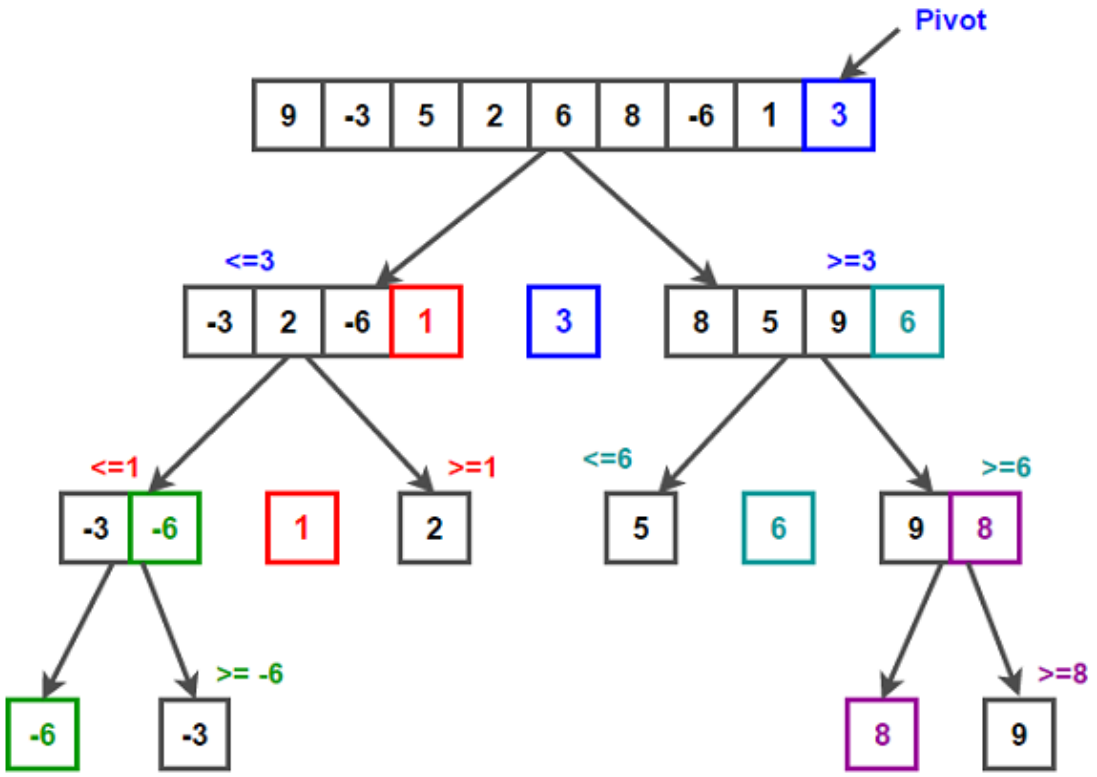
Quicksort

QuickSort is a divide-and-conquer sorting algorithm which segregates an array into smaller subarrays and then recursively sorts them.
Steps involved in the algorithm :
Choose a pivot element from the array.
Partition the array into two parts: one with elements less than or equal to the pivot, and another with elements greater than the pivot.
Recursively apply QuickSort to both subarrays.
Combine the sorted subarrays with the pivot in the middle.



Initial Array
Array: [9, -3, 5, 2, 6, 8, -6, 1, 3]
Pivot: 3 (in blue)

Step 1: Partitioning the Array
Left Partition (<= 3): [-3, 2, -6, 1]
Right Partition (>= 3): [8, 5, 9, 6]
The pivot 3 is put in the correct sorted position.

Step 2: Sorting Left Partition <= 3
Array: [-3, 2, -6, 1]
Pivot: 1 (in red)
Left Partition <= 1: [-3, -6]
Right Partition >= 1: [2]
The pivot 1 is at its correct position.

Step 3: Sorting Sub-partitions of the Left Partition
Array: [-3, -6]
Pivot: -6 (in green)
Left Partition: []
Right Partition: [-3]
The array becomes [-6, -3] fully sorted.

Step 4: Sorting the Right Partition (>= 3)
Array: [8, 5, 9, 6]
Pivot: 6 (in cyan)
Left Partition <= 6: [5]
Right Partition >= 6: [8, 9]
The pivot 6 is at its correct position.

Step 5: Sorting Sub-partitions of the Right Partition
Array: [8, 9]
Pivot: 8
Left Partition: []
Right Partition: [9]
The array becomes [8, 9] which is now completely sorted.

Step 6: Final Merge
In each recursion level, the pivot will go to the appropriate position, and then the sorted partitions will get merged:
Left sorted array: [-6, -3, 1, 2]
Pivot: 3
Right sorted array: [5, 6, 8, 9]
Final sorted result: [-6, -3, 1, 2, 3, 5, 6, 8, 9]

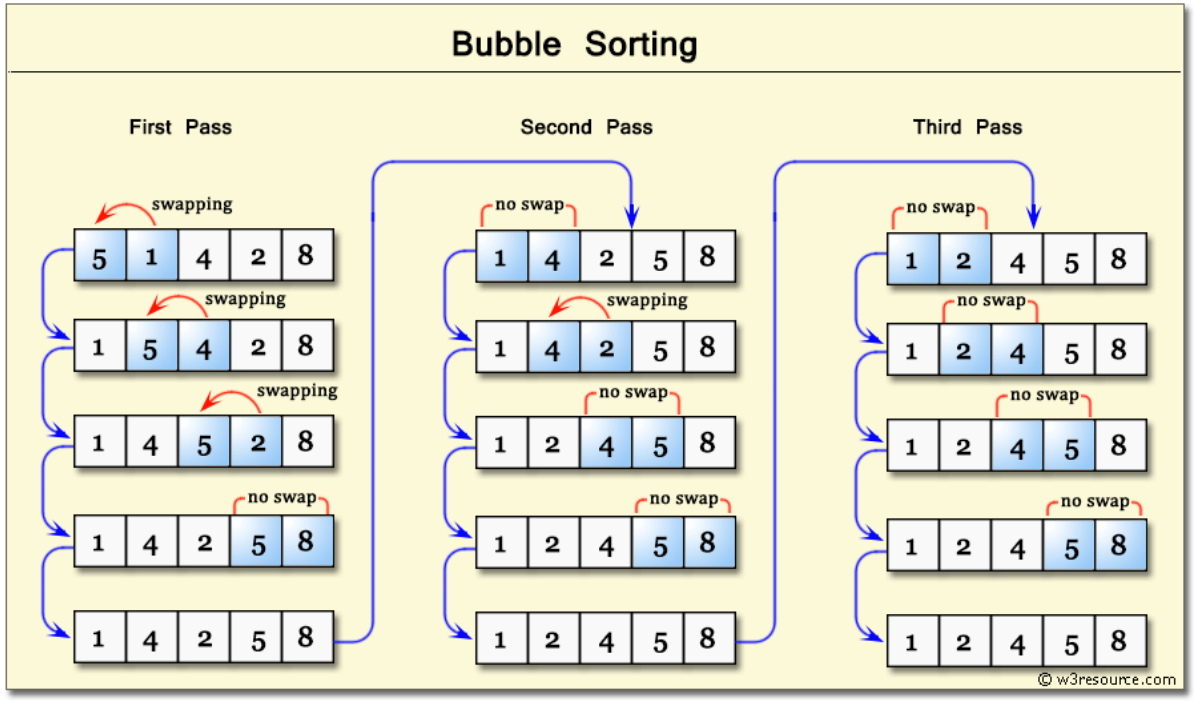
The time complexity of the Quicksort

1. Best Case: $O(n\log n)$

- In the best-case scenario, the pivot divides the array into two roughly equal halves at each step. For example, with the array [9, -3, 5, 2, 6, 8, -6, 1, 3] and the ideal pivot 5, the recursion tree has a depth of $\log n$, as the array size halves with each level. At every level, $O(n)$ comparisons are made to partition the array, resulting in a time complexity of $O(n\log n)$.
- 2. Average Case:$O(n\log n)$
- In the average-case scenario, the pivot divides the array into reasonably balanced partitions. For example, with the array [9, -3, 5, 2, 6, 8, -6, 1, 3] and a randomly chosen pivot of -3, the depth of the recursion tree and the total number of comparisons are similar to the best-case scenario. This results in a time complexity of $O(n\log n)$
- 3. Worst case : $O(n$^2)
- In the worst-case scenario, the pivot is poorly chosen, leading to highly unbalanced partitions. For example, if the pivot is always the smallest or largest element, one partition contains all but one element. The recursion tree becomes essentially a straight line with n levels. This results in a time complexity of $O(n$^2). For instance, with the input array [9, -3, 5, 2, 6, 8, -6, 1, 3] and a poorly chosen pivot of 9 (the largest element in this step), the total work is $n\times O(n)=O(n$^2).

Bubble sort

Bubble sort is a sorting algorithm that repeatedly compares adjacent elements in a list and swaps them if they are in the wrong order. This process is repeated until the entire list is sorted.
Step-by-step Process: Start at the beginning of the list. Compare the first pair of adjacent elements. If the first element is greater than the second, swap them; otherwise, leave them as they are.
Move to the next pair and repeat until the end of the list.
After one pass, the largest element "bubbles up" to its correct position.
Repeat the process for the rest of the unsorted elements.



Initial array:[5, 1, 4, 2, 8].

Step 1:
Compare 5 and 1. Swap them → [1, 5, 4, 2, 8]
Compare 5 and 4. Swap them → [1, 4, 5, 2, 8]
Compare 5 and 2. Swap them → [1, 4, 2, 5, 8]
Compare 5 and 8. No swap needed → [1, 4, 2, 5, 8]
At the end of Pass 1, the largest element (8) is in its correct position.

Step 2:
Compare 1 and 4. No swap → [1, 4, 2, 5, 8]
Compare 4 and 2. Swap them → [1, 2, 4, 5, 8]
Compare 4 and 5. No swap → [1, 2, 4, 5, 8]
Compare 5 and 8. No swap → [1, 2, 4, 5, 8]
At the end of Pass 2, the second largest element (5) is in its correct position.

Step 3:
Compare 1 and 2. No swap → [1, 2, 4, 5, 8]
Compare 2 and 4. No swap → [1, 2, 4, 5, 8]
Compare 4 and 5. No swap → [1, 2, 4, 5, 8]
At the end of Pass 3, the third largest element (4) is in its correct position.

Step 4:
Compare 1 and 2. No swap → [1, 2, 4, 5, 8]
Compare 2 and 4. No swap → [1, 2, 4, 5, 8]
At the end of Pass 4, the entire array is sorted: [1, 2, 4, 5, 8].

The time complexity of the Bubblesort

1. Best Case: $O(n)$

- In the best-case scenario, when the array is already sorted[1, 2, 3, 4, 5], no swaps are needed, and the algorithm terminates after a single pass. This results in a time complexity of $O(n)$, with $n-1$ comparisons and 0 swaps.

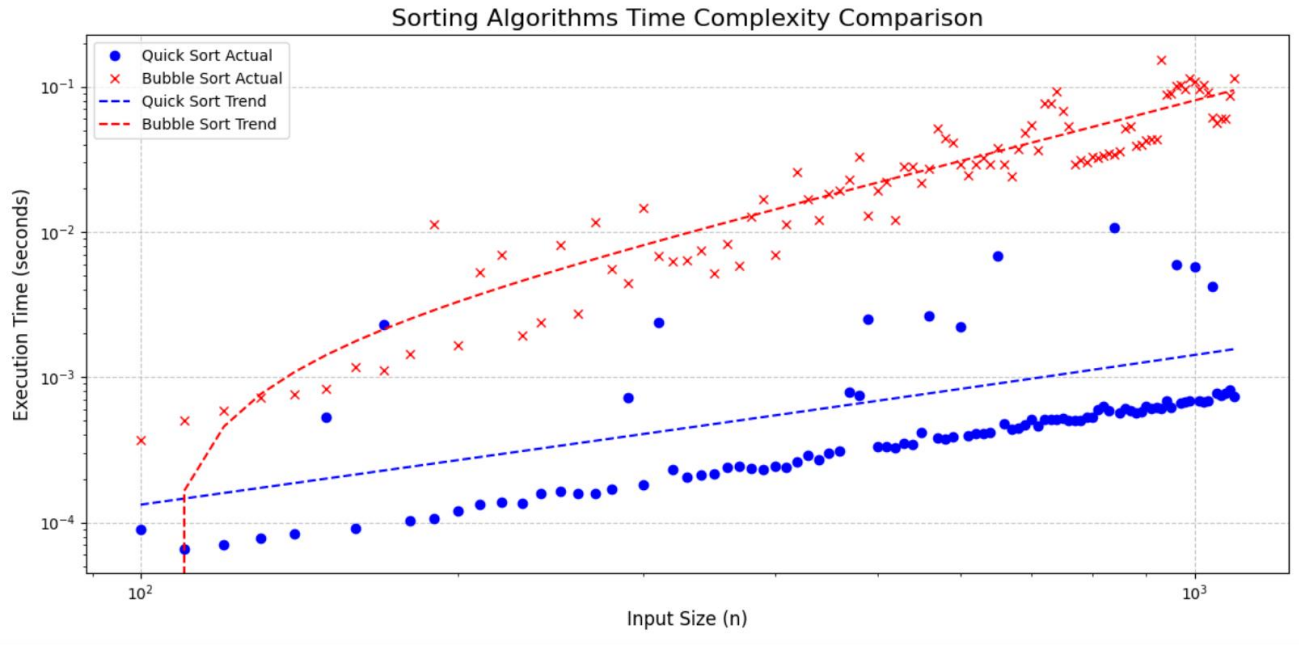
2. Average Case:$O(n\log n)$

- In the average-case scenario, when the array is in random order[5, 1, 4, 2, 8], multiple passes and some swaps are required. This results in a time complexity of $O(n$^2), with (n*($n-1$))/2 comparisons and fewer swaps compared to the worst case, but still quadratic on average.

3. Worst case : $O(n$^2)

- In the worst-case scenario, when the array is sorted in reverse order[5, 4, 3, 2, 1], every element needs to be compared and swapped in each pass. This results in a time complexity of $O(n$^2), with (n*($n-1$))/2 comparisons and an equal number of swaps.

Quick sort vs Bubble sort



In this plot, I used 100 random lists to test the performance of Bubble Sort and Quick Sort. The data was sourced from a CSV file to generate the plot. Regression lines were added to illustrate the time complexity of each sorting algorithm.

Quick Sort:
The execution time increases slowly as input size n grows, reflecting its efficiency.
The theoretical trend line follows O(nlogn), which is what one would expect for its time complexity.
The actual data points are very close to the trend line.

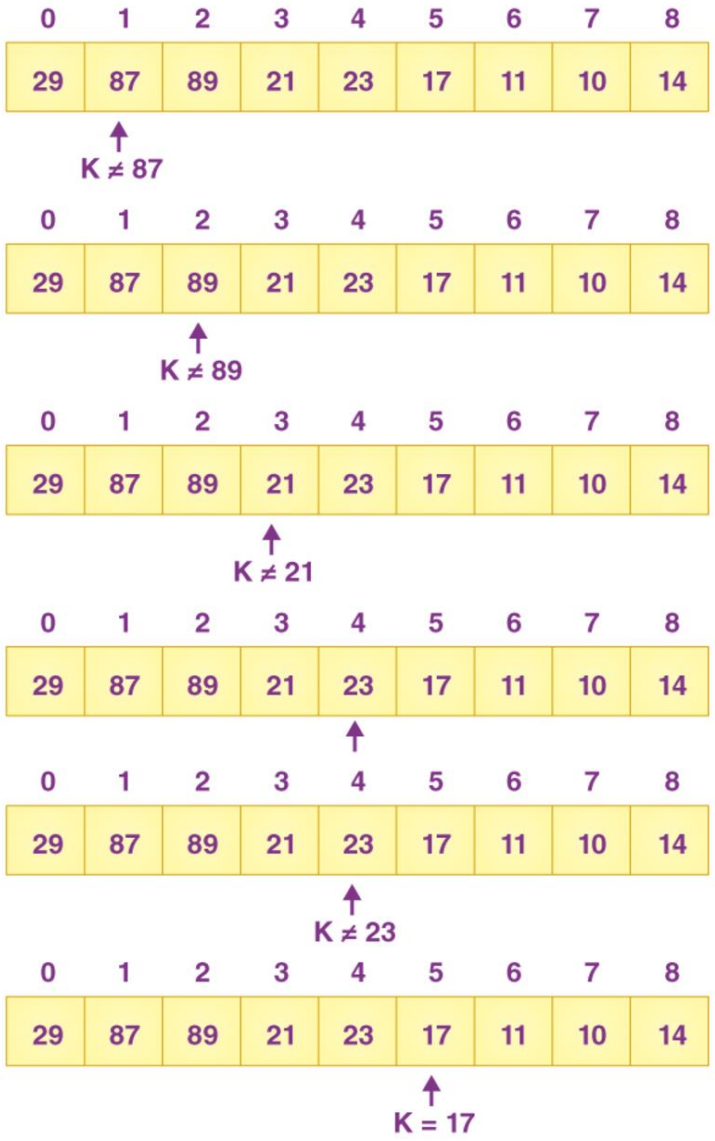
Bubble Sort:
The execution time increases rapidly with increasing input size n, showing its inefficiency.
The trend line suggests a complexity of O(n^2) which agrees with its theoretical performance.
Actual data points also follow this trend, especially for larger input sizes (right side).

Conclusion:
This chart clearly shows that Quick Sort outperforms Bubble Sort, especially for large input sizes. The slower growth of execution time for Quick Sort makes it a far more efficient option for sorting tasks.

linear search , target k = 17

Linear Search: The simplest search algorithm works by checking every element in the list one by one until the desired element is found or when the end of the list is reached.

Steps in Linear Search:
Begin with the first element of the array.
Compare each element with the target.
Return the index of the element if the target is found.
If the end of the list is reached without finding the target, return False, which indicates the target is not in the array.



1. **Step 1**: Compare K with the first element (index 0).
   - Element: 29
   - Comparison: $K \neq 29$
   - Result: Move to the next element.
2. **Step 2**: Compare K with the second element (index 1).
   - Element: 87
   - Comparison: $K \neq 87$
   - Result: Move to the next element.
3. **Step 3**: Compare K with the third element (index 2).
   - Element: 89
   - Comparison: $K \neq 89$
   - Result: Move to the next element.
4. **Step 4**: Compare K with the fourth element (index 3).
   - Element: 21
   - Comparison: $K \neq 21$
   - Result: Move to the next element.
5. **Step 5**: Compare K with the fifth element (index 4).
   - Element: 23
   - Comparison: $K \neq 23$
   - Result: Move to the next element.
6. **Step 6**: Compare K with the sixth element (index 5).
   - Element: 17
   - Comparison: $K = 17$
   - Result: Target K is found at index 5.

Output:

The target value $K = 17$ is located at **index 5** in the array.

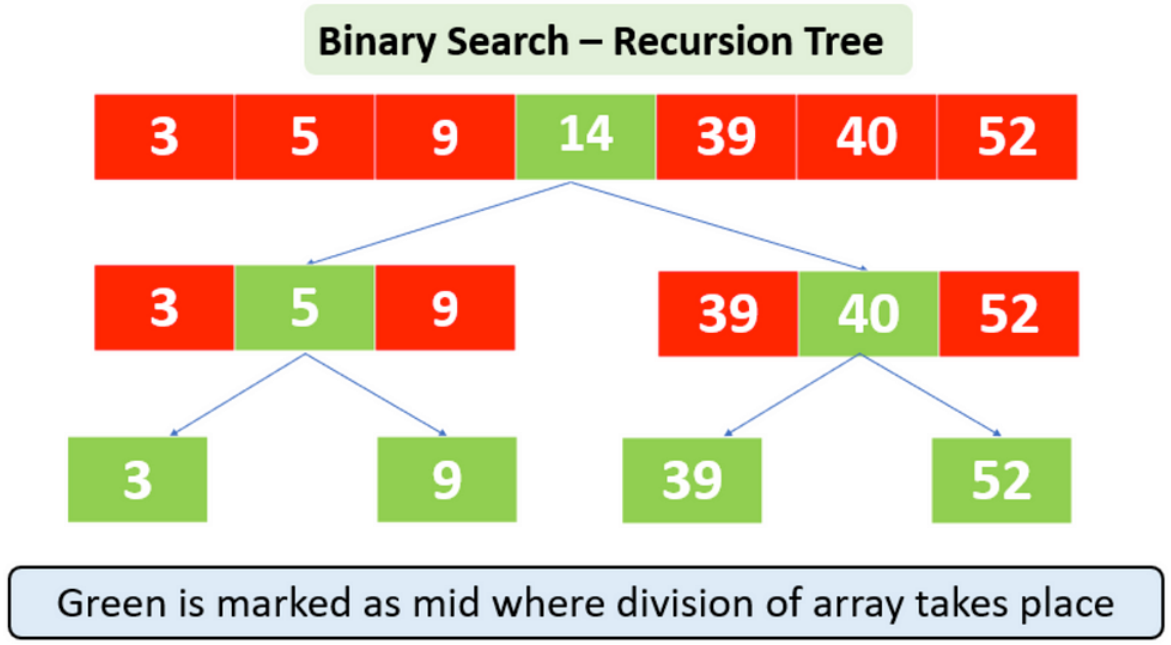
The time complexity of the linear search

- 1. Best Case: O(1)
- In the best-case scenario for a linear search, the target element is at the very beginning of the array. This means the algorithm finds the element after just one comparison, resulting in a time complexity of O(1). For example , in the list [29, 87, 89, 21, 23, 17, 11, 10, 14],if the target is 29 and it's the first element in the array, the search will immediately find it and return the index 0.
- 2. Average Case: O(n)
- In the average case for a linear search, the target element is typically found somewhere in the middle of the array. This means that, on average, about half of the elements need to be checked before finding the target. For example, if the target is 23, it might take several comparisons to find it. As a result, the time complexity is O(n), where n is the number of elements in the array.
- 3. Worst Case: O(n)
- In the worst-case scenario for a linear search, the target element is either at the very end of the array or not present at all. This requires checking all n elements before finding the target or concluding it's not there. For example, if the target is 14 and it's the last element, it takes 9 comparisons to find it. Thus, the time complexity is $O(n)$.

Binary search , target k = 40

Binary Search is a far quicker search algorithm compared to the linear search, but it will only work on sorted arrays. It works by repeatedly dividing the search interval in half.

Steps in Binary Search: Start with the middle element of the array.
If the middle element equals the target, return the index.
If the target is smaller than the middle element, search the left half.
If the target is larger than the middle element, then search the right half.
Repeat until the target is found or the interval is empty.



Initial Array: [3, 5, 9, 14, 39, 40, 52]
Step 1: Find Midpoint
Calculate the middle index of the array: mid = (0 + 6) // 2 = 3
The midpoint is array[3] = 14. Mark 14 in green as the division point.
Decision: Compare the target T with 14.
If T < 14, search the left half: [3, 5, 9].
If T > 14, search the right half: [39, 40, 52].

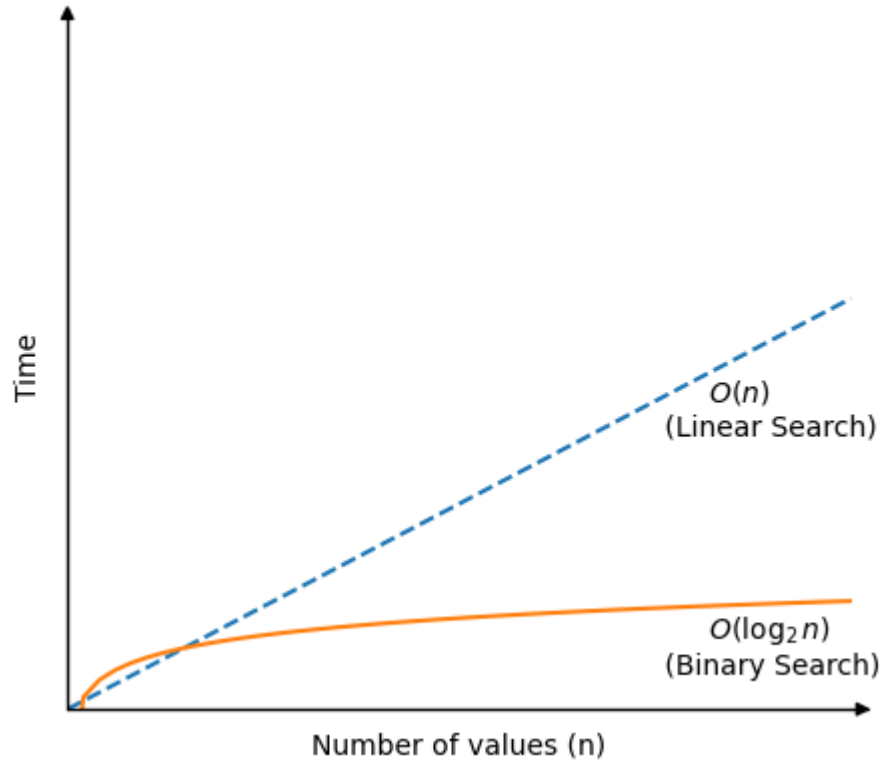
Left Sub-array:[3, 5, 9]
Step 2: Find Midpoint
For [3, 5, 9], calculate the middle index: mid = (0 + 2) // 2 = 1.
The midpoint is array[1] = 5. Mark 5 in green.
Decision: Compare the target T with 5.
If T < 5, search the left half: [3].
If T > 5, search the right half: [9].

Right Sub-array: [39, 40, 52]
Step 3: Find Midpoint
For [39, 40, 52], calculate the middle index: mid = (4 + 6) // 2 = 5.
The midpoint is array[5] = 40. Mark 40 in green.
Decision: Compare the target T with 40.
If T < 40, search the left half: [39].
If T > 40, search the right half: [52].

The time complexity of the Binary search

- 1. Best Case: O(1)
- In the best-case scenario for binary search, the target value is found at the middle of the array during the first comparison. This means the search is completed in just one step, resulting in a time complexity of O(1). For example, if the array is [3, 5, 9, 14, 39, 40, 52] and the target is 14 (the middle element), the search finds the target immediately.
- 2. Average Case: O(log n)
- In the average case for binary search, the target value can be at any random position in the array. It requires a moderate number of divisions to find. Binary search halves the search space at each step, resulting in a balanced recursive tree structure. On average, the target is found at a depth proportional to log2(n). This makes time complexity O(log n). Consider the following example: if the target element 39 must be found from the array [3, 5, 9, 14, 39, 40, 52], after a few comparisons it would result in a time complexity of O(log n).
- 3. Worst Case: O(log n)
- The worst case for binary search is that the target value either comes at the end of the search space or not in the array, hence requiring maximum numbers of divisions. Since with each comparison in binary search, it divides the search space into two, it leads to a time complexity of O(log n). For instance, for the target 52, from the array [3, 5, 9, 14, 39, 40, 52], it takes some number of comparisons; in the case of an absent target, it would continue until the space was empty.

Linear search vs Binary search



1. Dataset Size

As n becomes large, the difference between (O(n) and (O(logn) becomes huge.

For instance, to search through 1 billion elements in a dataset :Linear Search needs 1 billion comparisons while Binary Search needs Only ~30 comparisons

2. Sorted Data Requirement

Linear Search does not require the data to be sorted. Thus, it is applied in unsorted datasets where sorting is either too expensive or even unnecessary.

Binary Search requires sorted data. Thus, an initial sorting costs O(nlogn), but this is a one-time cost for static datasets. For dynamic datasets, i.e., frequent inserts/deletes, maintaining sorted order can be expensive.

3. Use Cases

Linear Search: Small datasets, data that is unsorted, or when the search has to be simple and flexible.

Binary Search: Large, sorted datasets, or cases where efficiency in search is crucial, such as databases or search engines.

Extra

Compare the time it takes for Bubble sort implemented using iteration with your implementation that uses recursion (you can do a simple paired t-test if you sort the same randomized lists)

- Time Comparison:
- size iterative_avg_time recursive_avg_time
- 0 100 0.000572 0.000006
- 1 200 0.000108 0.001971
- 2 300 0.002220 0.002638
- 3 400 0.005238 0.003798
- 4 500 0.007817 0.007162
- 5 600 0.011292 0.012066
- 6 700 0.016783 0.016400
- 7 800 0.021922 0.021855
- 8 900 0.029775 0.028165
- 9 1000 0.036501 0.034268

- Paired t-Test Results:
- t-statistic: 1.1363323173766038
- p-value: 0.25856071602874336

To find the critical t-value, consider the degrees of freedom (df), significance level (df), and whether the test is one-tailed or two-tailed.
Given the sample size n=1000, the degrees of freedom df=999. From a 95% confidence level, that is a two-tailed test, so α=0.05, with each tail α/2=0.025. The critical t-value for df=999 is approximately ±1.962.
For a 99% confidence level, which is a two-tailed test, α = 0.01, so that with each tail α/2= 0.005. The critical t-value for df=999 is approximately ±2.626.
This t-value of 1.7056 is less than both critical values, indicating the result is not significant at either confidence level.

Null Hypothesis ($H0$): There is no significant difference in the time taken by iterative and recursive Bubble Sort.
Alternative Hypothesis ($H1$): There is a significant difference in the times.
Since p>0.05:We fail to reject the null hypothesis. This means there is not enough evidence to conclude a significant difference in performance between iterative and recursive Bubble Sort for this dataset.

Use seaborn and pandas to create a scatter plot and linear regression with your measurements as data