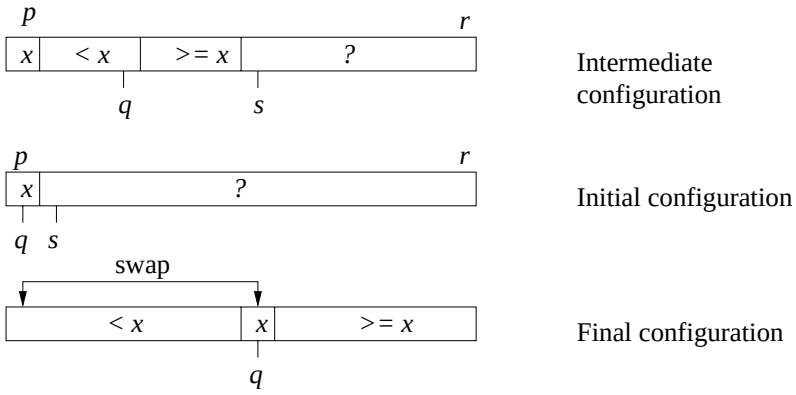

Figure 14: Partitioning intermediate structure.

all of the elements have been processed. To finish things off we swap $A[p]$ (the pivot) with $A[q]$, and return the value of q. Here is the complete code:

Partition

```
Partition(int p, int r, array A) {              // 3-way partition of A[p..r]
    x = A[p]                                    // pivot item in A[p]
    q = p
    for s = p+1 to r do {
        if (A[s] < x) {
            q = q+1
            swap A[q] with A[s]
        }
    }
    swap A[p] with A[q]                        // put the pivot into final position
    return q                                   // return location of pivot
}
```

An example is shown below.

Lecture 15: QuickSort

(Tuesday, Mar 17, 1998)

Revised: March 18. Fixed a bug in the analysis.

Read: Chapt 8 in CLR. My presentation and analysis are somewhat different than the text's.

QuickSort and Randomized Algorithms: Early in the semester we discussed the fact that we usually study the worst-case running times of algorithms, but sometimes average-case is a more meaningful measure. Today we will study QuickSort. It is a worst-case $\Theta(n^2)$ algorithm, whose expected-case running time is $\Theta(n \log n)$.

We will present QuickSort as a *randomized* algorithm, that is, an algorithm which makes random choices. There are two common types of randomized algorithms:

Monte Carlo algorithms: These algorithms may produce the wrong result, but the probability of this occurring can be made arbitrarily small by the user. Usually the lower you make this probability, the longer the algorithm takes to run.

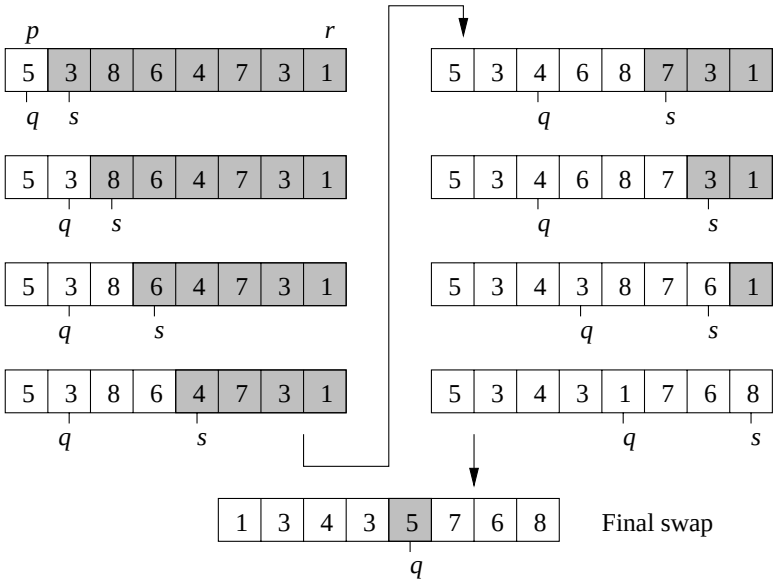

Figure 15: Partitioning example.

Las Vegas algorithms: These algorithms always produce the correct result, but the running time is a random variable. In these cases the expected running time, averaged over all possible random choices is the measure of the algorithm's running time.

The most well known Monte Carlo algorithm is one for determining whether a number is prime. This is an important problem in cryptography. The QuickSort algorithm that we will discuss today is an example of a Las Vegas algorithm. Note that QuickSort does not need to be implemented as a randomized algorithm, but as we shall see, this is generally considered the safest implementation.

QuickSort Overview: QuickSort is also based on the divide-and-conquer design paradigm. Unlike MergeSort where most of the work is done after the recursive call returns, in QuickSort the work is done before the recursive call is made. Here is an overview of QuickSort. Note the similarity with the selection algorithm, which we discussed earlier. Let $A[p..r]$ be the (sub)array to be sorted. The initial call is to $A[1..n]$.

Basis: If the list contains 0 or 1 elements, then return.

Select pivot: Select a random element x from the array, called the *pivot*.

Partition: Partition the array in three subarrays, those elements $A[1..q-1] \leq x$, $A[q] = x$, and $A[q+1..n] \geq x$.

Recurse: Recursively sort $A[1..q-1]$ and $A[q+1..n]$.

The pseudocode for QuickSort is given below. The initial call is QuickSort(1, n, A). The Partition routine was discussed last time. Recall that Partition assumes that the pivot is stored in the first element of A. Since we want a random pivot, we pick a random index i from p to r, and then swap $A[i]$ with $A[p]$.

QuickSort

```
QuickSort(int p, int r, array A) {              // Sort A[p..r]
    if (r <= p) return                          // 0 or 1 items, return
    i = a random index from [p..r]              // pick a random element
```

```
        swap A[i] with A[p]                    // swap pivot into A[p]
        q = Partition(p, r, A)                 // partition A about pivot
        QuickSort(p, q-1, A)                   // sort A[p..q-1]
        QuickSort(q+1, r, A)                   // sort A[q+1..r]
    }
```

QuickSort Analysis: The correctness of QuickSort should be pretty obvious. However its analysis is not so obvious. It turns out that the running time of QuickSort depends heavily on how good a job we do in selecting the pivot. In particular, if the rank of the pivot (recall that this means its position in the final sorted list) is very large or very small, then the partition will be unbalanced. We will see that unbalanced partitions (like unbalanced binary trees) are bad, and result is poor running times. However, if the rank of the pivot is anywhere near the middle portion of the array, then the split will be reasonably well balanced, and the overall running time will be good. Since the pivot is chosen at random by our algorithm, we may do well most of the time and poorly occasionally. We will see that the expected running time is $O(n \log n)$.

Worst-case Analysis: Let's begin by considering the worst-case performance, because it is easier than the average case. Since this is a recursive program, it is natural to use a recurrence to describe its running time. But unlike MergeSort, where we had control over the sizes of the recursive calls, here we do not. It depends on how the pivot is chosen. Suppose that we are sorting an array of size n, $A[1..n]$, and further suppose that the pivot that we select is of rank q, for some q in the range 1 to n. It takes $\Theta(n)$ time to do the partitioning and other overhead, and we make two recursive calls. The first is to the subarray $A[1..q-1]$ which has $q-1$ elements, and the other is to the subarray $A[q+1..n]$ which has $r-(q+1)+1 = r-q$ elements. So if we ignore the Θ (as usual) we get the recurrence:

$$T(n) = T(q-1) + T(n-q) + n.$$

This depends on the value of q. To get the worst case, we maximize over all possible values of q. As a basis we have that $T(0) = T(1) = \Theta(1)$. Putting this together we have

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ \max_{1 \le q \le n}(T(q-1) + T(n-q) + n) & \text{otherwise.} \end{cases}$$

Recurrences that have max's and min's embedded in them are very messy to solve. The key is determining which value of q gives the maximum. (A rule of thumb of algorithm analysis is that the worst cases tends to happen either at the extremes or in the middle. So I would plug in the value $q = 1$, $q = n$, and $q = n/2$ and work each out.) In this case, the worst case happens at either of the extremes (but see the book for a more careful analysis based on an analysis of the second derivative). If we expand the recurrence in the case $q = 1$ we get:

$$\begin{aligned} T(n) &\le T(0) + T(n-1) + n \;=\; 1 + T(n-1) + n \\ &= T(n-1) + (n+1) \\ &= T(n-2) + n + (n+1) \\ &= T(n-3) + (n-1) + n + (n+1) \\ &= T(n-4) + (n-2) + (n-1) + n + (n+1) \\ &= \ldots \\ &= T(n-k) + \sum_{i=-1}^{k-2} (n-i). \end{aligned}$$

For the basis, $T(1) = 1$ we set $k = n - 1$ and get

$$\begin{aligned} T(n) &\leq T(1) + \sum_{i=-1}^{n-3} (n-i) \\ &= 1 + (3 + 4 + 5 + \ldots + (n-1) + n + (n+1)) \\ &\leq \sum_{i=1}^{n+1} i \;=\; \frac{(n+1)(n+2)}{2} \in O(n^2). \end{aligned}$$

In fact, a more careful analysis reveals that it is $\Theta(n^2)$ in this case.

Average-case Analysis: Next we show that in the average case QuickSort runs in $\Theta(n \log n)$ time. When we talked about average-case analysis at the beginning of the semester, we said that it depends on some assumption about the distribution of inputs. However, in this case, the analysis does not depend on the input distribution at all—it only depends on the random choices that the algorithm makes. This is good, because it means that the analysis of the algorithm's performance is the same for all inputs. In this case the average is computed over all possible random choices that the algorithm might make for the choice of the pivot index in the second step of the QuickSort procedure above.

To analyze the average running time, we let $T(n)$ denote the average running time of QuickSort on a list of size n. It will simplify the analysis to assume that all of the elements are distinct. The algorithm has n random choices for the pivot element, and each choice has an equal probability of $1/n$ of occuring. So we can modify the above recurrence to compute an average rather than a max, giving:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \frac{1}{n} \sum_{q=1}^{n} (T(q-1) + T(n-q) + n) & \text{otherwise.} \end{cases}$$

This is not a standard recurrence, so we cannot apply the Master Theorem. Expansion is possible, but rather tricky. Instead, we will attempt a constructive induction to solve it. We know that we want a $\Theta(n \log n)$ running time, so let's try $T(n) \leq an \lg n + b$. (Properly we should write $\lceil \lg n \rceil$ because unlike MergeSort, we cannot assume that the recursive calls will be made on array sizes that are powers of 2, but we'll be sloppy because things will be messy enough anyway.)

Theorem: There exist a constant c such that $T(n) \leq cn \ln n$, for all $n \geq 2$. (Notice that we have replaced $\lg n$ with $\ln n$. This has been done to make the proof easier, as we shall see.)

Proof: The proof is by constructive induction on n. For the basis case $n = 2$ we have

$$\begin{aligned} T(2) &= \frac{1}{2} \sum_{q=1}^{2} (T(q-1) + T(2-q) + 2) \\ &= \frac{1}{2}((T(0) + T(1) + 2) + (T(1) + T(0) + 2) \;=\; \frac{8}{2} \;=\; 4. \end{aligned}$$

We want this to be at most $c2 \ln 2$ implying that $c \geq 4/(2 \ln 2) \approx 2.885$.

For the induction step, we assume that $n \geq 3$, and the induction hypothesis is that for any $n' < n$, we have $T(n') \leq cn' \ln n'$. We want to prove it is true for $T(n)$. By expanding the definition of $T(n)$, and moving the factor of n outside the sum we have:

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{q=1}^{n} (T(q-1) + T(n-q) + n) \\ &= \frac{1}{n} \sum_{q=1}^{n} (T(q-1) + T(n-q)) + n. \end{aligned}$$

Observe that if we split the sum into two sums, they both add the same values $T(0) + T(1) + \ldots + T(n-1)$, just that one counts up and the other counts down. Thus we can replace this with $2\sum_{q=0}^{n-1} T(q)$. Because they don't follow the formula, we'll extract $T(0)$ and $T(1)$ and treat them specially. If we make this substitution and apply the induction hypothesis to the remaining sum we have (which we can because $q < n$) we have

$$\begin{aligned} T(n) &= \frac{2}{n}\left(\sum_{q=0}^{n-1} T(q)\right) + n = \frac{2}{n}\left(T(0) + T(1) + \sum_{q=2}^{n-1} T(q)\right) + n \\ &\leq \frac{2}{n}\left(1 + 1 + \sum_{q=2}^{n-1}(cq \lg q)\right) + n \\ &= \frac{2c}{n}\left(\sum_{q=2}^{n-1}(cq \ln q)\right) + n + \frac{4}{n}. \end{aligned}$$

We have never seen this sum before. Later we will show that

$$S(n) = \sum_{q=2}^{n-1} q \ln q \leq \frac{n^2}{2}\ln n - \frac{n^2}{4}.$$

Assuming this for now, we have

$$\begin{aligned} T(n) &= \frac{2c}{n}\left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right) + n + \frac{4}{n} \\ &= cn\ln n - \frac{cn}{2} + n + \frac{4}{n} \\ &= cn\ln n + n\left(1 - \frac{c}{2}\right) + \frac{4}{n}. \end{aligned}$$

To finish the proof, we want all of this to be at most $cn \ln n$. If we cancel the common $cn \ln n$ we see that this will be true if we select c such that

$$n\left(1 - \frac{c}{2}\right) + \frac{4}{n} \leq 0.$$

After some simple manipulations we see that this is equivalent to:

$$\begin{aligned} 0 &\geq n - \frac{cn}{2} + \frac{4}{n} \\ \frac{cn}{2} &\geq n + \frac{4}{n} \\ c &\geq 2 + \frac{8}{n^2}. \end{aligned}$$

Since $n \geq 3$, we only need to select c so that $c \geq 2 + \frac{8}{9}$, and so selecting $c = 3$ will work. From the basis case we have $c \geq 2.885$, so we may choose $c = 3$ to satisfy both the constraints.

The Leftover Sum: The only missing element to the proof is dealing with the sum

$$S(n) = \sum_{q=2}^{n-1} q \ln q.$$

To bound this, recall the integration formula for bounding summations (which we paraphrase here). For any monotonically increasing function $f(x)$

$$\sum_{i=a}^{b-1} f(i) \leq \int_a^b f(x)dx.$$

The function $f(x) = x \ln x$ is monotonically increasing, and so we have

$$S(n) \leq \int_2^n x \ln x dx.$$

If you are a calculus macho man, then you can integrate this by parts, and if you are a calculus wimp (like me) then you can look it up in a book of integrals

$$\int_2^n x \ln x dx = \frac{x^2}{2} \ln x - \frac{x^2}{4}\Big|_{x=2}^{n} = \left(\frac{n^2}{2} \ln n - \frac{n^2}{4}\right) - (2 \ln 2 - 1) \leq \frac{n^2}{2} \ln n - \frac{n^2}{4}.$$

This completes the summation bound, and hence the entire proof.

Summary: So even though the worst-case running time of QuickSort is $\Theta(n^2)$, the average-case running time is $\Theta(n \log n)$. Although we did not show it, it turns out that this doesn't just happen much of the time. For large values of n, the running time is $\Theta(n \log n)$ with high probability. In order to get $\Theta(n^2)$ time the algorithm must make poor choices for the pivot at virtually every step. Poor choices are rare, and so continuously making poor choices are very rare. You might ask, could we make QuickSort deterministic $\Theta(n \log n)$ by calling the selection algorithm to use the median as the pivot. The answer is that this would work, but the resulting algorithm would be so slow practically that no one would ever use it.

QuickSort (like MergeSort) is not formally an in-place sorting algorithm, because it does make use of a recursion stack. In MergeSort and in the expected case for QuickSort, the size of the stack is $O(\log n)$, so this is not really a problem.

QuickSort is the most popular algorithm for implementation because its actual performance (on typical modern architectures) is so good. The reason for this stems from the fact that (unlike Heapsort) which can make large jumps around in the array, the main work in QuickSort (in partitioning) spends most of its time accessing elements that are close to one another. The reason it tends to outperform MergeSort (which also has good locality of reference) is that most comparisons are made against the pivot element, which can be stored in a register. In MergeSort we are always comparing two array elements against each other. The most efficient versions of QuickSort uses the recursion for large subarrays, but once the sizes of the subarray falls below some minimum size (e.g. 20) it switches to a simple iterative algorithm, such as selection sort.

Lecture 16: Lower Bounds for Sorting

(Thursday, Mar 19, 1998)

Read: Chapt. 9 of CLR.

Review of Sorting: So far we have seen a number of algorithms for sorting a list of numbers in ascending order. Recall that an *in-place* sorting algorithm is one that uses no additional array storage (however, we allow QuickSort to be called in-place even though they need a stack of size $O(\log n)$ for keeping track of the recursion). A sorting algorithm is *stable* if duplicate elements remain in the same relative position after sorting.