Lecture Notes, Fundamental Algorithms: Quicksort

The quicksort algorithm is based on a Partition procedure. Partition(A, i, j, k), given as input an array $A[i:k]$ reorders it about a pivot, typically the item initially in location i, i.e. item $e = A[i]$ in the input, so that after the partition:

- $A[j] = e$
- $A[h] \leq A[j]$, for $i \leq h < j$
- $A[\ell] > A[j]$, for $j \leq \ell \leq k$

Sometimes, the partitioning is defined to create three sets of items, the middle set comprising all the items with value e.

Given the Partition procedure, it is easy to give an algorithm Quicksort(A, i, k) which sorts an array $A[i:k]$. If there are one or zero items in A, i.e. $i \geq k$, there is nothing to do; otherwise:

- First partition, i.e. run Partition(A, i, j, k)
- Then recursively sort $A[i:j-1]$ and $A[j+1:k]$

Partition is readily implemented to run in $O(n)$ time in-place, i.e. with only $O(1)$ additional variables, and in particular with no additional array (in contrast to merge sort), though some care is needed to get it right.

A little terminology: $A[j]$ is called the pivot item.

Now we focus on quicksort's running time. If we are unlucky in our choice of pivot, we could end up with $j = i$ or $j = k$, and we then have a recursive problem of size $n-1$ to sort. If this repeats, the total number of instructions executed (the work) is $\Theta(n + (n-1) + (n-2) + \ldots + 1) = \Theta(\frac{1}{2}n(n+1)) = \Theta(n^2)$, which is not very efficient for sorting.

However, the average running time is much better, assuming all the input items are distinct and all input orderings are equally likely. Then, with probability $1/n$, the pivot item is going to be the i-th item in the sorted order. In this case, the recursive problems have sizes $i-1$ and $n-i$. This leads to the following recurrence for the running time, where c' is a suitable constant.

$$
\begin{aligned}
T(n) &\leq c'n + \frac{1}{n}\sum_{i=1}^{n}[T(i-1) + T(n-i)] \quad n > 1 \\
T(1) &\leq c' \\
T(0) &\leq c'
\end{aligned}
$$

Notice that $\sum_{i=1}^{n} T(i-1) = T(0) + \sum_{j=1}^{n-1} T(j) \leq c' + \sum_{i=1}^{n-1} T(i)$; also, on setting $j = n-i$ we obtain $\sum_{i=1}^{n} T(n-i) = \sum_{j=n-1}^{0} T(j) = \sum_{j=0}^{n-1} T(j) = c' + \sum_{i=1}^{n-1} T(i)$. Thus, setting $c \geq c' + 2c'/n$ ($c = 2c'$ suffices as $n \geq 2$ in this case), we can rewrite this recurrence equation as:

$$
\begin{aligned}
T(n) &\leq cn + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \quad n > 1 \\
T(1) &\leq c \\
T(0) &\leq c
\end{aligned}
$$

Rather than solve this recurrence equation directly (which can be done) we are going to obtain an upper bound by a simple argument.

Incidentally, the recurrence equation for the expected number of comparisons, $C(n)$, performed by quicksort is as follows:

$$C(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} C(i) \quad n > 1$$
$$C(1) = 0$$

To the reader: justify this.

To analyze the running time we ask is what is the probability the initial partition results in the *good* case, two recursive subproblems each of size at most $\frac{3}{4}n$? The answer is that this happens if the pivot lies between the $\lceil n/4 \rceil$-th and the $\lfloor 3n/4 \rfloor$-th items. So there are at most $n/2$ pivot choices for which this fails to happen, and hence the probability of the good case is at least $1/2$.

Suppose we are in the bad case. Then there is one subproblem of size more than $3n/4$. Again, the probability that the partitioning of this problem results in the good case, namely that both problems are of size at most $3n/4$, is at least $1/2$. One can iterate this argument.

Thus, to reach the good case, namely that the remaining recursive subproblems all have size at most $3n/4$, requires a first partition for sure, a second partition with probability at most $1/2$, a third partition with probability at most $(1/2)^2$, ..., an $(i+1)$-th partition with probability at most $(1/2)^i$, etc. Each of these partitions takes at most cn work, for some constant $c > 0$. So the expected work (i.e. the average amount of work) is at most $c(n[1+1/2+1/2^2+\ldots]) \leq c(2n) = O(n)$.

To analyze this process using the recursion tree, we view the first level as comprising the recursive calls of size at most $3n/4$ whose parents have size greater than $3n/4$. Suppose there are k of these subproblems, of sizes $n_1, n_2, \ldots, n_k$, resp. As they are disjoint, the total size of these problems is at most n. The expected work in partitioning the problem of size n_i to obtain recursive subproblems of size at most $(3/4)^2 n$ is at most cn_i. (Perhaps $n_i \leq (3/4)^2 n$, in which case no partitioning is needed — the problem already has this smaller size.) Thus the expected total work to partition all these subproblems to obtain recursive subproblems of size at most $(3/4)^2 n$ is at most $c\sum_{i=1}^{k} n_i \leq cn$.

Analogously, we define level j of the tree to comprise the subproblems of size at most $(3/4)^j n$ whose parents have size more than $(3/4)^j n$. Again, the partitioning of these subproblems to obtain the next level of subproblems will take at most cn work in expectation.

The leaf level problems, problems of size 1, will each require at most c work, for cn in total.

How many levels are there in the recursion tree? Remember the successive size bounds on the problem sizes are $n, (3/4)n, (3/4)^2 n, \ldots, (3/4)^i n, \ldots 1$. Thus there are at most $1 + \log_{4/3} n$ levels. We conclude that the overall work is at most $cn(1 + \log_{4/3} n) = O(n \log n)$.

In practice, Quicksort is highly efficient, and is the preferred algorithm for sorting data that fits in main memory.