Divide and Conquer: A Recursive Strategy for Better Algorithms

Janine Tiefenbruck
Today’s agenda

1. Compare iterative and recursive algorithms for Merge
2. Introduce the divide-and-conquer technique
3. Show how to give time analyses for divide-and-conquer algorithms
4. Show that simple recursions can be surprisingly powerful
5. Examples: Mergesort, Karatsuba multiplication
Example: Merging sorted arrays

In the merge problem, we are given two sorted arrays $A[1..n]$ and $B[1..m]$ and want to produce a sorted array containing the union of both lists. While this is interesting in its own right, it will also be a key sub-procedure in the recursive sorting algorithm MergeSort.

$$
A = \begin{array}{cccc}
2 & 7 & 9 & 11 \\
\end{array}
B = \begin{array}{cccc}
6 & 12 & 13 \\
\end{array}
$$

$$
C = \begin{array}{cccccccc}
2 & 6 & 7 & 9 & 11 & 12 & 13 \\
\end{array}
$$

We will present the merge algorithm first as an iterative algorithm and then show how to describe the same algorithm recursively.
Iterative merge algorithm

$$\text{IMerge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays})$$

1. $n \leftarrow k + \ell$
2. Initialize array $C[1, \ldots, n]$
3. $i \leftarrow 1, j \leftarrow 1$
4. \textbf{FOR} $t = 1$ \textbf{TO} $n$ \textbf{DO}:
5. \hspace{0.5cm} IF $i > k$ \textbf{THEN} $C[t] \leftarrow B[j], j++$
6. \hspace{0.5cm} IF $j > \ell$ \textbf{THEN} $C[t] \leftarrow A[i], i++$
7. \hspace{0.5cm} IF $A[i] \leq B[j]$ \textbf{THEN} $C[t] \leftarrow A[i], i++$
8. \hspace{0.5cm} ELSE $C[t] \leftarrow B[j], j++$
9. Return $C[1, \ldots, n]$.

How would we prove correctness?
Omitted, since our emphasis is on recursions today.
Iterative merge algorithm: correctness

**IMerge**($A[1, \ldots, k], B[1, \ldots, \ell]$: sorted arrays)

1. $n \leftarrow k + \ell$
2. Initialize array $C[1, \ldots, n]$
3. $i \leftarrow 1, j \leftarrow 1$
4. FOR $t = 1$ TO $n$ DO:
   5. IF $i > k$ THEN $C[t] \leftarrow B[j], j \leftarrow j + 1$
   6. IF $j > \ell$ THEN $C[t] \leftarrow A[i], i \leftarrow i + 1$
   7. IF $A[i] \leq B[j]$ THEN $C[t] \leftarrow A[i], i \leftarrow i + 1$
   8. ELSE $C[t] \leftarrow B[j], j \leftarrow j + 1$
5. Return $C[1, \ldots, n]$.

**Loop invariant:** After $t$ iterations, $C[1, \ldots, t]$ are the $t$ smallest elements of the union, they are sorted, and they contain all elements in $A[1, \ldots, i - 1]$ and $B[1, \ldots, j - 1]$. 
Iterative merge algorithm: time analysis

$IMerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays })$

1. $n \leftarrow k + \ell$
2. Initialize array $C[1, \ldots, n]$
3. $i \leftarrow 1, j \leftarrow 1$
4. FOR $t = 1$ TO $n$ DO:
5.   IF $i > k$ THEN $C[t] \leftarrow B[j], j++$
6.   IF $j > \ell$ THEN $C[t] \leftarrow A[i], i++$
7.   IF $A[i] \leq B[j]$ THEN $C[t] \leftarrow A[i], i++$
8.   ELSE $C[t] \leftarrow B[j], j++$
9. Return $C[1, \ldots, n]$.

Lines 5-8: Inside loop in line 4:

Lines 1-3, 9: Total:
Iterative merge algorithm: time analysis

\textit{IMerge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays})

1. \( n \leftarrow k + \ell \)
2. Initialize array \( C[1, \ldots, n] \)
3. \( i \leftarrow 1, j \leftarrow 1 \)
4. FOR \( t = 1 \) TO \( n \) DO:
   5. IF \( i > k \) THEN \( C[t] \leftarrow B[j], j++ \)
   6. IF \( j > \ell \) THEN \( C[t] \leftarrow A[i], i++ \)
   7. IF \( A[i] \leq B[j] \) THEN \( C[t] \leftarrow A[i], i++ \)
   8. ELSE \( C[t] \leftarrow B[j], j++ \)
9. Return \( C[1, \ldots, n] \).

Lines 5-8: \( O(1) \) \hspace{2cm} \text{Inside loop in line 4: } O(n)

Lines 1-3, 9: \( O(1) \) \hspace{2cm} \text{Total: } O(1 + n + 1) = O(n)
Recursive version

**Definition**

Let $v \circ C[1, \ldots, m]$ denote an array of length $m + 1$ whose first element is $v$ and the rest is $C[1, \ldots, m]$.

$$R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays})$$

1. IF $k = 0$ return $B[1, \ldots, \ell]$
2. IF $\ell = 0$ return $A[1, \ldots, k]$
4. ELSE return $B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell])$
Correctness, Merging sorted arrays

\[ RMerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays}) \]

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
   \[ A[1] \circ RMerge(A[2, \ldots, k], B[1, \ldots, \ell]) \]
4. ELSE return \( B[1] \circ RMerge(A[1, \ldots, k], B[2, \ldots, \ell]) \)

We want to show that \( RMerge(A[1, \ldots, k], B[1, \ldots, \ell]) \) is a sorted array containing all elements from either array. We’ll prove this by induction on \( n = k + \ell \), the total input size.
Correctness, base case

\(R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays})\)

1. IF \(k = 0\) return \(B[1, \ldots, \ell]\)
2. IF \(\ell = 0\) return \(A[1, \ldots, k]\)
4. ELSE return \(B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell])\)

**Base Case (\(n = 1\)):** When does this happen? What does the algorithm do in this case?
Correctness, base case

\[ RMerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays} ) \]

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
   \[ A[1] \circ RMerge(A[2, \ldots, k], B[1, \ldots, \ell]) \]
4. ELSE return \( B[1] \circ RMerge(A[1, \ldots, k], B[2, \ldots, \ell]) \)

\textbf{Base Case} (\( n = 1 \)): When does this happen? What does the algorithm do in this case?

If \( n = 1 \), then one array is empty and the other is of size 1. The algorithm returns the array of size 1, which has all the elements and is sorted trivially.
Correctness, induction step

\( \text{RMerge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays}) \)

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
   \[ A[1] \circ \text{RMerge}(A[2, \ldots, k], B[1, \ldots, \ell]) \]
4. ELSE return \( B[1] \circ \text{RMerge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)

**Induction hypothesis:** \( \text{RMerge}(A[1, \ldots, k], B[1, \ldots, \ell]) \) returns a sorted list that contains the elements of both arrays whenever \( k + \ell = n - 1 \).

**Induction step:** Goal: The same, but when \( k + \ell = n \).
Let \( A[1, \ldots, k] \) and \( B[1, \ldots, \ell] \) be sorted arrays where \( n = k + \ell \).

**Case 1:** If \( k = 0 \), we return \( B \), which is a sorted array containing all elements, and similarly if \( \ell = 0 \).
Correctness, induction step

\(RMerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays})\)

1. IF \(k = 0\) return \(B[1, \ldots, \ell]\)
2. IF \(\ell = 0\) return \(A[1, \ldots, k]\)
4. ELSE return \(B[1] \circ RMerge(A[1, \ldots, k], B[2, \ldots, \ell])\)

**Case 2a:** Neither \(k\) nor \(\ell\) are 0, and \(A[1] \leq B[1]\).
Correctness, induction step

\[ R\text{Merge}(A[1,\ldots, k], B[1,\ldots, \ell]: \text{sorted arrays}) \]

1. IF \( k = 0 \) return \( B[1,\ldots, \ell] \)

2. IF \( \ell = 0 \) return \( A[1,\ldots, k] \)

   \[ A[1] \circ R\text{Merge}(A[2,\ldots, k], B[1,\ldots, \ell]) \]

4. ELSE return \( B[1] \circ R\text{Merge}(A[1,\ldots, k], B[2,\ldots, \ell]) \)

**Case 2a:** Neither \( k \) nor \( \ell \) are 0, and \( A[1] \leq B[1] \). Then since \( A \) and \( B \) are sorted, \( A[1] \leq A[i] \) for any \( i \geq 1 \), and
\( A[1] \leq B[1] \leq B[j] \) for any \( j \geq 1 \). \( R\text{Merge} \) returns \( A[1] \) followed by \( R\text{Merge}[A[2,\ldots, k], B[1,\ldots, \ell]] \). By the induction hypothesis, since the total size of the two inputs is \( k + \ell - 1 = n - 1 \), the recursive call of \( R\text{Merge} \) returns a sorted list containing all the other elements \( A[2,\ldots, k], B[1,\ldots, \ell] \). Since \( A[1] \) is smaller than all these elements, the list we return is sorted, and it also contains exactly the elements from the two arrays.
Correctness, induction step

\(RMerge(A[1, \ldots, k], B[1, \ldots, \ell]): \) sorted arrays

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
4. ELSE return \( B[1] \circ RMerge(A[1, \ldots, k], B[2, \ldots, \ell]) \)

**Case 2b:** Neither \( k \) nor \( \ell \) are 0, and \( A[1] > B[1] \).
Correctness, induction step

\( R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays}) \)

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
4. ELSE return \( B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)

**Case 2b:** Neither \( k \) nor \( \ell \) are 0, and \( A[1] > B[1] \). The same argument, but reverse the roles of \( A \) and \( B \).
\[ \text{RMerge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays}) \]

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
4. ELSE return \( B[1] \circ \text{RMerge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)

Every step is constant time, except that we make one recursive call in either line 3 or line 4. Thus,

\[
T(1) = c \text{ for some constant } c \\
T(n) = T(n - 1) + c' \text{ for some constant } c'.
\]

This is of the same form as the same recurrence for base 2 exponentiation, so we already know \( T(n) \in O(n) \).
Divide-and-conquer is a form of recursive strategy for designing algorithms.

1. **Divide** an instance into several smaller instances of the same problem
2. **Recursively solve** each smaller instance.
3. **Conquer** by combining the solutions into the solution for the original instance.
We can use Merge as a sub-procedure in a MergeSort algorithm to sort an array.

To sort an unsorted array $A[1..n]$,

1. **Divide** $A$ into two sub-arrays, $A[1..n/2]$ and $A[n/2 + 1..n]$.
2. **Recursively sort** each sub-array.
3. **Conquer** by merging the two sorted sub-arrays into a single array.
MergeSort($A[1, \ldots, n]$)

1. IF $n = 1$ Return $A$
2. $B[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[1, \ldots, n/2])$
3. $C[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[n/2 + 1, \ldots, n])$
4. Return $\text{Merge}(B[1, \ldots, n/2], C[1, \ldots, n/2])$
For MergeSort to be correct, it should return a sorted array, and that array should contain exactly the elements $A[1], \ldots, A[n]$.

We’ll prove that MergeSort is correct by *strong induction* on $n$.

- In strong induction, you assume that the statement you want to show holds for all $n'$ with $1 \leq n' < n$.
- You must show that your statement is also true for $n$.
- We use strong induction whenever a recursive algorithm acting on an input of size $n$ makes calls with inputs of size other than $n - 1$. 
MergeSort correctness, base case

MergeSort(A[1, \ldots, n])

1. IF $n = 1$ Return $A$
2. $B[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[1, \ldots, n/2])$
3. $C[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[n/2 + 1, \ldots, n])$
4. Return $\text{Merge}(B[1, \ldots, n/2], C[1, \ldots, n/2])$

**Base Case:** If $n = 1$, the array $A$ has a single (sorted) element, $A[1]$, which is returned in line 1.
MergeSort correctness, strong induction step

MergeSort(A[1, . . . , n])

1. IF \(n = 1\) Return \(A\)
2. \(B[1, . . . , n/2] \leftarrow \text{MergeSort}(A[1, . . . , n/2])\)
3. \(C[1, . . . , n/2] \leftarrow \text{MergeSort}(A[n/2 + 1, . . . , n])\)
4. Return \(\text{Merge}(B[1, . . . , n/2], C[1, . . . , n/2])\)

**Strong Induction Hypothesis:** Assume MergeSort correctly sorts all arrays \(A[1..n']\) with \(1 \leq n' < n\).

**Inductive Step:** Show it correctly sorts arrays of size \(n\).

On an input of size \(n\), \(A[1..n]\), the strong induction hypothesis tells us that \(B[1..n/2]\) and \(C[1..n/2]\) are sorted, and together they contain all the elements of \(A\). By the correctness of \(\text{Merge}\), \(\text{Merge}(B, C)\) is a sorted list containing all the elements of \(A\).
Let $T_{MS}(n)$ represent the time that MergeSort takes on an array of size $n$.

Lines 2 and 3 take time $T_{MS}(n/2)$ each.
Line 4 takes time $T_{Merge}(n/2 + n/2) = T_{Merge}(n) \in O(n)$.

Thus, for constants $c$ and $c'$, we have the recurrence

$$T_{MS}(n) = 2T_{MS}(n/2) + cn$$ with $T_{MS}(1) = c'$.  

```
MergeSort(A[1, ..., n])

1  IF $n = 1$ Return A
2  $B[1, ..., n/2] \leftarrow \text{MergeSort}(A[1, ..., n/2])$
3  $C[1, ..., n/2] \leftarrow \text{MergeSort}(A[n/2 + 1, ..., n])$
4  Return $\text{Merge}(B[1, ..., n/2], C[1, ..., n/2])$
```
Any guesses?

What do you think $T(n)$ is?

a $T_{MS}(n) \in O(n^2)$?

b $T_{MS}(n) \in O(n \log n)$?

c $T_{MS}(n) \in O(n)$?

d $T_{MS}(n) \in O(n^{\log_2 3})$?

e None of the above
Solving the recurrence

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

\[
T_{MS}(n) = 2T_{MS}(n/2) + cn \\
= 4T_{MS}(n/4) + cn + 2(cn/2) \\
= 8T_{MS}(n/8) + cn + 2(cn/2) + 4(cn/4) \\
\vdots \\
= 2^k(T(n/2^k)) + cn + cn + cn + \ldots (k \text{ times})
\]

When \( k = \log n \), \( 2^k = n \), so \( T(n/2^k) = T(1) = c' \).

Thus, letting \( k = \log n \),

\[
T_{MS}(n) = nT(1) + cn + cn + \ldots (\log n \text{ times}) \\
= cn \log n + c'n \\
\in O(n \log n).
\]
Thus, for worst-case time, MergeSort is substantially better than any of the algorithms we’ve seen.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n^2$</th>
<th>$n \log n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>1,000,000</td>
<td>$\approx 10,000$</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000,000,000</td>
<td>$\approx 20,000,000$</td>
</tr>
</tbody>
</table>
We know from the grade-school multiplication algorithm that it typically takes $O(n^2)$ single digit operations to multiply two $n$ digit numbers.

We will omit some details in this presentation, but hopefully get across the main ideas.

How does divide and conquer compare? Note that the recursive algorithm we already saw was based on *numerical* division into 2, first recursively multiplying $x$ by $y/2$ and then using this to get $xy$. In divide and conquer, we want to divide not numerically, but into *equal amounts of information*. 
If $x = 12345678$ and $y = 24681357$, we can write

\[ x = (1234) \times 10^4 + (5678), \]

\[ y = (2468) \times 10^4 + (1357). \]

Now let’s multiply them.
More generally, say that in base 10,

\[ x = x_{n-1}x_{n-2}\ldots x_1x_0 \quad y = y_{n-1}\ldots y_1y_0. \]

Let

\[ x_h = x_{n-1}\ldots x_{n/2} \quad y_h = y_{n-1}\ldots y_{n/2} \]
\[ x_l = x_{n/2-1}\ldots x_0 \quad y_l = y_{n/2-1}\ldots y_0. \]

Then

\[ x = x_h(10)^{n/2} + x_l \quad y = y_h * (10)^{n/2} + y_l. \]

Then multiply these expressions:

\[ xy = x_hy_h(10^n) + (x_hy_l + x_ly_h)10^{n/2} + x_ly_l. \]
This immediately gives us a recursive algorithm.

1. IF \( n = 1 \), we are multiplying single digit numbers and we can just have a table giving us the answers.
2. Split \( x \) into \( x_h, x_l \) and \( y \) into \( y_h, y_l \).
3. Compute the following recursively \( H = x_h \times y_h, M_1 = x_h \times y_l, M_2 = x_l \times y_h, L = x_l \times y_l \).
4. Append \( n \) 0’s to \( H \) to multiply \( H \) by \( 10^n \).
5. Add \( M_1 \) and \( M_2 \) to get \( M \).
6. Append \( n/2 \) 0’s to \( M \) to multiply \( M \) by \( 10^{n/2} \).
7. Add \( H, M, \) and \( L \) to get the answer.

Correctness is by strong induction, using the formula we used to derive the algorithm.
Any guesses?

What do you think $T(n)$, the time taken by this algorithm, is?

a. $T_{MS}(n) \in \Theta(n^2)$?

b. $T_{MS}(n) \in O(n \log n)$?

c. $T_{MS}(n) \in O(n)$?

d. $T_{MS}(n) \in O(n^{\log_2 3})$?

e. None of the above
1. IF $n=1$, we are multiplying single digit numbers and we can just have a table giving us the answers.

2. Split $x$ into $x_h, x_l$ and $y$ into $y_h, y_l$.

3. Compute the following recursively $H = x_h \times y_h$, $M_1 = x_h \times y_l$, $M_2 = x_l \times y_h$, $L = x_l \times y_l$.

4. Append $n$ 0’s to $H$ to multiply $H$ by $10^n$.

5. Add $M_1$ and $M_2$ to get $M$.

6. Append $n/2$ 0’s to $M$ to multiply $M$ by $10^{n/2}$.

7. Add $H$, $M$, and $L$ to get the answer.

Everything except line 3 takes at most linear time.

In line 3, we make 4 recursive calls to the same algorithm on $n/2$ digit numbers. Thus,

$$T(n) = 4T(n/2) + cn \text{ with } T(1) = c'.$$
Solving the recurrence

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

Unraveling the recursion gives

\[
T(n) = 4T(n/2) + cn \\
= 16T(n/4) + cn + 4(cn/2) = 16T(n/4) + cn + 2cn \\
= 4^3 T(n/2^3) + cn + 2cn + 4cn \\
\vdots \\
= 4^k T(n/2^k) + cn + 2cn + 4cn + 8cn + \ldots + 2^{k-1}cn
\]

We stop unwinding when we \( n/2^k = 1 \), which is the base of the recursion, so at \( k = \log n \). Then we have

\[ T(n) = 4^{\log n} T(1) + cn + 2cn + 4cn + \ldots + 2^{\log n - 1}cn. \]
Solving the recurrence, continued

\[ T(n) = 4^{\log n} T(1) + cn + 2cn + 4cn + \cdots + 2^{\log n - 1} cn \]
\[ = 4^{\log n} c' + cn(1 + 2 + 4 + \cdots + 2^{\log n - 1}) \]

Remember, the same number came up in the Towers of Hanoi:

\[ 1 + 2 + 4 + 8 + \cdots + 2^{t-1} = 2^t - 1. \]

So,

\[ T(n) = 4^{\log n} c' + cn(2^{\log n} - 1) \]
\[ = c' n^2 + cn(n - 1) \]
\[ \in \Theta(n^2). \]
Solving the recurrence, continued

\[ T(n) = 4^{\log n} T(1) + cn + 2cn + 4cn + \cdots + 2^{\log n - 1} cn \]
\[ = 4^{\log n} c' + cn(1 + 2 + 4 + \cdots + 2^{\log n - 1}) \]

Remember, the same number came up in the Towers of Hanoi:
\[ 1 + 2 + 4 + 8 + \cdots + 2^{t-1} = 2^t - 1. \]

So,
\[ T(n) = 4^{\log n} c' + cn(2^{\log n} - 1) \]
\[ = c'n^2 + cn(n - 1) \]
\[ \in \Theta(n^2). \]

All that work and the algorithm is no better than what we already knew!
Historical Aside

Andrey Kolmogorov 1903 - 1987
Anatoly Karatsuba 1937 - 2008
Karatsuba’s big idea

Remember, our previous divide-and-conquer algorithm was based on:

\[ xy = x_h y_h (10^n) + (x_h y_l + x_l y_h)10^{n/2} + x_l y_l \]
\[ = H10^n + (M_1 + M_2)10^{n/2} + L. \]

We used one recursive multiplication to get each term (4 total). Karatsuba saw that we don’t need \( M_1 \) and \( M_2 \) individually, we just need their sum \( M = M_1 + M_2 \). Karatsuba’s trick was to go “outside the box” and perform one multiplication that gave us information about the sum:

\[(x_h + x_l)(y_h + y_l) = x_h y_h + x_h y_l + x_l y_h + x_l y_l \]
\[ = H + M_1 + M_2 + L \]
\[ = H + M + L \]

Since our other two multiplications give \( H \) and \( L \), we can get \( M \) by subtracting them from this value.
IF $n = 1$, we are multiplying single digit numbers and we can just have a table giving us the answers.

Split $x$ into $x_h, x_l$ and $y$ into $y_h, y_l$.

Compute $x_s = x_l + x_h$ and $y_s = y_l + y_h$.

Compute the following recursively $H = x_h \times y_h$, $S = x_s \times y_s$, $L = x_l \times y_l$.

Compute $M$ by subtracting $H$ and $L$ from $S$.

Append $n$ 0’s to $H$ to multiply $H$ by $10^n$.

Append $n/2$ 0’s to $M$ to multiply $M$ by $10^{n/2}$.

Add $H, M, L$ to get the answer.

Correctness is by strong induction, using the formula we used to derive the algorithm.
Any guesses?

What do you think $T(n)$, the time taken by this algorithm, is?

a. $T_{MS}(n) \in O(n^2)$?
b. $T_{MS}(n) \in O(n \log n)$?
c. $T_{MS}(n) \in O(n)$?
d. $T_{MS}(n) \in O(n^{\log_2 3})$?
e. None of the above
Karatsuba’s algorithm time analysis, recurrence

1. IF \( n = 1 \), we are multiplying single digit numbers and we can just have a table giving us the answers.
2. Split \( x \) into \( x_h, x_l \) and \( y \) into \( y_h, y_l \).
3. Compute \( x_s = x_l + x_h \) and \( y_s = y_l + y_h \).
4. Compute the following recursively \( H = x_h \times y_h \), \( S = x_s \times y_s \), \( L = x_l \times y_l \).
5. Compute \( M \) by subtracting \( H \) and \( L \) from \( S \).
6. Append \( n \) 0’s to \( H \) to multiply \( H \) by \( 10^n \).
7. Append \( n/2 \) 0’s to \( M \) to multiply \( M \) by \( 10^{n/2} \).
8. Add \( H, M, \) and \( L \) to get the answer.

Everything except line 4 takes at most linear time.

In line 4, we make 3 recursive calls to the same algorithm on \( n/2 \) digit numbers. Thus,

\[
T(n) = 3T(n/2) + cn \text{ with } T(1) = c'.
\]
In our first take, the recurrence was

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

Karatsuba’s algorithm has the recurrence

\[ T(n) = 3T(n/2) + cn \text{ with } T(1) = c'. \]

This differs from the first take only in that the 4 \(T(n/2)\) has changed to 3 \(T(n/2)\). While constant factors in the non-recursive part only change the result by the same constant factor, we’ll see that changing 4 recursive calls to 3 can make a much bigger difference.
Solving the recurrence

In many ways, the new recurrence is unraveled just like the previous one. But the numbers will be less clean. There is a general theorem, the Master Theorem, that solves recurrences of this general form, which you will learn in Algorithms.

\[ T(n) = 3T(n/2) + cn \text{ with } T(1) = c'. \]

\[
\begin{align*}
T(n) &= 3T(n/2) + cn \\
&= 9T(n/4) + cn + 3(cn/2) = 9T(n/4) + cn + (3/2)cn \\
&= 3^3 T(n/2^3) + cn + (3/2)cn + (3/2)^2 cn \\
&\vdots \\
&= 3^k T(n/2^3) + cn + (3/2)cn + (3/2)^2 cn + \cdots + (3/2)^{k-1} cn \\
&\vdots
\end{align*}
\]

The result is the same, but with 3’s replacing 4’s.
In our first take, the recurrence was

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

which had solution \( T(n) \in \Theta(4^{\log n}) \).

Karatsuba’s algorithm has the recurrence

\[ T(n) = 3T(n/2) + cn \text{ with } T(1) = c', \]

which has solution \( T(n) \in \Theta(3^{\log n}) \).
What is this $3^{\log n}$ function?

We saw earlier that $4^{\log n} = n^2$ so our first take was really $O(n^2)$.

Can we simplify $3^{\log n}$ to better compare Karatsuba’s algorithm to our first take?
What is this $3^{\log n}$ function?

We saw earlier that $4^{\log n} = n^2$ so our first take was really $O(n^2)$.

Can we simplify $3^{\log n}$ to better compare Karatsuba’s algorithm to our first take?

$$3^{\log n} = n^{\log 3} = n^{1.58\ldots} \in o(n^2).$$

So we really do have a faster multiplication algorithm.
Progress since then

1963 Toom and Cook develop a series of algorithms that are time $O(n^{1+\epsilon})$ for an arbitrarily small $\epsilon > 0$.

1971 Schonhage and Strassen reduce integer multiplication to polynomial multiplication and use methods such as the Fast Fourier Transform.

2007 Furer uses number theory to achieve the best known time for multiplication.

2015 It is still open whether there is a linear time algorithm for multiplication.