QuickSort Pseudocode

QUICKSORT(A, p, r)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if p &lt; r</td>
</tr>
<tr>
<td>2</td>
<td>q = Partition(A, p, r);</td>
</tr>
<tr>
<td>3</td>
<td>Quicksort(A, p, q-1);</td>
</tr>
<tr>
<td>4</td>
<td>Quicksort(A, q+1, r);</td>
</tr>
</tbody>
</table>

- What’s the call to sort the entire array?
### Partitioning the Array

**PARTITION(A, p, r)**

<table>
<thead>
<tr>
<th>Partition(A,p,r) // A:Array; p,r: integer indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x = A[r]</td>
</tr>
<tr>
<td>2 i = p - 1</td>
</tr>
<tr>
<td>3 for j = p to r-1</td>
</tr>
<tr>
<td>4 if A[j] &lt;= x</td>
</tr>
<tr>
<td>5 i = i + 1</td>
</tr>
<tr>
<td>6 swap(A[i], A[j])</td>
</tr>
<tr>
<td>7 swap (A[i+1], A[r])</td>
</tr>
<tr>
<td>8 return i+1</td>
</tr>
</tbody>
</table>

**Performance of Quicksort**

- What does the performance of quicksort depend on?
- What would give us the best case?
Best Case of Quicksort

How many levels?

Recurrence Equation?

Worst Case of Quicksort

Suppose our pivot element splits the array as unequally as possible. Thus instead of n/2 elements in the larger half, we get zero, meaning that the pivot element is the biggest element in the array. Now we have n-1 levels, instead of \( \lg n \).

\[ T(n) = T(n-1) + O(n), \] which has a solution of \( O(n^2) \)
Almost Worst Case of Quicksort

- Assume each partition is a 9 to 1 split
  - constant proportionality
  - What is the recurrence?

Fig 7.4  What does the recursion tree look like (9-1 split)?

\[ O(n \log n) \]
Average Case Analysis

- Let’s look at this intuitively.
- Running quicksort on a random array is likely to produce a mix of balanced and unbalanced partitions.
- It has been shown that 80% of the time partition produces “good” splits and 20% of the time it produces “bad” splits.

This is really no different than:

Thus, the \( O(n-1) \) of the bad split can be absorbed into the \( O(n) \) of the good split.
Average Case Analysis

- Running time of quicksort when alternating good and bad splits is similar to the running time for good splits alone
- $O(n \log n)$ but with a slightly larger constant hidden by the big-O notation

Empirical Support for $O(n\log(n))$

<table>
<thead>
<tr>
<th>n</th>
<th>Compare</th>
<th>Swap</th>
<th>Total</th>
<th>$n \log n$</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>23</td>
<td>119</td>
<td>142</td>
<td>33.2193</td>
<td>4.2746259</td>
</tr>
<tr>
<td>50</td>
<td>295</td>
<td>353</td>
<td>648</td>
<td>282.193</td>
<td>2.2963023</td>
</tr>
<tr>
<td>100</td>
<td>906</td>
<td>1022</td>
<td>1928</td>
<td>664.386</td>
<td>2.9019292</td>
</tr>
<tr>
<td>500</td>
<td>8734</td>
<td>7903</td>
<td>16637</td>
<td>4482.89</td>
<td>3.7112202</td>
</tr>
<tr>
<td>1000</td>
<td>16423</td>
<td>11860</td>
<td>2823</td>
<td>9965.78</td>
<td>2.8380105</td>
</tr>
<tr>
<td>5000</td>
<td>97973</td>
<td>67398</td>
<td>165371</td>
<td>61438.6</td>
<td>2.6916483</td>
</tr>
<tr>
<td>10000</td>
<td>296511</td>
<td>189527</td>
<td>486038</td>
<td>132877</td>
<td>3.6578004</td>
</tr>
</tbody>
</table>

Note: Generated random arrays of size N, took worst of 2 runs
Choosing Pivots

- How else could we choose pivots?

---

**RANDOMIZED-PARTITION, p 179**

```plaintext
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  swap (A[r], A[i])
3  return PARTITION(A, p, r)
```

```plaintext
RANDOMIZED-QUICKSORT(A,p,r)
1  If p < r
2  q = RANDOMIZED-PARTITION(A, p, r)
3  RANDOMIZED-QUICKSORT(A,p,q-1)
4  RANDOMIZED-QUICKSORT(A,q+1,r)
```
**MEDIAN-OF-3-PARTITION (Ex. 7-5)**

1. $i = \text{INDEX-MEDIAN-VALUE}(A,p,r)$
2. swap $(A[r], A[i])$
3. return PARTITION$(A, p, r)$

**MEDIAN-OF-3-QUICKSORT$(A,p,r)$**

1. If $p < r$
2. $q = \text{MEDIAN-OF-3-PARTITION}(A, p, r)$
3. $\text{MEDIAN-OF-3-QUICKSORT}(A,p,q-1)$
4. $\text{MEDIAN-OF-3-QUICKSORT}(A,q+1,r)$

---

**Improvement!**

Note: This count includes the comparisons used to determine the median of the three values. Saw similar improvements over dozens of trials.
Hoare Partition, p 185

```
HoarePartition(A, p, r)
1 x = A[p]
2 i = p - 1
3 j = r + 1
4 while TRUE
5   do
6     j = j - 1
7     while (A[j] > x)
8       i = i + 1
9     do
10    while (A[i] < x)
11   if (i < j)
12     swap(A[i], A[j])
13   else return j
```
Next time

- Heapsort (Chapter 6)
- Next assignment (Quicksort).