Introduction to Programming and Problem Solving

Week 10: List: Searching and Sorting

Lecturer: Lemi Agrey Oliver

Department of Information Technology

Kumi University

codingissweet@gmail.com

Content

- Introduction to searching and sorting
- Linear search and binary search
- Key concepts in list sorting
- Application of list sorting
- Bubble sort
- Merge sort
- Selection sort
- Merits and demerits of some algorithms
- etc

Introduction to List: Searching and Sorting

- Sorting and searching are fundamental operations in computer science and programming.
- In Python, you can perform sorting and searching on various data structures, including lists, arrays, and dictionaries.
- Sorting in Python:
- Sorting is the process of arranging elements in a particular order, such as ascending or descending.
- Python provides built-in functions and methods to perform sorting.
- List Sorting with sorted() and sort():

Introduction to List: Searching and Sorting+

- sorted(): This function returns a new sorted list without modifying the original list.
- sort(): This method sorts the list in-place, modifying the original list.
- Example using sorted():
- my_list = [3, 1, 2, 4]
- sorted_list = sorted(my_list) # Returns a new sorted list
- print(sorted_list) # Output: [1, 2, 3, 4]
- Example using sort():
- my_list = [3, 1, 2, 4]

Introduction to List: Searching and Sorting++

```
1.  my_list.sort()  # Sorts the list in-place
2.  print(my_list)  # Output: [1, 2, 3, 4]
```

- Custom Sorting with sorted() and sort():
- You can specify custom sorting criteria using the key parameter.
- Example using a custom key function:

```
1.    my_list=["apple", "Banana", "cherry", "date"]
2. #Sort case-insensitively
3. sorted_list = sorted(my_list, key=str.lower)
4. print(sorted_list)
5. Output: ['apple', 'Banana', 'cherry', 'date']
```

Searching in Python

- **Searching in Python:**
- Searching is the process of finding a specific element within a data structure.
- **List Searching:**
- You can use the in operator to check if an element is in a list or use the index() method to find the index of an element

Searching with in Operator

```
1. Example using in operator:
2. my_list = [1, 2, 3, 4, 5]
3. element = 3
4. if element in my_list:
5.     print(f"{element} found in the list.")
6. else:
7.     print(f"{element} not found in the list.")
8. Example using index():
9. my_list = [1, 2, 3, 4, 5]
10.element = 3
11.if element in my_list:
12.    index = my_list.index(element)
13.    print(f"{element} found at index {index}.")
14.else:
15.    print(f"{element} not found in the list.")
```

Linear search

- **Linear search**, also known as sequential search, is a straightforward searching algorithm used to find a specific element within a collection (typically an array or list).
- It works by checking each element one by one until the desired element is found or until the entire collection has been searched.
- While it's not the most efficient search algorithm, it's simple and works for unsorted collections.

Linear Search Algorithm

- **Preconditions:**
- We have a target element we want to find in the collection.
- The collection can be sorted or unsorted; it doesn't matter.
- **Linear Search Loop:**
- Iterate through the collection one element at a time, starting from the first element.
- Compare each element with the target element:

Linear Search Algorithm+

- If they are equal, the search is successful; return the index of the element.
- If the end of the collection is reached without finding the target, return a "not found" indicator
- Example 1: Searching for a Number

```
1. def linear_search(arr, target):
2.     for i in range(len(arr)):
3.         if arr[i] == target:
4.             return i  # Target found at index i
5.     return -1  # Target not found
6. my_list = [5, 9, 2, 7, 1, 8, 3, 6, 4]
7. target = 8
8. result = linear_search(my_list, target)
9. if result != -1:
10.    print(f"{target} found at index {result}.")
11.else:
12.    print(f"{target} not found in the list.")
```

- In this example, linear search successfully finds the target (8) at index 5.

Example 2: Searching for a String

```
1. def linear_search_strings(arr, target):
2.     for i in range(len(arr)):
3.         if arr[i] == target:
4.             return i  # Target found at index i
5.     return -1  # Target not found
6. my_list = ["apple", "banana", "cherry", "date", "grape", "lemon", "orange"]
7. target = "lemon"
8. result = linear_search_strings(my_list, target)
9. if result != -1:
10.    print(f"{target} found at index {result}.")
11. else:
12.    print(f"{target} not found in the list.")
```

Linear Search Algorithm+++

```
1. def linear_search(arr, target):
2.     for index, element in enumerate(arr):
3.         if element == target:
4.             return index  # Return the index of the found element
5.     return -1  # Return -1 if the element is not found
6. # Example usage:
7. my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
8. target = 4
9. result = linear_search(my_list, target)
10.if result != -1:
11.    print(f"{target} found at index {result}.")
12.else:
13.    print(f"{target} not found in the list.")
```

Linear Search Algorithm++++

- In the above example,
- We define a function linear_search that takes a list (arr) and a target element (target) as arguments.
- We use a for loop to iterate through the list. The enumerate function is used to obtain both the element and its index during each iteration.
- Inside the loop, we check if the current element is equal to the target element. If a match is found, we return the index of the found element.
- If no match is found after looping through the entire list, we return -1 to indicate that the target element is not in the list.

Merits of Linear search

- Simplicity: Linear search is easy to understand and implement. It's a straightforward algorithm that can be quickly coded in Python without the need for complex logic.
- Applicability: Linear search can be used for searching in both sorted and unsorted lists. This makes it versatile and suitable for various use cases.
- No Preprocessing: Linear search does not require any preprocessing or sorting of the data, making it ideal when the list is not sorted.
- Minimal Memory Usage: Linear search is an in-place algorithm, meaning it doesn't require additional memory to perform the search, making it memory-efficient.

Demerits of Linear Search

- Inefficiency for Large Datasets: The primary disadvantage of linear search is its inefficiency for large datasets. It has a time complexity of O(n), where n is the size of the list. This means that in the worst case, it may need to examine every element in the list to find the target, which can be very time-consuming for large datasets.
- Inefficient for Frequent Searches: If you need to search the same list repeatedly, linear search may not be the best choice, as it does not take advantage of any pre-existing order in the data. Other search algorithms like binary search (for sorted data) or hash tables may be more efficient in such cases.

Demerits of Linear Search+

- Lack of Early Termination: Linear search continues until the entire list is traversed, even if the target element is found early in the list. This can result in unnecessary iterations and comparisons.
- Not Suitable for Large Databases: In the context of large databases or datasets, where real-time or near-instantaneous search is essential, more advanced search algorithms and data structures are preferred.

Binary Search Algorithm

- Binary search is a powerful and efficient algorithm for finding a specific element in a sorted collection (typically a list or array).
- It works by repeatedly dividing the search space in half until the desired element is found.
- **How Binary search work**
- Preconditions:
- The list or array must be sorted in ascending order.
- We have a target element we want to find in the collection.

Binary Search Algorithm+

- Initialization:
- Set two pointers, left and right, to the first and last indices of the collection, respectively.
- Binary Search Loop:
- Repeat until left is less than or equal to right:
- Calculate the middle index as (left + right) // 2.
- Compare the element at the middle index with the target element:
- If they are equal, the search is successful; return the index.

Binary Search Algorithm++

- If the middle element is less than the target element, update left to middle + 1.
- If the middle element is greater than the target element, update right to middle - 1.
- Termination:
- If the loop terminates without finding the target element, it means the element is not in the collection. Return a "not found" indicator (e.g., -1).

Binary Search Algorithm+++

Example 1: Searching for a Number

```
1. def binary_search(arr, target):
2.     left, right = 0, len(arr) - 1
3.     while left <= right:
4.         mid = (left + right) // 2
5.         if arr[mid] == target:
6.             return mid  # Target found
7.         elif arr[mid] < target:
8.             left = mid + 1
9.         else:
10.             right = mid - 1
11.    return -1  # Target not found
```

```
13.my_list = [2, 4, 5, 10, 13, 18,
   23, 29, 31, 51, 64]
14.target = 10
15.result = binary_search(my_list,
   target)
16.if result != -1:
17.    print(f"{target} found at
   index {result}.")
18.else:
19.    print(f"{target} not found in
   the list.")
```

Binary Search Algorithm++++

- In this example, binary search successfully finds the target (10) at index 3.

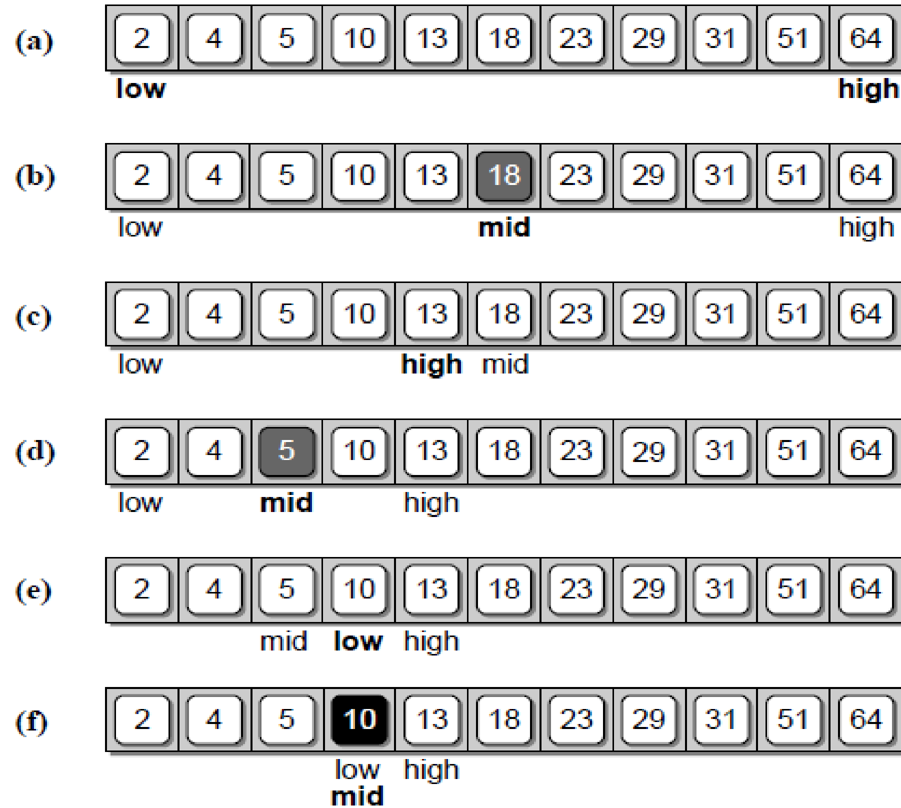

Fig.1 The steps performed by the binary search algorithm in searching for 10: (a)initial range of items, (b) locating the midpoint, (c) eliminating the upper half, (d) midpoint of the lower half, (e) eliminating the lower fourth, and (f) finding the target item. [1]

Binary Search Algorithm+++++

Example 2: Searching for a String

```
def binary_search_strings(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # Target not found
my_list = ["apple", "banana", "cherry", "date", "grape", "lemon", "orange"]
target = "date"
result = binary_search_strings(my_list, target)
if result != -1:
    print(f"{target} found at index {result}.")
else:
    print(f"{target} not found in the list.")
```

Output: date found at index 3.

Merits of Binary Search

- **Efficiency:** Binary search is highly efficient for searching in sorted lists or arrays. Its time complexity is O(log n), which means it can quickly narrow down the search space, making it ideal for large datasets.
- **Quick Find**: Binary search can find the target element in a fraction of the time required by linear search. This makes it a valuable choice when efficiency is a priority.
- **Fewer Comparisons:** Binary search typically requires fewer comparisons than other search algorithms, reducing the number of operations required to find the target element.

Merits of Binary Search+

- **Structured Data**: Binary search is suitable for working with structured data where elements have a natural ordering. This makes it applicable to various use cases, including searching in databases or sorted collections.
- **Predictable Performance:** Binary search consistently performs well for large datasets. Its time complexity remains logarithmic, regardless of the dataset's size.

Demerits of Binary Search

- **Sorted Data Requirement:** Binary search is only effective for sorted lists or arrays. If the data is not sorted, it must be sorted first, which can be an additional operation.
- **Complex Implementation**: Binary search requires a bit more complex implementation than linear search, making it less straightforward for those not familiar with the algorithm.
- **Limited Applicability:** It is not suitable for unsorted data or when the data structure is frequently modified, as it necessitates re-sorting after each modification.

Demerits of Binary Search

- **Space Complexity:** Binary search requires additional memory for recursive function calls, which can impact the space complexity in some situations.
- **Ineffective for Duplicate Elements**: Binary search may not handle cases where duplicate elements exist in the data, as it may not always locate the desired instance.

List Sorting

- List sorting is the process of rearranging the elements of a list or array in a specific order, typically in ascending or descending order.
- Sorting is a fundamental operation in computer science and programming, and it plays a crucial role in various applications and data manipulation tasks.
- Sorting allows data to be organized, making it easier to search, analyze, and present.

Key concepts in List sorting

- **Ordering:** Sorting arranges elements in a specific order based on a defined criteria. The most common criteria are numerical order (ascending or descending) and lexicographical order for strings.
- **Stability:** A stable sorting algorithm preserves the relative order of elements with equal values. In other words, if two elements have the same value, their original order is maintained in the sorted list.

Key concepts in List sorting

- **Comparison-Based vs. Non-Comparison-Based:** Many sorting algorithms compare elements to determine their order. Comparison-based algorithms have a time complexity of at least O(n log n) in the worst case.
- **Non-comparison-based algorithms**, like counting sort or radix sort, have linear time complexities.
- **In-Place vs. Not In-Place:** Sorting algorithms can be categorized as in-place or not in-place. In-place sorting algorithms modify the original list without using additional memory, while not in-place algorithms may require extra memory to store intermediate results.

Applications of List Sorting

- Sorting is a fundamental operation in computer science and has numerous applications, including:
- Searching: Sorted data allows for efficient search operations using algorithms like binary search.
- Data Analysis: Sorted data simplifies data analysis and visualization.
- Database Queries: Sorting is a crucial component of database systems to retrieve data in a specified order.

Applications of List Sorting+

- Rankings: Sorting is used in ranking algorithms for various purposes, such as search engine results and sports rankings.
- Data Presentation: Sorted data provides a more organized and user-friendly presentation of information.

Common Sorting Algorithms

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass-through is repeated until no more swaps are needed.
- Bubble Sort is one of the most straightforward sorting algorithms, but it is relatively inefficient for large datasets when compared to more advanced algorithms like Quick Sort or Merge Sort. Nonetheless, it's a useful algorithm for educational purposes and for sorting small datasets.
- Let us now look at bubble sort in details

Bubble Sort Algorithm:

- Loop: Start with the first element (index 0) and iterate through the list to the last element.
- Comparison: Compare each element with the element next to it. If the current element is greater than the next element, swap them.
- Passes: Repeat the above steps for each pass. In each pass, the largest unsorted element will 'bubble up' to its correct position at the end of the list.
- Termination: Continue these passes until no more swaps are needed. This means the list is sorted.

Bubble Sort Algorithm:

- Start at the beginning of the list.
- Compare the first two elements. If the first element is greater than the second, swap them.
- Move to the next pair of elements and repeat step 2 until you reach the end of the list. This guarantees that the largest element in the list "bubbles up" to the end of the list.
- Repeat steps 1-3 for the unsorted portion of the list (everything before the last sorted element).
- Continue this process until the entire list is sorted.

Bubble Sort Algorithm example1

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all elements in the list
    for i in range(n):
        # Flag to optimize by checking if any swaps occur in a pass
        swapped = False
        # Last i elements are already in place, so no need to check them
        for j in range(0, n-i-1):
            # If the element found is greater than the next element, swap them
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]  # Swap the elements
                swapped = True
        # If no two elements were swapped in this pass, the list is already sorted
        if not swapped:
            break

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(my_list)
print("Sorted list:", my_list)
```

Output: Sorted list: [11, 12, 22, 25, 34, 64, 90]

Explanation for the above code

- We start by defining a function bubble_sort that takes a list arr as its argument.
- We use two nested loops. The outer loop iterates through each element in the list, and the inner loop compares adjacent elements and swaps them if they are out of order. This process continues for each pass through the list.
- We use a swapped flag to optimize the algorithm. If no swaps occur in a pass, it means the list is already sorted, and we can break out of the loop early.
- The outer loop continues until no swaps occur, indicating that the list is sorted

Implementation of Bubble Sort Algorithm

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all elements in the list
    for i in range(n):
        # Flag to optimize by checking if any swaps occur in a pass
        swapped = False
        # Last i elements are already in place, so no need to check them
        for j in range(0, n-i-1):
            # If the element found is greater than the next element, swap them
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]  # Swap the elements
                swapped = True
        # If no two elements were swapped in this pass, the list is already sorted
        if not swapped:
            break

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(my_list)
print("Sorted list:", my_list)
```

Output: [11, 12, 22, 25, 64]

Analysis of Bubble Sort

- Time Complexity: The worst-case time complexity of Bubble Sort is O(n^2), making it inefficient for large datasets. However, its best-case time complexity is O(n) when the list is already sorted.
- Space Complexity: Bubble Sort is an in-place sorting algorithm, so it uses only a constant amount of extra space.
- Stability: Bubble Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements.
- While Bubble Sort is not typically used in practice for large datasets due to its inefficiency, it is easy to understand and can be useful for educational purposes. For larger datasets, more efficient sorting algorithms like Quick Sort, Merge Sort, or Python's built-in sorting functions should be preferred.

Selection Sort

- Selection Sort: Another straightforward algorithm that divides the list into a sorted and an unsorted part. It repeatedly selects the minimum element from the unsorted part and moves it to the sorted part.

Selection Sort Algorithm:

- Find the Minimum: Start with the first element of the list and assume it's the minimum. Compare it with each element in the unsorted part of the list to find the actual minimum element.

Selection Sort Algorithm+

- Swap: Once the minimum element in the unsorted part is found, swap it with the first element in the unsorted part.
- Expand the Sorted Part: The minimum element is now considered sorted. Repeat steps 1 and 2 for the next unsorted element (the element immediately following the previously sorted part).
- Termination: Continue this process until the entire list is sorted.

Python Implementation of Selection Sort

```
1.  def selection_sort(arr):
2.      n = len(arr)
3.      for i in range(n):
4.          # Assume the current element is the minimum
5.          min_idx = i
6.          # Find the minimum element in the unsorted part
7.          for j in range(i + 1, n):
8.              if arr[j] < arr[min_idx]:
9.                  min_idx = j
10.         # Swap the minimum element with the first unsorted element
11.         arr[i], arr[min_idx] = arr[min_idx], arr[i]
12. # Example usage:
13. my_list = [64, 25, 12, 22, 11]
14. selection_sort(my_list)
15. print(my_list)  # Output: [11, 12, 22, 25, 64]
```

Analysis of Selection Sort

- Time Complexity: The time complexity of Selection Sort is $O(n^2)$ in the worst case, making it inefficient for large datasets. Regardless of the input data, it performs the same number of comparisons and swaps.
- Space Complexity: Selection Sort is an in-place sorting algorithm, meaning it doesn't require additional memory other than a few variables for swapping.
- Stability: Selection Sort is not a stable sorting algorithm. It may change the order of equal elements.

Insertion Sort:

- Insertion Sort: An algorithm that builds the sorted portion of the list one item at a time by comparing each element and inserting it in its correct position.y

Insertion Sort Algorithm

- Starting with the Second Element: Begin with the second element (index 1) because a single-element list is already considered sorted.
- Comparison and Insertion: Compare the current element with the one before it. If the current element is smaller, move the previous element one position forward, creating space for the current element.
- Repeat: Continue this process for the rest of the elements, gradually expanding the sorted portion of the list.
- Termination: When all elements are processed, the list is fully sorted.

Python Implementation of Insertion Sort

```
1. def insertion_sort(arr):
2.     for i in range(1, len(arr)):
3.         key = arr[i]  # Current element to be inserted
4.         j = i - 1  # Index of the previous element
5.         # Compare key with each element before it and move greater elements
   one position ahead
6.         while j >= 0 and key < arr[j]:
7.             arr[j + 1] = arr[j]
8.             j -= 1
9.         arr[j + 1] = key  # Insert the current element in its correct position
10. # Example usage:
11. my_list = [45, 23, 87, 12, 32, 4]
12. insertion_sort(my_list)
13. print(my_list)  # Output: [11, 12, 22, 25, 64]
```

Output: [4, 12, 23, 32, 45, 87]

Python Implementation of Insertion Sort+

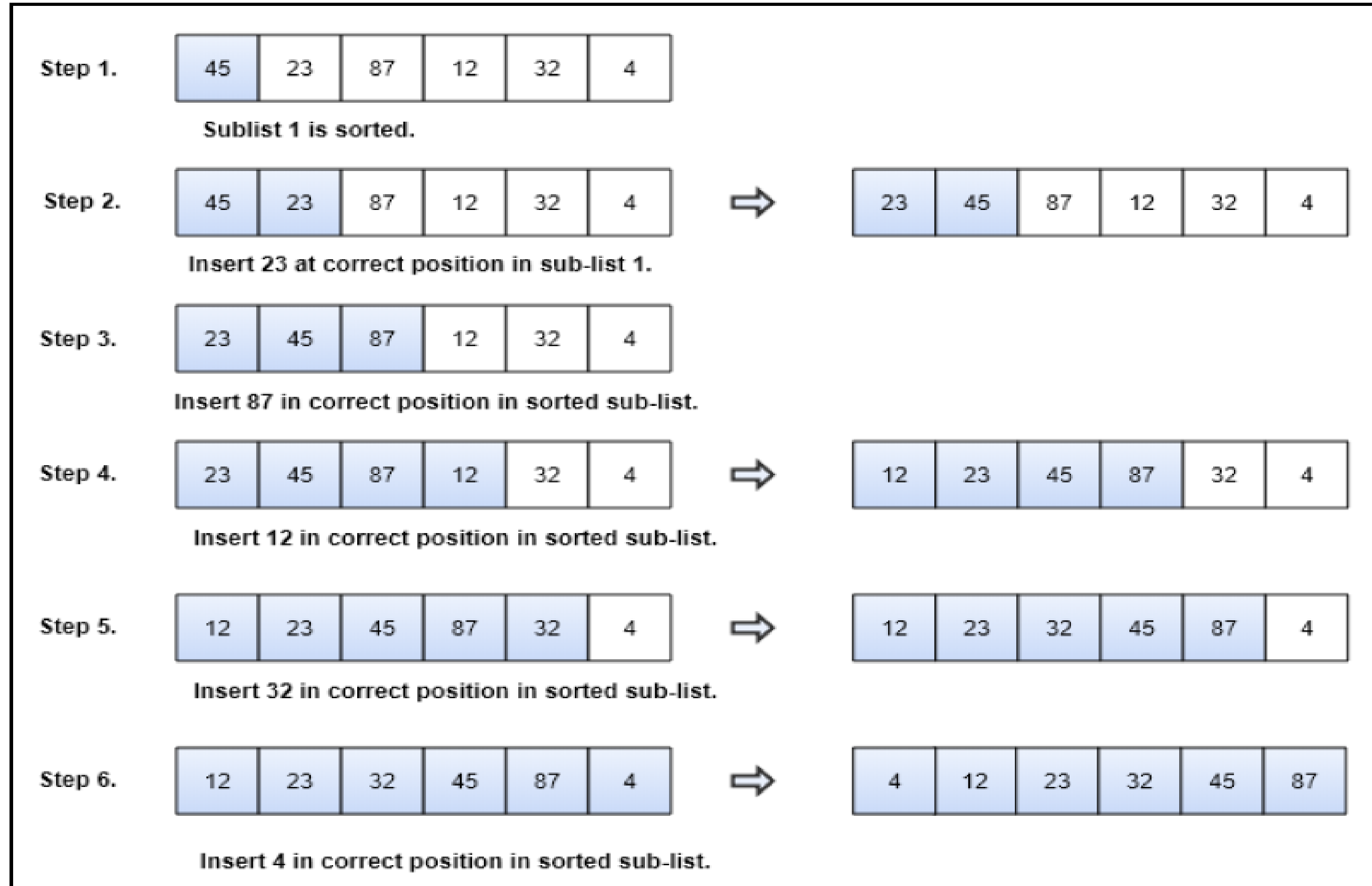

Fig . 2 illustrates the insertion sort algorithm[2]

Analysis of Insertion Sort

1. Time Complexity: The average and worst-case time complexity of Insertion Sort is $O(n^2)$, which makes it less efficient than more advanced sorting algorithms like Merge Sort or Quick Sort. However, for small datasets, Insertion Sort can be faster than other algorithms due to its lower overhead.
2. Space Complexity: Insertion Sort is an in-place sorting algorithm, meaning it doesn't use additional memory.
3. Stability: Insertion Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements.

Quick Sort

- Quick Sort: A highly efficient, divide-and-conquer sorting algorithm that selects a pivot element and partitions the list into elements less than the pivot and elements greater than the pivot. It then recursively sorts the two partitions.
- Quick Sort is known for its average and best-case performance, with an average time complexity of O(n log n), making it a popular choice for sorting large datasets.

Quick Sort Algorithm+

- Choose a Pivot: Select a pivot element from the list. The choice of the pivot can significantly affect the algorithm's performance.
- Partitioning: Rearrange the list so that all elements less than the pivot come before it, and all elements greater than the pivot come after it. The pivot is now in its sorted position.
- Recursive Sorting: Recursively apply the Quick Sort algorithm to the sub-arrays on the left and right of the pivot element.
- Termination: The process continues until the entire list is sorted. Sub-arrays with one or zero elements are considered sorted.

Python Implementation of Quick Sort

```
1.  def quick_sort(arr):
2.      if len(arr) <= 1:
3.          return arr  # Base case: lists with one or zero elements are already sorted
4.      else:
5.          pivot = arr[0]  # Choose the pivot (can be chosen differently)
6.          less_than_pivot = [x for x in arr[1:] if x <= pivot]
7.          greater_than_pivot = [x for x in arr[1:] if x > pivot]
8.          return quick_sort(less_than_pivot) + [pivot]+ quick_sort(greater_than_pivot)
9.  # Example usage:
10. my_list = [38, 27, 43, 3, 9, 82, 10]
11. sorted_list = quick_sort(my_list)
12. print(sorted_list)
13. # Output: [3, 9, 10, 27, 38, 43, 82]
```

Analysis of Quick Sort:

- Time Complexity: On average, Quick Sort has a time complexity of O(n log n), which makes it one of the fastest sorting algorithms for large datasets. However, in the worst case, it can have a time complexity of O(n^2) if the pivot choice is consistently poor.
- Space Complexity: Quick Sort is an in-place sorting algorithm and requires a small amount of additional memory for recursion. Its space complexity is generally O(log n) for the average case.
- Stability: Quick Sort is not a stable sorting algorithm. The relative order of equal elements may change after sorting.
- Quick Sort is a versatile and efficient sorting algorithm that is often preferred for large datasets due to its speed. However, choosing the pivot element can be critical for optimal performance, and in practice, it is often randomized to mitigate the worst-case time complexity.

Other sorting algorithms

- **Merge Sort**: Another efficient divide-and-conquer algorithm that divides the list into two halves, recursively sorts them, and then merges the sorted halves.
- Merge Sort is a popular and efficient comparison-based sorting algorithm known for its stability and guaranteed O(n log n) time complexity.
- It is a divide-and-conquer algorithm that divides an unsorted list into smaller sublists, recursively sorts these sublists, and then merges them to create a single, sorted list.
- Heap Sort: A comparison-based sorting algorithm that builds a max-heap or min-heap data structure and repeatedly extracts the maximum or minimum element.

Summary of the week 10

- This week, we delved into the interesting world of list searching and sorting. Our exploration covered various techniques for efficiently searching through lists, providing us with essential tools for data retrieval and analysis.
- Some of the methods we discussed included the membership operator, linear search and the widely used binary search algorithm, among others.
- Additionally, we embarked on a comprehensive study of sorting algorithms, a fundamental component of data manipulation and organization. The sorting algorithms we covered included the selection sort, bubble sort, insertion sort, merge sort, and several more.

Summary of the week 10+

- These algorithms empower us to arrange data in a systematic and meaningful manner, enabling us to draw insights, make sense of information, and streamline our computational processes.
- By understanding these search and sorting techniques, we've equipped ourselves with valuable skills to handle diverse data-related challenges and enhance our problem-solving abilities. This knowledge is not only beneficial in programming and computer science but also applicable in various real-world scenarios where data plays a pivotal role.
- Next week we shall explore object oriented programming which one of the most common and interesting programming paradigm.
- I hope that you will follow along till the end.

Reference

[1] Necaise, R. D. (2011). *Data Structures and Algorithms Using Python* (p. 131). JOHN WILEY & SONS, INC.

[2] Agarwal, B., & Agarwal, B. (2018). *Hands-On Data Structures and Algorithms with Python* (2nd ed., p. 252). Packt Publishing.