**Sorting Algorithms**

Visual Representation Link: https://visualgo.net/en/sorting

**Sorting Algorithm 1: Bubble Sort:**

**public** **static** **void** bubbleSort(**int**[] arr){

**int** swapCount = Integer.*MAX\_VALUE*;

**while**(swapCount > 0){

swapCount = 0;

**for**(**int** i = 0; i < arr.length - 1; i++){

**if**(arr[i] > arr[i+1]){

*swap*(arr, i , i + 1);

swapCount++;

}

}

}

}

**Higher Level Algorithm:** Look at the link and see the visual representation/ animation of bubble sort first. Now suppose you have an array with contents {5,4,3,2,1}. You iterate around the array and swap the current item with the next item if they are out of order. One complete iteration is called one pass. So the first pass would look like {5,4,3,2,1}, {4,5,3,2,1}, {4,3,5,2,1}, {4,3,2,5,1}, {4,3,2,1,5}. You keep making passes until you make a pass where there wasn’t a single swap which means the entire array was already sorted.

**Bubble Sort Execution Trace:**

Original Array: 5 4 3 2 1

New Pass:

i = 0 Swap Made?:Yes Array: 4 5 3 2 1

i = 1 Swap Made?:Yes Array: 4 3 5 2 1

i = 2 Swap Made?:Yes Array: 4 3 2 5 1

i = 3 Swap Made?:Yes Array: 4 3 2 1 5

New Pass:

i = 0 Swap Made?:Yes Array: 3 4 2 1 5

i = 1 Swap Made?:Yes Array: 3 2 4 1 5

i = 2 Swap Made?:Yes Array: 3 2 1 4 5

i = 3 Swap Made?:No Array: 3 2 1 4 5

New Pass:

i = 0 Swap Made?:Yes Array: 2 3 1 4 5

i = 1 Swap Made?:Yes Array: 2 1 3 4 5

i = 2 Swap Made?:No Array: 2 1 3 4 5

i = 3 Swap Made?:No Array: 2 1 3 4 5

New Pass:

i = 0 Swap Made?:Yes Array: 1 2 3 4 5

i = 1 Swap Made?:No Array: 1 2 3 4 5

i = 2 Swap Made?:No Array: 1 2 3 4 5

i = 3 Swap Made?:No Array: 1 2 3 4 5

New Pass:

i = 0 Swap Made?:No Array: 1 2 3 4 5

i = 1 Swap Made?:No Array: 1 2 3 4 5

i = 2 Swap Made?:No Array: 1 2 3 4 5

i = 3 Swap Made?:No Array: 1 2 3 4 5

Sorted Array: 5 4 3 2 1

**Time Complexity Analysis:** The first thing you need to take into account is that each pass contains n -1 iterations. The reason we have n-1 iterations is because we swap the current item with the next item so we can’t go until the end of the list. You can see this in the for-loop condition. In our example, our array has a size equal to 5. This means that there are four iterations per pass.

The total number of execution lines is ( number of passes) \* ( number of iterations per pass) \* (number of lines of code per iteration, which is constant so lets call it C). Since (number of iterations per pass) = n -1, then the Big Oh simplifies to (C) \* ( n -1 ) \* ( number of passes).

The second critical part you need to understand is that there will be a worst case n-passes. If you look at our example above, the first pass puts the 5 in its correct position. The second pass puts the 4 in its correct position. The third pass puts the three in its correct position. The 4th pass puts the 2 in its correct position. Since the 2,3,4,5 are in their correct position, the one must be in its correct position. This saves you a step but you must use a step to verify the array is sorted (no swaps made). So for an array of n items, you make a worst case of (n-1) passes to sort the second through last items, and then one pass to verify which yields n passes.

This yields a Big Oh of worst case (C lines oer iteration) \* (n-1 iterations per pass) \* ( n passes) = C \* (n2 – n ) = n2.

The best case scenerio is that the array is already sorted so you only need one pass which means O(n). The thing that varies is the number of passes which can range from 1 – 5 in this case or in general terms 1 – n. Let’s say on average, it is halfway in between 1 – n, so it is ( 1 + n )/ 2. This means that the average case is [( n + 1 )/2] \* (n) which is still O(n2).

**Space Complexity Analysis:** There is really only one temporary variable in the swap and one variable to keep track of the swap count. This is constant so it requires an additional O(1) space. Whether it is best case, average case, or worse case doesn’t matter since in each iteration, your re-using the same two variables and not introducing any new ones.

Summary: Fill in table before sending to google drive.

**Sorting Algorithm 1: Insertion Sort:**

**public** **static** **void** insertionSort(**int** array[]){

**for** (**int** i=1; i< array.length; i++){

**int** currentIndex = i;

**int** previousIndex = i -1;

**while** ((previousIndex >=0) && (array[previousIndex] >

array[currentIndex])){

*swap*(array, previousIndex, currentIndex);

currentIndex--;

previousIndex--;

}

}

}

**Higher Level Algorithm:**

Look at the link and see the visual representation/ animation of insertion sort first. Let’s define some terminology to make explaining things easier. With the second to last elements in the array, you continously swap it with the previous element while “the previous element is larger than it” and “you don’t reach the front list”. Let’s call this continuous swapping a “re-position” of an element in the array. So what insertion sort does is it re-positions the second to last elements in the array which results in the array being sorted. All the lines in the for loop really just “re-position “ element array[i]. If you see the animation, you’ll know what a re-position is and why it re-positions the second to last element in the array and not the first element.

Now let me explain the variables in the implementation above. There are three introduced variables which are current index, previous index, and the i-pointer. The i-pointer is used just to iterate around the array from the second element to the last element. The current and previous indexes are used for re-positions. In each re-position, the current index is initialized to the i-pointer and the previous index is initialized to index before it. Both these indexes are used to make comparisons and swaps, and they both decrement backwards until the re-position is complete. So while current index moves backward to re-position an item with continuous swaps, the i-pointer remains untouched so we don’t lose track of which part of the array we iterated through since current index gets decremented.

**Insertion Sort Execution Trace:**

Original Array: 5 4 3 2 1

New Re-position: Item 1

i-pointer:1 Swap Made? Yes Array: 4 5 3 2 1

i-pointer:1 Swap Made? No Array: 4 5 3 2 1

New Re-position: Item 2

i-pointer:2 Swap Made? Yes Array: 4 3 5 2 1

i-pointer:2 Swap Made? Yes Array: 3 4 5 2 1

i-pointer:2 Swap Made? No Array: 3 4 5 2 1

New Re-position: Item 3

i-pointer:3 Swap Made? Yes Array: 3 4 2 5 1

i-pointer:3 Swap Made? Yes Array: 3 2 4 5 1

i-pointer:3 Swap Made? Yes Array: 2 3 4 5 1

i-pointer:3 Swap Made? No Array: 2 3 4 5 1

New Re-position: Item 4

i-pointer:4 Swap Made? Yes Array: 2 3 4 1 5

i-pointer:4 Swap Made? Yes Array: 2 3 1 4 5

i-pointer:4 Swap Made? Yes Array: 2 1 3 4 5

i-pointer:4 Swap Made? Yes Array: 1 2 3 4 5

i-pointer:4 Swap Made? No Array: 1 2 3 4 5

Sorted Array: 1 2 3 4 5

**Notes:** Notice how the i-pointer always points to the current element being re-positioned and starts at 1 instead of zero. Also notice that the re-position always ends when a swap is not made. This happens when it fails the while loop condition.

**Big-Oh Time Complexity Analysis:** The total number of steps is ( n - 1 elements to re-position ) \* (number of swaps per re-position) \* ( number of lines per swap + the two lines to decrement the indexes). Since the last term in the product is constant (let’s call it C) then the time complexity analysis is Big Oh = (n – 1) \* ( number of swaps per re-position) \* C.

Now let’s analyze the number of swaps per re-position. This can vary from 1 – (n). For example, imagine the array with contents {1,2,3,4,5} and we are sorting this array. This is the best case scenario since it is already sorted. Now suppose we want to re-position the two. We just make one comparison with it and the 1 (in the while loop condition) and we already exit the loop and no the re-position is over. That is best case scenario and it is still 1 line/step because of the comparision. Now imagine you have the array {2,3,4,5,1} and you have to re-position the one. In this case, it goes from the last element in the array to the first element in the array which means it travel through the array backwards. So in the worst case, there are n-swaps per reposition. Let’s say the average is (1 + n)/2. This yields

Worst Case = ( n – 1 re-positions) \* ( n swaps per repositions ) \* ( C lines per swap) = O(n2).

Best Case = (n -1 repositions) \* ( 1 that is not a swap but a check for a swap) \* (C lines per swap ) = O(n).

Average Case = ( n – 1 re-positions) \* ( (1 + n) /2 swaps per repositions) \* (C lines per swap) = O(n2).

**Space Complexity Analysis:** There is really only one temporary variable in the swap and two variable to keep track of the current and previous elements in the re-positions. This is constant so it requires an additional O(1) space. Whether it is best case, average case, or worse case doesn’t matter since in each iteration, your re-using the same two variables and not introducing any new ones.

Summary: Fill in table before sending to google drive.

**Sorting Algorithm 3: Selection Sort**

**public** **static** **void** selectionSort(**int**[] array){

**for** (**int** i = 0; i < array.length-1; i++){

**int** minimumIndex = *getMinimumIndex*(array, i);

*swap*(array, minimumIndex, i);

}

}

**public** **static** **int** getMinimumIndex(**int**[] array, **int** first){

**int** minimumIndex = first;

**for** (**int** i = first + 1; i < array.length; i++){

**if** (array[i] < array[minimumIndex])

minimumIndex = i;

}

**return** minimumIndex;

}

**High Level Algorithm Explanation:**

If you look through the visual representation in the link, you should have a good idea about how selection sort works. You have an outer for-loop that just iterates around the array from the first index to the second last index. For each of these iterations, you find the minimum item in the rest of the array (the items after the current index). Once you find this minimum index, you swap it with the current item. The key to understanding this algorithm is understanding what the “get minimum index” method does. This method takes in an array and index, and finds the minimum item at that index and after. So for example, if you pass in the array {1,2,3,4,5} and pass in 2 as the index, then 3 would be returned since it’s the minimum index at or after the index 2. In the visual representation in the link, the red symbolizes the minimum index. You find the minimum index at or after the current index. Then you perform the swap.

**Selection Sort Execution Trace:**

Original Array: 5 4 3 2 1

--------------------------- New Iteration: i = 0 -------------------------

Step 1: Get Minimum Index starting from index i = 0

Current Item: array [1] = 4 Current Minimum: 5 New Minimum?: Yes

Current Item: array [2] = 3 Current Minimum: 4 New Minimum?: Yes

Current Item: array [3] = 2 Current Minimum: 3 New Minimum?: Yes

Current Item: array [4] = 1 Current Minimum: 2 New Minimum?: Yes

Completed finding the minimum which was 1

Step 2: Swap the 5 with the minimum 1 Updated Array: 1 4 3 2 5

--------------------------- New Iteration: i = 1 ---------------------------

Step 1: Get Minimum Index starting from index i = 1

Current Item: array [2] = 3 Current Minimum: 4 New Minimum?: Yes

Current Item: array [3] = 2 Current Minimum: 3 New Minimum?: Yes

Current Item: array [4] = 5 Current Minimum: 2 New Minimum?: No

Completed finding the minimum which was 2

Step 2: Swap the 4 with the minimum 2 Updated Array: 1 2 3 4 5

--------------------------- New Iteration: i = 2 ---------------------------

Step 1: Get Minimum Index starting from index i = 2

Current Item: array [3] = 4 Current Minimum: 3 New Minimum?: No

Current Item: array [4] = 5 Current Minimum: 3 New Minimum?: No

Completed finding the minimum which was 3

Step 2: Swap the 3 with the minimum 3 Updated Array: 1 2 3 4 5

--------------------------- New Iteration: i = 3 ---------------------------

Step 1: Get Minimum Index starting from index i = 3

Current Item: array [4] = 5 Current Minimum: 4 New Minimum?: No

Completed finding the minimum which was 4

Step 2: Swap the 4 with the minimum 4 Updated Array: 1 2 3 4 5

-------------------------------- Complete --------------------------------

Sorted Array: 5 4 3 2 1

**Notes:** So there are ( n – 1) iterations because we can’t check the minimum element of the rest of the array for the last element. The for-loop goes from the first element to the second last element. At each iteration, you check for the minimum element in the rest of the array and you perform a swap (Even if the current element is the minimum, you do a useless swap). As you get to the end of the array, the “rest of the array” area gets smaller so getting the minimum index goes by faster.

**Big Oh Time Comlexity Analysis:** So for the outer for loop, you have ( n -1 ) iterations which represents the first element to the second last element. For each of these, you do one call of get minimum index, and then you do one call of swap. So the

Big Oh = ( n – 1 iterations) \* ( 1 Swap per iteration ) \* ( 1 get minimum index per iteration). Since the swao is constant, then

Big Oh = ( n – 1 iterations) \* ( C ) \* ( 1 get minimum index per iteration).

Now let’s analyse the get minimum index function call. In the best case, you have an array that is already sorted. However, you still do all the same checks, you just get a lot of no’s in the new minimum column. The performance of this function ALWAYS starts at O(n-1) regardless of how well sorted the array already is and ALWAYS ends with O(1). It’s time gradually gets better, each iteration. So its performance varies between O(n-1) -> O(1) depending on which iteration it is. However, since Big Oh disregards constants and factors, the Big Oh for this function will be O(n).

Best Case = ( n -1 iterations) \* ( C steps per swap ) \* ( n for the get minimum index) = O (n2).

Worst Case = ( n -1 iterations) \* (C steps per swap) \* ( n for get minimum index) = O (n2).

Average Case = ( n – 1 iterations) \* ( C steps per swap) \* ( n for get minimum index) = O (n2).

**Space Comlexity Analysis:** The swap uses one temporary variable. The get minimum index uses one minimum index local variable. Those are it and they get re-used. So space complexity ids O(1).

**Sorting Algorithm 4: Merge Sort**

Merge sort is too complicated for you to be able to look at the visual representation or the code and understand it. We are going to have to eat an elephant one bite at a time.

**Part 1 - High Level Merge Method:** The first thing we want to grasp is the high level of the “merge” function in merge sort. Below is the method signature.

**public** **static** **void** merge(**int**[] array, **int**[] helper, **int** start, **int** firstSortedSubArrayEnd, **int** end) {}

This function takes an array that contains two sorted sub arrays, and sorts the array. For example, you look at the test for it below. The array I initialized is of size 8. That contains two sorted sub arrays {1, 3, 5, and 7} and {2, 4, 6, 8}. What this tells us is that we don’t need to use our traditional sorting algorithms. We can use an efficient method to sort it that makes use of the fact that we know that the left and right side is sorted, but the entire array is not sorted. We need to specify at which point does the first sorted sub array ends (which is three in this case). Finally we throw in a helper array which is empty that is used for temporary storage. The reason we throw it in the parameter and not initialize it in the method is because this method will be called a lot using recursion. We want the same storage to be passed around and not have a bunch of storage memory be initialized.

**public** **static** **void** testMerge() {

**int**[] array = **new** **int**[]{1,3,5,7,2,4,6,8};

**int** arraySize = array.length;

**int** start = 0;

**int** firstSortedSubArrayEnd = 3;

**int** end = array.length - 1;

*merge*(array, **new** **int**[arraySize], start, firstSortedSubArrayEnd, end);

System.*out*.println(Arrays.*toString*(array));//Prints sorted array

}

**Part 2 – Implementation of Merge Method:** There are four parts to the implementation of this merge method. All these four parts are fairly simple.

1. Copy the entire array into helper
2. Initialize three iterators you will be using. One iterator iterates over the left sub array in helper. The other iterates over the right sub array in helper. The third iterates over the entire array in the original array called array.

//Step 1: Copy array into helper. Method uses dest,src notation

*arrayCopy*(helper, array);

//Step 2: Initialize pointers

**int** HelperLeftIterator = start;

**int** helperRightIterator = firstSortedSubArrayEnd + 1;

**int** arrayIterator = start;

1. In order to correctly sort array, you need to iterate over both sub arrays and copy the elements in correct order into the original array. The while loop condition states while neither of the sub array pointers reached the end of their corresponding sub array.

/\* Step 3: Iterate through both helper sub arrays. Compare the left and right sub arrays and copy the smaller element from the two halves into the original array.\*/

**while**((HelperLeftIterator<= firstSortedSubArrayEnd) && (helperRightIterator<= end)){

**if** (helper[HelperLeftIterator] <= helper[helperRightIterator]) {

array[arrayIterator] = helper[HelperLeftIterator];

HelperLeftIterator++;

} **else** { // If right element is smaller than left element

array[arrayIterator] = helper[helperRightIterator];

helperRightIterator++;

}

arrayIterator++;

}

1. Now there is the possiblilty that the two sub arrays are different lengths. This means we completely iterator over one sub array put the other sub array still has some elements remaining. Since each sub array was originally sorted, we can just append this remaining amount in the helper array to the array. Note: If it is the right sub array, it should already be on the other end so only do the left one.

/\* Step 4: Copy remaining portion of helper into remaining of array.

**int** remaining = firstSortedSubArrayEnd - HelperLeftIterator;

**for** (**int** i = 0; i <= remaining; i++) {

array[arrayIterator + i] = helper[HelperLeftIterator + i];

}

**Binary Search:**

**public** **static** **int** binarySearchRecursive(**int**[] a, **int** x, **int** low, **int** high) {

**if** (low > high) **return** -1;

**int** mid = (low + high) / 2;

**if** (a[mid] < x) {

**return** *binarySearchRecursive*(a, x, mid + 1, high);

} **else** **if** (a[mid] > x) {

**return** *binarySearchRecursive*(a, x, low, mid - 1);

} **else** {

**return** mid;

}

}

**High Level Understanding:**

So for binary search, given a sorted integer array and an item ‘x’ that you are searching for, you either return the index in which you found this item or you return. Suppose you have the array {0,1,2,3,4,5,6,7,8} and you are search for the three. The first thing you do is check the value “a[mid] at the midpoint which in this example is (0 + 8 / 2 = 4) and a[4] = 4. So since 4 is greater than 3, you know it is somewhere in the first half of the array. So now you recurse and this time, you make your new end points the first half of the array, and the midpoint is 2. You keep doing this until you either find the item in which you return its location or your low > high. When your at the last element to check, your low index is you high index so after that, your low index becomes greater than your high index.

Time and Space Complexity:

Worst Case: The worst case is when you don’t find the item and return -1 and you end up searching the entire array. In every call of this function, you check if the low index is higher than the high index (constant), you compute the midpoint (constant) and your either done and return or call this function again recursively in the if else statement. So, when you double you data size, you add an extra step to your worst case scenerio. So this relationship is O(log n). Best case is you find it on your first try. Average case is about halfway between best and worst which is still O(logn).

**public** **static** **int** binarySearch(**int**[] a, **int** x) {

**int** low = 0;

**int** high = a.length - 1;

**int** mid;

**while** (low <= high) {

mid = (low + high) / 2;

**if** (a[mid] < x) {

low = mid + 1;

} **else** **if** (a[mid] > x) {

high = mid - 1;

} **else** {

**return** mid;

}

}

**return** -1;

}