Basic Algorithms
Fall 2012

Divide and Conquer
Divide and Conquer: a (somewhat) general theorem

The setup: a recursive algorithm that on inputs of size $n \geq n_0$, recursively solves

- $\leq a$ smaller sub-problems,
- each of size $\leq n/b + c$,
- with a “local” running time $\leq d n^e$

where $n_0, a, b, c, d, e$ are constants
Recursion tree analysis

At level 1, size $\leq n/b + c$

At level 2, size $\leq n/b^2 + c/b + c$

$\cdots$

At level $j$,

$$\text{size} \leq n/b^j + c/b^{j-1} + \cdots + c/b + c \leq n/b^j + C_1,$$

where $C_1 := c/(1 - 1/b)$

At level $j$, there are $\leq a^j$ nodes
Set \( k := \lfloor \log_b n \rfloor \), so \( n \leq b^k < bn \)

At level \( k \), all sizes are \( \leq 1 + C_1 \), and we can ignore all nodes at levels \( k + 1, k + 2, \ldots \) (their contribution to the total cost is at most a constant times the sum of costs at level \( k \))

Let \( w = \text{sum of costs at levels } 0, \ldots, k \)

For each \( j = 0 \ldots k \), sum of costs at level \( j \) is

\[
\leq a^j \cdot d(n/b^j + C_1)^e \\
\leq C_2 a^j (n/b^j)^e \\
= C_2 n^e (a/b^e)^j
\]
Therefore,

\[ w \leq C_2 n^e \sum_{j=0}^{k} \delta^j, \]

where \( \delta := a/b^e \)

**Case 1:** \( \delta < 1 \)

\[ \sum_{j=0}^{\infty} \delta^j = 1/(1 - \delta) \implies w \leq (C_2/(1 - \delta))n^e \]

Total running time \( = O(n^e) \)

**Case 2:** \( \delta = 1 \)

\[ \sum_{j=0}^{k} \delta^j = (k + 1) \implies w \leq C_2(k + 1)n^e \]

Total running time \( = O(n^e \log n) \)
Case 3: $\delta > 1$

$$
\sum_{j=0}^{k} \delta^j = \frac{\delta^{k+1} - 1}{\delta - 1}
$$

and so

$$
w \leq C_3 n^e \delta^k = C_3 n^e a^k / (b^k)^e \leq C_3 a^k
$$

$$
\leq C_3 a^{\log_b n + 1} = C_3 a \cdot a^{\log_b n}
$$

$$
= C_3 a \cdot b^{\log_b a \cdot \log_b n}
$$

$$
= C_3 a \cdot n^{\log_b a}
$$

Total running time $= O(n^{\log_b a})$
Summarizing — the “Master Theorem”

Let $f := \log_b a$

**Case 1:** $e > f \implies O(n^e)$

**Case 2:** $e = f \implies O(n^e \log n)$

**Case 3:** $e < f \implies O(n^f)$
Application: faster multiplication

Problem: multiply two \( n \)-bit integers

An “\( n \)-bit integer” is an integer \( a \) such that \( 0 \leq a < 2^n \)

An \( n \)-bit integer can be represented using an array of \( n \) bits (although in practice, one packs several bits into a “word”)

The sum of two \( n \)-bit integers is an \((n + 1)\)-bit integer, and can be computed in time \( O(n) \)

The product of two \( n \)-bit integers is a \((2n)\)-bit integer, and can be computed in time \( O(n^2) \)
Karatsuba’s multiplication algorithm

Input: two $n$-bit integers, $a$ and $b$

If $n$ is “very small”, use the naive algorithm

Otherwise, divide each number into two pieces:

$$a = a_1 2^k + a_0$$
$$b = b_1 2^k + b_0,$$

where $k := \lfloor n/2 \rfloor$

$$a: \begin{array}{c} a_1 \end{array} \begin{array}{c} a_0 \end{array}$$

$$b: \begin{array}{c} b_1 \end{array} \begin{array}{c} b_0 \end{array}$$
\[ ab = a_1 b_1 2^{2k} + (a_1 b_0 + a_0 b_1) 2^k + a_0 b_0 \]
If we recursively compute the four sub-products $a_1b_1, a_1b_0, a_0b_1, a_0b_0$, we get another $O(n^2)$ algorithm

- $e = 1$, $f = \log_2 4 = 2$, Case 3 of Master Theorem

Better idea:

- Compute $A \leftarrow a_1 + a_0$, $B \leftarrow b_1 + b_0$
- Recursively compute three products:
  - $H \leftarrow a_1b_1$, $L \leftarrow a_0b_0$, $F \leftarrow AB$
- Observe: $F = a_1b_1 + a_1b_0 + a_0b_1 + a_0b_0$
- Thus, we can compute $M \leftarrow F - (H + L)$, which is $a_1b_0 + a_0b_1$, and $P \leftarrow H2^{2k} + M2^k + L$, which is $ab$
Now apply Master Theorem: \( e = 1, \)
\[ f = \log_2 3 \approx 1.585 \]

Case 3: running time is \( O(n^{\log_2 3}) \)

Notes:

- Not the fastest method: using the Fast Fourier Transform, one can multiply two \( n \)-bit integers in time \( O(n \log n \log \log n) \)
- For \( n \) (roughly) in the range 500–10,000, Karatsuba is the fastest
- You use it every time you buy something from amazon.com, or use ssh — it’s used to implement public-key cryptosystems