

Lecture #4:

0.0.1 Divide & Conquer Method:

We now take up the concept of divide and conquer. We give several examples in what follows. The basic idea is to divide up the problem into smaller problems and recursively solve each and combine the solutions to make up the solution of the larger problem. Some times, this reduces the computational effort and it is these cases that we are interested in. The first few examples from the first lecture that you have most likely seen before this class were: (i) Binary Search; (ii) Merge Sort; and (iii) The Tiling Problem from lecture #1. Now we do a few more.

Example 1 Given an array of numbers $A[1, 2, \dots, n]$, find the maximum and the minimum of the set of the given numbers using only pairwise comparisons. We assume that n is a power of 2.

Algorithm 2 (A) Starting with both Max and Min set to $A[1]$, we compare each element in the array to the current values of these and if $A[i] > Max$, change Max to $A[i]$; if $A[i] < Min$, change Min to $A[i]$. We need to do the second comparison only if the first does not change the value of Max .

Algorithm 3 (A') First find Max and this takes $(n - 1)$ comparisons. Now find Min among the **remaining** elements and this takes $(n - 2)$ comparisons for a total of $2n - 3$ comparisons.

What is the (worst case) number of comparisons in Algorithm A? . It should be clear that any algorithm for finding the Max is $\Omega(n)$. Thus, the above algorithms are "optimal" as regards the "order". But may be there is another algorithm that is also $O(n)$ but the constant is lower. This is indeed the case as we shall see below.

Algorithm 4 (B) Consider the relation:

$$\begin{aligned} Max(A[1, 2, \dots, n]) &= \max\{Max(A[1, 2, \dots, \lfloor \frac{n}{2} \rfloor]), Max(A[\lfloor \frac{n}{2} \rfloor + 1, \dots, n])\} \\ Min(A[1, 2, \dots, n]) &= \min\{Min(A[1, 2, \dots, \lfloor \frac{n}{2} \rfloor]), Min(A[\lfloor \frac{n}{2} \rfloor + 1, \dots, n])\} \end{aligned}$$

So if we know the results for the two half arrays, by doing two more comparisons, we get the results for the full array. This gives the following recursion:

$$t(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ t(\lfloor \frac{n}{2} \rfloor) + t(\lceil \frac{n}{2} \rceil) + 2 & \text{if } n \geq 3 \end{cases}$$

We use the iteration method to solve this recursion (Why not use the Master Theorem?). Since we have assumed that n is a power of 2, let $n = 2^k$ for some k . Then,

$$\begin{aligned} t(n) &= 2 + 2t\left(\frac{n}{2}\right) \\ &= 2 + 2\left[2 + 2t\left(\frac{n}{4}\right)\right] \\ &= 2 + 4 + 8 + 16 + \dots \\ &= 2^{k-1}t(2) + \sum_{i=1}^{k-1} 2^i \end{aligned}$$

$$\begin{aligned} \sum_{i=1}^{k-1} 2^i &= 2 + 4 + 8 + \dots + 2^{k-1} \\ &= 2[1 + 2 + 4 + \dots + 2^{k-2}] \\ &= 2(2^{k-1} - 1) = 2^k - 2 \end{aligned}$$

Therefore, $t(n) = 2^k + 2^{k-1} - 2 = n + \frac{n}{2} - 2 = \frac{3}{2}n - 2 < 2n - 3$.

Exercise 5 Show by substitution method that for general values of n , $t(n) = \lceil \frac{3}{2}n \rceil - 2$.

Algorithm B does less work than Algorithms A'. The savings are only about 25% and this may be offset by the difficulty in implementing algorithm B. There is a much simpler algorithm that achieves the same bound as algorithm B.

Algorithm 6 (C) Define two arrays $B[1, 2, \dots, \frac{n}{2}]$ and $C[1, 2, \dots, \frac{n}{2}]$ of half the length of the array A as follows: $B[i] = \max(A[2i-1], A[2i]); C[i] = \min(A[2i-1], A[2i]); 1 \leq i \leq \frac{n}{2}$. Find the maximum of the numbers in the array B and the minimum of the numbers in the array C . The total effort here is also $\lceil \frac{3}{2}n - 2 \rceil$, but the algorithms is much simpler!

But we are really looking for much bigger savings! Let's move on to other examples.

Example 7 Fast Integer Multiplication

Let $X = x_1x_2x_3\dots x_n$ and $Y = y_1y_2y_3\dots y_n$ be two n -digit numbers that we wish to multiply. The ordinary method of computing the product $X \cdot Y$ involves n^2 multiplication of single digit numbers $y_i (1 \leq i \leq n)$ with $x_j (1 \leq j \leq n)$. Let us consider divide and conquer to develop a faster algorithm. Divide both X and Y into two numbers with half the number of digits in the original numbers. So we get:

$$\begin{aligned} X_1 &= x_1x_2x_3\dots x_{\frac{n}{2}} \\ X_2 &= x_{\frac{n}{2}+1}, x_{\frac{n}{2}+2}, \dots, x_n \\ Y_1 &= y_1y_2y_3\dots y_{\frac{n}{2}} \\ Y_2 &= y_{\frac{n}{2}+1}y_{\frac{n}{2}+2}\dots y_n \end{aligned}$$

We assume that n is a power of two for the sake of convenience. Thus,

$$\begin{aligned} X &= X_1 10^{\frac{n}{2}} + X_2 \\ Y &= Y_1 10^{\frac{n}{2}} + Y_2 \end{aligned}$$

and

$$X \cdot Y = X_1 \cdot Y_1 10^n + [X_1 \cdot Y_2 + X_2 \cdot Y_1] 10^{\frac{n}{2}} + X_2 \cdot Y_2$$

The last expression involves four products of integers of half the size compared to the original problem. The number of steps for adding these terms together and for doing the left shifting to get $X_1 \cdot Y_1 10^n$ from $X_1 \cdot Y_1$ is cn for some positive c . Hence, we obtain the following recurrence:

$$t(n) = 4t\left(\frac{n}{2}\right) + cn$$

If we use the master theorem to solve this recurrence, $a = 4, b = 2; \log_b a = 2; f(n) = \Theta(n)$ and so we are in Case 1. Hence $t(n) = \Theta(n^2)$. This is no improvement! If you look carefully, we need only to compute three terms: $X_1 \cdot Y_1, [X_1 \cdot Y_2 + X_2 \cdot Y_1]$, and $X_2 \cdot Y_2$. How we do it is up to us!. Notice

$$[X_1 \cdot Y_2 + X_2 \cdot Y_1] = [X_1 + X_2] \cdot [Y_1 + Y_2] - X_1 \cdot Y_1 - X_2 \cdot Y_2$$

Hence, if we compute the **three** products $X_1 \cdot Y_1, [X_1 + X_2] \cdot [Y_1 + Y_2]$ and $X_2 \cdot Y_2$ and two additions (subtractions actually) above, we get all the required terms in $X \cdot Y$. With this, our recurrence becomes

$$t(n) = 3t\left(\frac{n}{2}\right) + cn$$

Now we apply the master theorem, (again Case 1) we get $t(n) = \Theta(n^{\lg 3})$ which is better than before. We have indeed conquered! In order to see whether we can do even better (for example can we split the numbers into three equal parts and do the same thing) , we need to understand why this worked. For this purpose, let

$$\begin{aligned} X(t) &= X_1 \cdot t + X_2 \\ Y(t) &= Y_1 \cdot t + Y_2 \end{aligned}$$

$$\begin{aligned} Z(t) &= X(t) \cdot Y(t) \\ &= X_1 \cdot Y_1 t^2 + [X_1 \cdot Y_2 + X_2 \cdot Y_1]t + X_2 \cdot Y_2 \\ &= at^2 + bt + c \end{aligned}$$

Thus, $Z(t)$ is a second degree polynomial in t and has three coefficients a, b , and c . If we know the values of $Z(t)$ for three different values of t then we can

calculate the coefficients a, b , and c . The three values of t are normally chosen as $-1, 0$, and 1 . We use La Grange's formula:

$$\begin{aligned}
Z(t) &= \sum_{i=-1}^1 \prod_{j \neq i; j=-1}^1 \left[\frac{(t-j)}{(i-j)} Z(i) \right] \\
&= \left\{ \frac{t}{-1} \cdot \frac{t-1}{-1-1} Z(-1) \right\} + \left\{ \frac{t+1}{1} \cdot \frac{t-1}{-1} Z(0) \right\} + \left\{ \frac{t+1}{2} \cdot \frac{t}{1} Z(1) \right\} \\
&= \frac{t^2-t}{2} Z(-1) + (-t^2+1) Z(0) + \frac{t^2+t}{2} Z(1) \\
&= \left[\frac{1}{2} Z(-1) - Z(0) + \frac{1}{2} Z(1) \right] t^2 + \left[-\frac{1}{2} Z(-1) + \frac{1}{2} Z(1) \right] t + Z(0) \\
&= at^2 + bt + c
\end{aligned}$$

The three values $Z(-1), Z(0)$, and $Z(1)$ are computed by three multiplications on integers half the size of the original numbers.

$$\begin{aligned}
Z(-1) &= X(-1) \cdot Y(-1) \\
Z(0) &= X(0) \cdot Y(0) \\
Z(1) &= X(1) \cdot Y(1)
\end{aligned}$$

Note that

$$\begin{aligned}
Z(-1) &= X(-1) \cdot Y(-1) \\
&= [-X_1 + X_2] \cdot [-Y_1 + Y_2] \\
&= X_1 \cdot Y_1 - [X_1 \cdot Y_2 + X_2 \cdot Y_1] + X_2 \cdot Y_2 \\
Z(0) &= X(0) \cdot Y(0) \\
&= X_2 \cdot Y_2 \\
Z(1) &= X(1) \cdot Y(1) \\
&= [X_1 + X_2] \cdot [Y_1 + Y_2]
\end{aligned}$$

If we let $t = 10^{\frac{n}{2}}$, we get $Z = X \cdot Y = a10^n + b10^{\frac{n}{2}} + c$.

You can do the same process by dividing each number into three parts. the analysis is similar. In general, doing this we get $t(n) = \Theta(n^{\log_k(2k-1)})$ and we can make this $\Theta(n^{1+\epsilon})$ for arbitrary positive ϵ by letting k become large. This completes this example. What we have indirectly shown is how to multiply two polynomial functions.

In the above discussion as applied to multiplying two integers of equal size, we have been some what sloppy. $X(1) + X(2)$ may have more digits than each of these numbers. What if the size is not a power of 2; for example if it is odd? If we assume that $t(n)$ is an increasing function of n , (for large n), then we can solve the resulting recurrence:

$$\begin{aligned}
t(n) &= t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + t\left(\left\lceil \frac{n}{2} \right\rceil\right) + t\left(1 + \left\lceil \frac{n}{2} \right\rceil\right) + cn \\
&\leq 3t\left(1 + \left\lceil \frac{n}{2} \right\rceil\right) + cn
\end{aligned}$$

Now if we let $\hat{t}(n) = t(n+2)$, we can transform the above into:

$$\begin{aligned}\hat{t}(n) &= t(n+2) \leq 3t\left(1 + \left\lceil \frac{n+2}{2} \right\rceil\right) + c(n+2) \\ &\leq 3t\left(2 + \left\lceil \frac{n}{2} \right\rceil\right) + 2cn \quad \text{since } (n+2) \leq 2n \\ &= 3\hat{t}\left(\left\lceil \frac{n}{2} \right\rceil\right) + dn\end{aligned}$$

Hence $\hat{t}(n) = O(n^{\lg 3})$. Since $t(n)$ is increasing, $t(n) \leq t(n+2) = \hat{t}(n)$.
Hence $t(n) = O(n^{\lg 3})$.