# CMSC 351 - Introduction to Algorithms
# Spring 2012
# Lecture 11

**Instructor:** MohammadTaghi Hajiaghayi
**Scribe:** Rajesh Chitnis

## 1 Introduction

In this lecture we will look at QuickSort.

## 2 Quicksort

As we have seen in the mergesort, by divide-and-conquer, we could divide the problem into two equal-sized subproblems and solve each problem separately and combine the solutions to get a $O(n \log n)$ algorithm. But we needed to have extra space as it was not an in-place sort.

Quicksort is a divide-and-conquer algorithm which spends most of the effort in the divide step and very little in the conquer step. Suppose we know a number $X$ (called **pivot**) such that one-half of the elements are greater than or equal to $X$ and one-half of the elements are smaller than $X$. We can compare all elements by $n$ comparisons and partition the sequence into equal parts (we can do this without additional space). This is the divide step. We sort each subsequence recursively and the combine step is trivial since the two parts already occupy correct positions in the array and we do not need additional space.

Unfortunately we do not know $X$, but it works no matter which number is selected. However to do it in-place we use two pointers $L$ and $R$, where $L$ points to the left side of the array and $R$ points to the right side of the array. The pointers move towards each other such that:

**Induction Hypothesis:**
At step $k$ of the algorithm, pivot $\geq x_i$ for all $i < L$ and pivot$< x_j$ for all $j > R$.

The base case is trivial and when $L > R$ we are done. Two cases to consider: if either $x_L \leq$ pivot or $x_R >$ pivot we can move one of them and induction

hypothesis is preserved. Otherwise $x_L >$ pivot and $x_R \leq$ pivot. In this case we do a swap and move both of them. The whole partition needs $O(n)$ comparisons.

Divide-and-conquer algorithms work best when the parts have equal sizes, i.e., the pivot should be the median. We can find it also but not in this class. However choosing a random element from the sequence is also a good choice as we see in the analysis. At the end of the partition we put the pivot in its correct place, i.e. do swap($x_{pivot}, x_L$). Once we have the procedure PARTITION, the Quicksort algorithm is as follows:

**Algorithm 1** QUICK-SORT(L, R, X)

```
1: if L < R then
2:    PARTITION(X, L, R);
3:    QUICK-SORT(L,pivot-index - 1, X);
4:    QUICK-SORT(pivot-index + 1, R, X);
5: end if
```

**Running Time:** The running time of quicksort depends on the particular input. If the pivot always partitions into two equal parts then the running time is $T(n) = 2T(\frac{n}{2}) + O(n)$ ; $T(2) = 1$ which gives $T(n) = O(n \log n)$. This is similar to merge sort. However if the pivot is very close to one side of the sequence then the running time can be as bad as $O(n^2)$: say each time the pivot is the smallest element in the sequence, we do $n-1$ comparisons and place only the pivot in the right place, then $n-2$ comparisons, etc. In the algorithm we are choosing the pivot at random, but still there is a chance that we pick the smallest element at thus end up making $O(n^2)$ comparisons. However the chance is very small as we now show.

First each element has the same probability $\frac{1}{n}$ of being picked as the pivot. if the $i^{th}$ element is selected as the pivot, then the running time is $T(n) = (n-1) + T(i-1) + T(n-i)$. Thus the average running time (versus the worst case running time that we have usually considered so far in this class) is:

$$\begin{aligned}
T(n) &= \frac{1}{n}\Big(\sum_{i=1}^{n}(n-1) + T(i-1) + T(n-i)\Big) \\
&= (n-1) + \frac{1}{n}\Big(\sum_{i=1}^{n} T(i-1) + T(n-i)\Big) \\
&= (n-1) + \frac{1}{n}\sum_{i=1}^{n} T(i-1) + \frac{1}{n}\sum_{i=1}^{n} T(n-i) \\
&= (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} T(i)
\end{aligned}$$

As we have seen in the recurrence relations with full history this gives $T(n) = O(n \log n)$. In practice Quicksort is really fast and deserves the name.

# 3 Lower Bound for Sorting

We could improve the running time from $O(n^2)$ (insertion or selection sort) to $O(n \log n)$ (merge sort or quicksort). Can we do still better for comparison based (and not radix sort, for example). Unfortunately the answer is NO. We can use information theory to show that indeed any algorithm needs at least $\Omega(n \log n)$ comparisons. See the proof in the book [1] if you are interested.

# 4 Maximum and Minimum Elements

**The Problem:** Find the maximum and minimum elements in a given array of distinct elements.

The straightforward solution is $2n-3$ comparisons: $n-1$ for max and $n-2$ for min. Can we do better? If we try to do inductively from $n$ to $n-1$ we end up paying $2n$. But what about if we go from $n$ to $n-2$? Assume we know the solution for $n-2$. Consider $x_{n-1}$ and $x_n$ and say we know Max and Min so far. First we compare $x_{n-1}$ and $x_n$ and then compare the max of them with Max and the min of them with Min. So by 3 comparisons we update both Max and Min. Overall we need $\frac{3n}{2}$ comparisons. Can we do better by considering $x_{n-2}, x_{n-1}, x_n$? The answer is NO by any method.

# 5 The Order Statistic or Selection Problem

**The Problem:** Given a sequence $S = x_1, x_2, \ldots, x_n$ of elements, and integer $k$ such that $1 \leq k \leq n$, find the $k^{th}$-smallest element in $S$.

If $k$ is very close to 1 or $n$, then we can run the algorithm for max or min $k$ times. So the running time is $O(kn)$. We can also sort the sequence in $O(n \log n)$ time and find the $k^{th}$ element in $O(n)$ time. So if $O(kn) > O(n \log n)$, i.e., $k = \Omega(\log n)$ then sorting is faster.

But can we do better? YES. We can divide-and-conquer the same way as quicksort. In quicksort, the sequence is partitioned by a pivot into two subsequences. Here we need only to determine which subsequence contains the $k^{th}$ element and then continue recursively only for that subsequence. The rest of the elements can be ignored.

**Running Time:** As in Quicksort, the worst case is $O(n^2)$ but we can show the average running time is $O(n)$. In practice it is very fast even though there are some algorithms with running time $O(n)$ in worst case. Indeed this algorithm is the best for finding only the median, i.e., $k = \frac{n}{2}$.

# 6 Extra Reading

Read Sections 6.10 and 6.11 from the book [1] yourself. Due to lack of time we cannot cover them in the class. But there are very cute algorithms there and you will enjoy reading them.

# References

[1] Udi Manber, *Introduction to Algorithms - A Creative Approach*