Sorting

Elyse Cornwall

August 3, 2023

Contributions made from previous CS106B Instructors

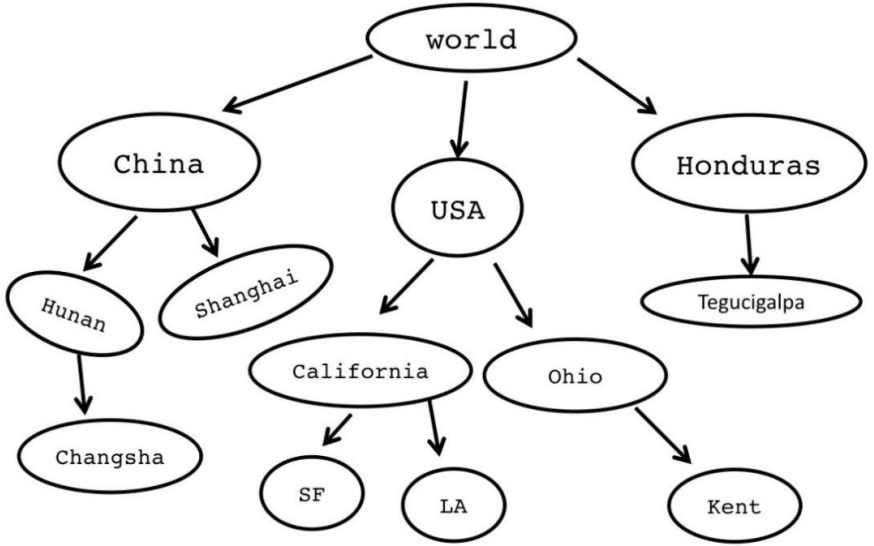
Announcements

- Assignment 5: Linked Lists is out, due next Wednesday
  - This is the penultimate assignment 😢
- Change of grading basis deadline is tomorrow at 5pm

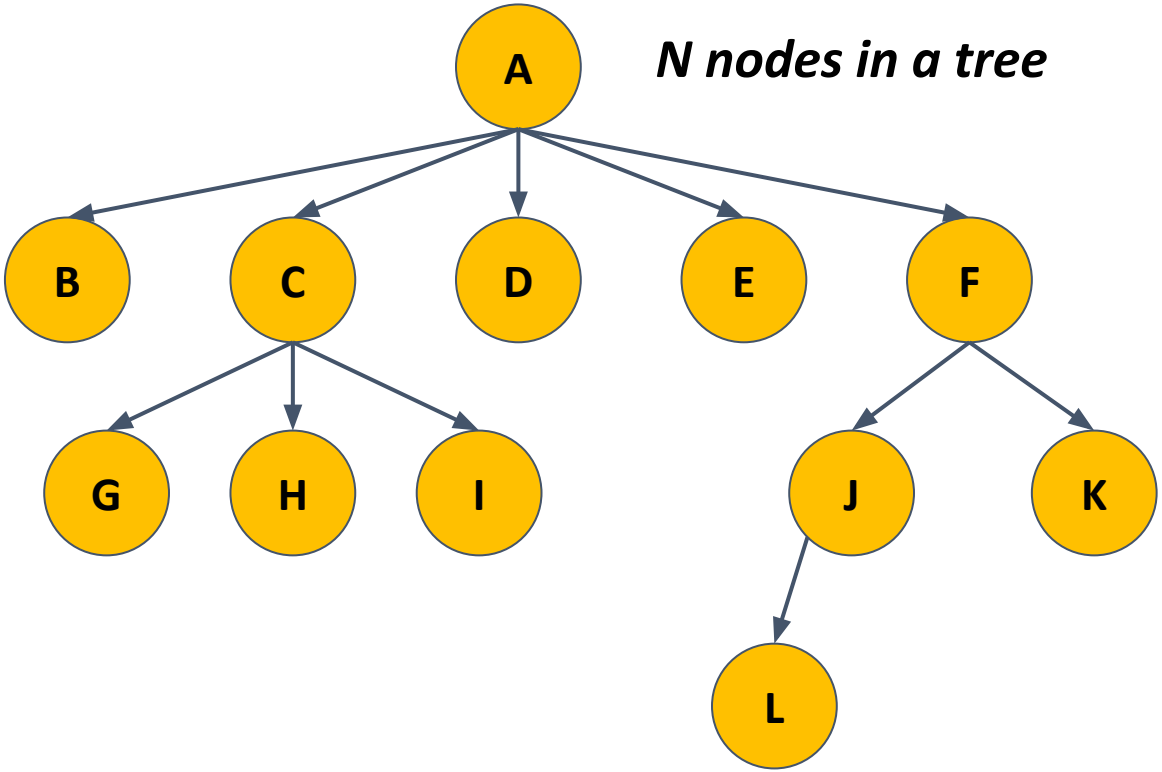
Recap: Trees

Uses

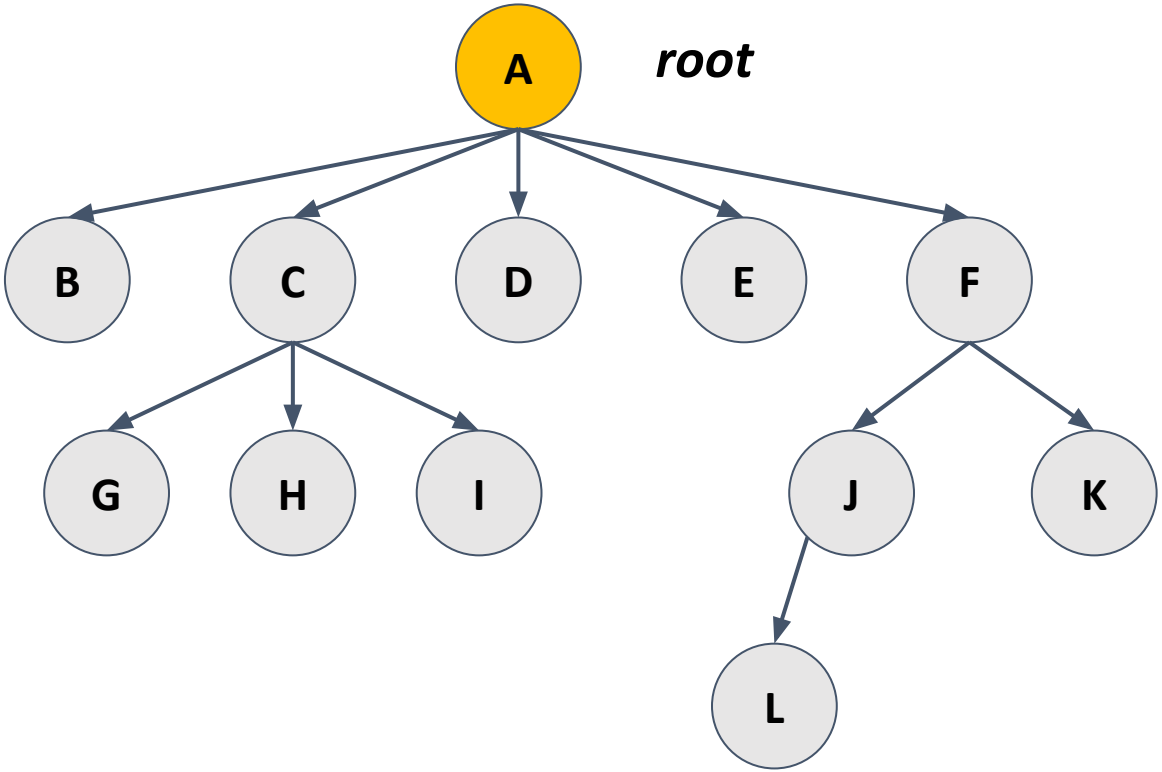
- Trees are useful in other ways besides visualizing recursion and modeling priority
  - Describe hierarchies



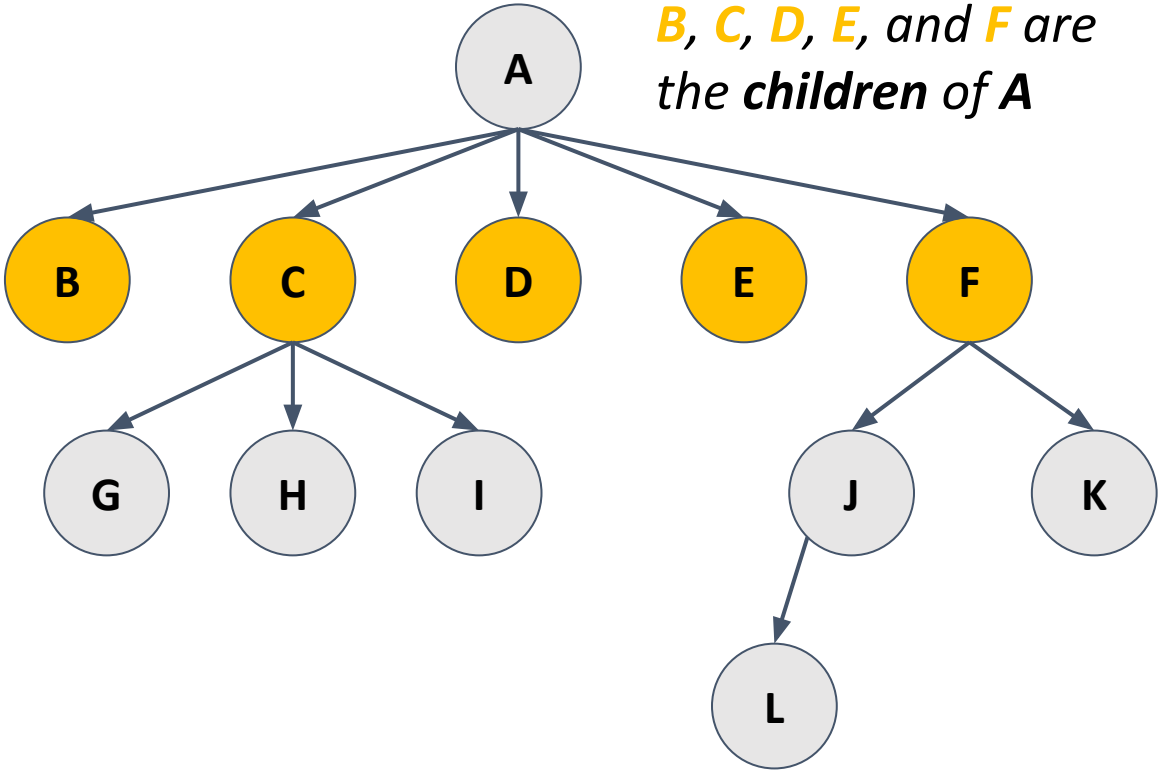
New Tree Terminology



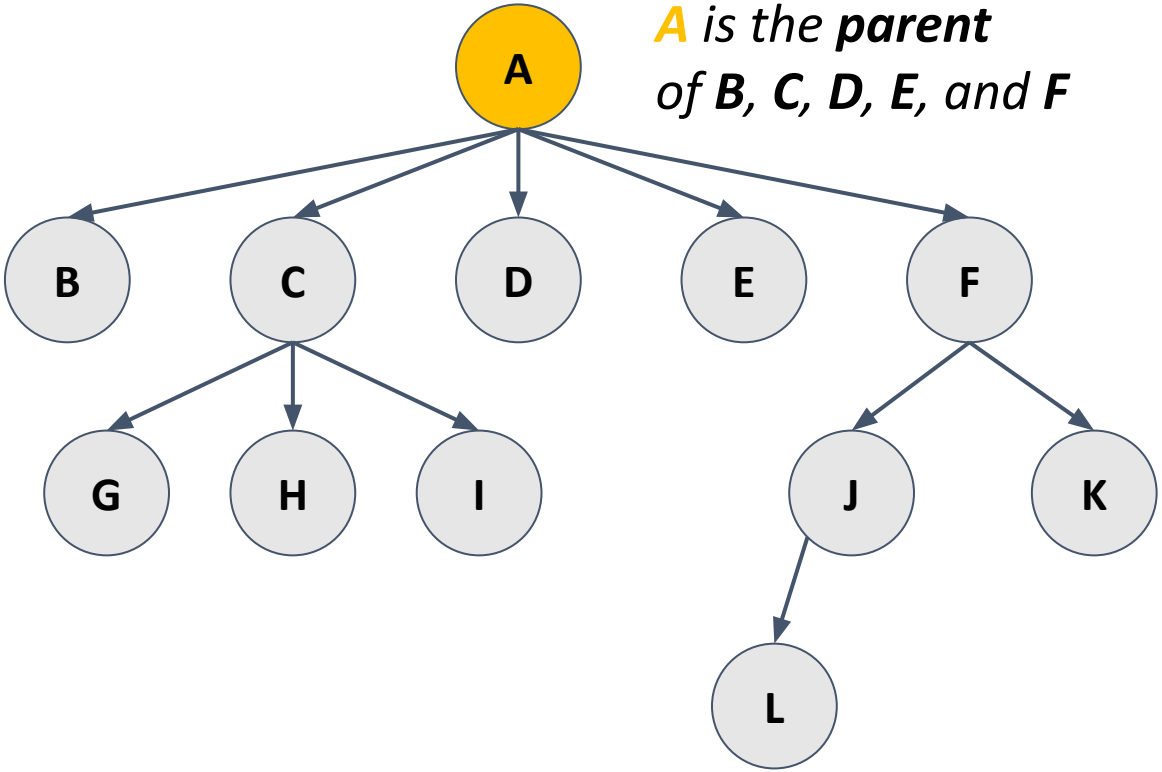
New Tree Terminology



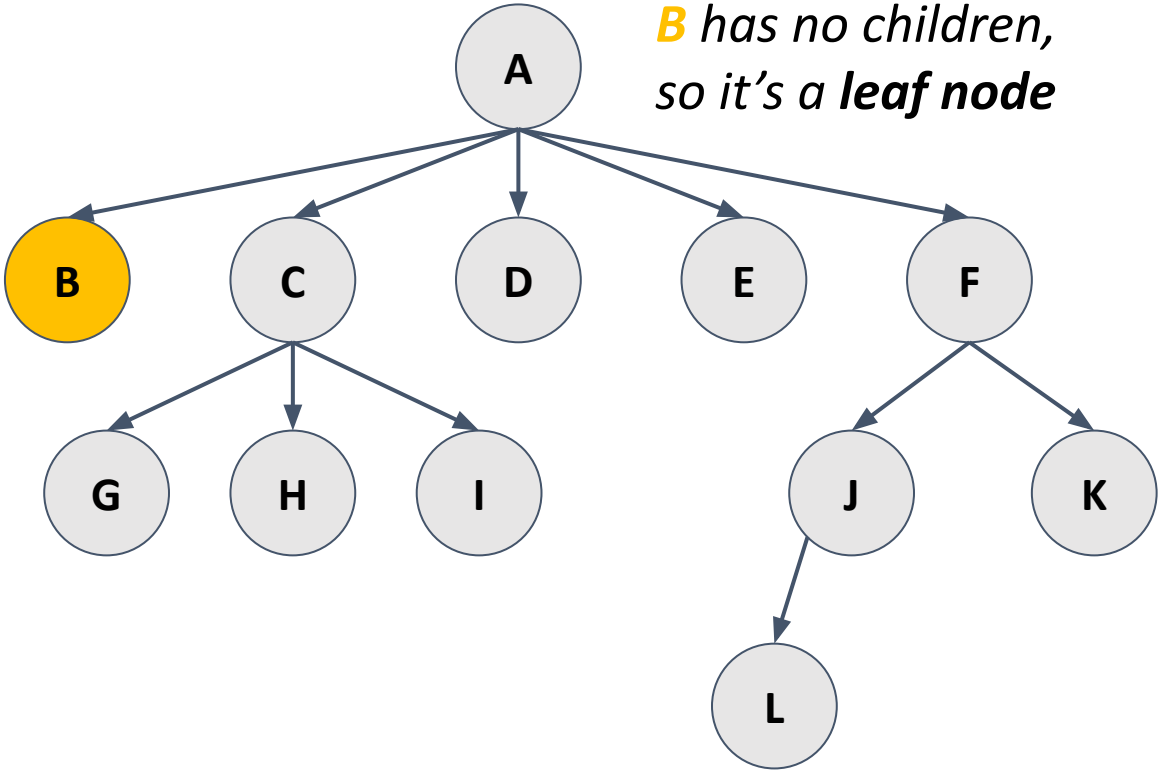
New Tree Terminology



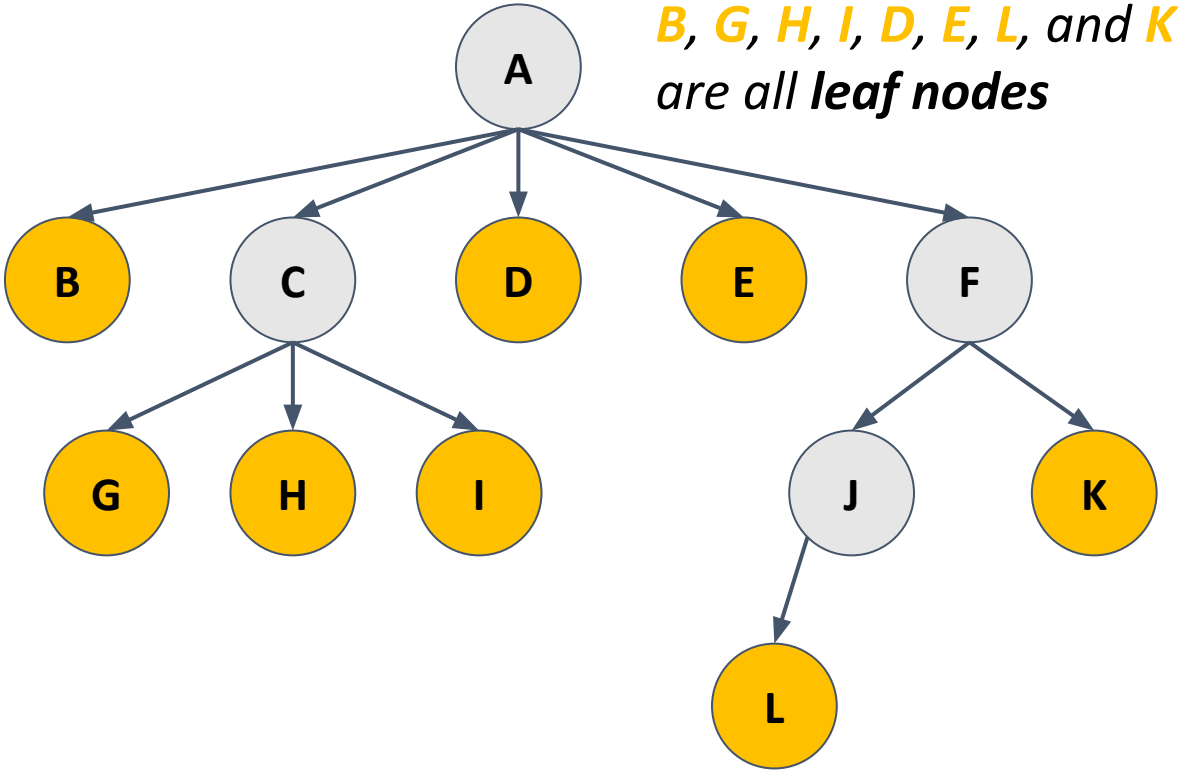
New Tree Terminology



New Tree Terminology



New Tree Terminology



B, *G*, *H*, *I*, *D*, *E*, *L*, and *K* are all ***leaf nodes***

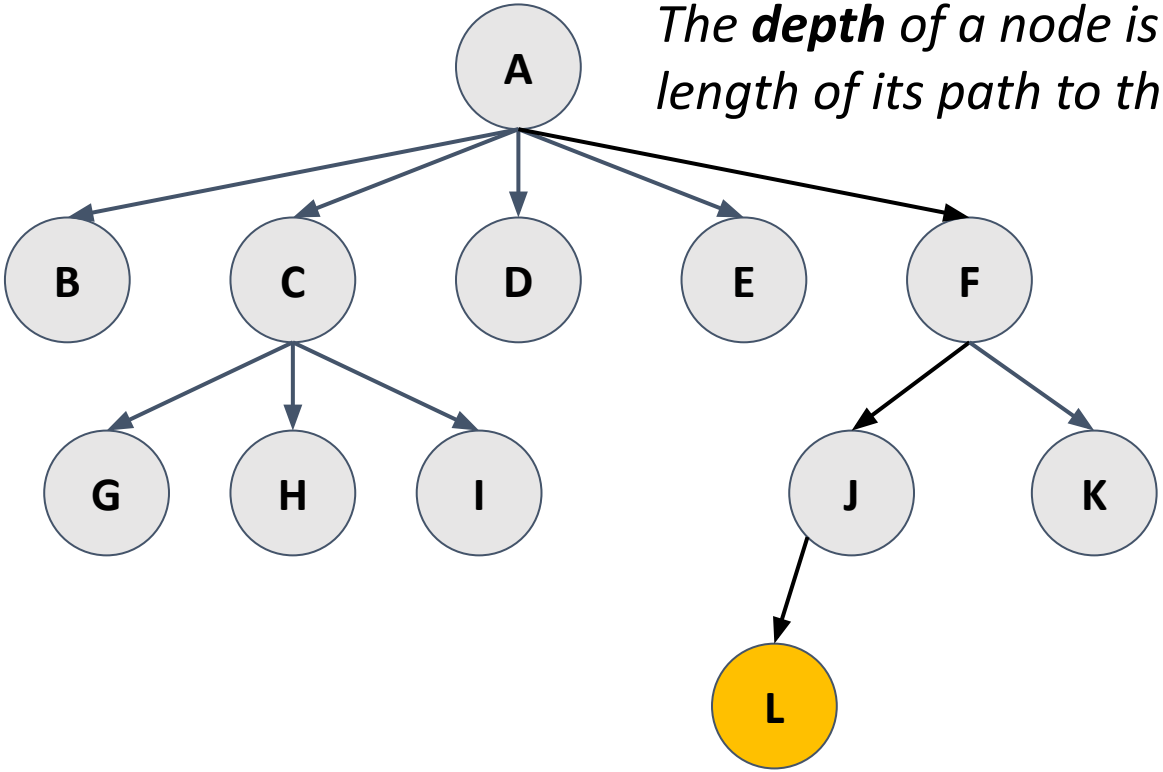
New Tree Terminology

depth: 0

depth: 1

depth: 2

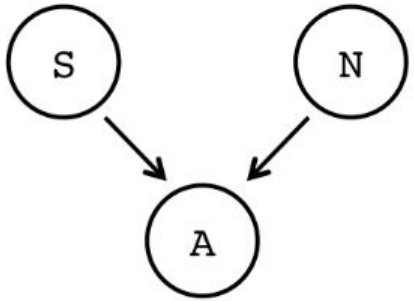
depth: 3



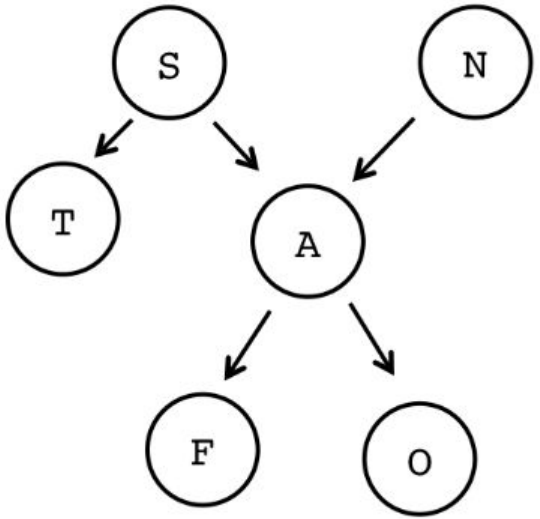
*The **depth** of a node is the length of its path to the root*

Tree Properties

- Any node in a tree can only have one parent

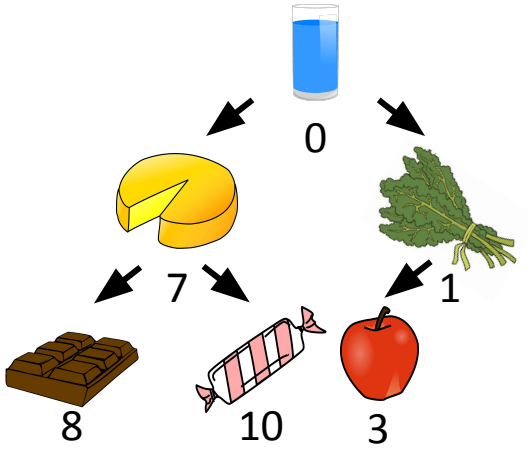


Not trees!



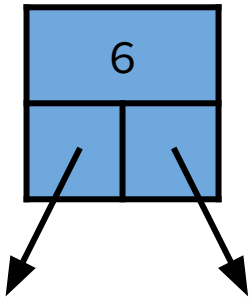
Binary Trees

- Today, we've seen that nodes in a tree can have a variable amount of children (subtrees)
- Previously, we've worked with binary trees
  - Most common trees in CS
  - Every node has either 0, 1, or 2 children
  - No node may have more than 2 children
  - Children are referred to as left child and right child



Building Binary Trees

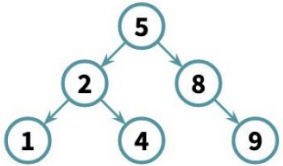
- A binary tree is composed of nodes
- Each node is a struct that contains:
  - A piece of data (like an int, or string)
  - A pointer to the left child
  - A pointer to the right child




```
struct TreeNode {
   int data;
   TreeNode* left;
   TreeNode* right;
};
```

Tree Traversal Recap

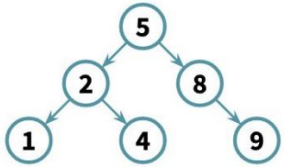
Pre-order



do something (aka cout)
traverse left subtree
traverse right subtree

5 2 1 4 8 9

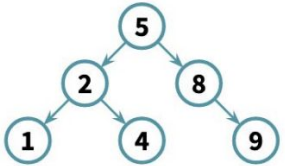
In-order



traverse left subtree
do something (aka cout)
traverse right subtree

1 2 4 5 8 9

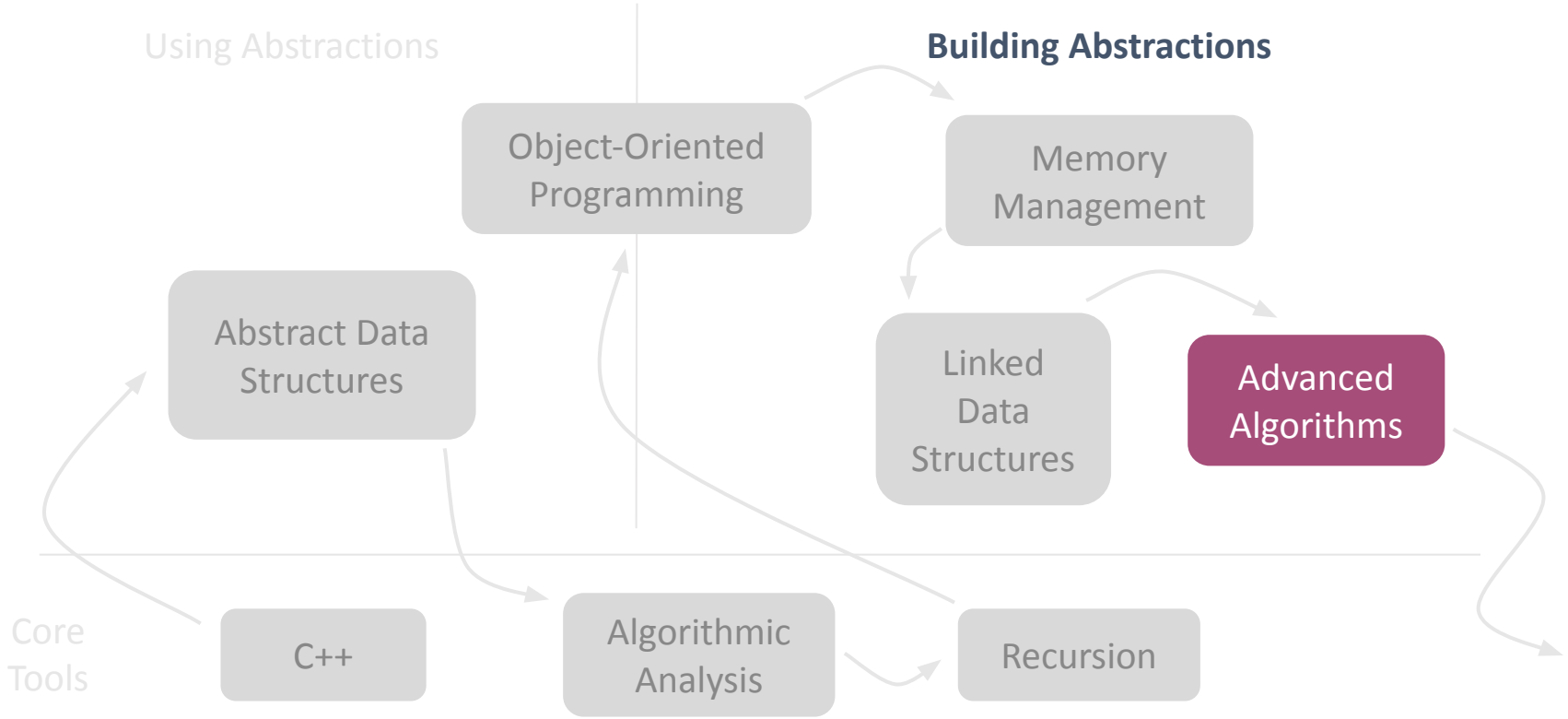
Post-order



traverse left subtree
traverse right subtree
do something (aka cout)

1 4 2 9 8 5

Roadmap

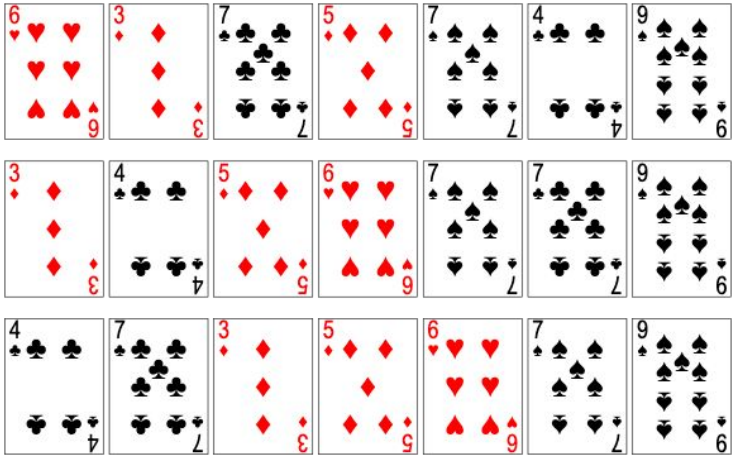

Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Sorting

Motivating sorting algorithms

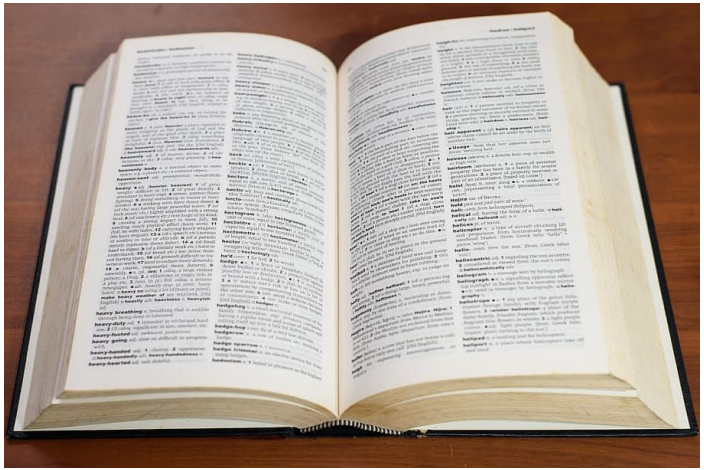
Sorting

- Goal: given some data points, arrange those data points into ascending/descending order by some quantity
  - E.g. sort cards by face value or suit



Sorting

- Sorted data is often easier to work with
- Sorted data can allow for faster insert/retrieval/deletion



Sorting

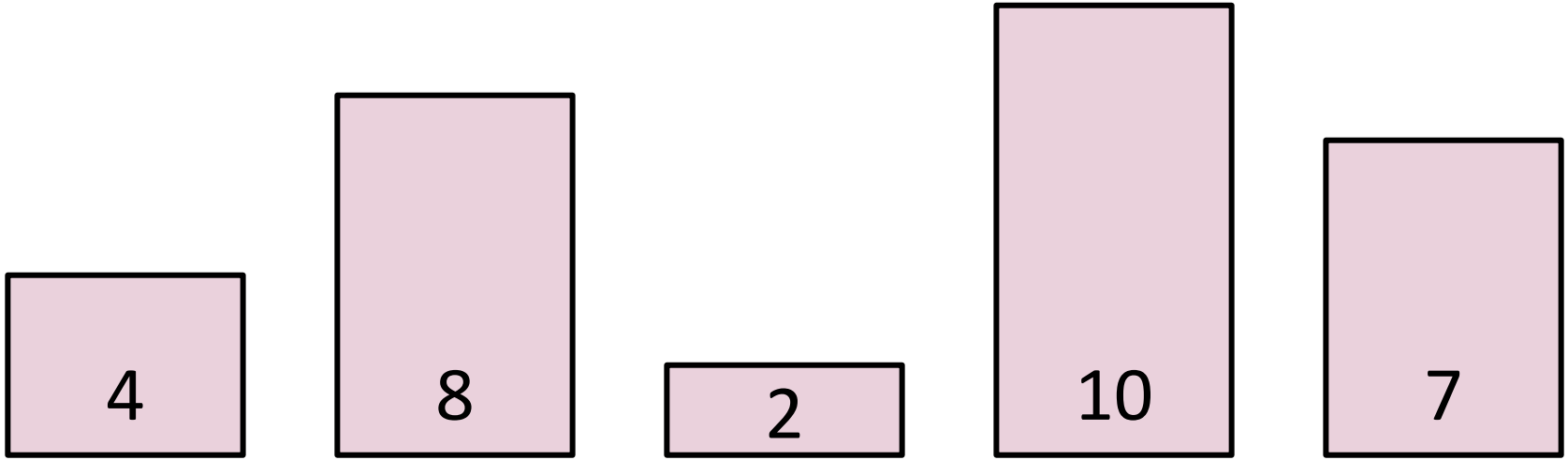
- Today we'll investigate and compare different sorting algorithms
- Motivating questions:
  - What are the different ways we can sort data?
  - What's the "best" strategy?

Selection Sort

Our first sorting algorithm

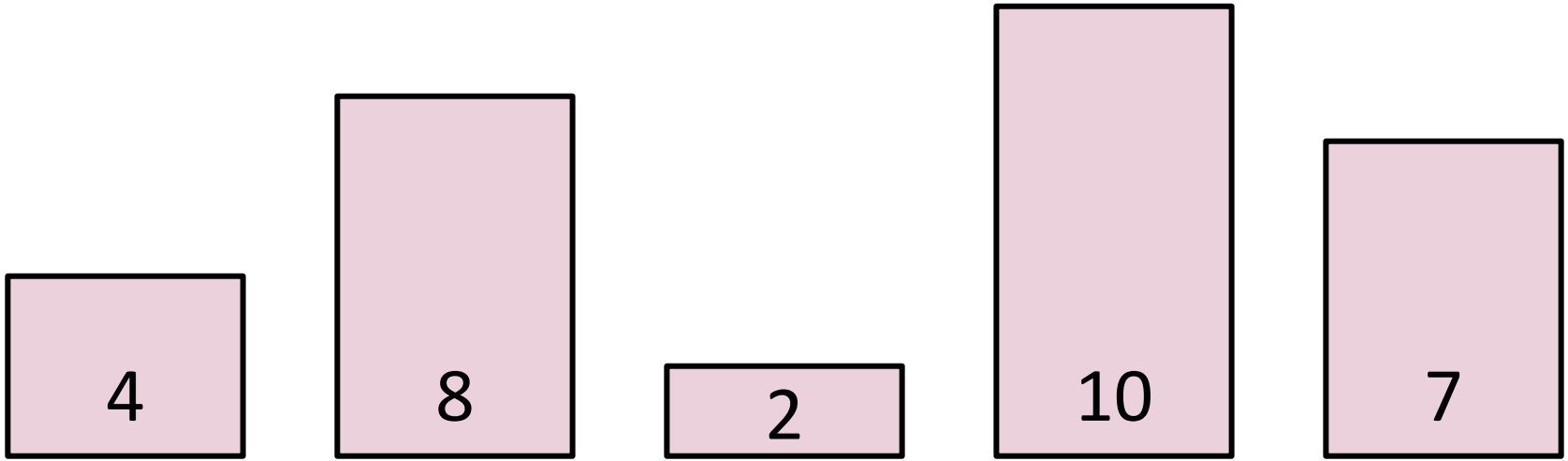
Selection Sort

- Let's say we have the following elements, that we'd like to sort in ascending numerical order



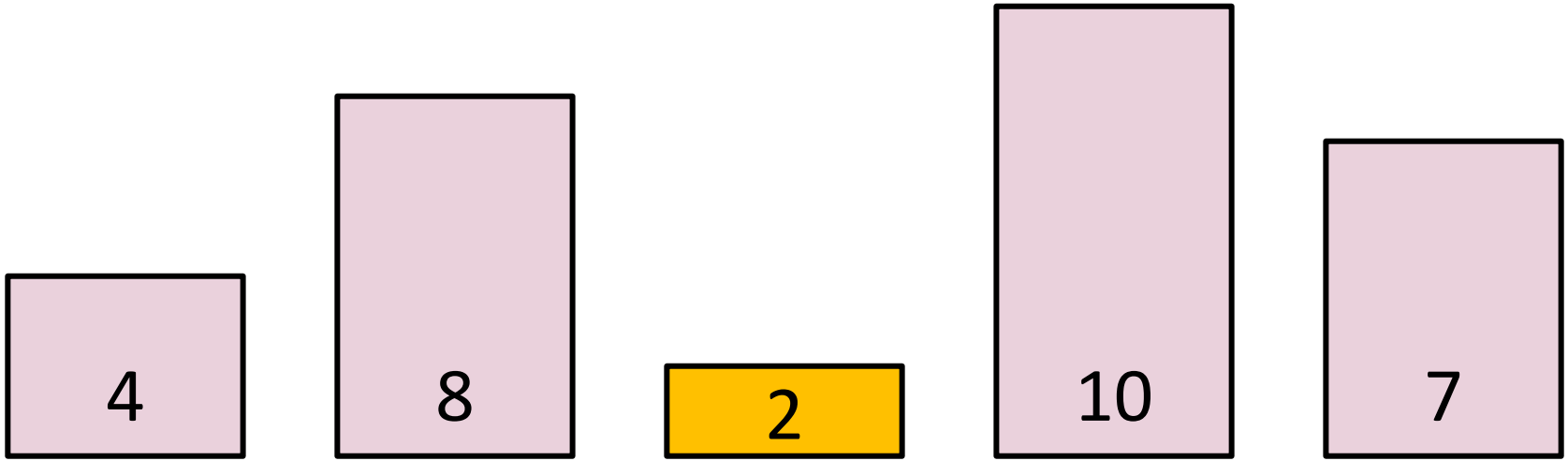
Selection Sort

- Idea: find the smallest element, put it in front of other elements



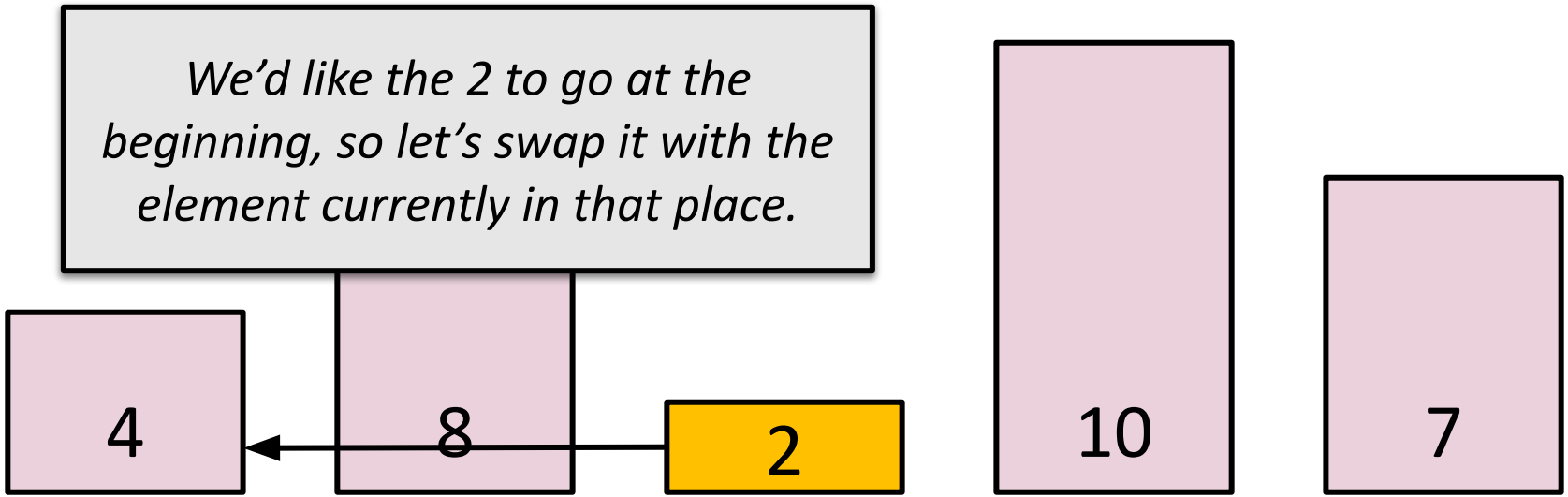
Selection Sort

- Idea: find the smallest element, put it in front of other elements



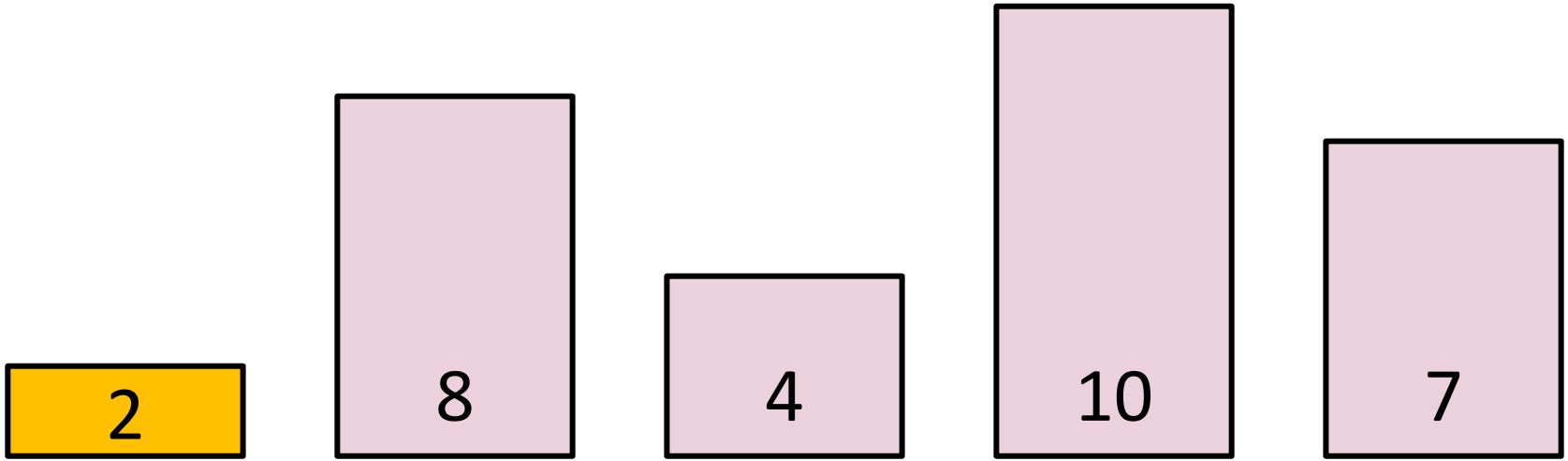
Selection Sort

- Idea: find the smallest element, put it in front of other elements



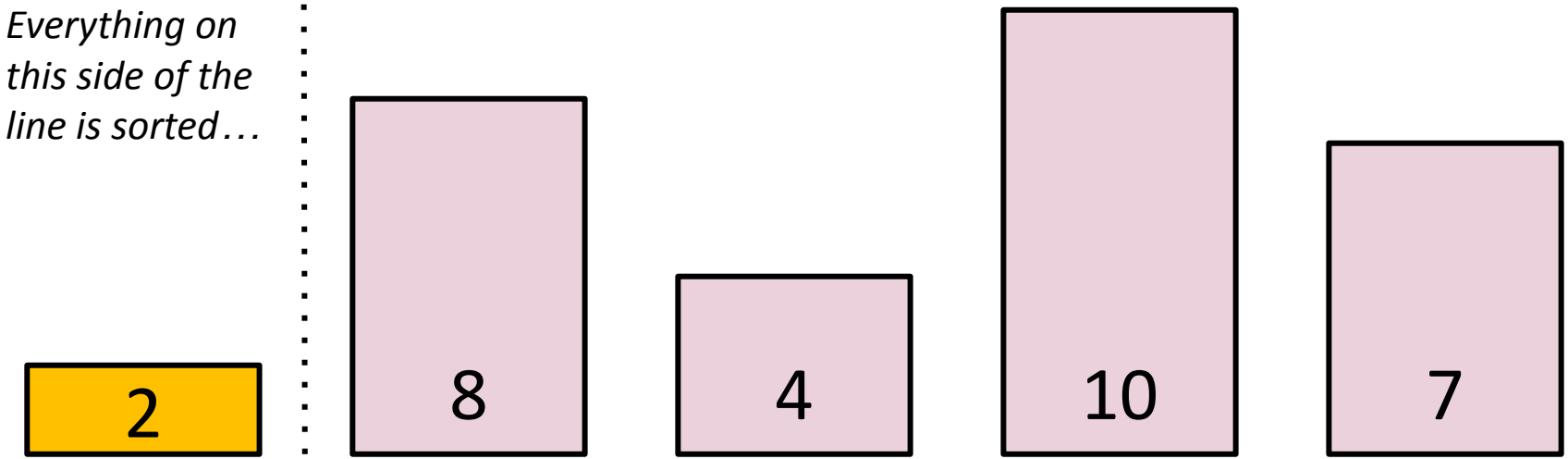
Selection Sort

- Idea: find the smallest element, put it in front of other elements



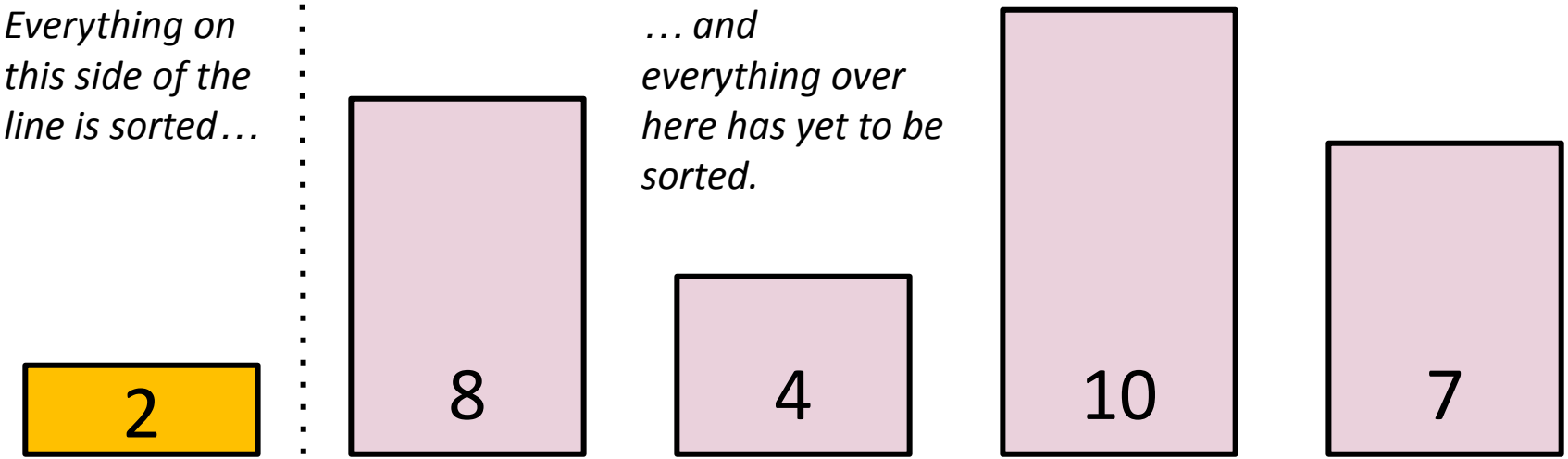
Selection Sort

- Idea: find the smallest element, put it in front of other elements



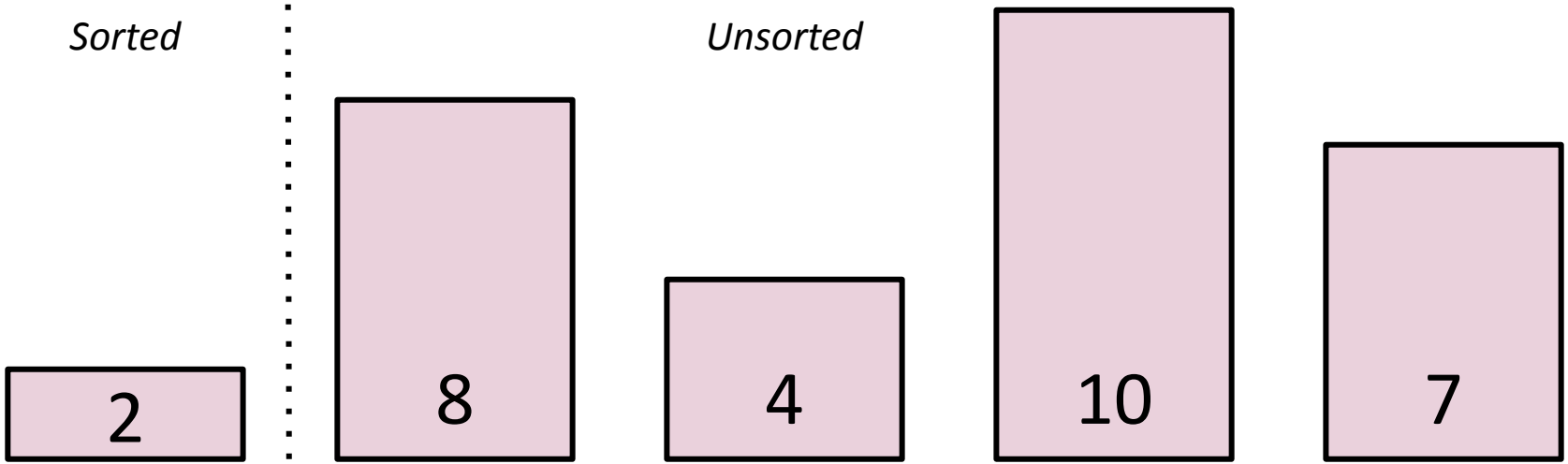
Selection Sort

- Idea: find the smallest element, put it in front of other elements



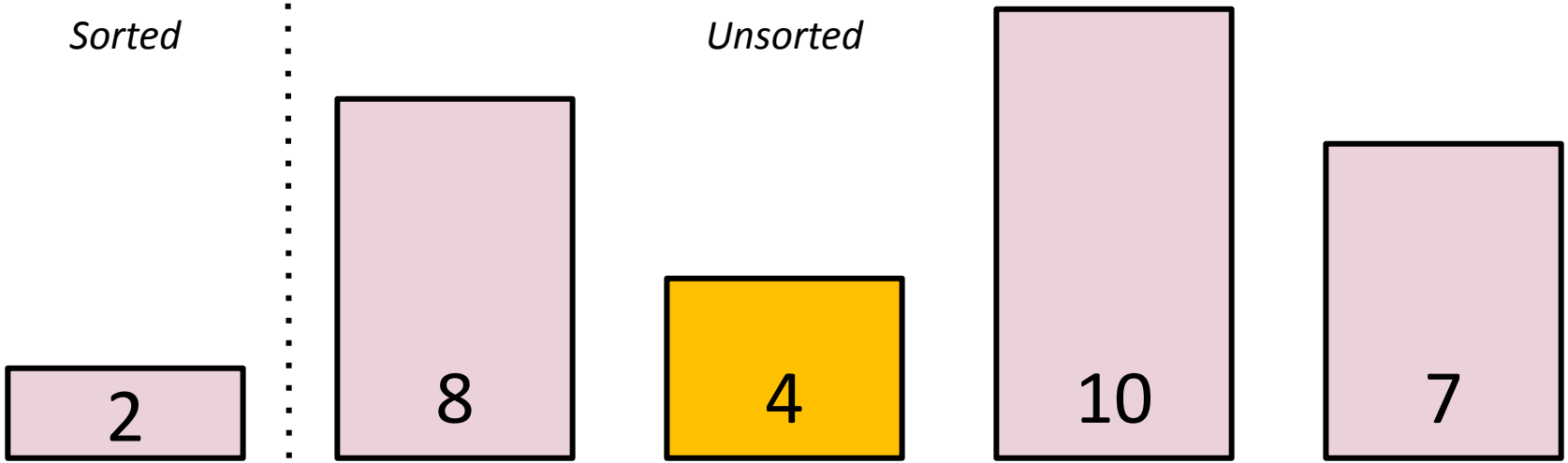
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



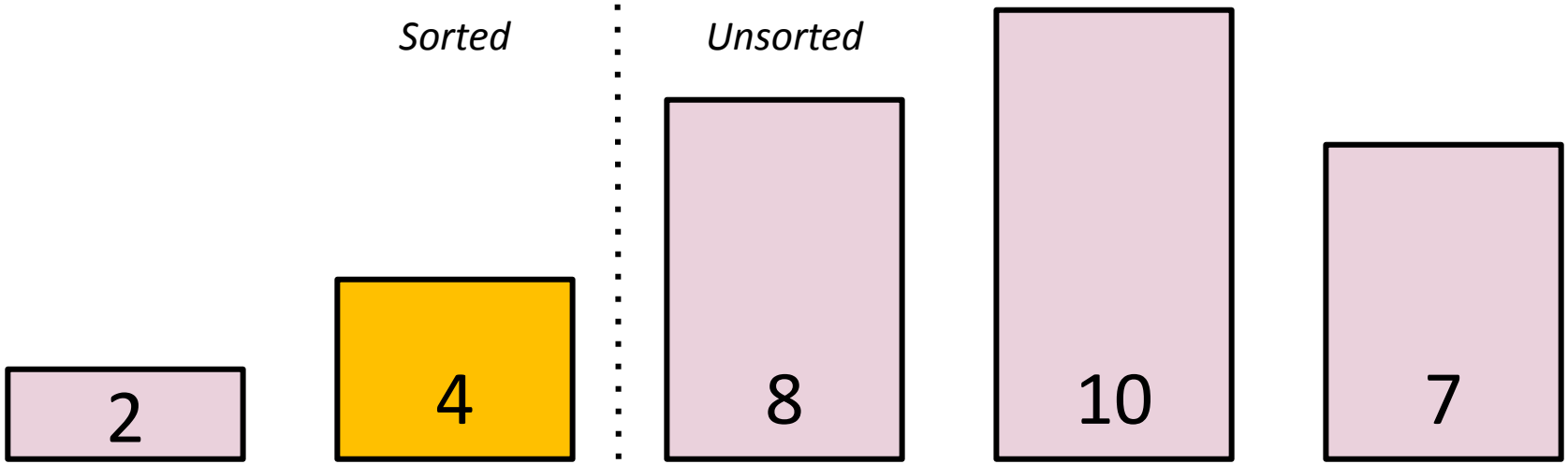
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



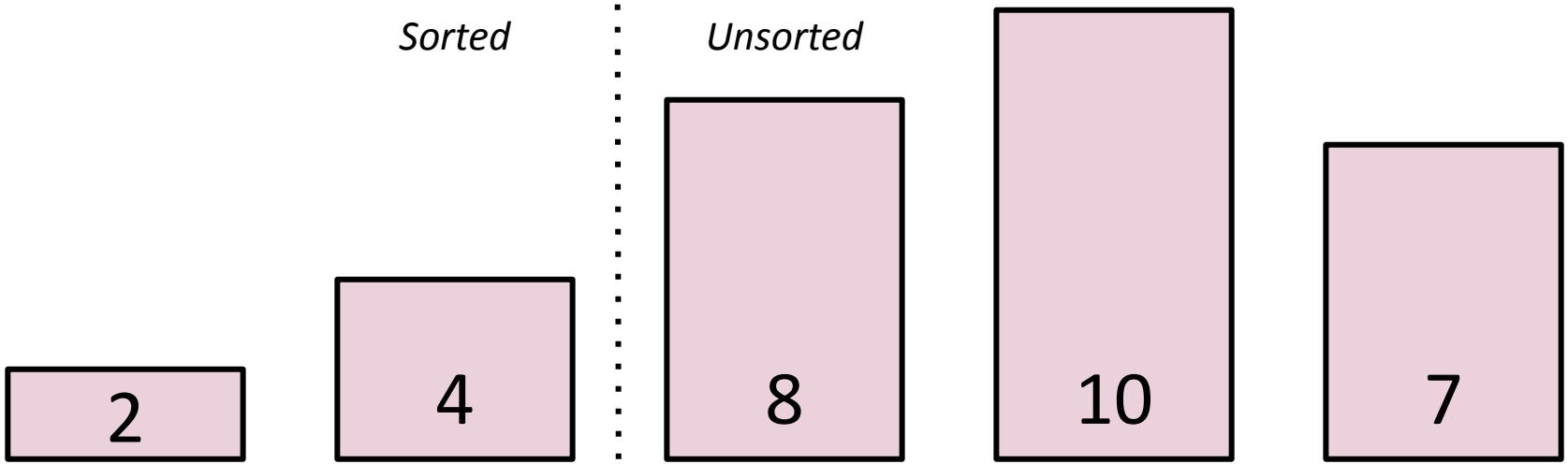
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



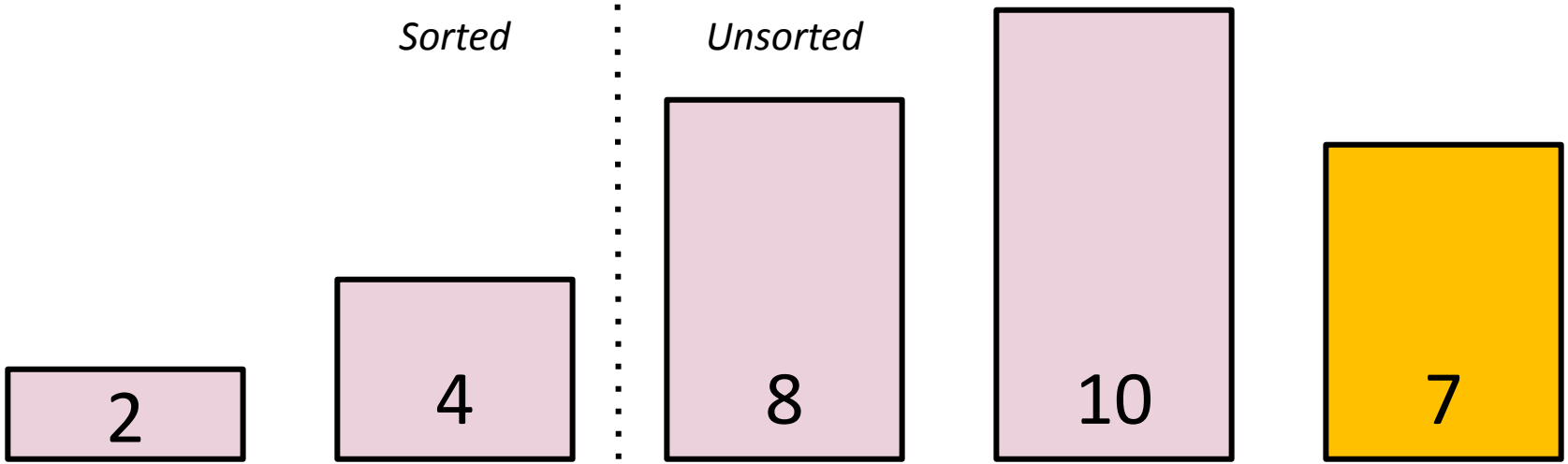
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



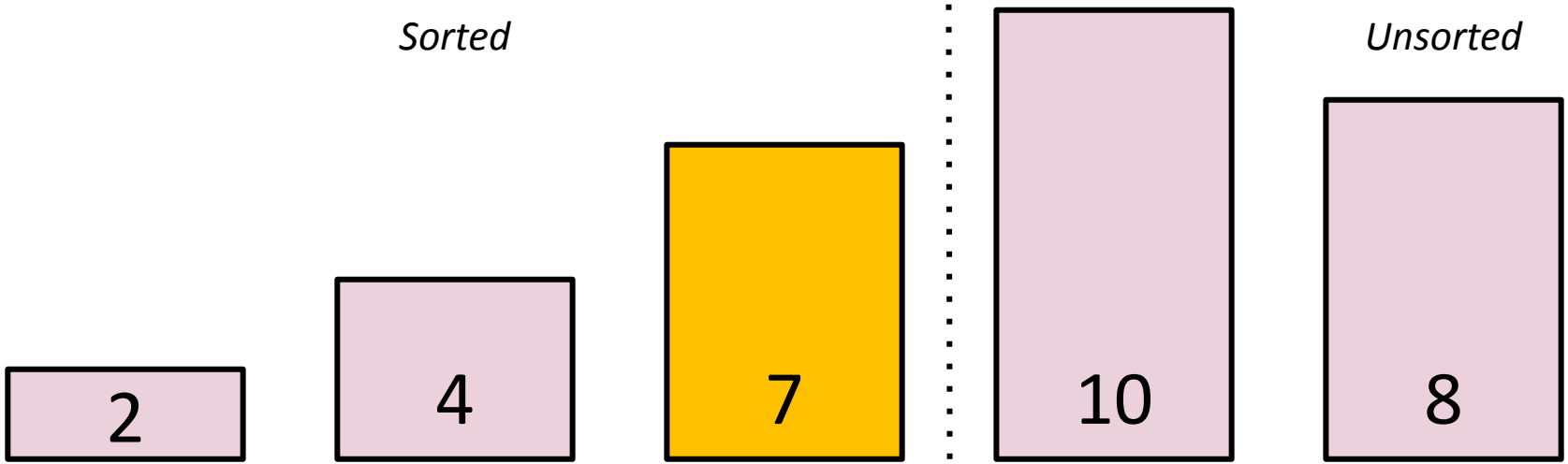
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



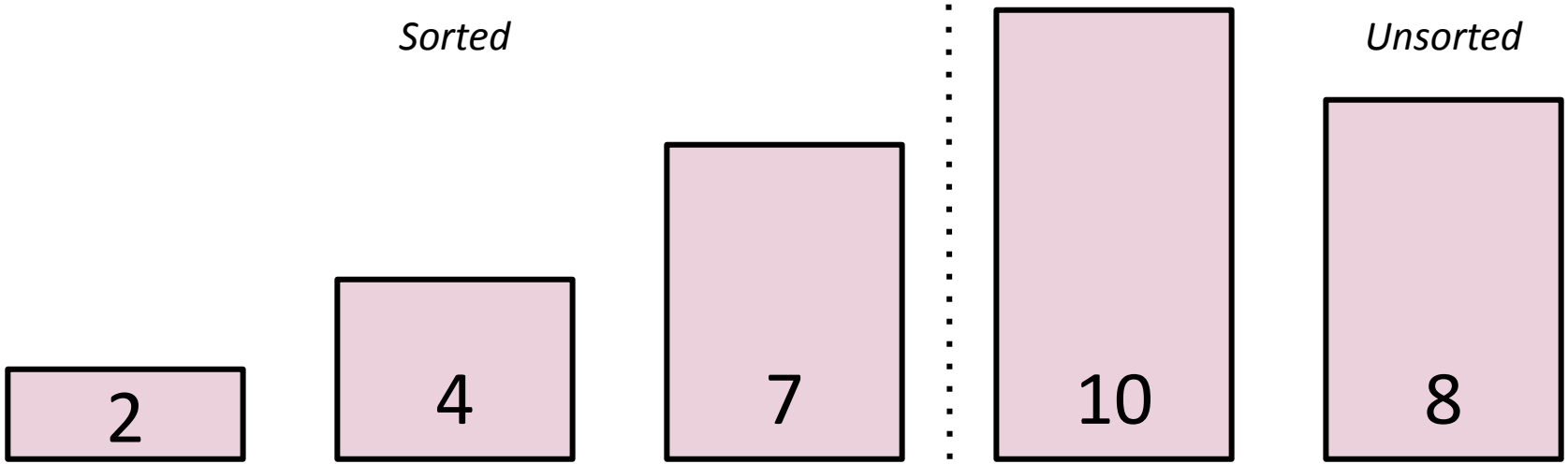
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



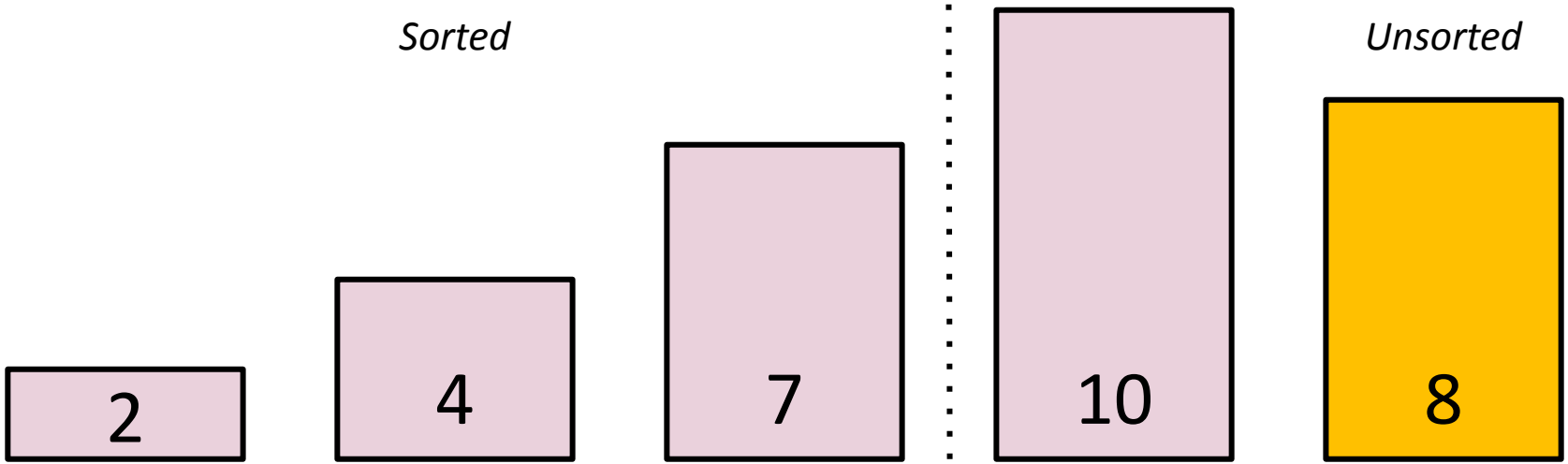
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



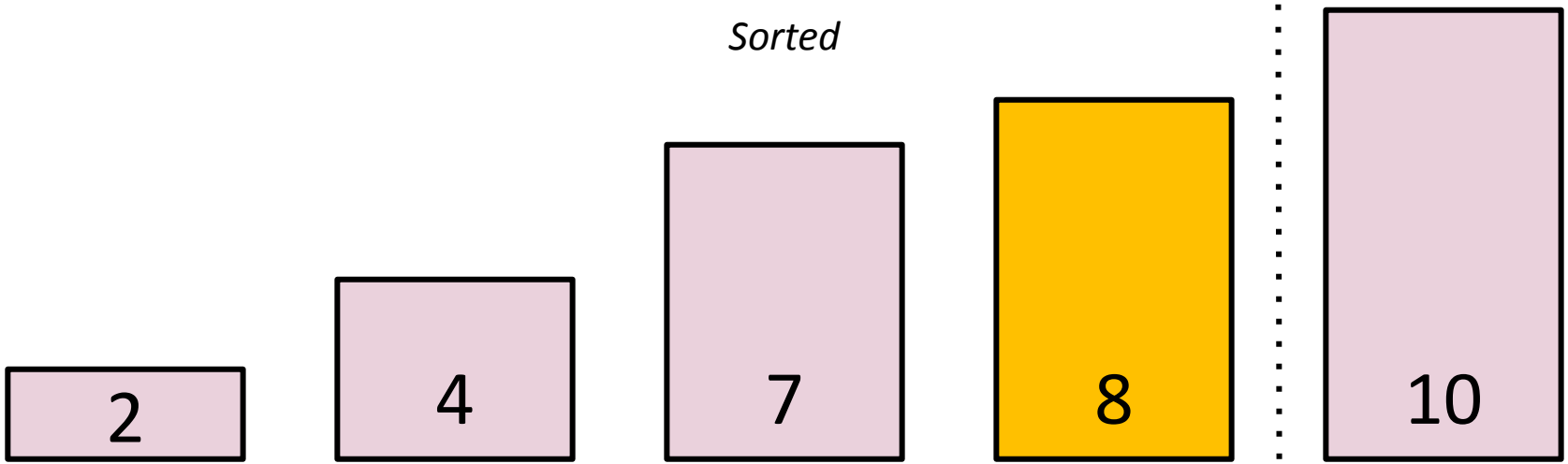
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



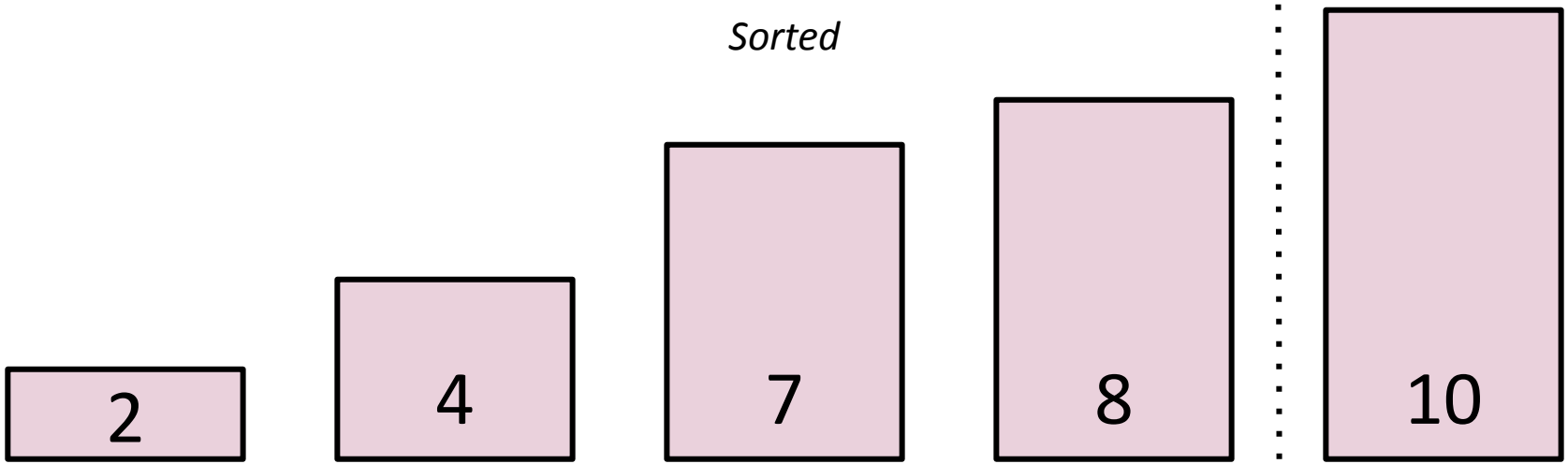
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



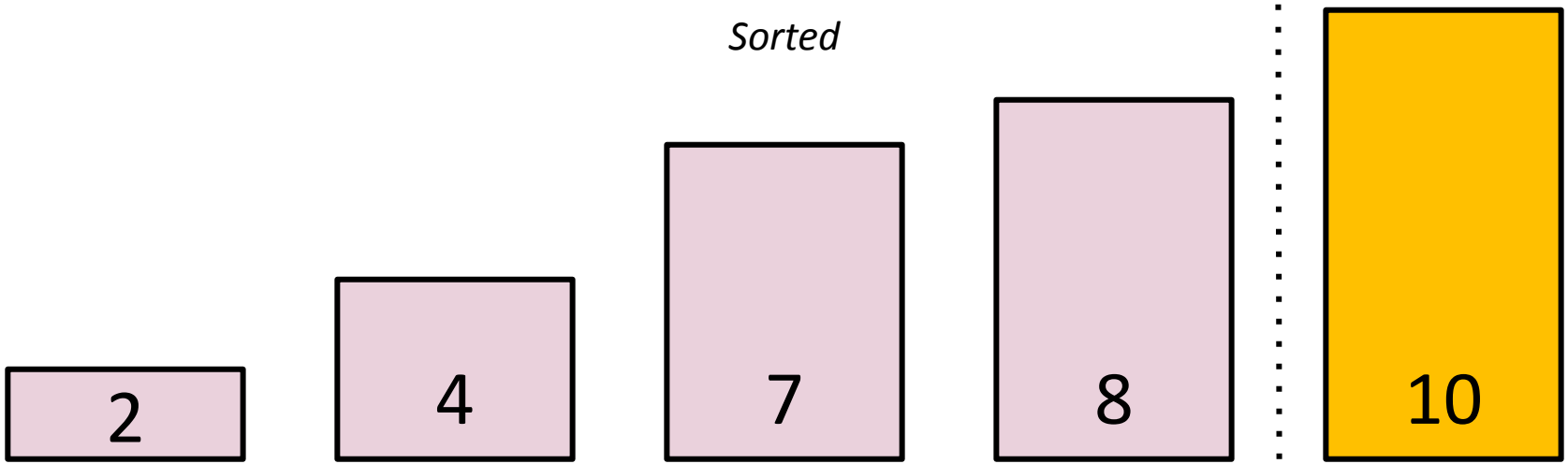
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



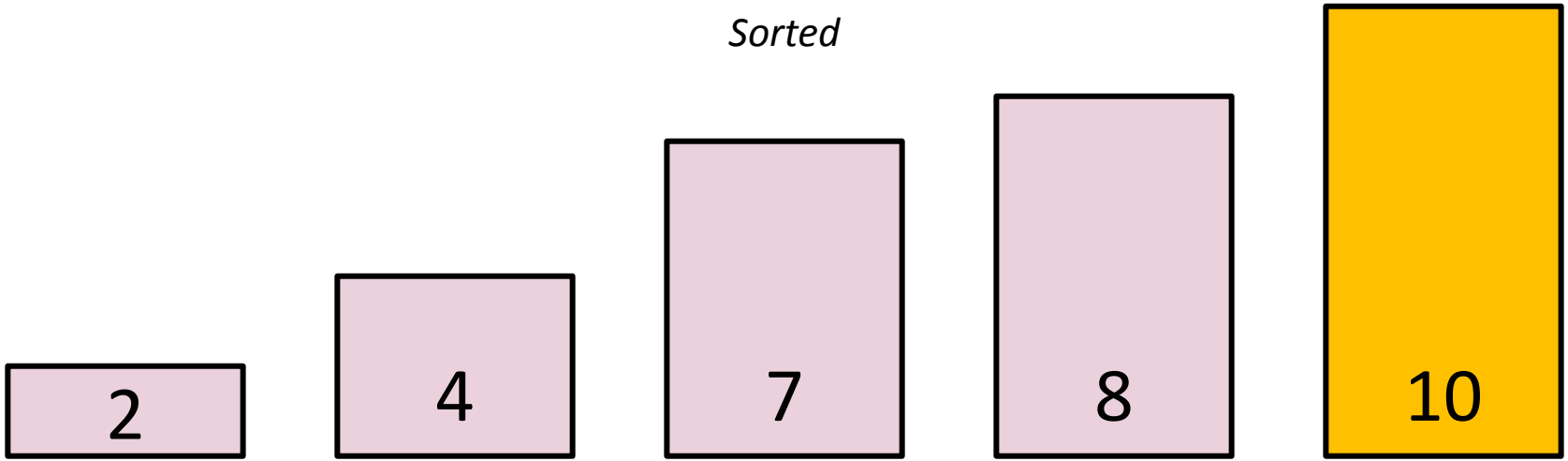
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



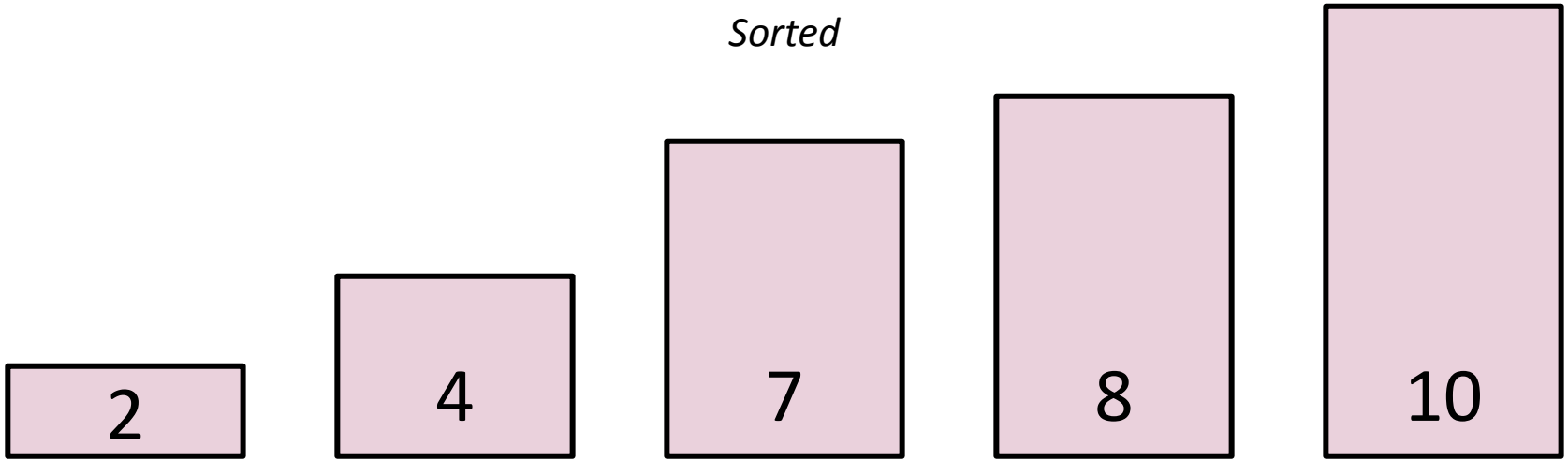
Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



Selection Sort

- Idea: find the smallest element, put it in front of other elements
- Repeat, putting the next smallest element in the next smallest spot



Demo: Selection Sort

Selection Sort Code

```
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = indexOfSmallest(elems, index);
        swap(elems, index, smallestIndex);
    }
}

int indexOfSmallest(const Vector<int>& elems, int startPoint) {
    int smallestIndex = startPoint;
    for (int i = startPoint + 1; i < elems.size(); i++) {
        if (elems[i] < elems[smallestIndex]) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
```

Selection Sort Runtime

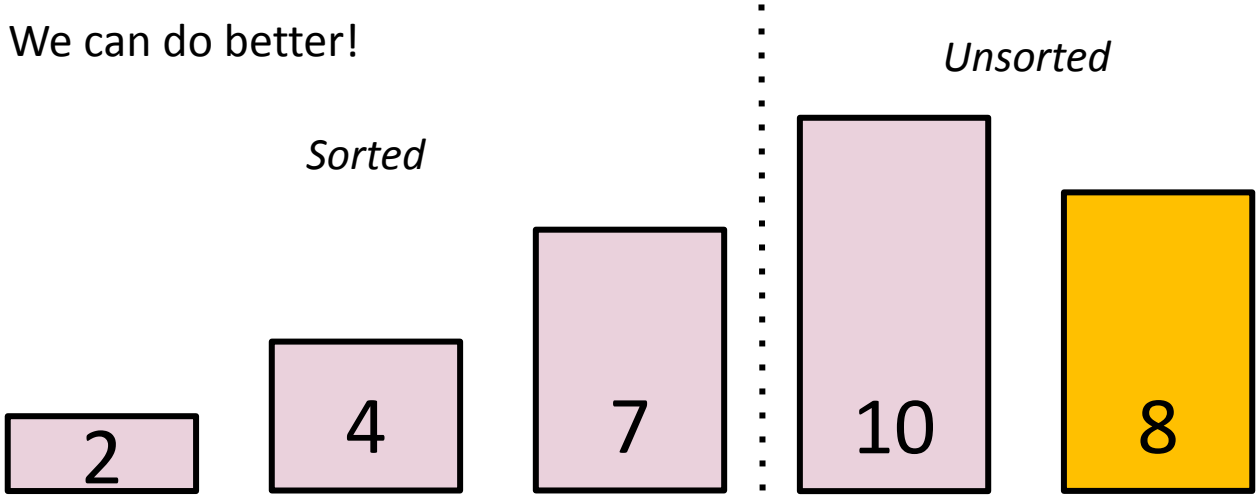
```
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = indexOfSmallest(elems, index);
        swap(elems, index, smallestIndex);
    }
}

int indexOfSmallest(const Vector<int>& elems, int startPoint) {
    int smallestIndex = startPoint;
    for (int i = startPoint + 1; i < elems.size(); i++) {
        if (elems[i] < elems[smallestIndex]) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
```

O(n) operation

O(n) operation

Selection Sort Recap

- Selection sort repeatedly takes the smallest of the remaining elements and places it in front of those remaining elements
- $O(n^2)$ sorting algorithm
  - We can do better!



Divide-and-Conquer Algorithms

Problem solving strategy to achieve better than $O(n^2)$ sorting

Why Divide-and-Conquer?

- Let's say selection sort on a vector with 400 elements takes x ms
- How long would selection sort take on a vector with 200 elements?

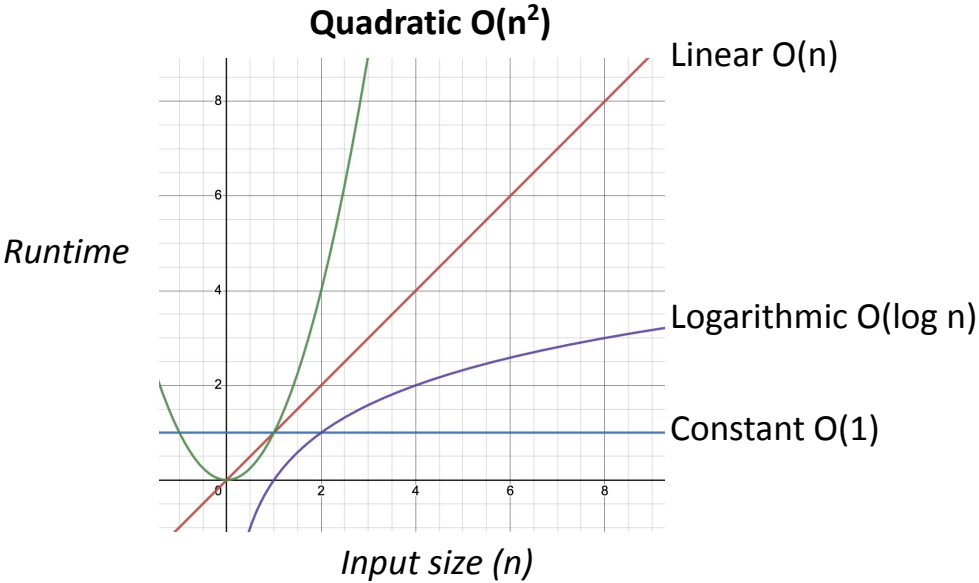
4	16	-2	2	54	13	47	6	19	2

4	16	-2	2	54

13	47	6	19	2

Why Divide-and-Conquer?

- Let's say selection sort on a vector with 400 elements takes x ms
- How long would selection sort take on a vector with 200 elements?

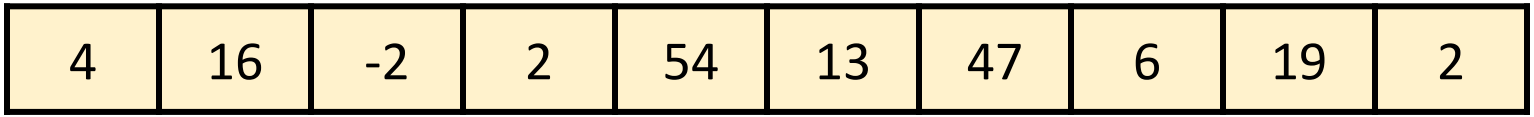


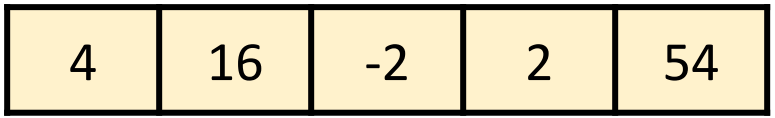
x/4 ms.

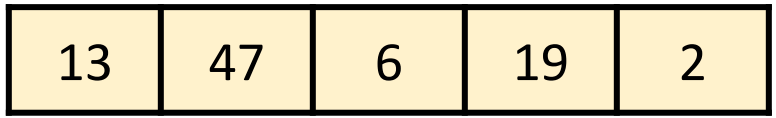
For a quadratic function, halving input size quarters the runtime.

Why Divide-and-Conquer?

- Let's say selection sort on a vector with 400 elements takes x ms
- Sorting two vectors with 200 elements each takes x/4 + x/4 = x/2 ms… sorting smaller arrays speeds us up!

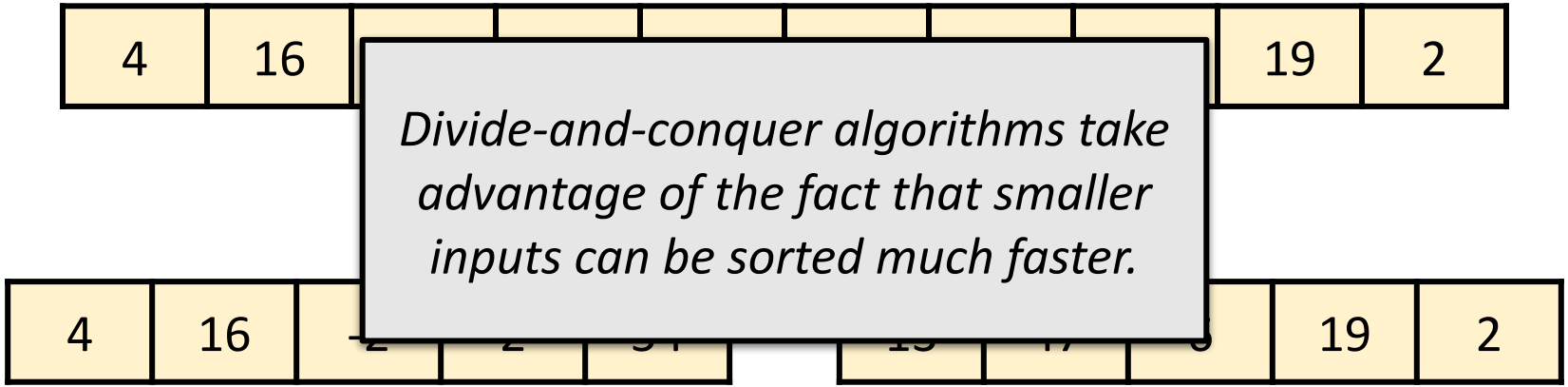






Why Divide-and-Conquer?

- Let's say selection sort on a vector with 400 elements takes x ms
- Sorting two vectors with 200 elements each takes x/4 + x/4 = x/2 ms… sorting smaller arrays speeds us up!



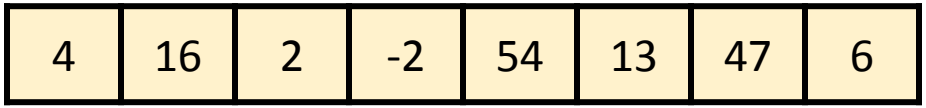
Merge Sort

You saw this on Assignment 3!

Recursive sorting algorithm:

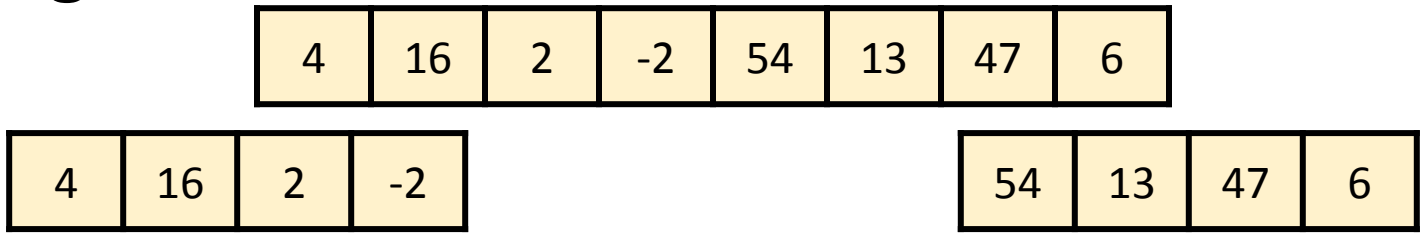
- Base case:
  - An empty or length-1 list is already sorted
- Recursive case:
  - Break each list in half and recursively sort (merge sort) each half
  - Merge them back into a single sorted list

Merge Sort



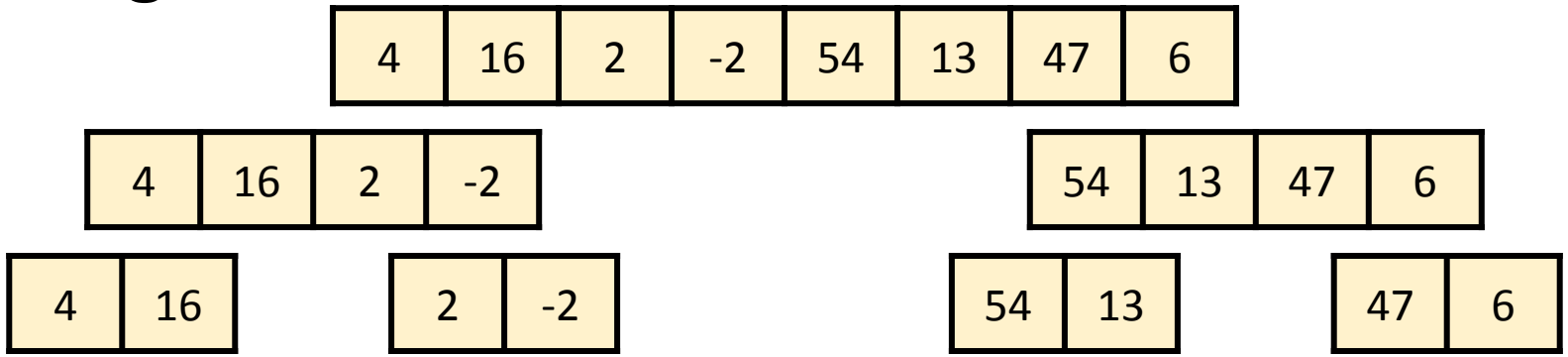
Split list in half

Merge Sort



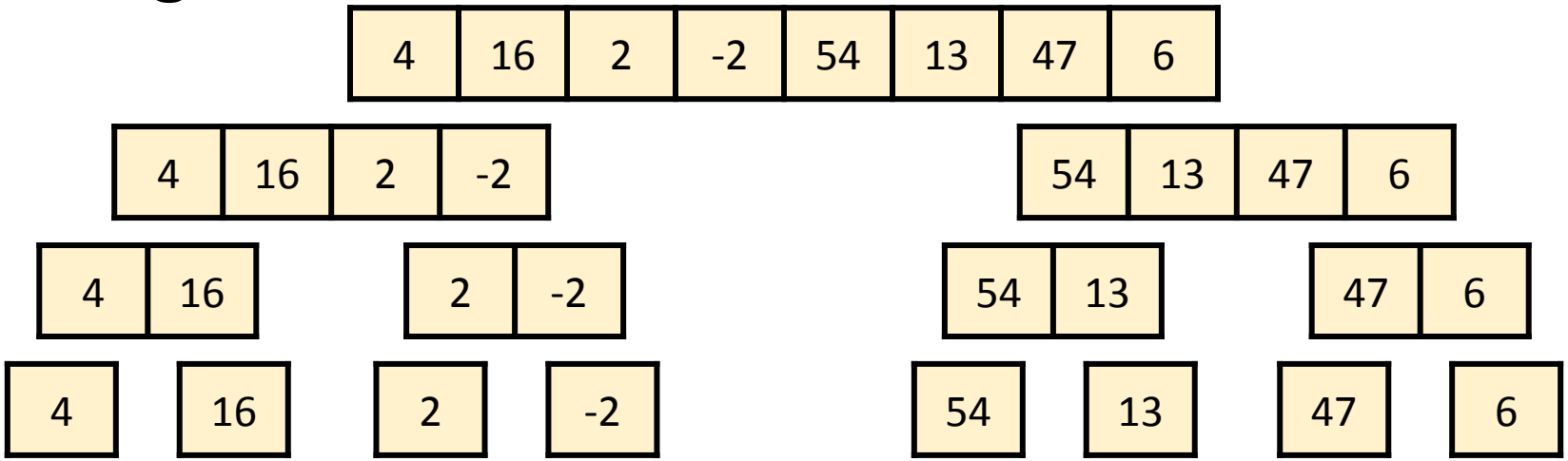
Split list in half

Merge Sort



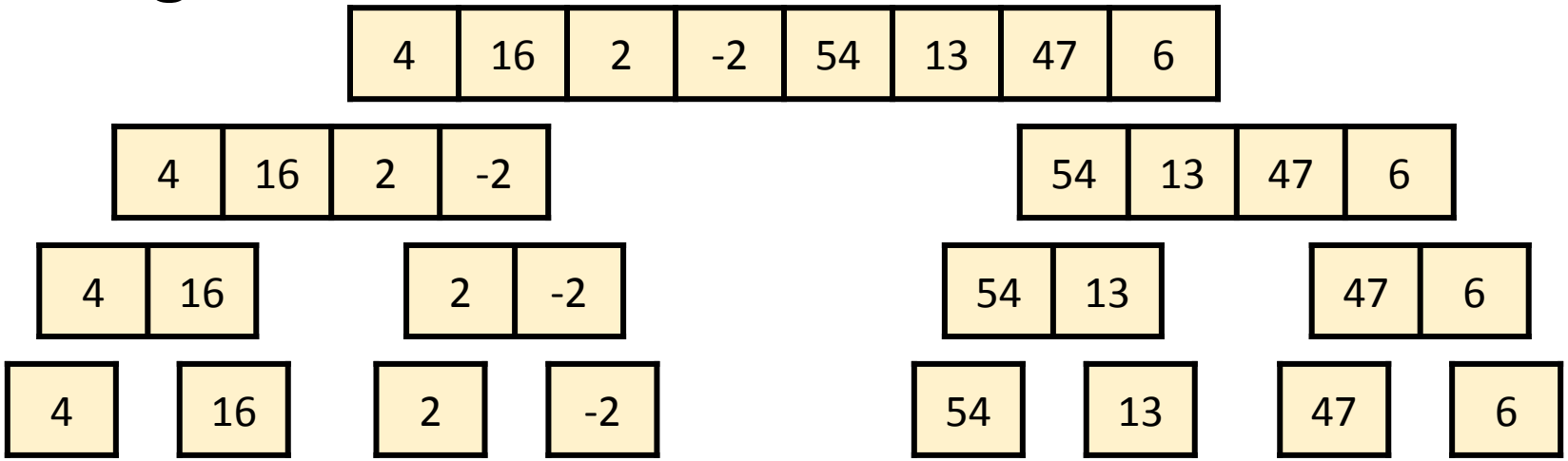
Split list in half

Merge Sort



Base case: size 0 and 1 lists are already sorted

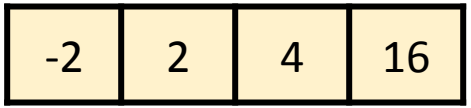
Merge Sort

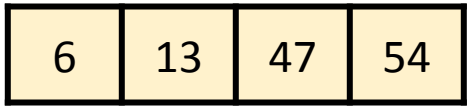


Repeatedly merge sorted lists

Merging Sorted Sequences

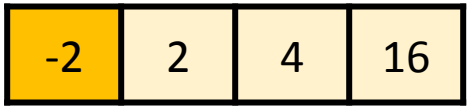
- Look at the first element of both sorted lists, take the smaller one and put it into the result list

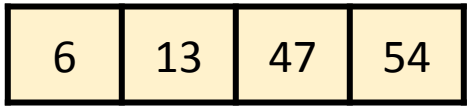




Merging Sorted Sequences

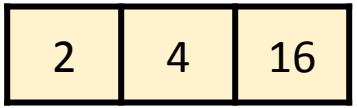
- Look at the first element of both sorted lists, take the smaller one and put it into the result list

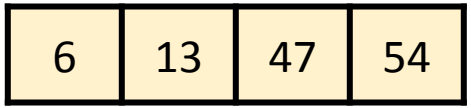


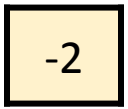


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

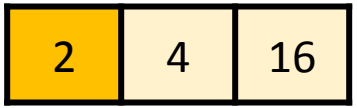


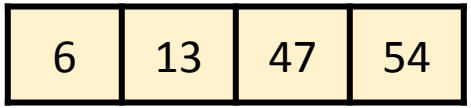


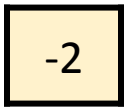


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

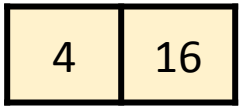


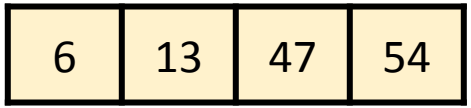


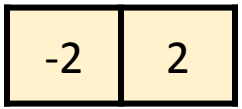


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

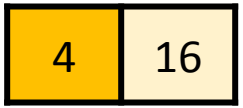


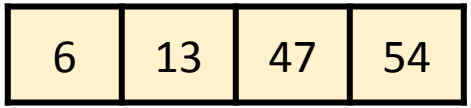


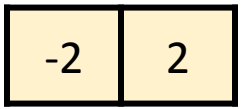


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

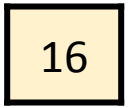


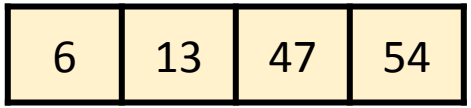


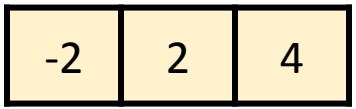


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

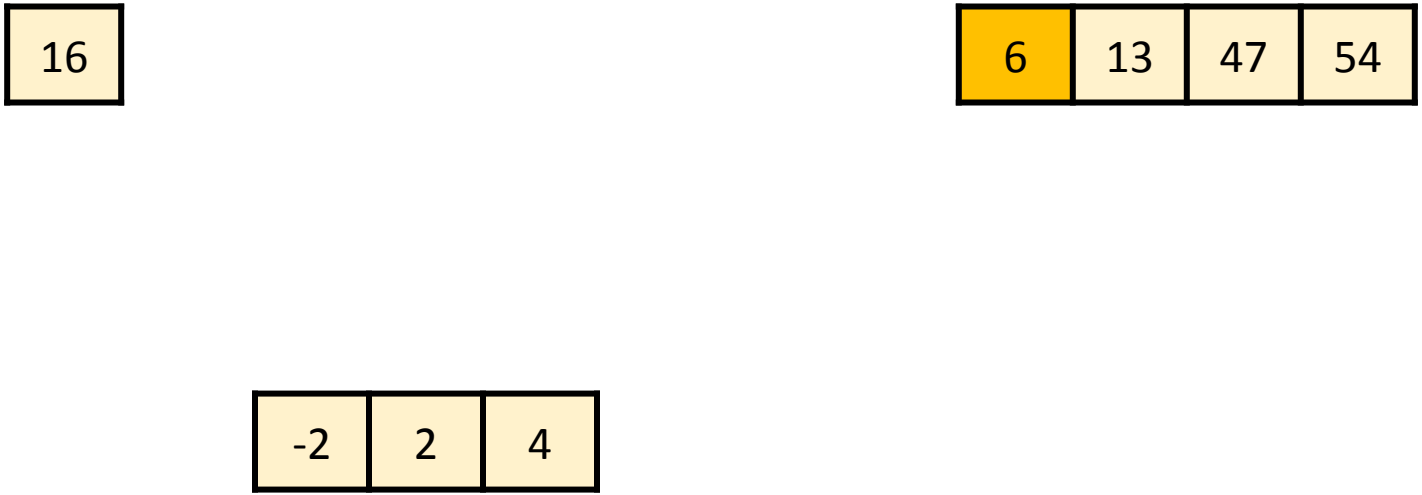






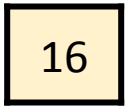
Merging Sorted Sequences

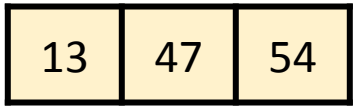
- Look at the first element of both sorted lists, take the smaller one and put it into the result list

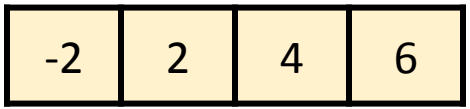


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

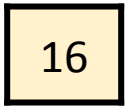


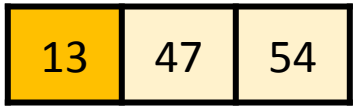


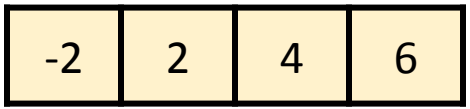


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

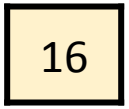


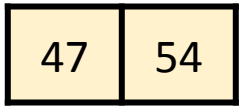


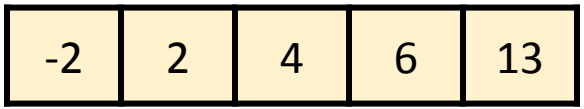


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list

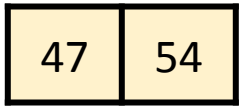


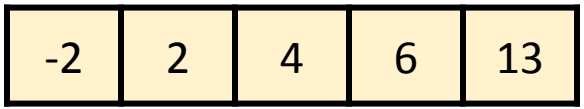




Merging Sorted Sequences

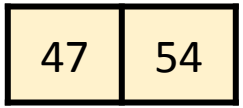
- Look at the first element of both sorted lists, take the smaller one and put it into the result list

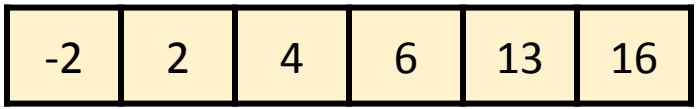




Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list



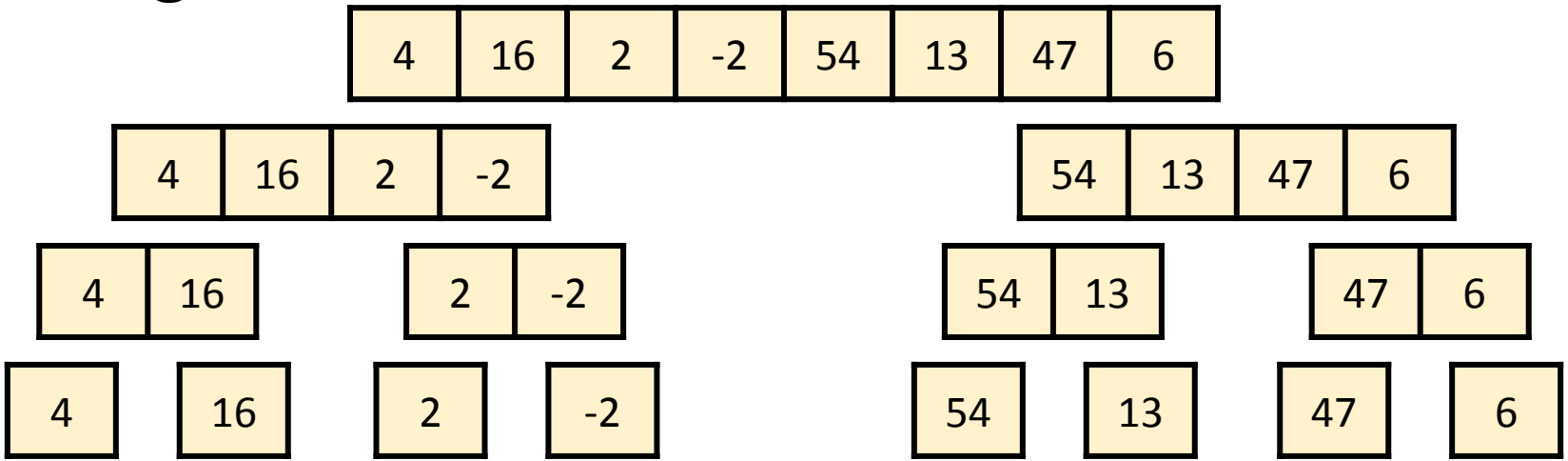


Merging Sorted Sequences

- Look at the first element of both sorted lists, take the smaller one and put it into the result list
- If one list becomes empty, add the other list to the end of result

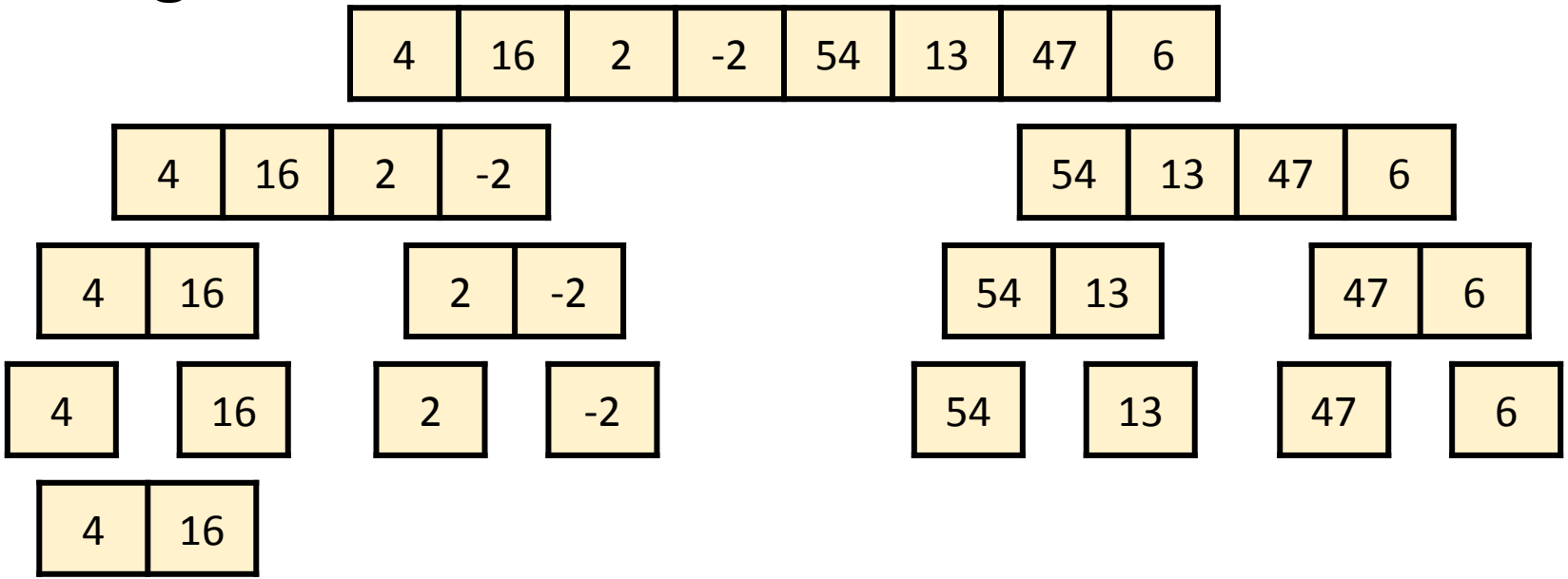
-2	2	4	6	13	16	47	54

Merge Sort



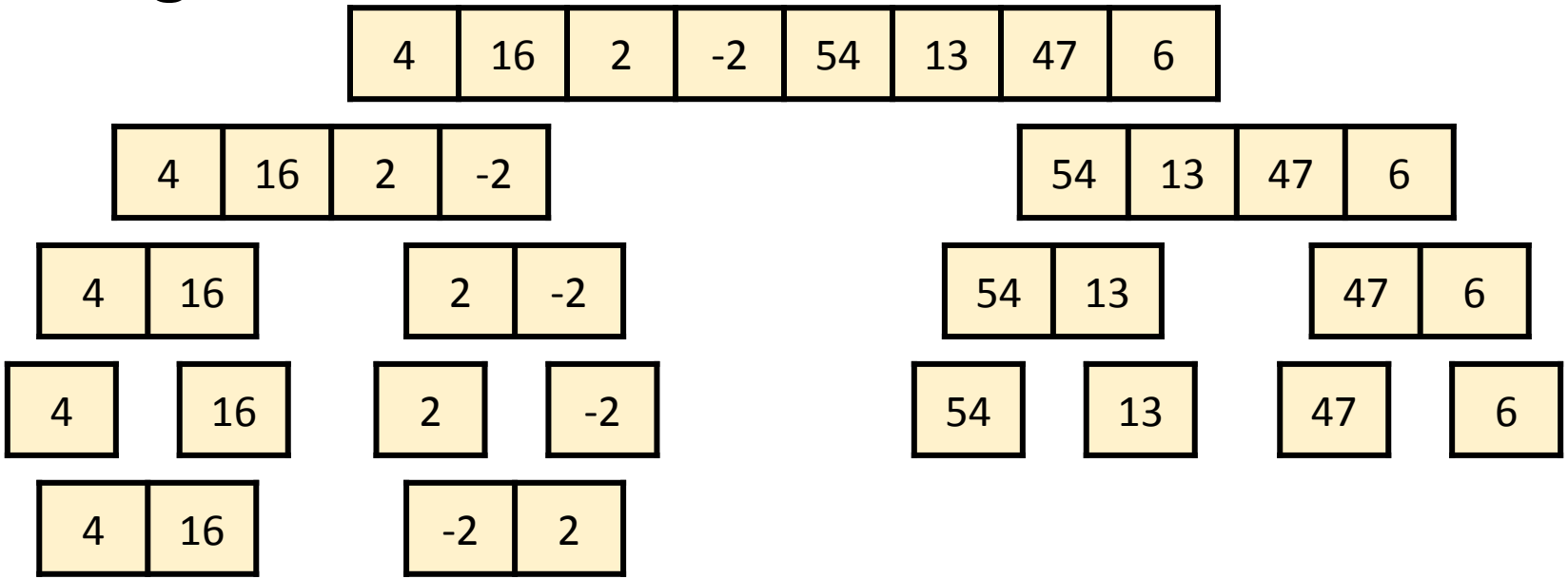
Let's merge some sorted lists!

Merge Sort



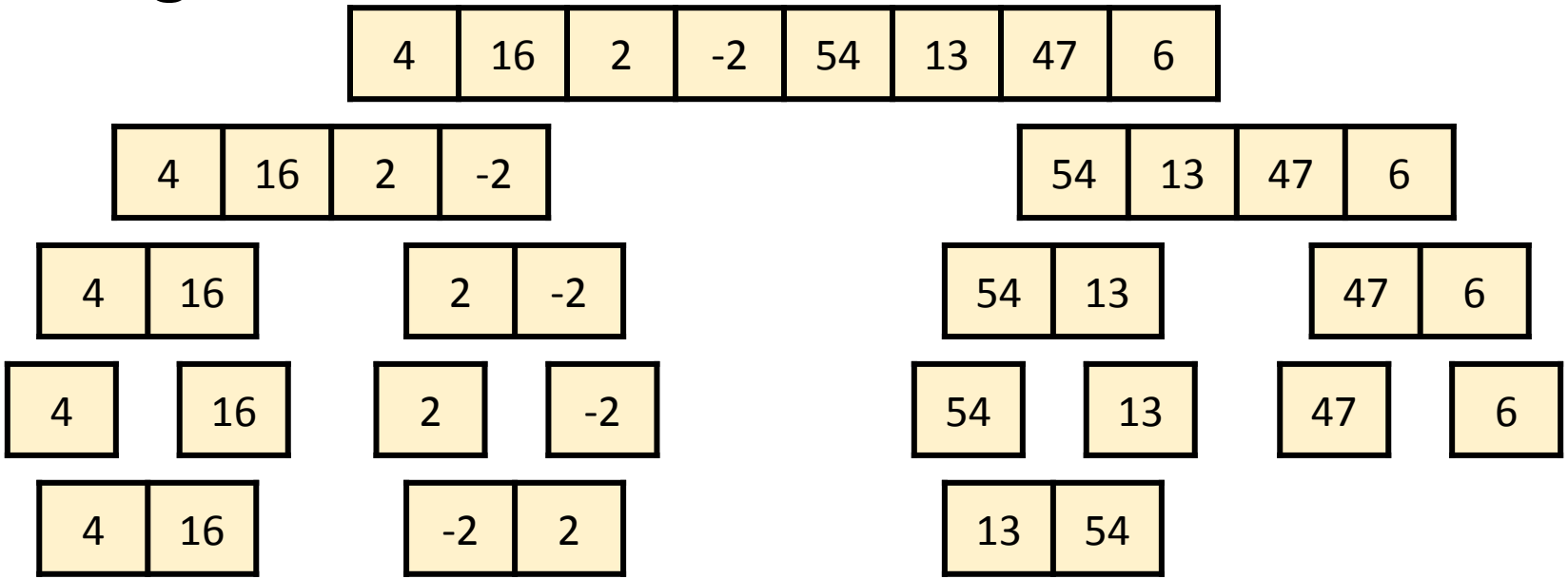
Let's merge some sorted lists!

Merge Sort



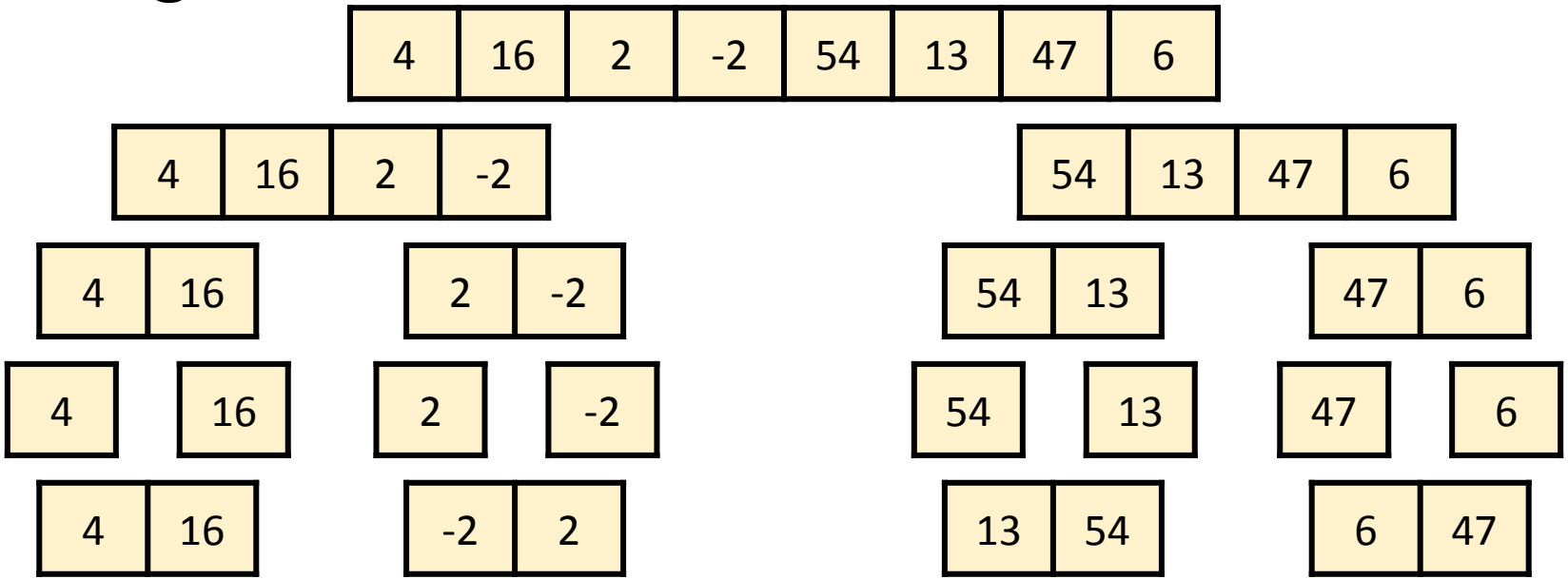
Let's merge some sorted lists!

Merge Sort

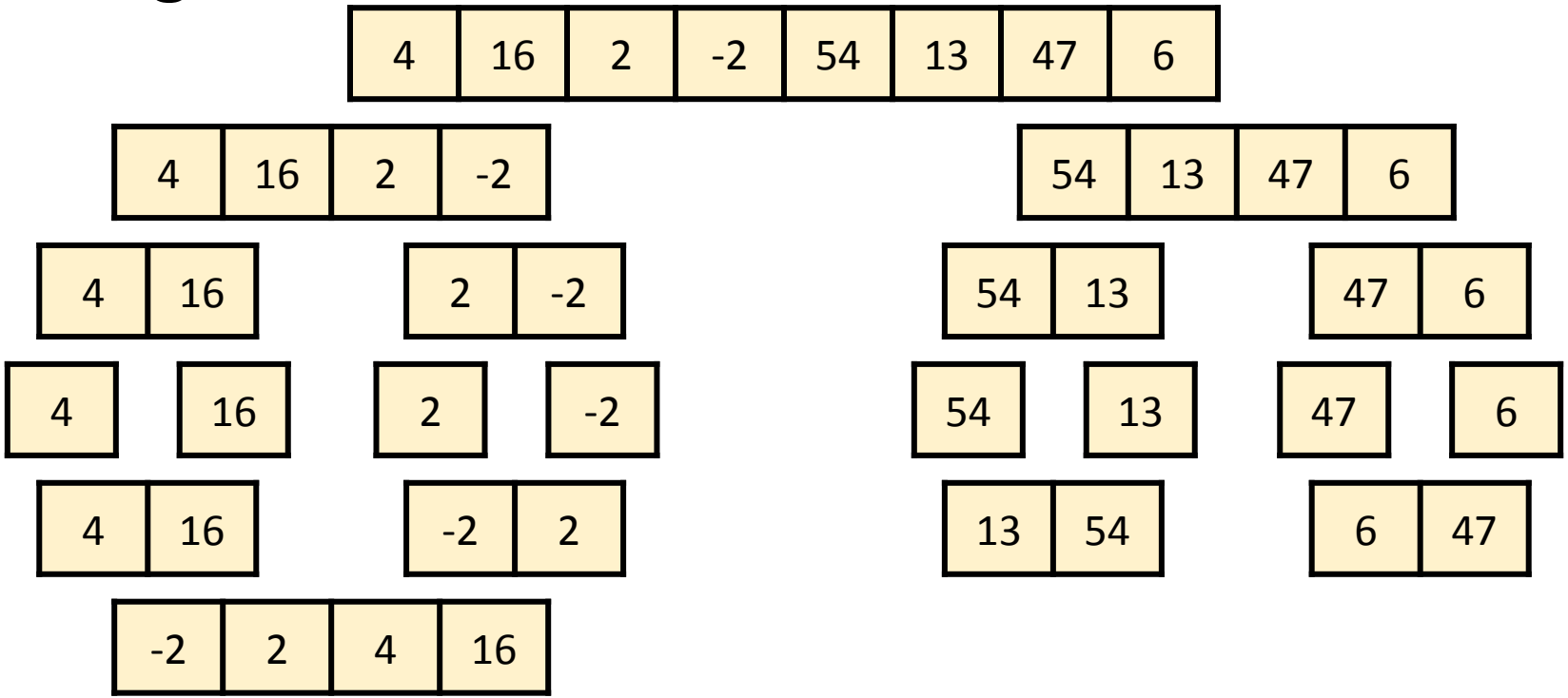


Let's merge some sorted lists!

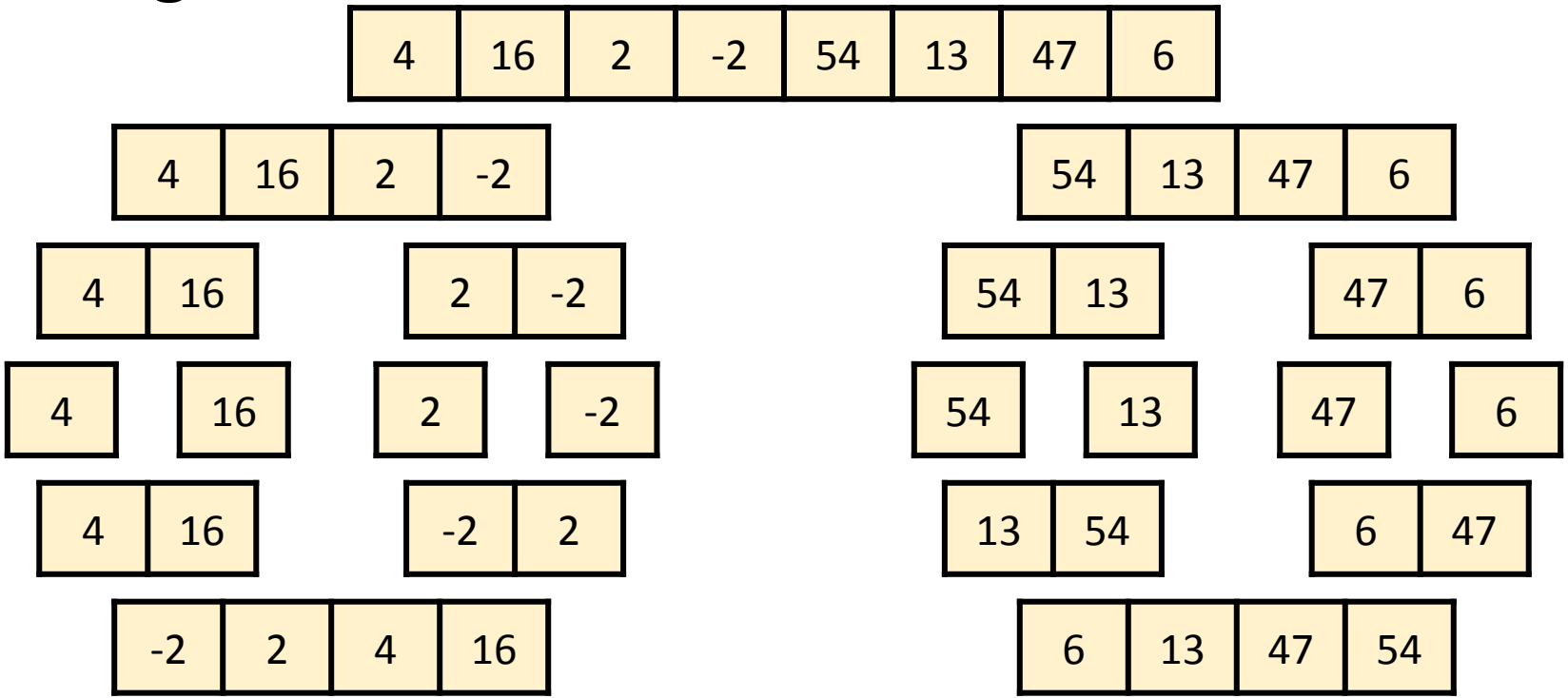
Merge Sort


4 16 2 -2 54 13 47 6
4 16 2 -2
54 13 47 6
4 16
2 -2
54 13
47 6
4
16
2
-2
54
13
47
6
4 16
-2 2
13 54
6 47

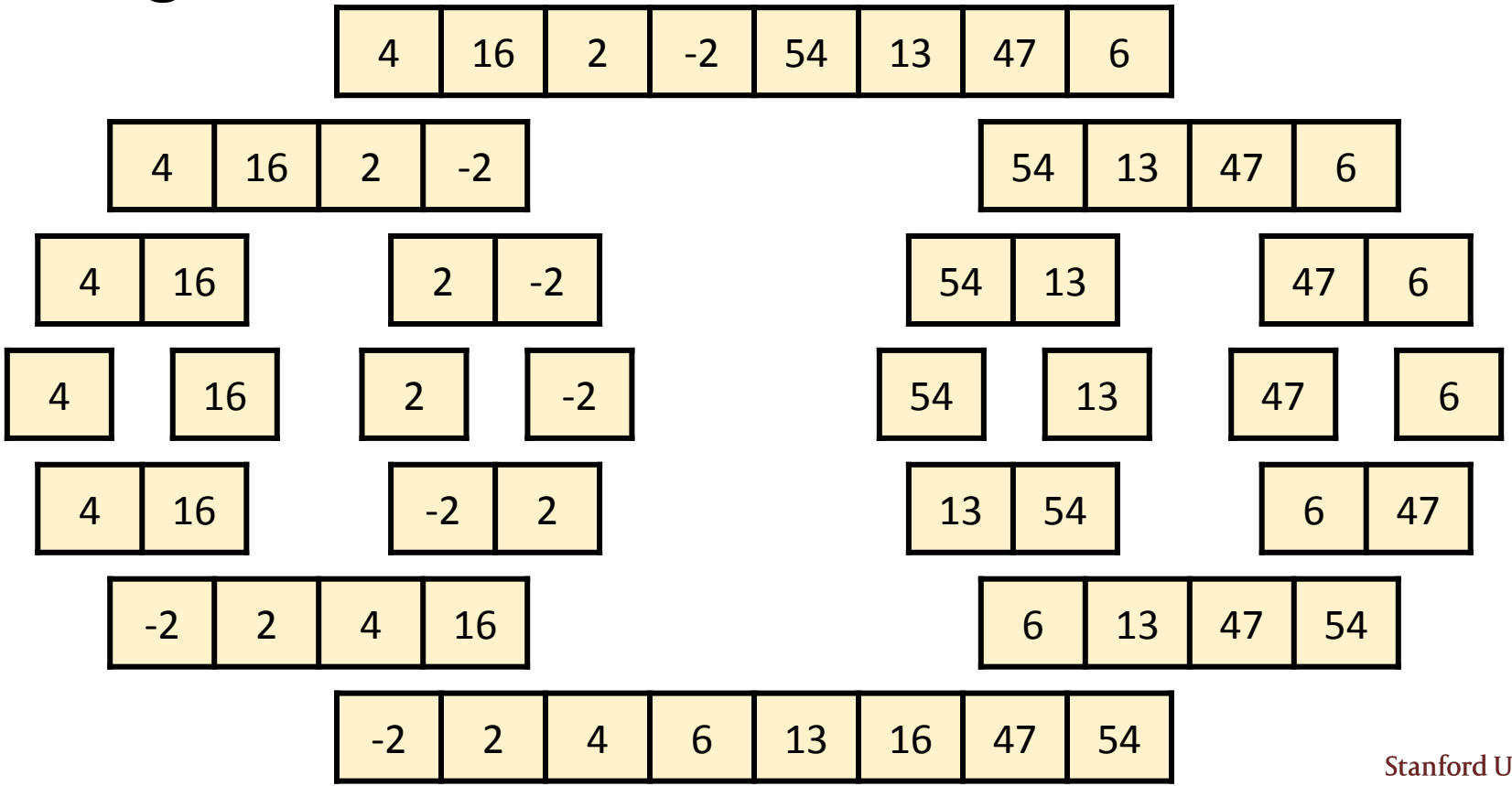
Merge Sort


4 16 2 -2 54 13 47 6
4 16 2 -2
54 13 47 6
4 16
2 -2
54 13
47 6
4
16
2
-2
54
13
47
6
4 16
-2 2
13 54
6 47
-2 2 4 16

Merge Sort


4 16 2 -2 54 13 47 6
4 16 2 -2
54 13 47 6
4 16
2 -2
54 13
47 6
4
16
2
-2
54
13
47
6
4 16
-2 2
13 54
6 47
-2 2 4 16
6 13 47 54

Merge Sort


4 16 2 -2 54 13 47 6
4 16 2 -2
54 13 47 6
4 16
2 -2
54 13
47 6
4
16
2
-2
54
13
47
6
4 16
-2 2
13 54
6 47
-2 2 4 16
6 13 47 54
-2 2 4 6 13 16 47 54

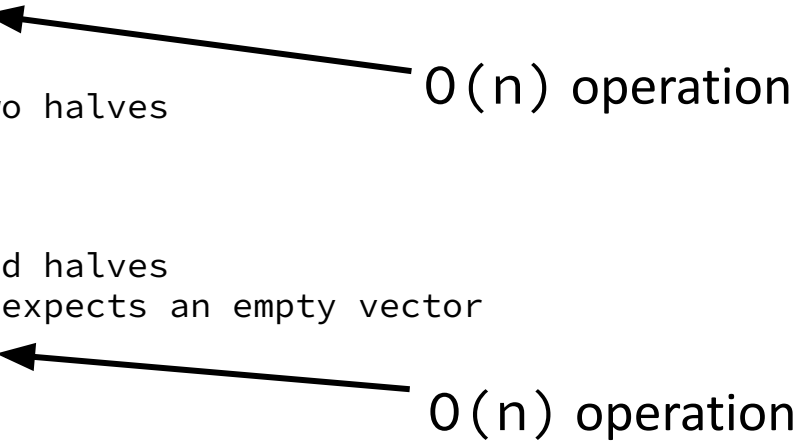
Demo: Merge Sort

Merge Sort Code

```
void mergeSort(Vector<int>& vec) {
   // Base case: vector is size 0 or 1, return
   if (vec.size() <= 1) return;

   // Split the list into two, equally sized halves
   Vector<int> left, right;
   split(vec, left, right);

   // Recursively sort the two halves
   mergeSort(left);
   mergeSort(right);

   // Fill vec with two sorted halves
   vec.clear(); // our merge expects an empty vector
   merge(vec, left, right);
}
```

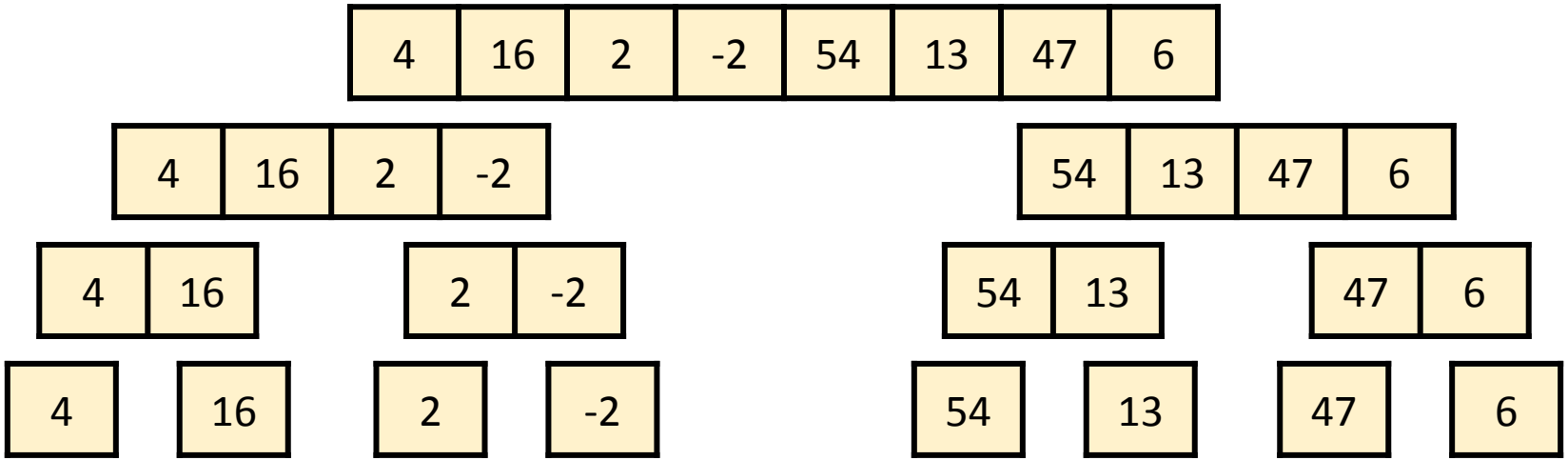
Merge Sort Code

```
void mergeSort(Vector<int>& vec) {
   // Base case: vector is size 0 or 1, return
   if (vec.size() <= 1) return;

   // Split the list into two, equally sized halves
   Vector<int> left, right;
   split(vec, left, right);

   // Recursively sort the two halves
   mergeSort(left);
   mergeSort(right);

   // Fill vec with two sorted halves
   vec.clear(); // our merge expects an empty vector
   merge(vec, left, right);
}
```



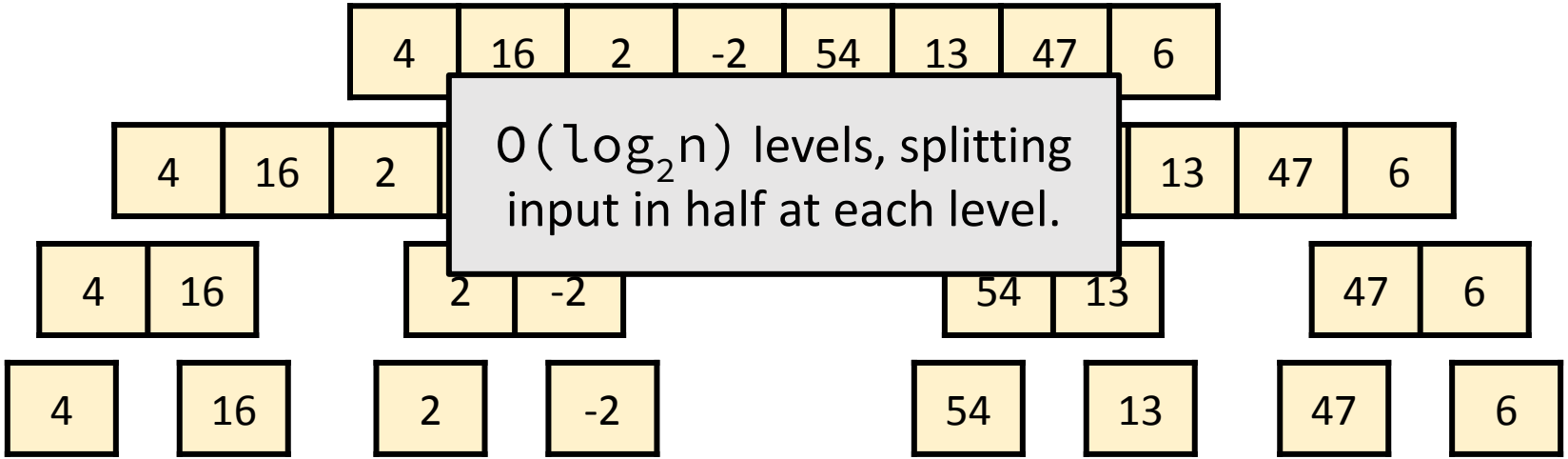
Merge Sort Runtime

- At each level, we do O(n) work
- How many levels are there?



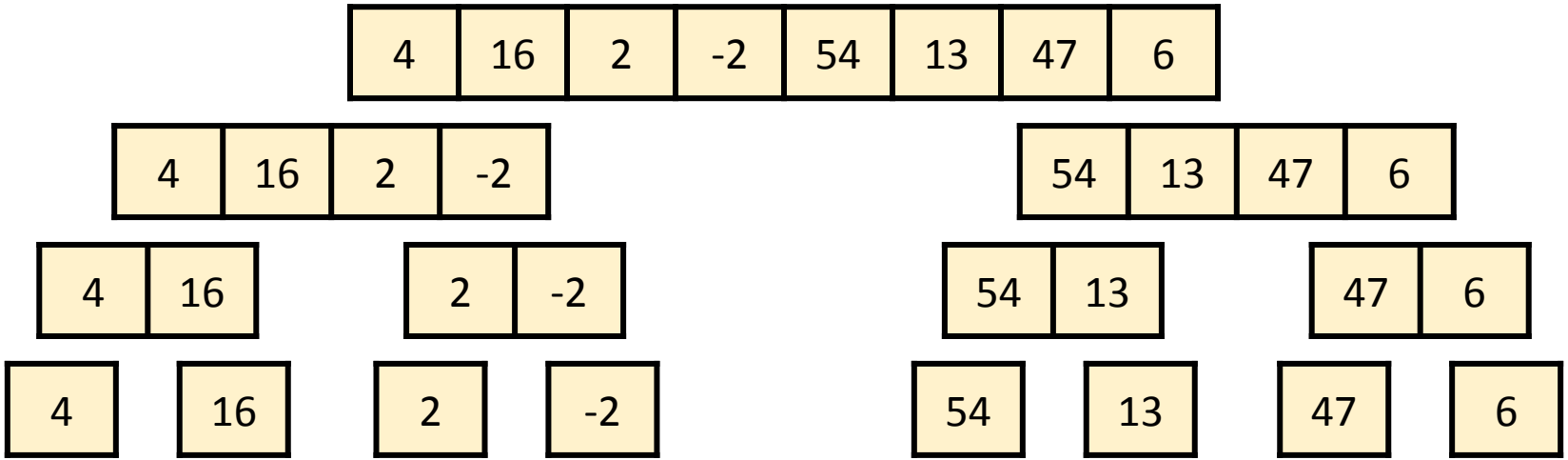
Merge Sort Runtime

- At each level, we do $O(n)$ work
- How many levels are there?



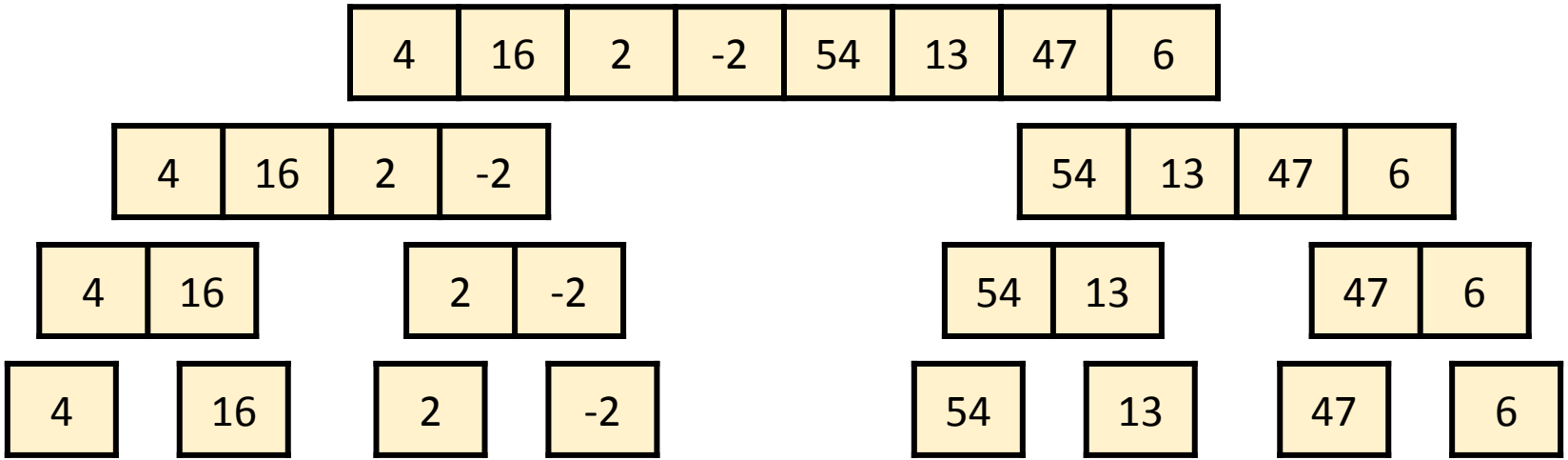
Merge Sort Runtime

- At each level, we do $O(n)$ work, and we have $O(\log n)$ levels



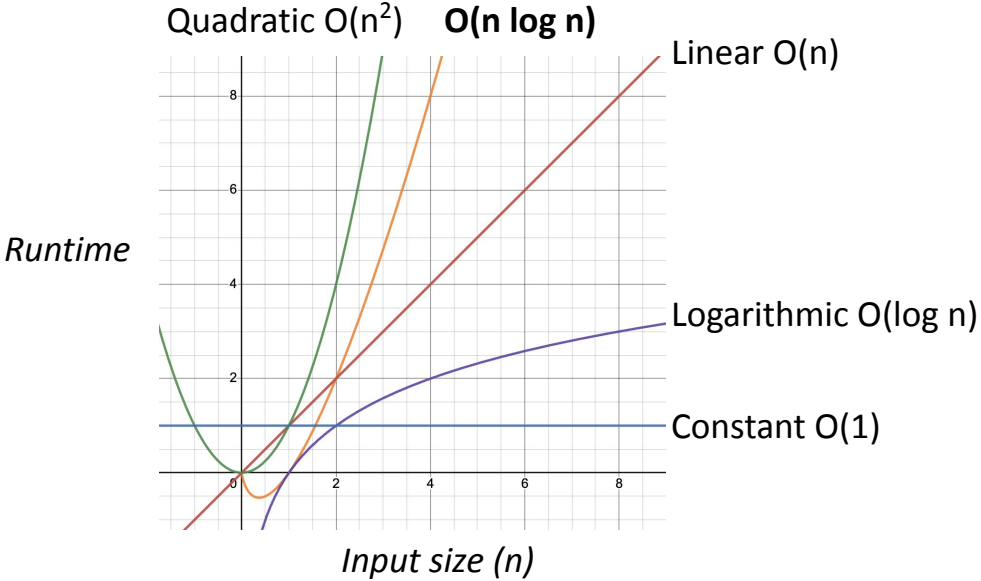
Merge Sort Runtime

- At each level, we do $O(n)$ work, and we have $O(\log n)$ levels
- Merge sort runtime is **$O(n \log n)$**, which is better than $O(n^2)$



Merge Sort Runtime

- At each level, we do $O(n)$ work, and we have $O(\log n)$ levels
- Merge sort runtime is **$O(n \log n)$**, which is better than $O(n^2)$



Merge Sort Runtime

👥 *mergeSort is a recursive function, but these O(n) helper functions were iterative. Why?*

```
void mergeSort(Vector<int>& vec) {
   // Base case: vector is size 0 or 1,
   if (vec.size() <= 1) return;

   // Split the list into two, equally sized halves
   Vector<int> left, right;
   split(vec, left, right);

   // Recursively sort the two halves
   mergeSort(left);
   mergeSort(right);

   // Fill vec with two sorted halves
   vec = {}; // our merge expects an empty vector
   merge(vec, left, right);
}
```

O(n) operation

O(n) operation

Merge Sort Runtime

Think about the stack frames! We don't want to do O(n) operations recursively, but we can make O(log n) recursive calls.

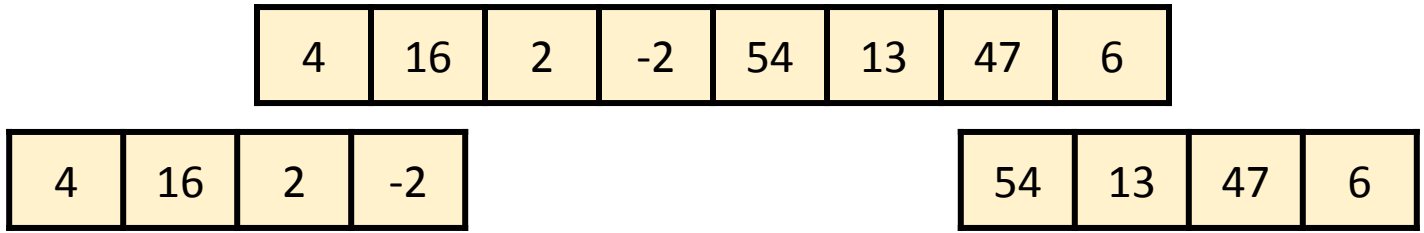
```
void mergeSort(Vector<int>& vec) {
   // Base case: vector is size 0 or 1,
   if (vec.size() <= 1) return;

   // Split the list into two, equally sized halves
   Vector<int> left, right;
   split(vec, left, right);

   // Recursively sort the two halves
   mergeSort(left);
   mergeSort(right);

   // Fill vec with two sorted halves
   vec = {}; // our merge expects an empty vector
   merge(vec, left, right);
}
```

O(n) operation

O(n) operation

Merge Sort Recap

- Recursively sort left and right half of input, then merge result back into one sorted sequence
- Divide step: easy (just split in half and recurse)
- Conquer step: hard (merge sorted sequences)
- `O(n log n)` sorting algorithm
  - This is better than Selection Sort!

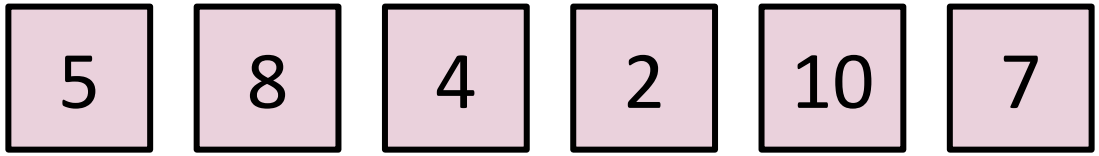


Quick Sort

You will see this on Assignment 5!

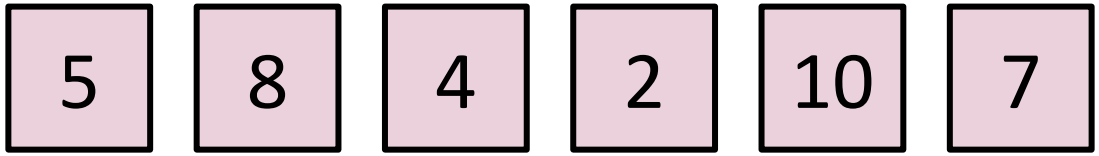
1. Choose a “pivot” element
2. Group your elements into three groups:
   a. Less than pivot
   b. Equal to pivot
   c. Greater than pivot
3. Recursively sort (quick sort) the less than and greater than groups
4. Concatenate the three sorted groups back together again

Quick Sort



Quick Sort

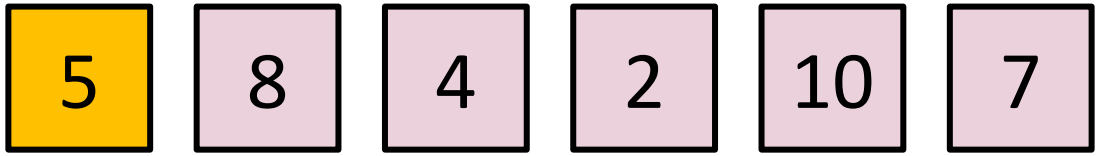
1. Choose a “pivot” element



Quick Sort

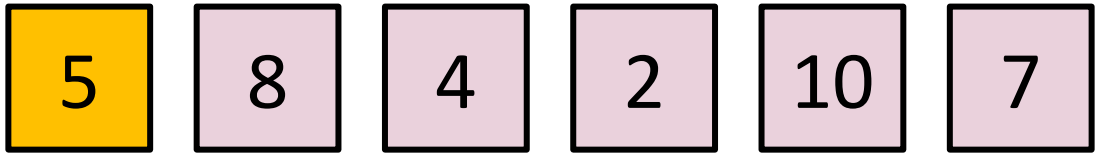
1. Choose a “pivot” element

We’ll just choose the first element



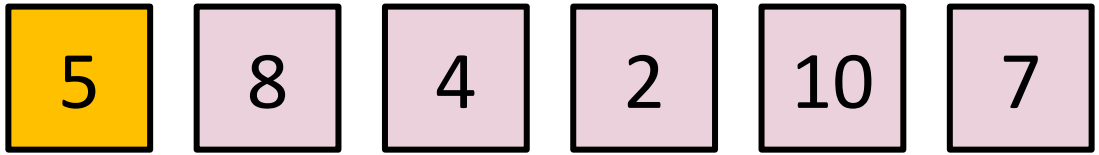
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



Quick Sort

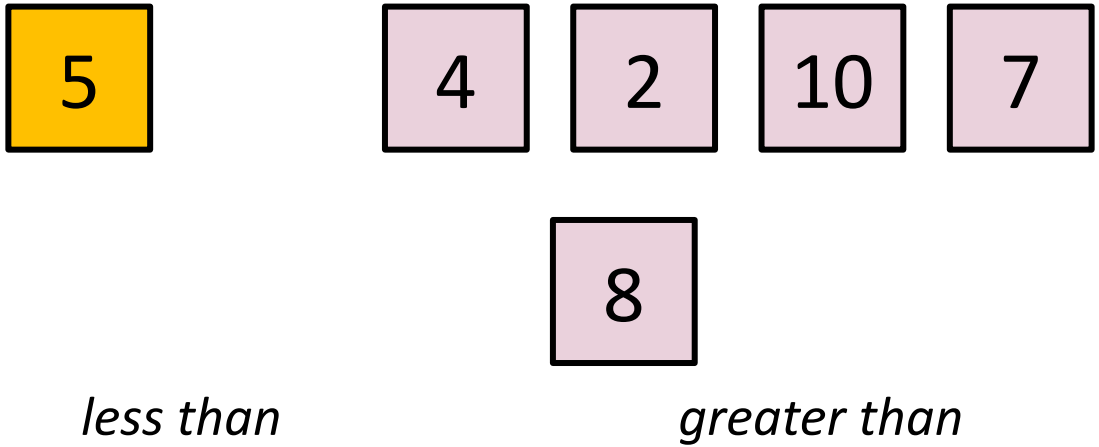
2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



less than *greater than*

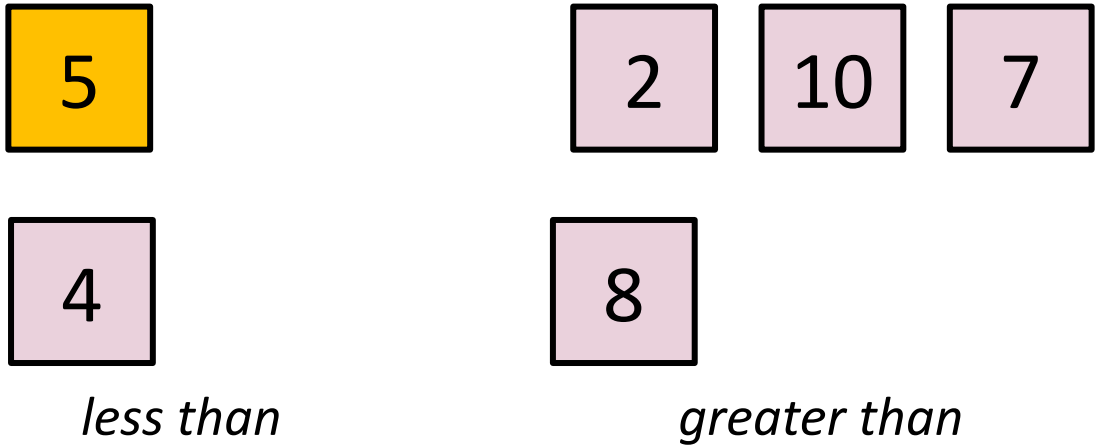
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



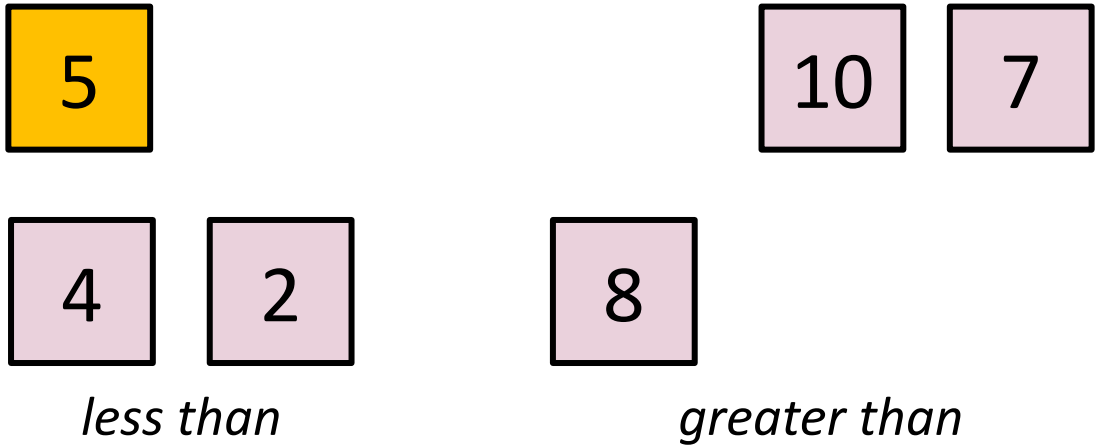
Quick Sort

2. Group your elements into three groups:
   a. Less than pivot
   b. Equal to pivot
   c. Greater than pivot



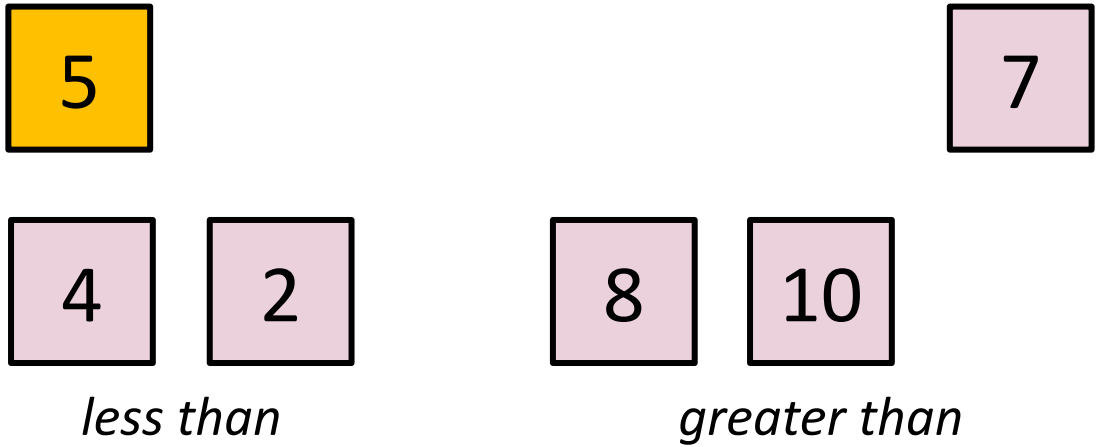
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



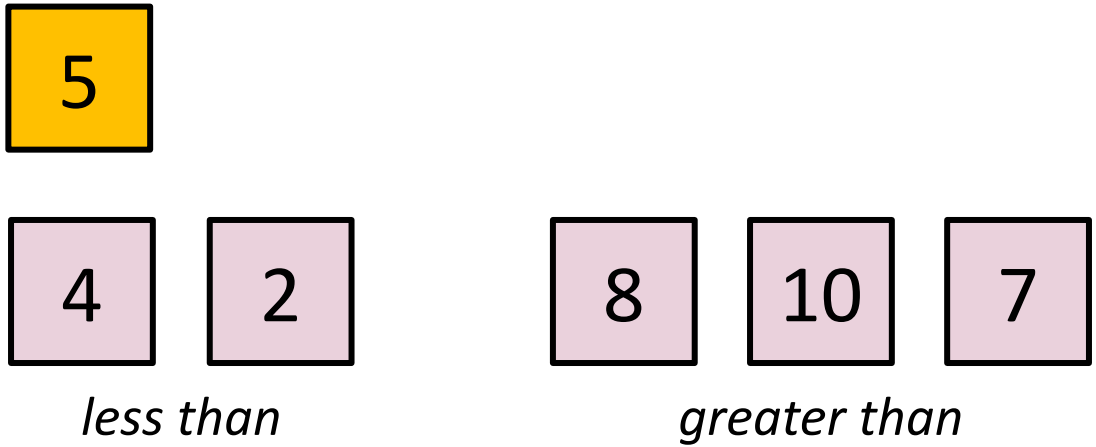
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



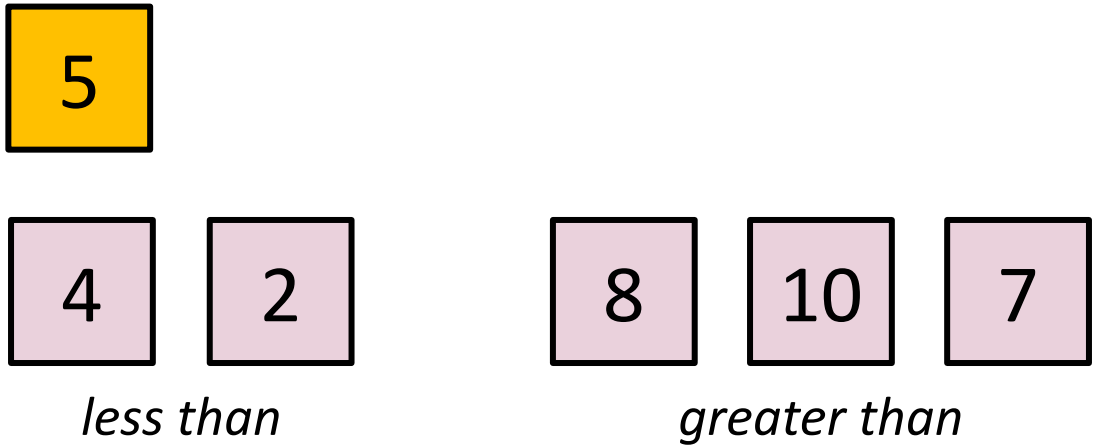
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



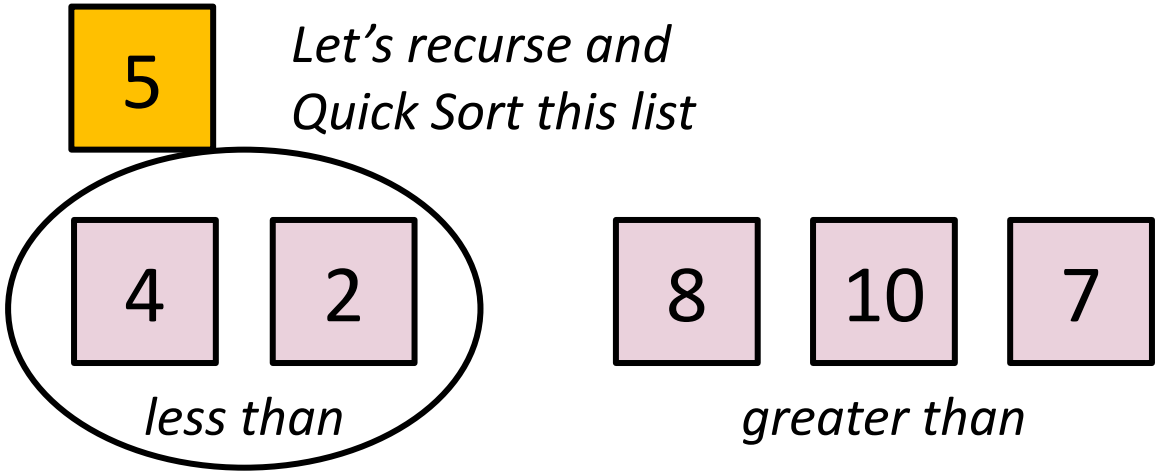
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



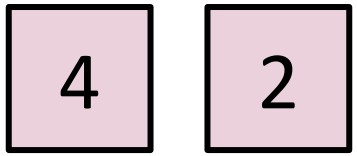
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



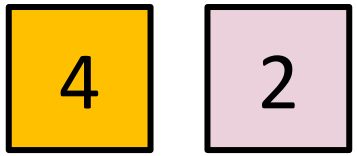
Quick Sort

1. Choose a “pivot” element



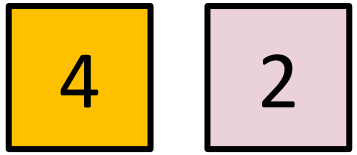
Quick Sort

1. Choose a “pivot” element



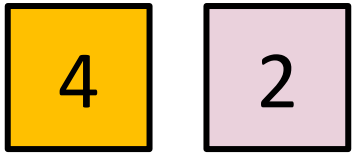
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



Quick Sort

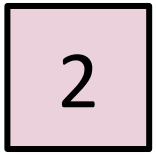
2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



less than *greater than*

Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot

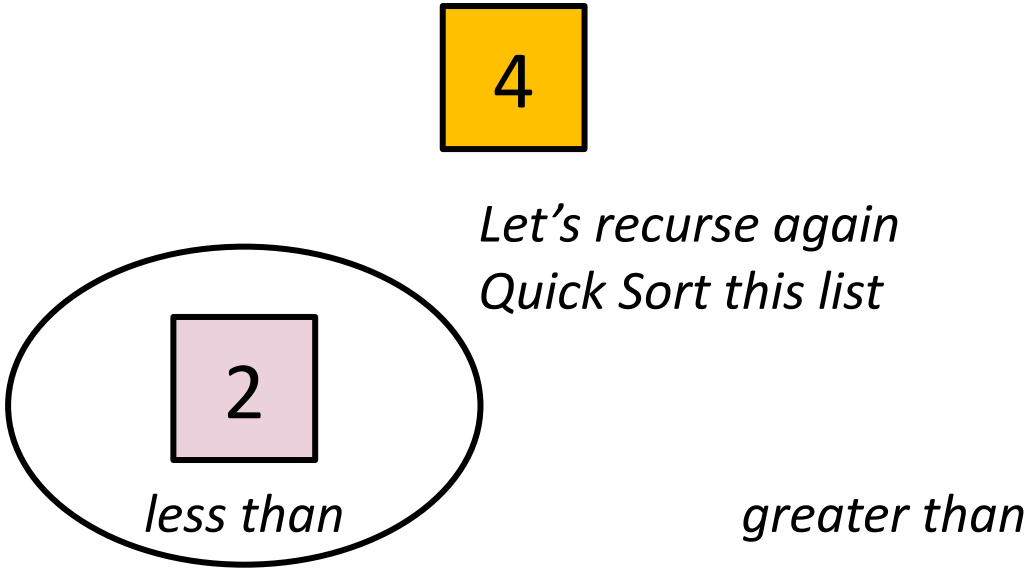


less than

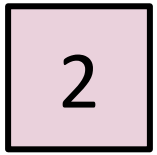
greater than

Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



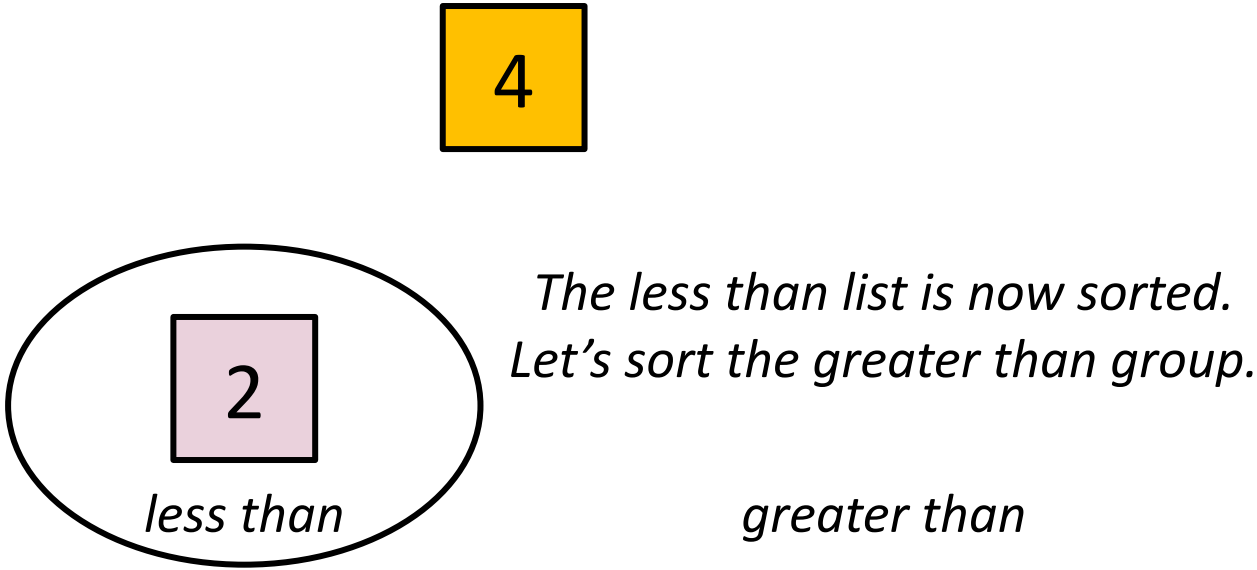
Quick Sort



A list of length 1 is trivially sorted, base case! Let's `return`*.*

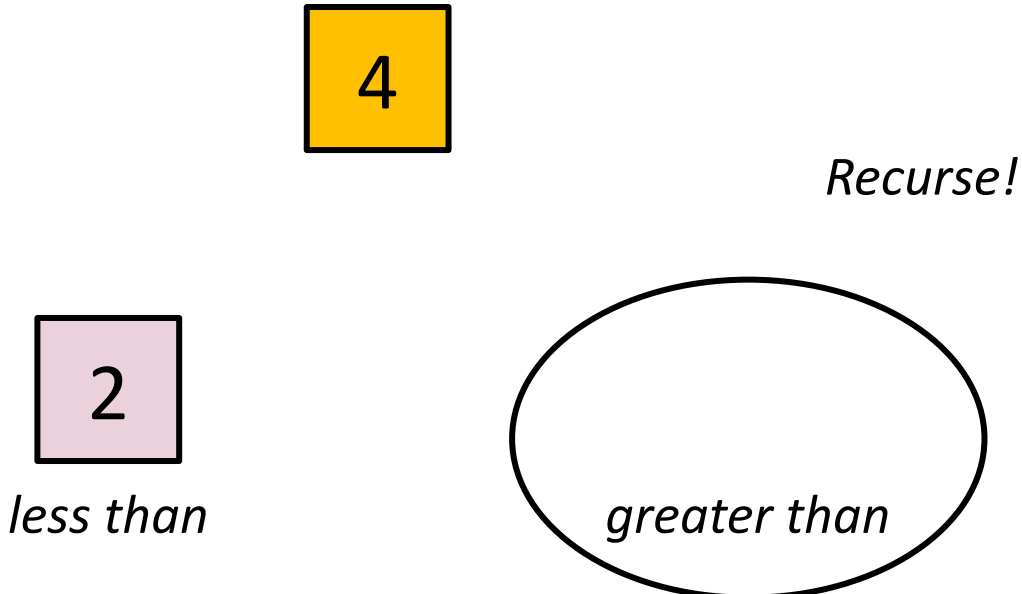
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups

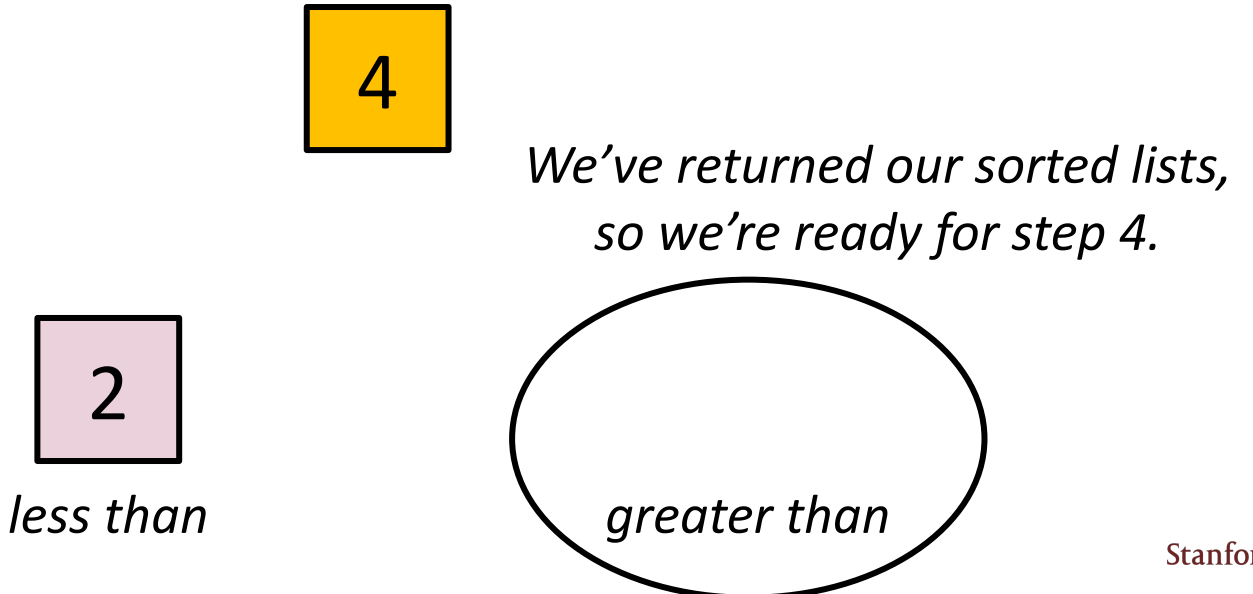


Quick Sort

A list of length 0 is trivially sorted, base case! Let's `return`*.*

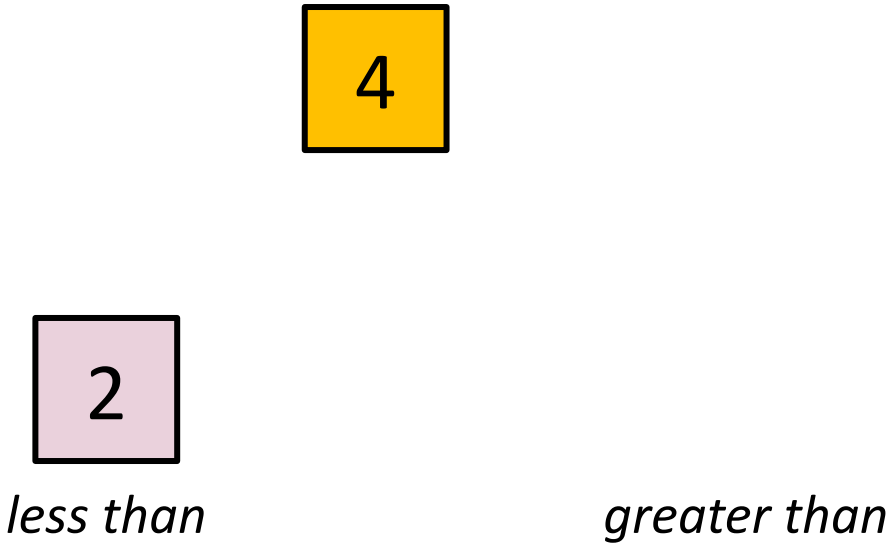
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



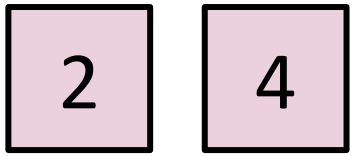
Quick Sort

4. Concatenate the three sorted groups back together again



Quick Sort

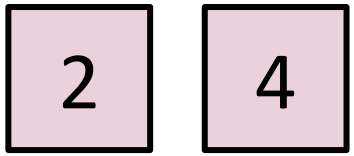
4. Concatenate the three sorted groups back together again



less than *greater than*

Quick Sort

4. Concatenate the three sorted groups back together again

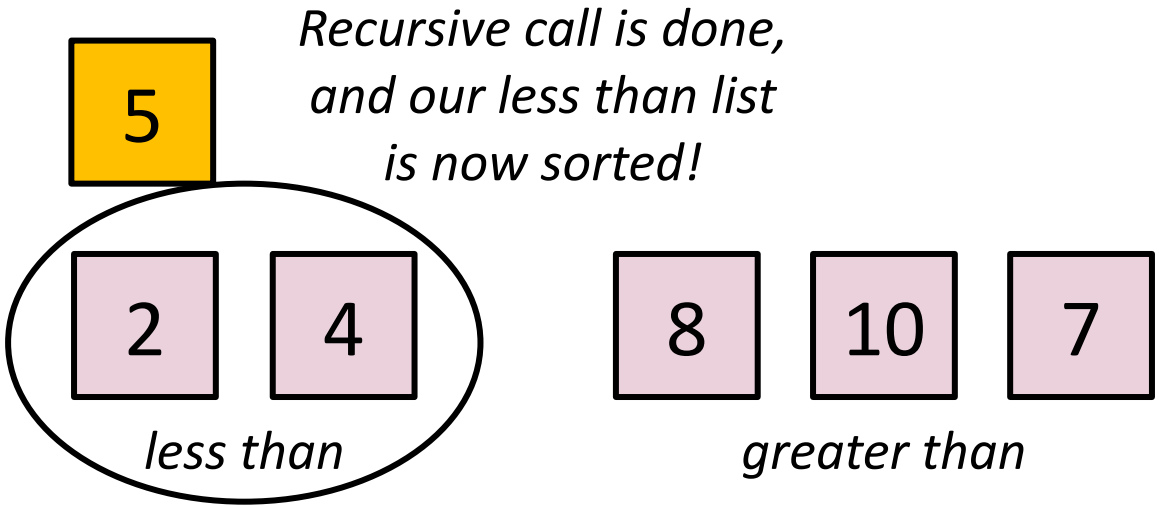


Now this list has been sorted.
`return` to the previous function call.

less than *greater than*

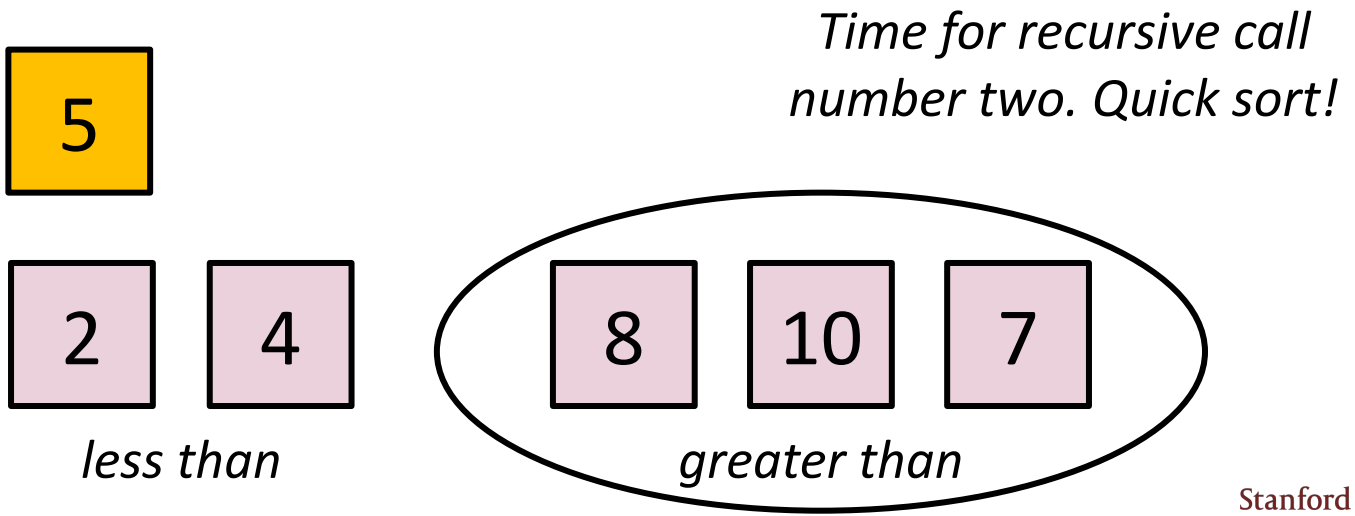
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



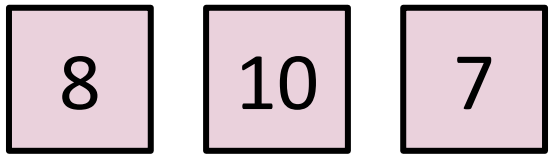
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



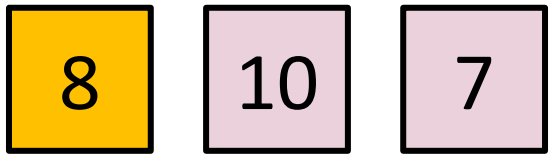
Quick Sort

1. Choose a “pivot” element



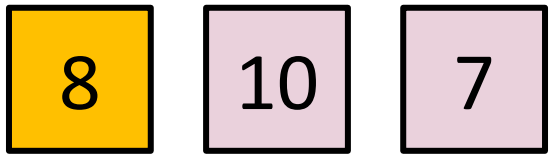
Quick Sort

1. Choose a “pivot” element



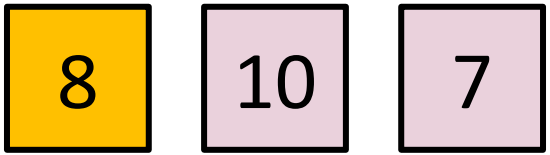
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



Quick Sort

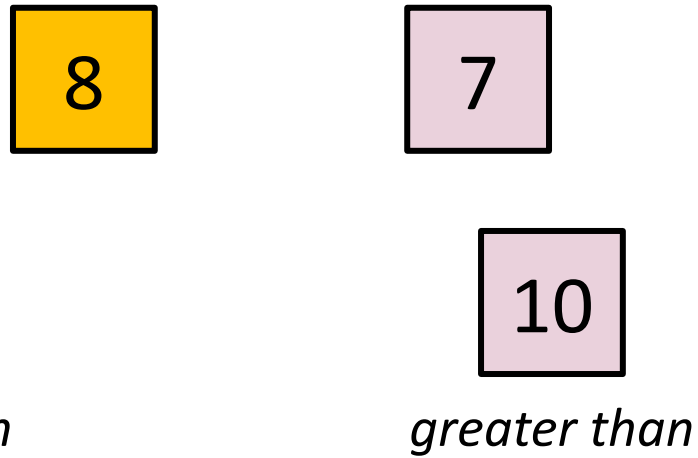
2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



less than *greater than*

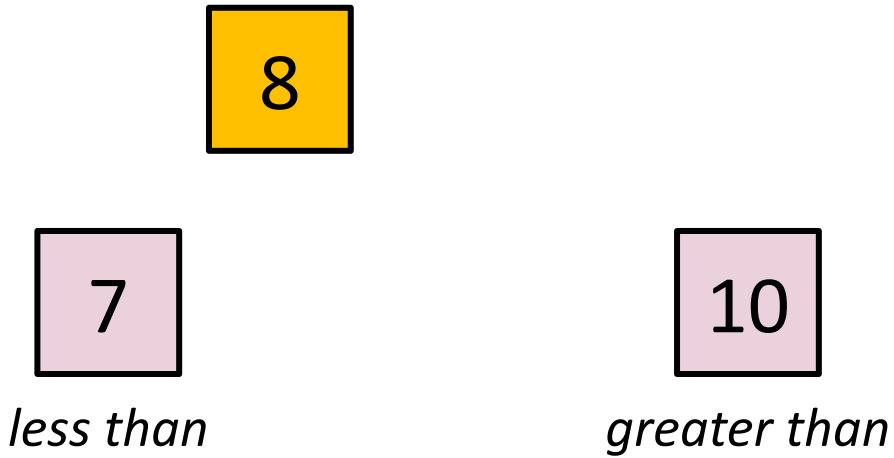
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



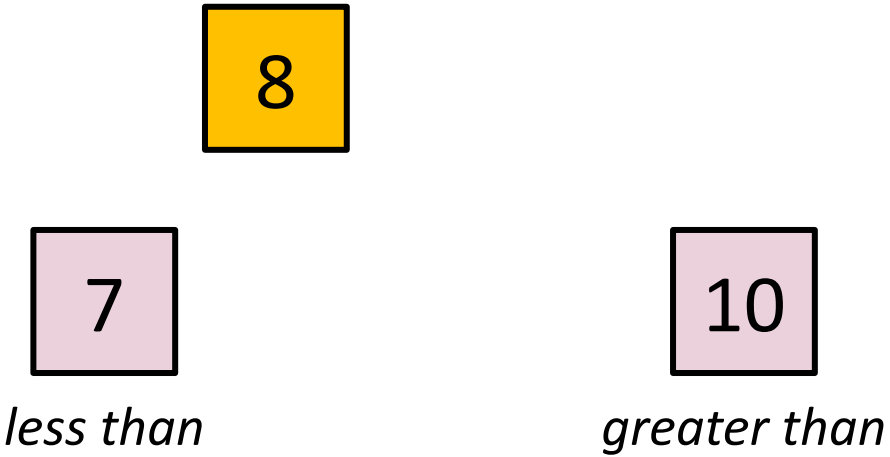
Quick Sort

2. Group your elements into three groups:
    a. Less than pivot
    b. Equal to pivot
    c. Greater than pivot



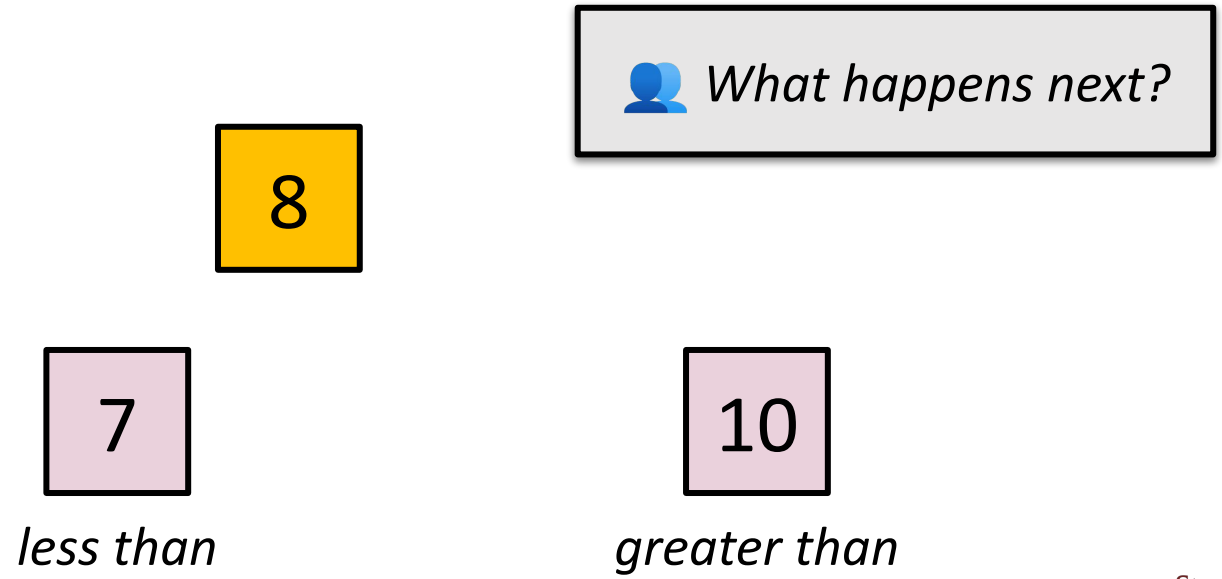
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



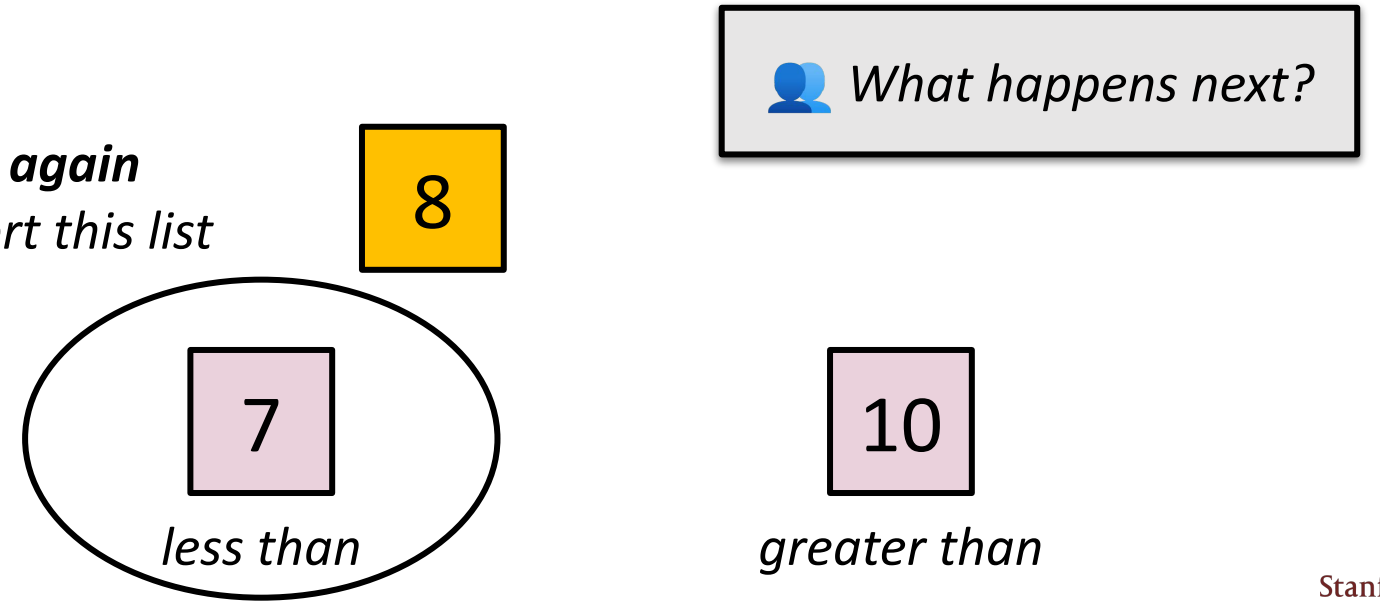
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups

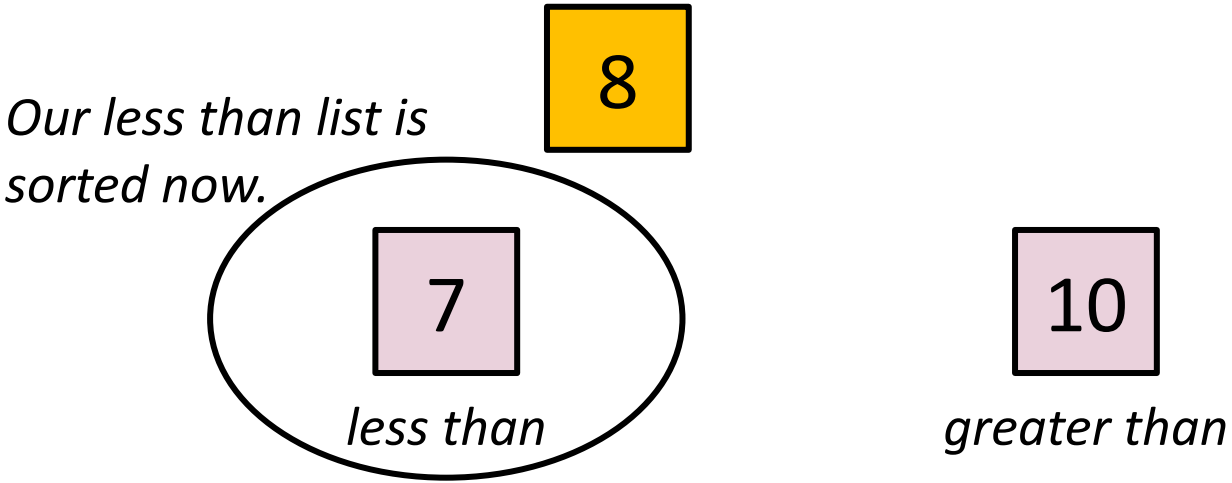


Quick Sort

A list of length 1 is trivially sorted, base case! Let's `return`*.*

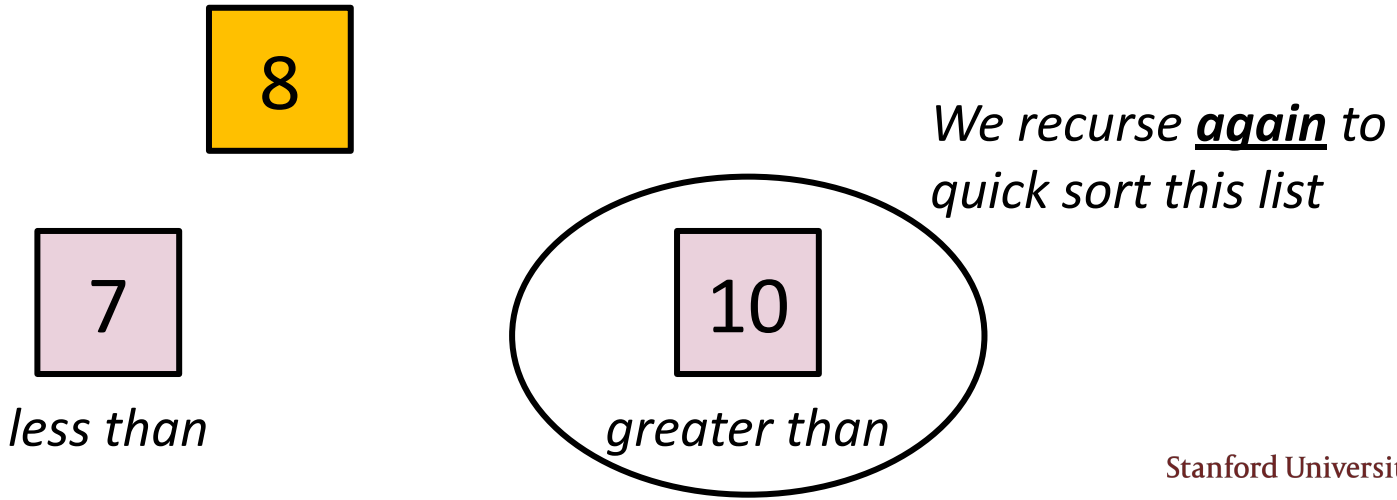
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups

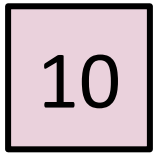


Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



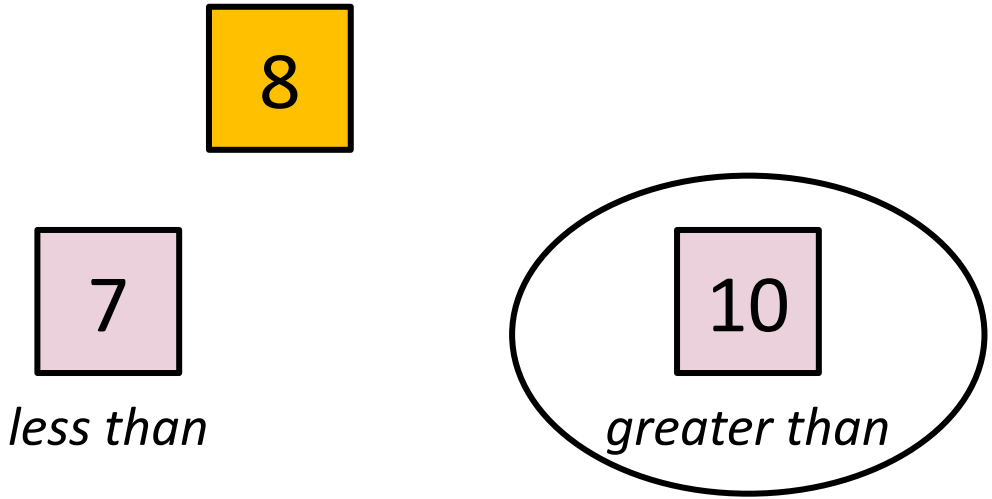
Quick Sort



A list of length 1 is trivially sorted, base case! Let's `return`*.*

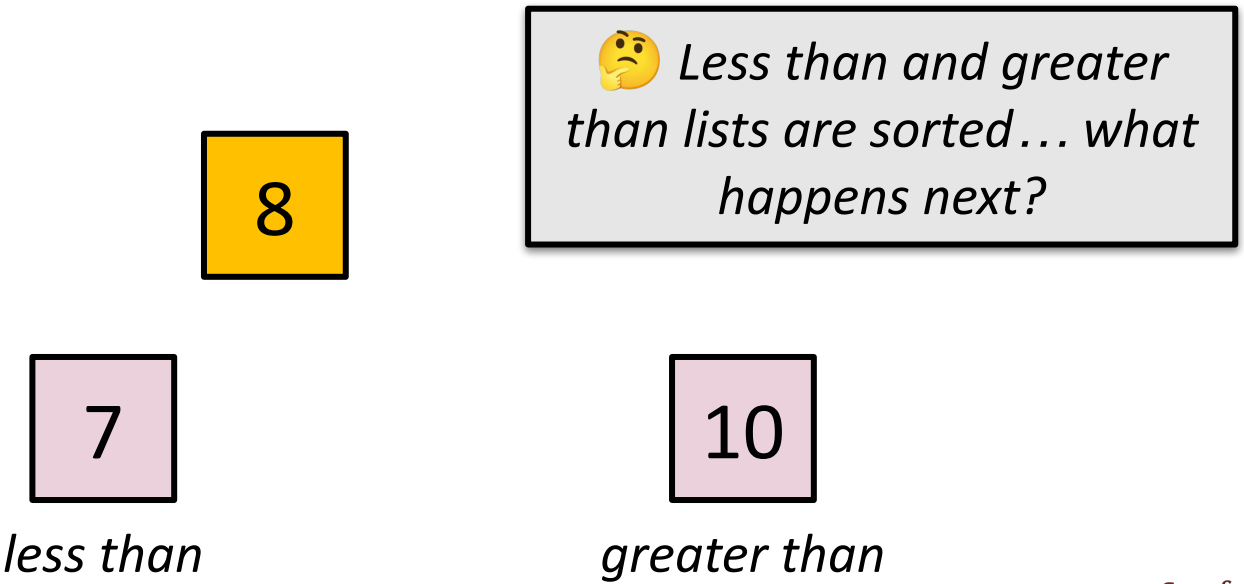
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



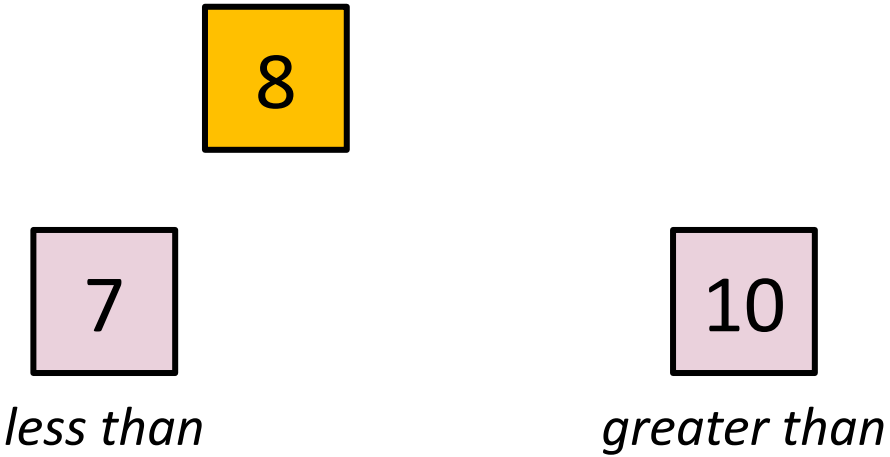
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



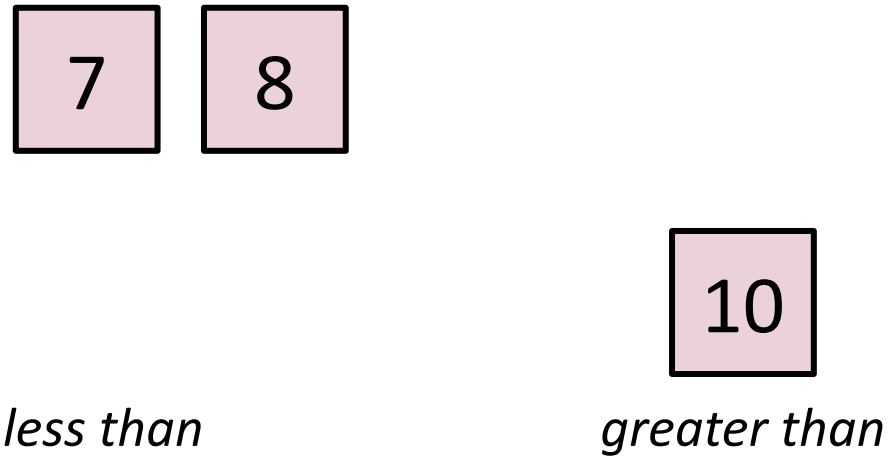
Quick Sort

4. Concatenate the three sorted groups back together again



Quick Sort

4. Concatenate the three sorted groups back together again



Quick Sort

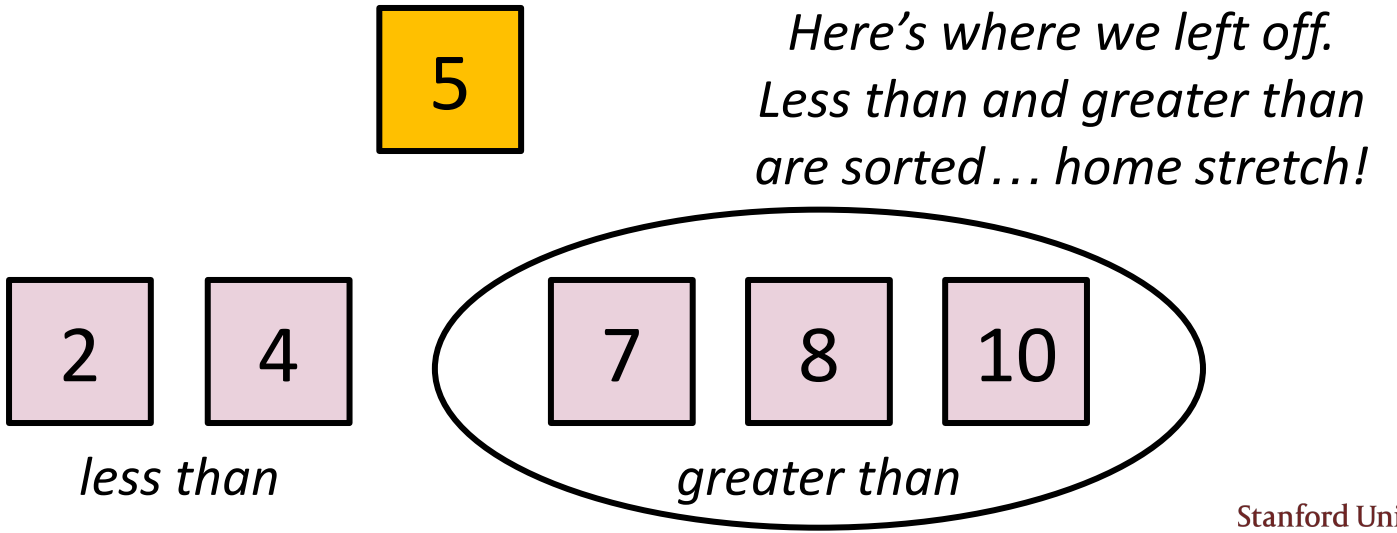
4. Concatenate the three sorted groups back together again

Now this list has been sorted.
`return` to the previous function call.

less than *greater than*

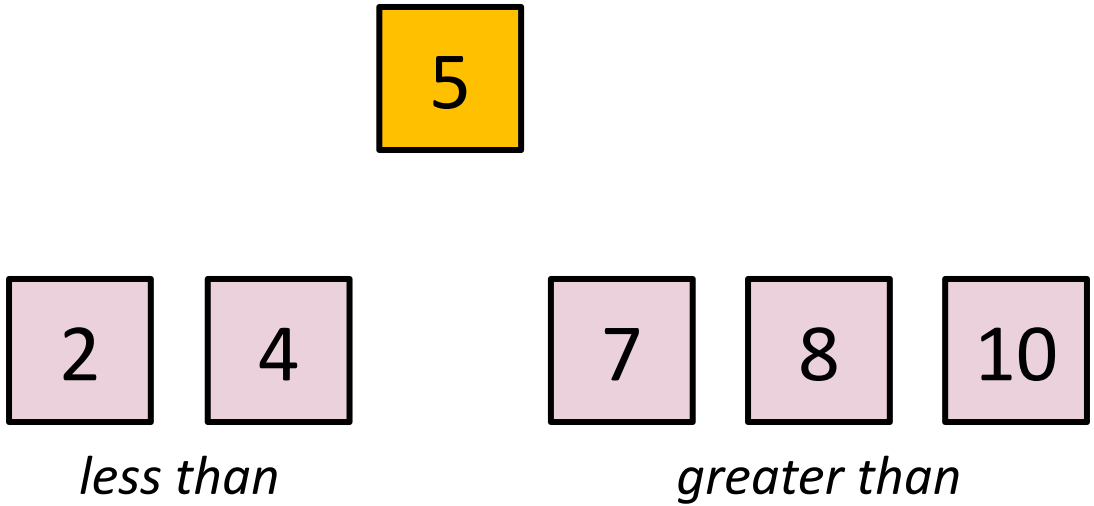
Quick Sort

3. Recursively sort (quick sort) the less than and greater than groups



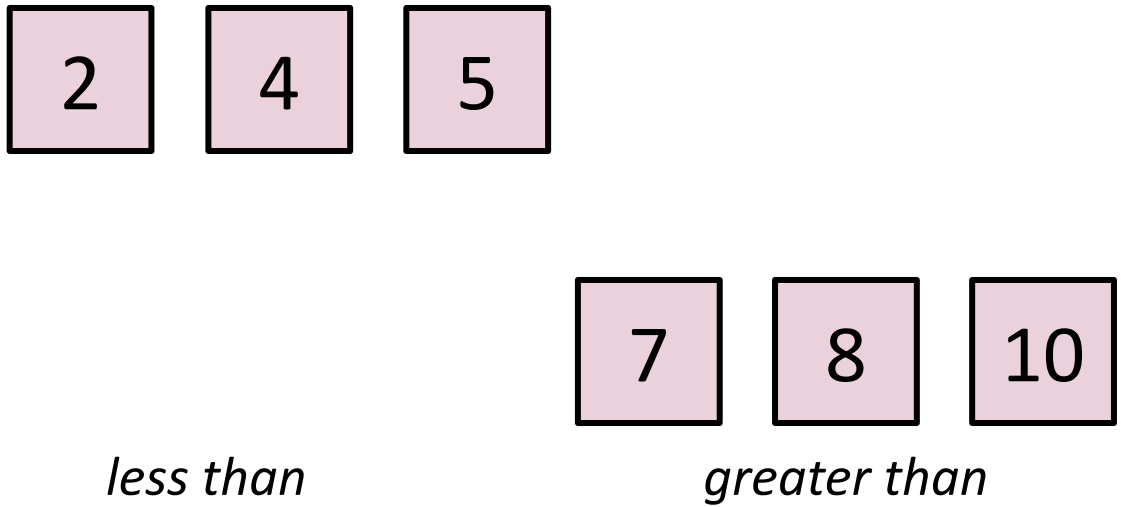
Quick Sort

4. Concatenate the three sorted groups back together again



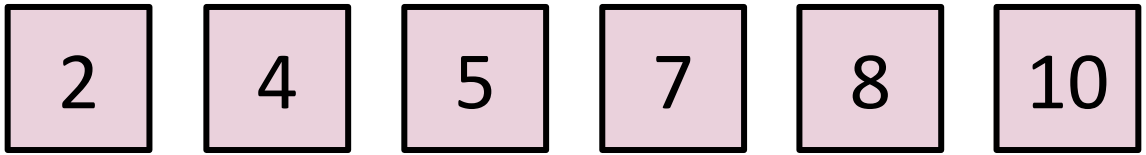
Quick Sort

4. Concatenate the three sorted groups back together again



Quick Sort

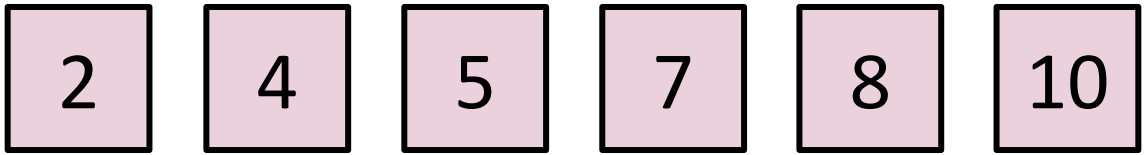
4. Concatenate the three sorted groups back together again



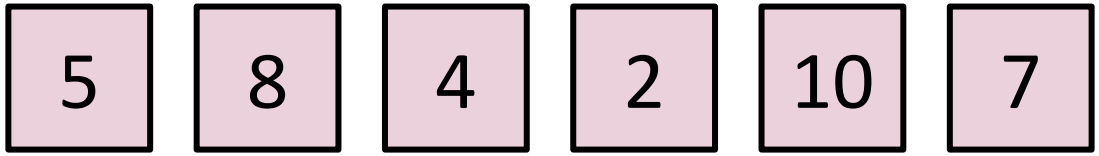
less than *greater than*

Quick Sort

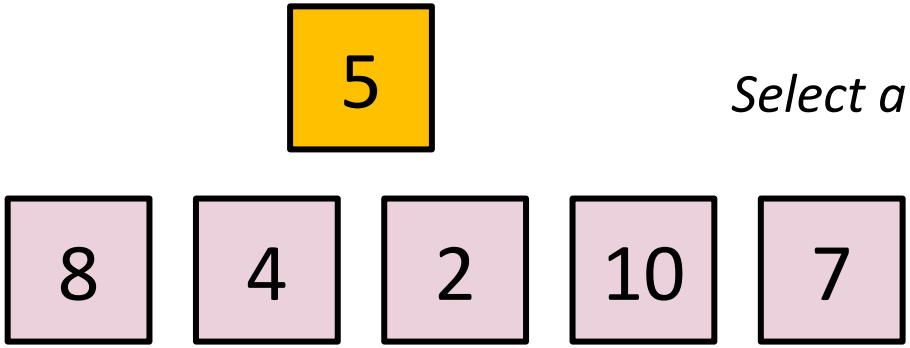
That's quick sort!
Let's look at these recursive calls from a high level.



Quick Sort

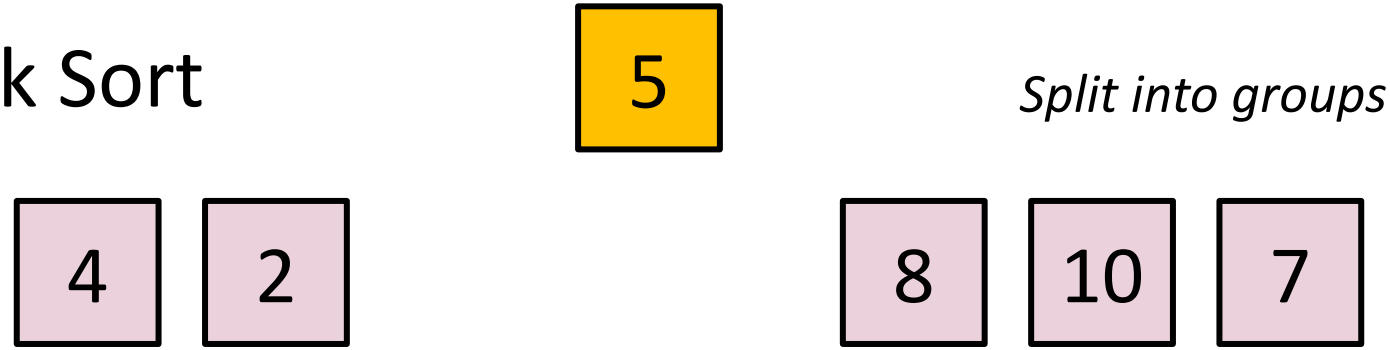


Quick Sort

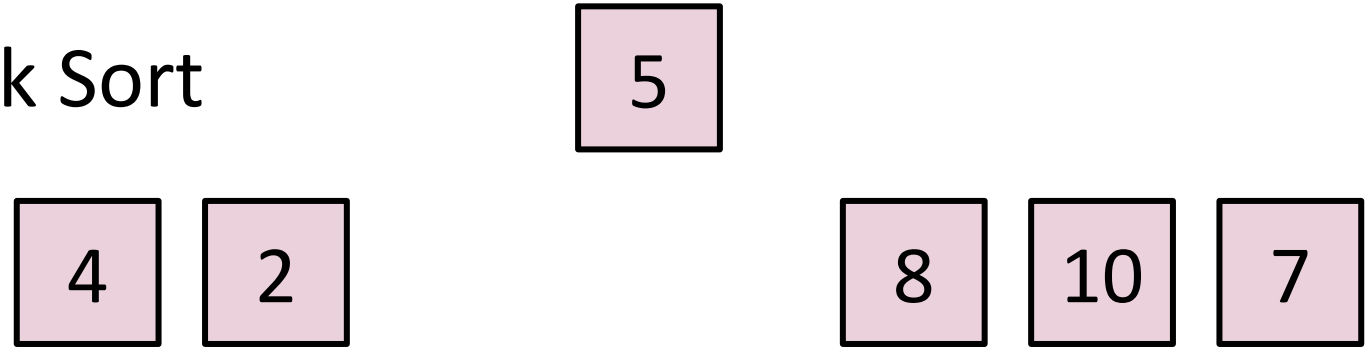


Select a pivot

Quick Sort

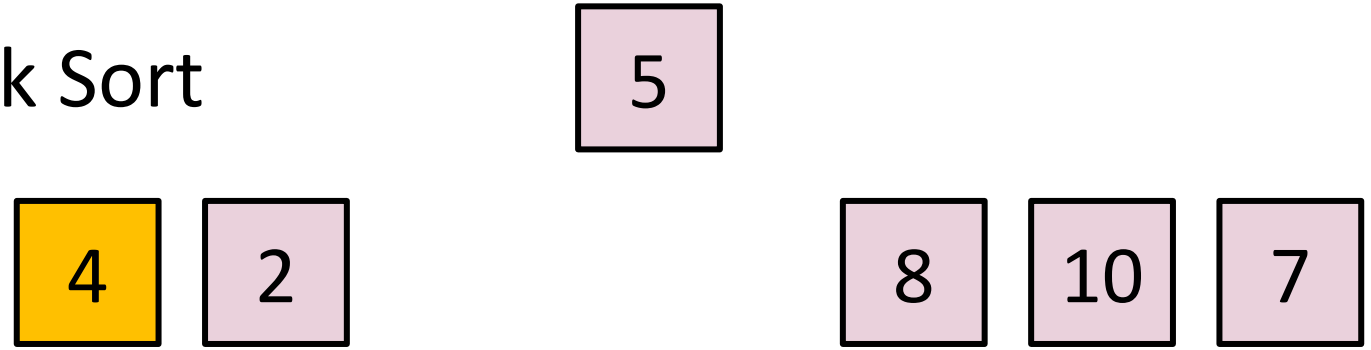


Quick Sort



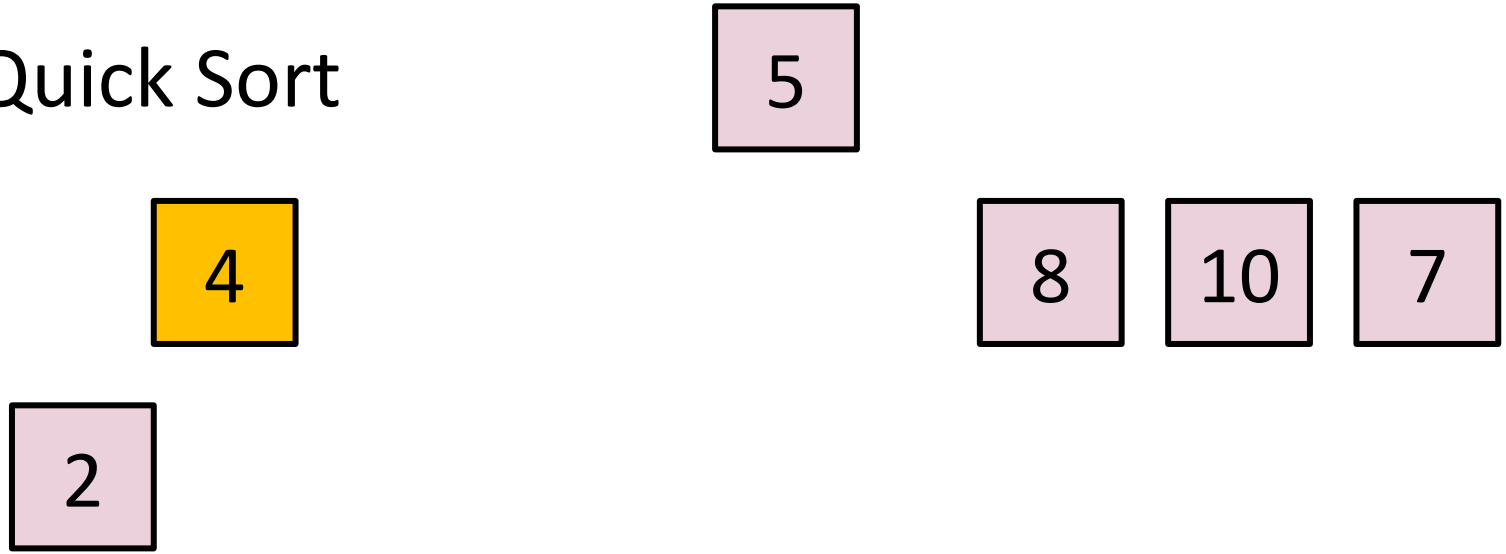
Recursive call:
quick sort

Quick Sort



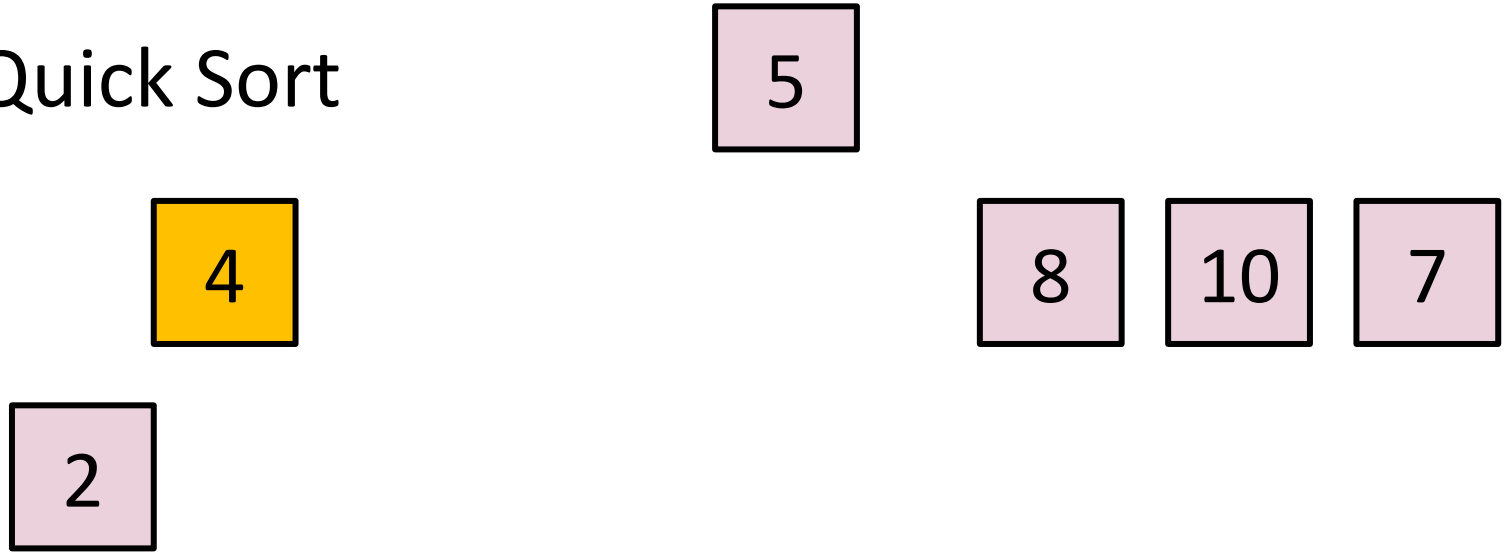
Select a pivot

Quick Sort



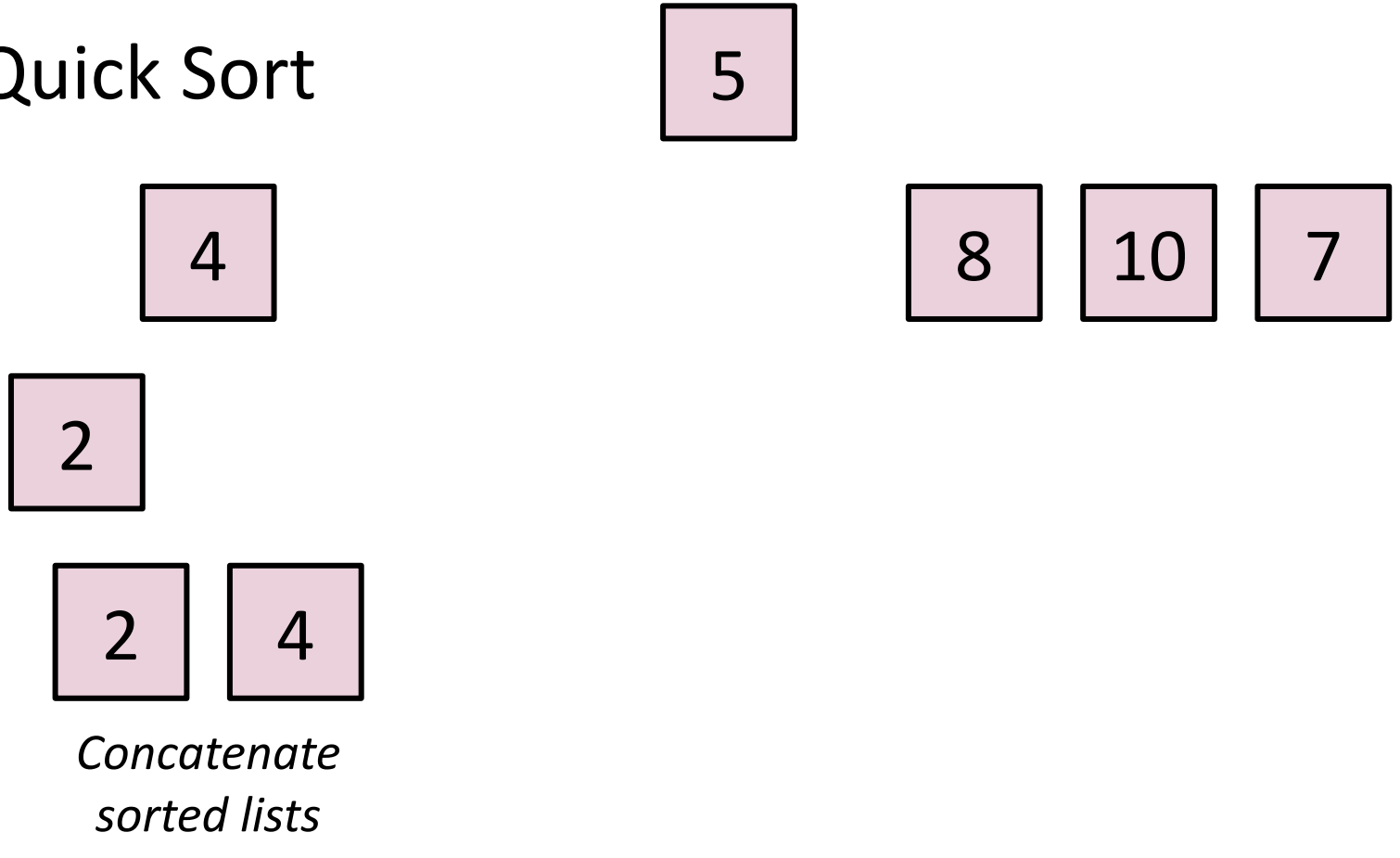
Split into groups

Quick Sort

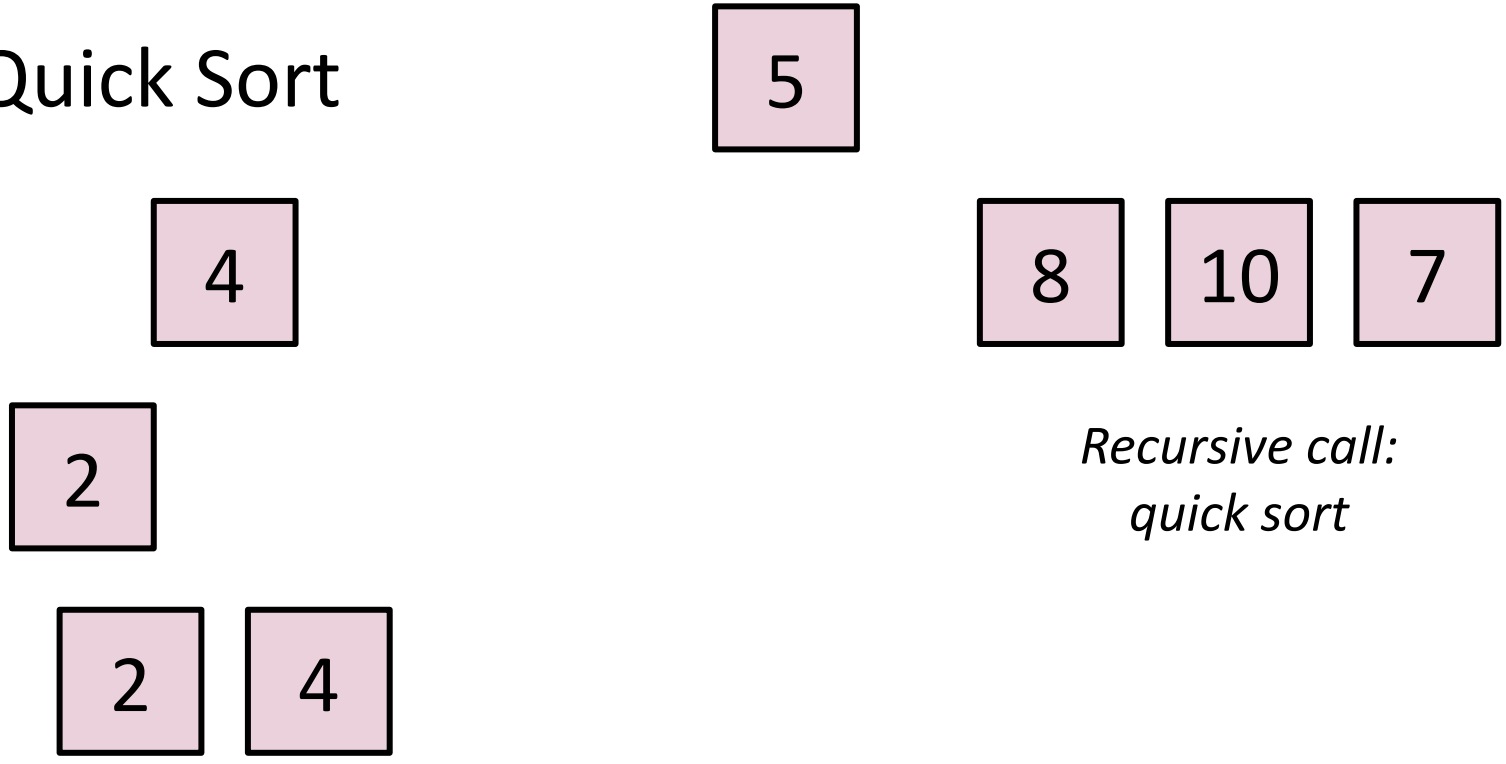


Hit base case:
Sorted!

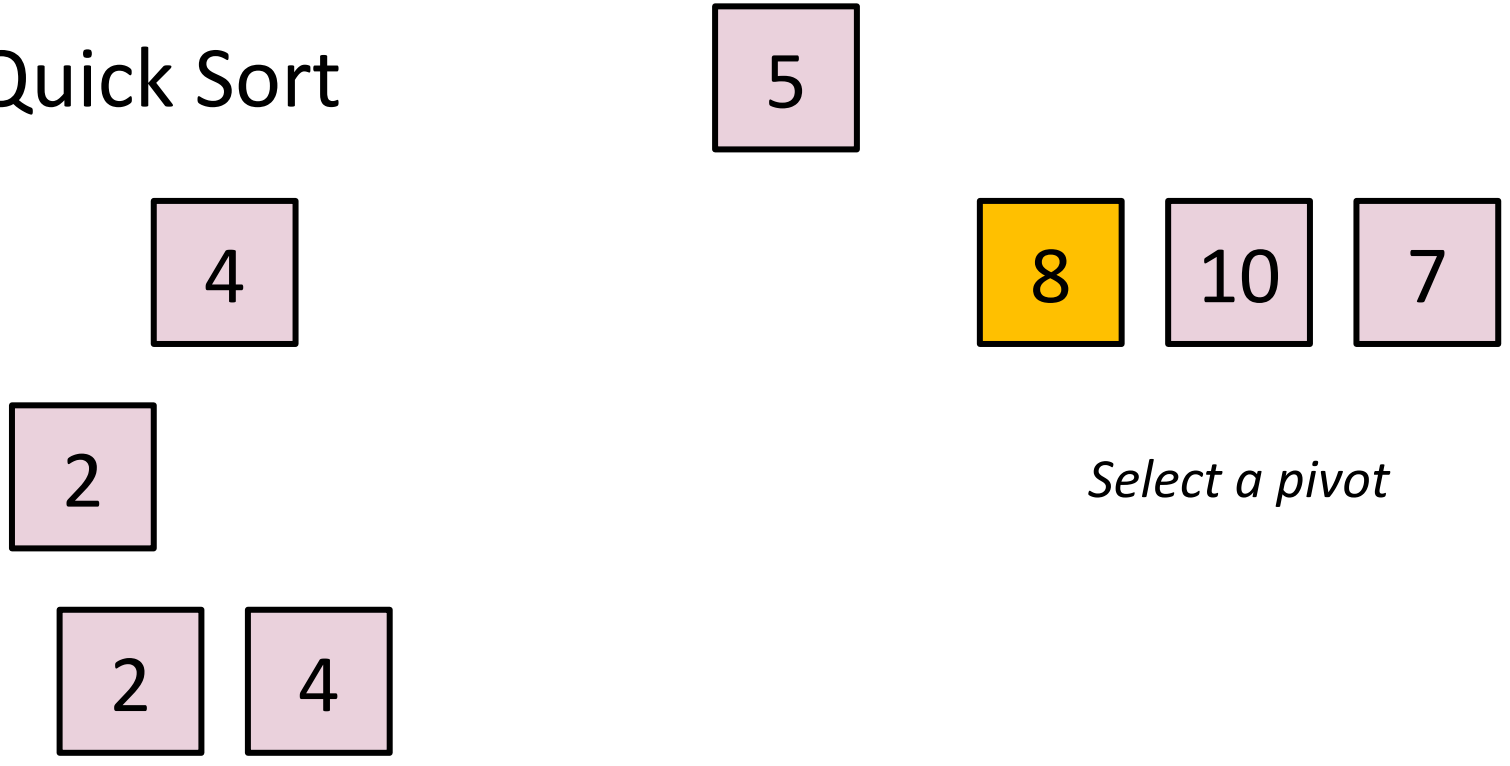
Quick Sort



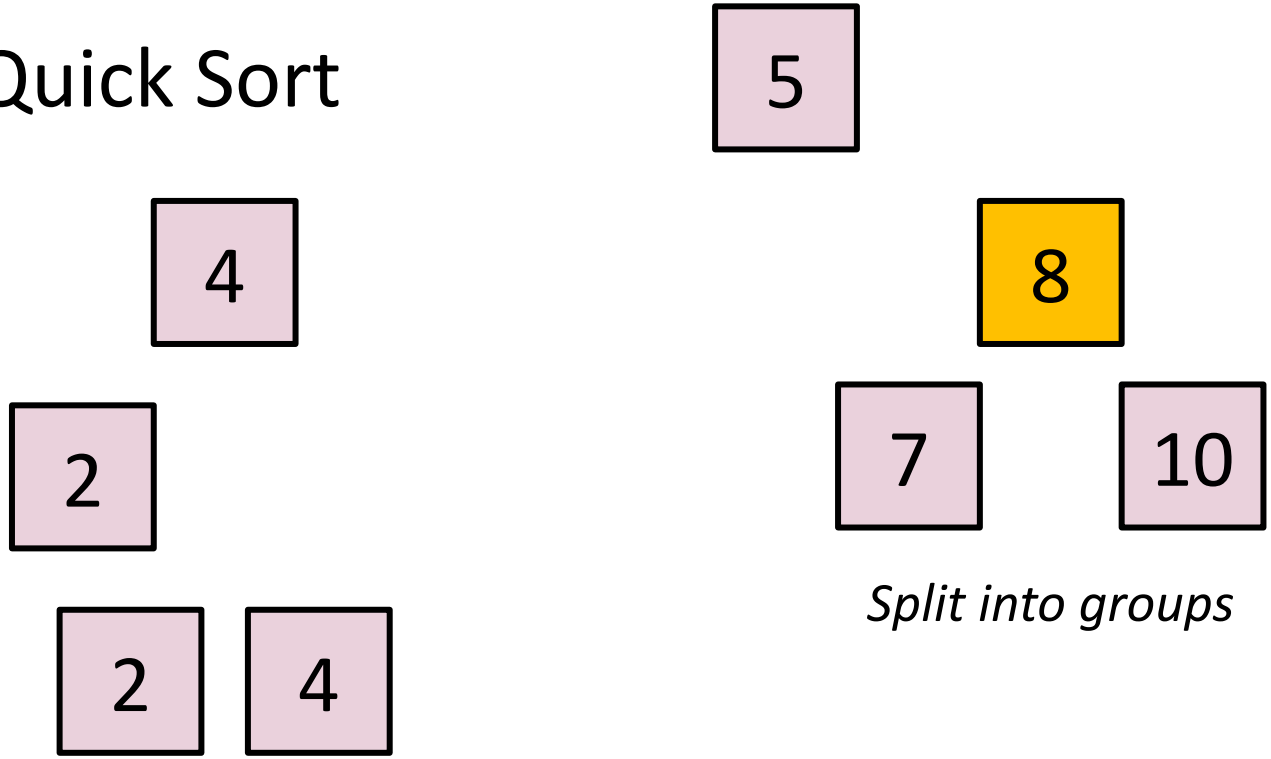
Quick Sort



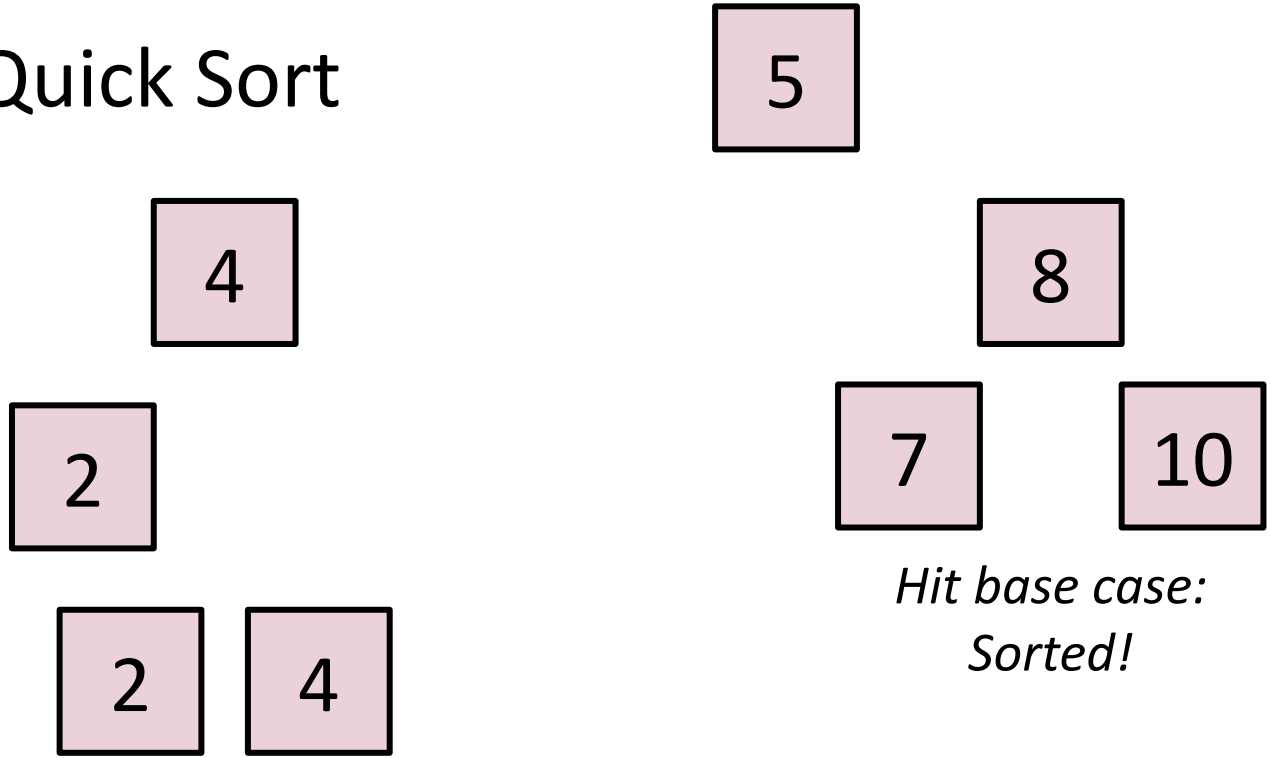
Quick Sort



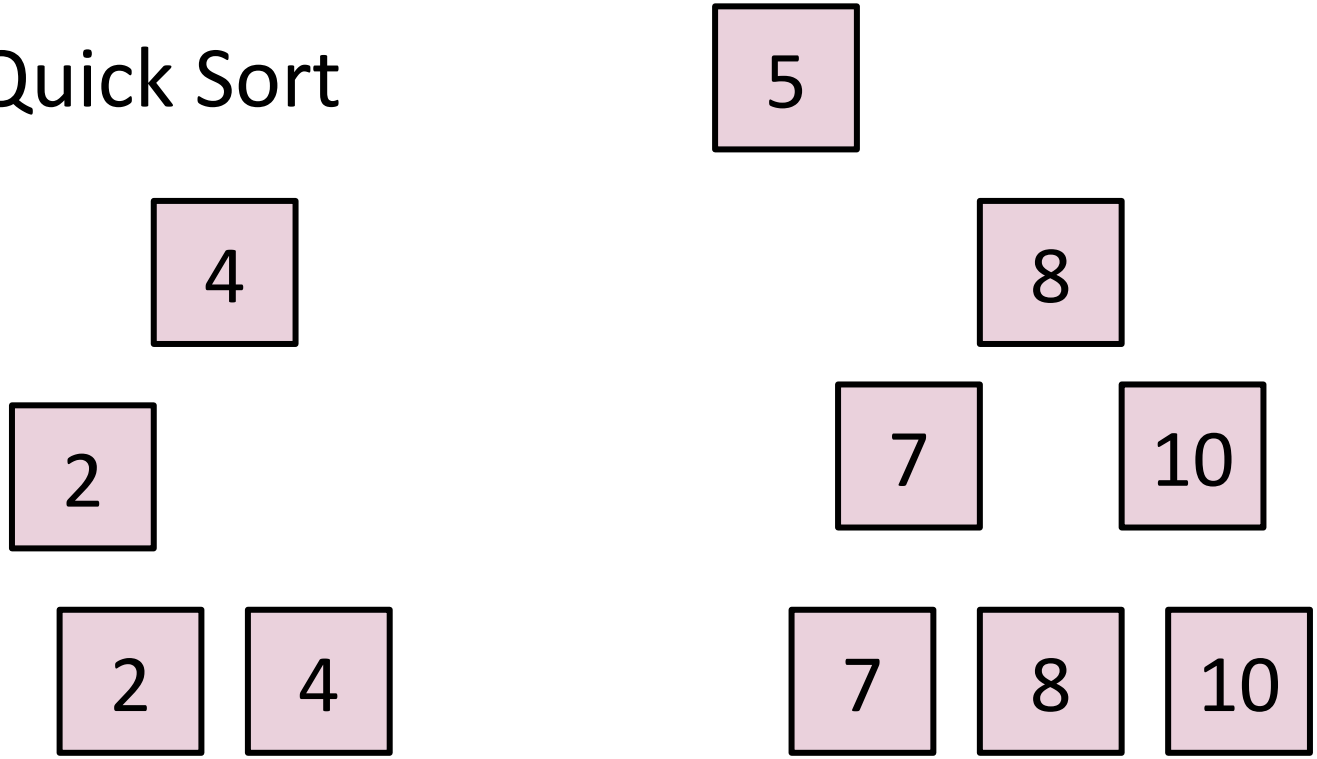
Quick Sort



Quick Sort

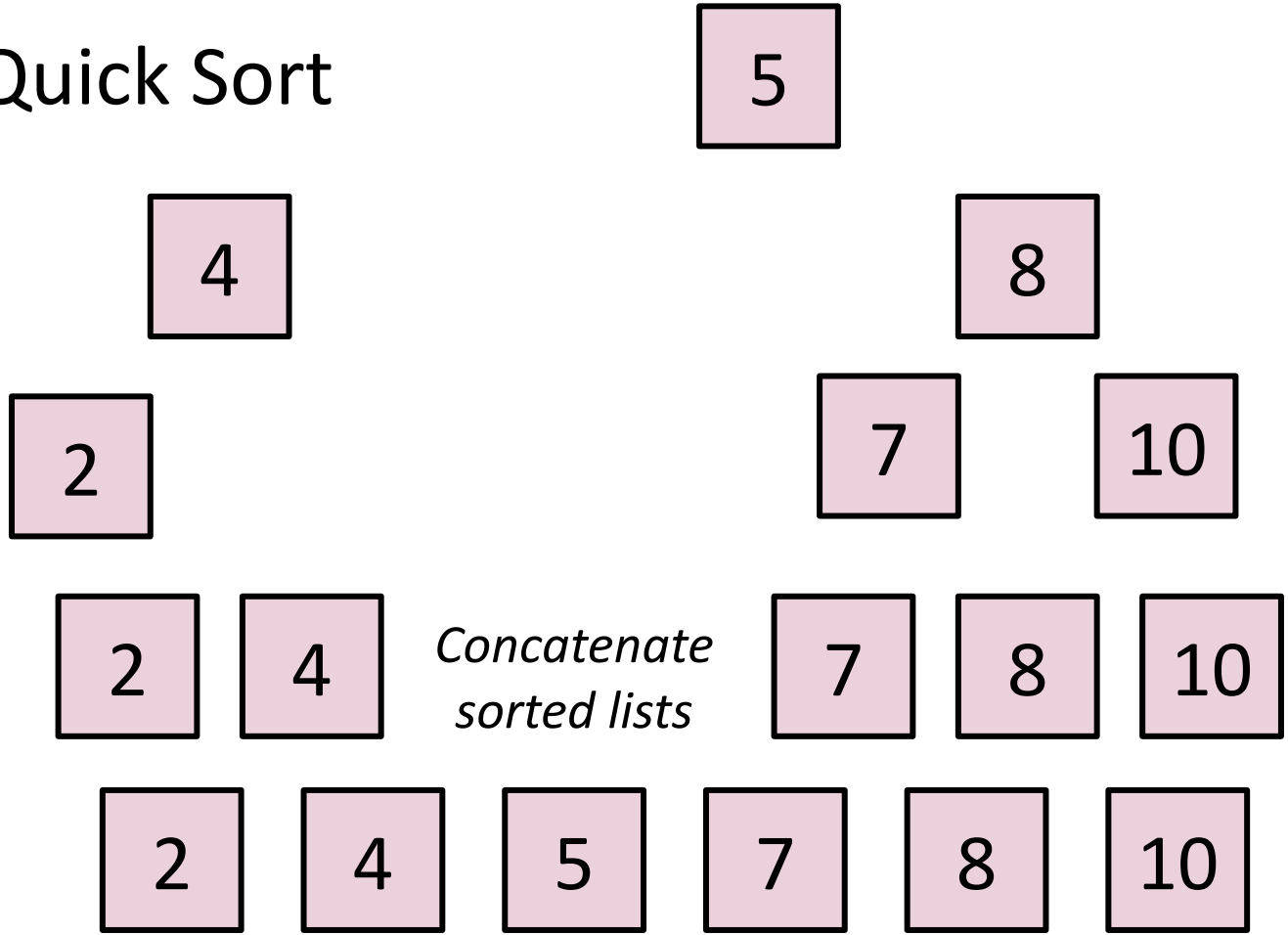


Quick Sort



Concatenate sorted lists

Quick Sort

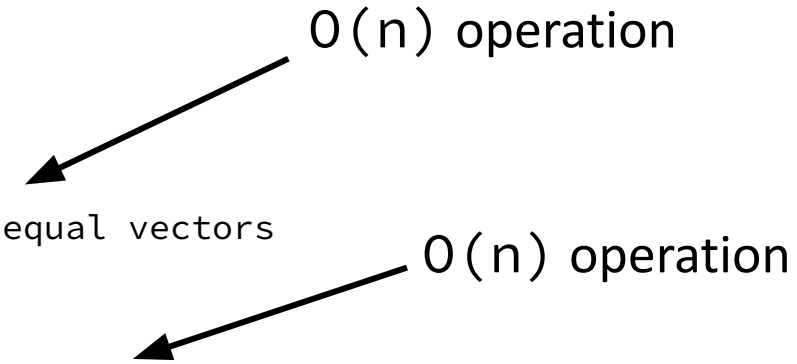


Quick Sort Pseudocode

```
void quickSort(Vector<int>& vec) {
    // base case
    if vector length <= 1, return

    // recursive case
    choose a pivot element
    partition into less, greater, equal vectors
    quickSort(less)
    quickSort(greater)
    concatenate less, equal, and greater
}
```

Quick Sort Runtime

```
void quickSort(Vector<int>& vec) {
    // base case
    if vector length <= 1, return

    // recursive case
    choose a pivot element
    partition into less, greater, equal vectors
    quickSort(less)
    quickSort(greater)
    concatenate less, equal, and greater
}
```



Quick Sort Runtime

- At each level, we do $O(n)$ work
- How many levels are there?

Quick Sort Runtime

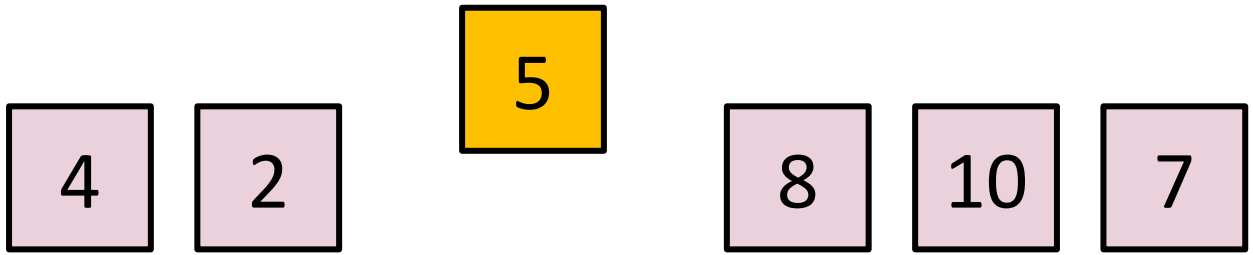
- At each level, we do $O(n)$ work
- How many levels are there?
  - In an average case, we split the list in half at each level: $O(\log n)$
  - In the worst case, we choose a "bad" pivot and have $O(n)$ levels

Quick Sort Runtime

- At each level, we do $O(n)$ work
- How many levels are there?
  - In an average case, we split the list in half at each level: $O(\log n)$
  - In the worst case, we choose a "bad" pivot and have $O(n)$ levels
- Average case runtime: $O(n \log n)$
- Worst case runtime: $O(n^2)$

Quick Sort Recap

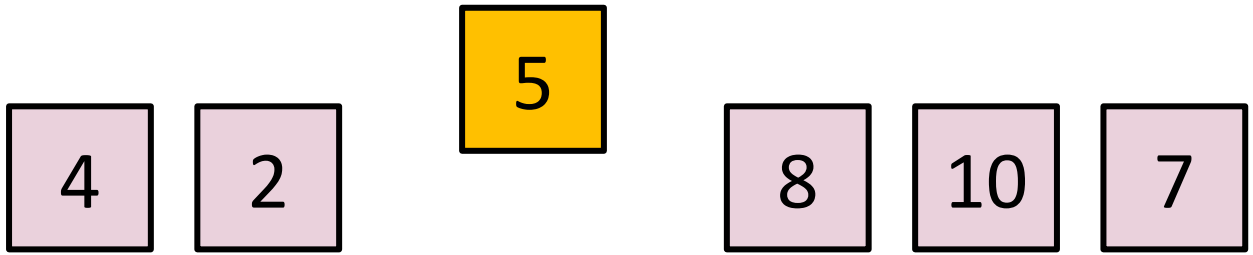
- Split into less, equal, and greater groups, recursively sort less and greater, then concatenate less + equal + greater
- Divide step: hard (partition into three groups)
- Conquer step: easy (concatenate)
- On average, `O(n log n)` sorting algorithm



Quick Sort Recap

- Split into less, equal, and greater groups, recursively sort less and greater, then concatenate less + equal + greater
- Divide step: hard
- Conquer step: ea
- On average, O(n log n) sorting algorithm

Can we do better?



The Fundamental Limit of Sorting Algorithms

Turns out, we can’t do better

How quickly can we sort?

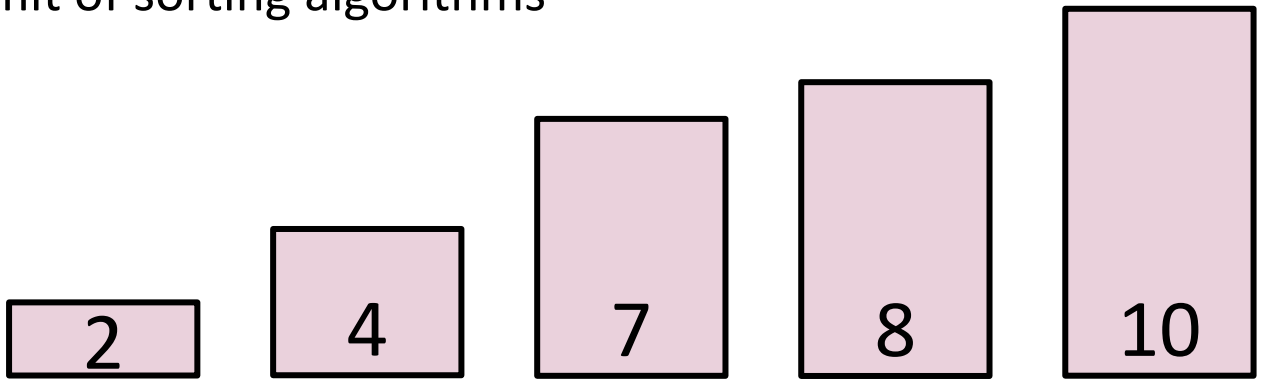
- There's a fundamental limit on the efficiency of sorting algorithms
- It's provable that it is not possible to guarantee a list has been sorted unless you do `O(n log n)` comparisons
  - Take CS161, *Design and Analysis of Algorithms*, to write this proof

How quickly can we sort?

- There's a fundamental limit on the efficiency of sorting algorithms
- It's provable that it is not possible to guarantee a list has been sorted unless you do `O(n log n)` comparisons
  - Take CS161, *Design and Analysis of Algorithms*, to write this proof
- Thus, we can't do better than Merge Sort and Quick Sort, at least in terms of Big-O runtime

Recap

- Intro to sorting: selection sort
- Divide-and-conquer algorithms
  - Merge sort
  - Quick sort
- Fundamental limit of sorting algorithms



Enjoy your weekend!