QuickSort

Invented by British computer scientist Tony Hoare in 1960 while studying in Moscow, published in 1961.

Divide-and-Conquer algorithm:

1. If the input array has < two elements, do nothing.
   Otherwise, call Partition: Pick a pivot key and use it to divide the array into two:

   \[
   \begin{array}{ccc}
   \leq & \text{pivot} & \geq \text{pivot} \\
   \end{array}
   \]

2. Sort the two subarrays recursively.
Algorithm QuickSort(A, p, r)

1. if p < r then
2. \( \text{split} \leftarrow \text{Partition}(A, p, r) \)
3. QuickSort(A, p, split - 1)
4. QuickSort(A, split + 1, r)
**Algorithm** Partition($A, p, r$)

1. $pivot \leftarrow A[r].key$
2. $i \leftarrow p - 1$
3. **for** $j \leftarrow p$ **to** $r - 1$ **do**
4. \hspace{1em} **if** $A[j] \leq pivot$
5. \hspace{2em} $i \leftarrow i + 1$
6. \hspace{2em} exchange $A[i]$ and $A[j]$
7. exchange $A[i + 1]$ and $A[r]$
8. **return** $i + 1$

**Invariant:** $i$ is 1 less than the leftmost $> pivot$ value in the range $p \ldots j$ (or is $j - 1$ if no $> pivot$ is there).
Partition discussion \((i \text{ and } j)\)
Partition example

\[ \begin{array}{cccccccccc}
  & p & 2 & 9 & 6 & 19 & 8 & 3 & 4 & 20 & 5 & 16 & r \\
  i & j & & & & & & & & & & \\
\end{array} \]
Partition example (done on video)

```
pivot → 16
```

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
<td>6</td>
<td>19</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finally we will swap A[i+1] (20) with A[r] (16) and return (i+1)
Correctness of Partition

Assume we are part-way through Partition and for this \( j \) (and this \( i \)), we have the Invariant (all \( A[i+1 \ldots j-1] \) are \( > pivot \)). Induction step:

- If \( A[j].key > pivot \), the algorithm changes nothing. And the range of \( > pivot \) cells expands by 1 (as \( j \) gets incremented).
- If \( A[j].key \leq pivot \), we will swap \( A[j] \) with \( A[i+1] \). \( A[i+1].key \) either …
  - was \( \leq pivot \) (and \( i = j-1 \)), in which case \( i' = i + 1 \) is \( j \) and we swap \( A[j] \) with itself, and then have the same pattern after \( j \) increments, or
  - was \( > pivot \), in which case …

After loop, Invariant implies that the first \( > pivot \) (if any) is in \( A[i+1] \). Hence swapping \( A[r] \) and \( A[i+1] \) gives a “partition” of the desired form.

\[ \text{IADS – Lecture 13 – slide 8} \]
Results from Partition

- We might get a fairly balanced partition, with “pivot” lying near the middle ("split" roughly halfway between \( p \) and \( r \)).
  - When this happens, the Divide-and-Conquer balance is like MergeSort (and also, we have \( \Theta(n) \) work at the “top level”)
- Alternatively, we could get a very unbalanced partition, with one side empty or very small
  - Then we are doing linear work to reduce the size of the problem to be solved only a tiny bit. More like BubbleSort.
- Or anything in between, depends on the original arrangement of keys (and what “rank” pivot has among the keys in the array).
Running Time of QuickSort

Partition

\[ T_{\text{partition}}(n) = \Theta(n) \]

QuickSort

\[ T_{\text{quickSort}}(n) \in \max_{1 \leq s \leq n-1} \left( T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n - s - 1) \right) + T_{\text{partition}}(n) + \Theta(1) \]

\[ = \max_{1 \leq s \leq n-1} \left( T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n - s - 1) \right) + \Theta(n). \]

Implies

\[ T_{\text{quickSort}}(n) \in \Theta(n^2) \]

To show \( \Omega(n^2) \) you need a specific structured input (not too hard).
QuickSort

The average-case running-time of a sorting algorithm is the average number of computational steps (comparisons) carried out on a uniform random permutation of the keys \( \{1, \ldots, n\} \).

- We don’t say “amortized” as we have a single computation, and are comparing input of exactly the same size.
- For sorting-algorithms, typically the running-time can be captured by the number of “comparisons” (these measures tend to be asymptotically equivalent).
- Uniform random permutation means all permutations arise with same probability.

The average-case running time of QuickSort is \( \Theta(n \lg(n)) \).
QuickSort

- QuickSort can be very fast in practice.
- But performs badly — $\Theta(n^2)$ — on sorted and almost sorted arrays.

Practical Improvements

- Different choice of pivot (key of middle item, random)
- Refined partitioning routine
- Use InsertionSort for small arrays (similar to “TimSort”)

RandomQuickSort

- The $\Theta(n \lg(n))$ result for average-case can be shown to carry over characterize (expected) running-time for RandomQuickSort (choose the pivot randomly).
The default sorting algorithm in python is “Timsort”, an optimized version of MergeSort developed by Tim Peters, a major contributor to the development of CPython.

- Does a pre-processing step looking for “runs” of strictly decreasing or (non-strict) increasing items.
- We understand how to handle “runs” (without sorting).
- Then merge the sorted runs, using InsertSort for short subarrays, and MergeSort for bigger subarrays.
Reading Material

Personal Reading:

- QuickSort and its analysis, sections 7.1, 7.2 and 7.4 of [CLRS]
- QuickSort can also be found in “Algorithms Illuminated”, Chapter 5 (not sure it's our version)
- “QuickShort” interview with Tony Hoare can be viewed at http://anothercasualcoder.blogspot.com/2015/03/my-quickshort-interview-with-sir-tony.html
- To read about “Timsort” ... read the listsort.txt file in the Objects directory from the python download.

Source code can be downloaded
https://www.python.org/downloads/source/