Divide-and-conquer, part 1: Mergesort

Russell Impagliazzo and Miles Jones
Thanks to Janine Tiefenbruck

http://cseweb.ucsd.edu/classes/sp16/cse21-bd/
April 13, 2016
Recursion

Last Time
1. Recursive algorithms and correctness
2. Coming up with recurrences
3. Using recurrences for time analysis

Today: Using recursion to design faster algorithms
Important example: Mergesort
Important sub-procedure: Merge
Example of ``divide-and-conquer” algorithm design

In the textbook: Sections 5.4, 8.3
Given two sorted lists:

\[ a_1 \ a_2 \ a_3 \ \ldots \ a_k \]
\[ b_1 \ b_2 \ b_3 \ \ldots \ b_\ell \]

produce a sorted list of length \( n = k + \ell \) which contains all their elements.

What's the result of merging the lists 1,4,8 and 2, 3, 10, 20?

A. 1,4,8,2,3,10,20
B. 1,2,4,3,8,10,20
C. 1,2,3,4,8,10,20
D. 20,10,8,4,3,2,1
E. None of the above.
Merging sorted lists: HOW

Given two **sorted** lists

\[ a_1, a_2, a_3, \ldots, a_k \]
\[ b_1, b_2, b_3, \ldots, b_\ell \]

produce a **sorted** list of length \( n = k + \ell \) which contains all their elements.

**Design an algorithm to solve this problem**

- Compare \( a_1 \) and \( b_1 \) and output the smaller one into a list.
- Remove it.
- Recurse on remaining lists.

- If \( k = 0 \) then output \( b_1, \ldots, b_\ell \).
- If \( \ell = 0 \) then output \( a_1, \ldots, a_k \).
Merging sorted lists: HOW

Similar to Rosen p. 369

A recursive algorithm

Idea: Find the smallest element.
  Put it first in the sorted list
  ``Delete'' it from the list it came from
  Merge the remaining parts of the lists recursively

If the input lists $a_1..a_k$ and $b_1..b_ℓ$
Are sorted, which elements could be the smallest in the merged list?

$$a_1 \lor b_1$$
Merging sorted lists: HOW

Similar to Rosen p. 369

A recursive algorithm

```
procedure RMerge(a_1, ..., a_k, b_1, ..., b_l; sorted lists)
  if first list is empty then return b_1, ..., b_l
  if second list is empty then return a_1, ..., a_k
  if a_1 <= b_1 then
    return a_1 o RMerge(a_2, ..., a_k, b_1, ..., b_l)
  else
    return b_1 o RMerge(a_1, ..., a_k, b_2, ..., b_l)
```

“o” = concatenate

Find the smallest element

Merge the remaining parts
Merging sorted lists: WHY

Similar to Rosen p. 369

A recursive algorithm
Focus on merging head elements, then rest.

Claim: returns a sorted list containing all elements from either list

Proof by induction on $n=k+\ell$, the total input size

```
procedure RMerge($a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)
  if first list is empty then return $b_1, \ldots, b_\ell$
  if second list is empty then return $a_1, \ldots, a_k$
  if $a_1 \leq b_1$ then
    return $a_1 \circ \text{RMerge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
  else
    return $b_1 \circ \text{RMerge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$
```
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

procedure $RMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)
    if first list is empty then return $b_1, \ldots, b_\ell$
    if second list is empty then return $a_1, \ldots, a_k$
    if $a_1 \leq b_1$ then
        return $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
    else
        return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$

What is the base case?
A. Both input lists are empty (n=0).
B. The first list is empty.
C. The second list is empty.
D. One of the lists is empty and the other has exactly one element (n=1).
E. None of the above.
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Base case: Suppose n=0. Then both lists are empty. So, in the first line we return the (trivially sorted) empty list containing all elements from the second list. But this list contains all (zero) elements from either list, because both lists are empty. 😊

procedure $R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)
if first list is empty then return $b_1, \ldots, b_\ell$
if second list is empty then return $a_1, \ldots, a_k$
if $a_1 \leq b_1$ then
    return $a_1 \circ R\text{Merge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
else
    return $b_1 \circ R\text{Merge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$

empty list is sorted trivially 🌟
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Induction Step: Suppose n≥1 and \( R\text{Merge}(a_1,\ldots,a_k,b_1,\ldots,b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n-1 \). What do we want to prove?

A. \( R\text{Merge}(a_1,\ldots,a_k,a_{k+1},b_1,\ldots,b_l) \) returns a sorted list containing all elements from either list.
B. \( R\text{Merge}(a_1,\ldots,a_k,b_1,\ldots,b_l,b_{l+1}) \) returns a sorted list containing all elements from either list.
C. \( R\text{Merge}(a_1,\ldots,a_k,b_1,\ldots,b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n \).
Merging sorted lists: WHY

**Induction Step**: Suppose \( n \geq 1 \) and \( \text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k + l = n - 1 \). We want to prove:

\[
\text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l) \text{ returns a sorted list containing all elements from either list whenever } k + l = n.
\]

**Case 1**: one of the lists is empty.

**Case 2**: both lists are nonempty.
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Induction Step: Suppose n>=1 and \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n-1 \). We want to prove:

\( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n \).

Case 1: one of the lists is empty: similar to base case. In first or second line return rest of list.
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Case 2a: both lists nonempty and \( a_1 \leq b_1 \)
Since both lists are sorted, this means \( a_1 \) is not bigger than
* any of the elements in the list \( a_2, \ldots, a_k \)
* any of the elements in the list \( b_1, \ldots, b_l \)
The total size of the input of \( RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l) \) is \( (k-1) + l = n-1 \) so by the IH, it returns a sorted list containing all elements from either list.
Prepending \( a_1 \) to the start maintains the order and gives a sorted list with all elements. 😊
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Case 2b: both lists nonempty and $a_1 > b_1$
Same as before but reverse the roles of the lists. 😊

```plaintext
procedure RMerge($a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)
  if first list is empty then return $b_1, \ldots, b_\ell$
  if second list is empty then return $a_1, \ldots, a_k$
  if $a_1 \leq b_1$ then
    return $a_1 \circ R\text{Merge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
  else
    return $b_1 \circ R\text{Merge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$
```

procedure \textit{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell: \text{sorted lists})
\begin{align*}
\text{if first list is empty then return } b_1, \ldots, b_\ell \\
\text{if second list is empty then return } a_1, \ldots, a_k \\
\text{if } a_1 \leq b_1 \text{ then} \\
\quad \text{return } a_1 \circ \textit{RMerge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell) \\
\text{else} \\
\quad \text{return } b_1 \circ \textit{RMerge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)
\end{align*}

If \( T(n) \) is the time taken by \textit{RMerge} on input of total size \( n \),
\[
T(0) = c \\
T(n) = T(n-1) + c'
\]
where \( c, c' \) are some constants
Merging sorted lists: WHEN

If $T(n)$ is the time taken by $RMerge$ on input of total size $n$,

$$
T(0) = c \\
T(n) = T(n-1) + c'
$$

where $c$, $c'$ are some constants

What's a solution to this recurrence equation?
A. $T(n) \in O(T(n - 1))$
B. $T(n) \in O(n)$
C. $T(n) \in O(n^2)$
D. $T(n) \in O(2^n)$
E. None of the above.
Merging sorted lists: WHEN

If $T(n)$ is the time taken by $RMerge$ on input of total size $n$,

$$
T(0) = c \\
T(n) = T(n-1) + c'
$$

where $c$, $c'$ are some constants

This the same recurrence as we solved Monday for counting 00’s inm a string. So we can just remember that this works out to $T(n) \in \theta(n)$
"We split into two groups and organized each of the groups, then got back together and figured out how to interleave the groups in order."
A divide & conquer (recursive) strategy:

- **Divide** list into two sub-lists
- **Recursively** sort each sublist
- **Conquer** by merging the two sorted sublists into a single sorted list
procedure MergeSort(a_1, \ldots, a_n)
    if n > 1 then
        \textcolor{red}{m := \lfloor n/2 \rfloor}
        L_1 := a_1, \ldots, a_m
        L_2 := a_{m+1}, \ldots, a_n
        return \textcolor{red}{RMerge(MergeSort(L_1), MergeSort(L_2))}
    else return a_1, \ldots, a_n

\textcolor{red}{(else return a_i,)} already sorted.
Merge Sort: WHY

procedure MergeSort(a₁, ..., aₙ)
    if n > 1 then
        m := ⌊n/2⌋
        L₁ := a₁, ..., aₘ
        L₂ := aₘ₊₁, ..., aₙ
        return RMerge( MergeSort(L₁), MergeSort(L₂) )
    else return a₁, ..., aₙ

Claim that result is a sorted list containing all elements.

Proof by strong induction on n:

Why do we need strong induction?
A. Because we're breaking the list into two parts.
B. Because the input size of the recursive function call is less than n.
C. Because we're calling the function recursively twice.
D. Because we're using a subroutine, RMerge.
E. Because the input size of the recursive function call is less than n-1.
Merge Sort: WHY

Claim that result is a sorted list containing all elements.

Proof by strong induction on n:

Base case: Suppose n=0.

Suppose n=1.
Merge Sort: WHY

procedure MergeSort(a₁, …, aₙ)
    if n > 1 then
        m := ⌊n/2⌋
        L₁ := a₁, …, aₘ
        L₂ := aₘ₊₁, …, aₙ
        return RMerge( MergeSort(L₁), MergeSort(L₂) )
    else return a₁, …, aₙ

Claim that result is a sorted list containing all elements.

Proof by strong induction on n:

Base case: Suppose n=0. Then, in the else branch, we return the empty list, (trivially) sorted.

Suppose n=1. Then, in the else branch, we return a₁, a (trivially) sorted list containing all elements. ☺
Claim that result is a sorted list containing all elements.

**Induction step**: Suppose \( n > 1 \). Assume, as the strong induction hypothesis, that \( \text{MergeSort} \) correctly sorts all lists with \( k \) elements, for any \( 0 \leq k < n \).

Goal: prove that \( \text{MergeSort}(a_1, \ldots, a_n) \) returns a sorted list containing all \( n \) elements.
Merge Sort: WHY

procedure MergeSort(a₁, ..., aₙ)
    if n > 1 then
        m := \lfloor n/2 \rfloor
        L₁ := a₁, ..., aₘ
        L₂ := aₘ₊₁, ..., aₙ
        return RMerge( MergeSort(L₁), MergeSort(L₂) )
    else return a₁, ..., aₙ

IH: MergeSort correctly sorts all lists with k elements, for any 0 ≤ k < n

Goal: prove that MergeSort(a₁, ..., aₙ) returns a sorted list containing all n elements.

Since n > 1, in the if branch we return RMerge( MergeSort(L₁), MergeSort(L₂) ), where L₁ and L₂ each have no more than (n/2) + 1 elements and together they contain all elements.

By IH, each of MergeSort(L₁) and MergeSort(L₂) are sorted and by the correctness of Merge, the returned list is a sorted list containing all the elements. 😊
procedure MergeSort($a_1, \ldots, a_n$)

if $n > 1$ then

$m := \lfloor n/2 \rfloor$

$L_1 := a_1, \ldots, a_m$

say $\theta(n)$

$L_2 := a_{m+1}, \ldots, a_n$

say $\theta(n)$

$T_{	ext{MergeSort}}(n) = 2T_{	ext{MergeSort}}(n/2) + T_{	ext{Merge}}(n) + c'' n$

where $c_0, c', c''$ are some constants

else return $a_1, \ldots, a_n$

end if

return $R\text{Merge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2))$

If $T_{\text{MS}}(n)$ is runtime of $\text{MergeSort}$ on list of size $n$,

$T_{\text{MS}}(0) = c_0$

$T_{\text{MS}}(1) = c'$

$T_{\text{MS}}(n) = 2T_{\text{MS}}(n/2) + T_{\text{Merge}}(n) + c'' n$

$T_{\text{MS}}(n) \in O(n)$

$2T_{\text{MS}}(n/2) + O(n)$

$= 2T_{\text{MS}}(n/2) + O(n) + c'' n$
Merge Sort: WHEN

\[
\text{procedure } \text{MergeSort}(a_1, \ldots, a_n) \\
\text{if } n > 1 \text{ then} \\
\theta(1) \ m := \lceil n/2 \rceil \\
? \quad L_1 := a_1, \ldots, a_m \\
? \quad L_2 := a_{m+1}, \ldots, a_n \\
\text{else return } a_1, \ldots, a_n \quad T_{\text{MS}}(n/2) \quad T_{\text{MS}}(n/2) \\
\text{return } R\text{Merge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2)) \quad T_{\text{Merge}}(n/2 + n/2)
\]

If \( T_{\text{MS}}(n) \) is runtime of MergeSort on list of size \( n \),

\[
T_{\text{MS}}(0) = c_0 \quad T_{\text{MS}}(1) = c' \\
T_{\text{MS}}(n) = 2T_{\text{MS}}(n/2) + cn
\]

where \( c_0, c, c' \) are some constants
Merging sorted lists: WHEN

If $T_{MS}(n)$ is runtime of MergeSort on list of size $n$,

\[
T_{MS}(0) = c_0 \quad \quad T_{MS}(1) = c'
\]

\[
T_{MS}(n) = 2T_{MS}(n/2) + cn
\]

where $c_0$, $c$, $c'$ are some constants

Solving the recurrence by **unravelling**:

\[
T_{MS}(n) = 2T_{MS}(n/2) + cn
\]

\[
= 2 \left( 2T_{MS}(n/4) + c(n/2) \right) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn
\]

\[
= 4 \left( 2T_{MS}(n/8) + c(n/4) \right) + 2cn = 8T_{MS}(n/8) + 3cn
\]

\[
\vdots
\]

\[
= 2^k T_{MS}(n/2^k) + k(cn)
\]
Merging sorted lists: WHEN

Solving the recurrence by **unravelling**:

\[ T_{MS}(n) = 2T_{MS}(n/2) + cn \]

\[ = 2(2T_{MS}(n/4) + c(n/2)) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn \]

\[ = 4(2T_{MS}(n/8) + c(n/4)) + 2cn = 8T_{MS}(n/8) + 3cn \]

\[ : \]

\[ = 2^k T_{MS}(n/2^k) + k(cn) \]

**What value of k should we substitute to finish unravelling (i.e. to get to the base case)?**

A. k  
B. n  
C. \(2^n\)  
D. \(\log_2 n\)  
E. None of the above.
Merging sorted lists: WHEN

Solving the recurrence by **unravelling**:

\[
T_{MS}(n) = 2T_{MS}(n/2) + cn
\]

\[
= 2\left( 2T_{MS}(n/4) + c(n/2) \right) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn
\]

\[
= 4\left( 2T_{MS}(n/8) + c(n/4) \right) + 2cn = 8T_{MS}(n/8) + 3cn
\]

\[
\vdots
\]

\[
= 2^kT_{MS}(n/2^k) + k(cn)
\]

With \( k = \log_2 n \), \( T_{MS}(n/2^k) = T_{MS}(n/n) = T_{MS}(1) = c' \) :

\[
T_{MS}(n) = 2^{\log n} T_{MS}(1) + (\log_2 n)(cn) = c'n + c n \log_2 n
\]
In terms of worst-case performance, Merge Sort outperforms all other sorting algorithms we've seen.

<table>
<thead>
<tr>
<th></th>
<th>n</th>
<th>n^2</th>
<th>n log n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1000</td>
<td>1,000,000,000</td>
<td>~10,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000,000,000</td>
<td>~20,000,000</td>
<td></td>
</tr>
</tbody>
</table>

*Divide and conquer wins big!*
Divide & Conquer

What we saw:

Dividing into subproblems each with a fraction of the size was a big win.

Will this work in other contexts?
Given two $n$-digit (or bit) integers
\[ a = a_{n-1} \ldots a_1 a_0 \]
and
\[ b = b_{n-1} \ldots b_1 b_0 \]
return the decimal (or binary) representation of their product.

\[
\begin{array}{c}
25 \\
\times 17 \\
\hline
175 \\
+ 250 \\
\hline
425
\end{array}
\]
Multiplication: HOW

Given two $n$-digit (or bit) integers

$$a = a_{n-1}...a_1a_0$$

and

$$b = b_{n-1}...b_1b_0$$

return the decimal (or binary) representation of their product.

25
def 17
\hline
175
+ 250
\hline
425

Compute partial products (using single digit multiplications), shift, then add.

How many operations? $O(n^2)$
Multiplication: HOW

Divide and conquer? Divide n-digit numbers into two n/2-digit numbers.

If a = 12345678 and b = 24681357, we can write

\[
a = (1234) \times 10^4 + (5678) \\
b = (2468) \times 10^4 + (1357)
\]

To multiply:

\[
\left( (1234) \times 10^4 + (5678) \right) \left( (2468) \times 10^4 + (1357) \right) = \\
(1234)(2468) \times 10^8 + (1234)(1357) \times 10^4 + (2468)(5678) \times 10^4 + (1357)(5678)
\]
Multiplication: WHEN

One 8-digit multiplication

\[
\begin{pmatrix}
12345678 \\
24681357
\end{pmatrix} \times \begin{pmatrix}
1234 \\
2468
\end{pmatrix} = (1234 \times 10^4 + 5678)(2468 \times 10^4 + 1357) =
\]

\[
(1234)(2468) \times 10^8 + (1234)(1357) \times 10^4 + (2468)(5678) \times 10^4 + (1357)(5678)
\]

Four 4-digit multiplications (plus some shifts, sums)
Multiplication: WHEN

One 8-digit multiplication

\[
\begin{pmatrix}
12345678 \\
24681357
\end{pmatrix}
= \left( (1234) \times 10^4 + (5678) \right) \left( (2468) \times 10^4 + (1357) \right) =
\]

\[
(1234)(2468) \times 10^8 + (1234)(1357) \times 10^4 + (2468)(5678) \times 10^4 + (1357)(5678)
\]

Four 4-digit multiplications (plus some shifts, sums)

\[T(n) = 4 \ T(n/2) + cn \quad \text{with} \quad T(1) = c' \quad \text{and} \quad c, c' \text{ constants}\]
Multiplication: WHEN

\[ T(n) = 4 \, T(n/2) + cn \quad \text{with } T(1) = c' \quad \text{and } c, c' \text{ constants} \]

\[ T(n) = 4T(n/2) + cn \\
= 4 \left( 4T(n/4) + c(n/2) \right) + cn = 16T(n/4) + 3cn \\
= 16 \left( 4T(n/8) + c(n/4) \right) + 3cn = 64T(n/8) + 7cn \\
\vdots \\
= 4^k T(n/2^k) + (2^k - 1)cn \quad \text{Unravelling} \]
Multiplication: WHEN

\[ T(n) = 4 \, T(n/2) + cn \]

with \( T(1) = c' \) and \( c, c' \) constants

Unravelling

Substitute \( k = \log_2 n \)

\[ T(n) = 4T(n/2) + cn \]
\[ = 4\left(4T(n/4) + c(n/2)\right) + cn = 16T(n/4) + 3cn \]
\[ = 16\left(4T(n/8) + c(n/4)\right) + 3cn = 64T(n/8) + 7cn \]
\[ \vdots \]
\[ = 4^kT(n/2^k) + (2^k - 1)cn \]

What's \( 2^{\log n} \) ?
A. \( n \)
B. \( n^2 \)
C. \( 2^n \)
D. 1
E. None of the above
Multiplication: WHEN

\[ T(n) = 4 \, T(n/2) + cn \text{ \ with \ } T(1) = c' \text{ \ and \ } c, \ c' \text{ \ constants} \]

\[
T(n) = 4T(n/2) + cn \\
= 4(4T(n/4) + c(n/2)) + cn = 16T(n/4) + 3cn \\
= 16(4T(n/8) + c(n/4)) + 3cn = 64T(n/8) + 7cn \\
\vdots \\
= 4^kT(n/2^k) + (2^k - 1)cn
\]

Substitute \( k = \log_2 n \) \[ T(n) = 4^{\log n} \, T(1) + (2^{\log n} - 1) \, cn \]

What's \( 4^{\log n} \)?

A. \( n \)
B. \( n^2 \)
C. \( 2^n \)
D. \( 2n \)
E. None of the above
Multiplication: WHEN

$T(n) = 4 \cdot T(n/2) + cn$  
with $T(1) = c'$  
and $c$, $c'$ constants

$T(n) = 4T(n/2) + cn$

$= 4( 4T(n/4) + c(n/2) ) + cn = 16T(n/4) + 3cn$

$= 16( 4T(n/8) + c(n/4) ) + 3cn = 64T(n/8) + 7cn$

$\vdots$

$= 4^k T(n/2^k) + (2^k - 1)cn$

Substitute $k = \log_2 n$

$T(n) = c' n^2 + (n-1) cn \in \Theta(n^2)$
Multiplication: HOW

Insight: replace one (of the 4) multiplications by (linear time) subtraction

Andrey Kolmogorov 1903 - 1987

Anatoly Karatsuba 1937 - 2008

Rosen p. 528
Multiplication: HOW

\[
\begin{pmatrix}
12345678 \\
24681357
\end{pmatrix}
\begin{pmatrix}
24681357 \\
24681357
\end{pmatrix}
= 
\begin{pmatrix}
(1234) \cdot 10^4 + (5678) \\
(2468) \cdot 10^4 + (1357)
\end{pmatrix}
\]

\[
(1234)(2468) \cdot 10^8 + (1234)(1357) \cdot 10^4 + (2468)(5678) \cdot 10^4 + (1357)(5678)
\]

\[
(1234)(2468) \cdot (10^8+10^4) + [(1234) - (5678)][(1357)-(2468)] \cdot 10^4 + (1357)(5678) \cdot (10^4+1)
\]

Insight: replace one (of the 4) multiplications by (linear time) subtraction
Karatsuba Multiplication: WHEN

Instead of

\[ T(n) = 4 \, T(n/2) + cn \quad \text{with } T(1) = c' \quad \text{and } c, c' \text{ constants} \]

get

\[ T_K(n) = 3 \, T_K(n/2) + d \, n \quad \text{with } T_K(1) = d' \quad \text{and } d, d' \text{ constants} \]

Unravelling is similar but with 3s instead of 4s

\[ T_K(n) \in \Theta(3^{\log_2 n}) \]
Karatsuba Multiplication: WHEN

\[3^{\log n} = (2^{\log 3})^{\log n} = (2^{\log n})^{\log 3} = n^{\log 3} = n^{1.58...}\]

so definitely better than \(n^2\)

Progress since then …

1963: Toom and Cook develop series of algorithms that are time \(O(n^{1+\epsilon})\).
2007: Furer uses number theory to achieve the best known time for multiplication.
2016: Still open whether there is a linear time algorithm for multiplication.