Everything you always wanted to know about Quick Sort,

What lessons could QuickSort teaches us about other algorithms

**Alon Efrat**

Based on slides curaicy of
Piotr Indyk and Carola Wenk

---

**Divide and conquer**

QuickSort an \( n \)-element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** \( x \) such that elements in lower subarray \( \leq x \) ≤ elements in upper subarray.

2. **Conquer:** Recursively sort the two subarrays.

   • **Combine:** Trivial.

---

**Partitioning subroutine**

\[
\text{PARTITION}(A, p, q) \Rightarrow A[p..q] \\
x \leftarrow A[p] \quad \Rightarrow \text{pivot} = A[p] \\
i \leftarrow p \\
\text{for } j \leftarrow p + 1 \text{ to } q \Rightarrow j \text{ is hunting for small keys} \\
do \text{ if } A[j] \leq x \Rightarrow \text{Should send } A[j] \text{ to the left.} \\
\quad \text{then}
\]

\[
i \leftarrow i + 1 \Rightarrow \text{Now } A[i] > x \\
\text{exchange } A[i] \leftrightarrow A[j] \Rightarrow \text{Fix } A[i] > x
\]

\[
\text{return } A[p] \leftrightarrow A[i]
\]

---

**Key:** Linear-time partitioning subroutine.

---

- Sorts “in place” (no need for extra space). Like insertion sort, but not like merge sort.
- Very practical (with tuning).

---

**Running time = \( O(n) \)** for \( n \) elements.
Example of partitioning

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
\end{array} \]

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
\end{array} \]

Example of partitioning

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
\end{array} \]

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
\end{array} \]
Example of partitioning
Pseudocode for quicksort

QUICKSORT(A, p, r)
if p < r //do something only if contains at least 2 keys
    then q ← PARTITION(A, p, r) //both perform partition, and return index of pivot
    QUICKSORT(A, p, q-1) //QS left part
    QUICKSORT(A, q+1, r) //QS right part

Initial call: QUICKSORT(A, 1, n)

Analysis of quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n) =$ worst-case running time on an array of $n$ elements.

Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$T(n) = T(0) + T(n - 1) + \Theta(n)$$
$$= \Theta(1) + T(n - 1) + \Theta(n)$$
$$= T(n - 1) + \Theta(n)$$
$$= \Theta(n^2) \text{ (arithmetic series)}$$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]

\[ \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right) \]

Best-case and almost best-case analysis

If we are lucky, PARTITION splits the array evenly:

\[ T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n) \] (same as merge sort)

What if the split is \( \frac{9}{10} : \frac{1}{10} \)?

That is, both sub-arrays contain at least 10% of the keys (possibly more)

\[ T(n) = T\left(\frac{1}{10} n\right) + T\left(\frac{9}{10} n\right) + \Theta(n) \]

We call such a partition an almost-optimal partition.

What is the running time in this case?

Analysis of “almost-best” case

\[ T(n) \]
Analysis of “almost-best” case

$T\left(\frac{1}{10} n\right) \rightarrow cn \rightarrow T\left(\frac{9}{10} n\right)$

Analysis of “almost-best” case

$T\left(\frac{1}{10} n\right) \rightarrow \frac{1}{10} cn \rightarrow T\left(\frac{9}{10} n\right) \rightarrow \frac{9}{10} cn \rightarrow T\left(\frac{81}{100} n\right)$

Analysis of “almost-best” case

$\Theta(1)$ leaves

Analysis of “almost-best” case

$\log_{10/9} n \leq cn \log_{10/9} n \leq O(n)$
QS needs $O(n \log n)$ if partition are almost optimal
Each time the algorithm invested some work, it moves a key from one location to another

Consider a key $x$.

When the algorithm starts, it is in an array of size $n$
Then $x$ is shifted into an array of size $\leq (0.9) \cdot n$
Next, $x$ “of size $\leq (0.9)^2 \cdot n$
Next, $x$ “of size $\leq (0.9)^3 \cdot n$

... After $k$ times that $x$ was shifted, its array’s size $\leq (0.9)^k \cdot n$

Max time that $x$ is shifted:
$(0.9)^k n \leq 1 \quad OR \quad k \leq \log(\frac{1}{0.9}) n \leq 8 \log_2 n = O(\log n)$
Next we need to multiply this number of the number of keys, yielding $O(n \log n)$

### Randomized quicksort

How can find a pivot that guarantees partitions with good ratios for $A[1..n]$?
We say that $q$ is a **good pivot** for if
- at least 10% of the elements of $A[1..n]$ are smaller than $q$, and
- at least 10% of the elements of $A[1..n]$ are larger than $q$.

![10% ≤ q](image)

**Best pivot:** Pick the **median** of $A[1..n]$, as pivot.
(median – an element that is larger than half of the keys)
Then the time would obey $T(n) = cn + 2T(n/2)$

**Problem** – need to work too hard to find the median (best pivot), so we will do with (only) a good pivot. (of course, we could first sort :-).)

### Finding a good pivot for $A[1..n]$

**5-random-elements method.**
- Pick the **indices** of 5 elements at random from $A[1..n]$.
- For $k=1$ to 5
  
  
  $X[k] = A[\lfloor n \cdot \text{rand}() \rfloor]$

- $A[1..n]$

- Set $q$ to be the median of $X[1..5]$

- $S: 10\% \leq q$

What it is the probability that $q$ is **not** a good pivot?
- Let $S$ be the elements of $A[1..n]$ which are the 10% smallest.
- The probability that an elements picked at random is in $S$ is 0.1.
- $q$ is in $S$ only if **at least 3** of the 5 elements that we pick are in $S$.
- The probability that this happens is
  
  $0.15 + 5 \cdot 0.1^4 \cdot 0.9 + 10 \cdot 0.1^3 \cdot 0.9^2 = 0.00001 + 0.00045 + 0.00810 = 0.00856$

- This is also the probability that $q$ is in the 10% largest elements.
- In other words: with probability $\geq 98\%$, $q$ is a good pivot.
Putting it together

- If we performed a partition which is **not** almost optimal, nothing dramatically bad happens, we just wasted some time. Each such partition takes linear time, but has no effect.

- However, each partition is, with probability $\geq 98\%$ is good, and we obtain an almost-optimal pivot.

- Hence the expected time of QuickSort (if the 5 random keys methods is used) is $O(n \log n) + 0.02 \cdot O(n \log n) = O(n \log n)$

Randomized quicksort – cont

Finding good pivots

Putting it together, during QS, each time that we need to find a pivot, we use the “5 random elements” method. With probability 98%, we find a good pivot.

The overall time that we spend on good partitions is much smaller than the time we spent on bad partitions. (note – bad partitions are not harmful – they are just not helpful)

So the recursions formula $T(n) = cn + T(n/10) + T(9n/10)$ still apply, leading to running time $O(n \log n)$.

This is expected running time – there is a chance that the actual running time is $\Theta(n^2)$, but the probability that it happens is very slim.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort behaves well even with caching and virtual memory.

Median Selection

- (CLRS Section 9.2, page 185).
- For $A[1..n]$ (all different elements) we say that the rank of $x$ is $i$ if exactly $i-1$ elements in $A$ are smaller than $x$.
- In particular, the median is the $\lfloor n/2 \rfloor$-smallest.
- To find the median, we could sort and pick $A[\lfloor n/2 \rfloor]$ (taken $O(n \log n)$).
- We can do better.
Median Selection-cont

RS( A, p, r, i) {  
  //Randomize Selection: Returns i’st smallest element in A[p..r].  
  //Assumption: Input is valid and elements are different.
  •If p==r return A[p] 
  •q=PARTITION(A,p,r) ;
  •//Partition using the 5-random element method
  •k=q-p
  •If i==k+1 return A[q]
  •If i<k return RS(A, p, q-1, i ) // Note the difference from QS
  •Else return RS(A, q+1, r, i-k-1)
}  

Time analysis
  • Recall: With high probability, we pick a good pivot:  
    •Not in the 10% smallest or largest:
    •Hence, we get rid of at least 10% of the elements of A
    •So, T(n)=cn+T(0.9 n).
      •T(n)=cn+0.9n+0.9^2n+0.9^3n+... = cn(1+0.9+0.9^2+0.9^3+...)
      •cn(1/(1-0.9)) = O(n).
  •So the expected time is linear. (yuppie)

  As in the case of QS, partitions which are not good are not harmful, just not helpful.