Divide and Conquer

A first example: Merge Sort

A generic recursive sorting algorithm

Input: a list \(L\)
Output: a sorted list

if \(|L| \leq 1\) then \(\quad // \ |L| \) means length of \(L\)
  return \(L\)
else
  split \(L\) into two nonempty sublists \(L_1\) and \(L_2\)
  recursively sort \(L_1\) and \(L_2\)
  return \(merge(L_1, L_2)\)
A linear time merge algorithm

merge($L_1, L_2$)
Input: sorted lists $L_1$ and $L_2$
Output: a sorted list $L$

initialize $L$ to empty list
while $L_1$ and $L_2$ are both non-empty do
  if $\text{head}(L_1) \leq \text{head}(L_2)$ then
    move $\text{head}(L_1)$ to tail of $L$
  else
    move $\text{head}(L_2)$ to tail of $L$
while $L_1$ not empty do
  move $\text{head}(L_1)$ to tail of $L$
while $L_2$ not empty do
  move $\text{head}(L_2)$ to tail of $L$

Running time analysis:
each loop iteration moves one element to $L$
$\Rightarrow$ total number of loop iterations $|L_1| + |L_2|$
An array implementation

Input: sorted arrays $A[0..m)$, $B[0..n)$
Output: sorted array $C[0..m+n)$

$i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$
while $i < m$ and $j < n$ do
    if $A[i] \leq B[j]$ then
        $C[k++] \leftarrow A[i++]$
    else
        $C[k++] \leftarrow B[j++]$
while $i < m$ do
    $C[k++] \leftarrow A[i++]$
while $j < n$ do
    $C[k++] \leftarrow B[j++]$
Back to our recursive sorting algorithm . . .

Let $n = |L|$

Total running time:

- the “local computation” time: $O(n)$, plus
- the time spent in the recursive calls

The total running time is determined by the strategy used to split $L$ into two sublists

**Unbalanced strategy:** always split $L$ into sublists of size $n - 1$ and 1

total time is $O(T)$, where

$$T = n + (n - 1) + (n - 2) + \cdots + 1$$

$\implies O(n^2)$ running time (essentially insertion sort)
Balanced strategy: the Merge Sort Algorithm

Always split \( L \) into two sublists of (roughly) equal size

Running time analysis using a recursion tree:

- Every node in the tree corresponds to a single call
  - its children correspond to the recursive calls
- We associate with each node with the subproblem size and local cost of the corresponding call
- We add up all the local costs — usually level by level
Example: \( n = 8 \)

More generally, assume \( n = 2^k \)

At level \( j = 0, \ldots, k \):

- \( 2^j \) nodes, each with local cost \( 2^{k-j} \)
- Cost per level: \( 2^k = n \)
- Total cost: \( n(k + 1) = O(n \log n) \)
More observations

**Good news:**

- Merge Sort is a *stable* sort: items whose keys have equal value retain their relative positions

**Bad news:**

- The array implementation of Merge Sort is not an *in place*: $O(n)$ auxiliary space is needed
Divide and Conquer: a (somewhat) general theorem

The setup: a recursive algorithm that on inputs of size $n \geq n_0 > 1$, recursively solves

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

where $n_0, a, b, c, d, e$ are constants

"$T(n) \leq aT(n/b + c) + O(n^e)$"

Simplification: assume $c = 0$

General case: exercise
Recursion tree analysis

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

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

... 

At level $j$, size $\leq n/b^j$

At level $j$, there are $\leq a^j$ nodes

Set $k := \lceil \log_b n \rceil$, so $n \leq b^k < bn$

No levels past 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

$$\leq a^j \cdot d(n/b^j)^e = d \cdot n^e(a/b^e)^j$$
Therefore,

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

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

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

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

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

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

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

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 for some constant $C$, we have

$$w \leq Cn^e \delta^k = Cn^e a^k / (b^k)^e \leq C a^k$$

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

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

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

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:** \( e < f \implies O(n^f) \)

**Example:** Merge Sort: \( a = 2, b = 2, e = 1 \implies f = 1 \), Case 2, \( T(n) = O(n \log n) \)

**Example:** Binary Search: \( a = 1, b = 2, e = 0 \implies f = 0 \), Case 2, \( T(n) = O(\log n) \)
Application: faster multiplication

Problem: multiply two $n$-digit integers

An “$n$-digit integer” is an integer $a$ such that $0 \leq a < R^n$, where $R$ is the “radix” or “base”

Think of the radix $R$ as a constant, usually a power of 2 (for example, $R = 2^{32}$ or $2^{64}$)

An $n$-digit integer can be represented using an array of $n$ machine words
Addition of $n$-digit integers

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

input: $a = (a_{n-1}, \ldots, a_0), b = (b_{n-1}, \ldots, b_0)$
output: $c = (c_n, c_{n-1}, \ldots, c_0)$

$carry \leftarrow 0$
for $i$ in $[0..n)$ do
  $t \leftarrow a_i + b_i + carry$  // $[0..2R)$
  $c_i \leftarrow t \mod R$
  $carry \leftarrow \lfloor t/R \rfloor$  // $\{0, 1\}$
$c_n \leftarrow carry$
Multiplication of $n$-digit integers

The product of two $n$-digit integers is a $(2n)$-digit integer, and can be computed in time $O(n^2)$

input: $a = (a_{n-1}, \ldots, a_0)$, $b = (b_{n-1}, \ldots, b_0)$
output: $c = (c_{2n-1}, \ldots, c_0)$

initialize $c_i \leftarrow 0$ for $i$ in $[0..2n)$

for $i$ in $[0..n)$ do
  
  // $c \leftarrow c + R^i b_i \cdot a$
  carry $\leftarrow 0$
  for $j$ in $[0..n)$ do
    $t \leftarrow c_{i+j} + b_i \cdot a_j + carry$ // $[0..R^2)$
    $c_{i+j} \leftarrow t \mod R$
    carry $\leftarrow \lfloor t/R \rfloor$ // $[0..R)$
  $c_{i+n} \leftarrow carry$
Karatsuba’s multiplication algorithm

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

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

Otherwise, divide each number into two pieces:

$$a = a_{hi}R^k + a_{lo}$$
$$b = b_{hi}R^k + b_{lo},$$

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

\[\begin{array}{c|c}
\text{a:} & a_{hi} & a_{lo} \\
\hline
\text{b:} & b_{hi} & b_{lo} \\
\end{array}\]
\[ ab = a_{hi} b_{hi} R^{2k} + (a_{hi} b_{lo} + a_{lo} b_{hi}) R^k + a_{lo} b_{lo} \]

\[ \approx n \]

\[ a_{hi} b_{hi} \]

\[ a_{hi} b_{lo} + a_{lo} b_{hi} \]

\[ + \]

\[ a_{lo} b_{lo} \]

\[ \approx 2n \]
One idea:
Recursively compute the four sub-products
\[ a_{hi}b_{hi}, \ a_{hi}b_{lo}, \ a_{lo}b_{hi}, \ a_{lo}b_{lo} \]
Case 3 of Master Theorem: \( e = 1, f = \log_2 4 = 2 \)
\( \implies \) another \( O(n^2) \) algorithm

A better idea:
Compute \( A \leftarrow a_{hi} + a_{lo}, \ B \leftarrow b_{hi} + b_{lo} \)
Recursively compute three products:
\[ H \leftarrow a_{hi}b_{hi}, \ L \leftarrow a_{lo}b_{lo}, \ F \leftarrow AB \]
Observations:
\[ F = a_{hi}b_{hi} + a_{hi}b_{lo} + a_{lo}b_{hi} + a_{lo}b_{lo} \]
\[ M := F - (H + L) = a_{hi}b_{lo} + a_{lo}b_{hi} \]
\[ P := HR^{2k} + MR^k + L = ab \]
Case 3 of Master Theorem: \( e = 1, f = \log_2 3 \approx 1.585 \)
Running time is \( O(n^{\log_2 3}) \)
Notes:

• Karatsuba is *not* the fastest method: using the Fast Fourier Transform, one can multiply two \( n \)-digit integers in time \( O(n \log n \log \log n) \)

• For 500–10,000 bit numbers, 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