

Lecture 2: Multiplication, Divide & Conquer and Recurrence Relations

Based on Chapter 2 of the Textbook

1 Multiplication of Large Numbers

1.1 The Problem

Goal: Given two n -digit numbers X and Y , compute their product $X \cdot Y$.

$$X = x_{n-1} \dots x_0$$

$$Y = y_{n-1} \dots y_0$$

Example:

$$X = 5723$$

$$Y = 6171$$

Output $X \cdot Y = 35316633$

1.2 Approach 1: Grade-School Algorithm

The traditional method (the one you learned in grade school) involves multiplying every digit of X by every digit of Y and adding the shifted results. What is the running time?

- It takes $O(n)$ time to write down the n shifted products (not $O(n^2)$ since there are only 10 base 10) or 2 (base 2) possibilities for the products before the shift).
- Then you have to add these n numbers together and each has $O(n)$ digits. This requires $n - 1$ additions each of which takes $O(n)$ time, for a total of $O(n^2)$ time to add these n shifted products.
- **Running Time:** $T(n) = O(n^2)$.

1.3 Approach 2: Divide & Conquer (Naive)

We can write $X = 5723 = 57 \cdot 100 + 23$, $Y = 61 \cdot 100 + 71$ and so $X \cdot Y = 57 \cdot 61 \cdot 10^4 + (57 \cdot 71 + 61 \cdot 23) \cdot 10^2 + 23 \cdot 71$.

In general we can split the n -digit numbers into two halves of size $n/2$. (Let's assume n is a power of 2 by padding with 0's if needed).

$$X = X_L \cdot 10^{n/2} + X_R$$

$$Y = Y_L \cdot 10^{n/2} + Y_R$$

The product becomes:

$$X \cdot Y = (X_L \cdot 10^{n/2} + X_R)(Y_L \cdot 10^{n/2} + Y_R)$$

Expanding this yields:

$$X \cdot Y = X_L Y_L 10^n + (X_L Y_R + X_R Y_L) 10^{n/2} + X_R Y_R$$

We can turn this identity into an algorithm:

Algorithm 1 Recursive Multiplication (Mult)

```
1: function MULT( $X, Y$ )
2:   if  $n = 1$  then return  $X \cdot Y$ 
3:   end if
4:   Split  $X, Y$  into  $X_L, X_R, Y_L, Y_R$ 
5:   return  $10^n X_L Y_L + 10^{n/2}(X_L Y_R + Y_L X_R) + X_R Y_R$ 
6: end function
```

We are calling 4 subproblems of size $n/2$ each then combining them using addition of $O(n)$ -digit numbers (which takes $O(n)$ time).

Recurrence Relation:

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

Solving this yields $T(n) = O(n^2)$, which is the same as the grade-school method. (But it is a different approach!)

One way to solve it is to guess $T(n) = n^k$, which yields $n^k = 4/2^k n^k + O(n)$; or in other words, $2^k = 4$ or $k = 2$, yielding $T(n) = O(n^2)$.

1.4 Approach 3: a smarter recursive algorithm

(this is Karatsuba's Algorithm).

Key Insight: We can reduce the number of multiplications from 4 to 3 using the following identity:

$$X_L Y_R + X_R Y_L = (X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R$$

This allows us to compute the middle term using values we already need to compute ($X_L Y_L$ and $X_R Y_R$) plus just one new multiplication: $(X_L + X_R)(Y_L + Y_R)$.

The identity becomes:

$$X \cdot Y = X_L Y_L 10^n + [(X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R] 10^{n/2} + X_R Y_R$$

We can implement this as a recursive algorithm as follows:

Algorithm 2 Karatsuba Multiplication (Mult2)

```
1: function MULT2( $X, Y$ )
2:   if  $n = 1$  then return  $X \cdot Y$ 
3:   end if
4:   Split  $X, Y$  into  $X_L, X_R, Y_L, Y_R$ 
5:    $P_1 = \text{Mult2}(X_L, Y_L)$ 
6:    $P_2 = \text{Mult2}(X_R, Y_R)$ 
7:    $P_3 = \text{Mult2}(X_L + X_R, Y_L + Y_R)$ 
8:   return  $10^n P_1 + 10^{n/2}(P_3 - P_1 - P_2) + P_2$ 
9: end function
```

New Recurrence:

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

Using the Master Theorem (derived below), since $3 > 2^1$, the complexity is determined by the leaves:

$$k = \log_2 3 \approx 1.58$$

$$T(n) = O(n^{1.58})$$

2 Solving Recurrence Relations

2.1 General Setup

Many divide and conquer algorithms follow this pattern:

- Split a problem of size n into a subproblems of size n/b .

- Recombining results takes $O(n^d)$ time.

a, b, d are the three parameters of the problem.

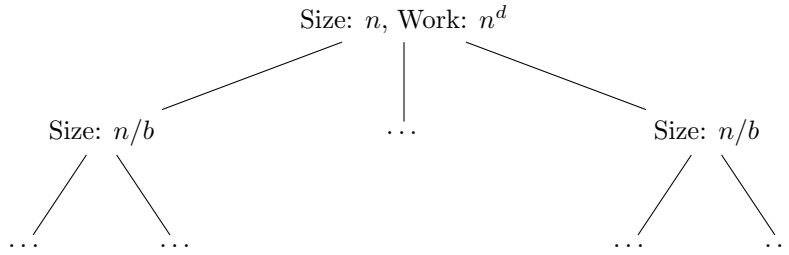
The general recurrence is:

$$T(n) = aT(n/b) + O(n^d)$$

We will solve this recurrence relation by drawing the computation as a tree.

2.2 The Recursion Tree Method

We can visualize the work done at each level of the recursion.



Some facts about this tree (say the top is level 0):

- The tree has depth $\log_b n$ since you need to divide n by b that many times to get to 1.
- At level j there are a^j nodes, each representing a subproblem of size n/b^j .
- At each node you spend $O((n/b^j)^d)$ recombination time

Now we can add up all the work level by level.

Analysis by Level:

- **Level 0:** 1 problem, work n^d .
- **Level 1:** a problems, each size n/b , total work $a(n/b)^d$.
- **Level j :** a^j problems, total work $a^j(n/b^j)^d = n^d(\frac{a}{b^d})^j$.
- **Depth:** The tree has depth $\log_b n$.

The work done at each successive level gets multiplied by a factor (a/b^d) . This is a geometric series, and its behavior depends on whether (a/b^d) is greater than, less than, or equal to 1.

2.3 The Master Theorem

The total running time depends on the ratio a vs b^d . There are 3 cases:

1. **Case 1** ($a > b^d$): The work grows at each level (bottom-heavy). The bottom level dominates.

$$T(n) = O(n^{\log_b a})$$

2. **Case 2** ($a < b^d$): The work shrinks at each level (top-heavy). The top level dominates.

$$T(n) = O(n^d)$$

3. **Case 3** ($a = b^d$): The work is equal at every level.

$$T(n) = O(n^d \log n)$$

3 Examples

3.1 Binary Search

Given a sorted array A and a number k , decide if k is in A .

- **Logic:** Compare k to the middle element. Recurse on one half.
- **Recurrence:** $T(n) = T(n/2) + O(1)$.
- **Parameters:** $a = 1, b = 2, d = 0$.
- **Analysis:** Since $b^d = 2^0 = 1$ and $a = 1$, we have $a = b^d$.
- **Result (Case 3):** $T(n) = O(n^0 \log n) = O(\log n)$.

3.2 Merge Sort

Given an array, sort it by recursively sorting halves and merging them.

- **Logic:** Split array in 2, sort both halves, Merge in $O(n)$ time.
- **Recurrence:** $T(n) = 2T(n/2) + O(n)$.
- **Parameters:** $a = 2, b = 2, d = 1$.
- **Analysis:** Since $b^d = 2^1 = 2$ and $a = 2$, we have $a = b^d$.
- **Result (Case 3):** $T(n) = O(n^1 \log n) = O(n \log n)$.