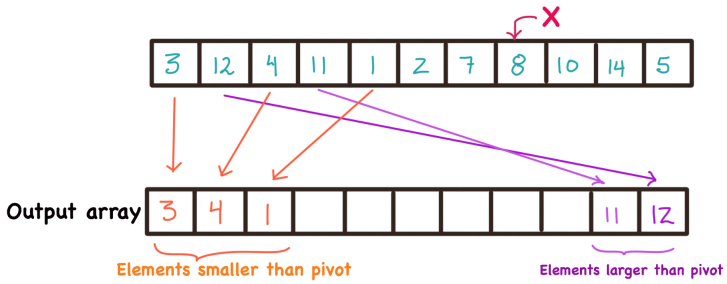
# Quicksort

In this lecture we look at one of the most practical sorting algorithms - *Quicksort*. This sorting algorithm performs very well on average: its expected running time is $O(n \log n)$. It does however have a worst-case runtime of $O(n^2)$, but this occurs extremely rarely. One of the main advantages of Quicksort is that in runs in place, meaning the entire sorting algorithm takes place within the original array.

## 1 Partitioning

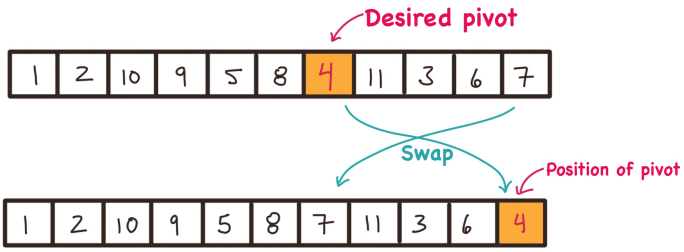
We begin by formally describing the algorithm that *partitions* the elements of an array around the pivot $x$. Such a procedure was already discussed in the section on Randomized-Select and Select. That partitioning algorithm uses an *output* array, $B$, and partitions the elements by examining them one at a time, and organizing them in the output array. The example below uses pivot $x = 8$:



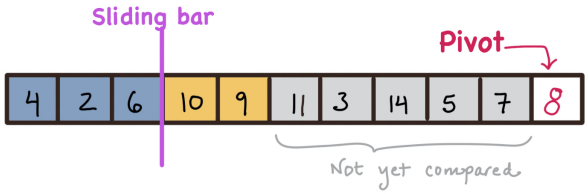

This approach runs in time $\Theta(n)$, since it examines each element once as it places it in the correct position in the output array. However, it requires an external array of size $n$. In the next section, we look at how the partitioning operation can be done without using an external array.

### 1.1 How to partition in place:

In this section, we describe how the partitioning procedure can be done *in place*. This means that we do not need a separate output array, and instead reorganize the array elements *inside* the original array. We begin by selecting a random pivot, which is done by simply selecting a random index between the start and finish indices of the array. Next we **swap** the pivot into the last position of the array.



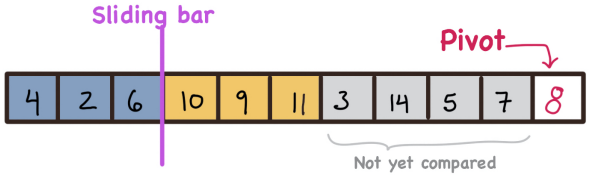

Let's look now at how the algorithm runs. Instead of organizing the elements on the left and the right of the array as shown above, we are going to organize the elements to the left and right of a *"sliding bar"*. As we examine the elements one by one in the array, we place them either to the right or left of this bar, depending on whether or not they are bigger or smaller than the pivot. The array is divided into *three sections* during the processing:



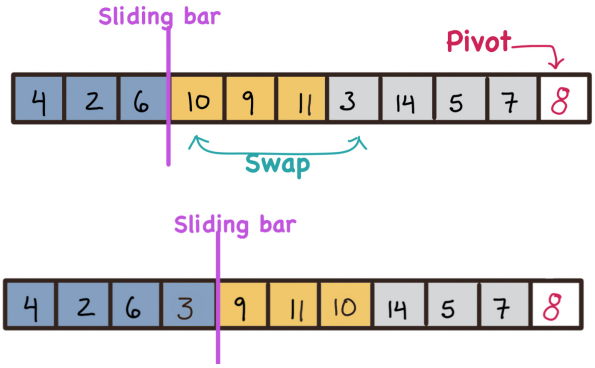

- Elements that are to the left of the bar (blue) are those that are smaller than the pivot.
- Elements that are in the section to the right of the bar (yellow) are those that are greater than the pivot.
- Elements in grey have not yet been compared to the pivot.

Let's look at how the algorithm carries out one step. The next element to be compared to the pivot is the next grey element in the array - in the above example that would be an 11. There are two cases that can arise. Either the next element is *larger* than the pivot, or the next element is *smaller* than the pivot.

- **The next element is larger then the pivot**: This is the case that is demonstrated in our above example: the 11 is larger than the pivot. Then it simply needs to be included in the yellow section. The sliding bar is *not* moved in this case, instead, we just include the next grey element in the yellow section:

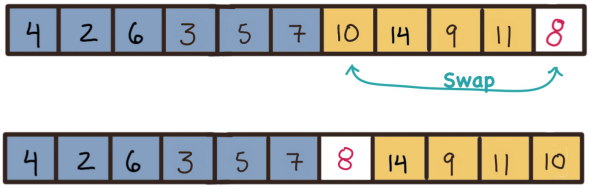



- **The next element is smaller than the pivot:** The element 3 is next in line, and it is smaller than the pivot. So it should go into the blue section. Instead of shifting all the elements over to make place to insert 3 into the blue section, we simply swap the first element in the yellow section with 3, and move the sliding bar to the right by one:




The result is that the yellow section has stayed the same size, but has shifted by one. The blue section has grown by one.

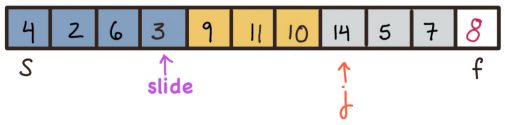
When the last element of the array has been processed, it remains only to place the pivot between the yellow and blue sections. By simply swapping the pivot with the first element of yellow section, the partitioning procedure is complete:



### The Partition Algorithm:

We now look at the details of the algorithm. The Partition algorithm takes as input array $A$ and the start and finish indices: $s$ and $f$. The algorithm works as illustrated above, using the following indices:

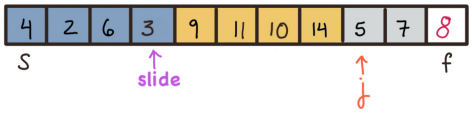
- Index **j** marks the first element of the grey section (the unprocessed items).
- index **slide** marks the last element of the blue section.



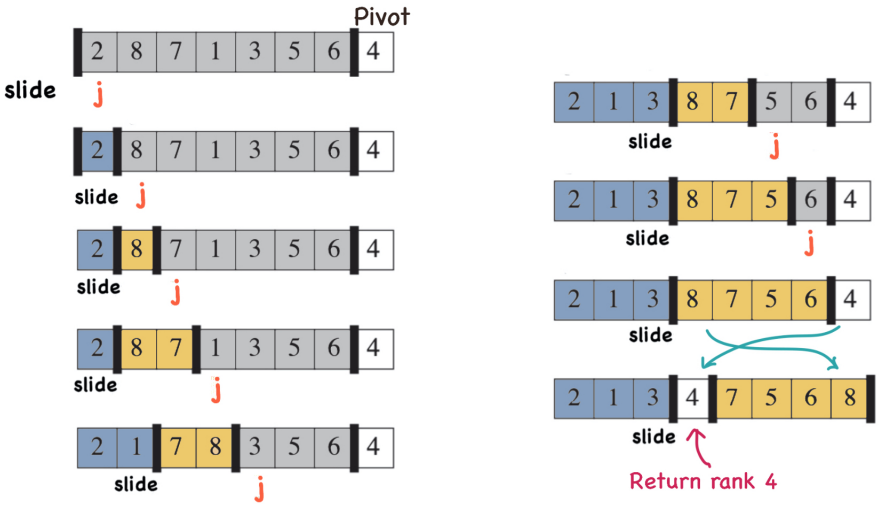

PARTITION(A, s,f)

```
p = Random(s,f)
Swap A[p] and A[f]
slide =s-1
for j = s to f-1
       if A[j] < A[f]
          slide++
          swap A[j] with A[slide]
swap A[slide+1]  with A[f]
return slide+1
```

In the example above, the value of $A[j] = 14$, which is larger than the pivot. Therefore the variable *slide* is unchanged and $j$ is increased. The result is the following array:




At the start of the next loop, $j$ will be increased by one. One important aspect of this algorithm is that **returns the index of the pivot**. The example below (from CLRS) shows the execution of *Partition* on array $A$ indexed from 1 to 11. Note that in the final step, the pivot in swapped into its correct position, and the rank of the pivot is returned:




Runtime:

This algorithm loops over the array only one time, performing a constant number of steps at each iteration. Thus the overall runtime is $\Theta(n)$.

We can summarize the important aspects of this algorithm as:

PARTITION(A,s,f)

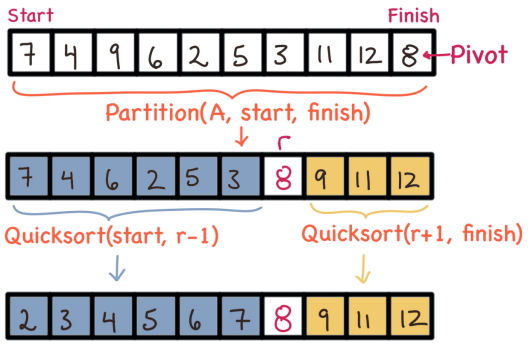
- Partitions the elements between index $s$ and index $f$ around a randomly selected pivot.
- Runs in time $\Theta(n)$
- Returns the **index** of the pivot
- Runs in place

## 2 QuickSort

Quicksort is a divide-and-conquer algorithm, like Mergesort. There are fundamental differences between the two algorihtms in how they carry out the divide and conquer steps. Mergesort divides the array in *half*. We shall see that Quicksort on the other hand divides the array into two subarrays of *random sizes*. Secondly, Mergesort does most of its"work" in the Merge step *after* the recursive calls are complete. Quicksort on the other hand performs a Partition step *before* the recursive calls.

The divide-and-conquer steps for Quicksort work as follows:

- **Divide**: We call $Partition(A, s, f)$, which returns the index $r$ of the pivot. The result is an array that is partitioned into two subarrays, with the pivot separating the two.
- **Conquer**: Call Quicksort recursively on the left and right subarrays, using the index $r$ from the Divide step above:




Note that there is no third step like in Mergesort. Once each side of the array is sorted in place - the final array will be sorted.

### The Algorithm:

The Quicksort algorithm below sorts the array between the positions $s$ and $f$. Recall that the Partition algorithm we describe above actually returns the **index** of the pivot, allowing us to define the subarrays passed to Quicksort. The entire algorithm sorts the values *in-place*. Note that we begin by checking if $s = f$, in which case the array has size one, and we simply return without doing anything.

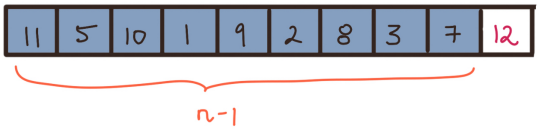
QUICKSORT(A,s,f)

```
if s >= f return
r = PARTITION(A,s,f)
QUICKSORT(A,s,r-1)
QUICKSORT(A,r+1,f)
```

We now look at the runtime, $T(n)$ of Quicksort.

## 2.1 Worst-case runtime

Quicksort could be extremely inefficient if the array is partitioned in a very unbalanced way. Suppose the partition step of Quicksort *always selected* the maximum or the minimum element as the pivot. In this case, the recursive calls to Quicksort, would be on an array of size $n-1$ and on an array of size 0.




Since the partition step runs in time $\Theta(n)$, the overall runtime of this scenario is

$$T(n) = T(n-1) + cn$$

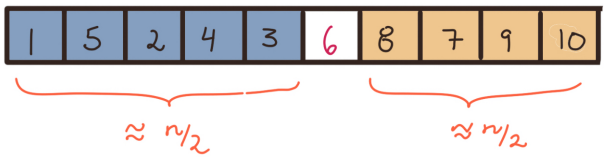
which has a solution which we have seen many times now:

$$T(n) = c(1 + 2 + 3 + \ldots + n) = \Theta(n^2)$$

Unfortunately in the worst-case the runtime of Quicksort is that of Insertion sort.

## 2.2 Best-case runtime

Suppose that the PARTITION algorithm *always picked* a pivot that was in fact the *median* of the elements. Then Quicksort would be called on two arrays, each of size approximately $n/2$:




The recurrence for the runtime of this best-case situation is then:

$$T(n) = 2T(n/2) + \Theta(n)$$

which has solution $T(n) = \Theta(n \log n)$ by Master Method. This is of course an ideal situation, and we cannot simply assume that the pivot is indeed the median at every stage of Quicksort.

In the next section we look at how to "*average out*" the good and bad cases. We shall show that if we have enough splits that are similar to the best-case scenario, then the overall runtime of Quicksort is indeed $O(n \log n)$.

## 2.3 Expected run-time

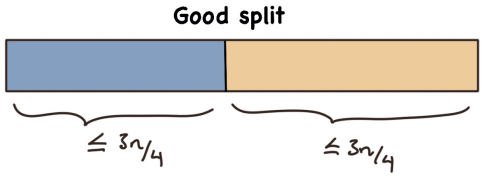
When the pivots are selected randomly, the result is that the actual runtime of the algorithm is a **random variable**: its exact value depends on the particular pivots chosen during execution. Each runtime is associated with a particular "chance" or probability of happening.

We saw in the previous section that bad choices for a pivot are possible, resulting in a very slow runtime. In fact, we showed that continuously selected the maximum as the pivot results in a runtime of $O(n^2)$. The good news is that we only have a very small chance of performing bad splits over and over. Let's determine the chance of this actually happening, in the specific case where $n = 5$. The chance of selecting the maximum of 5 elements as the first pivot is only $1/5$. The result is that the left subarray has size 4 after the partition. The chance that again we pick the maximum as the pivot is $1/4$, etc. Therefore over all executions of Quicksort, the chance that we select the maximum as the pivot *every* time is $\frac{1}{5} \cdot \frac{1}{4} \cdot \frac{1}{3} \cdot \frac{1}{2} = 0.0083$, which is 0.8%, extremely unlikely!

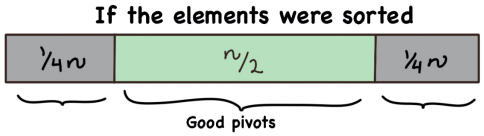

This example illustrates that the worst-case example for $n = 5$ elements is quite rare.

In this section we analyze the **Expected runtime**, which is what we "expect" that runtime to be. The intuition behind the analysis is much like that of *Randomized-Select* from our previous lecture. We consider two types of splits:

- A **good split** is when the array is split into left and right subarrays, each having size **less than** $3n/4$.




  This happens when the selected pivot has **rank** in the *"middle"* half of the elements:




  As we saw with Randomized-Select, the chance of picking a **good pivot** is $1/2$, since exactly half the elements in the array lie in the middle sorted section.

- A **bad split** is when then array is partitioned in such a way that one of the subarrays has size *greater* than $3n/4$. This happens when the pivot is selected from one of the two grey regions above. The chance of picking a bad pivot, and thus making a bad split is also $1/2$.

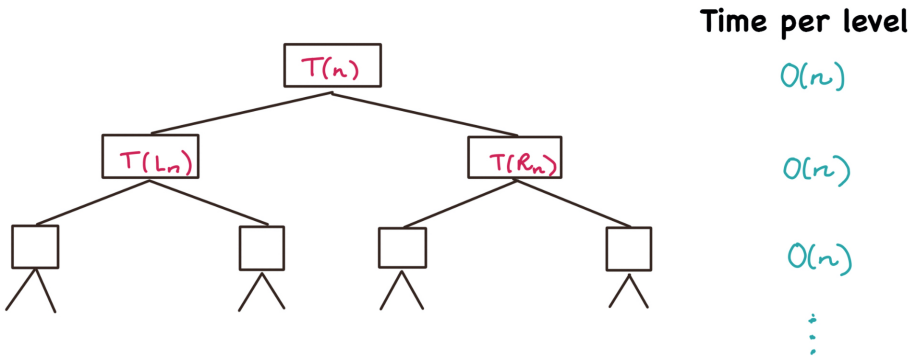
We use the following fact (also seen in Randomized-Select) regarding the expected number of good splits:

**Since the probability of a good split is $1/2$, we expect every second split to be a good split**

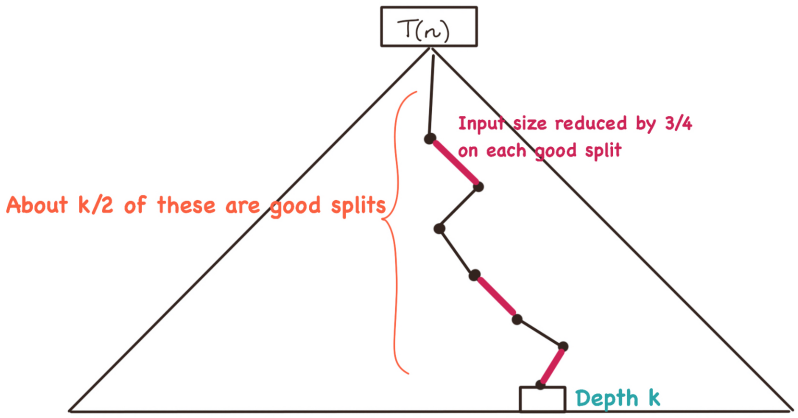
Let's consider the recursion tree for this randomized version of Quicksort. The runtime for an input of size $n$ is:

$$T(n) = T(L_n) + T(R_n) + \Theta(n)$$

where $L_n$ and $R_n$ are the sizes of the left and right subarrays produced by the partition. Note that these sizes that are **random** because they depend on which pivot we chose.




The *time per level* of the recursion tree above is $O(n)$, since the total of $L_n$ and $R_n$ at any level is $O(n)$. We now need to analyze the height of this tree. Since about every second partition results in a *good* split, this means that we expect half the splits on the path to a node of depth $k$ to be good.




If *each* branch of the tree reduced the input by 3/4, the height of the tree would be $\log_{4/3} n$. But this is expected to happen only every *second* branch. So the expected height is approximately $2\log_{4/3} n$, which is $O(\log n)$. Since the runtime per level is $O(n)$, then the expected runtime of Quicksort is $O(n \log n)$.

**Summary of the above analysis:**

- The chance of making a **good** split is 1/2
- This means about every second split in the tree is a good split
- The good splits reduce the input size by 3/4 in the worst case.
- The tree height is expected to be $O(\log n)$.
- The runtime of each level in the tree is $O(n)$
- The overall expected runtime is $O(n \log n)$

Interestingly, this means that we can expect Quicksort to behave much like the best-case analysis - both of them result in a running time of $O(n \log n)$, the only difference is that there is a slightly higher constant hidden behind the big-Oh.

We finish this section by noting that this is not a rigorous analysis of Quicksort. You can find the expected runtime analysis in CLRS 7.4.2, where the result comes from the linearity of expectation.