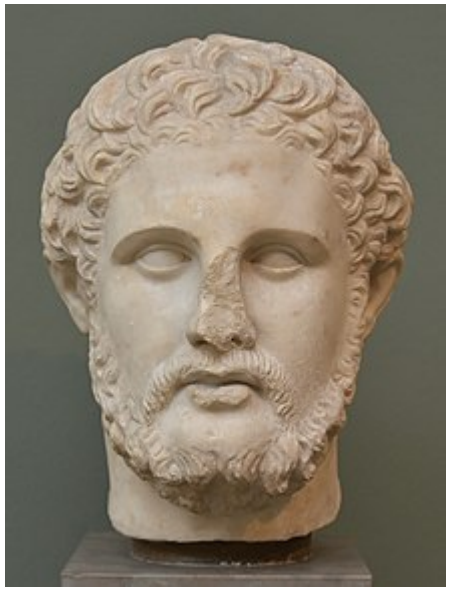
Lecture 2: Divide And Conquer



Section 1

Instructor Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000sylabus

Some business

No complaints about watching lectures via Canvas, going to keep doing it this way for now.

- Have fixed the layout so only the screen should be recorded
- Sharing screen directly from my iPad now, should go more smoothly (fingers crossed!)

Homework 1 to be released this evening; we will talk a bit about it at the end.

Decided against Discord/Slack, but also realized Canvas “discussions” are not full featured

- I will set up a Piazza instead (very sorry to do this late and add another thing!)

Student → TA assignment to come

Today

Some common growth functions, plotted

Loop invariants, take 2

We break things off with BubbleSort (feat. bad memes)

Introduction to Divide and Conquer

Very brief LaTeX “demo”

From last time: Asymptotes and Runtimes

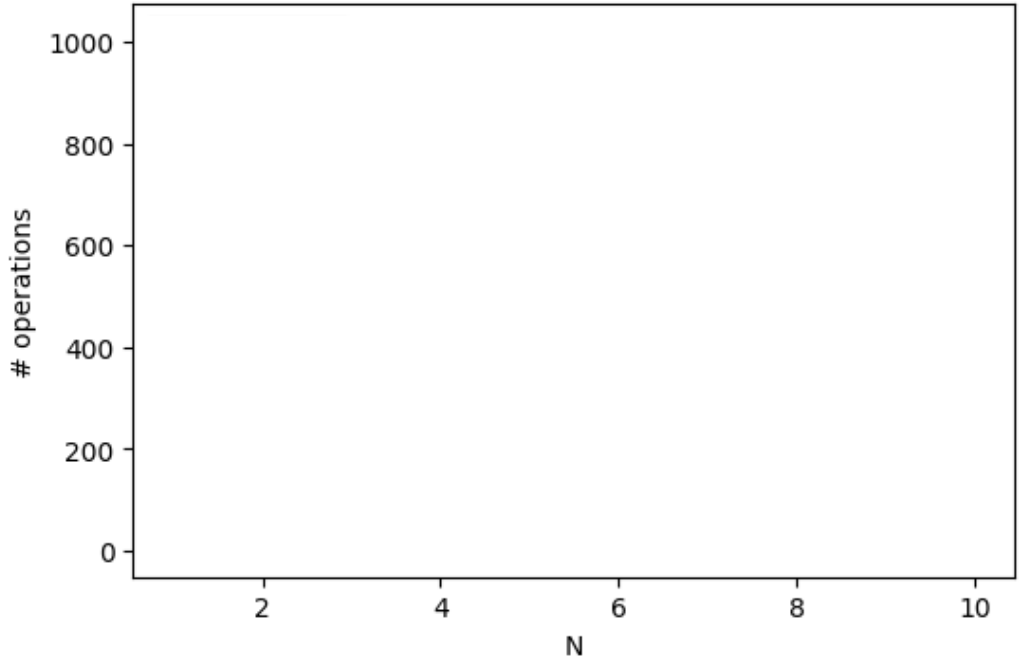
"…an **asymptote** (/ˈæsɪmptoʊt/) of a curve is a line such that the distance between the curve and the line approaches zero as one or both of the *x* or *y* coordinates tends to infinity." – Asymptote on Wikipedia

What do asymptotes have to do with algorithms?

From last time: Asymptotes and Runtimes

"…an **asymptote** (/ˈæsɪmptoʊt/) of a curve is a line such that the distance between the curve and the line approaches zero as one or both of the *x* or *y* coordinates tends to infinity." – Asymptote on Wikipedia

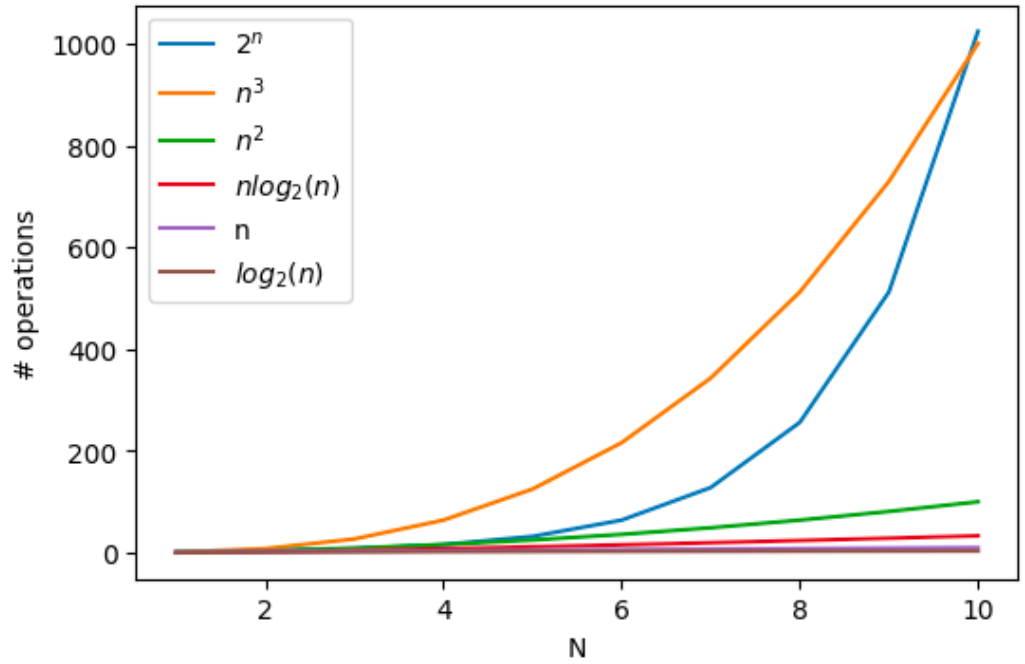
What do asymptotes have to do with algorithms?



From last time: Asymptotes and Runtimes

"…an **asymptote** (/ˈæsɪmptoʊt/) of a curve is a line such that the distance between the curve and the line approaches zero as one or both of the *x* or *y* coordinates tends to infinity." – Asymptote on Wikipedia

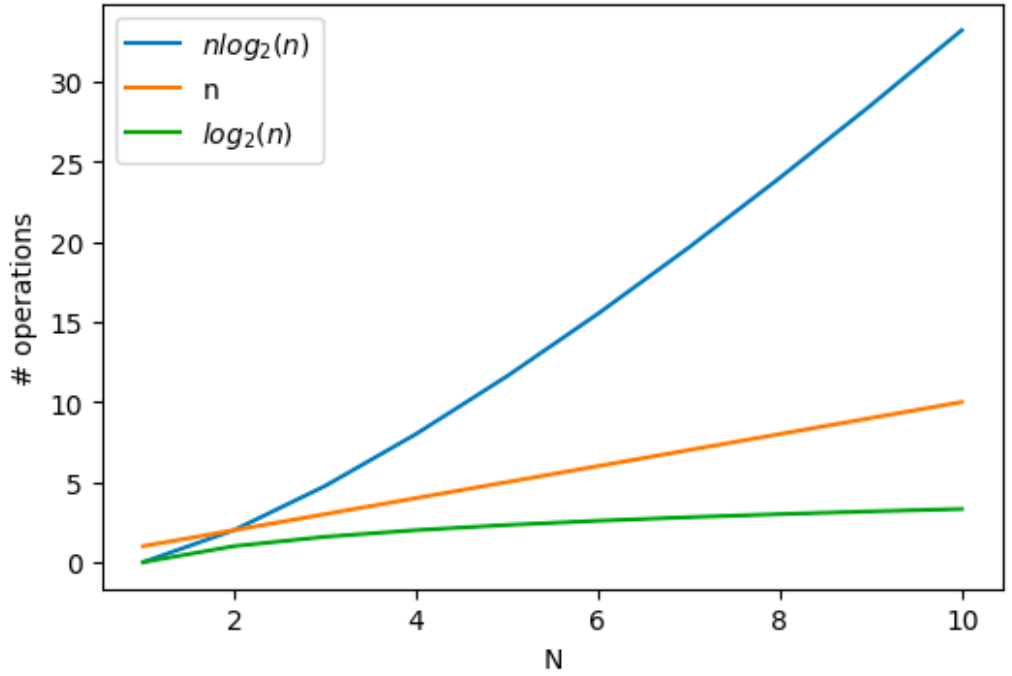
What do asymptotes have to do with algorithms?



From last time: Asymptotes and Runtimes

"…an **asymptote** (/ˈæsɪmptoʊt/) of a curve is a line such that the distance between the curve and the line approaches zero as one or both of the *x* or *y* coordinates tends to infinity." – Asymptote on Wikipedia

What do asymptotes have to do with algorithms?



From last time: Sorting

Sorting is extremely important to computer users and scientists!

A simple example: Finding the median of a set of numbers

```
Input: L, a list of N numbers
Output: The median of L
Procedure:
    1. Sort L
    2. If N is odd, return the number at L[⌈N/2⌉]
    3. If N is even, return the mean of the
       numbers at L[⌈N/2⌉] and L[⌈N/2⌉+1]
```

From last time: Bubble Sort

Idea: Items “bubble up” to the top as they are sorted pairwise

```
Input: L, a list of N numbers
Output: L sorted in ascending order
Procedure:
    Let swapped = True
    while swapped = True:
        swapped = False
        for i from 1 to N-1:
            if L[i] > L[i+1]:
                Swap L[i] and L[i+1]
                swapped = True
```

Loop Invariant Definition

A loop invariant is a formal statement about the relationship between variables in [an algorithm] which holds true just before the loop is ever run (**establishing the invariant**) and is true again at the bottom of the loop, each time through the loop (**maintaining the invariant**).

Definition source: https://www.cs.miami.edu/home/burt/learning/Math120.1/Notes/LoopInvar.html

Loop Invariant Definition

A loop invariant is a formal statement about the relationship between variables in [an algorithm] which holds true just before the loop is ever run (**establishing the invariant**) and is true again at the bottom of the loop, each time through the loop (**maintaining the invariant**).

```
Input: L, a list of N numbers
Output: L sorted in ascending order
Procedure:
    Let swapped = True
    while swapped = True:
        swapped = False
        for i from 1 to N-1:
            if L[i] > L[i+1]:
                Swap L[i] and L[i+1]
                swapped = True
```

Definition source: https://www.cs.miami.edu/home/burt/learning/Math120.1/Notes/LoopInvar.html

Loop Invariant Definition

A loop invariant is a formal statement about the relationship between variables in [an algorithm] which holds true just before the loop is ever run (**establishing the invariant**) and is true again at the bottom of the loop, each time through the loop (**maintaining the invariant**).

```
Input: L, a list of N numbers
Output: L sorted in ascending order
Procedure:
    Let swapped = True
    while swapped = True:
        swapped = False
        for i from 1 to N-1:
            if L[i] > L[i+1]:
                Swap L[i] and L[i+1]
                swapped = True
```

Bubble sort loop invariant: After every iteration, the largest previously unsorted value is in its correct position.

Definition source: https://www.cs.miami.edu/home/burt/learning/Math120.1/Notes/LoopInvar.html

Loop Invariant Definition

A loop invariant is a formal statement about the relationship between variables in [an algorithm] which holds true just before the loop is ever run (**establishing the invariant**) and is true again at the bottom of the loop, each time through the loop (**maintaining the invariant**).

```
Input: L, a list of N numbers
Output: L sorted in ascending order
Procedure:
    Let swapped = True
    while swapped = True:
        swapped = False
        for i from 1 to N-1:
            if L[i] > L[i+1]:
                Swap L[i] and L[i+1]
                swapped = True
```

Bubble sort loop invariant: After every iteration, the largest previously unsorted value is in its correct position.

After *m* iterations of the while loop, the *m* largest values are in their correct positions.

Definition source: https://www.cs.miami.edu/home/burt/learning/Math120.1/Notes/LoopInvar.html

Loop Invariant Definition

A loop invariant is a formal statement about the relationship between variables in [an algorithm] which holds true just before the loop is ever run (**establishing the invariant**) and is true again at the bottom of the loop, each time through the loop (**maintaining the invariant**).

```
Input: L, a list of N numbers
Output: L sorted in ascending order
Procedure:
    Let swapped = True
    while swapped = True:
        swapped = False
        for i from 1 to N-1:
            if L[i] > L[i+1]:
                Swap L[i] and L[i+1]
                swapped = True
```

Bubble sort loop invariant: After every iteration, the largest previously unsorted value is in its correct position.

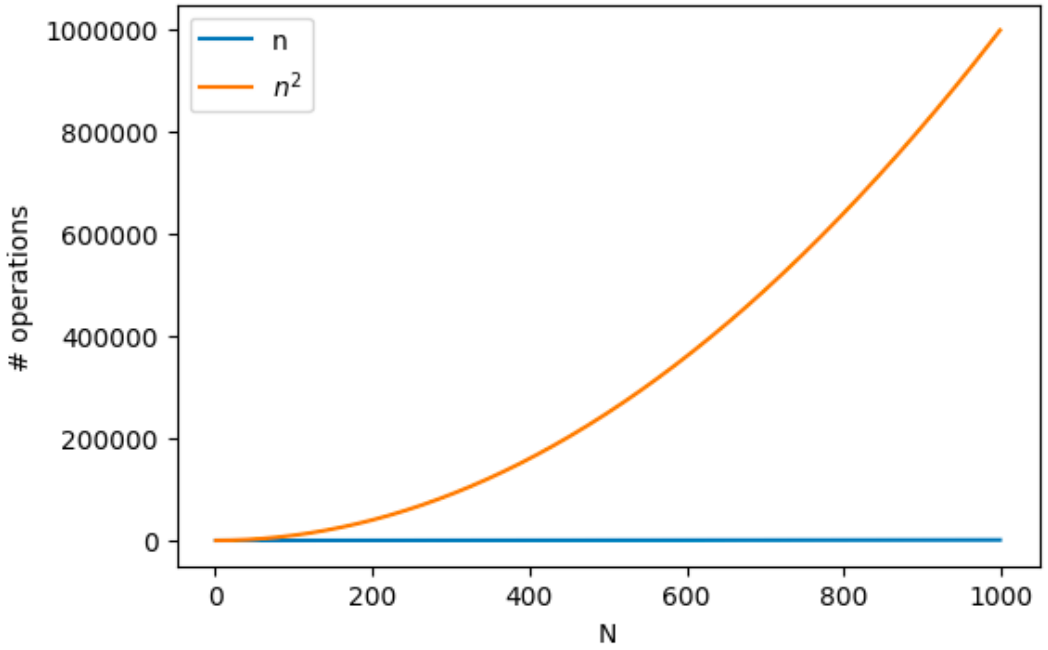
After *m* iterations of the while loop, the *m* largest values are in their correct positions.

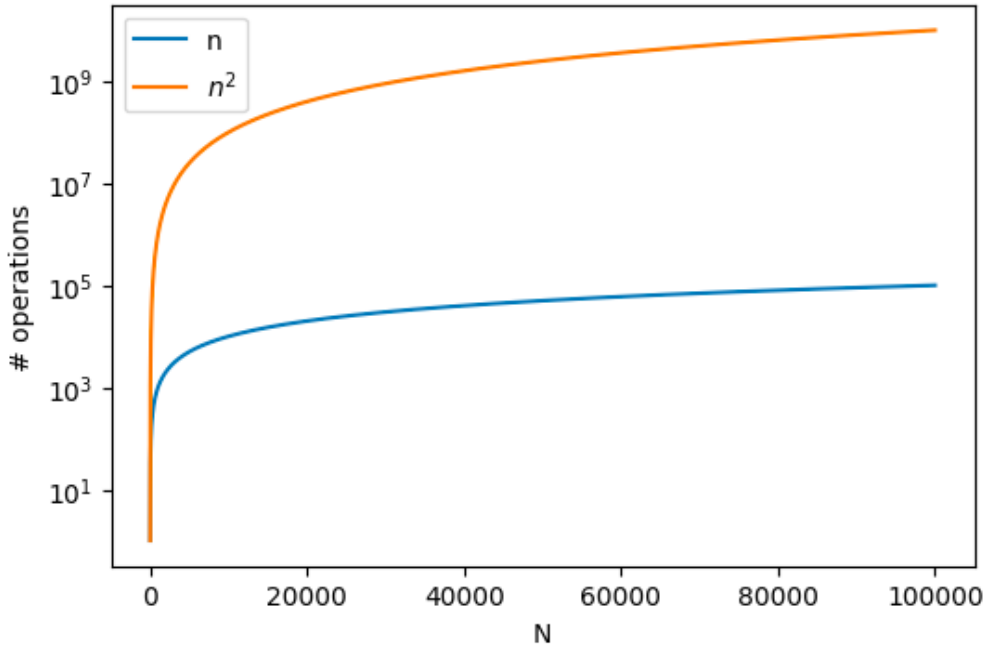
After *n* iterations, all values are in their correct positions.

Definition source: https://www.cs.miami.edu/home/burt/learning/Math120.1/Notes/LoopInvar.html

Dumping BubbleSort: $O(n^2)$ is just not practical!

Dumping BubbleSort: $O(n^2)$ is just not practical!





Dumping BubbleSort: $O(n^2)$ is just not practical!

Enter: Divide and Conquer

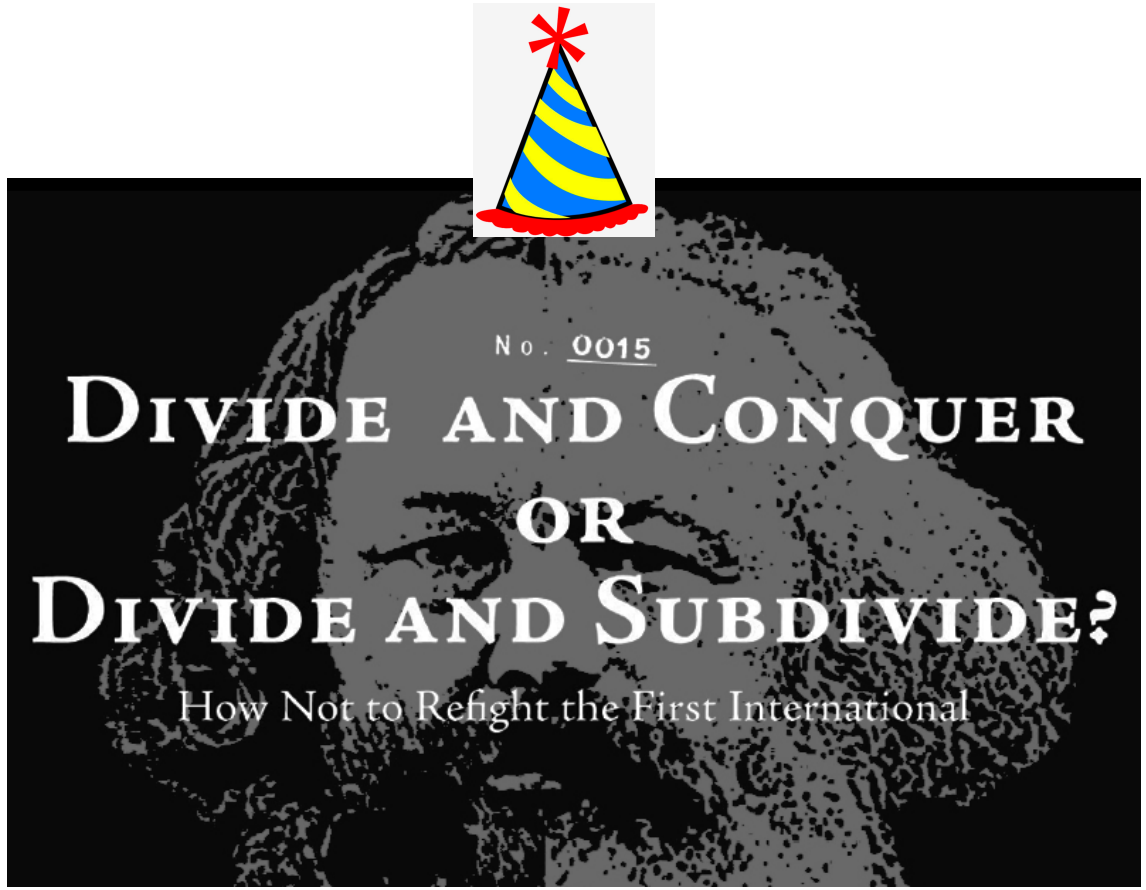
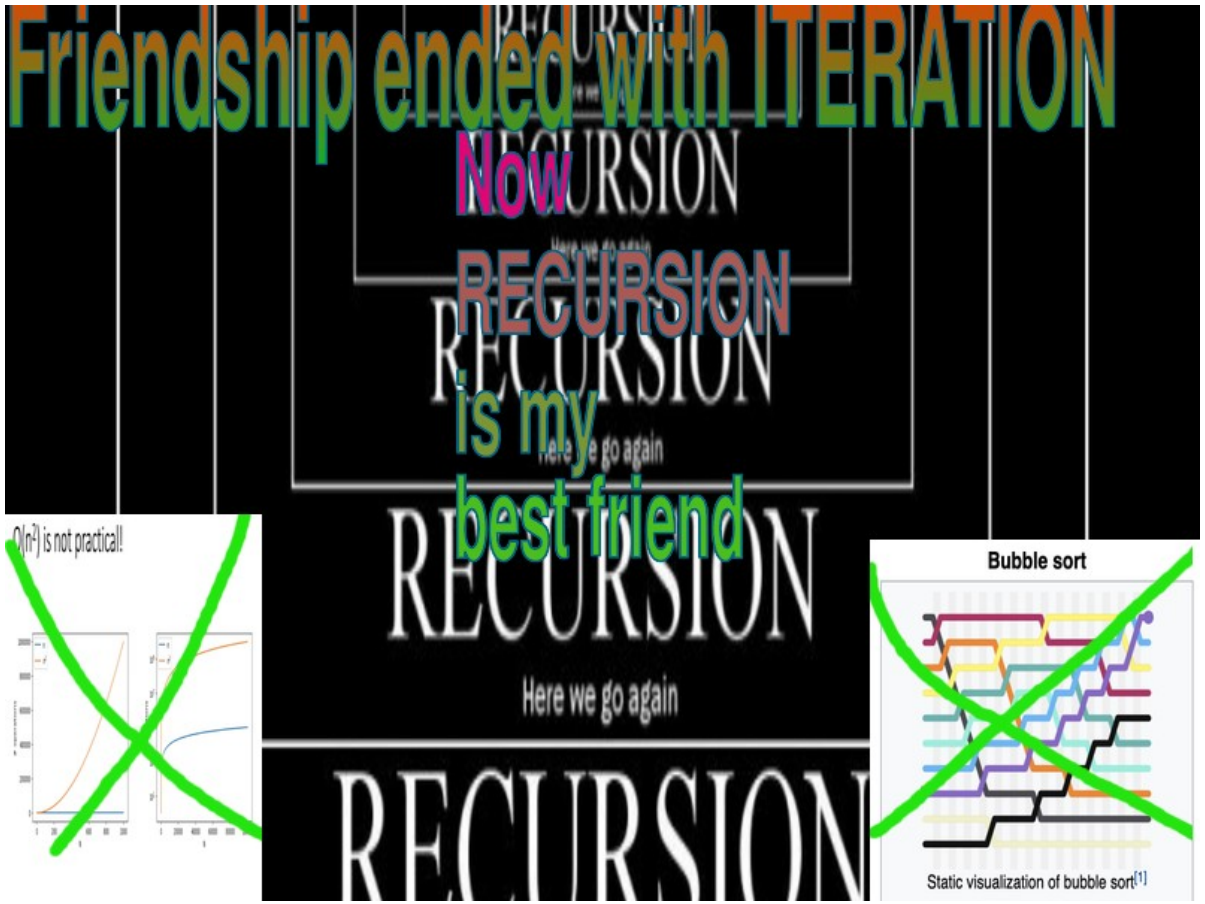


Image credit: libcom.org

What if….

Instead of sorting the entire input at once (as in bubble sort)….



…we could break the problem into smaller pieces to be sorted separately?

Merge Sort

Idea: Speed up sorting by splitting the input in half, sorting the smaller pieces separately, then merging the output.

Merge Sort

Idea: Speed up sorting by splitting the input in half, sorting the smaller pieces separately, then merging the output.

```
MERGESORT(A[1..n]):
  if n > 1
    m ← ⌊n/2⌋
    MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
    MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
    MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i+1
    else if i > m
      B[k] ← A[j]; j ← j+1
    else if A[i] < A[j]
      B[k] ← A[i]; i ← i+1
    else
      B[k] ← A[j]; j ← j+1

  for k ← 1 to n
    A[k] ← B[k]
```

Erickson book section 1.4

Merge Sort Example

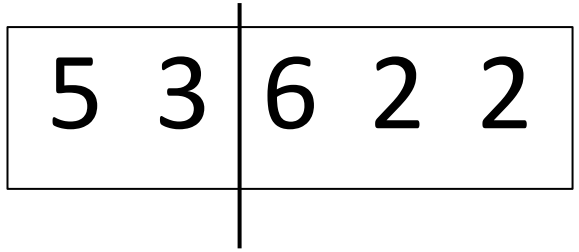
```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

5	3	6	2	2

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

Merge Sort Example

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

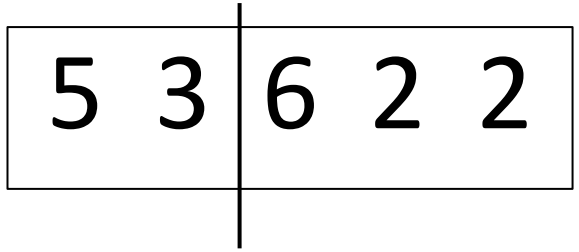


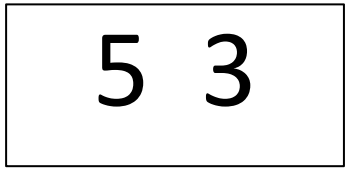

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

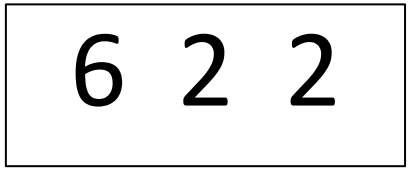
Merge Sort Example

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```



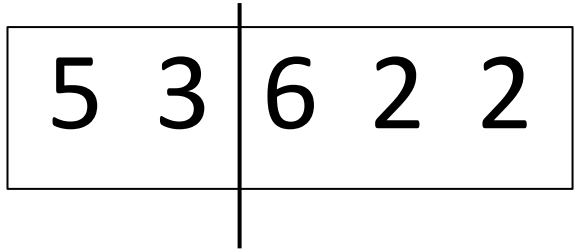


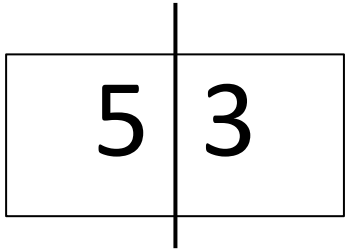


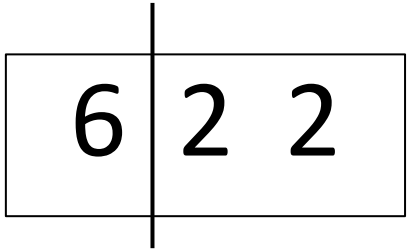
Merge Sort Example

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```



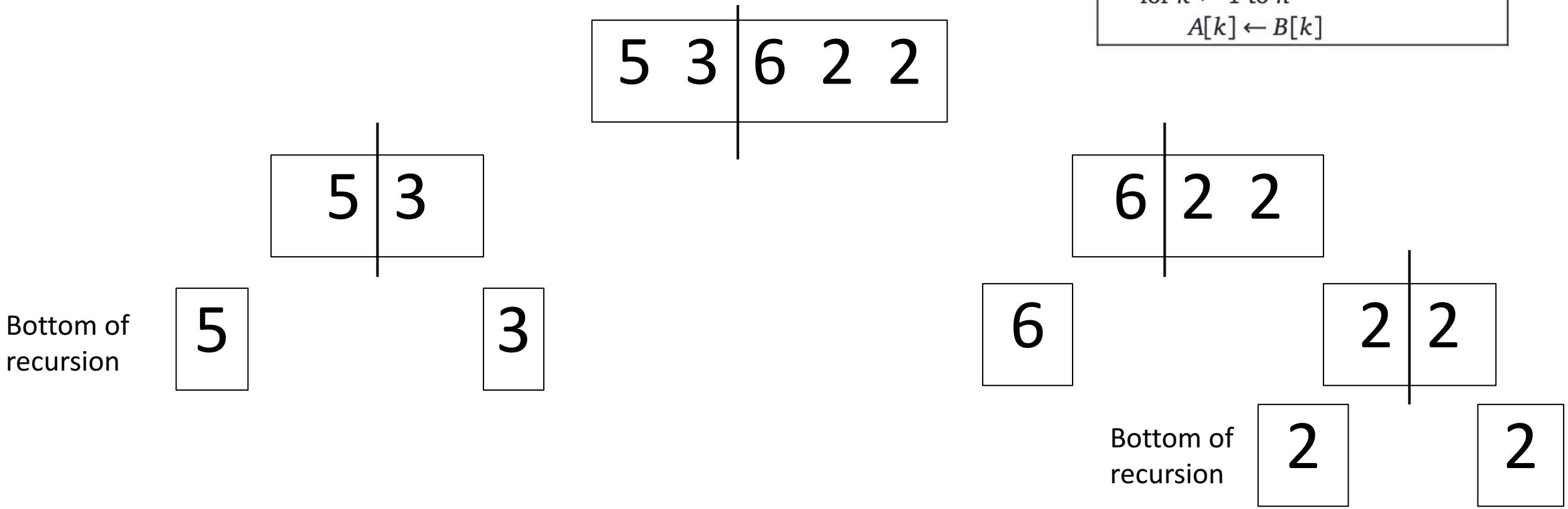




Merge Sort Example

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

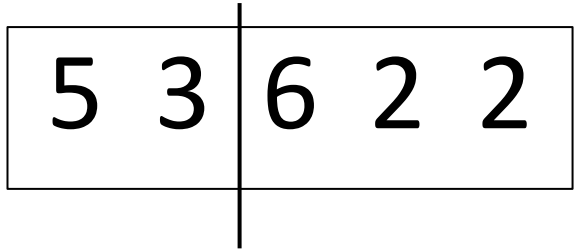
```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```



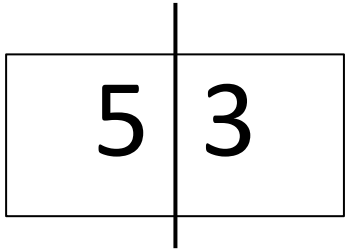
Merge Sort Example

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

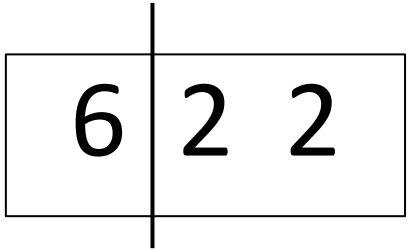


MERGE

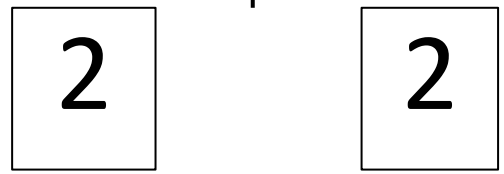


Bottom of recursion

MERGE



Bottom of recursion



Merge Sort Example

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

5	3	6	2	2

MERGE

3	5	2	2	6

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

Proof of Correctness

We can show formally that the output of MergeSort is correct by using 2 *proofs by induction*!

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

Erickson book section 1.4

Proof by Induction Reminder

3 main steps to a proof by induction:

Merge Sort: Proof of Correctness

First show that MERGE is correct, then MergeSort.

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

Merge: Proof of Correctness

```
MERGE(A[1..n], m):
  i ← 1;  j ← m + 1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1
      else if i > m
          B[k] ← A[j];  j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1
      else
          B[k] ← A[j];  j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
```

We will show that for all k from 0 to n, the last n-k-1 iterations of the main loop correctly merge A[i..n] and A[j..m] into B[k..n].

Base case:

Merge: Proof of Correctness

```
MERGE(A[1..n], m):
  i ← 1;  j ← m + 1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1
      else if i > m
          B[k] ← A[j];  j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1
      else
          B[k] ← A[j];  j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
```

We will show that for all k from 0 to n, the last n-k-1 iterations of the main loop correctly merge A[i..n] and A[j..m] into B[k..n].

Inductive Hypothesis:

Merge: Proof of Correctness

```
MERGE(A[1..n], m):
  i ← 1;  j ← m + 1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1
      else if i > m
          B[k] ← A[j];  j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1
      else
          B[k] ← A[j];  j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
```

We will show that for all k from 0 to n, the last n-k-1 iterations of the main loop correctly merge A[i..n] and A[j..m] into B[k..n].

Proof:

MergeSort: Proof of Correctness

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

Base Case:

Inductive Hypothesis:

Proof:

MergeSort: Runtime Analysis

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])          ⟨⟨Recurse!⟩⟩
      MERGESORT(A[m+1..n])        ⟨⟨Recurse!⟩⟩
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i+1
      else if i > m
          B[k] ← A[j];  j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i+1
      else
          B[k] ← A[j];  j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

Let's write down a *recurrence relation* that describes the runtime:

Next Time

Recurrence Relations + Recurrence Trees

Formal Asymptotic Analysis

More Divide & Conquer

Suggested Readings:

Now: Brief LaTeX “demo”