# CS 170 Lecture Notes, Fall 2020

## Algorithms and Intractable Problems

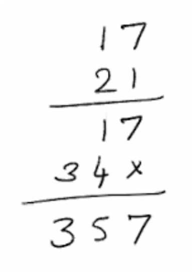
Professors: Avishay Tal, Umesh Vazirani
Scribe: Vishal Raman

## Contents

# §1 September 1st, 2020

## §1.1 Naive Multiplication

Recall the example of Fibonacci: we went from a complexity of $O(2^n) \to O(n^2) \to O(f(n))$, where $f(n)$ is the runtime for multiplying $n$-bit numbers. The naive algorithm is the usual multiplication algorithm for multiplying by hand.



Listing 1: Naive n-bit multiplication

```
function multiply(x, y):
Input: n-bit integers, x, y, y >= 0
Output: Product

    if (y == 0) return 0;
    s = multiply(x, floor(y/2))
    if y is even:
        return 2z
    else
        return x+2z
```

If we write $x = \sum_{i=0}^{n-1} x_j 2^i, y = \sum_{i=0}^{n-1} y_i 2^i$, so

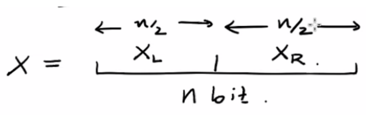
$$xy = \left(\sum_{i=0}^{n-1} x_j 2^i\right)\left(\sum_{i=0}^{n-1} y_i 2^i\right) = \sum_{j,k=0}^{n} x_j y_k 2^{j+k}.$$

## §1.2 Divide and Conquer: Karatsuba's Algorithm

For a Divide and Conquer problem, we do the following:

1. Break problem into pieces.
2. Solve pieces recursively.
3. Glue solutions of pieces to get solution of original problem.

We first have $x$ an n-bit number that we break up into $x_L, x_R$, each $n/2$ bit numbers. Similarly, we break $y$ into $y_L, y_R$.

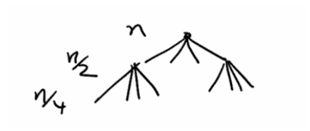


Note that $x = 2^{n/2}x_L + x_R, y = 2^{n/2}y_L + y_R,$, so

$$xy = 2^n x_L x_L + 2^{n/2}(x_L y_R + x_R y + L) + x_R y_R.$$

We now have 4 multiplications, involving $n/2$-bit numbers. Multiplication by $2^m$ can be shifting ($O(m)$ time), and addition is $O(1)$. Hence, we have a recurrence equation for the runtime,

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



Note that the depth of the recursion tree is $\log n$ and there are $4^{\log n} = n^2$ leaves. But this would have the same runtime as the naive algorithm, so more work is required to optimize.

We note the following trick from Gauss:

$$(a + bi)(c + di) = (ac - bd) + (ad + bc),$$

and note that

$$(a + b)(c + d) = (ac + bd) + (ad + bc),$$

so it suffices to compute $ac, bd, (a + b)(c + d)$, since $(a + b)(c + d) - ac - bd = ad + bc$, which is the other term.

We can apply the trick in the following way:

$$(2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L x_L + 2^{n/2}(x_L y_R + x_R y + L) + x_R y_R$$

and

$$(x_L + x_R)(y_L + y_R) = x_L y_L + x_R y_R + (x_L y_R + y_L x_R).$$

Our new recurrence relation is

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

Now, the branching factor is only 3, so the number of leaves is given by

$$3^{\log n} = n^{\log 3}.$$

## §1.3 Master Theorem

We have the following method of solving recurrence relations:

**Theorem 1** (Master Theorem)
If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for constants $a > 0, b > 1, d \geq 0$, then

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

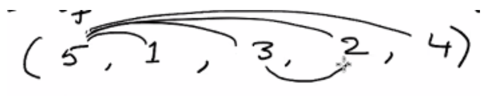
## §2 September 3rd, 2020

### §2.1 Sorting

Recall merge sort: we have an array $a[1, \ldots, n]$ and we want to sort the elements in non-decreasing order. We run merge sort on each half, and merge the two halves by taking 2 pointers and picking the smallest element from one of the two halves. Merge sort has a runtime of

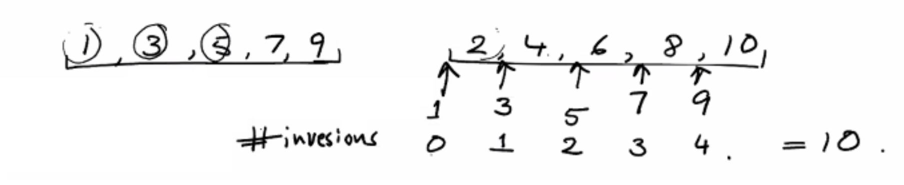
$$T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n).$$

We now consider the problem of counting the number of inversions. We define an **inversion** of an array $a$ to be a pair of indicies $i, j$ so that $i < j$ but $a[i] > a[j]$.



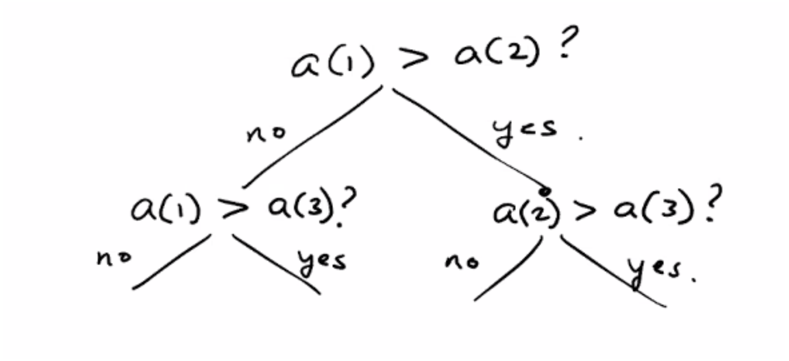
We will use a divide-and-conquer approach. We split the two halves, and for each half, we count inversions and sort the halves.

When calling the recursive operation on $a[1, \ldots, n]$ it should return $b[1, \ldots, n/2]$, the sorted left half and $k_1$, the number of inversions in the left half, $c[1, \ldots, n/2]$ the sorted right half, and $k_2$ the number of inversions in the right half. Then, when merging both halves, we can read off the number of inversions by counting the number of skipped elements in the right half when inserting elements from the left half.



The merging operation takes $O(n)$, and we would like $T(n) = 2T(n/2) + O(n)$ to get a runtime of $T(n) = O(n \log n)$. Proof of correctness can be done by induction on $n$, which is left as an exercise.

Can we do better than $O(n \log n)$? As an abstraction, consider an array $a[1, \ldots, n]$. We can make a query $a(i) > a(j)$ for each $i, j \in \{1, \ldots, n\}$, for $i < j$.



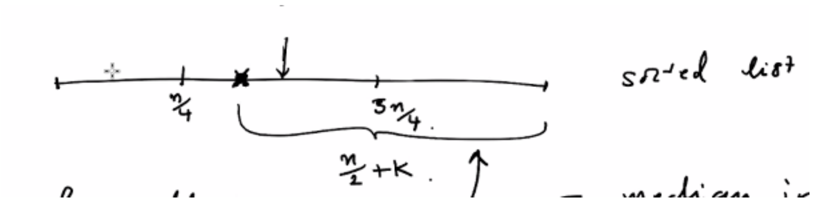
In order to have a proper sorting algorithm, each leaf must contain exactly one permutation. Hence, the number of leaves is at least $n!$, the number of permutations with $n$ elements. The height of the tree(which is the worst case runtime), is given by

$$\Omega(\log n!) = \Omega(\log(n/e)^n) = \Omega(n \log n).$$

## §2.2 Median Finding

Suppose we have an array $a[1, \ldots, n]$. We wish to find the median of the list.

Suppose we can pick out some element $\ell$ between $n/4$ and $3n/4$ in the sorted list. If we make $n$ comparisons with the element, we'd know how many elements are less than the element and how many are larger than it, so we'd know which half contained the median (either $[1, \ell]$ or $[\ell + 1, n]$).



If $k < n$, then find the k-th smallest element in the left half. Otherwise, find the $k - n$-th element in the right half. We have the following recurrence,

$$T(n) \leq T(\frac{3n}{4}) + O(n) \Rightarrow T(n) = O(n).$$

To actually do this, pick some element at random. We can show the expected number of times it takes to pick a proper element is 2. So, redefine $T(n)$ is be the expected time to find the $k$-th smallest in $n$ element array.

Suppose the time to find the element is $T_1 = n$. Then, the time to find the next element is $T_2 = (3/4)n$, the next would be $T_3 = (3/4)^2 n$, so we obtain a geometric series $T = \sum T_i = O(n)$.

Now, $E[T_1] = 2n, E[T_2] = 2(3/4)n, \ldots$. Note that

$$E(T) = E(\sum_{i=1}^{d} T_i) = \sum E(T_i) = O(n),$$

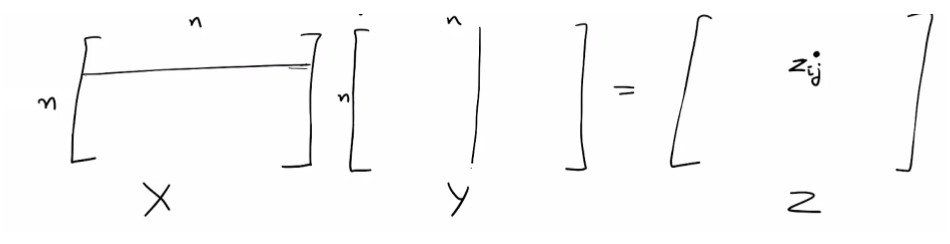
by linearity of expectation.

Suppose $P[+1] = p$ and $W$ is the waiting time for a heads.

$$E(W) = p + (1-p)(1 + E(W)) = 1 + (1-p)E(W) \Rightarrow E(w) = \frac{1}{p}.$$

## §2.3 Matrix Multiplication

We are given two $n$ by $n$ matrices $X, Y$, and we would like to compute $XY = Z$. This is naively $O(n^3)$:



Note that

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

We try a divide-and-conquer approach. Split $X$ and $Y$ as follows:

$$\begin{array}{c@{}c} & \begin{array}{cc} \frac{n}{2} & \frac{n}{2} \end{array} \\ \begin{array}{c} \frac{n}{2} \\ \frac{n}{2} \end{array} & \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \end{array} \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right] = \left[\begin{array}{c|c} AE+BG & AF+BH \\ \hline CE+DG & \end{array}\right]$$

Then,

$$T(n) = 8T(n/2) + O(n^2) \Rightarrow T(n) = O(n^3).$$

This has lower constant factors due to memory accessing, but the runtime can be further improved with Strassen's algorithm:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

$$XY = \begin{bmatrix} \overbrace{P_5 + P_4 - P_2 + P_6}^{AE+BG} & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

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

We have

$$T(n) = 7T(n/2) + O(N^2) \Longrightarrow T(n) = O(n^{\log_2(7)}).$$

# §3 September 8th, 2020

## §3.1 Fast Fourier Transform

The Fast Fourier Transform(FFT) is a method of multiplying two polynomials

$$\left(\sum_{i=0}^{n-1} a_i x^i\right)\left(\sum_{i=0}^{n-1} b_i b^i\right) = \sum_{i,j} a_i b_j x^{i+j},$$

which naively is done in $O(n^2)$. With FFT, we can do it in $O(n \log n)$.

- It allows us to do fast integer multiplication. We can reduce it from $O(n^{\log 3})$ to $O(n \log n)$.
- Signal processing algorithms: If we sample times $a_0, a_1, a_2, \dots, a_{n-1}$ from a signal, we can reconstruct the output signal from a unit input at time $t = 0$, if our processor is a linear, time-invariant one.