

## Homework 1A

**Big-O**

For the following list of functions, cluster the functions of the same order into one group, and then rank the groups in increasing order.

- (a)  $n \log(n)$
- (b)  $n^{1.01}$
- (c)  $n\sqrt[3]{n}$
- (d)  $2^{\log_3(n)}$
- (e)  $n + \log(n)$
- (f)  $2^{\log(\log(n))}$
- (g)  $10n \log(10n) + 10$
- (h)  $(\log(n))^{10}$
- (i)  $\log(n^{10})$
- (j)  $42n$

**Solution:** For ease of ordering, manipulate each function like so:

- (a)  $n \log(n)$
- (b)  $n^{1.01}$
- (c)  $n\sqrt[3]{n} \rightarrow n^{4/3}$
- (d)  $2^{\log_3(n)} \rightarrow n^{\log_2 3}$
- (e)  $n + \log(n) \rightarrow n$
- (f)  $2^{\log(\log(n))} \rightarrow \log n$
- (g)  $10n \log(10n) + 10 \rightarrow 10n(\log(n) + \log(10)) \rightarrow n \log(n)$
- (h)  $(\log(n))^{10} \rightarrow 2^{\log \log^{10}(n)}$  (manipulate other functions to compare exponents)
- (i)  $\log(n^{10}) \rightarrow 10 \log(n) = \log(n)$
- (j)  $42n \rightarrow n$

Hence, we can group  $\{f, i\} < \{h\} < \{d\} < \{e, j\}, \{a, g\} < \{b\} < \{c\}$ .

## Slow Multiplication

Consider the following multiplication algorithm:

```
def slowmult(x, y):  
    result = 0  
    for i from 1 to x (inclusive):  
        result += y  
    return result
```

Assume  $x$  and  $y$  are  $n$ -bit numbers. What is the running time of this algorithm in terms of  $n$ ?

**Solution:** The runtime of this algorithm is  $O(n2^n)$ . This is because the for loop iterates a total of  $x$  times, and  $x$  has  $n$  bits meaning it can represent a number as large as  $2^n$  meaning  $x = O(2^n)$ .

Within each iteration, we are adding *result* and  $y$  together, which costs  $O(n)$  time per iteration, giving a total runtime of  $O(2^n) \times O(n) = O(n2^n)$ .

### Fast Multiplication

Let  $x$  and  $y$  be two  $n$ -bit numbers where  $n$  is divisible by 3. Let  $x_L, x_M, x_R$  consist of the first, middle, and last third of the bits in  $x$  and define  $y_L, y_M, y_R$  similarly.

- (a) Express  $xy$  in terms of  $x_L, x_M, x_R, y_L, y_M, y_R$ .

**Solution:** See the following expansion and simplification:

$$\begin{aligned} xy &= (2^{\frac{2n}{3}}x_L + 2^{\frac{n}{3}}x_M + x_R)(2^{\frac{2n}{3}}y_L + 2^{\frac{n}{3}}y_M + y_R) \\ &= 2^{\frac{4n}{3}}x_Ly_L + 2^{\frac{3n}{3}}x_Ly_M + 2^{\frac{2n}{3}}x_Ly_R \\ &\quad + 2^{\frac{3n}{3}}x_My_L + 2^{\frac{2n}{3}}x_My_M + 2^{\frac{n}{3}}x_My_R \\ &\quad + 2^{\frac{2n}{3}}x_Ry_L + 2^{\frac{n}{3}}x_Ry_M + x_Ry_R \\ &= 2^{\frac{4n}{3}}x_Ly_L + 2^n(x_Ly_M + x_My_L) + 2^{\frac{2n}{3}}(x_Ly_R + x_My_M + x_Ry_L) \\ &\quad + 2^{\frac{n}{3}}(x_My_R + x_Ry_M) + x_Ry_R \end{aligned}$$

- (b) Give a recursive algorithm that calculates the above polynomial with a recurrence relation of:

$$T(n) = 6T\left(\frac{n}{3}\right) + O(n)$$

**Solution:** In the above algorithm, we are required to make 9 recursive calls which would give an incorrect runtime. Through algebraic manipulation, we can reduce the total number of multiplications.

To design our algorithm, first create a base case table when multiplying 1-bit numbers (using Piazza clarification that  $n$  is a power of 3). This table is  $0 \times 1, 1 \times 0, 0 \times 0, 1 \times 1$  which gives  $O(1)$  resolution of base cases.

Further, we compute 3 multiplications recursively:

$$A = x_Ly_L \quad B = x_My_M \quad C = x_Ry_R$$

We also compute three more terms recursively given by:

$$\begin{aligned} D &= (x_L + x_M)(y_L + y_M) = (x_L y_L + x_L y_M + x_M y_L + x_M y_M) \\ E &= (x_L + x_R)(y_L + y_R) = (x_L y_L + x_R y_L + x_R y_R) \\ F &= (x_M + x_R)(y_M + y_R) = (x_M y_M + x_M y_R + x_R y_M + x_R y_R) \end{aligned}$$

None of these terms are immediately present in our polynomial, but by adding/removing copies of  $A, B, C$ , we can reconstruct the polynomial with only these 6 recursive calls.

$$\begin{aligned} D - A - B &= (x_L + x_M)(y_L + y_M) - x_L y_L - x_M y_M = x_L y_M + x_M y_L \\ B + E - A - C &= x_M y_M + (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R \\ &= x_M y_M + x_L y_R + x_R y_L \\ F - B - C &= (x_M + x_R)(y_M + y_R) - x_M y_M - x_R y_R = x_M y_R + x_R y_M \end{aligned}$$

These three terms ARE present in the polynomial and only require a total of 6 recursive multiplications to construct. Hence, our polynomial can be written as:

$$= 2^{\frac{4n}{3}} A + 2^n (D - A - B) + 2^{\frac{2n}{3}} (B + E - A - C) + 2^{\frac{n}{3}} (F - B - C) + C$$

Now, we can describe the entire algorithm. The algorithm takes an input of bits for  $x$  and  $y$  and checks them against a 1 bit base case table (returning the base case). Otherwise, we split  $x$  and  $y$  into three parts by splicing the bits into 3 partitions (left, right, and middle).

Using the calls from above, we recursively call our algorithm to construct  $A, B, C, D, E, F$  and return the reconstructed polynomial with the equation given above.

The runtime of this is given by 6 recursive calls, all of which have size of  $n/3$ , and  $O(n)$  non recursive work (bit partitioning and polynomial reconstruction). All in all, this combines to give the target runtime recurrence relation of  $T(n) = 6T(n/3) + O(n)$ .