WEEK-07 DAY-3
Wednesday - *Sorting Algorithms*

Sorting Algorithms

The objective of this lesson is for you to get experience implementing common sorting algorithms that will come up during a lot of interviews. It is also important for you to understand how different sorting algorithms behave when given output.

At the end of this, you will be able to

1. Explain the complexity of and write a function that performs `bubble sort` on an array of numbers.
2. Explain the complexity of and write a function that performs `selection sort` on an array of numbers.
3. Explain the complexity of and write a function that performs `insertion sort` on an array of numbers.
4. Explain the complexity of and write a function that performs `merge sort` on an array of numbers.
5. Explain the complexity of and write a function that performs `quick sort` on an array of numbers.
6. Explain the complexity of and write a function that performs a binary search on a sorted array of numbers.

Bubble Sort

Bubble Sort is generally the first major sorting algorithm to come up in most introductory programming courses. Learning about this algorithm is useful educationally, as it provides a good introduction to the challenges you face when tasked with converting unsorted data into sorted data, such as conducting logical comparisons, making swaps while iterating, and making optimizations. It's also quite simple to implement, and can be done quickly.

Bubble Sort is *almost never* a good choice in production. simply because:

- It is not efficient
- It is not commonly used
- There is a stigma attached to using it

"But...then...why are we..."

It is *quite useful* as an educational base for you, and as a conversational base for you while interviewing, because you can discuss how other more elegant and efficient algorithms improve upon it. Taking naive code and improving upon it by weighing the technical tradeoffs of your other options is 100% the name of the game when trying to level yourself up from a junior engineer to a senior engineer.

The algorithm bubbles up

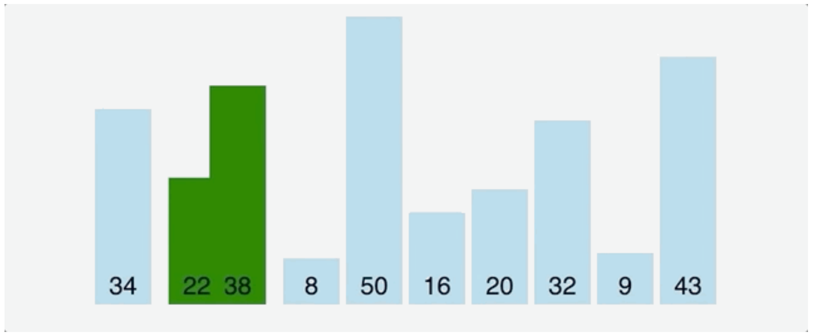
As you progress through the algorithms and data structures of this course, you'll eventually notice that there are some recurring funny terms. "Bubbling up" is one of those terms.

When someone writes that an item in a collection "bubbles up," you should infer that:

- The item is *in motion*
- The item is moving *in some direction*
- The item *has some final resting destination*

When invoking Bubble Sort to sort an array of integers in ascending order, the largest integers will "bubble up" to the "top" (the end) of the array, one at a time.

The largest values are captured, put into motion in the direction defined by the desired sort (ascending right now), and traverse the array until they arrive at their end destination. See if you can observe this behavior in the following animation (courtesy http://visualgo.net):



As the algorithm iterates through the array, it compares each element to the element's right neighbor. If the current element is larger than its neighbor, the algorithm swaps them. This continues until all elements in the array are sorted.

How does a pass of Bubble Sort work?

Bubble sort works by performing multiple *passes* to move elements closer to their final positions. A single pass will iterate through the entire array once.

A pass works by scanning the array from left to right, two elements at a time, and checking if they are ordered correctly. To be ordered correctly the first element must be less than or equal to the second. If the two elements are not ordered properly, then we swap them to correct their order. Afterwards, it scans the next two numbers and continue repeat this process until we have gone through the entire array.

See one pass of bubble sort on the array `[2, 8, 5, 2, 6]`. On each step the elements currently being scanned are in **bold**.

- [**2**, **8**, 5, 2, 6] - ordered, so leave them alone
- [2, **8**, **5**, 2, 6] - not ordered, so swap
- [2, 5, **8**, **2**, 6] - not ordered, so swap
- [2, 5, 2, **8**, **6**] - not ordered, so swap
- [2, 5, 2, 6, 8] - the first pass is complete

Because at least one swap occurred, the algorithm knows that it wasn't sorted. It needs to make another pass. It starts over again at the first entry and goes to the next-to-last entry doing the comparisons, again. It only needs to go to the next-to-last entry because the previous "bubbling" put the largest entry in the last position.

- [**2**, **5**, 2, 6, 8] - ordered, so leave them alone
- [2, **5**, **2**, 6, 8] - not ordered, so swap
- [2, 2, **5**, **6**, 8] - ordered, so leave them alone
- [2, 2, 5, 6, 8] - the second pass is complete

Because at least one swap occurred, the algorithm knows that it wasn't sorted. Now, it can bubble from the first position to the last-2 position because the last two values are sorted.

- [**2**, **2**, 5, 6, 8] - ordered, so leave them alone
- [2, **2**, **5**, 6, 8] - ordered, so leave them alone
- [2, 2, 5, 6, 8] - the third pass is complete

No swap occurred, so the Bubble Sort stops.

Ending the Bubble Sort

During Bubble Sort, you can tell if the array is in sorted order by checking if a swap was made during the previous pass performed. If a swap was not performed during the previous pass, then the array must be totally sorted and the algorithm can stop.

You're probably wondering why that makes sense. Recall that a pass of Bubble Sort checks if any adjacent elements are **out of order** and swaps them if they are. If we don't make any swaps during a pass, then everything must be already **in order**, so our job is done. Let that marinate for a bit.

Pseudocode for Bubble Sort

```
Bubble Sort: (array)
  n := length(array)
  repeat
  swapped = false
  for i := 1 to n - 1 inclusive do

      /* if this pair is out of order */
      if array[i - 1] > array[i] then

        /* swap them and remember something changed */
        swap(array, i - 1, i)
        swapped := true

      end if
    end for
  until not swapped
```

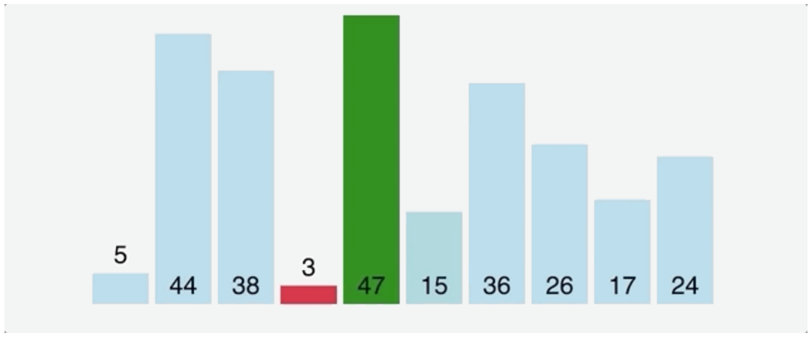
Selection Sort

Selection Sort is very similar to Bubble Sort. The major difference between the two is that Bubble Sort bubbles the *largest* elements up to the end of the array, while Selection Sort selects the *smallest* elements of the array and directly places them at the beginning of the array in sorted position.

Selection sort will utilize swapping just as bubble sort did. Let's carefully break this sorting algorithm down.

The algorithm: select the next smallest

Selection sort works by maintaining a sorted region on the left side of the input array; this sorted region will grow by one element with every "pass" of the algorithm. A single "pass" of selection sort will select the next smallest element of unsorted region of the array and move it to the sorted region. Because a single pass of selection sort will move an element of the unsorted region into the sorted region, this means a single pass will shrink the unsorted region by 1 element whilst increasing the sorted region by 1 element. Selection sort is complete when the sorted region spans the entire array and the unsorted region is empty!



The algorithm can be summarized as the following:

1. Set MIN to location 0
2. Search the minimum element in the list
3. Swap with value at location MIN
4. Increment MIN to point to next element
5. Repeat until list is sorted

The pseudocode

In pseudocode, the Selection Sort can be written as this.

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for
end procedure
```

Insertion Sort

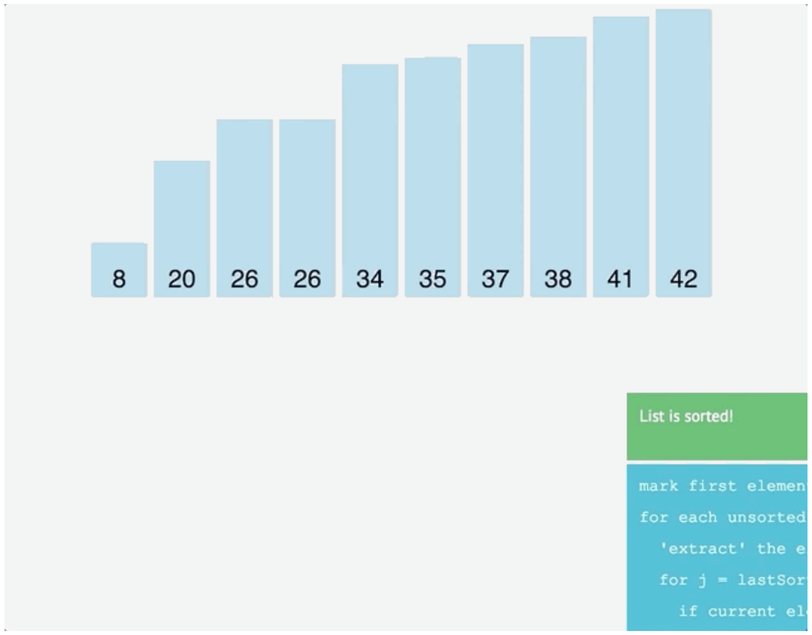
With Bubble Sort and Selection Sort now in your tool box, you're starting to get some experience points under your belt! Time to learn one more "naive" sorting algorithm before you get to the efficient sorting algorithms.

The algorithm: insert into the sorted region

Insertion Sort is similar to Selection Sort in that it gradually builds up a larger and larger sorted region at the left-most end of the array.

However, Insertion Sort differs from Selection Sort because this algorithm does not focus on searching for the right element to place (the next smallest in our Selection Sort) on each pass through the array. Instead, it focuses on sorting each element in the order they appear from left to right, regardless of their value, and inserting them in the most appropriate position in the sorted region.

See if you can observe the behavior described above in the following animation:



```
mark first elemen
for each unsorted
  'extract' the e
  for j = lastSor
    if current el
```

The Steps

Insertion Sort grows a sorted array on the left side of the input array by:

1. If it is the first element, it is already sorted. return 1;
2. Pick next element
3. Compare with all elements in the sorted sub-list
4. Shift all the elements in the sorted sub-list that is greater than the value to be sorted
5. Insert the value
6. Repeat until list is sorted

These steps are easy to confuse with selection sort, so you'll want to watch the video lecture and drawing that accompanies this reading as always!

The pseudocode

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while

      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

Merge Sort

You've explored a few sorting algorithms already, all of them being quite slow with a runtime of O(n^2). It's time to level up and learn your first time-efficient sorting algorithm! You'll explore **merge sort** in detail soon, but first, you should jot down some key ideas for now. The following points are not steps to an algorithm yet; rather, they are ideas that will motivate how you can derive this algorithm.

- it is easy to merge elements of two sorted arrays into a single sorted array
- you can consider an array containing only a single element as already trivially sorted
- you can also consider an empty array as trivially sorted

The algorithm: divide and conquer

You're going to need a helper function that solves the first major point from above. How might you merge two sorted arrays? In other words you want a `merge` function that will behave like so:

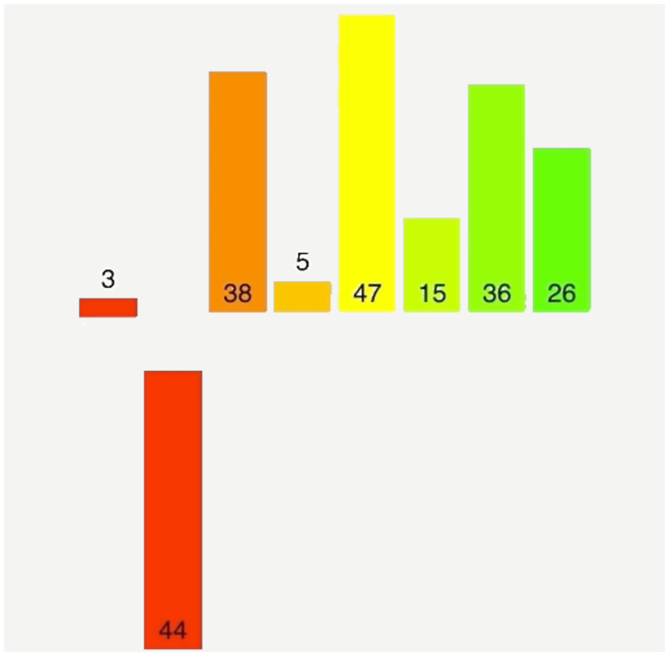
```
let arr1 = [1, 5, 10, 15];
let arr2 = [0, 2, 3, 7, 10];
merge(arr1, arr2); // => [0, 1, 2, 3, 5, 7, 10, 10, 15]
```

Once you have that, you get to the "divide and conquer" bit.

The algorithm for merge sort is actually *really* simple.

1. if there is only one element in the list, it is already sorted. return that array.
2. otherwise, divide the list recursively into two halves until it can no more be divided.
3. merge the smaller lists into new list in sorted order.

The process is visualized below. When elements are moved to the bottom of the picture, they are going through the `merge` step:



The pseudocode for the algorithm is as follows.

```
procedure mergesort( a as array )
   if ( n == 1 ) return a

   /* Split the array into two */
   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( a as array, b as array )
   var result as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of result
         remove b[0] from b
      else
         add a[0] to the end of result
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of result
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of result
      remove b[0] from b
   end while

   return result
end procedure
```

Quick Sort

Quick Sort has a similar "divide and conquer" strategy to Merge Sort. Here are a few key ideas that will motivate the design:

- it is easy to sort elements of an array relative to a particular target value
- an array of 0 or 1 elements is already trivially sorted

Regarding that first point, for example given `[7, 3, 8, 9, 2]` and a target of 5, we know `[3, 2]` are numbers less than 5 and `[7, 8, 9]` are numbers greater than 5.

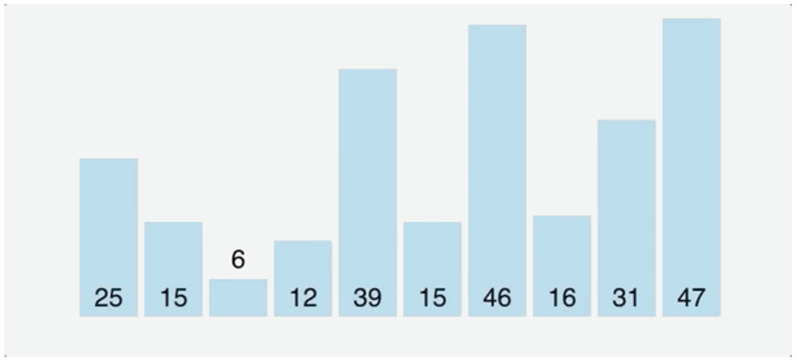
How does it work?

In general, the strategy is to divide the input array into two subarrays: one with the smaller elements, and one with the larger elements. Then, it recursively operates on the two new subarrays. It continues this process until of dividing into smaller arrays until it reaches subarrays of length 1 or smaller. As you have seen with Merge Sort, arrays of such length are automatically sorted.

The steps, when discussed on a high level, are simple:

1. choose an element called "the pivot", how that's done is up to the implementation
2. take two variables to point left and right of the list excluding pivot
3. left points to the low index
4. right points to the high
5. while value at left is less than pivot move right
6. while value at right is greater than pivot move left
7. if both step 5 and step 6 does not match swap left and right
8. if left ≥ right, the point where they met is new pivot
9. repeat, recursively calling this for smaller and smaller arrays

Before we move forward, see if you can observe the behavior described above in the following animation:



The algorithm: divide and conquer

Formally, we want to partition elements of an array relative to a pivot value. That is, we want elements less than the pivot to be separated from elements that are greater than or equal to the pivot. Our goal is to create a function with this behavior:

```javascript
let arr = [7, 3, 8, 9, 2];
partition(arr, 5);  // => [[3, 2], [7,8,9]]
```

Partition

Seems simple enough! Let's implement it in JavaScript:

```javascript
// nothing fancy
function partition(array, pivot) {
  let left = [];
  let right = [];

  array.forEach(el => {
    if (el < pivot) {
      left.push(el);
    } else {
      right.push(el);
    }
  });

  return [ left, right ];
}

// if you fancy
function partition(array, pivot) {
  let left = array.filter(el => el < pivot);
  let right = array.filter(el => el >= pivot);
  return [ left, right ];
}
```

You don't have to use an explicit `partition` helper function in your Quick Sort implementation; however, we will borrow heavily from this pattern. As you design algorithms, it helps to think about key patterns in isolation, although your solution may not feature that exact helper. Some would say we like to divide and conquer.

The pseudocode

It is *so* small, this algorithm. It's amazing that it performs so well with so little code!

```
procedure quickSort(left, right)

  if the length of the array is 0 or 1, return the array

  set the pivot to the first element of the array
  remove the first element of the array

  put all values less than the pivot value into an array called left
  put all values greater than the pivot value into an array called right

  call quick sort on left and assign the return value to leftSorted
  call quick sort on right and assign the return value to rightSorted

  return the concatenation of leftSorted, the pivot value, and rightSorted

end procedure
```

Binary Search

We've explored many ways to sort arrays so far, but why did we go through all of that trouble? By sorting elements of an array, we are organizing the data in a way that gives us a quick way to look up elements later on. For simplicity, we have been using arrays of numbers up until this point. However, these sorting concepts can be generalized to other data types. For example, it would be easy to modify our comparison-based sorting algorithms to sort strings: instead of leveraging facts like `0 < 1`, we can say `'A' < 'B'`.

Think of a dictionary. A dictionary contains alphabetically sorted words and their definitions. A dictionary is pretty much only useful if it is ordered in this way. Let's say you wanted to look up the definition of "stupendous." What steps might you take?

- you open up the dictionary at the roughly middle page
  - you land in the "m" section
- you know "s" comes somewhere after "m" in the book, so you disregard all pages before the "m" section. Instead, you flip to the roughly middle page between "m" and "z"
  - you land in the "u" section

- you know "s" comes somewhere before "u", so you can disregard all pages after the "u" section. Instead, you flip to the roughly middle page between the previous "m" page and "u"
- ...

You are essentially using the `binarySearch` algorithm in the real world.

The Algorithm: "check the middle and half the search space"

Formally, our `binarySearch` will seek to solve the following problem:

```
Given a sorted array of numbers and a target num, return a boolean indicating w
```

Programmatically, we want to satisfy the following behavior:

```
binarySearch([5, 10, 12, 15, 20, 30, 70], 12);  // => true
binarySearch([5, 10, 12, 15, 20, 30, 70], 24);  // => false
```

Before we move on, really internalize the fact that `binarySearch` will only work on **sorted** arrays! Obviously we can search any array, sorted or unsorted, in `O(n)` time. But now our goal is be able to search the array with a sub-linear time complexity (less than `O(n)`).

The pseudocode

```
procedure binary search (list, target)
  parameter list: a list of sorted value
  parameter target: the value to search for

  if the list has zero length, then return false

  determine the slice point:
    if the list has an even number of elements,
      the slice point is the number of elements
      divided by two
    if the list has an odd number of elements,
      the slice point is the number of elements
      minus one divided by two

  create an list of the elements from 0 to the
    slice point, not including the slice point,
    which is known as the "left half"
  create an list of the elements from the
    slice point to the end of the list which is
    known as the "right half"

  if the target is less than the value in the
    original array at the slice point, then
    return the binary search of the "left half"
    and the target
  if the target is greater than the value in the
    original array at the slice point, then
    return the binary search of the "right half"
    and the target
  if neither of those is true, return true
end procedure binary search
```