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

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

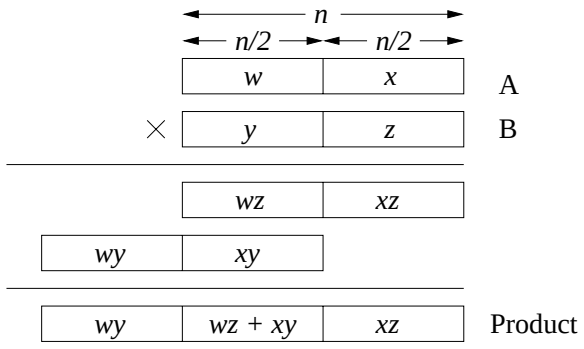

Figure 9: Long integer multiplication.

If we think of w, x, y and z as $n/2$ digit numbers, we can express A and B as

$$\begin{aligned} A &= w \cdot 10^m + x \\ B &= y \cdot 10^m + z, \end{aligned}$$

and their product is

$$mult(A, B) = mult(w, y)10^{2m} + (mult(w, z) + mult(x, y))10^m + mult(x, z).$$

The operation of multiplying by 10^m should be thought of as simply shifting the number over by m positions to the right, and so is not really a multiplication. Observe that all the additions involve numbers involving roughly $n/2$ digits, and so they take $\Theta(n)$ time each. Thus, we can express the multiplication of two long integers as the result of 4 products on integers of roughly half the length of the original, and a constant number of additions and shifts, each taking $\Theta(n)$ time. This suggests that if we were to implement this algorithm, its running time would be given by the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 4T(n/2) + n & \text{otherwise.} \end{cases}$$

If we apply the Master Theorem, we see that $a = 4$, $b = 2$, $k = 1$, and $a > b^k$, implying that Case 1 holds and the running time is $\Theta(n^{\lg 4}) = \Theta(n^2)$. Unfortunately, this is no better than the standard algorithm.

Faster Divide-and-Conquer Algorithm: Even though the above exercise appears to have gotten us nowhere, it actually has given us an important insight. It shows that the critical element is the number of multiplications on numbers of size $n/2$. The number of additions (as long as it is a constant) does not affect the running time. So, if we could find a way to arrive at the same result algebraically, but by trading off multiplications in favor of additions, then we would have a more efficient algorithm. (Of course, we cannot simulate multiplication through repeated additions, since the number of additions must be a constant, independent of n.)

The key turns out to be a algebraic "trick". The quantities that we need to compute are $C = wy$, $D = xz$, and $E = (wz + xy)$. Above, it took us four multiplications to compute these. However, observe that if instead we compute the following quantities, we can get everything we want, using only three multiplications (but with more additions and subtractions).

$$\begin{aligned} C &= mult(w, y) \\ D &= mult(x, z) \\ E &= mult((w + x), (y + z)) - C - D \;=\; (wy + wz + xy + xz) - wy - xz \;=\; (wz + xy). \end{aligned}$$

Finally we have

$$mult(A, B) = C \cdot 10^{2m} + E \cdot 10^m + D.$$

Altogether we perform 3 multiplications, 4 additions, and 2 subtractions all of numbers with $n/2$ digitis. We still need to shift the terms into their proper final positions. The additions, subtractions, and shifts take $\Theta(n)$ time in total. So the total running time is given by the recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(n/2) + n & \text{otherwise.} \end{cases}$$

Now when we apply the Master Theorem, we have $a = 3$, $b = 2$ and $k = 1$, yielding $T(n) \in \Theta(n^{\lg 3}) \approx \Theta(n^{1.585})$.

Is this really an improvement? This algorithm carries a larger constant factor because of the overhead of recursion and the additional arithmetic operations. But asymptotics says that if n is large enough, then this algorithm will be superior. For example, if we assume that the clever algorithm has overheads that are 5 times greater than the simple algorithm (e.g. $5n^{1.585}$ versus n^2) then this algorithm beats the simple algorithm for $n \geq 50$. If the overhead was 10 times larger, then the crossover would occur for $n \geq 260$.

Review for the Midterm: Here is a list topics and readings for the first midterm exam. Generally you are responsible for anything discussed in class, and anything appearing on homeworks. It is a good idea to check out related chapters in the book, because this is where I often look for ideas on problems.

Worst-case, Average-case: Recall that a worst-case means that we consider the highest running time over all inputs of size n, average case means that we average running times over all inputs of size n (and generally weighting each input by its probability of occuring). (Chapt 1 of CLR.)

General analysis methods: Be sure you understand the induction proofs given in class and on the homeworks. Also be sure you understand how the constructive induction proofs worked.

Summations: Write down (and practice recognizing) the basic formulas for summations. These include the arithmetic series $\sum_i i$, the quadratic series, $\sum_i i^2$, the geometric series $\sum_i x^i$, and the harmonic series $\sum_i 1/i$. Practice with simplifying summations. For example, be sure that you can take something like

$$\sum_i 3^i \left(\frac{n}{2^i}\right)^2$$

and simplify it to a geometric series

$$n^2 \sum_i (3/4)^i.$$

Also be sure you can apply the integration rule to summations. (Chapt. 3 of CLR.)

Asymptotics: Know the formal definitions for Θ, O, and Ω, as well as how to use the limit-rule. Know the what the other forms, o and ω, mean informally. There are a number of good sample problems in the book. I'll be happy to check any of your answers. Also be able to rank functions in asymptotic order. For example which is larger $\lg \sqrt{n}$ or $\sqrt{\lg n}$? (It is the former, can you see why?) Remember the following rule and know how to use it.

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \qquad\qquad \lim_{n\to\infty} \frac{\lg^b n}{n^c} = 0.$$

(Chapt. 2 of CLR.)

Recurrences: Know how to analyze the running time of a recursive program by expressing it as a recurrence. Review the basic techniques for solving recurrences: guess and verify by induction (I'll provide any guesses that you need on the exam), constructive induction, iteration, and the (simplified) Master Theorem. (You are NOT responsible for the more complex version given in the text.) (Chapt 4, Skip 4.4.)

Divide-and-conquer: Understand how to design algorithms by divide-and-conquer. Understand the divide-and-conquer algorithm for MergeSort, and be able to work an example by hand. Also understand how the sieve technique works, and how it was used in the selection problem. (Chapt 10 on Medians; skip the randomized analysis. The material on the 2-d maxima and long integer multiplication is not discussed in CLR.)

Lecture 11: First Midterm Exam

(Tuesday, March 3, 1998)

First midterm exam today. No lecture.

Lecture 12: Heaps and HeapSort

(Thursday, Mar 5, 1998)

Read: Chapt 7 in CLR.

Sorting: For the next series of lectures we will focus on sorting algorithms. The reasons for studying sorting algorithms in details are twofold. First, sorting is a very important algorithmic problem. Procedures for sorting are parts of many large software systems, either explicitly or implicitly. Thus the design of efficient sorting algorithms is important for the overall efficiency of these systems. The other reason is more pedagogical. There are many sorting algorithms, some slow and some fast. Some possess certain desirable properties, and others do not. Finally sorting is one of the few problems where there provable lower bounds on how fast you can sort. Thus, sorting forms an interesting case study in algorithm theory.

In the sorting problem we are given an array $A[1..n]$ of n numbers, and are asked to reorder these elements into increasing order. More generally, A is of an array of records, and we choose one of these records as the *key value* on which the elements will be sorted. The key value need not be a number. It can be any object from a *totally ordered* domain. Totally ordered means that for any two elements of the domain, x, and y, either $x < y$, $x =$, or $x > y$.

There are some domains that can be partially ordered, but not totally ordered. For example, sets can be partially ordered under the subset relation, $\subset$, but this is not a total order, it is not true that for any two sets either $x \subset y$, $x = y$ or $x \supset y$. There is an algorithm called *topological sorting* which can be applied to "sort" partially ordered sets. We may discuss this later.

Slow Sorting Algorithms: There are a number of well-known slow sorting algorithms. These include the following:

Bubblesort: Scan the array. Whenever two consecutive items are found that are out of order, swap them. Repeat until all consecutive items are in order.

Insertion sort: Assume that $A[1..i-1]$ have already been sorted. Insert $A[i]$ into its proper position in this subarray, by shifting all larger elements to the right by one to make space for the new item.

Selection sort: Assume that $A[1..i-1]$ contain the $i-1$ smallest elements in sorted order. Find the smallest element in $A[i..n]$, and then swap it with $A[i]$.

These algorithms are all easy to implement, but they run in $\Theta(n^2)$ time in the worst case. We have already seen that MergeSort sorts an array of numbers in $\Theta(n \log n)$ time. We will study two others, HeapSort and QuickSort.