

Learning Outcomes

COMP202
Complexity of Algorithms
Fundamental Solution Techniques
Divide & Conquer
[See Chapter 5 in Goodrich and Tamassia.]

At the conclusion of this set of lecture notes, you should:

1. Understand the general idea of Divide-and-Conquer algorithms.
2. Be able to utilize the “Master Method” that may be used to derive closed form expressions for (some) recurrence relations.
3. Be familiar with some of the classical Divide-and-Conquer algorithms.

Divide-and-Conquer

We have already discussed the *Divide-and-Conquer method* when we talked about sorting.

MergeSort is a classical Divide-and-Conquer algorithm, and, QuickSort is very much like a Divide-and-Conquer algorithm (although it doesn't divide the problem into equal-sized subproblems).

The problem we discussed about counting inversions in permutations used a Divide-and-Conquer method to solve it (namely, a modification of the MergeSort algorithm).

Divide-and-Conquer (cont.)

To remind you, here is the general outline for using this method:

- ▶ *Divide*: If the input size is *small* then solve directly, otherwise divide the input data into two or more *disjoint* subsets.
- ▶ *Recur*: *Recursively* solve the sub-problems associated with the subsets.
- ▶ *Conquer*: Take the *solutions* to the *sub-problems* and merge them into a solution to the original problem.

Divide-and-Conquer (cont.)

To *analyze* the running time of a Divide-and-Conquer algorithm we typically utilize a *recurrence relation*, where

- ▶ $T(n)$ denotes the running time on an input of size n .

We then want to characterize $T(n)$ using an equation that relates $T(n)$ to values of function T for problem sizes smaller than n .

Divide-and-Conquer (cont.)

For another example, the running time of the binary search method (on sorted arrays) can be described as

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ T(n/2) + c & \text{otherwise} \end{cases}$$

where the constant c here represents the work needed to determine the midpoint of the sublist and do the comparison to determine if the sought for value has been found (or updating the bottom/top index to determine the next sublist to search).

Divide-and-Conquer (cont.)

For example, the running time of the MergeSort algorithm can be written in this form:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

where c is a small constant that represents how much work is done for the comparisons for merging the lists, etc.

(Note: Strictly speaking this should be written as

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn & \text{otherwise} \end{cases}$$

but in terms of the *asymptotic running time*, the “floors” and “ceilings” are unnecessary, and this can be formally proven.)

Substitution Method

One way to solve a *Divide-and-Conquer* recurrence equation is to use the *iterative substitution method*, a.k.a. “plug-and-chug”, e.g.

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

Assuming that n is a power of 2 (for simplicity) we get

$$T(n) = 2 \left[2T(n/2^2) + c(n/2) \right] + cn = 2^2 T(n/2^2) + 2cn.$$

After i iterations we have

$$T(n) = 2^i T(n/2^i) + icn.$$

For $i = \log_2 n$, we get

$$T(n) = 2^{\log_2 n} \cdot c + cn \log_2 n = cn + cn \log_2 n = \Theta(n \log n).$$

Divide-and-Conquer style recurrence relations

We might also “guess” a solution to a divide-and-conquer style recurrence relation, and then formally prove this using the method of *mathematical induction* (a method you discussed in COMP 108).

Instead of pursuing this avenue, let us consider the more general case, where

$$T(n) = \begin{cases} c & \text{if } n \leq d \\ aT(n/b) + f(n) & \text{if } n > d \end{cases}$$

for some constants a, b, c , and d , and some function $f(n)$.

Divide-and-Conquer style recurrence relations (cont.)

With a careful analysis of the previous expression, we can make some conclusions about the (asymptotic) form of $T(n)$.

For example, if $f(n)$ is “smaller” than $n^{\log_b a}$ (in a suitable fashion), then the first term is going to dominate the summation, and we will have $T(n) = \Theta(n^{\log_b a})$.

If, on the other hand, we have $n^{\log_b a}$ is “smaller” than $f(n)$ (and the function f is “nice”), then the summation will dominate this expression and we will have $T(n) = \Theta(f(n))$.

And there’s a third case to consider too...

Divide-and-Conquer style recurrence relations (cont.)

We can “unwind” this recurrence relation in this manner:

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ &= a \left[aT(n/b^2) + f(n/b) \right] + f(n) \\ &= a^2 T(n/b^2) + af(n/b) + f(n) \\ &= a^3 T(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \\ &= a^4 T(n/b^4) + a^3 f(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \\ &= \dots \\ &= a^{\log_b n} T(1) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \\ &= n^{\log_b a} T(1) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j). \end{aligned}$$

(We use $a^{\log_b n} = n^{\log_b a}$ to get the final line.)

The Master Method

We make the following claim:

Theorem

Suppose that $T(n)$ and $f(n)$ satisfy the recurrence relation defined previously (for some constants a, b, c , and d).

Then

1. If there is a constant $\varepsilon > 0$ such that $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$.
2. If there is a constant $k \geq 0$ such that $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$.
3. If there are constants $\varepsilon > 0$ and $\delta < 1$ such that $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$ and $a f(n/b) \leq \delta f(n)$ for $n \geq d$, then $T(n)$ is $\Theta(f(n))$.

Examples using the Master Method

Consider these examples:

1. $T(n) = 4T(n/2) + n$.

In this case we have $n^{\log_b a} = n^{\log_2 4} = n^2$. Therefore we are in Case 1, since $f(n) = n = O(n^{2-\varepsilon})$ for $\varepsilon = 1$. Thus, we conclude that $T(n)$ is $\Theta(n^2)$ by the Master Method.

2. $T(n) = 2T(n/2) + cn$. (This is the MergeSort type of recurrence relation.)

Here we have $n^{\log_b a} = n^{\log_2 2} = n^1 = n$. Hence we are in Case 2 with $k = 0$ (why?), so that $T(n)$ is $\Theta(n \log n)$.

3. $T(n) = 2T(n/2) + n \log n$.

As before, we have $\log_2 2 = 1$, so $n^{\log_b a} = n$. We are also in Case 2, with $k = 1$ as $f(n)$ is $\Theta(n \log n)$.

Therefore, we conclude $T(n)$ is $\Theta(n \log^2 n)$.

Examples using the Master Method (cont.)

4. $T(n) = 9T(n/3) + n^{2.5}$.

In this case we have $n^{\log_b a} = n^{\log_3 9} = n^2$.

We are in Case 3, since $f(n) = n^{2.5} = \Omega(n^{2+\varepsilon})$, for $\varepsilon = 1/2$, and $af(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2}f(n)$. (So $\delta = (1/3)^{1/2}$ works for the statement of Case 3.) This means that $T(n)$ is $\Theta(n^{2.5})$ by the Master Method.

5. $T(n) = 7T(n/3) + n$.

Here we see that $n^{\log_b a} = n^{\log_3 7}$. Since $1 < \log_3 7$, we see that there is some (small) $\varepsilon > 0$ such that $1 < \log_3 7 - \varepsilon$ and therefore $f(n) = n = O(n^{\log_3 7 - \varepsilon})$. Hence, we see that $T(n)$ is $\Theta(n^{\log_3 7})$.

Fast multiplication of integers

Now we consider a Divide-and-Conquer method to multiply integers (especially when the integers are very large, such as those used in various encryption schemes).

Suppose that I and J are two integers with n binary digits each.

We can easily find $I + J$ and $I - J$ in time $\Theta(n)$. Computing the product $I \cdot J$, using the common grade-school algorithm, take $\Theta(n^2)$ time. Can we do better? (Assuming that n is quite large...)

We will assume now that n is a power of 2 (which we can do by padding I and J with zeroes at the beginning).

Fast multiplication of integers (cont.)

Taking an inspiration (maybe?) from the MergeSort algorithm, let us consider writing I and J in two parts, namely the high-order and low-order bits.

In other words, we can write I and J as

$$\begin{aligned} I &= I_h 2^{n/2} + I_\ell \\ J &= J_h 2^{n/2} + J_\ell \end{aligned}$$

We observe that multiplying an integer by a power of 2 (like 2^k) is easy to do on a computer, as it is a (left) shift operation.

If a shift by one bit takes constant time to perform, then multiplying by 2^k takes $\Theta(k)$ time.

Fast multiplication of integers (cont.)

Given the representation of I and J above, in terms of the high- and low-order bits, let's concentrate on computing the product of I and J .

Given the above representation, we see that

$$\begin{aligned} I \cdot J &= (I_h 2^{n/2} + I_\ell) \cdot (J_h 2^{n/2} + J_\ell) \\ &= I_h J_h 2^n + I_\ell J_h 2^{n/2} + I_h J_\ell 2^{n/2} + I_\ell J_\ell \end{aligned}$$

We can use a Divide-and-Conquer type of algorithm to compute this product.

Namely, let's divide I and J into the high- and low-order bits as above, compute four products of these (size $n/2$ bit) pieces, and then combine them together as above (performing some shift operations, to multiply by $2^{n/2}$ or 2^n where needed).

Fast multiplication of integers (cont.)

The Master Method, however, gives us insight into how we could improve our approach.

If we can reduce the *number* of recursive calls (by reducing the value of $a = 4$), we could improve the overall running time.

So let us consider the product

$$(I_h - I_\ell)(J_\ell - J_h) = I_h J_\ell - I_h J_h - I_\ell J_\ell + I_\ell J_h.$$

This might be an odd product to consider, but note that it consists of two of the products we want to compute, namely

$$I_h J_\ell \text{ and } I_\ell J_h,$$

together with two of the products that we (will) compute recursively, namely

$$I_h J_h \text{ and } I_\ell J_\ell.$$

Fast multiplication of integers (cont.)

Computing the shifts and additions requires, in total, time $O(n)$ so we end up with a recurrence relation that looks like (for $n \geq 2$)

$$T(n) = 4T(n/2) + cn$$

for some constant c .

Applying the Master Method, we find we are in Case 1, namely since $n^{\log_2 4} = n^2$, and $f(n) = cn = O(n^{2-\epsilon})$ for, say, $\epsilon = 1/2$, we find that $T(n) = \Theta(n^2)$.

Hmmm, so our Divide-and-Conquer method hasn't helped us do things any quicker...

Fast multiplication of integers (cont.)

So our Divide-and-Conquer strategy becomes this:

1. Write I and J in terms of their high- and low-order bits.
2. Compute (recursively) the products $I_h J_h$, $I_\ell J_\ell$, and $(I_h - I_\ell)(J_\ell - J_h)$.
3. Combine these together to get

$$\begin{aligned} I \cdot J &= (I_h 2^{n/2} + I_\ell)(J_h 2^{n/2} + J_\ell) \\ &= I_h J_h 2^n + [(I_h - I_\ell)(J_\ell - J_h) + I_h J_h + I_\ell J_\ell] 2^{n/2} + I_\ell J_\ell \\ &= I_h J_h 2^n + I_\ell J_h 2^{n/2} + I_h J_\ell 2^{n/2} + I_\ell J_\ell. \end{aligned}$$

The important point is that this requires only three recursive multiplications of integers with $n/2$ bits.

Fast multiplication of integers (cont.)

The recurrence relation describing the running time to multiply the integers now becomes

$$T(n) = 3T(n/2) + cn$$

for a constant $c > 0$.

Since $1 < \log_2 3$, we can find a small $\varepsilon > 0$ such that $f(n) = cn = O(n^{\log_2 3 - \varepsilon})$. So we're still in Case 1 of the Master Method, but now we have this result:

Theorem

We can multiply two n -bit integers in time $\Theta(n^{\log_2 3}) = O(n^{1.585})$.

Matrix Multiplication (cont.)

For example, we have the following products

$$\begin{pmatrix} 2 & 5 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} 1 & -2 \\ 3 & 0 \end{pmatrix} = \begin{pmatrix} 17 & -4 \\ 8 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 3 & 4 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 2 \\ -1 & 2 & 0 \\ 4 & 5 & -2 \end{pmatrix} = \begin{pmatrix} 15 & 28 & -4 \\ -5 & -3 & 2 \\ 4 & 8 & 0 \end{pmatrix}.$$

Matrix Multiplication: Another example

Suppose we are given two $n \times n$ matrices X and Y , and we wish to compute their product $Z = X \cdot Y$, which is defined in this manner:

$$Z[i, j] = \sum_{k=1}^n X[i, k] \cdot Y[k, j].$$

Each element $Z[i, j]$ takes time $\Theta(n)$ to compute (since there are n multiplications and $n - 1$ additions to perform), assuming that any multiplication or addition takes constant time.

As there are n^2 elements to compute for Z , this naturally leads to an $\Theta(n^3)$ time algorithm.

Matrix Multiplication (cont.)

Another useful way of viewing this product (especially for large values of n) is in terms of products of *sub-matrices*

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

where

$$\begin{aligned} I &= AE + BG \\ J &= AF + BH \\ K &= CE + DG \\ L &= CF + DH. \end{aligned}$$

However, this gives a *Divide-and-Conquer* algorithm with running time $T(n)$, such that $T(n) = 8T(n/2) + bn^2 = \Theta(n^3)$ (using the Master Method).

Strassen's Algorithm for matrix multiplication

Volker Strassen (in 1969) realized that matrix multiplication can be performed using a divide and conquer approach that uses a *smaller* number of multiplications.

First define the following seven matrix products:

$$\begin{aligned}S_1 &= A(F - H) \\S_2 &= (A + B)H \\S_3 &= (C + D)E \\S_4 &= D(G - E) \\S_5 &= (A + D)(E + H) \\S_6 &= (B - D)(G + H) \\S_7 &= (C - A)(E + F)\end{aligned}$$

Strassen's Algorithm (cont.)

We can then find $I, J, K,$ and L using the S_i 's as follows:

$$\begin{aligned}I &= S_4 + S_5 + S_6 - S_2 \\J &= S_1 + S_2 \\K &= S_3 + S_4 \\L &= S_1 + S_5 + S_7 - S_3\end{aligned}$$

Strassen's Algorithm (cont.)

Thus, we can compute $Z = X \cdot Y$ using only *seven* recursive multiplications of matrices of size $n/2 \times n/2$.

Therefore, the running time of Strassen's Algorithm satisfies this recursive relation:

$$T(n) = 7 T(n/2) + bn^2.$$

Using this recursive formula, together with the Master Method, we obtain this result:

Theorem: We can multiply two $n \times n$ matrices in $O(n^{\log_2 7}) = O(n^{2.808})$ time.