Understand how Searching and Sorting algorithms work in Python

# Searching Algorithms

## Linear Search

Problem Description

Write a Python program that implements linear search. The program accepts a list and a key as input, and it finds the index of the key in the list using linear search.

What is Linear Search?

Linear search is a simple search algorithm used to find the position of a target value within a list. It sequentially checks each element until a match is found or the list is exhausted.

Linear Search Concept and Example

Here’s a step-by-step example of how linear search works in Python for the given list [5, 4, 3, 2, 1, 10, 11, 2] and the target value “1“:

* Step 1: Start with the given list: [5, 4, 3, 2, 1, 10, 11, 2] and set the target value to 1.
* Step 2: Begin at the first element of the list, which is 5.
* Step 3: Compare 5 with the target value (1). Since they don’t match, move to the next element, which is 4.
* Step 4: Compare 4 with the target value (1). Since they don’t match, move to the next element, which is 3.
* Step 5: Compare 3 with the target value (1). Since they don’t match, move to the next element, which is 2.
* Step 6: Compare 2 with the target value (1). Since they don’t match, move to the next element, which is 1.
* Step 7: Compare 1 with the target value (1). They match! Return the index of the element, which is 4.
* Step 8: The linear search is successful, and the index 4 is returned.

Linear Search Algorithm

**def** linear\_search(alist, key):

**for** i **in** range(len(alist)):

**if** alist[i] == key:

**return** i

**return** -1

This algorithm iterates through each element in the list and compares it with the key. If a match is found, it returns the index of the element. If no match is found after iterating through the entire array, it returns -1 to indicate that the key was not found.

Program/Source Code

Here is the source code of a Python program to implement linear search. The program output is shown below.

**def** linear\_search(alist, key):

"""Return index of key in alist. Return -1 if key not present."""

**for** i **in** range(len(alist)):

**if** alist[i] == key:

**return** i

**return** -1

alist = input('Enter the list of numbers: ')

alist = alist.split()

alist = [int(x) **for** x **in** alist]

key = int(input('The number to search for: '))

index = linear\_search(alist, key)

**if** index < 0:

**print**('{} was not found.'.format(key))

**else**:

**print**('{} was found at index {}.'.format(key, index))

Program Explanation

1. Create a function **linear\_search** that takes a list and key as arguments.
2. The list and key is passed to **linear\_search**.
3. If the return value is -1, the key is not found, and a message is displayed. Otherwise, the index of the found item is displayed.

Time Complexity: O(n)  
The time complexity of the linear search algorithm is O(n), where n is the size of the input list. In the worst case, the algorithm may need to iterate through all n elements of the list to find the key or determine its absence.

Space Complexity: O(1)  
The space complexity of the linear search algorithm is O(1), as it does not require any additional space that grows with the input size. The algorithm only uses a fixed amount of memory to store variables, regardless of the size of the input list.

Runtime Test Cases

**Test Case 1**: In this case, we use the linear search algorithm to find the element's position, and the elements are entered in random order, i.e. {5 4 3 2 1 10 11 2}, and the element to be searched is “1”.

Enter the list of numbers: 5 4 3 2 1 10 11 2

The number to search for: 1

1 was found at index 4.

**Test Case 2**: In this case, we use the linear search algorithm to find the position of element, and the elements are entered in random order i.e {3 8 5 6} and the element to be searched is “2”.

Enter the list of numbers: 3 8 5 6

The number to search for: 2

2 was not found.

## Binary Search (without Recursion)

Problem Description

The program takes a list and key as input and finds the index of the key in the list using binary search.

Real-World Applications

Binary search is a fundamental algorithm with many real-world applications due to its efficiency in searching sorted data. Here are some examples:

* + **Databases**. **Indexing**: Databases use binary search to quickly locate records using indexed fields. This speeds up query processing significantly.
  + **File Systems**. **File Search**: Operating systems use binary search to find files in sorted directories, improving file access times.
  + **Libraries and APIs**. **Standard Libraries**: Many programming languages' standard libraries implement binary search for searching sorted collections, such as arrays or lists.
  + **Gaming**. **Collision Detection**: In some games, binary search is used to quickly detect collisions or interactions between objects sorted by their positions.
  + **Networking**. **Routing Tables**: Binary search helps in efficiently searching routing tables to determine the best path for data packets.
  + **Finance**. **Stock Market Analysis**: Binary search can be used to quickly find specific stock prices or historical data in sorted datasets.
  + **Text Processing**. **Dictionary Lookup**: Spell checkers and text editors use binary search to quickly find words in a sorted dictionary.
  + **E-commerce**. **Product Search**: Online stores use binary search to quickly find products in sorted catalogs, enhancing user experience.
  + **Genomics**. **DNA Sequence Search**: Researchers use binary search to find specific DNA sequences within large, sorted genomic databases.
  + **Artificial Intelligence**. **Decision Trees**: Binary search is used in decision trees to quickly navigate through sorted decision nodes.

Problem Solution

1. Create a function binary\_search that takes a list and key as arguments.
2. The variable start is set to 0 and end is set to the length of the list.
3. The variable start keeps track of the first element in the part of the list being searched while end keeps track of the element one after the end of the part being searched.
4. A while loop is created that iterates as long as start is less than end.
5. mid is calculated as the floor of the average of start and end.
6. If the element at index mid is less than key, start is set to mid + 1 and if it is more than key, end is set to mid. Otherwise, mid is returned as the index of the found element.
7. If no such item is found, -1 is returned.

Program/Source Code

Here is the source code of a Python program to implement binary search without using recursion. The program output is shown below.

**def** binary\_search(alist, key):

"""Search key in alist[start... end - 1]."""

start = 0

end = len(alist)

**while** start < end:

mid = (start + end)//2

**if** alist[mid] > key:

end = mid

**elif** alist[mid] < key:

start = mid + 1

**else**:

**return** mid

**return** -1

alist = input('Enter the sorted list of numbers: ')

alist = alist.split()

alist = [int(x) **for** x **in** alist]

key = int(input('The number to search for: '))

index = binary\_search(alist, key)

**if** index < 0:

**print**('{} was not found.'.format(key))

**else**:

**print**('{} was found at index {}.'.format(key, index))

Program Explanation

1. The user is prompted to enter a list of numbers.
2. The user is then asked to enter a key to search for.
3. The list and key is passed to binary\_search.
4. If the return value is -1, the key is not found and a message is displayed, otherwise the index of the found item is displayed.

Runtime Test Cases

Case 1:

Enter the sorted list of numbers: 3 5 10 12 15 20

The number to search for: 12

12 was found at index 3.

Case 2:

Enter the sorted list of numbers: -3 0 1 5 6 7 8

The number to search for: 2

2 was not found.

Case 3:

Enter the sorted list of numbers: 5

The number to search for: 5

5 was found at index 0.

## Binary Search (with Recursion)

Problem Description

The program takes a list and key as input and finds the index of the key in the list using binary search.

Problem Solution

1. Create a function binary\_search that takes a list and the variables start, end and key as arguments. The function searches for the key in the range [start… end – 1].
2. The base case consists of testing whether start is less than end. If not, -1 is returned.
3. mid is calculated as the floor of the average of start and end.
4. If the element at index mid is less than key, binary\_search is called again wit start=mid + 1 and if it is more than key, it is called with end=mid. Otherwise, mid is returned as the index of the found element.

Program/Source Code

Here is the source code of a Python program to implement binary search using recursion. The program output is shown below.

**def** binary\_search(alist, start, end, key):

"""Search key in alist[start... end - 1]."""

**if** **not** start < end:

**return** -1

mid = (start + end)//2

**if** alist[mid] < key:

**return** binary\_search(alist, mid + 1, end, key)

**elif** alist[mid] > key:

**return** binary\_search(alist, start, mid, key)

**else**:

**return** mid

alist = input('Enter the sorted list of numbers: ')

alist = alist.split()

alist = [int(x) **for** x **in** alist]

key = int(input('The number to search for: '))

index = binary\_search(alist, 0, len(alist), key)

**if** index < 0:

**print**('{} was not found.'.format(key))

**else**:

**print**('{} was found at index {}.'.format(key, index))

Program Explanation

1. The user is prompted to enter a list of numbers.
2. The user is then asked to enter a key to search for.
3. The list and key is passed to binary\_search with start=0 and end=length of the list.
4. If the return value is -1, the key is not found and a message is displayed, otherwise the index of the found item is displayed.

Runtime Test Cases

Case 1:

Enter the sorted list of numbers: 4 5 6 7 8 9 10

The number to search for: 9

9 was found at index 5.

Case 2:

Enter the sorted list of numbers: 3 4 5 10

The number to search for: 8

8 was not found.

Case 3:

Enter the sorted list of numbers: 7

The number to search for: 7

7 was found at index 0.

# Sorting Algorithms

## Bubble Sort

What is Bubble Sort?  
**Bubble Sort** in Python is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order, gradually moving larger elements towards the end of the list.

Problem Description

Write a Python program to implement bubble sort.

Bubble Sort Algorithm using Python

**def** bubble\_sort(lst):

n = len(lst)

**for** i **in** range(n-1):

**for** j **in** range(n-1-i):

**if** lst[j] > lst[j+1]:

lst[j], lst[j+1] = lst[j+1], lst[j]

In the code above, the **bubble\_sort** function takes a list (lst) as input and performs the Bubble Sort algorithm. It iterates through the list multiple times, comparing adjacent elements and swapping them if they are in the wrong order. This process is repeated until the list is fully sorted.

Bubble Sort Example:

Here’s an example of the Bubble Sort algorithm applied to the list “**4 2 38 10 5**“:

* **Step 1:** Start with the given list [4, 2, 38, 10, 5].
* **Step 2:** Compare the first pair of adjacent elements (4 and 2). Since 4 is greater than 2, swap them.  
  List becomes [2, 4, 38, 10, 5].
* **Step 3:** Continue comparing adjacent elements and swapping if necessary.  
  [2, 4, 10, 38, 5]  
  [2, 4, 10, 5, 38]
* **Step 4:** Repeat the process until the list is fully sorted.  
  [2, 4, 10, 5, 38]  
  [2, 4, 5, 10, 38]  
  [2, 4, 5, 10, 38] (sorted)

The final sorted list is **[2, 4, 5, 10, 38]**.

Program/Source Code

Here is the source code of a Python program to implement bubble sort. The program output is shown below.

**def** bubble\_sort(alist):

**for** i **in** range(len(alist) - 1, 0, -1):

no\_swap = True

**for** j **in** range(0, i):

**if** alist[j + 1] < alist[j]:

alist[j], alist[j + 1] = alist[j + 1], alist[j]

no\_swap = False

**if** no\_swap:

**return**

alist = input('Enter the list of numbers: ').split()

alist = [int(x) **for** x **in** alist]

bubble\_sort(alist)

**print**('Sorted list: ', end='')

**print**(alist)

Program Explanation

1. Define the **bubble\_sort** function that takes a list (**alist**) as input.
2. Iterate through the list in reverse order, starting from the last index **(len(alist) – 1)** and ending at the second index (0) using a step size of -1.
3. Set a flag **no\_swap** to True, indicating no swaps have occurred yet.
4. Enter a nested loop that iterates from the first index (0) to the current outer loop index (**i**).
5. Compare adjacent elements and swap them if the element on the right is smaller than the element on the left.
6. Set the **no\_swap** flag to False if a swap occurs, indicating that the list is not yet fully sorted.
7. Check if the **no\_swap** flag is still True after the inner loop completes. If it is, return from the function as the list is already sorted.
8. Outside the function, prompt the user to enter a list of numbers, which are stored as a string and split into individual elements.
9. Convert the elements from strings to integers using a list comprehension.
10. Call the **bubble\_sort** function to sort the list using the Bubble Sort algorithm.
11. Print the sorted list.

Time Complexity of Bubble Sort Program:

* **Best case:** O(n)
* **Average case:** O(n2)
* **Worst case:** O(n2)

The time complexity of the program is O(n2) in both average and worst cases, where n is the number of elements in the input list. The best case time complexity is O(n) when the list is already sorted.

Space Complexity: O(1)

The space complexity of the program is O(1) since it uses a constant amount of additional space.

Runtime Test Cases

**Testcase 1:** In this case, we are entering the numbers “4, 2, 38, 10, and 5” as input to sort them using bubble sort in ascending order.

Enter the list of numbers: 4 2 38 10 5

Sorted list: [2, 4, 5, 10, 38]

**Testcase 2:** In this case, we are entering the numbers “5, 4, 3, 2, and 1” as input to sort them using bubble sort in ascending order.

Enter the list of numbers: 5 4 3 2 1

Sorted list: [1, 2, 3, 4, 5]

**Testcase 3:** In this case, we are entering the numbers “7, 3, 1, -5, 2, and 10” as input to sort them using bubble sort in ascending order.

Enter the list of numbers: 7 3 1 -5 2 10

Sorted list: [-5, 1, 2, 3, 7, 10]

## Selection Sort

What is Selection Sort?

**Selection Sort** in Python is a sorting algorithm that iterates through a list, finds the minimum element, and swaps it with the current position. This process is repeated until the list is sorted.

Problem Description

Write a program that sorts a list by implementing selection sort.

Selection Sort Algorithm

**def** selection\_sort(arr):

n = len(arr)

**for** i **in** range(n-1):

min\_index = i

**for** j **in** range(i+1, n):

**if** arr[j] < arr[min\_index]:

min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

In this code, **arr** represents the input list to be sorted. The algorithm iterates through the list and finds the minimum element in each iteration, swapping it with the current position. This process continues until the list is sorted in ascending order.

Selection Sort Example

Here’s a step-by-step explanation of how the Selection Sort algorithm sorts the list “3 1 4 5 2 6” in ascending order:

* Start with the given list: “3 1 4 5 2 6”.
* Find the minimum element in the list, which is 1.
* Swap the minimum element (1) with the first element (3), resulting in “1 3 4 5 2 6”.
* Move to the next position and find the minimum element, which is 2.
* Swap the minimum element (2) with the second element (3), resulting in “1 2 4 5 3 6”.
* Repeat this process until the list is sorted completely.
* After applying the Selection Sort algorithm, the sorted list will be “1 2 3 4 5 6”.

**Selection Sort Algorithm Implementation**

Here is the source code of a Python program to implement selection sort. The program output is shown below.

**def** selection\_sort(alist):

**for** i **in** range(0, len(alist) - 1):

smallest = i

**for** j **in** range(i + 1, len(alist)):

**if** alist[j] < alist[smallest]:

smallest = j

alist[i], alist[smallest] = alist[smallest], alist[i]

alist = input('Enter the list of numbers: ').split()

alist = [int(x) **for** x **in** alist]

selection\_sort(alist)

**print**('Sorted list: ', end='')

**print**(alist)

Program Explanation

1. Create a function **selection\_sort** that takes a list as argument.
2. Inside the function create a loop with a loop variable i that counts from 0 to the length of the list – 1.
3. Create a variable smallest with initial value i.
4. Create an inner loop with a loop variable j that counts from i + 1 up to the length of the list – 1.
5. Inside the inner loop, if the elements at index j is smaller than the element at index smallest, then set smallest equal to j.
6. After the inner loop finishes, swap the elements at indexes i and smallest.
7. The sorted list is displayed.

Time Complexity of Selection Sort Algorithm:

* **Best Case:** O(n2)
* **Worst Case:** O(n2)
* **Average Case:** O(n2)

The time complexity of the Selection Sort algorithm is O(n2) in all cases, where “n” is the number of elements in the list. This is because it involves nested loops where each element is compared with all the remaining elements.

Space Complexity: O(1)

The space complexity is O(1) as the algorithm operates on the input list in-place without requiring any additional space that grows with the input size.

Runtime Test Cases

**Testcase 1:** Here, the elements are entered in random order.

Enter the list of numbers: 3 1 4 5 2 6

Sorted list: [1, 2, 3, 4, 5, 6]

**Testcase 2:** In this case, the user enters the list of numbers as “2 10 5 38 1 7”, and the program will output:

Enter the list of numbers: 2 10 5 38 1 7

Sorted list: [1, 2, 5, 7, 10, 38]

**Testcase 3:** In this scenario, the elements are entered in reverse sorted order.

Enter the list of numbers: 5 3 2 1 0

Sorted list: [0, 1, 2, 3, 5]

## Insertion Sort

What is Insertion Sort?

**Insertion Sort** in Python is basically the insertion of an element from a random set of numbers, to its correct position where it should actually be, by shifting the other elements if required.

Problem Description

Write a Python program to sort a list by using insertion sort.

Insertion Sort Algorithm

function insertion\_sort(arr):

**for** i **in** range(1, length(arr)):

key = arr[i]

j = i - 1

**while** j >= 0 **and** arr[j] > key:

arr[j + 1] = arr[j]

j = j - 1

arr[j + 1] = key

The **insertion\_sort** function takes an array as input and performs the insertion sort algorithm. It starts iterating from the second element **(i = 1)** and compares it with the elements in the sorted portion **(j = i – 1)** from right to left. If an element is greater than the key, it is shifted one position to the right. Finally, the key is inserted into its correct position within the sorted portion.

Insertion Sort Working Example

Here’s a step-by-step example of how the Insertion Sort algorithm works on the list [2, 4, 1, 5, 8, 0]:

* Step 1: Initial state: [2, 4, 1, 5, 8, 0]
* Step 2: Start with the second element (4) and compare it with the first element (2). Since 4 is greater, no swap is needed.  
  State: [2, 4, 1, 5, 8, 0]
* Step 3: Move to the next unsorted element (1). Compare it with the sorted portion (2, 4) and insert it in the correct position.  
  State: [1, 2, 4, 5, 8, 0]
* Step 4: Repeat the process for the next unsorted element (5).  
  State: [1, 2, 4, 5, 8, 0]
* Step 5: Repeat the process for the next unsorted element (8).  
  State: [1, 2, 4, 5, 8, 0]
* Step 6: Repeat the process for the next unsorted element (0). Compare it with the sorted portion (1, 2, 4, 5, 8) and insert it in the correct position.  
  State: [0, 1, 2, 4, 5, 8]
* Step 7: The entire list is now sorted: [0, 1, 2, 4, 5, 8].

By iteratively considering one element at a time and inserting it into the correct position within the sorted portion of the list, the Insertion Sort algorithm gradually builds a sorted list.

Program/Source Code

Here is the source code of a Python program to implement insertion sort. The program output is shown below.

**def** insertion\_sort(alist):

**for** i **in** range(1, len(alist)):

temp = alist[i]

j = i - 1

**while** (j >= 0 **and** temp < alist[j]):

alist[j + 1] = alist[j]

j = j - 1

alist[j + 1] = temp

alist = input('Enter the list of numbers: ').split()

alist = [int(x) **for** x **in** alist]

insertion\_sort(alist)

**print**('Sorted list: ', end='')

**print**(alist)

Program Explanation

1. Define the function **insertion\_sort** that takes a list (alist) as input.
2. Start a loop that iterates from the second element **(i = 1)** to the end of the list (len(alist)).
3. Assign the value of the current element to a temporary variable (temp).
4. Set the initial index for comparison as the previous element **(j = i – 1)**.
5. Enter a nested loop that continues while the index is greater than or equal to 0 and the temporary value is less than the element at the current index **(temp < alist[j])**.
6. Within the nested loop, shift the element at the current index one position to the right **(alist[j + 1] = alist[j])**.
7. Decrement the index by 1 **(j = j – 1)**.
8. After exiting the nested loop, insert the temporary value into its correct position in the sorted portion of the list **(alist[j + 1] = temp)**.
9. Outside the function, prompt the user to enter a list of numbers, which are stored as a string and split into individual elements.
10. Convert the elements from strings to integers using a list comprehension.
11. Call the **insertion\_sort** function, passing the converted list as an argument.
12. Print the sorted list.

Time Complexity: O(n2)

The time complexity of the code is O(n2), where n is the number of elements in the input list.

Space Complexity: O(1)

The space complexity is O(1) since it uses a constant amount of additional space.

Runtime Test Cases

**Testcase 1**: Here, the elements are entered in random order.

Enter the list of numbers: 2 4 1 5 8 0

Sorted list: [0, 1, 2, 4, 5, 8]

**Testcase 2:** In this scenario, the elements are entered in reverse sorted order.

Enter the list of numbers: 5 4 3 2 0 -1

Sorted list: [-1, 0, 2, 3, 4, 5]

**Testcase 3:** In this case, the user enters the list of numbers as “3 4 1 4 5 0 7”, and the program will output:

Enter the list of numbers: 3 4 1 4 5 0 7

Sorted list: [0, 1, 3, 4, 4, 5, 7]

## Merge Sort

What is Merge Sort?

Merge Sort in Python is a recursive sorting algorithm that divides the input list into smaller halves, sorts them separately, and then merges them back together to produce a sorted list.

Problem Description

Write a Python program that sorts a list by using merge sort.

Merge Sort Algorithm

**def** merge\_sort(arr, start, end):

**if** start >= end:

**return**

mid = (start + end) // 2

merge\_sort(arr, start, mid)

merge\_sort(arr, mid + 1, end)

merge(arr, start, end, mid)

The **merge\_sort** function takes an array, a start index, and an end index as input. It checks if the start index is greater than or equal to the end index, and if so, it returns. Otherwise, it calculates the middle index and recursively calls **merge\_sort** on the left and right halves of the array. Finally, it merges the divided halves using the merge function.

Merge Sort Algorithm Working:

Here’s a step-by-step example of how the Merge Sort algorithm would sort the list “3 1 5 8 2 5 1 3”:

* Initial list: [3, 1, 5, 8, 2, 5, 1, 3]
* Split the list into halves: [3, 1, 5, 8] and [2, 5, 1, 3]
* Recursively split the left half: [3, 1] and [5, 8]
* Recursively split the right half: [2, 5] and [1, 3]
* Merge the divided halves of the right side: [1, 2, 3, 5]
* Merge the divided halves of the left side: [1, 3, 5, 8]
* Merge the sorted halves of the left and right sides: [1, 1, 2, 3, 5, 5, 8]
* The final sorted list: [1, 1, 2, 3, 5, 5, 8]

The **Merge Sort algorithm** repeatedly divides the list into smaller halves, sorts them individually, and then merges them back together in a sorted manner. This process continues until the entire list is sorted.

Implementation of Merge Sort Algorithm

Here is the source code of a Python program to implement merge sort. The program output is shown below.

**def** merge\_sort(alist, start, end):

'''Sorts the list from indexes start to end - 1 inclusive.'''

**if** end - start > 1:

mid = (start + end)//2

merge\_sort(alist, start, mid)

merge\_sort(alist, mid, end)

merge\_list(alist, start, mid, end)

**def** merge\_list(alist, start, mid, end):

left = alist[start:mid]

right = alist[mid:end]

k = start

i = 0

j = 0

**while** (start + i < mid **and** mid + j < end):

**if** (left[i] <= right[j]):

alist[k] = left[i]

i = i + 1

**else**:

alist[k] = right[j]

j = j + 1

k = k + 1

**if** start + i < mid:

**while** k < end:

alist[k] = left[i]

i = i + 1

k = k + 1

**else**:

**while** k < end:

alist[k] = right[j]

j = j + 1

k = k + 1

alist = input('Enter the list of numbers: ').split()

alist = [int(x) **for** x **in** alist]

merge\_sort(alist, 0, len(alist))

**print**('Sorted list: ', end='')

**print**(alist)

Program Explanation

1. Create a function **merge\_sort** that takes a list and two variables start and end as arguments.
2. The function **merge\_sort** will sort the list from indexes start to end – 1 inclusive.
3. If end – start is not greater than 1, then return.
4. Otherwise, set mid equal to the floor of **(start + end)/2**.
5. Call **merge\_sort** with the same list and with start = start and end = mid as arguments.
6. Call **merge\_sort** with the same list and with start = mid and end = end as arguments.
7. Call the function **merge\_list**, passing the list and the variables start, mid and end as arguments.
8. The function **merge\_list** takes a list and three numbers, start, mid and end as arguments and assuming the list is sorted from indexes start to mid – 1 and from mid to end – 1, merges them to create a new sorted list from indexes start to end – 1.
9. The sorted list is displayed.

Time complexity: O(n log n)

The Merge Sort algorithm has a time complexity of O(n log n) in all cases, where n is the number of elements in the list. This means the algorithm scales efficiently even for large input sizes.

Space Complexity: O(n)

The Merge Sort algorithm has a space complexity of O(n) in all cases. It requires additional memory to store temporary arrays during the merging process, with the size proportional to the input size.

Runtime Test Cases

**Testcase 1:** Here, the elements are entered in random order.

Enter the list of numbers: 3 1 5 8 2 5 1 3

Sorted list: [1, 1, 2, 3, 3, 5, 5, 8]

**Testcase 2:** Here, the elements are entered in reverse order.

Enter the list of numbers: 5 3 2 1 0

Sorted list: [0, 1, 2, 3, 5]