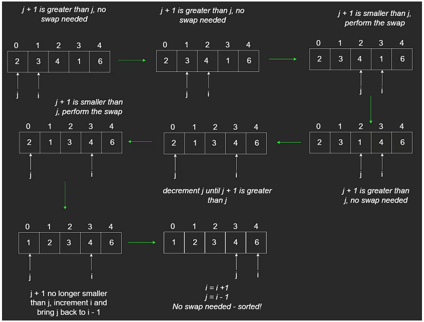
Sorting

Presented by Yasin Ceran

January 31, 2024

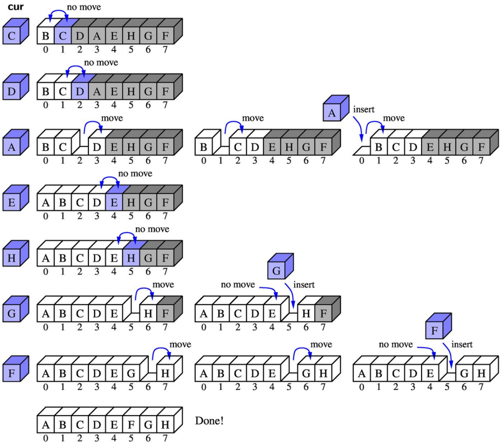
Concept

- 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.



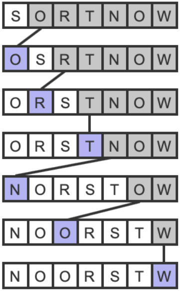
Implementation

```python
def insertionSort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        j = i - 1
        while j >= 0 and arr[j + 1] < arr[j]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            j -= 1
    return arr
```



Stability

- A sorting algorithm is said to be stable if it guarantees that the relative order of two equivalent elements remains the same in the result as in the original sequence
- Insertion sort is a stable sorting algorithm.
- It maintains the relative order of equal elements.
- Example:



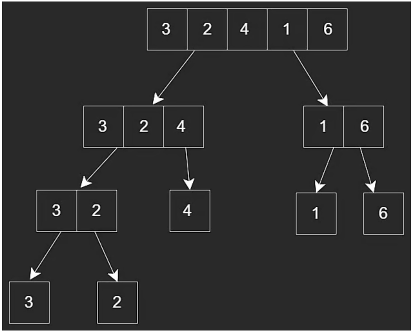
Time and Space Complexity

- Time Complexity:
  - Best Case: $O(n)$, when the array is already sorted.
  - Worst Case: $O(n^2)$, when the array is in reverse order.
- Space Complexity: $O(1)$, as it requires a constant amount of additional space.

Concept

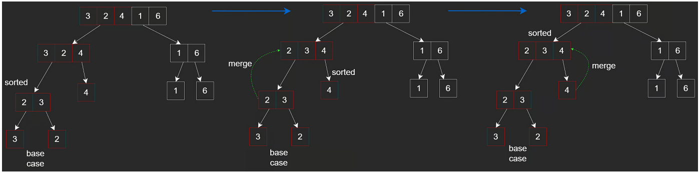
- Merge sort involves splitting an array into halves until subarrays of size one are reached, then sorting and merging them.
- Example: Consider an array of size 5: [3,2,4,1,6]. The goal is to sort it in non-decreasing order.
- This process is recursive, using a two-branch recursion approach.

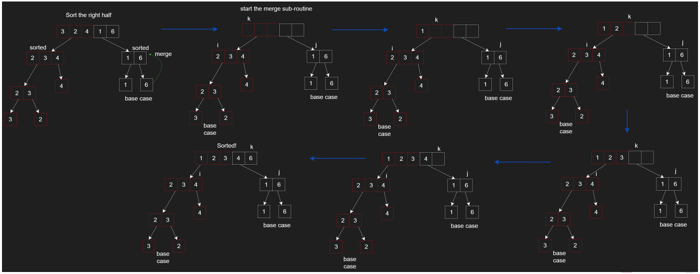
```python
def mergeSort(arr, s, e):
    if e - s + 1 <= 1:
        return arr
    m = (s + e) // 2
    mergeSort(arr, s, m)
    mergeSort(arr, m + 1, e)
    merge(arr, s, m, e)
    return arr
```



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.



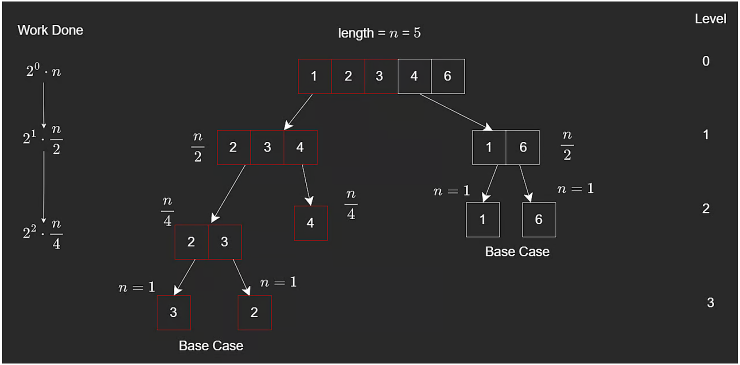


Pseudocode

```
# 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
```

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.



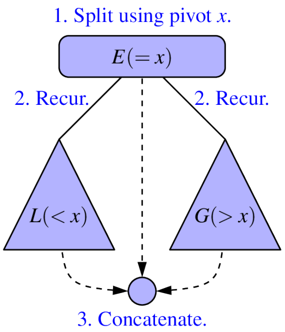
Stability

- Merge Sort is a stable algorithm.
- It maintains the relative order of equal elements in the sorted array.
- This is ensured during the merge process when equal elements are compared.

```
if leftSubarray[i] <= rightSubarray[j]:
    arr[k] = leftSubarray[i]
    i += 1
```

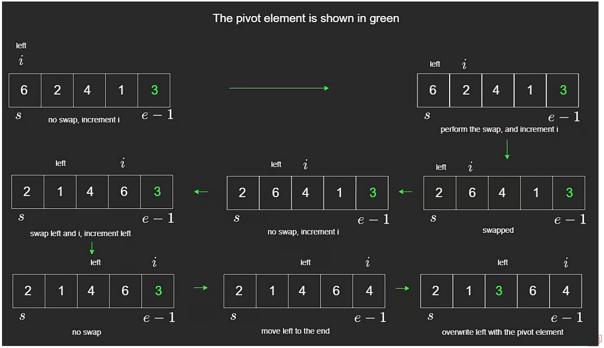
Picking a Pivot Value

- 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.



Implementation

- Check if the base case (array of size 1) is reached.
- If not, select a pivot and set a left pointer at the start of the subarray.
- Move elements smaller than the pivot to the left.
- Swap the pivot with the leftmost element greater than the pivot.
- This process partitions the array into two halves, recursively applied to each half.



Pseudocode

```python
# Implementation of QuickSort
def quickSort(arr, s, e):
    if e - s + 1 <= 1:
        return

    pivot = arr[e]
    left = s # pointer for left side

    # Partition: elements smaller than pivot on left side
    for i in range(s, e):
        if arr[i] < pivot:
            tmp = arr[left]
            arr[left] = arr[i]
            arr[i] = tmp
            left += 1

    # Move pivot in-between left & right sides
    arr[e] = arr[left]
    arr[left] = pivot

    # Quick sort left side
    quickSort(arr, s, left - 1)

    # Quick sort right side
    quickSort(arr, left + 1, e)

    return arr
```

Time Complexity

- Best Case: $O(n \log n)$, achieved with a balanced partition.
- Worst Case: $O(n^2)$, occurs when the pivot is the smallest or largest element.
- Unbalanced partitions increase the number of groups, leading to higher complexity.

Stability

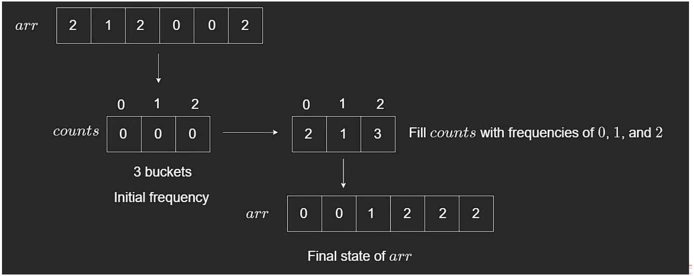
- Quicksort is not a stable sorting algorithm.
- It may exchange non-adjacent elements, changing the relative order of equal elements.
- Example: In [7,3,7,4,5], the two 7s might change their relative positions.

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.

Implementation

```python
def bucketSort(arr):
    # Assuming arr only contains 0, 1 or 2
    counts = [0, 0, 0]

    # Count the quantity of each val in arr
    for n in arr:
        counts[n] += 1

    # Fill each bucket in the original array
    i = 0
    for n in range(len(counts)):
        for j in range(counts[n]):
            arr[i] = n
            i += 1
    return arr
```



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.

Summary:

Algorithm	Big-O Time Complexity	Note
Insertion Sort	$O(n^2)$	Efficient for small or nearly sorted data
Merge Sort	$O(n \log n)$	Stable and efficient for large data
Quick Sort	$O(n \log n)$	Fast but can degrade to $O(n^2)$
Bucket Sort	$O(n)$	Optimal for uniform distribution within a range

Table: Comparison of sorting algorithms

- **Insertion Sort**: Best suited for small datasets or nearly sorted arrays. It's simple but inefficient for large datasets.
- **Merge Sort**: Ideal for large datasets due to its stable and consistent $O(n \log n)$ performance. It requires extra space.
- **Quick Sort**: Offers excellent average-case performance. However, its worst-case performance can be problematic with certain pivot choices.
- **Bucket Sort**: Extremely efficient for sorting data within a specific range. It's particularly powerful when the data is uniformly distributed over the range.