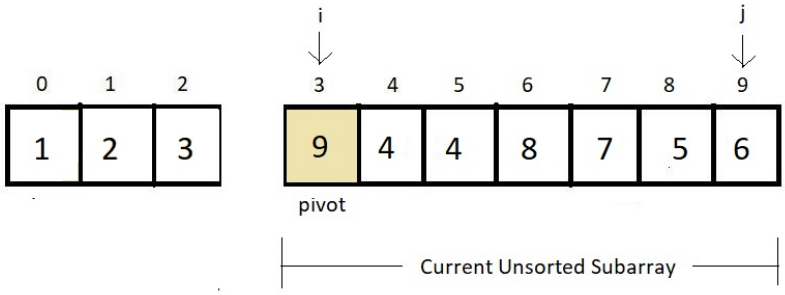
Now since the first subarray has just a single element, we consider it sorted. Let's now sort the second subarray. Follow all the partition steps from the beginning.



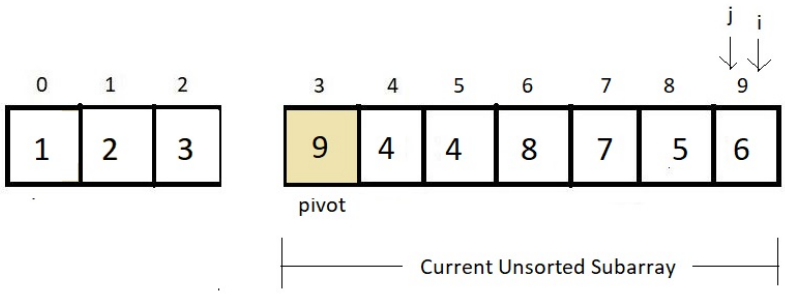
Now, our new pivot is the element at index 2. And i and j are 2 and 9, respectively, marking the start and the end of the subarray. Follow steps 3 and 4.



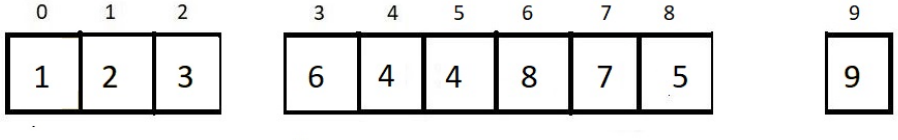
And since there were no elements smaller than 3, j crosses i in the very first iteration. This means 3 was already at its sorted index. And there are no elements to its left; the only unsorted subarray is {9, 4, 4, 8, 7, 5, 6}. And our new situation becomes:



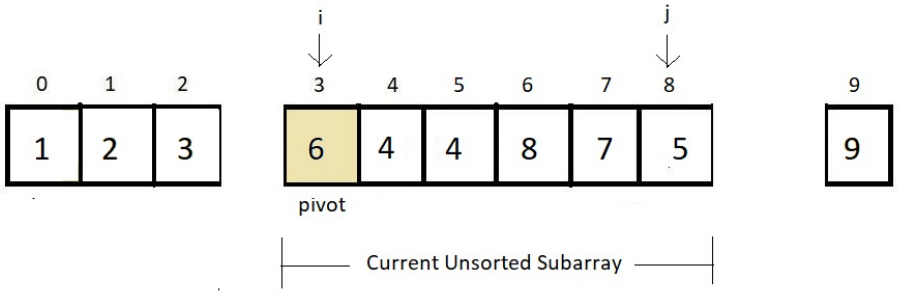
Repeating steps 3 and 4.



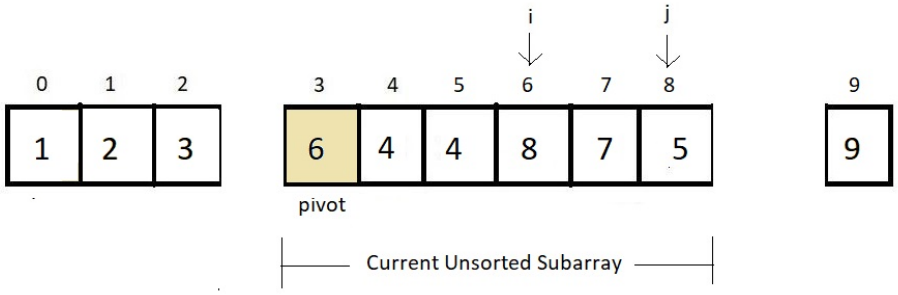
We found that there was no element greater than 9, and hence *i* reached the last. And 6 was the first element j found to be smaller than 9, and they collided. And this is where we do step 7. Swap the pivot element and the element at index j.



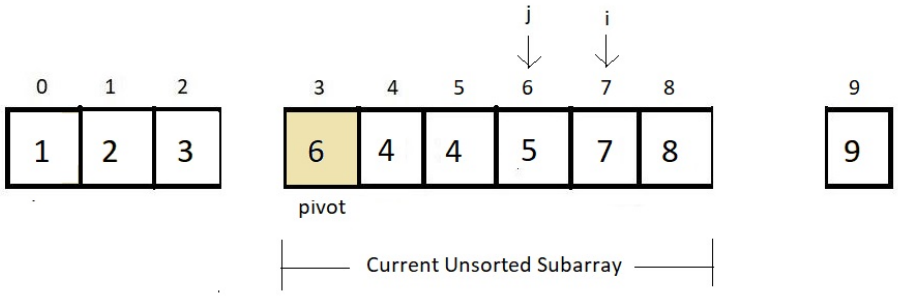
And since there are no elements to the right of element 9, we just have one sorted subarray {6, 4, 4, 8, 7, 5}. Let's call a partition on this as well.



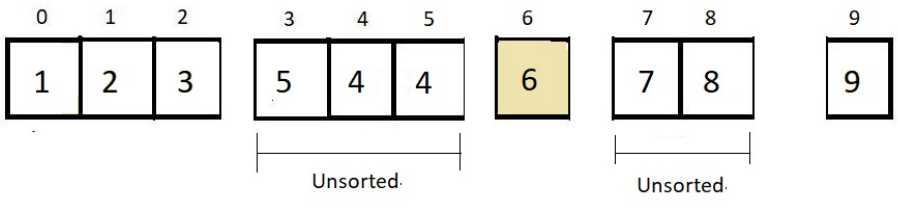
Repeat steps 3 and 4.



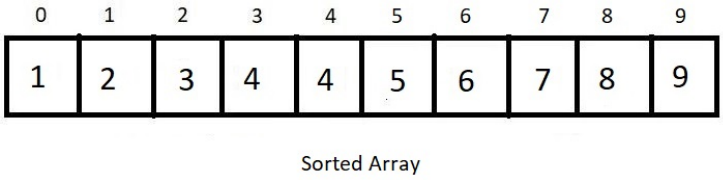
And since the condition j<=i has not been met yet, we just swap the elements at index i and j and continue our search.



And now i and j crossed each other, and now only we swap our pivot element and element at j.



And now, we again divide and consider only the elements that remain unsorted to the left of the pivot and the right of the pivot. And moving things further would just waste our time. We can assume that things move as expected, and it will get sorted at the end and would look something like this.



Sorted Array

Things may not have made much sense since you are here for the first time. Go through the concepts again. And if you are concerned about the divide and conquer thing, we really don't have to worry about how things will go till the end. Recursions are meant to work like this. Rather we will see the program for implementing the QuickSort algorithm, and things will automatically become clear to you. So, let's just appreciate the tough part and try making it simpler by programming its implementation.

Having finished discussing the functioning of the quick sort algorithm, let's now move to the programming part. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. Before we proceed with the core concepts, let's just copy the *printArray* part in our current programs. This just helps a lot seeing the contents of the array before and after the sorting. Anyways, I have attached the snippet for *printArray* as well.

```
void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

Code Snippet 1: Creating the *printArray* function

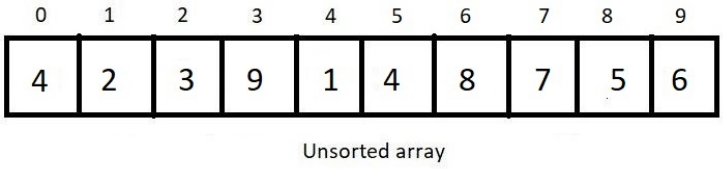
2. The next step is to define an array of elements. As always, we define an array of integers.

3. Define an integer variable for storing the size/length of the array.

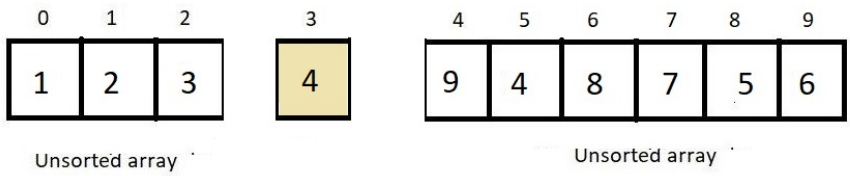
Understanding the quickSort function:

4. If you recall what we did every time we were given an unsorted subarray, we just applied a partition on it. Now, since partition is a different job, we will have a different function for that. But the *quicksort* function just intends to follow things to partition. Like, if you pass an array to *quicksort*, it further passes it to the

partition function, and the partition returns the pivot index after applying all the steps we discussed earlier. *Quicksort* stores this index and recursively calls itself with smaller subarrays which lie on the left and the right of the pivot index.



For example, if you call *quicksort* passing the above array, It would pass it further to the partition, and the partition would return the new index of the pivot element, which is 4. Partition returns 3, the new position of 4. Now, quicksort recursively calls itself on the left and the right subarrays highlighted below.



Creating the quickSort function:

5. Create a void function *quickSort* and pass the address of the array and the lower index, which would be 0 for the first time, and the higher index, which would be *length -1* for the first time, as parameters. Create an integer variable *partitionIndex* for holding the index provided by the *partition*. Now recursively call the quickSort function twice but with parameters changed to (low, *partitionIndex-1)* for the left subarray and (*partitionIndex+1*, high) for the right subarray, instead of just (low, high). But ain't we forgetting something? The basics of recursion demand a base condition to stop the recursion. Hence, the base condition here would be when our low becomes greater than or equal to our high. This is when our recursion stops.

```
void quickSort(int A[], int low, int high)
{
    int partitionIndex; // Index of pivot after partition

    if (low < high)
    {
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1);  // sort left subarray
        quickSort(A, partitionIndex + 1, high); // sort right subarray
    }
}
```

Code Snippet 2: Creating the *quickSort* function

Creating the partition function:

6. Create a void function *partition,* and pass the address of the array and the *low* and the *high* of the subarray as parameters. Create an integer variable *pivot* that takes the element at the low index. Create two index variables, *i and j,* and make them hold *low+1* and *high*

Create a while loop and run until the index *i* reaches an element greater than or equal to the pivot or the array finishes. Till then, keep increasing i by 1. Similarly, create another while loop and run until our index j reaches an element smaller than the pivot or the array finishes. Till then, keep decreasing j by 1. After finishing all the above tasks, we swap the elements at indices i and j.

The above process is repeated using a do-while loop until i becomes greater than j. And when it does, the loop breaks, and before we return, we swap our pivot with the element at index j. And that should finish our job.

```
int partition(int A[], int low, int high)
{
    int pivot = A[low];
    int i = low + 1;
    int j = high;
    int temp;

    do
    {
        while (A[i] <= pivot)
        {
            i++;
        }

        while (A[j] > pivot)
        {
            j--;
        }

        if (i < j)
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    } while (i < j);

    // Swap A[low] and A[j]
    temp = A[low];
    A[low] = A[j];
    A[j] = temp;
    return j;
}
```

Code Snippet 3: Creating the *partition* function

Here is the whole source code:

```c
#include <stdio.h>

void printArray(int *A, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}

int partition(int A[], int low, int high)
{
    int pivot = A[low];
    int i = low + 1;
    int j = high;
    int temp;

    do
    {
        while (A[i] <= pivot)
        {
            i++;
        }

        while (A[j] > pivot)
        {
            j--;
        }

        if (i < j)
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    } while (i < j);

    // Swap A[low] and A[j]
    temp = A[low];
    A[low] = A[j];
    A[j] = temp;
    return j;
}

void quickSort(int A[], int low, int high)
{
    int partitionIndex; // Index of pivot after partition

    if (low < high)
    {
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1);  // sort left subarray
        quickSort(A, partitionIndex + 1, high); // sort right subarray
    }
}

int main()
{
```

```
    //int A[] = {3, 5, 2, 13, 12, 3, 2, 13, 45};
    int A[] = {9, 4, 4, 8, 7, 5, 6};
    // 3, 5, 2, 13, 12, 3, 2, 13, 45
    // 3, 2, 2, 13i, 12, 3j, 5, 13, 45
    // 3, 2, 2, 3j, 12i, 13, 5, 13, 45 --> first call to partition returns 3
    int n = 9;
    n =7;
    printArray(A, n);
    quickSort(A, 0, n - 1);
    printArray(A, n);
    return 0;
}
```

Code Snippet 4: Program to implement the Quick Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 7.

```
    int A[] = {9, 4, 4, 8, 7, 5, 6};
    int n = 7;
    printArray(A, n);
    quickSort(A, 0, n-1);
    printArray(A, n);
```

Code Snippet 5: Using the *quickSort* function

And the output we received was:

```
9 4 4 8 7 5 6
4 4 5 6 7 8 9
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output of the above program

So, our array got sorted. Follow the dry run we did in the comments in the source code. Practice sorting small arrays using the partition method on your own, and you'll feel confident about it.

And, this was all about the quick sort algorithm. We have finished writing the program for quick sort and visualizing it at the same time. I know it was a long lecture but worth going through to understand the very basics of the quicksort algorithm to its completion. Analysis of the algorithm is still left, and we have kept it for the next lecture. Review all concepts, and contact us again if you have questions.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll see the analysis part of the Quicksort Algorithm. Till then, keep coding.