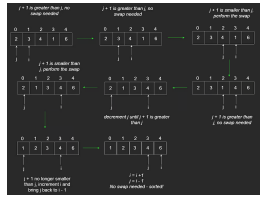
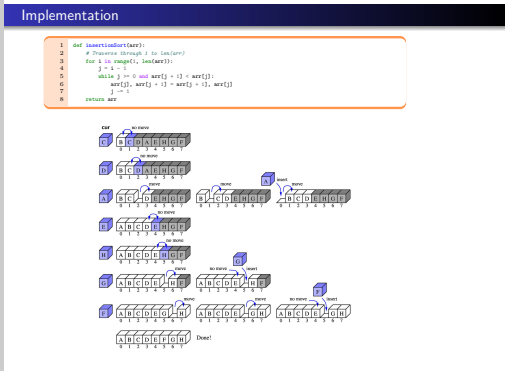
Concept

- Insertion sort works by dividing an array into sub-arrays and sorting them individually.
- Example: Consider an array of size 5: [2,3,4,1,6]. The goal is to sort this array.
  - The first element forms a sorted subarray of size 1.
  - In the next step, a subarray of size 2 ([2,3]) is sorted, and this process continues until the entire array is sorted.
- The process involves comparing and swapping elements using two pointers, i and j, where j is always behind i.

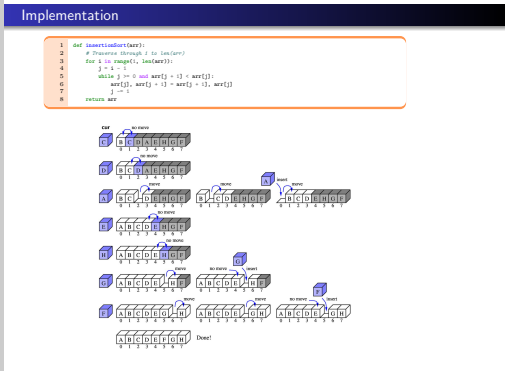


Insertion sort is one sorting algorithm used to sort data in different data structures. It is one of the simplest sorting algorithms that works best when the data size is small (we will discuss why this is the case soon).



Say that we have an array of size 5, populated with values: $[2, 3, 4, 1, 6]$. Our goal is to sort the array so we end up with $[1, 2, 3, 4, 6]$. Insertion sort says to break the arrays into sub-arrays and sort them individually, which results in a sorted array. If we had an array of size 1, that would already be sorted because there is no other element to compare it to. As such, we assume that the first element is sorted because we treat it as its own subarray.

The next subarray will be of size 2, which is $[2, 3]$. To perform the sort, the two elements will need to be compared. For an array of size 2, this is trivial. However, when we get to the full array of size 5, there is no way to keep track of where each element is without using pointers. So, let's take two pointers, i and j.

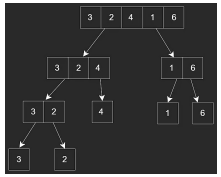


j will always be behind i such that it will never cross i. The i pointer points of the index $n-1$ where n is the size of the current sub-array. The j pointer starts off with being one index behind i and as long as j does not go out of bounds, that is, it is not at a negative index, and the $j+1$ element is smaller than the jth element, we keep decrementing j. This will ensure that we have sorted all the elements before the ith index before moving to the next sub-array (iteration).

Concept

- Merge sort involves splitting an array into halves until subarrays of size one are reached, then sorting and merging them.
- Example: Consider an array of size 5: [3,2,4,1,6]. The goal is to sort it in non-decreasing order.
- This process is recursive, using a two-branch recursion approach.

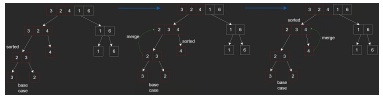
```
def mergeSort(arr, s, e):
    if e - s + 1 <= 1:
        return arr
    m = (s + e) // 2
    mergeSort(arr, s, m)
    mergeSort(arr, m + 1, e)
    merge(arr, s, m, e)
    return arr
```



The concept behind merge sort is very simple. Keep splitting the array into halves until the subarrays hit a size of one, sort them, and merge the sorted arrays, which will result in an ultimate sorted array. You might have figured out that this sounds exactly like the fibonacci sequence using recursion, and you would be right! We can, and will be using recursion to perform this. More specifically, two branch recursion.

Visualization

- The mergeSort function recursively sorts the left and right halves of the array.
- After sorting each half, the merge function combines them into a sorted array.



The enqueue operation adds elements to the tail of the queue until the size of the queue is reached. Since adding to the end of the queue requires no shifting of the elements, this operation runs in $\mathcal{O}(1)$. The pseudocode and visual demonstrates this.

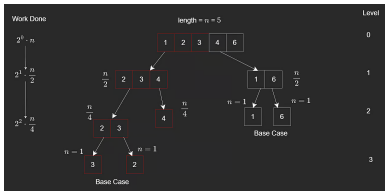
Pseudocode

```
# Merge in-place
def merge(arr, s, m, e):
    # Copy the sorted left & right halfs to temp arrays
    L = arr[s: m + 1]
    R = arr[m + 1: e + 1]

    i = 0 # index for L
    j = 0 # index for R
    k = s # index for arr

    # Merge the two sorted halfs into the original array
    while i < len(L) and j < len(R):
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # One of the halfs will have elements remaining
    while i < len(L):
        arr[k] = L[i]
        i += 1
        k += 1
    while j < len(R):
        arr[k] = R[j]
        j += 1
        k += 1
```

Recall that even though the visual only demonstrates the merging of the ultimate sub-array, recursion tells us that the merge happens on every level after the arrays are sorted because we would never have gotten to the ultimate array if the subarrays had not been sorted and merged. The piece of code used for i pointer and j pointer is actually referred to as the two pointer technique and is extremely useful when we have two arrays and need to go through them simultaneously to perform some logic. This could actually be used to combine two arrays and do so in $\mathcal{O}(n)$ time.

Time Complexity

- Merge Sort runs in $O(n \log n)$ time.
- This is due to the recursion depth ($\log n$) and the merge step, which takes n steps at each level.
- Overall time complexity is a product of these two factors.



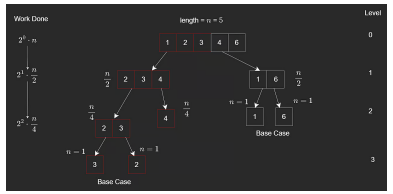
Merge Sort runs in $O(n \log n)$. How so? Let's do some analysis. Recall from the Fibonacci example when we kept splitting each sub-problem into two other sub-problems. We have a similar case here because our recursive tree is a tree with a branching-factor of 2, except we are going in the opposite direction. If n is the length of our array at any given level, our subarrays in the next level have a length $n/2$.

From our example above, we go from $n = 4$ to $n = 2$ to $n = 1$ which is the base case.

The question here is how many times can we divide n by 2 until we hit the base case, i.e., this would look like $n/2^x$ where x is the number of times we need to divide n by two until we get to one. Indeed,

Time Complexity

- Merge Sort runs in $O(n \log n)$ time.
- This is due to the recursion depth ($\log n$) and the merge step, which takes n steps at each level.
- Overall time complexity is a product of these two factors.



$$n/2^x = 1$$

Isolate n by multiplying both sides by 2^x,

$$n = 2^x$$

To solve for x, take the logarithm of both sides

$$\log n = \log 2^x$$

According to logarithm rules, we can simplify this to:

$$\log n = x \log 2$$

$\log 2$ is basically asking 2 to the power of what is equal to 2, i.e., $2^? = 2$, which is just 1,

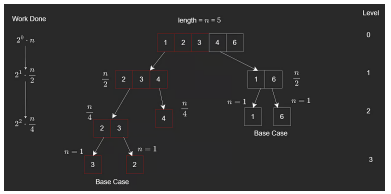
$$\log n = x \times 1$$

$$\log n = x$$

The ultimate answer is $x = \log n$.

Time Complexity

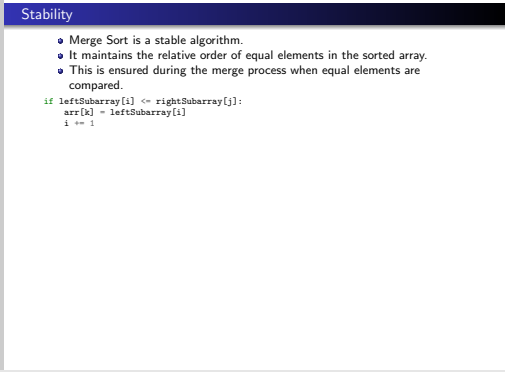
- Merge Sort runs in $O(n \log n)$ time.
- This is due to the recursion depth ($\log n$) and the merge step, which takes n steps at each level.
- Overall time complexity is a product of these two factors.



Having solved for x, our resulting answer is $\log n$ which is the complexity for the merge-Sort recursive call. We are not done yet, however.

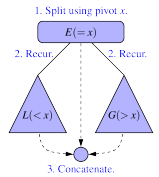
Let's analyze the merge subroutine. The merge subroutine will take n steps because at any level of the tree, we have n elements to compare and sort, where n is the length of the original array.

This results in an $O(n \log n)$ time complexity. The visual below gives more detail on how we arrived at our result. If you are interested, you can solve for the summation of the "Work Done" column on the left, and you will arrive at the same answer.



QMerge Sort proves to be a stable algorithm because if we have a pair of duplicates, say, 7, the 7in the left subarray will always end up in the merged array first followed by the 7 in the right subarray. This is because when we compare the *ith* element in the left subarray to the *jth* element in the right subarray for equality, we pick the one in the left subarray, maintaining the relative order.

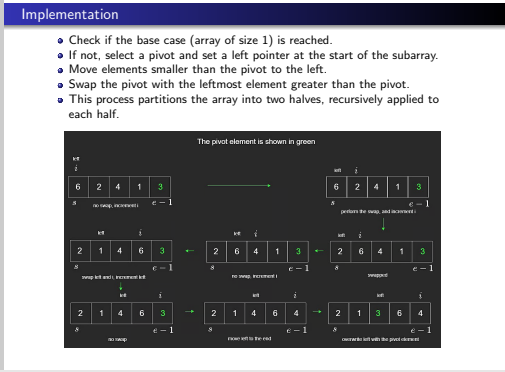
Picking a Pivot Value
- Quicksort involves selecting a pivot value and partitioning the array around it.
- Common pivot selection strategies:
  - First index
  - Last index
  - Median value
  - Random pivot
- The choice of pivot depends on the data's size and initial order.
- For simplicity, we often use the last index as the pivot.



The idea behind quicksort is to pick an index, which is called the pivot, and partition the array such that every value to the left is less than or equal to the pivot and every value to the right is greater than the pivot. Generally, there are several tested and tried options when it comes to picking a pivot:

- Pick the first index.
- Pick the last index.
- Pick the median (pick the first, last, and the middle value and find their median and perform the split at the median).
- Pick a random pivot.

You may be asking which pivot to pick? This is dependent on the data itself, both the size and the initial order. For coding interviews, we can keep things simple, so in this chapter we will use the last index as our pivot.



We will pick a pivot if we haven't already hit the base case which is an array of size 1 and pick a left pointer, which will point to the left-most element of the current subarray to begin with. Then, we will iterate through our array and if we find an element smaller than our pivot element, we will swap the current element with the element at our left pointer and increment the left pointer.

Once this condition terminates, we will bring the left element to the end of the array and bring the pivot element to the left position. This makes sense because once we have exhausted all the elements that are smaller than the pivot element, and put them to the left of the pivot, the left pointer will now be at the first element that is greater than the pivot. And, every element after it will also be greater than the pivot. This is why we move the left element to the end and then bring the pivot element to the left.

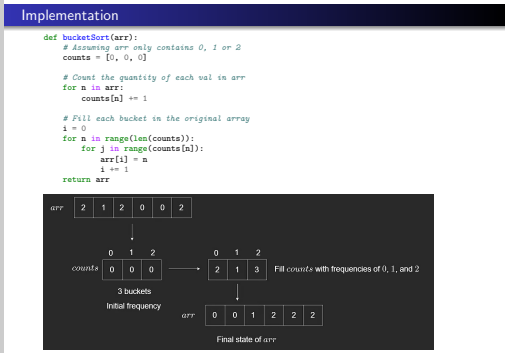
This results in all the elements less than or equal to the pivot to be on the left and the rest on the right.

Concept

- Bucket sort is effective for datasets with values within a specific range.
- Example: Consider an array of size 6 with values ranging from 0 to 2.
- The algorithm creates a bucket for each number (0, 1, 2) to count the frequency.
- These buckets are positions in an array mapping the frequencies of each number.

Imagine we have an array of size 6 and it contains values of an inclusive range of 0-2. The idea behind bucket sort is to create a "bucket" for each one of the numbers and map them to their respective buckets.
There will be a bucket for 0, 1, and 2. This bucket, which is just a position in a specified array will contain the frequencies of each one of the values within the range. For the sake of this example, we only have three values and accordingly we will have three buckets.

The term bucket will really just translate into a position in an array where we will map these frequencies.



The first part of the pseudocode, right before we do i $= 0$, corresponds to filling up each one of the buckets. The *i* pointer will keep track of the next insertion position for our original array, `arr`. The *n* pointer keeps track of the current position of the `counts` array. The *j* pointer keeps track of the number of times that `counts[n]` has appeared. So, knowing that, we have our `counts` array which is $[2, 1, 3]$. And, our original input array is $[2, 1, 2, 0, 0, 2]$.

At the first iteration, $n = 0$, which corresponds to 2 in `counts`. Our inner loop will run two times, overwriting `arr[0]` and `arr[1]` to be 0. This makes perfect sense because we had two zeros and putting them in `arr` in an adjacent manner will result in them being sorted. This process will continue for each number and the ultimate state of `arr` will be $[0, 0, 1, 2, 2, 2]$ which is the ultimate goal.

Time Complexity and Stability

- The time complexity of bucket sort is $O(n)$.
- The first loop runs for each element, and the nested loop depends on each value's frequency.
- Total operations correlate with the number of elements, not their square.
- Bucket sort is not a stable sorting algorithm.
- It overwrites the original array without preserving the relative order of values.
- No swapping is involved, making it unstable.

You may be looking at the nested for loop and immediately thinking, that is $O(n^2)$. That is not quite right. Let's do some analysis. We know that for the first for loop, we are performing n steps since we are going through all the elements and counting frequency.

The first for loop will run n times where n is the length of the counts array. However, the inner loop will only run until `counts[n]`, which is different every time. The first time it will be 2, then 1 and then 3. Therefore, our algorithm belongs to $O(n)$.

It should be noted that nested for loops don't always mean a time complexity of $O(n^2)$. As we saw above, it is important to analyze how many times the inner for loop executes on each outer for loop iteration. More information on this can be found in the lessons section for Big-O Notation.