# CS32 Discussion
# Section 1B
# Week 7

TA: Zhou Ren

# Complexity

- By far the most important concept you need to get out of this class.
  - Hey, I came up with this smart algorithm that runs **in linear time**.
  - This algorithm sucks – **in the worst case** it runs in **exponential time**.
  - This algorithm requires sorting of items, thus it will not get better than **O(n log n)**
  - This problem is **intractable**… I just proved that this cr@p cannot be solved in **polynomial-time**.
- You must to be able to comprehend these conversations, even if you are an EE major!

# Complexity?

- In the first week of class (I think) I said we will not only build programs that work, but programs that work **efficiently**.
- How do we assess efficiency?
- Another way of saying the same thing is: how do we quantify how **complex** a program/ code is?

# Size of input vs. Running time

- A program gets some kind of **input**, does something meaningful (hopefully), and produces some **output**.
- Naturally, the size of input determines how long a program runs.
  - Sorting an array of 1000 items should run a lot longer than sorting an array of 10 items.
  - But how much longer?
- Sometimes, the size of input doesn't matter.
  - Figuring out the size of a C++ string (s.size()): always the same running time.

# Big Question

- Given an input of size **n**, approximately how long does the algorithm take to finish the task, in terms of **n**?

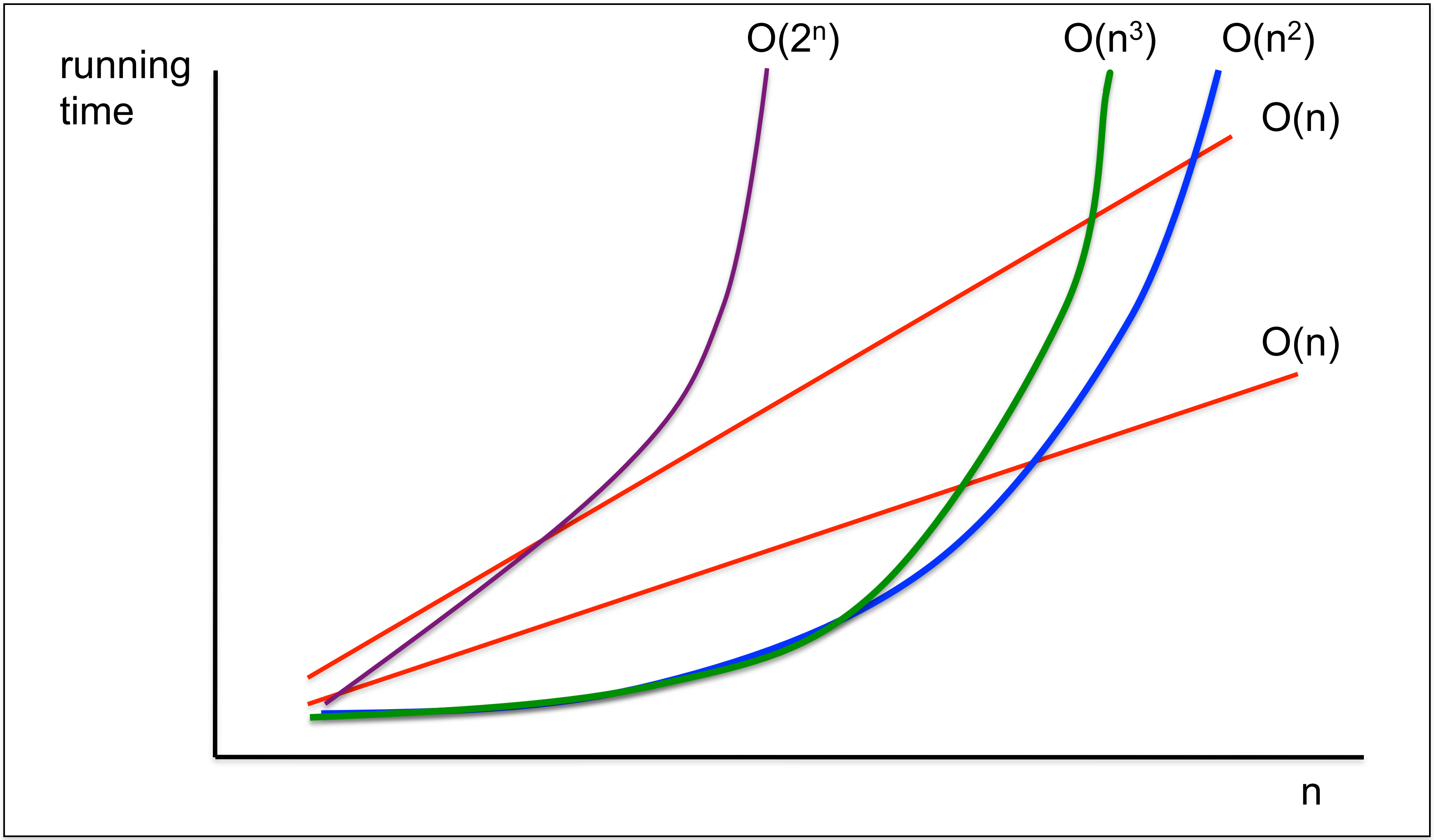
  → Big-O notation

# Big-O

- If your algorithm takes …
  - about n steps: $O(n)$
  - about 2n steps: $O(n)$
  - about $n^2$ steps: $O(n^2)$
  - about $3n^2 + n$ steps: $O(n^2)$
  - about $2^n$ steps: $O(2^n)$

- Which one grows faster in the long term?
  - $10000n$ vs. $0.00001n^2$

# Big-O



# Efficiency

- Algorithms are considered efficient if it runs reasonably fast even with a large input.
- Algorithms are considered inefficient if it runs slow even with a small input.

- For more precise definitions, take CS180 and 181. In this class, we will use these simple intuitions to analyze algorithms.

# Linear Search

- Unsorted array – look for an item

```
linear_search( array arr, size n, value v )
{
  for (i=0 to n-1)
  {
    if (arr[i] == v)
        return i;
  }
  return -1;
}
```

Best case?

Average case?

Worst case?

# Linear Search

- Unsorted array – look for an item

```
linear_search( array arr, size n, value v )
{
  for (i=0 to n-1)
  {
    if (arr[i] == v)
       return i;
  }
  return -1;
}
```

Best case?
v is found in the first slot (a[0]) – takes 1 step

Average case?
takes n/2 = ½n steps (assuming v can appear at any location in the array with an equal probability)

Worst case?
not found – n steps

# Linear Search

- Unsorted array – look for an item

```
linear_search( array arr, size n, value v )
{
  for (i=0 to n-1)
  {
    if (arr[i] == v)
        return i;
  }
  return -1;
}
```

Best case?
v is found in the first slot (a[0]) – **O(1)**

Average case?
**O(n)** (assuming v can appear at any location in the array with an equal probability)

Worst case?
not found – **O(n)**

# Linear Search

- Unsorted array – look for an item

```
linear_search( array arr, size n, value v )
{
  for (i=0 to n-1)
  {
    if (arr[i] == v)
      return i;
  }
  return -1;
}
```

Best case?
v is found in the first slot (a[0]) – **O(1)**

Average case?
**O(n)** (assuming v can appear at any location in the array with an equal probability)

Worst case?
not found – **O(n)**

Usually, assessing complexity involves counting nested loops – only one loop means it is likely to be **O(n)**

# All Pairs

- Find all ordered pairs

```
all_pairs( array arr, size n )
{
  for (i=0 to n-1)
    for (j=0 to n-1)
      if (i ≠ j)
        print "{arr[i] arr[j]}";
}
```

Best case?

Average case?

Worst case?

# All Pairs

- Find all ordered pairs

```
all_pairs( array arr, size n )
{
  for (i=0 to n-1)
    for (j=0 to n-1)
      if (i ≠ j)
        print "{arr[i] arr[j]}";
}
```

Best case?

Average case?

Worst case?

All **O(n²)**

# Unit Operations (O(1) operations)

- Addition, Subtraction, Multiplication, Division
- Comparison, Assignment
- Input, Output of a small value (e.g. short string, an integer, etc.)

- If O(1) operations are repeatedly done in a loop for n times, then that loop is O(n).
- If this loop is within a loop that repeats n times, then this outer loop takes $O(n^2)$.

# Big-O Arithmetic

- More generally:
- If things happen sequentially, we add Big-Os.
  - $O(1)$ operation followed by $O(1) = O(1) + O(1) = O(1)$
- If one thing happens within another, then we multiply Big-Os.
  - $O(1)$ operation within a $O(n)$ loop $= O(1) \times O(n) = O(n)$

- $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
- $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

# Order of Complexity

| Big O | Name | n = 128 |
|---|---|---|
| $O(1)$ | constant | 1 |
| $O(\log n)$ | logarithmic | 7 |
| $O(n)$ | linear | 128 |
| $O(n \log n)$ | "n log n" | 896 |
| $O(n^2)$ | quadratic | 16192 |
| $O(n^k)$, $k >= 1$ | polynomial | |
| $O(2^n)$ | exponential | $10^{40}$ |
| $O(n!)$ | factorial | $10^{214}$ |

# Linear Search

- Unsorted array – look for an item

```
linear_search( array arr, size n, value v )
{
  for (i=0 to n-1) O(n)
  {
    if (arr[i] == v)
        return i;      O(1)
  }
  return -1;
}
```

Best case?
v is found in the first slot (a[0]) – **O(1)**

Average case?
**O(n)** (assuming v can appear at any location in the array with an equal probability)

Worst case?
not found – **O(n)**

Usually, assessing complexity involves counting nested loops – only one loop means it's likely to be **O(n)**

# All Pairs

- Find all ordered pairs

```
all_pairs( array arr, size n )
{
  for (i=0 to n-1)  O(n)
    for (j=0 to n-1)  O(n)
      if (i ≠ j)
        print "{arr[i] arr[j]}";  O(1)
}
```

Best case?

Average case?

Worst case?

All **$O(n^2)$**

# Binary Search

- Find an item v in a **sorted** array

```
binary_search( array arr, value v, start index s, end index e )
{
   if (s > e)
       return -1
   find the middle point i = (s + e) / 2
   if (arr[i] == v)
      return i
   else if (arr[i] < v)
      return binary_search(arr, v, i+1, e)
   else
      return binary_search(arr, v, s, i-1)
}
```

Best case?

Average case?

Worst case?

# Binary Search

- Find an item v in a **sorted** array

```
binary_search( array arr, value v, start index s, end index e )
{
  if (s > e)
      return -1
  find the middle point i = (s + e) / 2
  if (arr[i] == v)
     return i
  else if (arr[i] < v)
     return binary_search(arr, v, i+1, e)
  else
     return binary_search(arr, v, s, i-1)
}
```

Best case?
O(1) – by now you can see the best case analysis doesn't help much

Average case?
**O(log n)**

Worst case?
**O(log n)**

# Binary Search

- At every iteration, we divide the search space in half.
- You keep dividing the size by 2 until it becomes 1.
- It takes ~ $\log_2 n$ steps to get to 1.
- $\log_{10} n = (1 / \log_2 10) \log_2 n$
- So the base does not matter.

# Why Big-O's are important

- You'll be asked about it in job interviews!!!!!

# Sorting Algorithms

- We now switch gears and discuss some well known sorting algorithms.

# Selection Sort

| | | | | |
|---|---|---|---|---|
| 4 | 3 | 1 | 5 | 2 |
| **1** | 3 | 4 | 5 | 2 |
| **1** | **2** | 4 | 5 | 3 |
| **1** | **2** | **3** | 5 | 4 |
| **1** | **2** | **3** | **4** | 5 |

- Find the smallest item in the unsorted portion, and place it in front.
- What is the running time (complexity) of this algorithm?

# Insertion Sort

| | | | | |
|---|---|---|---|---|
| 4 | 3 | 1 | 5 | 2 |
| 3 | 4 | 1 | 5 | 2 |
| 1 | 3 | 4 | 5 | 2 |
| 1 | 3 | 4 | 5 | 2 |
| 1 | 2 | 3 | 4 | 5 |

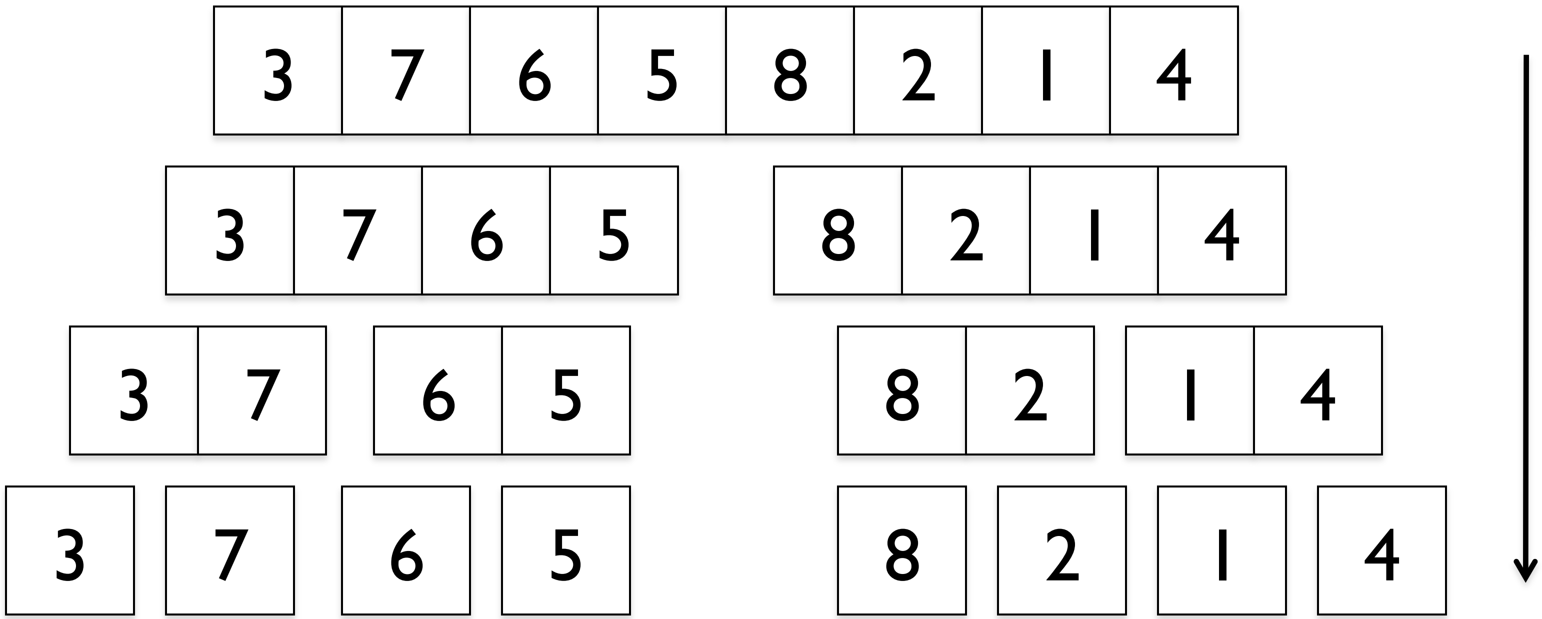
- Pick one from the unsorted part, and place it in the "right" position in the sorted part.
- Best case?
- Avg. case?
- Worst case?

# Insertion Sort

| | | | | |
|---|---|---|---|---|
| **4** | **3** | 1 | 5 | 2 |
| **3** | **4** | **1** | 5 | 2 |
| **1** | **3** | **4** | **5** | 2 |
| **1** | **3** | **4** | **5** | **2** |
| **1** | **2** | **3** | **4** | **5** |

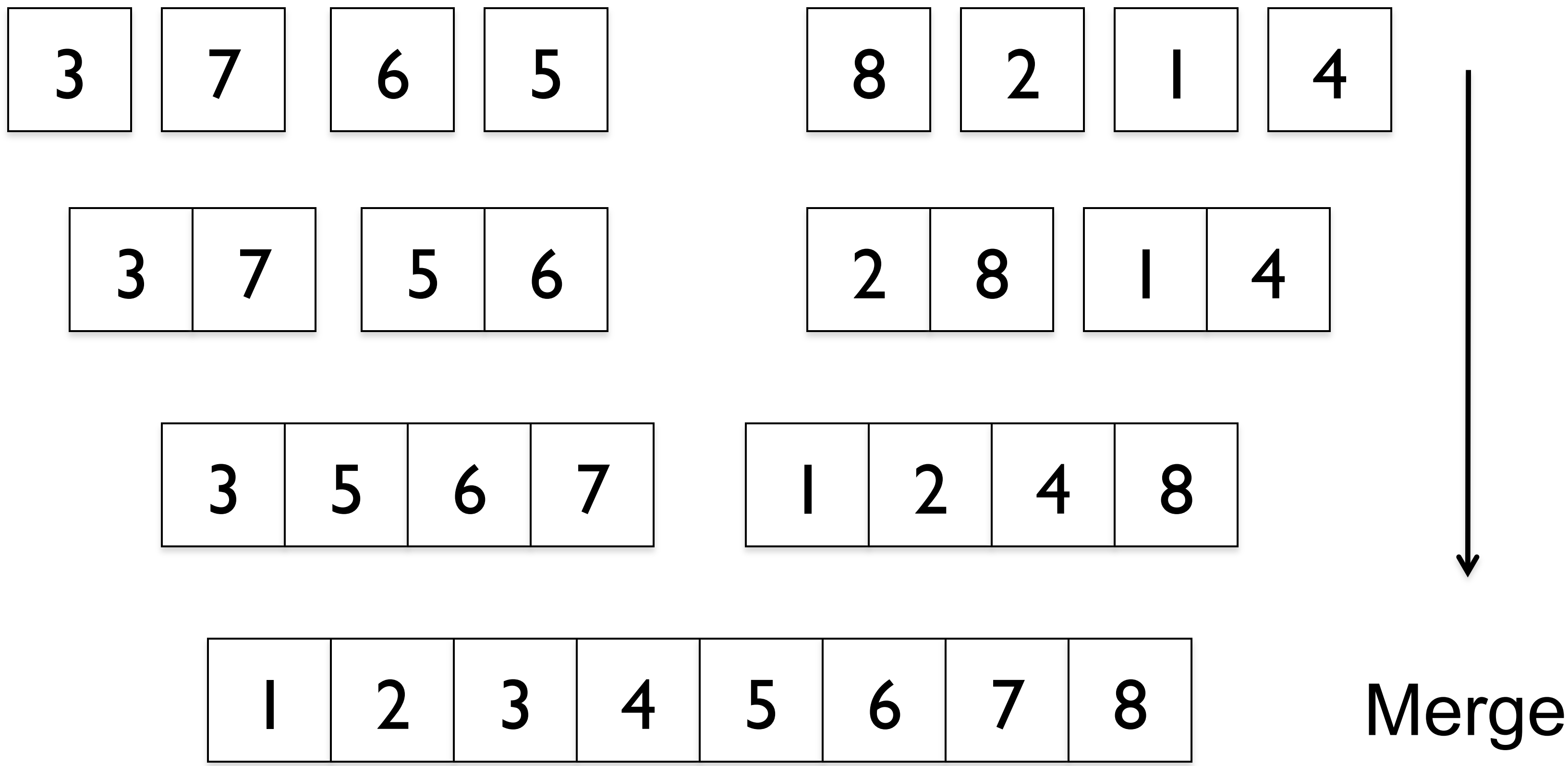
- Pick one from the unsorted part, and place it in the "right" position in the sorted part.
- Best case? **O(n)**
- Avg. case? **O($n^2$)**
- Worst case? **O($n^2$)**

# Merge Sort

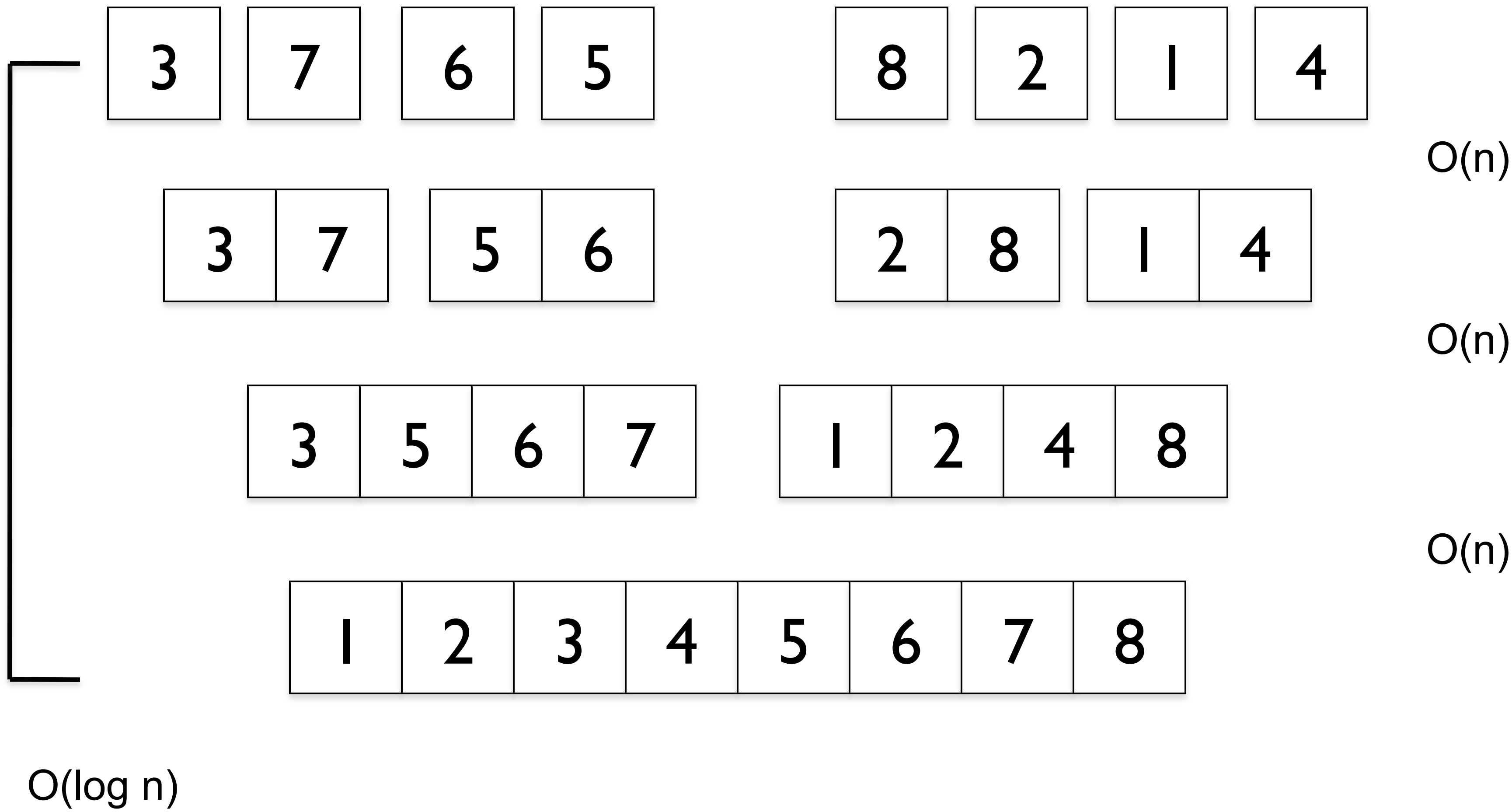



Keep splitting

# Merge Sort



# Merge Sort: Running Time?

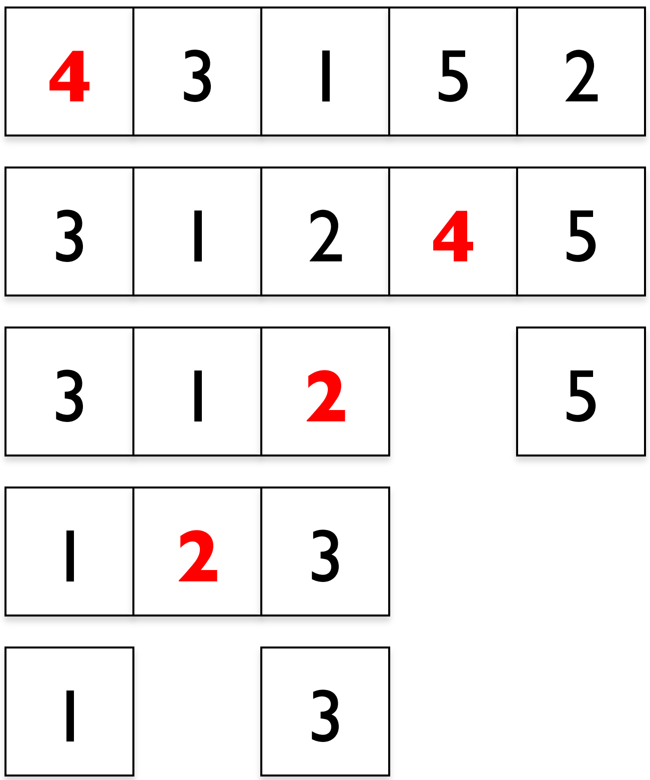



$O(n)O(\log n) = \mathbf{O(n \log n)}$

# General Sorting: Running Time

- $O(n \log n)$ is faster than $O(n^2)$ – merge sort is more efficient than selection sort or insertion sort.
- $O(n \log n)$ is the best average complexity that a general (comparison) sorting algorithm can get (assuming you know nothing about the data set).
- With more information about the data set provided, you can sometimes sort things almost linearly.

# Quick Sort




- Pick a **pivot**, and move numbers that are less than the pivot to front, and ones that are greater than the pivot to end. (Does this sound familiar?)
- On average, $O(n \log n)$
- Depending on how you pick your pivots, it can be as bad as $O(n^2)$

# Quick Questions

- Given an unsorted array of n items, what is the best you can do to search for an item, if you are to run this search only once?
- Given an unsorted array of n items, what is the best you can do to search for an item, if you are to run this search 100 times? (assume: n >> 100)
- Given an unsorted array of n items, what is the best you can do to search for an item, if you are to run this search n times?