

# 3

## 003 - Searching and Sorting

### SEARCHING:

Searching is the process of finding a particular piece of data within a data structure.

#### :: Linear Search:

The linear search algorithm is given a list of values and a key. It returns the first index of where the key is found or -1 if the key is not part of the list. We use -1 to indicate the state of not finding the value because 0 is a perfectly valid index into the list.





The code for this is pretty straight forward. Start at the beginning of the list and search until you either find the item or you reach the end of the list.

```
def linear_search(my_list, key):  
    for i in range(0, len(my_list)):  
        if my_list[i] == key:  
            return i  
  
    return -1
```

- Time Complexity:  $O(n)$
- Linear Time Complexity

## Binary Search:

The binary search algorithm goes like this: *Track the range of possible indexes by storing the first/last index where key may be found* initially this is 0 and (length of list) - 1. *Calculate the mid point index between those first/last indexes* look at the value at the middle element *if we found it we are done* if the key is smaller than middle element, then key can only be found between first index and element before middle element. Thus, set last index to middle index - 1. \* if key is greater than middle element then key can only be found after that middle element, thus set first index to middle index + 1.

```
Python   Copy  Caption 
```

```
def binarySearch(arr, key):
    # Calculate the start and end index of the array
    start = 0
    end = len(arr) - 1

    # start should always be less than the end
    while (start <= end):
        # Calculate the middle index
        middle = start + ((int)((end - start) / 2))
        # Check whether the element in the middle is the key
        if arr[middle] == key:
            return middle
        # Check whether middle element is greater than key
        elif arr[middle] > key:
            # The element has to be towards the left
            end = middle - 1
        # Check whether the middle element is smaller than the key
        elif arr[middle] < key:
            # The element has to be on the right of the middle
            start = middle + 1
    return -1

print(binarySearch([100, 200, 300, 400, 500, 600, 700, 800], 700))
```

- To perform binary search:
  - The list must be sorted. This allows us to split the array into two pieces, one where the key might be found and the other where the key definitely can't be found.
  - The second requirement is the list must allow fast random access (ie be array-like). That is getting to any element of the list takes the same amount of time. In the list from the C++ standard library this is not the case.
- Time Complexity:  $O(\log n)$

## SORTING:

Sorting is the process of taking a set of data and creating an ordering based on some criteria. The criteria must allow for items to be compared and used to determine what should come first and second.

### Bubble Sort:

**Bubble sort is called bubble sort because the algorithm repeatedly bubbles the largest item within the list to the back/end of the list. Also known as sinking sort / exchange sort.**

1. Start with first pair of numbers
2. Compare them and if they are not correctly ordered, swap them
3. Go through every pair in list doing the above 2 steps
4. Repeat the above 3 steps  $n-1$  times (ie go through entire array  $n-1$  times) and you will sort the array

```
def bubbleSort(arr):
    for i in range(0, len(arr)):
        for j in range(0, (len(arr) - i - 1)):
            if arr[j] > arr[j + 1]:
                temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
    return arr

print(bubbleSort([5, 2, 1, 4, 3, 7, 9, 5, 6, 9, 7, 8, 9, 4, 2, 3, 4, 5]))
```

- **Space Complexity:**  $O(1)$  [Constant inplace sorting algorithm]
- **Time Complexity:**
  - Best Case:  $O(n)$
  - Worst Case:  $O(n^2)$

## Selection Sort:

Selection sort works by selecting the smallest value out of the unsorted part of the array and placing it at the back of the sorted part of the array.

1. Find the smallest number in the between index 0 and n-1
2. Swap it with the value at index 0
3. Repeat above 2 steps each time, starting at one index higher than previous and swapping into that position

```
def selectionSort(arr):
    for i in range(0, len(arr) - 1):
        minIndex = i
        for j in range(i + 1, len(arr)):
            if (arr[j] < arr[minIndex]):
                minIndex = j
        temp = arr[i]
        arr[i] = arr[minIndex]
        arr[minIndex] = temp
    return arr

print(selectionSort([5, 2, 1, 4, 3, 7, 9, 5, 6, 9, 7, 8, 9, 4, 2, 3, 4, 5]))
```

- **Space Complexity:**  $O(1)$  [Constant inplace sorting algorithm]
- **Time Complexity:**
  - Best Case:  $O(n^2)$
  - Worst Case:  $O(n^2)$
- It performs well on small data sets.

## Insertion Sort:

Insertion sort is called insertion sort because the algorithm repeatedly inserts a value into the part of the array that is already sorted. It essentially chops the array into two pieces. The first piece is sorted, the second is not. We repeatedly take a value/number from the second piece and insert it into the already sorted first piece of the array.

1. Start with the second value in the list (note that the first piece/portion of the list contains just the first value initially.)
2. Put that value into a temporary variable. This makes it possible to move items into the spot the second value occupies at the time and this spot is now considered to be an empty spot in the sorted piece/portion of the array.
3. If, when compared with the last value in the first piece/portion of the list, the value in the temporary variable should go into the empty spot, put it in.
4. Otherwise, move the last value in the sorted piece/portion part into the empty spot.
5. Repeat steps 3 to 4 until the value in the temporary variable is placed somewhere into the first sorted piece/portion of the array.
6. Repeat steps 2 to 5 for every value in the second piece/portion/part of the array until all values are placed in the first part.

```
def insertionSort(arr):
    for i in range(0, (len(arr) - 1)):
        j = i + 1
        while j > 0:
            if (arr[j] < arr[j - 1]):
                temp = arr[j]
                arr[j] = arr[j - 1]
                arr[j - 1] = temp
                j -= 1
            else:
                break
    return arr

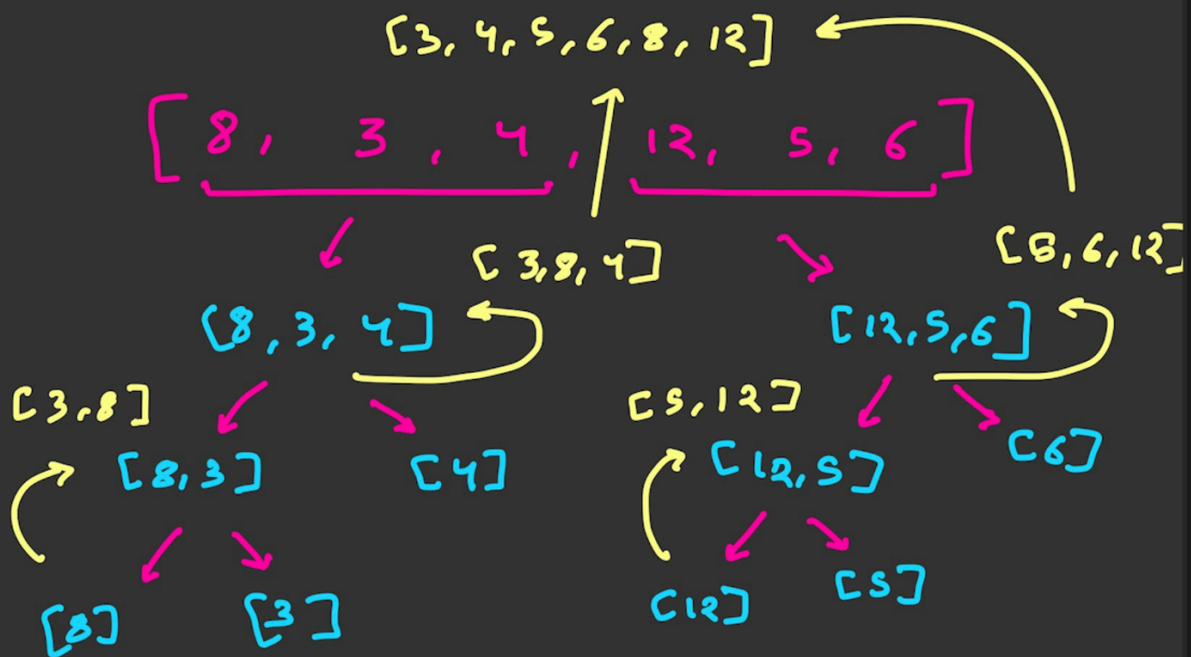
print(insertionSort([5, 2, 1, 4, 3, 7, 9, 5, 6, 9, 7, 8, 9, 4, 2, 3, 4, 5]))
```

- **Space Complexity:**  $O(1)$  [Constant inplace sorting algorithm]
- **Time Complexity:**
  - Best Case:  $O(n)$
  - Worst Case:  $O(n^2)$

## Merge Sort:

The merge sort works on the idea of merging two already sorted lists. If there existed two already sorted list, merging the two together into a single sorted list can be accomplished in  $O(n)$  time where  $n$  is the combined length of the two lists. Thus, if we are able to take our original list, split it into two pieces and then somehow get them into a sorted state, we can then use the merge algorithm to put these two pieces back together.

1. Divide the array in two parts.
2. Sort both parts using recursion.
3. Merge the sorted parts,
  - Create an empty merged list
  - Have a way to "point" at the first element of each of the two list
  - Compare the values being pointed at and pick the smaller of the two
  - Copy the smaller to the end of the merged list, and advance the "pointer" of the list that the value came from
  - Continue until one of the list is completely copied then copy over remainder of the rest of the list



:: Press 'space' for AI, '/' for commands...

### MERGE SORT:

```
def mergeSort(arr):
    if (len(arr) == 1):
        return arr

    mid = int(len(arr) / 2)

    left = mergeSort(arr[0:mid])
    right = mergeSort(arr[mid:len(arr)])
    return merge(left, right)
```

```

def merge(first, second):
    mix = [0]*(len(first) + len(second))
    i, j, k = 0, 0, 0
    while (i < len(first) and j < len(second)):
        if (first[i] < second[j]):
            mix[k] = first[i]
            i += 1
        else:
            mix[k] = second[j]
            j += 1
        k += 1
    # It may be possible that one of the arrays is not complete
    # Only one of the following two loops will run
    while (i < len(first)):
        mix[k] = first[i]
        i += 1
        k += 1
    while (j < len(second)):
        mix[k] = second[j]
        j += 1
        k += 1

    return mix

print(mergeSort([5, 2, 1, 4, 3, 7, 9, 5, 6, 9, 7, 8, 9, 4, 2, 3, 4, 5]))

# OR

```



```

def merge_sort(items):
    if len(items) <= 1:
        return items

    middle_index = len(items) // 2
    left_split = items[:middle_index]
    right_split = items[middle_index:]

    left_sorted = merge_sort(left_split)
    right_sorted = merge_sort(right_split)

    return merge(left_sorted, right_sorted)

def merge(left, right):
    result = []

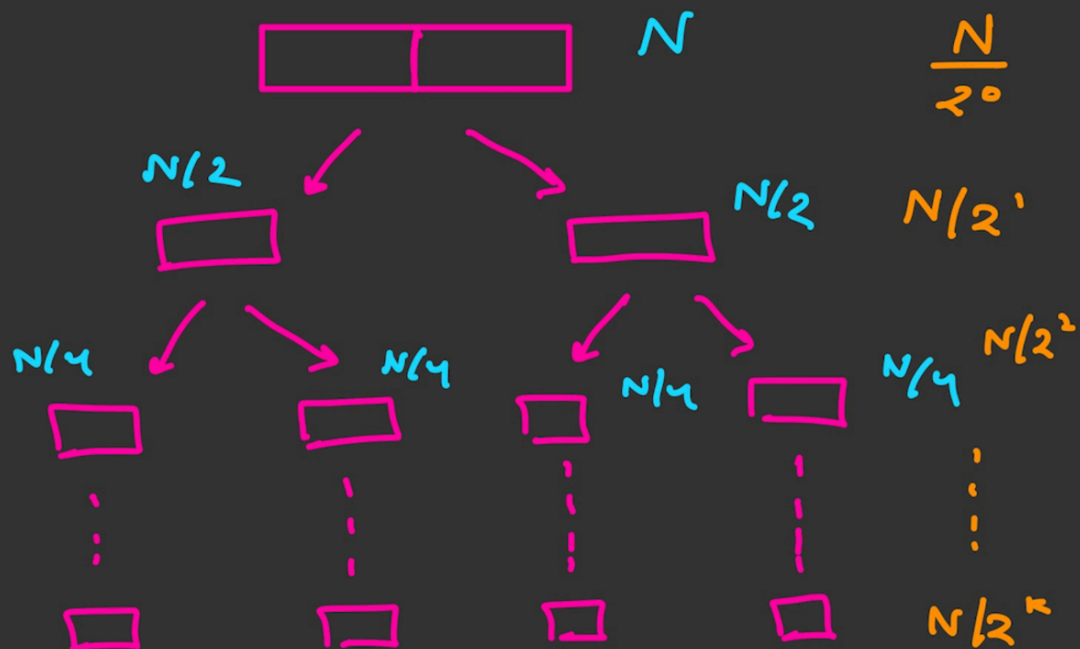
    while (left and right):
        if left[0] < right[0]:
            result.append(left[0])
            left.pop(0)
        else:
            result.append(right[0])
            right.pop(0)

    if left:
        result += left
    if right:
        result += right

    return result

```

- Space Complexity:  $O(n)$
- Time Complexity:  $O(n * \log n)$



\* At every level,  $N$  elements are being merged.

$$\Rightarrow 1 = \frac{N}{2^k}$$

$$2^k = N$$

$$k \log 2 = \log N$$

$$\Rightarrow k = \log_2 N$$

Complexity = Total Level  
× work at  
every level

TIME &  
COMPLEXITY :  $O(n \log n)$

## Quick Sort:

- Choose a Pivot in the array. After the first pass:
  - Elements smaller than the pivot would be on the left of the array
  - Elements larger than the pivot would be on the right of the array
- Now the pivot element is at its correct position in the array, however, the rest of the array is not sorted yet.
- This is where recursion comes into the picture and applies the same algorithm on the left and the right part of the array.
- This is different from insertion sort as it doesn't sort the array and break it down into further pieces if the subarray is already sorted.
- So, it keeps on breaking down the array into smaller pieces and putting the pivot at the right position.

```
def quickSort(arr, low, high):
    if (low >= high):
        return arr
    # Setting the last element as the pivot
    pivot = arr[high]

    start, end = low, high
    while (start <= end):
        # Also a reason why if its already sorted, it will not swap
        while arr[start] < pivot:
            start += 1
        while arr[end] > pivot:
            end -= 1

        if (start <= end):
            arr[start], arr[end] = arr[end], arr[start]
            start += 1
            end -= 1

    # Now my pivot is at the correct index, sort the two halves now
    quickSort(arr, low, end)
    quickSort(arr, start, high)
    return arr
```

```

unsorted_list = [3, 7, 12, 24, 36, 42]
sorted_list = quickSort(unsorted_list, 0, len(unsorted_list) - 1)
print("Sorted list:", sorted_list)

# OR

from random import randrange, shuffle

def quicksort(list, start, end):
    # this portion of list has been sorted
    if start >= end:
        return
    print("Running quicksort on {}".format(list[start: end + 1]))
    # select random element to be pivot
    pivot_idx = randrange(start, end + 1)
    pivot_element = list[pivot_idx]
    print("Selected pivot {}".format(pivot_element))
    # swap random element with last element in sub-lists
    list[end], list[pivot_idx] = list[pivot_idx], list[end]

    # tracks all elements which should be to left (lesser than) pivot
    less_than_pointer = start

    for i in range(start, end):
        # we found an element out of place
        if list[i] < pivot_element:
            # swap element to the right-most portion of lesser elements
            print("Swapping {} with {}".format(list[i], list[less_than_pointer]))
            list[i], list[less_than_pointer] = list[less_than_pointer], list[i]
            # tally that we have one more lesser element
            less_than_pointer += 1
    # move pivot element to the right-most portion of lesser elements
    list[end], list[less_than_pointer] = list[less_than_pointer], list[end]
    print("{} successfully partitioned".format(list[start: end + 1]))
    # recursively sort left and right sub-lists
    quicksort(list, start, less_than_pointer - 1)
    quicksort(list, less_than_pointer + 1, end)

```

1. The `quickSort` function takes three arguments: the input array `arr`, the starting index `low`, and the ending index `high`.
2. The first condition checks if the current subarray has one or zero elements (i.e., `low >= high`). If true, it means the subarray is already sorted or empty, so the function simply returns the array as it is.
3. The pivot element is chosen as the last element of the subarray, `arr[high]`. This is a common approach, but other pivot selection strategies are also possible.
4. The variables `start` and `end` are initialized to `low` and `high`, respectively. These variables represent the left and right pointers for partitioning the subarray.
5. The while loop (`while start <= end`) continues until the pointers cross each other, indicating that the partitioning step is complete.
6. Inside the while loop, there are two nested while loops. The first loop (`while arr[start] < pivot`) increments the `start` pointer until an element greater than or equal to the pivot is found. The second loop (`while arr[end] > pivot`) decrements the `end` pointer until an element less than or equal to the pivot is found.
7. If `start` is still less than or equal to `end` after the two nested while loops, it means there are two elements on the wrong side of the pivot, and they need to be swapped. The elements at indices `start` and `end` are swapped using a tuple assignment: `arr[start], arr[end] = arr[end], arr[start]`. Then both `start` and `end` are updated by incrementing `start` and decrementing `end` by one.
8. The while loop continues until the pointers cross each other, ensuring that all elements greater than the pivot are on the right side, and all elements less than the pivot are on the left side.
9. Once the partitioning step is complete, the pivot element is in its correct sorted position. It is placed between the two subarrays: the left subarray with elements less than the pivot and the right subarray with elements greater than the pivot.
10. The `quickSort` function is recursively called twice to sort the left and right subarrays. The first recursive call is made with the arguments `low` and `end` to sort the left subarray (elements less than the pivot). The second recursive call is made with the arguments `start` and `high` to sort the right subarray (elements greater than the pivot).
11. Finally, the sorted array is returned as the output of the function.

- **Space Complexity:**
  - **Best Case:**  $O(\log n)$ 
    - This occurs when the array is divided in almost two equal parts
  - **Worst Case:**  $O(n)$ 
    - This occurs when the pivot element is the largest or the smallest element in the array. Now, the two parts are uneven as one part is empty.
- **Time Complexity:**
  - **Best Case:**  $O(n \log n)$
  - **Worst Case:**  $O(n^2)$