Why is QuickSort so efficient?
def partition(L, first, last):
    # We pick the element L[first] as the "pivot" around which we partition the list
    p = first

    # We process the rest of the elements, one-by-one, in left-to-right order
    for current in range(p+1, last+1):
        # If L[current] is smaller than the pivot, it needs to move into the first block,
        # to the left of the pivot.
        if L[current] < L[p]:
            swap(L, current, p+1)
            swap(L, p, p+1)
            p = p + 1

    return p
The **partition** function in action

- Suppose
  
  \[ L = [7, 2, 13, 19, 3, 19, 8, 11, 12, 16, 1, 7] \]

- Say we call
  
  \[ \text{partition}(L, 0, 11) \]
First few iterations of \textit{partition}

<table>
<thead>
<tr>
<th>Processed</th>
<th>Unprocessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] 7 [ ]</td>
<td>[2, 13, 19, 3, 19, 8, 11, 12, 16, 1, 7]</td>
</tr>
<tr>
<td>\textbf{swaps}: 2 \leftrightarrow 2, 2 \leftrightarrow 7</td>
<td></td>
</tr>
<tr>
<td>[2] 7 [ ]</td>
<td>[13, 19, 3, 19, 8, 11, 12, 16, 1, 7]</td>
</tr>
<tr>
<td>0 swaps</td>
<td></td>
</tr>
<tr>
<td>[2] 7 [13]</td>
<td>[19, 3, 19, 8, 11, 12, 16, 1, 7]</td>
</tr>
<tr>
<td>0 swaps</td>
<td></td>
</tr>
<tr>
<td>[2] 7 [13, 19]</td>
<td>[3, 19, 8, 11, 12, 16, 1, 7]</td>
</tr>
<tr>
<td>\textbf{swaps}: 3 \leftrightarrow 13, 3 \leftrightarrow 7</td>
<td></td>
</tr>
<tr>
<td>[2, 3] 7 [19, 13]</td>
<td>[19, 8, 11, 12, 16, 1, 7]</td>
</tr>
</tbody>
</table>
The rest of the iterations

\[[2, 3] 7 [19, 13] || [19, 8, 11, 12, 16, 1, 7]\]
\[[2, 3] 7 [19, 13, 19] || [8, 11, 12, 16, 1, 7]\]
\[[2, 3] 7 [19, 13, 19, 8] || [11, 12, 16, 1, 7]\]
\[[2, 3] 7 [19, 13, 19, 8, 11] || [12, 16, 1, 7]\]
\[[2, 3] 7 [19, 13, 19, 8, 11, 12] || [16, 1, 7]\]
\[[2, 3] 7 [19, 13, 19, 8, 11, 12, 16] || [1, 7]\]
\[[2, 3, 1] 7 [13, 19, 8, 11, 12, 16, 19] || [7]\]
\[[2, 3, 1] 7 [13, 19, 8, 11, 12, 16, 19, 7] ||\]

The function returns 3.
The QuickSort function

def generalQuickSort(L, first, last):
    # Base case: if first == last, then there is only one element in the
    # slice that needs sorting. So there is nothing to do.
    
    # Recursive case: if there are 2 or more elements in the slice L[first:last+1]
    if first < last:
        # Divide step: partition returns an index p such that
        # first <= p <= last and everthing in L[first:p] is <= L[p]
        # and everything in L[p+1:last+1] is >= L[p]
        p = partition(L, first, last)

        # Conquer step
        generalQuickSort(L, first, p-1)
        generalQuickSort(L, p+1, last)

    # Combine step: there is nothing left to do!
quickSort in action

• L = [3, 6, 9, 1, 3]. Suppose we call quickSort(L).

Calling quicksort on [3, 6, 9, 1, 3]
Divide step gives [1] 3 [9, 6, 3]
Calling quickSort on [1]
Calling quickSort on [9, 6, 3]
Divide step gives [6, 3] 9 []
Calling quickSort on [6, 3]
Divide step gives [3] 6 []
Calling quickSort on [3]
Calling quickSort on []
Calling quickSort on []
quickSort in action

[3, 6, 9, 1, 3]

1

Base Case

9, 6, 3

9

Base Case

6, 3

6

Base Case

[3]

Base Case

[ ]

[ ]
Efficiency of quickSort

- **Key observation 1**: partition was designed so as to take $n$ steps on a list of size-$n$.

- **Key observation 2**: the relative sizes of the two blocks resulting from partition plays a critical role in determining the overall running time of quickSort.
Best case example
Worst case example

\[
\begin{align*}
1 + 2 + 3 + \ldots + n &= n^{2/2}.
\end{align*}
\]
So how does one pick a good pivot?

Simple (and effective) solution:
Pick a random element as the pivot!

Code
# Execute these two lines of code at the
# beginning of partition
r = random.randint(first, last)
swap(L, first, last)