Illinois Institute of Technology
Department of Computer Science

# Lecture 5: September 10, 2014

CS 430 Introduction to Algorithms
Fall Semester, 2014

## 1 More on quicksort: other efficiency metrics

In the previous lecture we arrived at an expression for the efficiency of quicksort by counting the number of comparisons. There are a number of other metrics we can use, though.

### 1.1 Exchanges

For instance, we can count the number of exchanges that take place. If we view the partition algorithm as a sequence of exchanges,[1] we can count the number of exchanges that are performed on the array.
In the worst case, $\lfloor \frac{n}{2} \rfloor$ exchanges take place at each level of the recursion. We have the recurrence

$$E(n) = E\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + E\left(\left\lceil \frac{n}{2} \right\rceil\right) + \left\lfloor \frac{n}{2} \right\rfloor$$

where $E(0) = E(1) = 0$. It turns out that this recurrence has a nice solution:

$$E(n) = \sum_{i=0}^{n-1} \nu(i)$$

where $\nu(i)$ is the number of 1-bits when $i$ is written in binary, clearly no greater than $\lceil \lg i \rceil$. Dividing by $n$,

$$\frac{E(n)}{n} = \frac{1}{n} \sum_{i=0}^{n-1} \nu(i)$$

which is the average number of 1-bits in $i$ written in binary.[2] This is approximately $\frac{1}{2} \lg n$, so we have

$$\begin{aligned} \frac{E(n)}{n} &\approx \frac{1}{2} \lg n \\ E(n) &\approx \frac{1}{2} n \lg n \\ &= \Theta(n \log n) \end{aligned}$$

### 1.2 Stack depth

The expected depth of the recursion stack is somewhat simpler. Although the algorithm makes two recursive calls, the two calls are not made at the same time. This yields a recurrence of

$$S(n) = 1 + \frac{1}{n} \sum_{i=0}^{n-1} S(i)$$

[1] The PARTITION algorithm given in section 7.1 of CLRS uses this approach.
[2] This average has been studied extensively by H. DeLange in "Sur la fonction sommatoire de la fonction somme des chiffres," *Enseignment Mathématique* **21**, pp. 31–47 (1975).

which can be solved as we solved the (more complex) running-time recurrences for quicksort, giving

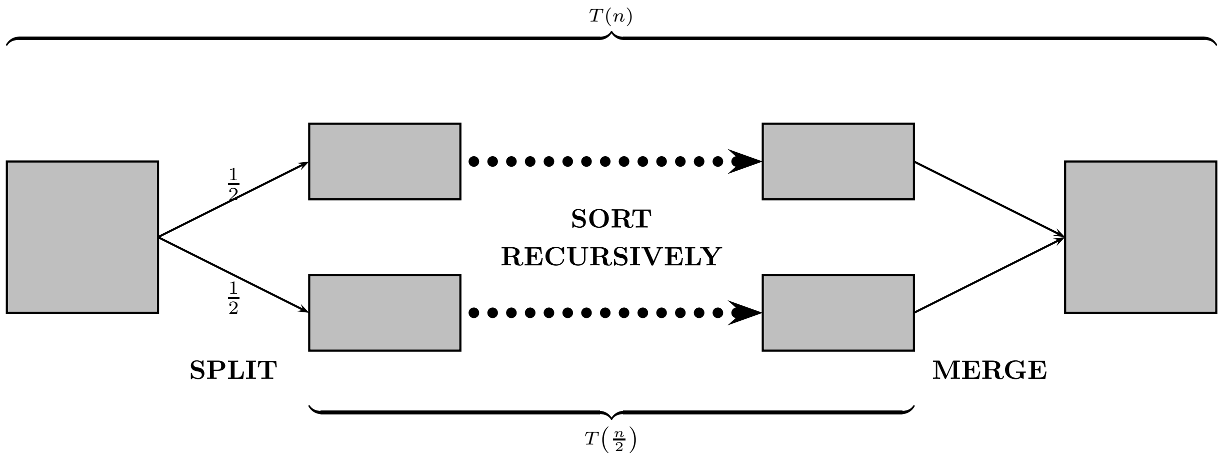
$$S(n) = O(\log n)$$

## 2 Divide and conquer

The divide-and-conquer paradigm is used often in the formulation of algorithms. A divide-and-conquer algorithm will typically involve splitting a problem into smaller components (*divide*), solving the problem on those components (*conquer*), and combining the results in some meaningful fashion (*combine*).

### 2.1 Mergesort

For instance, to sort an array with the mergesort algorithm, we divide it in half, sort each half recursively, and merge the two halves:



A recurrence for the running time of mergesort is

$$M(n) = M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + M\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$$

where $M(0) = M(1) = 0$. It turns out that this yields

$$\begin{aligned} M(n) &= \sum_{k=1}^{n} \lceil \lg k \rceil \\ &= n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1 \end{aligned}$$

Since it is impossible to sort an array in fewer than $\lg n! = \sum_{k=1}^{n} \lg k$ comparisons, the running time of mergesort is less than $n$ from optimal. Unfortunately, the complicated data movement involved yields an algorithm that is not competitive in real life.

### 2.2 Selection

The selection problem can be stated as follows: Given $x_1, x_2, \ldots, x_n$, find the $k$th largest element. The lower bound on the running time of this problem is linear.

### 2.2.1 A naive approach

We can take a simple approach to this problem. How can we solve the problem if $k = 1$? That is, how can we find the maximum element? We can make one pass through the array, keeping track of the largest element seen so far.[3] This requires $n - 1$ comparisons.

What if $k = 2$? We can first find the maximum as outlined above, eliminate it from the array, and find the maximum of the new array. This can be done in $(n-1) + (n-2) = 2n - 3$ comparisons.

We can use selection to find the median: solve the problem for $k = \frac{n}{2}$. How does that perform? Not well: it requires $(n-1) + (n-2) + \cdots + \left(n - \frac{n}{2}\right) \approx \frac{n^2}{2}$ comparisons. This is $\Theta(n^2)$—we could do better by sorting in $\Theta(n \log n)$ time and indexing at $\frac{n}{2}$. But can we do better than that?

### 2.2.2 Selection in $n \log n$ time

Yes. Consider a tournament in which we would like to determine the two best players. If we used an approach like the above algorithm, we would first find the best player and then repeat the entire set of matches, with the champion eliminated, to determine the next best player. In fact, we need only compare the direct losers to the champion.

We can use this idea in the selection problem. We proceed as before for $k = 1$. For $k = 2$, we then need examine only $\lg n$ additional elements, totalling $n + \lg n$. For $k = 3$, we examine $\lg n$ more than that, for a total of $n + 2 \lg n$. For $k = \frac{n}{2}$, we must examine $n + \frac{n}{2} \lg n$. Unfortunately, this is $\Theta(n \log n)$—better than the naive solution, but no better than sorting, and significantly more difficult to understand.

### 2.2.3 Selection in expected linear time

The approach in (randomized) quicksort is to partition the array based on some random element and to then recursively run quicksort on each of the two partitions. Here, though, say the partition element is the $i$th element of the array and that we are searching for the $k$th element. If $k = i$, we have found the solution. If $k < i$, we only need to recursively search the smaller partition. If $k > i$, we only need to recursively search the larger partition (for the $(k - i)$th element).

The worst-case running time is $\Theta(n^2)$: if we are, say, searching for the minimum, but the partition routine partitions around the maximum each time, we reach this upper bound.

Fortunately, the average-case running time is far better. We arrive at the recurrence

$$c(n) = n + 1 + \frac{1}{n} \sum_{i=0}^{n-1} c(i)$$

As before, multiplying both sides by $n$ yields

$$nc(n) = n(n+1) + \sum_{i=0}^{n-1} c(i) \tag{1}$$

Substituting $n - 1$ for $n$ gives

$$(n-1)c(n-1) = n(n-1) + \sum_{i=0}^{n-2} c(i) \tag{2}$$

[3]This algorithm is treated more thoroughly in section 9.1 of CLRS.

Subtracting (2) from (1),

$$\begin{aligned} nc(n) - (n-1)c(n-1) &= 2n + c(n-1) \\ nc(n) - nc(n-1) &= 2n \\ c(n) - c(n-1) &= 2 \\ c(n) &= c(n-1) + 2 \\ &= 2n \\ &= \Theta(n) \end{aligned}$$

### 2.2.4 Selection in worst-case linear time

If we can somehow guarantee the location of a good partitioning element, we can reduce the running time to $\Theta(n)$ worst-case. The approach is as follows:[4]

- Arrange the elements of the array in columns of five. (If the number of elements is not a multiple of five, the last column will have fewer than five elements.) There will be $\lfloor \frac{n}{5} \rfloor$ such columns. This can be done in linear time.
- Use any sorting technique to sort each column. Sorting five elements takes constant time; since there are $\lfloor \frac{n}{5} \rfloor$ columns to sort, this step takes linear time.
- Notice that the third element in each column of five is its median. Recursively find the median of those $\lfloor \frac{n}{5} \rfloor$ numbers. Call it $x$.
- Partition on $x$.

What do we know about the median of medians $x$? There are clearly $\frac{n}{10}$ medians of columns that exceed $x$. Furthermore, in each of those columns, at least two additional elements must exceed $x$ as well. So at least $\frac{3n}{10}$ elements exceed $x$. Likewise, $\frac{n}{10}$ medians of columns are less than $x$, and at least two additional elements of their columns must also be less than $x$—so at least $\frac{3n}{10}$ are less than $x$. These observations limit how "off center" our partition element can be.

If we use this algorithm, then *in the worst case* we make the recursive call on an array of size $\frac{7n}{10}$. This yields the recurrence

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$$

We can use induction to prove that this recurrence grows linearly[5]. This is a reasonable guess, since the recurrence is of the form $T(n) = cn + T(\alpha n) + T(\beta n)$, where $\alpha + \beta < 1$, so we expect the $cn$ term to dominate.[6] Say that $T(n) \leq Kn$. Then we want

$$\begin{aligned} cn + \frac{K}{5}n + \frac{7K}{10}n &\leq Kn \\ n\left(c + \frac{9}{10}K\right) &\leq Kn \\ c + \frac{9}{10}K &\leq K \end{aligned}$$

This holds for sufficiently large $K$.

[4]This is presented nicely in section 9.3 of CLRS.

[5]This is the "substitution method" from Section 4.3 of CLRS.

[6]This holds because of the convexity of the function $T(n) = cn$. Section 4.5 of CLRS, the proof of the master theorem, formalizes this intuition.

### 2.2.5 Quicksort in worst-case $\Theta(n \log n)$ time

We can use this approach for the partition routine in the quicksort algorithm to yield a worst-case $\Theta(n \log n)$ running time. Unfortunately, the additional labor involved causes a very large constant factor "hidden" in the $\Theta$ notation, yielding an algorithm that is not useful in practice. Nevertheless, this approach to worst-case linear-time selection is useful and has many applications.