

## Lecture 10

*Lecturer: Andre Wibisono**Scribe: Michał Gerasimiuk*

## 1 Overview

Recall that the divide-and-conquer algorithms from Lecture 10 have the following running times:

- **Binary search**  $T(n) = T(\frac{n}{2}) + O(1) \implies T(n) = O(\log n)$ ,
- **Mergesort**  $T(n) = 2T(\frac{n}{2}) + O(n) \implies T(n) = O(n \log n)$ ,
- **Karatsuba integer multiplication**  $T(n) = 3T(\frac{n}{2}) + O(n) \implies T(n) = O(n^{\log_2 3})$ .

In this lecture, we will learn how to solve a recurrence and get a closed-form running time for any divide-and-conquer algorithm. We will also analyze Strassen's matrix multiplication algorithm.

## 2 Master theorem

Suppose that you have an algorithm that takes an input of size  $n$ . It is a divide-and-conquer algorithm, so it splits the problem into  $A$  sub-problems of size  $\frac{n}{B}$ , where  $A, B > 0$ .<sup>1</sup> You know that combining solutions to sub-problems takes  $O(n^D)$  time, where  $D \geq 0$ . The running time of your algorithm is given by the recurrence

$$T(n) = A \cdot T\left(\frac{n}{B}\right) + O(n^D).$$

If you were to visualize the relationship between the original problem and all its sub-problems, you would get a tree in which all non-leaf nodes have  $A$  children. Your algorithm would have a base case when dividing the sub-problem size by  $B$  is no longer possible. This means that the tree has depth

$$k = \log_B n.$$

In each level of the tree, you have  $A$  times as many nodes as in the previous level. The number of leaf nodes in tree is then

$$w = A^{\log_B n},$$

which is equal to

$$w = n^{\log_B A}.$$

---

<sup>1</sup>If you are interested in sub-problems of different sizes, you can have a look at the Akra-Bazzi theorem.

**Theorem** (Master theorem). For a recurrence of the form given above, let  $R = \frac{A}{B^D}$ . Then,

$$T(n) = \begin{cases} O(n^D) & \text{if } R < 1, \\ O(n^D \log n) & \text{if } R = 1, \\ O(n^{\log_B A}) & \text{if } R > 1. \end{cases}$$

**Example 1** (Binary search). After checking the middle element of a list, we search either the left or right half and immediately return the result we get. The recurrence is

$$T(n) = T\left(\frac{n}{2}\right) + O(1),$$

so  $A = 1, B = 2, D = 0$ . This means that  $R = \frac{1}{2^0} = 1$ , and

$$T(n) = O(n^0 \log n) = O(\log n).$$

**Example 2** (Mergesort). We sort two halves of an array and build a sorted array by merging them. The recurrence is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

so  $A = 2, B = 2, D = 1$ . This means that  $R = \frac{2}{2^1} = 1$ , and

$$T(n) = O(n \log n).$$

**Example 3** (Naive integer multiplication). If we split each integer into high-order and low-order bits, we need to do four smaller variable multiplications followed by constant multiplications and additions. The recurrence is

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n),$$

so  $A = 4, B = 2, D = 1$ . This means that  $R = \frac{4}{2^1} = 2$ , and

$$T(n) = O(n^{\log_2 4}) = O(n^2).$$

**Example 4** (Karatsuba integer multiplication). Karatsuba's algorithm uses more additions/subtractions but requires only three smaller multiplications. The recurrence is

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n),$$

so  $A = 3, B = 2, D = 1$ . This means that  $R = \frac{3}{2}$ , and

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.59}).$$

*Proof.* Recall how we visualize the applications of a divide-and-conquer algorithm as a tree. On the tree's  $k$ th level, we have  $A^k$  sub-problems, and it takes  $O((\frac{n}{B^k})^D)$  work to combine their solutions. (No work is actually done on the sub-problems unless we are at the leaf-level, where the problems are constant-sized.) This means that we do

$$T_k(n) = O\left(\frac{A^k n^D}{B^{kD}}\right) = \left(\frac{A}{B^d}\right)^k O(n^D)$$

work. Recall that  $\frac{A}{B^D}$  is  $R$ , so we can express this as

$$T_k(n) = R^k O(n^D).$$

The total work for this algorithm is

$$T(n) = \sum_{k=0}^{\log_B n} T_k(n) = O(n^D) \sum_{k=0}^{\log_B n} R^k$$

because the tree has  $\log_B n$  levels. From Pset 1, we know that

$$\sum_{k=0}^{\log_B n} R^k = \begin{cases} O(1) & \text{if } R < 1, \\ O(\log_B n) & \text{if } R = 1, \\ O(R^{\log_B n}) & \text{if } R > 1, \end{cases}$$

which gives us

$$T(n) = \begin{cases} O(n^D) & \text{if } R < 1, \\ O(n^D \log n) & \text{if } R = 1, \\ O(n^D R^{\log n}) & \text{if } R > 1, \end{cases}$$

where the third formula is the same as

$$O\left(n^D \frac{A^{\log_B n}}{(B^D)^{\log_B n}}\right) = O\left(n^D \frac{A^{\log_B n}}{(B^{\log_B n})^D}\right) = O\left(n^D \frac{A^{\log_B n}}{n^D}\right) = O(A^{\log_B n}) = O(n^{\log_B A}).$$

□

### 3 Matrix multiplication

For two  $n$  by  $n$  arrays of numbers  $X$  and  $Y$ , the product  $XY$  is defined such that the element in row  $i$  and column  $j$  is given by

$$(XY)_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}.$$

A naive algorithm for matrix multiplication would loop over all  $n^2$  entries of  $XY$  and compute this sum of  $n$  numbers, giving a running time of  $O(n^3)$ .

We can try a divide-and-conquer algorithm by decomposing  $X$  and  $Y$  into four  $\frac{n}{2}$  by  $\frac{n}{2}$  block matrices each. Then,

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

This requires solving 8 smaller problems, each of size  $\frac{n}{2}$ , and combining them by filling an  $n$  by  $n$  matrix. This means that  $A = 8, B = 2, D = 2$ , so  $R = \frac{8}{2^2} = 2$ , and the running time is

$$T(n) = O(n^{\log_2 8}) = O(n^3),$$

which is no better than the naive approach.

However, it is enough to compute only 7 products:

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \\ P_5 &= (A + D)(E + H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$

and combine them as

$$XY = \begin{bmatrix} P_4 + P_5 + P_6 - P_2 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}.$$

For this algorithm, we have  $A = 7, B = 2, D = 2$ , so  $R = \frac{7}{4}$ , and the running time is

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81}).$$

There are even faster algorithms for matrix multiplication. The asymptotically best one today runs in  $O(n^{2.37286})$  time.