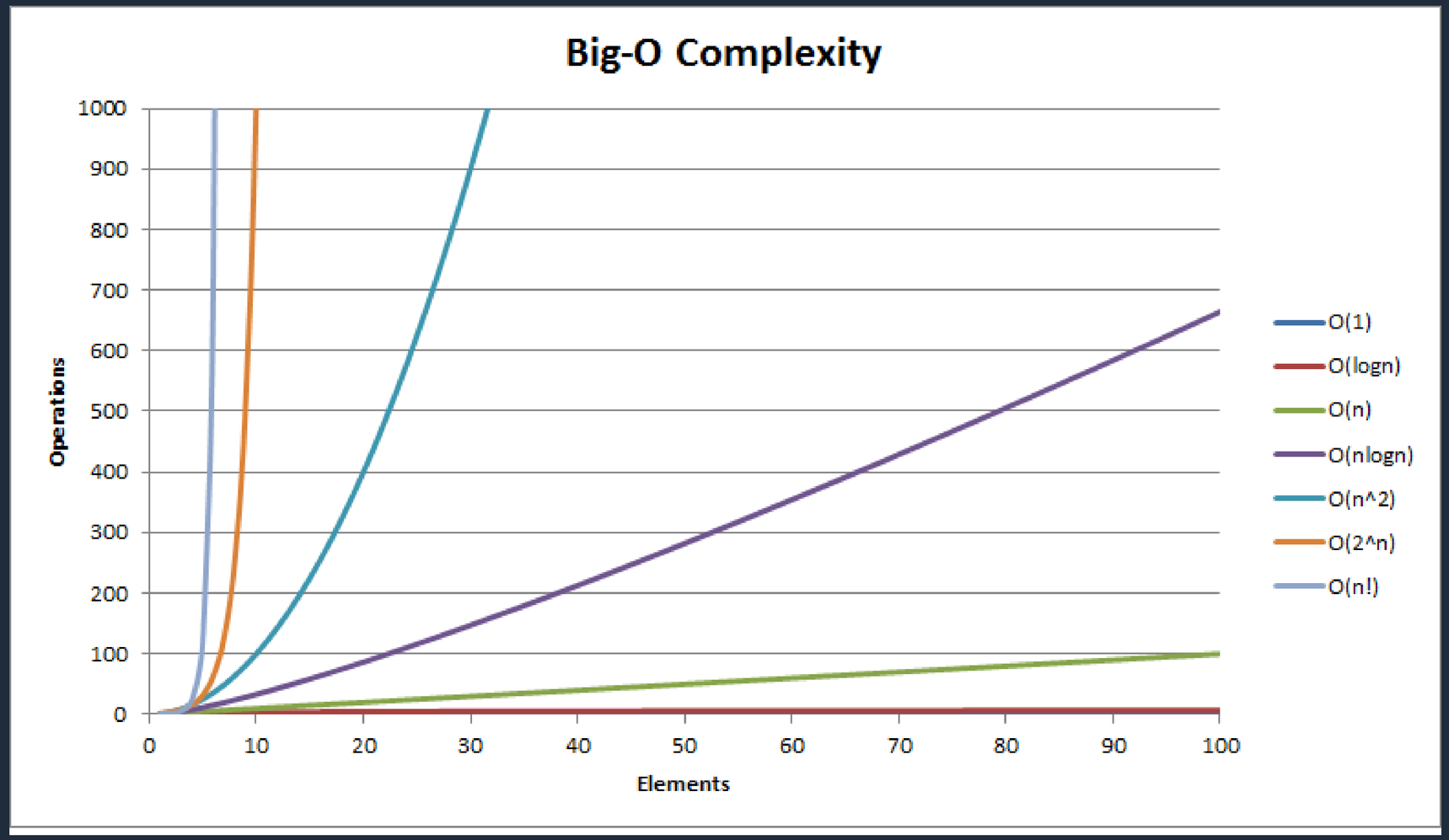
COMPLEXITY

PRESENTED BY: SABA, SAM, AND WILLIAM

WHAT IS TIME COMPLEXITY?

Simply put, determining the running time of an algorithm

WHAT IS TIME COMPLEXITY?



What does it mean?

Definition: Algorithmic complexity is concerned about how fast or slow a particular algorithm performs. We define complexity as a numerical function T(n) - time versus the input size n.

WHAT IS TIME COMPLEXITY? (SOME DEFINITIONS)

Formally there are three definitions for time complexity, O(big oh), Θ(big theta), Ω(big omega), each describing a different trait

Lets say we can model the running time of an algorithm with a function, $F(n)\ where\ , T: N \rightarrow N$

We say that $F(n)\ \epsilon\ \mathrm{O}(G(n))$, when $\exists\ c > 0, \exists n_o > 0, s.t\ \forall n \geq n_0, F(n) \leq C * G(n)$

In more laymen terms, we say that $G(n)$ is the upper bound of $F(n)$ and that it will take at most $O(G(n))$ steps

WHAT IS TIME COMPLEXITY? (SOME DEFINITIONS) cont'd

We say that $F(n)\ \epsilon\ \Omega(G(n))$, when $\exists\ c > 0, \exists n_o > 0, s.t\ \forall n\ \geq\ n_0, F(n)\ \geq C * G(n)$

In more laymen terms, we say that $\mathrm{F}(n)$ is bounded below by $\mathrm{G}(n)$, and that it will take at least $\Omega(G(n))$ steps

We say that $F(n)\ \epsilon\Theta(G(n))$, when $\exists\ c, d\ > 0, \exists n_o > 0, s.t\ \forall n\ \geq\ n_0,\ C * G(n)\ \leq F(n)\ \leq C * G(n)$

In more laymen terms, we say that $\mathrm{F}(n)$ is tightly bounded by $\mathrm{G}(n)$, and that it will take time proportional to $\Theta(G(n))$

WHAT IS COMPLEXITY (PAST DEFINITIONS)

```
1   print("Complexity is easy")
```

How complex is this code?

What about this?

```
1   print("Complexity is easy")
2   print("Complexity is still easy")
3   print("Complexity is incredibly easy")
```

WHAT IS COMPLEXITY (PAST DEFINITIONS)

What does it even mean?

Complexity is always seen in the sense of input size. So a program running in O(1) would not change in how long it takes for the code to run no matter what input size you give the program.

So any operation that does not take in the size of an input has O(1)

In general, big O can be seen as the “best worse case scenario”.

…MORE COMPLEX COMPLEXITY

So given what we just gave you,
what would be an example of O(n)?

WHAT IS COMPLEXITY?

```
1  def bigOn(n):
2
3      for(i = 0; i < n; i++):
4          print("ezpz complexity")
5
6      return(i)
```

O(n)

This is a very simple albit useless example of O(n). As the input sizee moves up, the program takes increasingly longer to run. In other words, the best worst case run time is n.

WHAT IS COMPLEXITY?

```
def [redacted](n, multiDimArr):
    for(i = 0; i < multiDimArr.length; i++):
        for(j = 0; j < multiDimArr[i].length; j++):
            print(multiDimArr[i][j])

    return(i)
```

So what time complexity do you think this function has? O(?)

WHAT IS COMPLEXITY?

```
def bigOn2(n, multiDimArr):
    for(i = 0; i < multiDimArr.length; i++):
        for(j = 0; j < multiDimArr[i].length; j++):
            print(multiDimArr[i][j])

    return(i)
```

$O(n^2)$

This is a another simple algorithm that takes a 2D array and prints out each element in the matrix. This is done by looping through, first the rows than another loop for each individual column of a certain row. So if one loop is ‘n’ steps, than two loops, yes you guessed it is n^2 steps.

WHAT IS COMPLEXITY?

```
def [redacted](n, multiDimArr):
    for(i = 0; i < multiDimArr.length; i++):
        print("Row number: " + str(i))
    for(j = 0; j < multiDimArr.length[0]; j++):
        print("Column number: " + str(i))
    return(i)
```

So what time complexity do you think this function has? O(?)

WHAT IS COMPLEXITY?

```
def bigOn2(n, multiDimArr):
    for(i = 0; i < multiDimArr.length; i++):
        print("Row number: " + str(i))
    for(j = 0; j < multiDimArr.length[0]; j++):
        print("Column number: " + str(i))
    return(i)
```

So what time complexity do you think this program has? O(?)

Ans: O(n)

This is because, this program does not contain a nested loop but just two for loops. So in terms of complexity if we analyze each step we have O(n+n). However since big O is about relative complexity not absolute, we can define O(n+n) ⇔ O(n) .

WHAT IS COMPLEXITY?

```
def search(root, word, depth=1):
    if not root:
        return 0, 0
    else if root.value == word:
        return depth, root.count
    else if word < root.value:
        return search(root.left, word, depth + 1)
    else:
        return search(root.right, word, depth + 1)
```

So what time complexity do you think this function has? O(?)

WHAT IS COMPLEXITY?

```
def bigOlogn(root, word, depth=1):
    if not root:
        return 0, 0
    else if root.value == word:
        return depth, root.count
    else if word < root.value:
        return bigOlogn(root.left, word, depth + 1)
    else:
        return bigOlogn(root.right, word, depth + 1)
```

O(log(n))

Can someone explain why this algorithm for a binary tree, has time complexity of O(log(n))?

WHAT IS COMPLEXITY

```
def bigOlogn(root, word, depth=1):
    if not root:
        return 0, 0
    else if root.value == word:
        return depth, root.count
    else if word < root.value:
        return bigOlogn(root.left, word, depth + 1)
    else:
        return bigOlogn(root.right, word, depth + 1)
```

O(log(n))

The reason this algorithm is O(log(n)), is because of the base cases of this recursive search function. So in this case this algorithm is a search algorithm for a BST. By definition of a binary tree data structure, the right most leaf of the tree contains the greatest value and the left most leaf of the tree contains the smallest value. Thus if we are searching for a element let's call it "word", then by comparing the value of the element from the root element of the tree, we can avoid searching half of the tree in the first step. When we hit the recursive step it continues to repeat this pattern(ignoring the side that does not satisfy the condition) as it recurses down the tree. Once the condition is met the element position is returned.

BREAK TIME

BREATHE AND RELAX!

Walkthrough : 1

Merge Sort

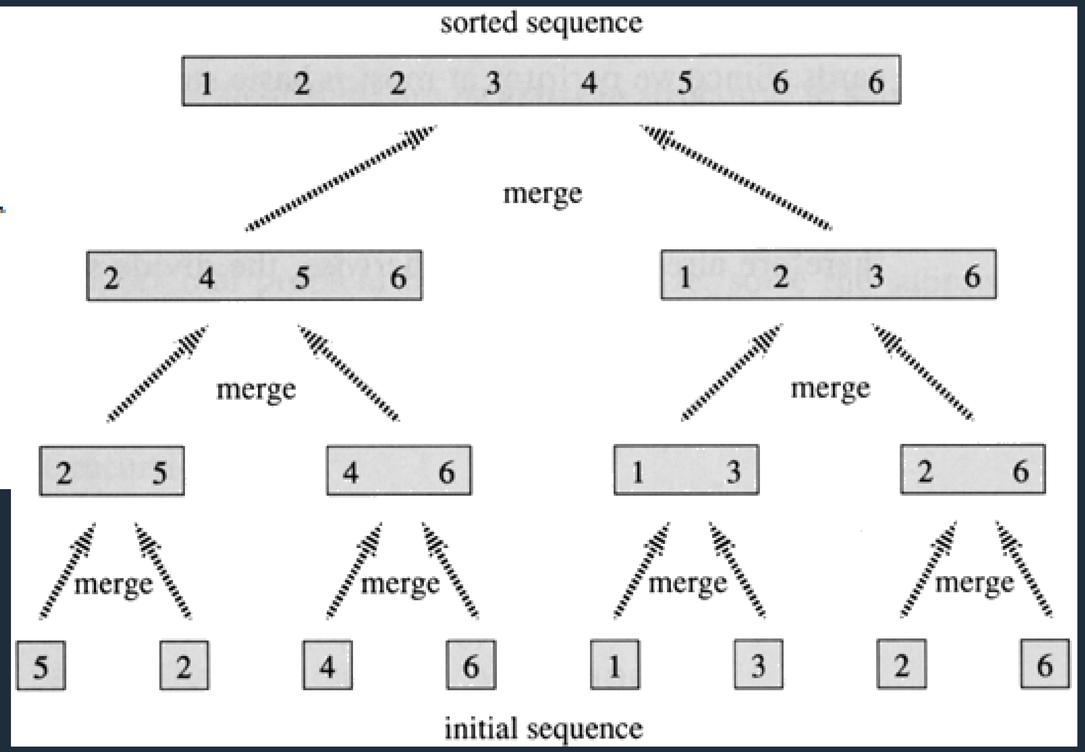
Merge Sort

So what is Merge Sort?

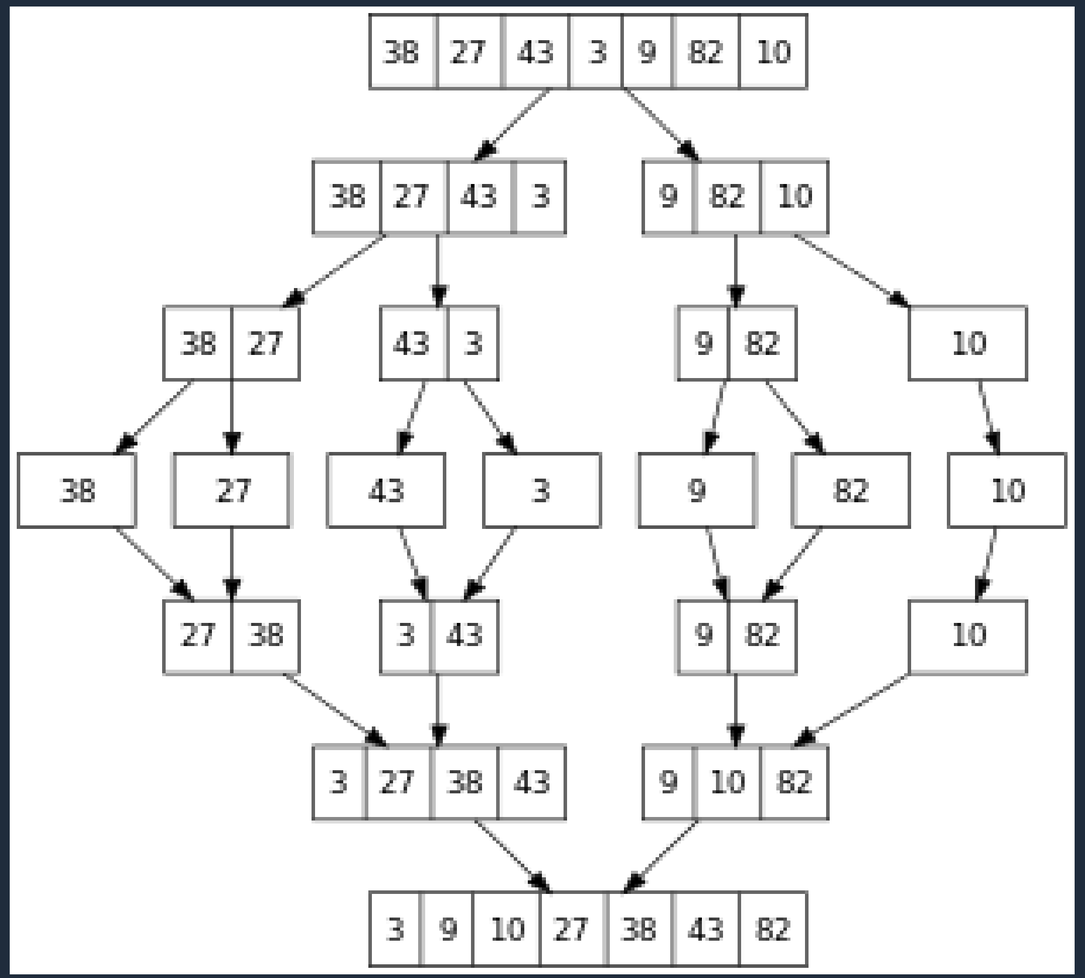
Merge sort is a sort algorithm for rearranging lists (or any other data structure that can be accessed sequentially, e.g. file streams) into a specified order. It is a particularly good example of the divide and conquer algorithmic paradigm. Merge sort has an average and worst-case performance of O(n log(n)).

Merge Sort

```
def MergeSort (Array):
    if len(Array) == 1:
        return Array
    else:
        Middle= ((Last + First)/2) //rounded down to the nearest integer
        LeftHalfArray = MergeSort(Array(First..Middle))
        RightHalfArray = MergeSort(Array(Middle+1..Last))
        ResultArray = Merge(LeftHalfArray, RightHalfArray)
        return ResultArray
```



Merge Sort



So what is really happening?

-First the midpoint of the array is computed using integer division taking O(1) time.

-Then the array is divided into sub arrays of size $\frac{n}{2}$ for each n element until each sub array has only 1 element, then the sub arrays are sorted using a comparator. O(log(n)) time.

-Finally all the sub arrays are merged into one array which takes O(n) time.

```
def MergeSort (Array):
    if len(Array) == 1:
        return Array
    else:
        Middle= ((Last + First)/2) //rounded down to the nearest inte
        LeftHalfArray = MergeSort(Array(First..Middle))
        RightHalfArray = MergeSort(Array(Middle+1..Last))
        ResultArray = Merge(LeftHalfArray, RightHalfArray)
        return ResultArray
```

Merge Sort

Why is it O(n*log(n))?

Because the basic comparator takes log(n) steps but does it for size n sublists it has a time complexity of O(n*log(n))

Walkthrough : 2

Radix Sort

Radix Sort (LSD)

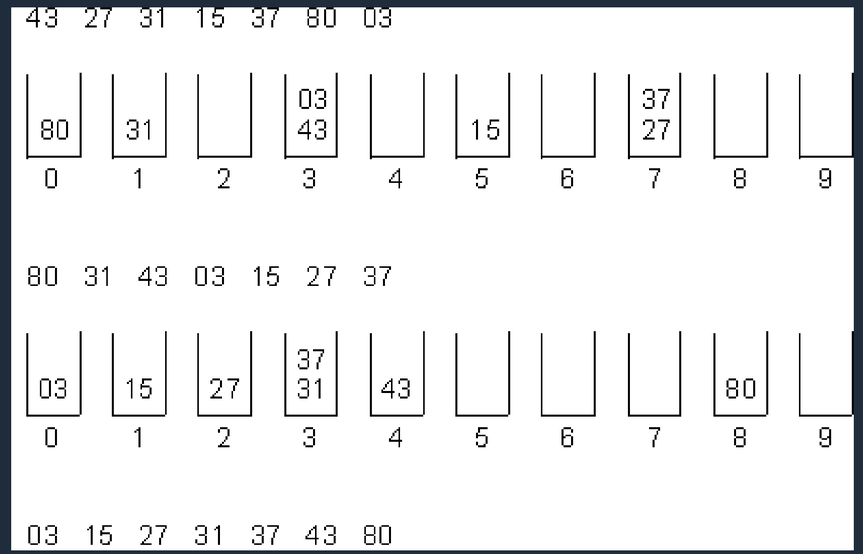
So what is Radix Sort?

Definition: A multiple pass distribution sort algorithm that distributes each item to a bucket according to part of the item's key beginning with the least significant part of the key. After each pass, items are collected from the buckets, keeping the items in order, then redistributed according to the next most significant part of the key.

This is interesting because radix sort is not a comparison based sort but instead is a distributive based sort which utilizes the special property of integers.

Radix Sort (LSD)

Unsorted	Sorted by 1s	Sorted by 10s	Sorted by 100s
123	123	123	123
583	583	625	154
154	154	154	456
567	625	456	567
689	456	567	583
625	567	583	625
456	689	689	689



So how does it work?

1. It works by distributing the input array into buckets by passing through the individual digits of a given number, one-by-one beginning with the least significant digit(LSD). Here, the number of buckets are a total of ten, which is why key values starts from 0 to 9.
2. After each pass, the numbers are collected from the buckets, and pushed into the collection again.
3. This recursively redistributes the numbers as in the above step '1' but keeping in mind the most significant digit (MSD) and step ‘2’.

Radix Sort (LSD)

radixsort(A, n) {
  for(i=0;i<k;i++) {
  for(j=0;j<s_i;j++) bin[j] = EMPTY;
	$O(s_i)$

for(j=0;j<n;j++) {
  move A_i
  to the end of bin[A_i->f_i]
  }
	$O(n)$

for(j=0;j<s_i;j++)
  concatenate bin[j] onto the end of A;
  }
}
	$O(s_i)$

Total	$\sum_{i=1}^{k} O(s_i + n) = O(kn + \sum_{i=1}^{k} s_i) = O(n + \sum_{i=1}^{k} s_i)$

So what time complexity does radix sort have?

Radix sort has a time complexity of O(nk) with k being the number of bins being used and n being the number of elements in the array. However in the general case since we are sorting integers we only have 0-9 bins, so as we discussed in slide 13, Big-O is about relative complexity not absolute complexity, thus we can generalize O(nk) ⇔ O(n)