1 What it Does

Sorts a list of elements on which there is a total order. Think of integers or real numbers.

2 Overview

QuickSort works by first choosing an element in the list called the pivot value (at some initial pivot index) and then sorting the list (including potentially moving the the pivot value) so that every element smaller than pivot value is to the left of it and every element larger than the pivot value is to the right of it. This is called the partitioning process.

We then apply QuickSort recursively to the left and right sublists.

When QuickSort is applied to a single element it does nothing, since a single element is always sorted.

The choice of pivot value is nuanced. For now we will consistently choose the final value in the list. If another value is chosen it is simply first exchanged with the final value in the list before proceeding.
3 Partitioning Overview

The encoding of the partitioning process can seem a bit convoluted so it’s worth summarizing what is effectively happening in the following way which clarifies that it does what we claim it does:

Pick the leftmost element which is greater than the pivot and swap it with the first subsequent element which is less than or equal to the pivot. Repeating until there are no subsequent elements left. The final swap will be with the actual pivot element and the result will be that all elements to the left of it will be less than or equal to it and all elements to the right of it will be greater than it.

Example 3.1. Consider the list $A$

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
index & 0 & 1 & 2 & 3 & 4 \\
\hline
$A$ & 5 & 2 & 4 & 1 & 3 \\
\hline
\end{tabular}
\end{center}

The leftmost element greater than $p = 3$ is $A[0] = 5$. The first subsequent element less than or equal to $p = 3$ is $A[1] = 2$ so we swap those two:

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
index & 0 & 1 & 2 & 3 & 4 \\
\hline
$A$ & 2 & 5 & 4 & 1 & 3 \\
\hline
\end{tabular}
\end{center}

The leftmost element greater than $p = 3$ is $A[1] = 5$. The first subsequent element less than or equal to $p = 3$ is $A[3] = 1$ so we swap those two:

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
index & 0 & 1 & 2 & 3 & 4 \\
\hline
$A$ & 2 & 1 & 4 & 5 & 3 \\
\hline
\end{tabular}
\end{center}

The leftmost element greater than $p = 3$ is $A[2] = 4$. The first subsequent element less than or equal to $p = 3$ is $A[4] = 3$ so we swap those two:

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
index & 0 & 1 & 2 & 3 & 4 \\
\hline
$A$ & 2 & 1 & 3 & 5 & 4 \\
\hline
\end{tabular}
\end{center}

The leftmost element greater than $p = 3$ is $A[3] = 5$. There are no subsequent elements less than or equal to $p = 3$. Thus we are done.

Observe that the pivot values is such that all elements to the left are less than or equal to it and all elements to the right are greater than it.
4 Pseudocode

The actual algorithmic implementation is a bit more nuanced and has a small quirk.

\[
\begin{align*}
&\text{\texttt{\textbackslash PRE: A is a list of length n.}} \\
&\text{\texttt{\textbackslash Note that A is considered global here.}} \\
&\text{\texttt{function quicksort(A,l,r)}} \\
&\quad \text{\texttt{if l<r then}} \\
&\quad \quad \text{\texttt{resultingpivotindex = partition(A,l,r)}} \\
&\quad \quad \text{\texttt{quicksort(A,l,resultingpivotindex-1)}} \\
&\quad \quad \text{\texttt{quicksort(A,resultingpivotindex+1,r)}} \\
&\quad \text{\texttt{end}} \\
&\text{\texttt{end}} \\
&\text{\texttt{function partition(A,l,r)}} \\
&\quad \text{\texttt{\textbackslash To use a different pivotvalue}} \\
&\quad \text{\texttt{\textbackslash swap it with A[r] here.}} \\
&\quad \text{\texttt{pivotvalue = A[r]}} \\
&\quad \text{\texttt{t = 1}} \\
&\quad \text{\texttt{for i = l to r-1}} \\
&\quad \quad \text{\texttt{if A[i] \leq pivotvalue}} \\
&\quad \quad \quad \text{\texttt{A[t] <-> A[i]}} \\
&\quad \quad \quad \text{\texttt{t = t + 1}} \\
&\quad \text{\texttt{end}} \\
&\quad \text{\texttt{A[t] <-> A[r]}} \\
&\quad \text{\texttt{return t}} \\
&\text{\texttt{end}} \\
&\text{\texttt{quicksort(A,0,n-1)}} \\
&\text{\texttt{\textbackslash POST: A is sorted.}}
\end{align*}
\]

Loosely speaking t keeps track of the leftmost value larger than the pivot value and i hunts down the subsequent value less than or equal to the pivot value. We say “loosely” because if the list starts with values less than or equal to the pivot value then the algorithm will swap them with themselves for a while.
Example 4.1. Let's trace the implementation on the list:

\[
\begin{array}{ccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 2 & 1 & 6 & 1 & 8 & 4 & 5 \\
\end{array}
\]

The pivot it pivot value=5.
We set \(t=0\) and iterate from \(i=0\):

\[
\begin{array}{ccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 2 & 1 & 6 & 1 & 8 & 4 & 5 \\
t,i & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
p=5
\end{array}
\]

We see \(A[0]=2\leq5\) is true and swap \(A[i]\leftarrow A[t]\) which effectively swaps \(A[0]\leftarrow A[0]\).
We increase \(t\) so now \(t=1\). We now have \(i=1\):

\[
\begin{array}{ccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 2 & 1 & 6 & 1 & 8 & 4 & 5 \\
t,i & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\
p=5
\end{array}
\]

We see \(A[1]=1\leq5\) is true and swap \(A[i]\leftarrow A[t]\) which effectively swaps \(A[1]\leftarrow A[1]\).
We increase \(t\) so now \(t=2\). We now have \(i=2\):

\[
\begin{array}{ccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 2 & 1 & 6 & 1 & 8 & 4 & 5 \\
t,i & 1 & 3 & 3 & 3 & 3 & 3 & 3 \\
p=5
\end{array}
\]

We see \(A[2]=6\leq5\) is false and we do nothing. We now have \(i=3\):

\[
\begin{array}{ccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 2 & 1 & 6 & 1 & 8 & 4 & 5 \\
t,i & 1 & 4 & 4 & 4 & 4 & 4 & 4 \\
p=5
\end{array}
\]

Notice that finally \(t\) indicates the location of the leftmost element greater than the pivot value.
We see \(A[3]=1\leq5\) is true and swap \(A[i]\leftarrow A[t]\) which effectively swaps \(A[3]\leftarrow A[2]\).
We increase \(t\) so now \(t=3\). We now have \(i=4\):

\[
\begin{array}{ccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 2 & 1 & 1 & 6 & 8 & 4 & 5 \\
t,i & 1 & 5 & 5 & 5 & 5 & 5 & 5 \\
p=5
\end{array}
\]

We see \(A[4]=6\leq5\) is false and we do nothing. We now have \(i=5\):

\[
\begin{array}{ccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 2 & 1 & 1 & 6 & 8 & 4 & 5 \\
t,i & 1 & 6 & 6 & 6 & 6 & 6 & 6 \\
p=5
\end{array}
\]

Now we are done.

Here is another example more briefly:

**Example 4.2.** For example consider this list:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Set $t=0$ and $i=0$:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

No swap, iterate $i=1$:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Swap, iterate $t=1$ and iterate $i=2$:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>4</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Swap, iterate $t=2$ and iterate $i=3$:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>
Swap, iterate \( t=3 \) and iterate \( i=4 \):

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 10 & 7 & 4 & 3 & 5 & 6 & 2 & 11 & 5 \\
t & 7 & i & 4 & p=5 \\
\end{array}
\]

No swap, iterate \( i=5 \):

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 10 & 7 & 4 & 3 & 5 & 6 & 2 & 11 & 5 \\
t & 7 & i & 3 & p=5 \\
\end{array}
\]

Swap, iterate \( t=4 \) and iterate \( i=6 \):

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 14 & 7 & 10 & 3 & 5 & 6 & 2 & 11 & 5 \\
t & 10 & i & 5 & p=5 \\
\end{array}
\]

Swap, iterate \( t=5 \) and iterate \( i=7 \):

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 14 & 3 & 7 & 10 & 5 & 6 & 2 & 11 & 5 \\
t & 7 & i & 6 & p=5 \\
\end{array}
\]

Swap, iterate \( t=6 \) and iterate \( i=8 \):

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 14 & 3 & 5 & 10 & 6 & 2 & 11 & 5 \\
t & 10 & i & 2 & p=5 \\
\end{array}
\]

No swap, iterate \( i=9 \):

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 14 & 3 & 5 & 7 & 10 & 6 & 2 & 11 & 5 \\
t & 10 & i & 11 & p=5 \\
\end{array}
\]

Swap, iterate \( t=7 \) and iterate \( i=10 \):

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 14 & 3 & 5 & 2 & 10 & 6 & 7 & 11 & 5 \\
t & 10 & i & 7 & p=5 \\
\end{array}
\]

Loop done. Swap \( A[t] \leftrightarrow A[r] \) —

\[
\begin{array}{cccccccccccc}
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
A & 4 & 3 & 1 & 14 & 3 & 5 & 2 & 5 & 6 & 7 & 11 & 10 \\
\end{array}
\]
5 Pivot Value Choice

For simplicity we choose the last value of the sublist as the pivot but this is perhaps less than idea. For example if the list is already sorted then choosing the last element as the pivot value results in the algorithm taking as long as possible and not actually changing anything - try it on some data and see!

Intuitively if the input is close to sorted then choosing the last element (or in fact the first element) as the pivot will result in very slow running time.

In order to prevent this there are other ways we can choose the initial pivot value. One way is simply randomly. This adds very little time to the process since typically random number generation is $\Theta(1)$. Whether it helps or not is something we’ll have to see.

It might be tempting to choose the initial pivot index so that the pivot value is the median, since that results in a nice balanced partition, but this is in and of itself challenging as we’ll see later.

6 Pseudocode Time Complexity

Let $T(n)$ be the time complexity of a call to QuickSort.

A call to QuickSort on a list of length $n$ invokes both a partitioning call and two subsequent recursive calls to QuickSort. If the `resultingpivotindex` returns index $k$ then one of them to a sublist of length $k$ and one of them to a sublist of length $n - k - 1$.

Those recursive calls take time $T(k)$ and $T(n - k - 1)$ respectively. The partition call is $\Theta(n)$ and hence we have the recurrence relation:

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

1. **Worst Case:** The worst-case occurs when the `resultingpivotindex` is the first or last element in the sublist.

   This results in the sublist being only one element smaller than the list itself and the other sublist being length zero. Without loss of generality if $k = 0$ in the above relation we have

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

   which results in $T(n) = \Theta(n^2)$.

   Explanation omitted for now. I had one but it was wrong and I haven’t replaced it yet.

2. **Best Case:** The best-case occurs when the `resultingpivotindex` is in the middle of the sublist.
This results in the sublists being of equal size. Then in the above relation we have

\[ T(n) = 2T(n/2) + \Theta(n) \]

which results in \( T(n) = \Theta(n \lg n) \) by the Master Theorem.

3. **Average-Case:**

As per earlier discussions we need to understand what “average” means. At its most basic level we would need to look at all possible permutations of a list, apply QuickSort to each, and average the results.

This can be done mathematically using the following constructive inductive approach: First figure out \( T(0) \) and \( T(1) \). Then the average-case can be calculated by adding the partition time to the average cases of the two recursive calls.

However no matter the approach it now becomes relevant how we are choosing the pivot. This didn’t matter in the best-case and worst-case situations by definition of “best-case” and “worst-case”.

If we choose the initial pivot index randomly then it doesn’t matter how the list is arranged at the start because the pivot selection effectively eliminates (or accounts for, depending on how we want to look at it) the impact on this initial arrangement.

So for the sake of argument let’s suppose that we choose the pivot randomly. For simplicity we’ll also assume the elements in the list are distinct. In this case the recurrence relation is as follows as we are finding the average time over all possible pivot index choices:

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
\frac{1}{n} \sum_{k=0}^{n-1} [T(k) + T(n - k - 1) + \Theta(n)] & \text{otherwise}
\end{cases}
\]

This is not easy to solve but essentially the idea is to show that this is \( \mathcal{O}(n \lg n) \). This is done by constructive induction. I am omitting the proof for now.

7 **Auxiliary Space**

QuickSort uses \( \Theta(1) \) auxiliary space.

8 **Stability**

QuickSort is not stable.
9 In-Place

QuickSort is in-place.

10 Notes

A few notes:

1. Ideally (mathematically) at each step the median of the values would be used as the pivot element. The exercises ask you to think about why. There is a method we’ll see later called the Median of Medians which can find the median in $O(n)$ time. However it’s still slow, relatively speaking, and thus...

2. In practice choosing an element randomly is the usual approach, even though this the actual implementation is slightly slower. The exercises ask you to think about why.

3. After $k$ iterations we can draw some conclusions about which elements are correctly placed. The exercises discuss this further.
11 Thoughts, Problems, Ideas

1. Show the steps of the first partition of QuickSort on the list \([10, 6, 7, 2, 4, 3]\). Use the final index as the initial pivot index.

2. Show the full steps of QuickSort on the list \([10, 6, 7, 2, 4, 3]\). Use the final index as the initial pivot index.

3. Consider QuickSort used on the list \([11, 6, 5, 45, 23, 7, 2]\). Which initial pivot index and pivot value should be used as the first pivot to ensure that the result of the first partition is equally balanced? Show the result of doing this first partition.

4. Suppose the resulting pivot index somehow ended up consistently one third of the way through the list. What would be the corresponding recurrence relation? Can the Master Theorem be used to solve this? Can you say anything about the time complexity?

5. Consider the theoretical case where the resulting pivot index consistently ends up 1/2 of the way through the list and the theoretical case where the resulting pivot index consistently ends up 1/4 of the way through the list. Essentially the corresponding recurrence relations would be something like:

\[
T_{1/2}(n) = T([n/2]) + T([n/2]) + \Theta(n)
\]

\[
T_{1/4}(n) = T([n/4]) + T([3n/4]) + \Theta(n)
\]

Suppose in addition that:

- The \(\Theta(n)\) term is actually \(5n + 2\).
- You know that \(T(0) = 0\) and \(T(1) = 2\).

(a) Find each time complexity for values \(n = 0, 10, 20, ..., 100\). (You are welcome to do this in recursive code and if you do, include your code. It’s not hard - each is six simple lines of Python, for example.)

(b) Plot the corresponding data and connect with smooth lines.

(c) Which of these does your data suggest has a better \(\Theta\) time complexity? Explain.

6. Give a specific example which illustrates the fact that QuickSort is not stable. Illustrate where in the process the stability breaks down. You don’t need to show the entire implementation, just enough to justify.
7. Consider these two approaches to pivot value selection:
   - Obtain index of the median, use as initial pivot index.
   - Choose random index, use as initial pivot index.

In practice both of these are average case $\Theta(n \log n)$.

(a) Mathematically, using the median is better. Why?
(b) In practice, using a random element is the standard approach. Why not the median?

8. Suppose a list has $n = 2^k - 1$ elements for some $k$ and suppose that somehow, magically, the index of the median is chosen at every stage for the initial pivot index. In this case we can explicitly calculate the number of calls to `quicksort`, the length of each list it is called on, and the number of subsequent calls to `partition`. Remember that `quicksort` only calls `partition` in the case $l<r$ so when $l=r$ (list of length 1) `quicksort` exits.

Consider the case where $n = 2^4 - 1 = 15$. Observe there will be:

   - 1 initial call to `quicksort` with a list of length 15, resulting in 1 call to `partition` and then ...
   - a total of 2 calls to `quicksort` with lists of length 7, resulting in a total of 2 calls to `partition` and then ...
   - a total of 4 calls to `quicksort` with lists of length ???, resulting in a total of ??? calls to `partition` and then ...
   - and so on, until it ends.

(a) Complete the counting argument above - finish the third bullet point and then the remaining bullet points until the argument ends. There should be only four bullet points total.
(b) What would the counting argument be for $n = 2^k - 1$ for an arbitrary $k$? It’s not hard, just generalize the pattern in (a).
(c) Suppose that each call to `partition` on a list of length $i$ takes time $c_1i$. Ignoring all other time requirements (the call to `partition` is the important one) write down and evaluate the sum which gives the total time requirement of the algorithm. Does the time complexity of this result correspond to the best-case analysis?

9. Modify the QuickSort pseudocode so that it chooses the first element as the pivot value.

10. Modify the QuickSort pseudocode so that it randomly chooses a pivot.

11. Modify the QuickSort pseudocode so that it sorts the list in decreasing order.
import random
A = []
for i in range(0,15):
    A.append(random.randint(0,100))
n = len(A)
print(A)
def quicksort(l,r,indent):
    if l<r:
        resultingpivotindex = partition(l,r,indent+2)
        quicksort(l,resultingpivotindex-1,indent+2)
        quicksort(resultingpivotindex+1,r,indent+2)
        print(indent*'_' + ' Recombine: ' + str(A[l:r+1]))

def partition(l,r,indent):
    print(indent*'_' + ' Subarray: ' + str(A[l:r+1]))
    # To use a different pivotvalue
    # swap it with A[r] here.
    pivotvalue = A[r]
    t = l
    for i in range(l,r):
        if A[i] <= pivotvalue:
            temp = A[t]
            A[t] = A[i]
            A[i] = temp
            t = t + 1
    temp = A[t]
    A[r] = temp
    print(indent*'_' + ' Pivot around final element. ')
    print(indent*'_' + ' Result: ' + str(A[l:r+1]))
    return(t)

quicksort(0,n-1,0)
print(A)
Output:

```
[50, 4, 24, 90, 92, 84, 33, 27, 81, 13, 2, 44, 62, 28, 80]
___Subarray: [50, 4, 24, 90, 92, 84, 33, 27, 81, 13, 2, 44, 62, 28, 80]
___Pivot around final element.
___Result: [50, 4, 24, 33, 27, 13, 2, 44, 62, 28, 80, 92, 81, 84, 90]
____Subarray: [50, 4, 24, 33, 27, 13, 2, 44, 62, 28]
____Pivot around final element.
____Result: [4, 24, 27, 13, 2, 28, 50, 44, 62, 33]
_____Subarray: [4, 24, 27, 13, 2]
_____Pivot around final element.
_____Result: [2, 24, 27, 13, 4]
_______Subarray: [24, 27, 13, 4]
_______Pivot around final element.
_______Result: [4, 27, 13, 24]
_________Subarray: [27, 13, 24]
_________Pivot around final element.
_________Result: [4, 24, 27, 13, 2]
________Recombine: [4, 24, 27]
________Recombine: [4, 13, 24, 27]
_____Recombine: [2, 13, 24, 27]
_____Subarray: [24, 27]
_____Pivot around final element.
_____Result: [24, 27]
______Recombine: [24, 27]
Recombine: [2, 4, 24, 27]
-----------------------
```

```
[2, 4, 13, 24, 27, 28, 33, 44, 50, 62, 80, 81, 84, 90, 92]
```
13 Python Test - Pivot on Random Element

Code:

```python
import random
A = []
for i in range(0,13):
    A.append(random.randint(0,100))
n = len(A)
print(A)
def quicksort(l,r,indent):
    if l<r:
        pivotindex = partition(l,r,indent+2)
        quicksort(l,pivotindex-1,indent+2)
        quicksort(pivotindex+1,r,indent+2)
        print(indent*'_' + 'Recombine: ' + str(A[l:r+1]))
def partition(l,r,indent):
    print(indent*'_' + 'Subarray: ' + str(A[l:r+1]))
    p = random.randint(l,r)
    temp = A[p]
    A[r] = temp
    pivot = A[r]
t = l
    for i in range(l,r):
        if A[i] <= pivot:
            temp = A[t]
            A[t] = A[i]
            A[i] = temp
            t = t + 1
    temp = A[t]
    A[r] = temp
    print(indent*'_' + 'Pivot around index ' + str(p-l))
    print(indent*'_' + 'Result: ' + str(A[l:r+1]))
    return(t)
quicksort(0,n-1,0)
print(A)
```
Output:

```plaintext
[92, 55, 6, 43, 30, 43, 37, 66, 63, 24, 92, 3, 79]
__Subarray: [92, 55, 6, 43, 30, 43, 37, 66, 63, 24, 92, 3, 79]
__Pivot around index 2
__Result: [3, 6, 79, 43, 30, 43, 37, 66, 63, 24, 92, 92, 55]
___Subarray: [79, 43, 30, 43, 37, 66, 63, 24, 92, 92, 55]
___Pivot around index 3
___Result: [43, 30, 37, 24, 43, 66, 63, 55, 92, 92, 79]
____Subarray: [43, 30, 37, 24]
____Pivot around index 3
____Result: [24, 30, 37, 43]
_____Subarray: [30, 37, 43]
_____Pivot around index 2
_____Result: [30, 37, 43]
______Subarray: [30, 37]
______Pivot around index 1
______Result: [30, 37]
______Recombine: [30, 37]
_____Recombine: [30, 37, 43]
____Recombine: [24, 30, 37, 43]
____Subarray: [66, 63, 55, 92, 92, 79]
____Pivot around index 1
____Result: [55, 63, 66, 92, 92, 79]
_____Subarray: [66, 92, 92, 79]
_____Pivot around index 0
_____Result: [66, 92, 92, 79]
______Subarray: [92, 92, 79]
______Pivot around index 1
______Result: [92, 92, 79]
_______Subarray: [92, 79]
_______Pivot around index 0
_______Result: [92, 79]
________Recombine: [92, 79]
________Recombine: [92, 79, 92]
-----Recombine: [66, 79, 92, 92]
-----Recombine: [66, 79, 92, 92]
-----Recombine: [55, 63, 66, 79, 92, 92]
-----Recombine: [24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
Recombine: [3, 6, 24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
[3, 6, 24, 30, 37, 43, 43, 55, 63, 66, 79, 92, 92]
```