Finally, in the recurrence $T(n) = 4T(n/3) + n$ (which corresponds to Case 1), most of the work is done at the leaf level of the recursion tree. This can be seen if you perform iteration on this recurrence, the resulting summation is

$$n \sum_{i=0}^{\log_3 n} \left(\frac{4}{3}\right)^i .$$

(You might try this to see if you get the same result.) Since $4/3 > 1$, as we go deeper into the levels of the tree, that is deeper into the summation, the terms are growing successively larger. The largest contribution will be from the leaf level.

Lecture 9: Medians and Selection

(Tuesday, Feb 24, 1998)

Read: Todays material is covered in Sections 10.2 and 10.3. You are not responsible for the randomized analysis of Section 10.2. Our presentation of the partitioning algorithm and analysis are somewhat different from the ones in the book.

Selection: In the last couple of lectures we have discussed recurrences and the divide-and-conquer method of solving problems. Today we will give a rather surprising (and very tricky) algorithm which shows the power of these techniques.

The problem that we will consider is very easy to state, but surprisingly difficult to solve optimally. Suppose that you are given a set of n numbers. Define the *rank* of an element to be one plus the number of elements that are smaller than this element. Since duplicate elements make our life more complex (by creating multiple elements of the same rank), we will make the simplifying assumption that all the elements are distinct for now. It will be easy to get around this assumption later. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank n.

Of particular interest in statistics is the *median*. If n is odd then the median is defined to be the element of rank $(n + 1)/2$. When n is even there are two natural choices, namely the elements of ranks $n/2$ and $(n/2) + 1$. In statistics it is common to return the average of these two elements. We will define the median to be either of these elements.

Medians are useful as measures of the *central tendency* of a set, especially when the distribution of values is highly skewed. For example, the median income in a community is likely to be more meaningful measure of the central tendency than the average is, since if Bill Gates lives in your community then his gigantic income may significantly bias the average, whereas it cannot have a significant influence on the median. They are also useful, since in divide-and-conquer applications, it is often desirable to partition a set about its median value, into two sets of roughly equal size. Today we will focus on the following generalization, called the *selection problem*.

Selection: Given a set A of n distinct numbers and an integer k, $1 \leq k \leq n$, output the element of A of rank k.

The selection problem can easily be solved in $\Theta(n \log n)$ time, simply by sorting the numbers of A, and then returning $A[k]$. The question is whether it is possible to do better. In particular, is it possible to solve this problem in $\Theta(n)$ time? We will see that the answer is yes, and the solution is far from obvious.

The Sieve Technique: The reason for introducing this algorithm is that it illustrates a very important special case of divide-and-conquer, which I call the *sieve technique*. We think of divide-and-conquer as breaking the problem into a small number of smaller subproblems, which are then solved recursively. The sieve technique is a special case, where the number of subproblems is just 1.

The sieve technique works in phases as follows. It applies to problems where we are interested in finding a single item from a larger set of n items. We do not know which item is of interest, however after doing some amount of analysis of the data, taking say $\Theta(n^k)$ time, for some constant k, we find that we do not know what the desired item is, but we can identify a large enough number of elements that *cannot* be the desired value, and can be eliminated from further consideration. In particular "large enough" means that the number of items is at least some fixed constant fraction of n (e.g. $n/2$, $n/3$, $0.0001n$). Then we solve the problem recursively on whatever items remain. Each of the resulting recursive solutions then do the same thing, eliminating a constant fraction of the remaining set.

Applying the Sieve to Selection: To see more concretely how the sieve technique works, let us apply it to the selection problem. Recall that we are given an array $A[1..n]$ and an integer k, and want to find the k-th smallest element of A. Since the algorithm will be applied inductively, we will assume that we are given a subarray $A[p..r]$ as we did in MergeSort, and we want to find the kth smallest item (where $k \le r - p + 1$). The initial call will be to the entire array $A[1..n]$.

There are two principal algorithms for solving the selection problem, but they differ only in one step, which involves judiciously choosing an item from the array, called the *pivot element*, which we will denote by x. Later we will see how to choose x, but for now just think of it as a random element of A. We then partition A into three parts. $A[q]$ contains the element x, subarray $A[p..q-1]$ will contain all the elements that are less than x, and $A[q+1..r]$, will contain all the element that are greater than x. (Recall that we assumed that all the elements are distinct.) Within each subarray, the items may appear in any order. This is illustrated below.

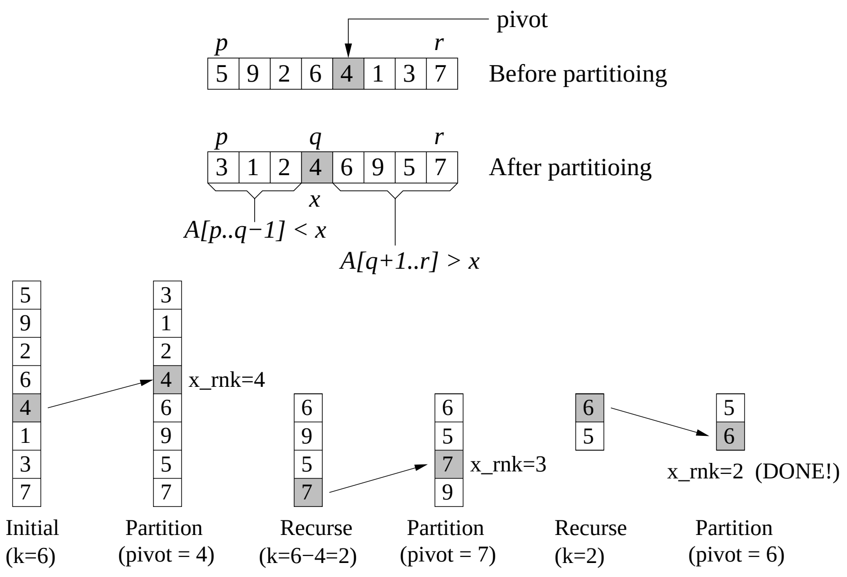

Figure 7: Selection Algorithm.

It is easy to see that the rank of the pivot x is $q-p+1$ in $A[p..r]$. Let $x\_rnk = q-p+1$. If $k = x\_rnk$, then the pivot is the kth smallest, and we may just return it. If $k < x\_rnk$, then we know that we need to recursively search in $A[p..q-1]$ and if $k > x\_rnk$ then we need to recursively search $A[q+1..r]$. In this latter case we have eliminated q smaller elements, so we want to find the element of rank $k-q$. Here is the complete pseudocode.

Selection

```
Select(array A, int p, int r, int k) {      // return kth smallest of A[p..r]
    if (p == r) return A[p]                 // only 1 item left, return it
```

```
        else {
            x = Choose_Pivot(A, p, r)           // choose the pivot element
            q = Partition(A, p, r, x)           // partition <A[p..q-1], x, A[q+1..r]>
            x_rnk = q - p + 1                   // rank of the pivot
            if (k == x_rnk) return x            // the pivot is the kth smallest
            else if (k < x_rnk)
                return Select(A, p, q-1, k)     // select from left subarray
            else
                return Select(A, q+1, r, k-x_rnk)// select from right subarray
        }
    }
```

Notice that this algorithm satisfies the basic form of a sieve algorithm. It analyzes the data (by choosing the pivot element and partitioning) and it eliminates some part of the data set, and recurses on the rest. When $k = x\_rnk$ then we get lucky and eliminate everything. Otherwise we either eliminate the pivot and the right subarray or the pivot and the left subarray.

We will discuss the details of choosing the pivot and partitioning later, but assume for now that they both take $\Theta(n)$ time. The question that remains is how many elements did we succeed in eliminating? If x is the largest or smallest element in the array, then we may only succeed in eliminating one element with each phase. In fact, if x is one of the smallest elements of A or one of the largest, then we get into trouble, because we may only eliminate it and the few smaller or larger elements of A. Ideally x should have a rank that is neither too large nor too small.

Let us suppose for now (optimistically) that we are able to design the procedure `Choose_Pivot` in such a way that is eliminates exactly half the array with each phase, meaning that we recurse on the remaining $n/2$ elements. This would lead to the following recurrence.

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n/2) + n & \text{otherwise.} \end{cases}$$

We can solve this either by expansion (iteration) or the Master Theorem. If we expand this recurrence level by level we see that we get the summation

$$T(n) \;=\; n + \frac{n}{2} + \frac{n}{4} + \cdots \;\le\; \sum_{i=0}^{\infty} \frac{n}{2^i} \;=\; n \sum_{i=0}^{\infty} \frac{1}{2^i}.$$

Recall the formula for the infinite geometric series. For any c such that $|c| < 1$, $\sum_{i=0}^{\infty} c^i = 1/(1-c)$. Using this we have

$$T(n) \le 2n \in O(n).$$

(This only proves the upper bound on the running time, but it is easy to see that it takes at least $\Omega(n)$ time, so the total running time is $\Theta(n)$.)

This is a bit counterintuitive. Normally you would think that in order to design a $\Theta(n)$ time algorithm you could only make a single, or perhaps a constant number of passes over the data set. In this algorithm we make many passes (it could be as many as $\lg n$). However, because we eliminate a constant fraction of elements with each phase, we get this convergent geometric series in the analysis, which shows that the total running time is indeed linear in n. This lesson is well worth remembering. It is often possible to achieve running times in ways that you would not expect.

Note that the assumption of eliminating half was not critical. If we eliminated even one per cent, then the recurrence would have been $T(n) = T(99n/100) + n$, and we would have gotten a geometric series involving $99/100$, which is still less than 1, implying a convergent series. Eliminating *any* constant fraction would have been good enough.

Choosing the Pivot: There are two issues that we have left unresolved. The first is how to choose the pivot element, and the second is how to partition the array. Both need to be solved in $\Theta(n)$ time. The second problem is a rather easy programming exercise. Later, when we discuss QuickSort, we will discuss partitioning in detail.

For the rest of the lecture, let's concentrate on how to choose the pivot. Recall that before we said that we might think of the pivot as a random element of A. Actually this is not such a bad idea. Let's see why.

The key is that we want the procedure to eliminate at least some constant fraction of the array after each partitioning step. Let's consider the top of the recurrence, when we are given $A[1..n]$. Suppose that the pivot x turns out to be of rank q in the array. The partitioning algorithm will split the array into $A[1..q-1] < x$, $A[q] = x$ and $A[q+1..n] > x$. If $k = q$, then we are done. Otherwise, we need to search one of the two subarrays. They are of sizes $q-1$ and $n-q$, respectively. The subarray that contains the kth smallest element will generally depend on what k is, so in the worst case, k will be chosen so that we have to recurse on the larger of the two subarrays. Thus if $q > n/2$, then we may have to recurse on the left subarray of size $q-1$, and if $q < n/2$, then we may have to recurse on the right subarray of size $n-q$. In either case, we are in trouble if q is very small, or if q is very large.

If we could select q so that it is roughly of middle rank, then we will be in good shape. For example, if $n/4 \le q \le 3n/4$, then the larger subarray will never be larger than $3n/4$. Earlier we said that we might think of the pivot as a random element of the array A. Actually this works pretty well in practice. The reason is that roughly half of the elements lie between ranks $n/4$ and $3n/4$, so picking a random element as the pivot will succeed about half the time to eliminate at least $n/4$. Of course, we might be continuously unlucky, but a careful analysis will show that the expected running time is still $\Theta(n)$. We will return to this later.

Instead, we will describe a rather complicated method for computing a pivot element that achieves the desired properties. Recall that we are given an array $A[1..n]$, and we want to compute an element x whose rank is (roughly) between $n/4$ and $3n/4$. We will have to describe this algorithm at a very high level, since the details are rather involved. Here is the description for Select_Pivot:

Groups of 5: Partition A into groups of 5 elements, e.g. $A[1..5]$, $A[6..10]$, $A[11..15]$, etc. There will be exactly $m = \lceil n/5 \rceil$ such groups (the last one might have fewer than 5 elements). This can easily be done in $\Theta(n)$ time.

Group medians: Compute the median of each group of 5. There will be m group medians. We do not need an intelligent algorithm to do this, since each group has only a constant number of elements. For example, we could just BubbleSort each group and take the middle element. Each will take $\Theta(1)$ time, and repeating this $\lceil n/5 \rceil$ times will give a total running time of $\Theta(n)$. Copy the group medians to a new array B.

Median of medians: Compute the median of the group medians. For this, we will have to call the selection algorithm recursively on B, e.g. `Select(B, 1, m, k)`, where $m = \lceil n/5 \rceil$, and $k = \lfloor (m+1)/2 \rfloor$. Let x be this median of medians. Return x as the desired pivot.

The algorithm is illustrated in the figure below. To establish the correctness of this procedure, we need to argue that x satisfies the desired rank properties.

Lemma: The element x is of rank at least $n/4$ and at most $3n/4$ in A.

Proof: We will show that x is of rank at least $n/4$. The other part of the proof is essentially symmetrical. To do this, we need to show that there are at least $n/4$ elements that are less than or equal to x. This is a bit complicated, due to the floor and ceiling arithmetic, so to simplify things we will assume that n is evenly divisible by 5. Consider the groups shown in the tabular form above. Observe that at least half of the group medians are less than or equal to x. (Because x is

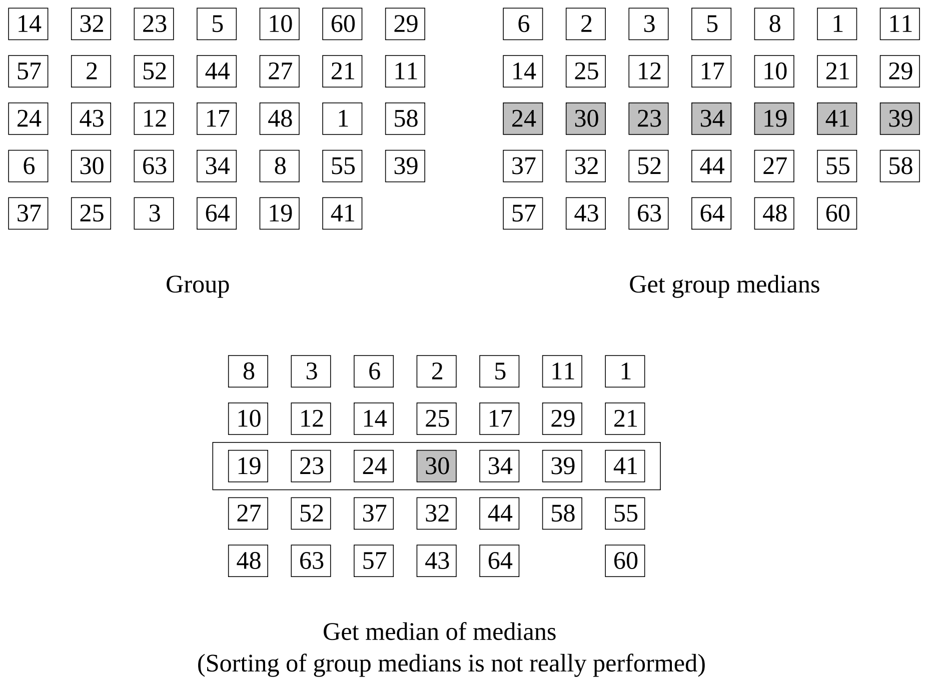

Figure 8: Choosing the Pivot. 30 is the final pivot.

their median.) And for each group median, there are three elements that are less than or equal to this median within its group (because it is the median of its group). Therefore, there are at least $3((n/5)/2 = 3n/10 \geq n/4$ elements that are less than or equal to x in the entire array.

Analysis: The last order of business is to analyze the running time of the overall algorithm. We achieved the main goal, namely that of eliminating a constant fraction (at least $1/4$) of the remaining list at each stage of the algorithm. The recursive call in `Select()` will be made to list no larger than $3n/4$. However, in order to achieve this, within `Select_Pivot()` we needed to make a recursive call to `Select()` on an array B consisting of $\lceil n/5 \rceil$ elements. Everything else took only $\Theta(n)$ time. As usual, we will ignore floors and ceilings, and write the $\Theta(n)$ as n for concreteness. The running time is

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ T(n/5) + T(3n/4) + n & \text{otherwise.} \end{cases}$$

This is a very strange recurrence because it involves a mixture of different fractions ($n/5$ and $3n/4$). This mixture will make it impossible to use the Master Theorem, and difficult to apply iteration. However, this is a good place to apply constructive induction. We know we want an algorithm that runs in $\Theta(n)$ time.

Theorem: There is a constant c, such that $T(n) \leq cn$.

Proof: (by strong induction on n)

Basis: ($n = 1$) In this case we have $T(n) = 1$, and so $T(n) \leq cn$ as long as $c \geq 1$.

Step: We assume that $T(n') \leq cn'$ for all $n' < n$. We will then show that $T(n) \leq cn$. By definition we have

$$T(n) = T(n/5) + T(3n/4) + n.$$

Since $n/5$ and $3n/4$ are both less than n, we can apply the induction hypothesis, giving

$$\begin{aligned} T(n) &\leq c\frac{n}{5} + c\frac{3n}{4} + n \;=\; cn\left(\frac{1}{5}+\frac{3}{4}\right) + n \\ &= cn\frac{19}{20} + n \;=\; n\left(\frac{19c}{20}+1\right). \end{aligned}$$

This last expression will be $\leq cn$, provided that we select c such that $c \geq (19c/20) + 1$. Solving for c we see that this is true provided that $c \geq 20$.

Combining the constraints that $c \geq 1$, and $c \geq 20$, we see that by letting $c = 20$, we are done.

A natural question is why did we pick groups of 5? If you look at the proof above, you will see that it works for any value that is strictly greater than 4. (You might try it replacing the 5 with 3, 4, or 6 and see what happens.)

Lecture 10: Long Integer Multiplication

(Thursday, Feb 26, 1998)

Read: Todays material on integer multiplication is not covered in CLR.

Office hours: The TA, Kyongil, will have extra office hours on Monday before the midterm, from 1:00-2:00. I'll have office hours from 2:00-4:00 on Monday.

Long Integer Multiplication: The following little algorithm shows a bit more about the surprising applications of divide-and-conquer. The problem that we want to consider is how to perform arithmetic on long integers, and multiplication in particular. The reason for doing arithmetic on long numbers stems from cryptography. Most techniques for encryption are based on number-theoretic techniques. For example, the character string to be encrypted is converted into a sequence of numbers, and encryption keys are stored as long integers. Efficient encryption and decryption depends on being able to perform arithmetic on long numbers, typically containing hundreds of digits.

Addition and subtraction on large numbers is relatively easy. If n is the number of digits, then these algorithms run in $\Theta(n)$ time. (Go back and analyze your solution to the problem on Homework 1). But the standard algorithm for multiplication runs in $\Theta(n^2)$ time, which can be quite costly when lots of long multiplications are needed.

This raises the question of whether there is a more efficient way to multiply two very large numbers. It would seem surprising if there were, since for centuries people have used the same algorithm that we all learn in grade school. In fact, we will see that it is possible.

Divide-and-Conquer Algorithm: We know the basic grade-school algorithm for multiplication. We normally think of this algorithm as applying on a digit-by-digit basis, but if we partition an n digit number into two "super digits" with roughly $n/2$ each into longer sequences, the same multiplication rule still applies.

To avoid complicating things with floors and ceilings, let's just assume that the number of digits n is a power of 2. Let A and B be the two numbers to multiply. Let $A[0]$ denote the least significant digit and let $A[n-1]$ denote the most significant digit of A. Because of the way we write numbers, it is more natural to think of the elements of A as being indexed in decreasing order from left to right as $A[n-1..0]$ rather than the usual $A[0..n-1]$.

Let $m = n/2$. Let

$$\begin{aligned} w &= A[n-1..m] \qquad & x &= A[m-1..0] \quad \text{and} \\ y &= B[n-1..m] \qquad & z &= B[m-1..0]. \end{aligned}$$