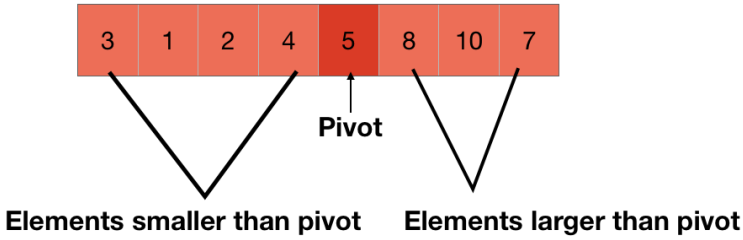
QuickSort

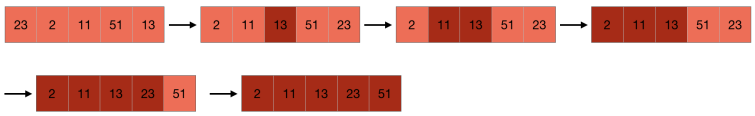
Quicksort is another sorting algorithm which uses Divide and Conquer for its implementation. Quicksort is also the practical choice of algorithm for sorting because of its good performance in the average case which is $\Theta(n \lg n)$. Unlike the Merge Sort (/course/algorithms-merge-sort/), Quicksort doesn't use any extra array in its sorting process and even if its average case is same as that of the Merge Sort (/course/algorithms-merge-sort/), the hidden factors of $\Theta(n \lg n)$ are generally smaller in the case of Quicksort.

So, Quicksort sounds like the perfect sorting algorithm we always needed, but hold on for a second, Quicksort is also not as perfect as it sounds. In the worst case, Quicksort gives us a quadratic ($O(n^2)$) performance which might be concerning in many cases. So, let's move forward and discuss in detail about Quicksort.

Quicksort first chooses a pivot and then partition the array around this pivot. In the partitioning process, all the elements smaller than the pivot are put on one side of the pivot and all the elements larger than it on the other side.



This partitioning process is repeated on the smaller subarrays and hence finally results in a sorted array.



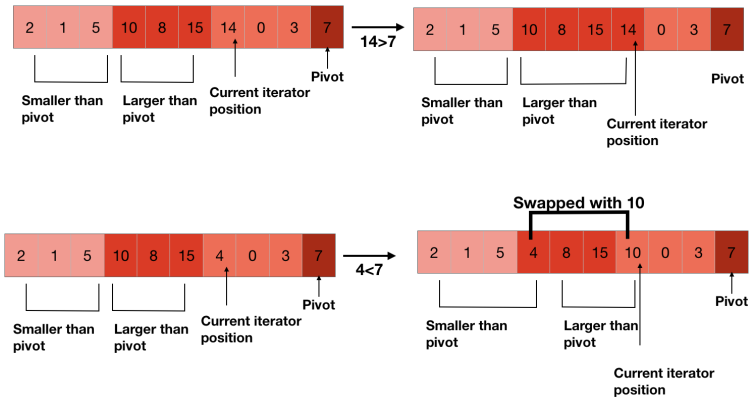
So, let's first focus on making the partition function.

Partition in Quicksort

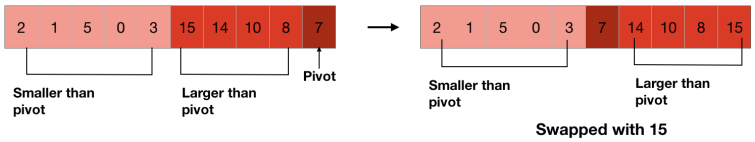
The first step of doing a partition is choosing a pivot. We are going to always select the last element of the array as the pivot in our algorithm and focus mainly on the concepts behind the Quicksort. However, there can be different ways of choosing the pivot like the median of the elements, the first element of the array, random element, etc.

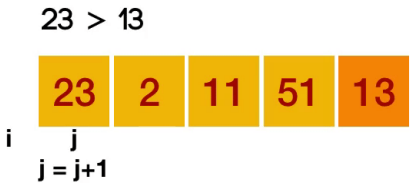
After choosing the pivot, our next task is to place all the elements smaller than the pivot on one side and all the elements larger than the pivot on another side. We will do this by iterating over the array and on finding an element larger than the pivot, doing nothing. This will start making a cluster of elements larger than the pivot.

Now on finding any element smaller than the pivot, we will swap it with the first element of the cluster of the larger elements. In this way, we will start making another cluster of elements smaller than the pivot.



At last, we will swap the pivot with the first element of the cluster of the larger elements.





Thus, we have successfully done the partition of the array. So, let's write the code for the same.

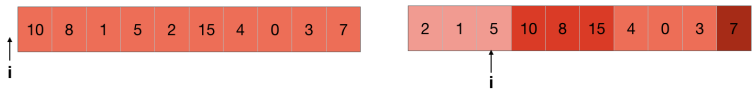
Code for Partition in Quicksort

As stated above, this partition is going to repeatedly occur on smaller subarrays after dividing the array into subarrays. So, the function for partition is going to take an array, a starting index and an ending index i.e., `PARTITION(A, start, end)`.

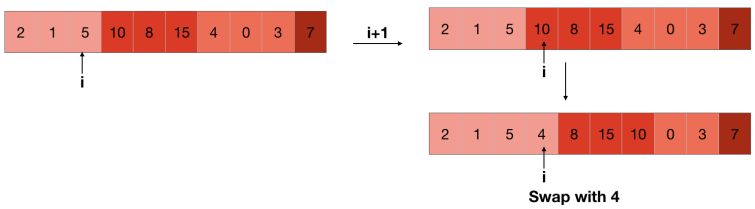
The next task is to make the pivot element. In our case, it is the last element i.e., `pivot = A[end]`.

After this, we have to iterate over the array and check if the element is smaller or larger than the pivot and then swap it with the first element of the cluster of the larger elements. So, we will use a variable to know the first element of the cluster of the larger elements.

Initially, we will start our iteration by setting this pointer outside our array i.e., `i = start-1` and this pointer will always point to the last element of the cluster of the smaller elements.



Whenever we will have a need to swap the elements, we will increase its value to point the first element of the cluster of the larger elements.




```
i = start-1
for j in start to end-1
  if A[j] <= pivot
```

Now, if the condition is true i.e., the element is smaller than the pivot, then we have to swap it with the first element of the cluster of the larger elements. So, we will increase the value of 'i' by 1 and then do the swapping.

```
for j in start to end-1
   if A[j] <= pivot
      i = i+1
      swap(A[i], A[j])
```

Thus, we are done with making the clusters and the only task left is to swap the pivot with the first element of the cluster of the larger elements.

```
for j in start to end-1
   if A[j] <= pivot
      ...
   swap(A[i+1], A[end]) // swapping pivot
```

One last thing we want from our PARTITION function is to return the index of the pivot, so we can then use this index to divide our array further into subarrays.

```
for j in start to end-1
   ...
   swap(A[i+1], A[end]) // swapping pivot
return i+1
```

That's it. We are done with the code to partition an array.

```
PARTITION(A, start, end)
   pivot = A[end]
   i = start-1

   for j in start to end-1
      if A[j] <= pivot
         i = i+1
         swap(A[i], A[j])
   swap(A[i+1], A[end]) //swapping pivot
   return i+1
```

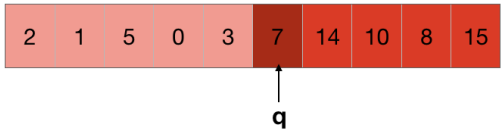
Now we are done with the PARTITION function, so let's focus on repeating it to subarrays.

Code for Quicksort

From the previous two chapters, we already have been applying divide and conquer to break the array into subarrays but we were using the middle element to do so. In quicksort, we will use the index returned by the PARTITION function to do this.

Here also, we will continue breaking the array until the size of the array becomes 1 i.e., until `start < end`.

So, we will first start by partitioning our array i.e., `q = PARTITION(A, start, end)`. 'q' is storing the index of the pivot here. After this, we will again repeat this process to the subarray from 'start' to 'q-1' and 'q+1' to 'end', leaving the pivot element because that is already in the place.



```
QUICKSORT(A, start, end)
  if start < end
     q = PARTITION(A, start, end)
     QUICKSORT(A, start, q-1)
     QUICKSORT(A, q+1, end)
```

C **Python** **Java**

```c
#include <stdio.h>

// function to swap values of two variables
void swap(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

int partition(int a[], int start, int end) {
  int pivot = a[end];
  int i = start-1;
  int j;

  for(j=start; j<=end-1; j++) {
    if(a[j]<=pivot) {
      i=i+1;
      swap(&a[i], &a[j]);
    }
  }
  swap(&a[i+1], &a[end]);
  return i+1;
}

void quicksort(int a[], int start, int end) {
  if(start < end) {
    int q = partition(a, start, end);
    quicksort(a, start, q-1);
    quicksort(a, q+1, end);
  }
}

int main() {
  int a[] = {4, 8, 1, 3, 10, 9, 2, 11, 5, 6};
  quicksort(a, 0, 9);

  //printing array
  int i;
  for(i=0; i<10; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
  return 0;
}
```

Analysis of Quicksort

We have already stated that Quicksort takes $\Theta(n^2)$ time in the worst case and $\Theta(nlgn)$ in the best case. Let's analyze the above code and confirm these running times.

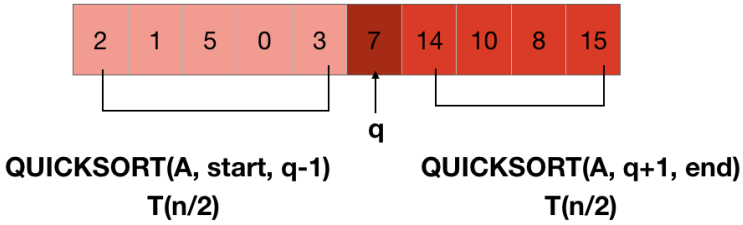
Now, the total running time of the QUICKSORT function is going to be the summation of the time taken by the `PARTITION(A, start, end)` and two recursive calls to itself. The comparison (`if start < end`) is going to take a constant amount of time and thus, we are ignoring it.

Clearly, the PARTITION function is a linear (i.e., $\Theta(n)$) one because there is just one loop which is iterating over the entire array (of size n) and all the other statements are constant time taking processes.

Let's take a case when the partition is always balanced i.e., every time, each of the two halves separated by the pivot has no more than $\frac{n}{2}$ elements. It means that one of the halves has $\lfloor \frac{n}{2} \rfloor$ and another has $\lceil \frac{n}{2} \rceil - 1$ elements. For example, if n is 5, then one half has $\lfloor \frac{5}{2} \rfloor = 2$ elements and another has

$\lceil \frac{5}{2} \rceil - 1 = 2$ elements and thus, a total of 2+2+1(pivot) = 5 elements.

In that case, `QUICKSORT(A, start, q-1)` and `QUICKSORT(A, q+1, end)` will take $T\left(\frac{n}{2}\right)$ each and the PARTITION function is going to take $\Theta(n)$ time.



$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

We have already dealt with this same equation many times in this course and know that it is going to take $\Theta(n \lg n)$ time.

Now, let's take the case when the partition is completely unbalanced every time i.e., each time, one half is of the size $n-1$ and other of size 0.

In this case, the total running time would be:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$

$$=> T(n) = T(n-1) + \Theta(n)$$

We can solve this recurrence equation by iteration method i.e., putting replacing $T(n-1)$ with $T(n-2) + \Theta(n)$ and again $T(n-2)$ with $T(n-2) + \Theta(n)$ and so on.

$$T(n) = T(n-1) + \Theta(n)$$

Replacing $T(n-1)$ with $(T(n-2) + \Theta(n))$,

$$=> T(n) = T(n-2) + \Theta(n) + \Theta(n)$$

Replacing $T(n-2)$ with $(T(n-3) + \Theta(n))$,

$$=> T(n) = T(n-3) + \Theta(n) + \Theta(n) + \Theta(n)$$

Similarly,

$$T(n) = T(n-i) + i * \Theta(n)$$

Thus, in the base case, when there will be an only single element, the running time will become

$$T(n) = T(0) + n * \Theta(n)$$

which is equal to $\Theta(n^2)$

So, we have seen that Quicksort takes $\Theta(nlgn)$ time in the balanced partition and $\Theta(n^2)$ time in the unbalanced one.

Let's take a look at the case when the partition is quite unbalanced and see the running time.

Analysis of Quicksort in Nearly Unbalanced Partition

Let's take a case when the partition is putting 99% of elements on one side and 1% of elements on another side (i.e., $\frac{99n}{100}$ and $\frac{n}{100}$) in each repetition.

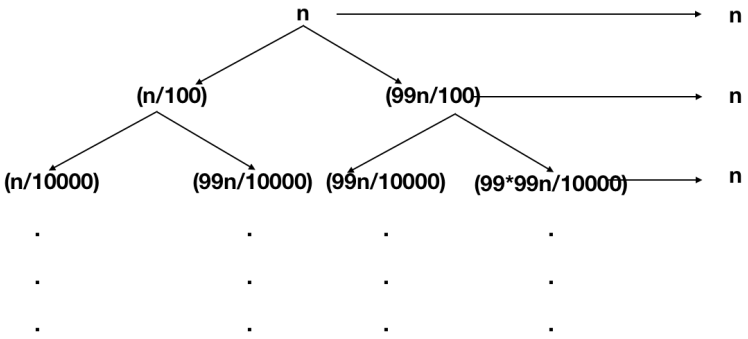
In this case, we can write the running time as:

$$T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{100}\right) + \Theta(n)$$

or,

$$T(n) = T\left(\frac{99n}{100}\right) + T\left(\frac{n}{100}\right) + cn$$

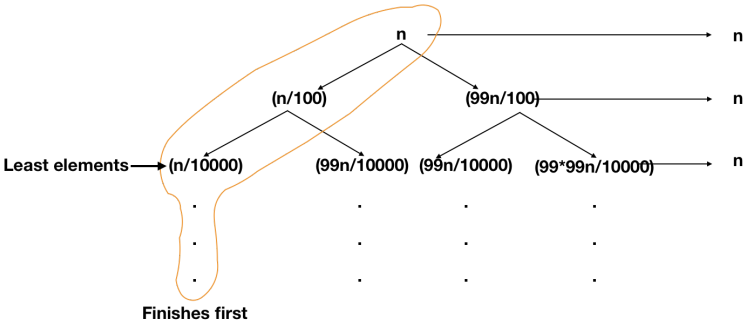
Let's take a look at the recursion tree of the above equation.



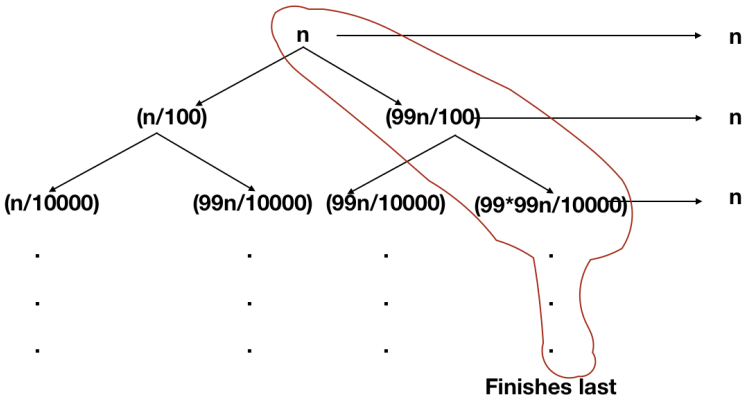
</>

The actual division should be $\frac{(n-1)}{100}$ and $\frac{99(n-1)}{100}$ instead of $\frac{n}{100}$ and $\frac{99n}{100}$ respectively because the pivot element is eliminated from the partition each time. But assuming $\frac{n}{100}$ and $\frac{99n}{100}$ will make the mathematics clean and we still are able to get the running time correctly.

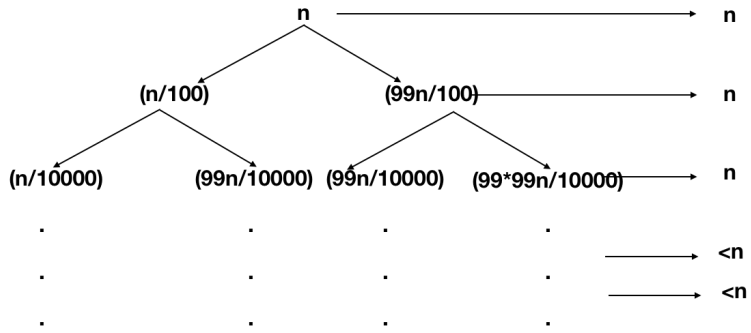
We can see that initially, the cost is cn for all levels. This will follow until the left-most branch of the tree reaches its base case (size of 1) because the left-most branch has least elements in each division, so it will finish first.



Similarly, the right-most branch will reach its base case at last because it has the maximum number of elements in each division.



Also, after the finishing of the left-most branch, the cost at each level will become less than cn as shown in the above picture.



From the above picture, we can see that at level i, the right-most node has $n * \left(\frac{99}{100}\right)^i$ elements. For the last level,

$$n * \left(\frac{99}{100}\right)^i = 1$$

$$=> i = \log_{\frac{100}{99}} n$$

So, there are a total of $\left(\log_{\frac{100}{99}} n\right) + 1$ levels.

Thus,

$$T(n) = \left(\underbrace{cn + cn + cn + \ldots + cn + (< cn) + (< cn) + \ldots + (< cn)}_{\left(\log_{\frac{100}{99}} n\right) + 1 \text{ times}} \right) < \left(\left(\left(\log_{\frac{100}{99}} n\right) + 1 \right) * cn \right)$$

$$= O(n \log_{\frac{100}{99}} n)$$

Also, there is a property of the log:

$$\log_a n = \frac{\log_b n}{\log_b a}$$

for positive values of a, b and n.

Thus,

$$\log_{\frac{100}{99}} n = \frac{\log_2 n}{\log_2 \frac{100}{99}}$$

Ignoring the constant term $\left(\log_2 \frac{100}{99}\right)$ and using this value in $O(n \log_{\frac{100}{99}} n)$, we can write,

$$T(n) = O(n \log n)$$

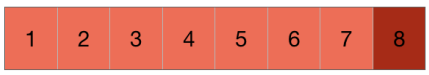
Also, the best case of Quicksort is $\Theta(n \lg n)$, so there can't exist any case for which the running time can become better than $n \lg n$. Thus, the above running time $O(n \lg n)$ can be written as $\Theta(n \lg n)$.

So, we have seen that even in the case of extremely unbalanced partition, the Quicksort results into $\Theta(n \lg n)$ running time.

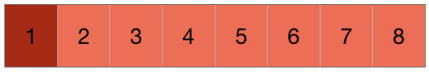
Choice of Pivot in Quicksort

We have seen the analysis of the Quicksort algorithm and we know that the running time can go up to $\Theta(n^2)$ in the worst case. Also, the partition is an important part of the algorithm and the running time depends directly on the partition which is being done in the algorithm and we can make the correct partition by choosing the right pivot.

In this chapter, for the sake of explaining the concepts, we have always taken the last element of the array as the pivot but think about the case when the input array is sorted or reverse-sorted. In that case, choosing the last element every time will result in partition of sizes n-1 and 0 and thus, will give us a quadratic running time i.e., $\Theta(n^2)$. This will also happen when we will always choose the first element of the array as the pivot.



Elements smaller than pivot = 7
Elements larger than pivot = 0

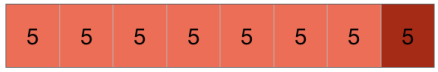


Elements smaller than pivot = 0
Elements larger than pivot = 7

Of course, we have discussed that the probability of occurrence of the worst case is very less and we are not concerned with this. The common problem is the DoS attack when an attacker makes the server resource unavailable for the intended users. So, an attacker can intentionally give such input to the server which will take $\Theta(n^2)$ time and thus make the server busy. The common solution is to pick a random pivot which will make the algorithm fair any input.

One can also think about choosing the median as the pivot but in that case, we should take care of the method of choosing the median for large data.

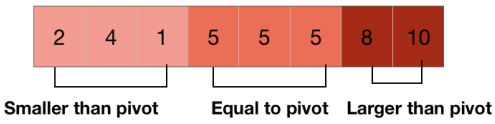
One of the common concerns with the Quicksort is also the repetition of the same element. Take a case when all the elements of the input array are equal, then all the elements will come on one side of the pivot and no element will be on another side because no element will be smaller than the pivot.



Elements smaller than or equal to pivot = 7
Elements larger than pivot = 0

This is also the case when one side has n-1 elements and another side has 0 element and thus will take $\Theta(n^2)$ time. We can overcome this problem by choosing 2 pivots and make 3 partitions of the array i.e., one smaller than the pivot, one equal to it and another for elements larger than it.

Choosing 2 pivots also increases the performance of the algorithm. Also, Dual-Pivot Quicksort is the default sorting algorithm in Java because it gives $\Theta(n \lg n)$ running time for many inputs for which the normal Quicksort goes to $\Theta(n^2)$ time.



Of course, there is a lot of stuff related to the quicksort which we can't cover alone in this chapter. You can explore the articles on BlogsDope (https://www.codesdope.com/blog/) for extra topics. Also, you can contribute by writing articles on BlogsDope (https://www.codesdope.com/blog/submit-article/) and share your knowledge with the world.

> The best way to predict the future is to implement it.
>
> - David H. Hansson