QUICKSORT

TABLE OF CONTENTS

What Is Quicksort?

Quicksort is a recursive sorting routine that works by partitioning the array so that items with smaller keys are separated from those with larger keys and recursively applying itself to the two groups.

Advantages of Quicksort

- Its average-case time complexity to sort an array of *n* elements is O(*n* lg *n*).
- On the average it runs very fast, even faster than Merge Sort.
- It requires no additional memory.

Disadvantages of Quicksort

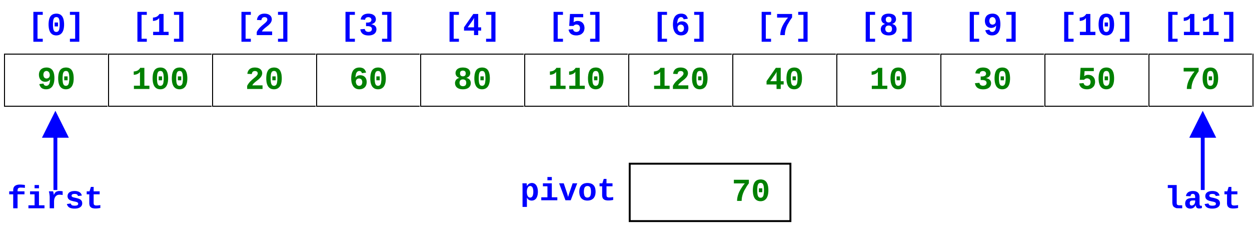
- Its running time can differ depending on the contents of the array.
- Quicksort's running time degrades if given an array that is almost sorted (or almost reverse sorted). Its worst-case running time, O(n^2) to sort an array of *n* elements, happens when given a sorted array.
- It is not stable.

Partition Algorithm (pages 325-333)

Most of the work of Quicksort is done in the Partition algorithm so we study it first. There is a Partition Workshop applet available to help you visualize this algorithm.

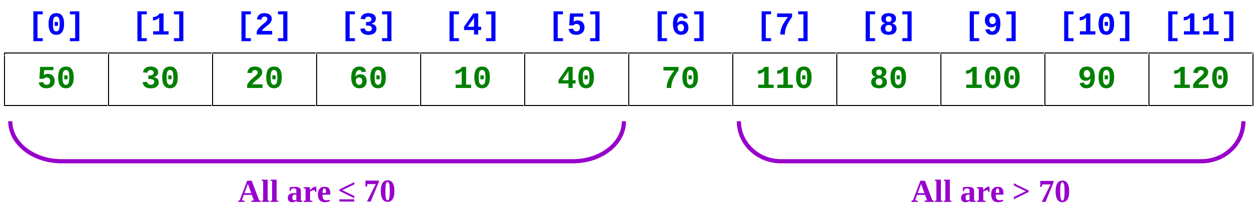
Partition considers the array elements **`a[first]`** through **`a[last]`**. Given a **`pivot`** value, it partitions the elements in the array so that all items with keys less than the pivot are moved to lower positions in the array and all items with keys greater than the pivot are moved to higher positions.

For example, given the array:



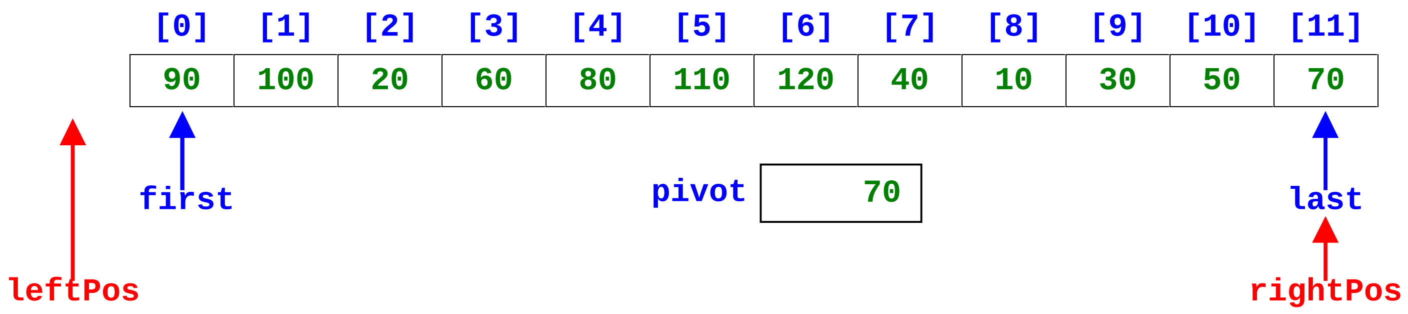
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
90	100	20	60	80	110	120	40	10	30	50	70

The Partition algorithm yields:

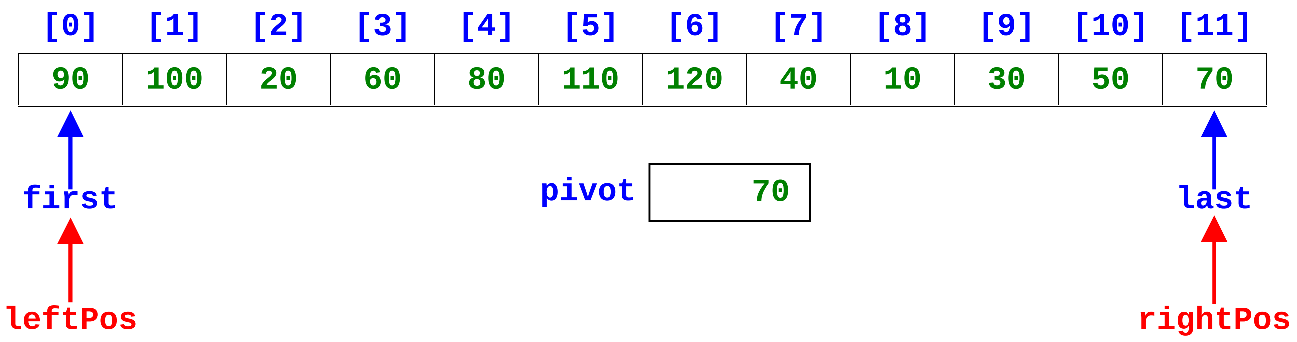


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
50	30	20	60	10	40	70	110	80	100	90	120

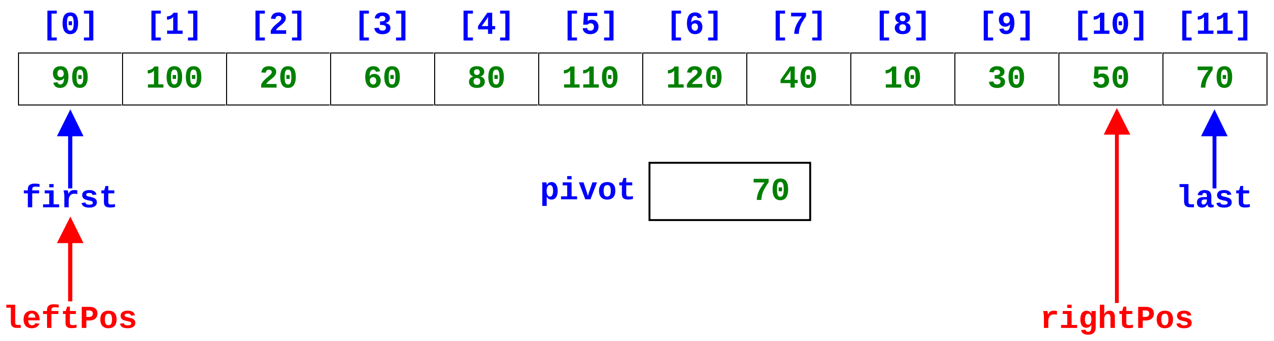
In the Partition workshop applet, the pivot value is arbitrary. The Quicksort algorithm, however, uses `a[last]` as the pivot value. To accomplish the partitioning, Partition uses two variables, `leftPos` and `rightPos`, initialized to the position below `first` and `last`, respectively. The initial setup is:



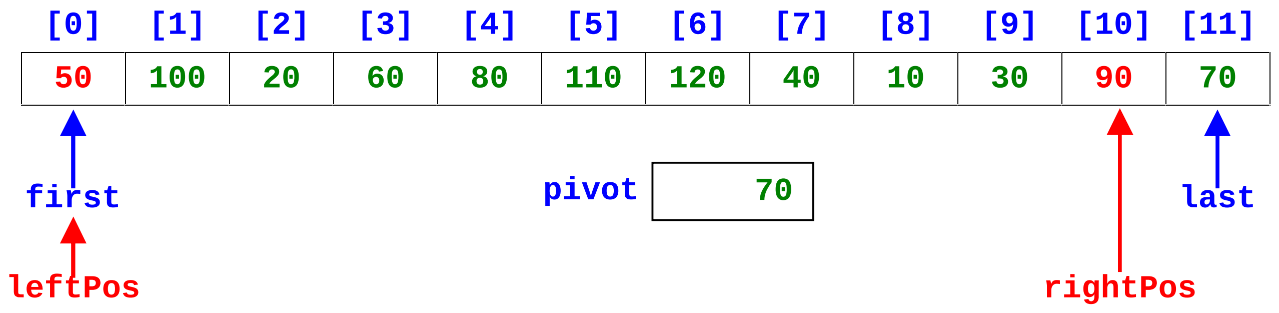
The algorithm increments `leftPos` until the item at that position is greater than or equal to the pivot value:



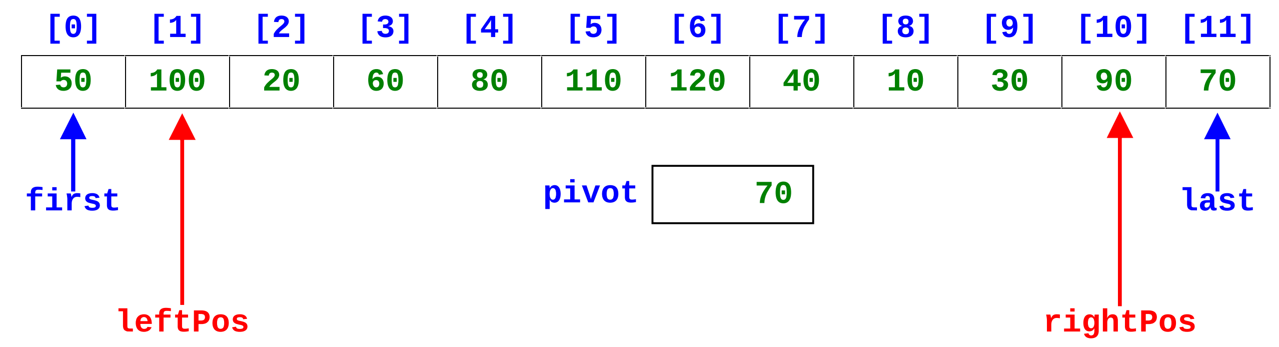
It decrements `rightPos` until the item at that position is less than or equal to the pivot value:



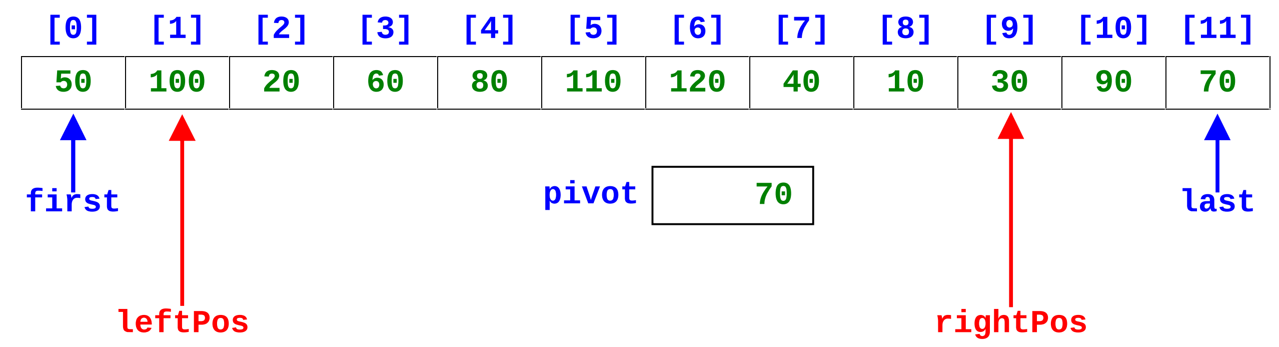
Then it swaps the values at those two positions:



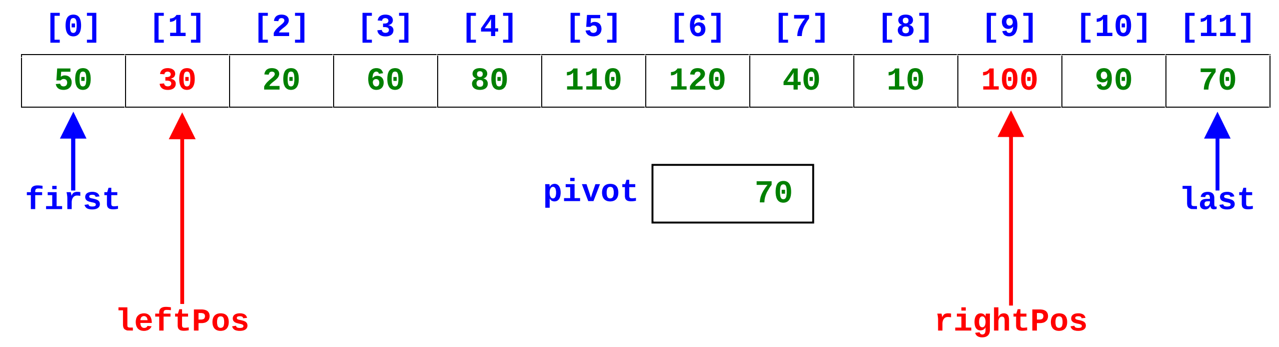
Again, increment `leftPos` until `a[leftPos] >= pivot`:



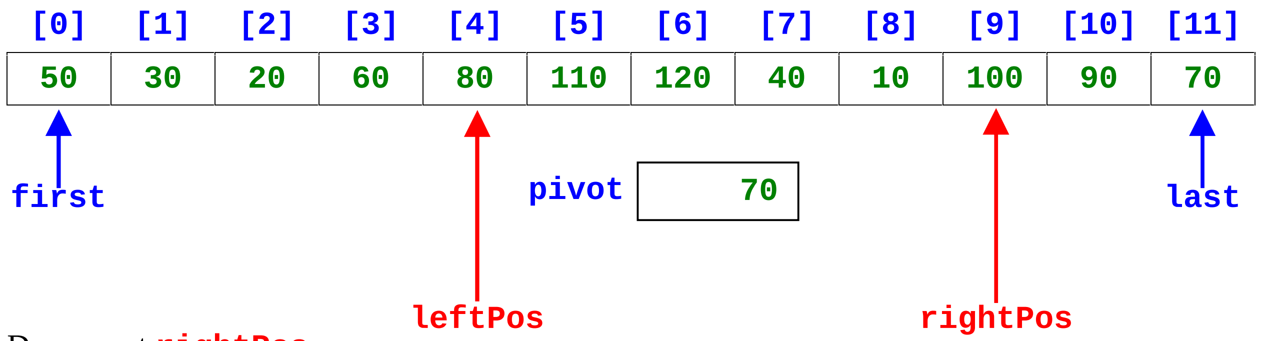
Decrement `rightPos` until `a[rightPos] <= pivot`:



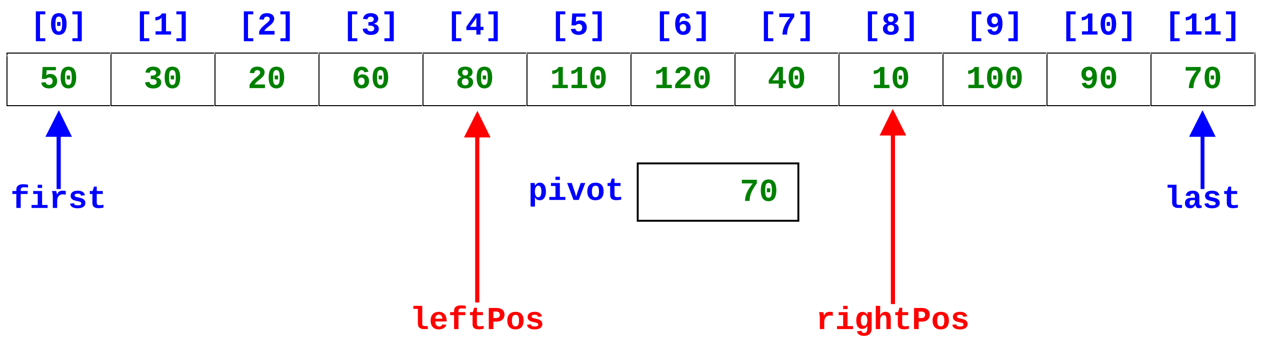
And swap:



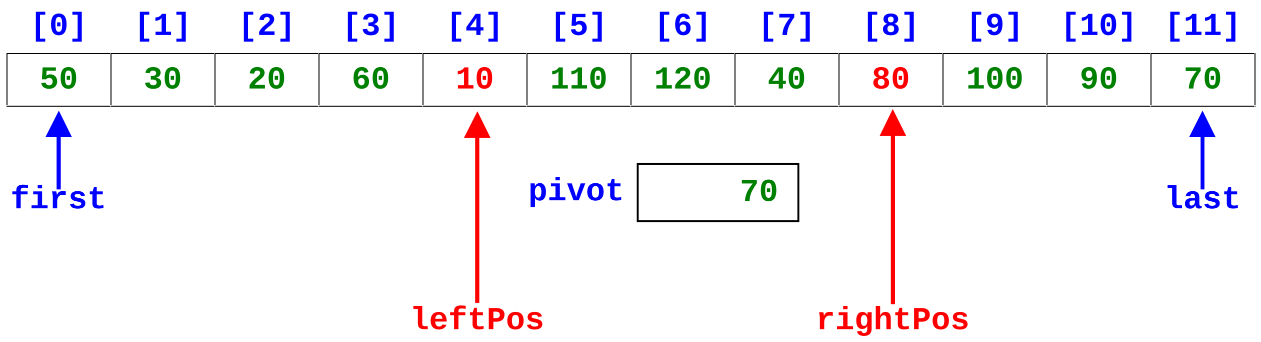
Continuing, increment **leftPos**:



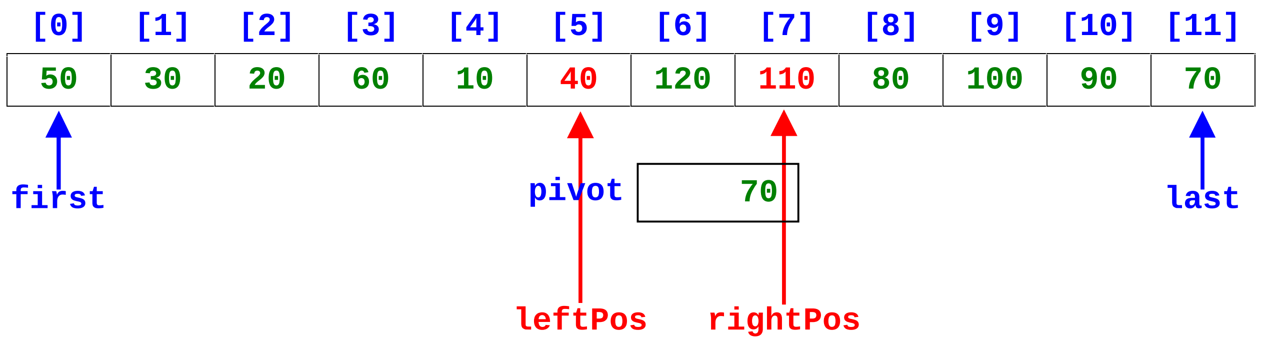
Decrement **rightPos**:



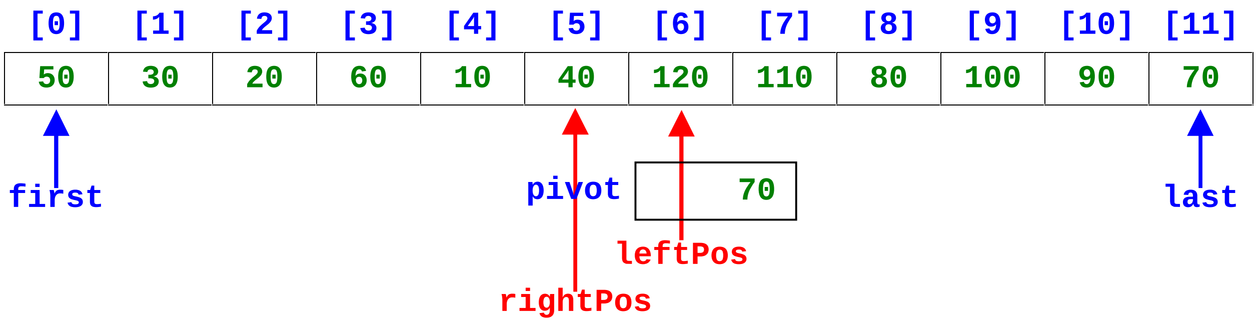
And swap:



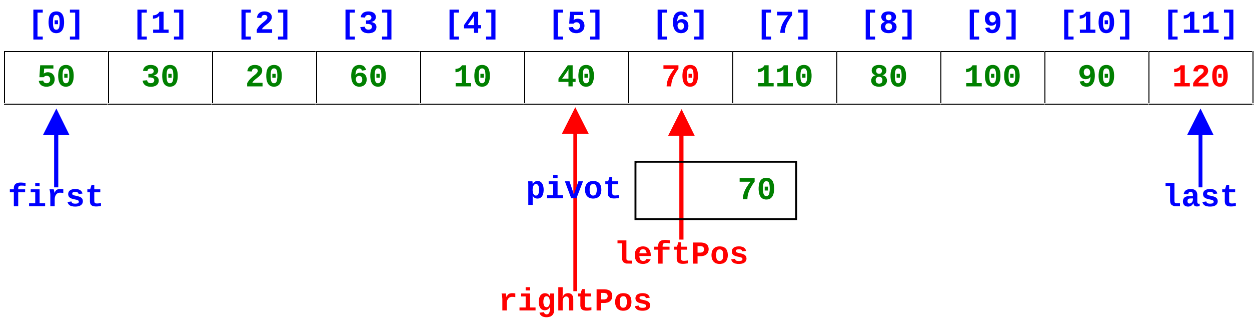
Continuing through the next swap:



Eventually this stops when the two variables cross paths:



The Partition algorithm finishes by swapping the pivot at position `last` with the item at `leftPos`:



The algorithm:

```
public int partition( int first, int last )
{
   leftPos = first;
   rightPos = last + 1;
   pivot = a[last];

   while ( leftPos < rightPos )
   {
      increment leftPos until a[leftPos] >= pivot;
      decrement rightPos until a[rightPos] <= pivot;
      if (leftPos < rightPos)
         swap a[leftPos] and a[rightPos]
   }
   swap a[leftPos] and a[last];
   return leftPos;
}
```

Quicksort Algorithm

Quicksort starts by partitioning the array. For the example above, the result of the first partition is:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
50	30	20	60	10	40	70	110	80	100	90	120

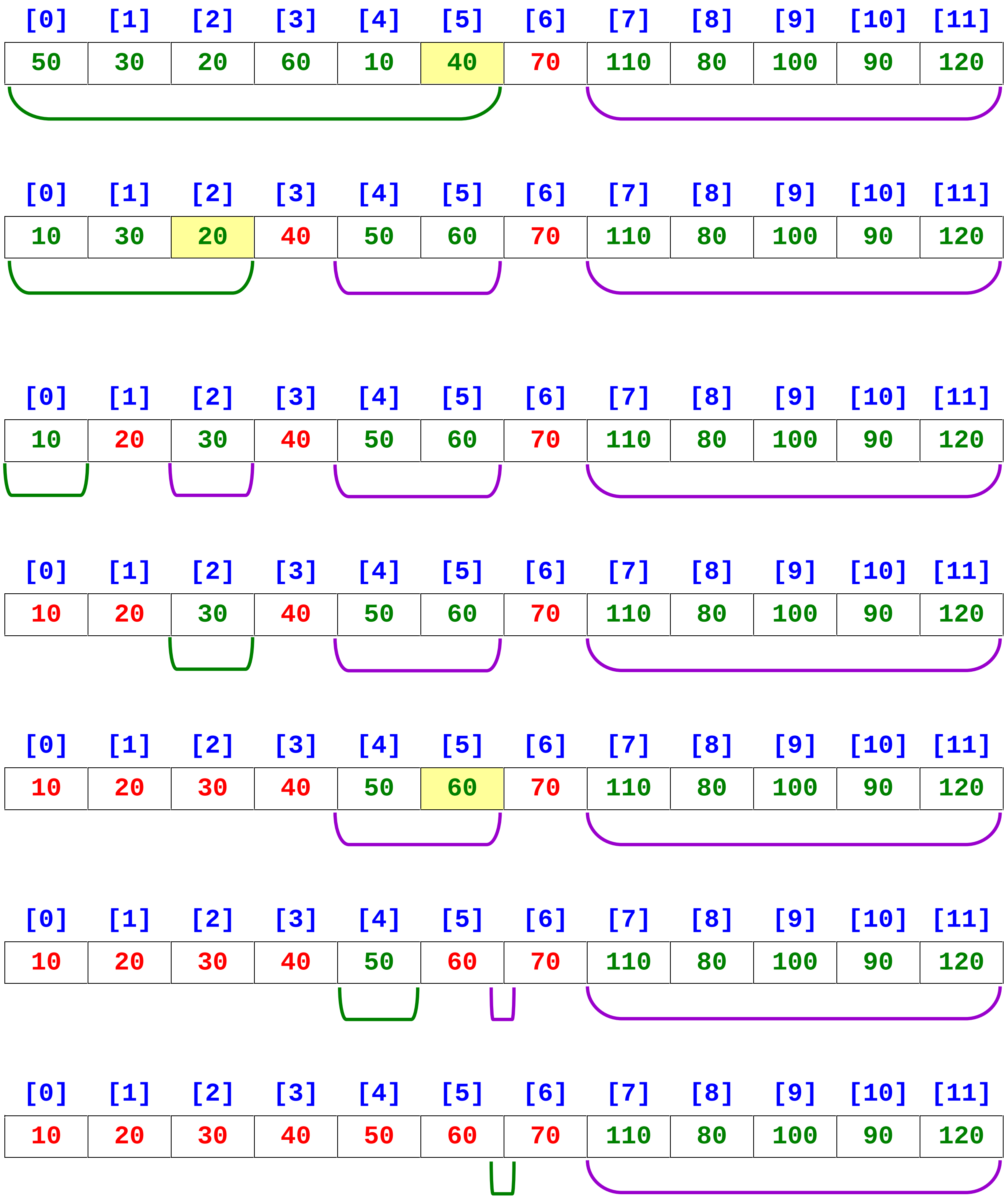
$\underbrace{[0] \ldots [5]}_{\text{All are} \leq 70}$ $\underbrace{[7] \ldots [11]}_{\text{All are} > 70}$

You can see that the pivot value, 70, holds the same position it would if the array were sorted. All items to the left of it are smaller and all items to the right of it are larger. Thus, in terms of sorting, the 70 is in the right spot and need not be considered further. Quicksort can continue by simply applying itself recursively to the two segments of the array above and below the pivot. Here's the algorithm; notice that the Partition algorithm returns the final position of the pivot (6 in the above example).

```
public void recquicksort( int first, int last )
{
   if ( last - first + 1 <= SEGMENT_SIZE )
      return;
   else
   {
      int p = partition( first, last );
      recquicksort( first, pivot-1 );
      recquicksort( pivot+1, last );
   }
}
```

`SEGMENT_SIZE` is a named constant. The computation `last - first + 1` yields the number of elements in the array segment from `first` to `last`. The `if` statement stops the recursion if the segment size is smaller than the named constant. The purpose of this is explained later; for now, just consider `SEGMENT_SIZE` to be equal to 1 since no sorting is necessary for one or fewer items.

We will continue tracing the Quicksort on the data given above. This is the same data as shown in Figure 7.12, but that figure is incorrect. Following is the correct trace for that data.



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
50	30	20	60	10	40	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	30	20	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	110	80	100	90	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	80	90	100	110	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	80	90	100	110	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	80	90	100	110	120

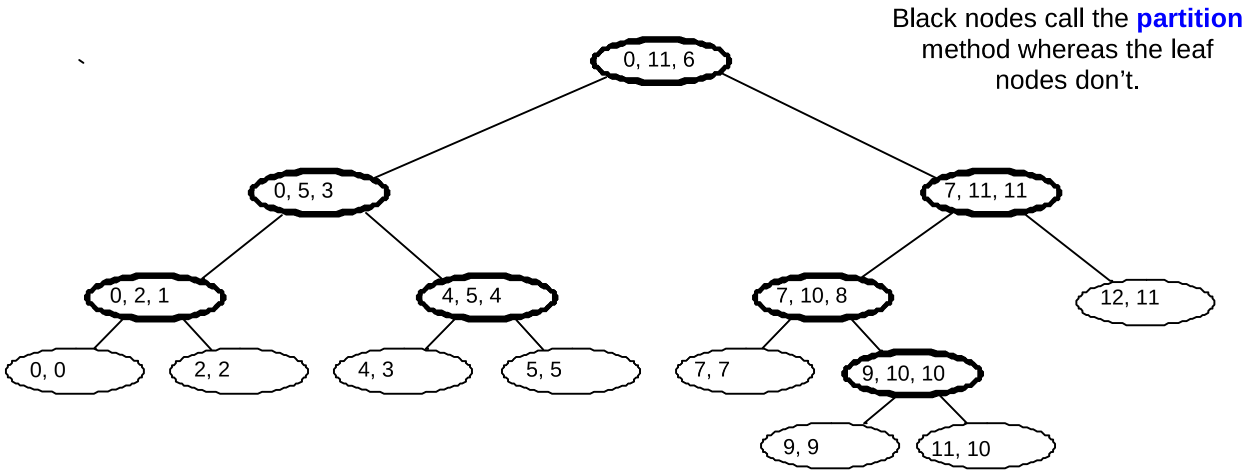
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	80	90	100	110	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	80	90	100	110	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	80	90	100	110	120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
10	20	30	40	50	60	70	80	90	100	110	120

Following is the recursion tree showing the execution of the rec**quickSort** algorithm for the above example. The three values listed are those of **first**, **last** and the position where the call to **partition** leaves the pivot value.



Improved Partitioning (pages 344—350)

To avoid the worst-case scenario, several authors have suggested different ways to select the pivot value.

1. An easy fix is to choose the pivot value randomly instead of always choosing the item at `a[last]`. You can accomplish this by replacing the statement:

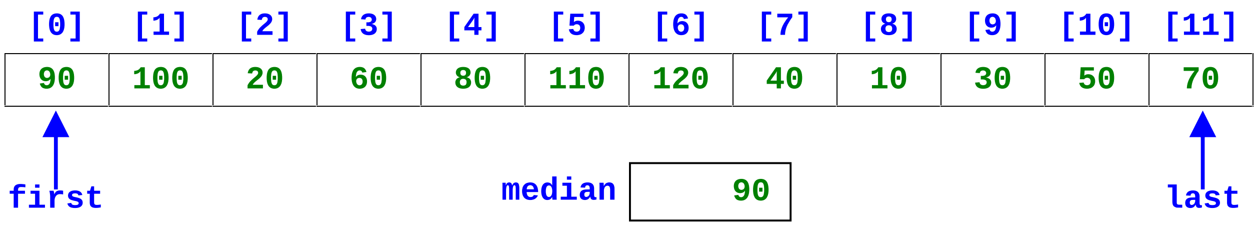
```
   double pivot = a[last];
```

with:

```
   int pos = (int)Math.random(first-last+1)+last;
   double pivot = a[pos];
```

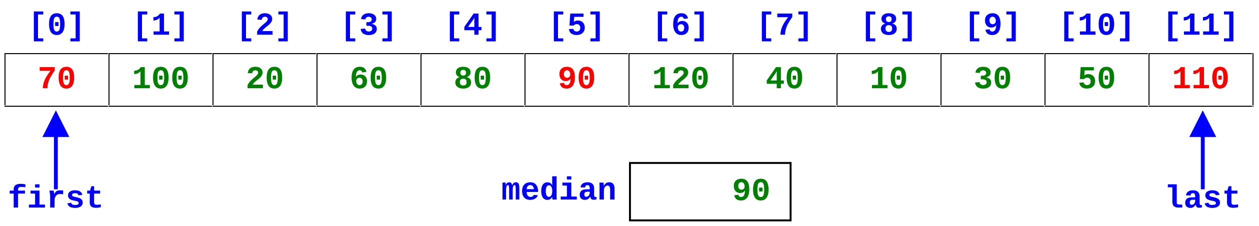
Weiss[1] considers this effort to be pointless. The computation of the random number adds a nontrivial amount of run time to the algorithm and doesn't guarantee that the worst-case is avoided. Through sheer bad luck, the random choices of pivot values may be those that lead to the worst-case behavior. Furthermore, it doesn't reduce the running time of the rest of the algorithm.

2. A technique that does work at the expense of some additional run time is the ***median-of-three technique*** covered on pages 345 through 350. The pivot is chosen to be the median of the first, last and middle elements of the array. In terms of the algorithm, choose the median of `a[first]`, `a[last]` and `a[first+last/2]`. For example, the median for the following array is 90:



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
90	100	20	60	80	110	120	40	10	30	50	70

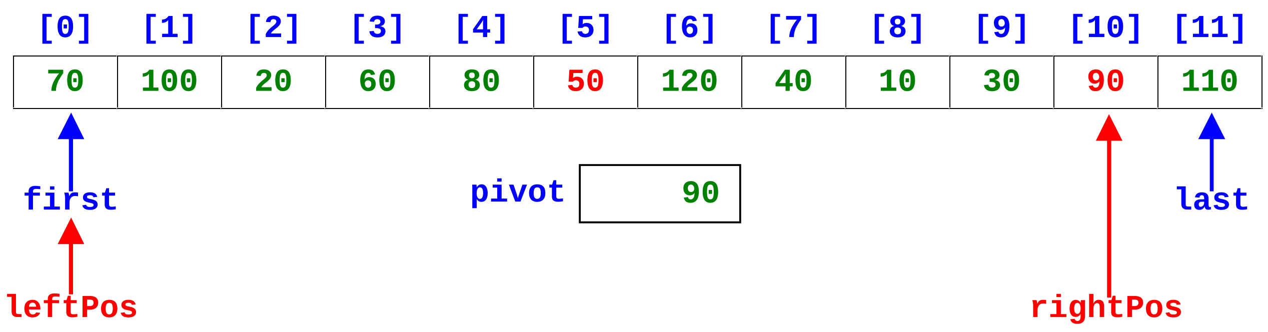
The implementation of the median-of-three partitioning is on page 348. The algorithm chooses the pivot by putting `a[first]`, `a[last]` and `a[first+last/2]` into ascending order:



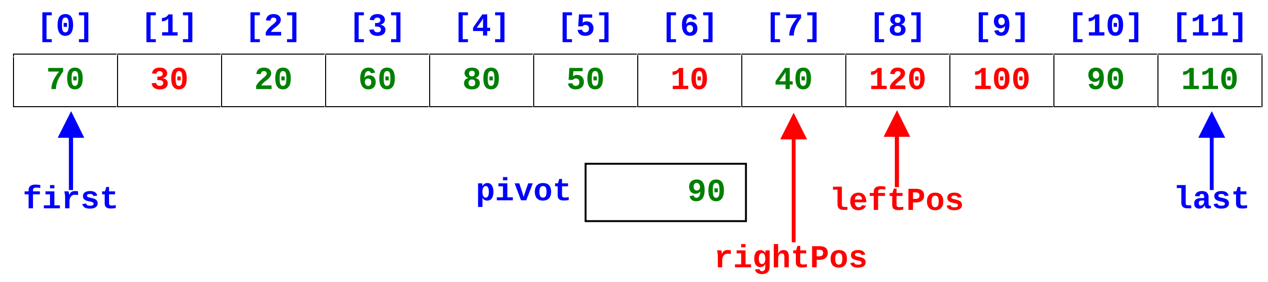
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
70	100	20	60	80	90	120	40	10	30	50	110

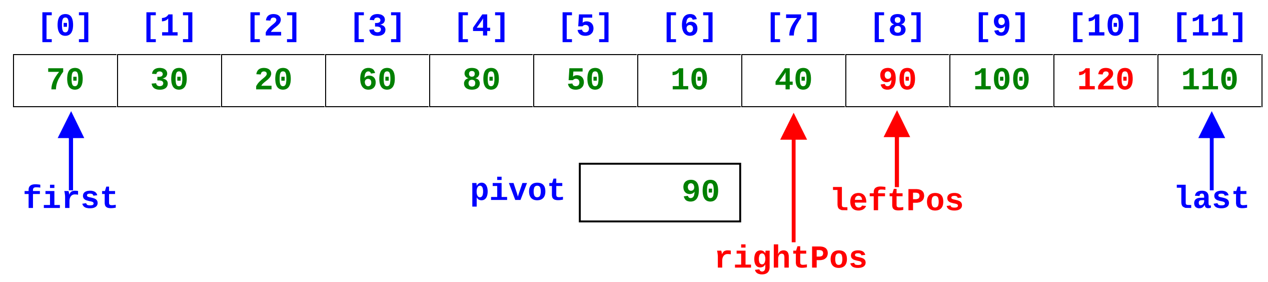
[1] Weiss, Mark Allen, *Data Structures and Algorithm Analysis in Java*, page 242.

This means that `a[first+last/2]` is the median and, hence, the pivot. Furthermore, `a[first]` and `a[last]` have already been correctly partitioned; i.e. `a[first] <= pivot` and `a[last] >= pivot`. Thus, the Partition algorithm can commence by swapping the pivot value with the item immediately to the left of the right end and starting its left and right markers at `first` and `last-1`:



The Partition now proceeds as before, yielding:





According to Weiss, median-of-three partitioning not only eliminates the worst-case behavior of Quicksort, it also reduces it's the running time by 5%.[2]

[2] Ibid, page 243.

Small Partitions (pages 350—354)

When using median-of-three partitioning, the Quicksort recursion will not work for partitions smaller than three elements, so we must substitute a different sorting method such as Straight Selection or Linear Insertion sort.

This can be implemented fairly simply in one of two ways:

1. Change **`SEGMENT_SIZE`** to some value larger than 1. Then **`recquicksort`**'s initial **`if`** statement:

   ```
   if ( last - first + 1 <= SEGMENT_SIZE )
         return;
   ```

   will stop at small segments leaving the entire array almost, but not quite, sorted. Once Quicksort is finished, you can run the entire array through the Linear Insertion sort to complete the job. Since the array is almost sorted, Linear Insertion will run in approximately O(n) time.

2. Change **`SEGMENT_SIZE`** to some value larger than 1 and change **`recquicksort`**'s initial **`if`** statement to:

   ```
   if (last - first + 1 <= SEGMENT_SIZE )
      otherSort( first, last );
   ```

   Where **`othersort`** is say, linear insertion sort, which works well on small seqments.

Sedgewick says that either of these techniques, when used with median-of-three partitioning, results in about 20% reduction in the running time of Quicksort. Furthermore, this improvement is about the same for any value of *M* from 5 to 25.[3]

[3] Sedgewick, Robert, *Algorithms*, Addison-Wesley Publishing Company, 1988, page 124.