# Quicksort

## 1 Algorithm

Quicksort is an **in-place** sorting algorithm with an average running time of $O(n \log n)$. It operates by successive partitions around a random pivot, which puts the pivot element in the correct, sorted position.

```
Quicksort(A, p, r):
   if n==1: return
   q = Partition(A, p, r)
   Quicksort(A, p, q-1)
   Quicksort(A, q+1, r)
```

For partitioning, it is often best to assume that the pivot is the first element, which means that it will have to be pre-swapped in the main routine. A variable `i` will mark the split between elements (first element larger than pivot).

```
Partition(A, l, r):  //l, r define subarray
   p = A[l]
   i = l+1
   for j in [l+1,...,r]:
      if A[j] < p:
         swap(A, i++, j)
   swap(A, l, i-1)   //Put the pivot in the proper place
```

### 1.1 Duplicate Keys

Quicksort goes quadratic unless partitioning stops on equal keys (imagine a uniform array). This problem can be solved by 3-way partitioning, with the middle third with all of the keys equal to the pivot.

## 2 Analysis

Per partition, there is guaranteed to be $n - 1$ comparisons. The worst-case for the algorithm is that on each run of the partition subroutine, the smallest or largest element is picked. This means that the number of comparisons is

$$\sum_{i=1}^{n-1}(i-1) = \frac{n(n-1)}{2} \tag{1}$$

The best case for the algorithm is picking the median every time. The

recurrence for this case:

$$Q(n) = 2Q\left(\frac{n-1}{2}\right) + (n-1) \tag{2}$$

$$Q(1) = 0 \tag{3}$$

Solving this recurrence gives a comparison-based runtime on the order of $n \lg n$.

### 2.1 Average Case Estimate

A naive way to describe the average case is to say that the partition subroutine gives a 25-75% split on every iteration. The recurrence for this case:

$$Q(n) = Q\left(\frac{n-1}{4}\right) + Q\left(\frac{3(n-1)}{4}\right) + (n-1) \tag{4}$$

$$\approx Q\left(\frac{n}{4}\right) + Q\left(\frac{3n}{4}\right) + (n-1) \tag{5}$$

$$Q(1) = 0 \tag{6}$$

Solve this recurrence by constructive induction, guessing that the solution of the recurrence is bounded by

$$Q(n) \stackrel{?}{\leq} an \lg n \tag{7}$$

Substitute into the recurrence to get the smallest constant value for this solution.

$$Q(n) \approx Q\left(\frac{n}{4}\right) + Q\left(\frac{3n}{4}\right) + (n-1) \tag{8}$$

$$\leq a\frac{n}{4}\lg\frac{n}{4} + a\frac{3n}{4}\lg\frac{3n}{4} + (n-1) \tag{9}$$

$$= \frac{an}{4}(\lg n - \lg 4) + \frac{3an}{4}\left(\lg n - \lg\frac{4}{3}\right) + (n-1) \tag{10}$$

$$= an\lg n + \left(1 - \frac{a}{2} - \frac{3a}{4}\lg\frac{4}{3}\right)n - 1 \tag{11}$$

For the induction to hold, the coefficient of $n$ must be $\leq 0$.

$$1 - \frac{a}{2} - \frac{3a}{4}\lg\frac{4}{3} \leq 0 \tag{12}$$

This gives $a \geq 1.23$, so an estimate for the comparison-based runtime from the naive analysis is $1.23 \lg n$.

## 2.2 Average Case by Induction

When the algorithm partitions on the $q$-th smallest element in the array, the recurrence looks like

$$Q(n) = Q(q-1) + Q(n-q) + (n-1) \tag{13}$$

For a perfectly random partition subroutine or a perfectly shuffled array, it is equally likely to partition on any one of the elements. Therefore, the average case of the algorithm is the sum of comparisons done for all values of $q$ divided by the number of possibilities for $q$.

$$Q(n) = \sum_{q=1}^{n} \frac{1}{n}[Q(q-1) + Q(n-q) + (n-1)] \tag{14}$$

$$= \frac{1}{n}\sum_{q=1}^{n}(Q(q-1) + Q(n-q)) + (n-1) \tag{15}$$

$$= \frac{1}{n}\sum_{q=1}^{n} Q(q-1) + \frac{1}{n}\sum_{q=1}^{n} Q(n-q) + (n-1) \tag{16}$$

A change of variable in this step yields a nicer formula:

$$Q(n) = \frac{2}{n}\sum_{q=0}^{n} Q(q) + (n-1) \tag{17}$$

Now use constructive induction to solve the recurrence/sum. Guess that the solution to the recurrence is $S(n) \leq an \lg n$ (dropped $q = 0$ from the summation because $Q(0) = 0$ and 0 doesn't play nice with logs).

$$S(n) \leq \frac{2}{n}\sum_{q=1}^{n} aq \lg q + (n-1) \tag{18}$$

Approximate an upper bound for the sum with an integral, which is reasonably accurate for asymptotically large values of $n$.

$$\sum_{q=1}^{n} \leq \int_{1}^{n} ax \lg x dx \tag{19}$$

$$= a\int_{1}^{n} x \lg x dx \tag{20}$$

$$= a\left[\frac{x^2 \lg x}{2} - \frac{x^2}{4 \ln 2}\right]_{1}^{n} \tag{21}$$

$$= a\left(\frac{n^2 \lg n}{2} - \frac{n^2}{4 \ln 2} + \frac{1}{4 \ln 2}\right) \tag{22}$$

Substituting into the recurrence:

$$S(n) \leq \frac{2a}{n}\left(\frac{n^2 \lg n}{2} - \frac{n^2}{4\ln 2} + \frac{1}{4\ln 2}\right) + (n-1) \tag{23}$$

$$= an\lg n - \frac{an}{2\ln 2} + \frac{a}{2n\ln 2} + n - 1 \tag{24}$$

$$= an\lg n + \left(1 - \frac{a}{2\ln 2}\right)n - 1 + \frac{a}{2n\ln 2} \tag{25}$$

For the induction to hold (that is, for $S(n) \leq an\lg n$), the coefficient of the $n$ term must be less than or equal to 0. Note the coefficient in front of the $1/n$ term does not matter because $1/n$ will always be some constant less than or equal to 1. This gives $a \geq 2\ln 2$, and because we only care about the smallest upper-bound for the recurrence, $a = 2\ln 2 \approx 1.38$.

A little algebra makes the final average-case solution much nicer:

$$S(n) \leq 2\ln 2 \cdot n\lg n \tag{26}$$

$$= 2\ln 2 \cdot \frac{n\ln n}{\ln 2} \tag{27}$$

$$= 2n\ln n \tag{28}$$

So finally, we can conclude that the average-case comparison-based running time of Quicksort, given a randomly shuffled array or a truly random partition subroutine, is

$$S(n) \leq 2n\ln n \in \Theta(n\lg n) \tag{29}$$

### 2.3 Average Case by Summation

A small leap of intuition yields the same average-case result as the inductive method using only summations. The key initial insight is that given a randomly shuffled list with sorted order

$$x_1 < x_2 < x_3 < \ldots < x_n$$

the probability that $x_1$ is compared to $x_n$ sometime in the Quicksort routine is $2/n$: the pivot in the first step must be either one of those elements or they will never be compared to each other.

Now imagine taking two elements $x_i, x_j$ such that $i < j$. In each step, if $x_p$, the pivot element, is chosen such that $i < p < j$, then the two elements will never be compared. The intuitive leap then is to "chop off" the ends of the array before $i$ and after $j$, which reduces this problem to the one solved

above. That is, given a sub-array with its smallest element as $x_i$ and its largest element as $x_j$, what is the probability that $x_i$ and $x_j$ are compared?

$$P(x_i <=> x_j) = \frac{2}{j-i+1} \tag{30}$$

The average number of comparisons done in Quicksort is then equal to the sum of the probabilities for all $(i, j)$ pairs that $x_i$ is compared to $x_j$.

$$\begin{aligned}
Q(n) &= \sum_{1 \le i < j \le n} \frac{2}{j-i+1} && (31)\\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} && (32)\\
&= 2 \sum_{i=1}^{n} \sum_{j=2}^{n-i+1} \frac{1}{j} && (33)\\
&= 2 \sum_{i=1}^{n} (H_{n-i+1} - 1) && (34)\\
&= 2 \sum_{i=1}^{n} H_{n-i+1} - 2n && (35)\\
&= 2 \sum_{i=1}^{n} H_i - 2n && (36)
\end{aligned}$$

In the above, $H_i$ is the i-th harmonic number (harmonic sum with $n = i$). Substitute Euler's calculation for $H_n$.

$$\begin{aligned}
Q(n) &= 2 \sum_{i=1}^{n} H_i - 2n && (37)\\
&\approx 2 \sum_{i=1}^{n} \left[ \ln i + \gamma + \Theta\left(\frac{1}{i}\right) \right] - 2n && (38)\\
&= 2 \ln n! + 2(\gamma - 1)n + 2 \sum_{i=1}^{n} \Theta\left(\frac{1}{i}\right) && (39)\\
&= 2 \ln n! + 2(\gamma - 1)n + \Theta(\ln n) && (40)\\
& && (41)
\end{aligned}$$

Use Stirling's formula to approximate $n!$:

$$\begin{aligned} Q(n) &= 2\ln n! + 2(\gamma-1)n + \Theta(\ln n) && (42)\\ &\approx 2\ln\left[\left(\frac{n}{e}\right)^n\sqrt{2\pi n}\right] + 2(\gamma-1)n + \Theta(\ln n) && (43)\\ &= 2n\ln n - 2n\ln e + \ln n + 2(\gamma-1)n + \Theta(\ln n) && (44)\\ &= 2n\ln n + 2(\gamma-2)n + \Theta(\ln n) && (45) \end{aligned}$$

The high-order result is the exact same as with the inductive method, but the summation proof yields a few more lower-order terms that are ignored in the analysis anyways.

## 3 Practical Improvements

- Recurse on the smaller sub-array first to guarantee $O(\lg n)$ extra memory on the recursive stack.
- Insertion or selection sort cutoff around 10 elements to avoid the extra overhead associated with recursive calls.
- Estimate true median for partition by taking median of a size 3 sample (median-of-3 method) for medium arrays.
- Tukey's ninther for large arrays: pick 9 items, and take the median of the medians.

## 4 Why Quicksort?

Consider the three major $O(n \lg n)$ sorts: Mergesort, Heapsort, and Quicksort. Quicksort is most often used in library sorting functions because of two reasons.

- In-place: Quicksort uses $O(\lg n)$ space on the stack for recursion, whereas the practical version of Mergesort is not an in-place algorithm.
- Cache efficiency: with modern hardware, when one value is pulled into the cache, some of the values in the surrounding memory are usually pulled into the cache as well. Quicksort takes advantage of this **spatial locality**, while in Heapsort, the elements at the bottom levels are too far away from each other in the array representation.