

1.1.5 Algorithm

```
ImprovedMult(N, M):
  Let X be 2(log(N + M) / 2)
  Write N = AX + B, M = CX + D
  P1 = Product(A, C)
  P2 = Product(B, D)
  P3 = Product(A + B, C + D)
  Return P1 X2 + [P3 - P1 - P2]X + P3
```

The first two lines take $\mathcal{O}(n)$ time, the three `Product` calls take $\mathcal{O}(n^2)$, and the last step takes $\mathcal{O}(n)$, for a total runtime of $\mathcal{O}(n^2)$. Despite there not being an asymptotic difference, note that this algorithm runs in $\mathcal{O}(\frac{3}{4}n^2 + n)$ time, whereas the naive algorithm runs in $\mathcal{O}(n^2)$ time, so there is a slight improvement.

1.1.6 Further Improvements

Our algorithm still uses the naive algorithm (`Product`). However, we don't actually need to use this; we can just use our own implementation! We introduce *Karatsuba's Algorithm*.

```
KaratsubaMult(N, M)
  If N + M < 99
    Return Product(N, M)
  Let X be 2(log(N + M) / 2)
  Write N = AX + B, M = CX + D
  P1 = KaratsubaMult(A, C)
  P2 = KaratsubaMult(B, D)
  P3 = KaratsubaMult(A + B, C + D)
  Return P1 X2 + [P3 - P1 - P2]X + P3
```

Here, the pre-processing and post-processing still takes $\mathcal{O}(n)$ time. The three recursive calls are a bit tricky.

1.1.7 Runtime Recurrence

Karatsuba's multiplication on inputs of size n spends $\mathcal{O}(n)$ time, and then makes three recursive calls to problems of approximately half the size. If $T(n)$ is the runtime for n -bit inputs, we have the recurrence:

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ 3T(n/2 + \mathcal{O}(1)) + \mathcal{O}(n) & \text{Otherwise} \end{cases}$$

This isn't exactly a clean recurrence that we can solve. That being said, this runs in roughly $\mathcal{O}(n^{\log_2(3)})$ time. We will explain this later.

1.2 Generalization

We will often get runtime recurrences with divide and conquer looking something like:

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ aT\left(\frac{n}{b} + \mathcal{O}(1)\right) + \mathcal{O}(n^d) & \text{Otherwise} \end{cases}$$

Here, the second line is saying a subproblems of size $\frac{n}{b}$. Note that the recursive subcalls are a constant *fraction* of the size of the original. For example, if $T(n) = 2T(n-1)$, then $T(n) = \mathcal{O}(2^n)$.

1.2.1 Tracking Recursive Calls

We have:

- 1 recursive calls of size n
- a recursive calls of size $n/b + O(1)$
- a^2 recursive calls of size $n/b^2 + O(1)$
- \dots
- a^k recursive calls of size $n/b^k + O(1)$

So, the total runtime is:

$$\begin{aligned} \text{Total Runtime} &= \sum_{k=0}^{\log_b(n)} a^k \mathcal{O}\left(\left(\frac{n}{b^k}\right)^d\right) \\ &= \mathcal{O}(n^d) \sum_{k=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^k \end{aligned}$$

There are several cases to consider.

1. $a > b^d$: Increasing geometric series dominated by last term. The runtime is dominated by recursive calls at the bottom level. The runtime would be $\mathcal{O}(n^{\log_b(a)})$.
2. $a < b^d$: Decreasing geometric series is dominated by the first term. Runtime is mostly based on the cleanup steps at the top level. The runtime is $\mathcal{O}(n^d)$.
3. $a = b^d$: Every level of the recursion does the same amount of work. The runtime is $\mathcal{O}(n^d \log(n))$.

1.2.2 Master Theorem

Theorem 1.1: Master Theorem

Let $T(n)$ be given by the recurrence:

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ aT\left(\frac{n}{b}\right) + \mathcal{O}(n^d) & \text{Otherwise} \end{cases}$$

Then we have:

$$T(n) = \begin{cases} \mathcal{O}(n^{\log_b(a)}) & a > b^d \\ \mathcal{O}(n^d \log(n)) & a = b^d \\ \mathcal{O}(n^d) & a < b^d \end{cases}$$

1.2.3 Example: Runtime

Suppose that a divide and conquer algorithm needs to solve 4 recursive subproblems of half the size and do $\mathcal{O}(n^2)$ additional work. What is the runtime?

Here, we have 4 subproblems ($a = 4$) of half the size ($b = 2$) and we need to do $\mathcal{O}(n^2)$ additional work ($d = 2$). So, our recurrence is given by

$$T(n) = \begin{cases} \mathcal{O}(1) & n = \mathcal{O}(1) \\ 4T\left(\frac{n}{2}\right) + \mathcal{O}(n^2) & \text{Otherwise} \end{cases}$$

Since $b^d = 2^2 = 4$ and $a = b^d$, it follows that the runtime is

$$\mathcal{O}(n^2 \log(n))$$

1.3 Matrix Multiplication

Suppose we wanted to multiply $n \times n$ matrices.

1.3.1 Recall

If $AB = C$, then

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

The naive algorithm computes this sum of n terms for each of n^2 entries, so the runtime is given by $\mathcal{O}(n^3)$.

1.3.2 Block Matrix Multiplication

If we divide the matrix into blocks, we can get the product of the full matrix in terms of the products of the blocks.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Here, A, B, C, \dots are $(n/2) \times (n/2)$ matrices.

1.3.3 Divide and Conquer Algorithm

We can compute 8 products of $(n/2) \times (n/2)$ matrices, and do some addition to get an answer. The runtime is

$$T(n) = 8T(n/2) + \mathcal{O}(n^2)$$

The Master Theorem states that this would give us $\mathcal{O}(n^3)$ runtime. And, this isn't hard to see why. If we unrolled what this algorithm was doing, we're essentially just doing the naive multiplication.