Searching

Searching is occasionally dependent on Sorting. We sort and store data in a particular order for finding it quickly during a search procedure. We will learn two basic searching techniques: Sequential search(searches through a sequence) and Binary search(search through a sorted random-access sequence) in this chapter.

Sequential/Linear Search

The easiest way to search for an item in a sequence is to start at the beginning and keep going through each item at a time for finding a match with the key. If the match is found, we stop and return the location/index of the item in the sequence. If we go through all the items of the list and no match is found, that means the key was not in the sequence. This process is called "Sequential/Linear Search".
For n element sequence, the worst-case requires n comparisons. The benefit of this searching technique is it can work with unsorted data and it does not require random access.

The following shows how to sequentially search for a key in a list.

Linear/Sequential search	**Output**

`def sequential_search(data, key):`
`  for index in range(len(data)):`
`  if key == data[index]:`
`  return index #index if found`
`  return -1 #sentinel if not found`	

#Key exists in the data
`data = [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]`
`sequential_search(data, 100)`
	`6`

#Key does not exist in the data `sequential_search(data, 30)`	`-1`

Binary Search

If the data in the sequence is already sorted, and the sequence supports random access (it can be accessed with an index number), then we can significantly improve the search performance by using binary search.

We're already quite familiar with this search technique — we use something similar when we look up a word in a dictionary. We see if the item is in the middle of the list. If so, we've found it. If not, we check if the item is larger or smaller than the item we are looking for. If it's larger, then we look at the right half of the list (to the right of the middle element). Or else we look at the left half (to the left of the middle element). Then we again look at the middle element of the half we have chosen and repeat the process. We continue dividing up the list and repeat the process, until either we find the item, or we run out of data to search (ie., we cannot divide the data anymore, in which case we can say the item was not in the list).

The following shows how to search for a key in a list with sorted elements.

Input: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300].

Key: 15 **(Key exists in the data)**

```
Current section: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]
================================
Left: 0 , Right: 9
Mid point: 4 , Mid value: 20
15 < 20 ||| Moving left
Current section: [-5, 5, 10, 15]
================================
Left: 0 , Right: 3
Mid point: 1 , Mid value: 5
15 > 5 ||| Moving right
Current section: [10, 15]
================================
Left: 2 , Right: 3
Mid point: 2 , Mid value: 10
15 > 10 ||| Moving right
Current section: [15] (FOUND THE KEY)
================================
Left: 3 , Right: 3 (STOP)
```

The following shows how to search for a key in a list with sorted elements.

Input: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300].

Key: 16 **(Key does not exist in the data)**

Current section: [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]
==========
Left: 0 , Right: 9
Mid point: 4 , Mid value: 20
16 < 20 ||| Moving left
Current section: [-5, 5, 10, 15]
==========
Left: 0 , Right: 3
Mid point: 1 , Mid value: 5
16 > 5 ||| Moving right
Current section: [10, 15]
==========
Left: 2 , Right: 3
Mid point: 2 , Mid value: 10
16 > 10 ||| Moving right
Current section: [15] **(KEY NOT FOUND)**
==========
Left: 3 , Right: 3 (STOP)
==========

Binary search

```
def binary_search(data, size, key):
     left = 0
     right = size - 1

     print(data)
     while left <= right:

        mid = int((left+ right)//2)
        if key == data[mid]:
              return mid
        elif key > data[mid]:
              left = mid + 1
        else:
              right = mid - 1

      return -1 #sentinel if not found

data = [-5, 5, 10, 15, 20, 50, 100, 150, 200, 300]
binary_search(data, len(data), 5)
```

Sorting

Arranging data in an ordered sequence is called "**sorting**". This order can be done in two ways: ascending and descending.

Ascending order: arranged from the smallest to the largest number.
Descending order: arranged from the largest to the smallest number.

Python has 2 built-in functions called sort() and sorted(). Besides these, we will look into two sorting algorithms named Selection sort and Bubble sort.

The sorted() function

The sorted() function makes a copy of the original list and returns the sorted list. It does not modify the original list.

Syntax of sorted() Function
`sorted(iterable, key, reverse=False)`
Parameters
Iterable: Sequence data type (string, list, tuple) or collection data type(dictionary, set) **Key(optional):** It is a function that works like a key and helps with custom sorting. The function named passed here can be built-in or custom-made. **Reverse(optional):** Its default value is False. For False, it sorts in the ascending order, and for True, it sorts in descending order. **Return Type:** Returns a sorted copy of the original list.

Example:	**Output**

#1) Ascending sort of a list without the optional reverse
 	
`numbers = [10, 20, 70, 60, 40]`
`sorted_numbers = sorted(numbers)`
`print("Sorted:", sorted_numbers)`
`print("Original:", numbers )`
	`Sorted: [10, 20, 40, 60, 70]`
	
	`Original: [10, 20, 70, 60, 40]`

#2) Ascending sort of a tuple without the optional reverse
 	
`print("Sorted:", sorted_numbers)`
`print("Original:", numbers )`
	`Sorted: [10, 20, 40, 60, 70]`
	
	`Original: (10, 20, 70, 60, 40)`

#3) Ascending sort of a String without the optional reverse. For the string, sorting is done depending on the ASCII values.
 	
`unsorted_chars = "CSE110 Rocks"`
`sorted_chars = sorted(unsorted_chars.lower(), reverse = False)`
`print("Sorted:", sorted_chars)`
`print("Original:", unsorted_chars)`
	`Sorted: [' ', '0', '1', '1', 'c', 'c', 'e', 'k', 'o', 'r', 's', 's']`
	
	`Original: CSE110 Rocks`

#4) Descending sort of a String with the optional reverse. For the string, sorting is done depending on the ASCII values.
 	
`unsorted_chars = "CSE110 Rocks"`
`sorted_chars = sorted(unsorted_chars.lower(), reverse = True)`
`print("Sorted:", sorted_chars)`
`print("Original:", unsorted_chars)`
	`[Sorted: ['s', 's', 'r', 'o', 'k', 'e', 'c', 'c', '1', '1', '0', ' ']`
	
	`Original: CSE110 Rocks`

#6) Descending sort of a Dictionary with the optional reverse. For the Dictionary, sorting is done depending on key values.
 	
`numbers = {1:"A", 100:"B", -5:"C", 44:"D", 2:"E"}`
`sorted_numbers = sorted(numbers, reverse = True)`
`print(sorted_numbers)`
 	
`for key in sorted_numbers:`
`  print(numbers[key], end = " ")`
	`[100, 44, 2, 1, -5]`
	
	`B D E A C`

#7) Ascending sort of a list without the optional reverse. Here, the key is the built-in len function. So the sorting is done depending on the length of the string.
 	
`words = ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
`sorted_words = sorted(words, key = len)`
`print("Sorted:", sorted_words)`
`print("Original:", words)`
	`Sorted: ['g', 'ef', 'pqr', 'abcd', 'hjklmno']`
	
	`Original: ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
	`s`

The sort() function

The sort() function modifies the original list. Unlike sorted(), it can only be used with lists.

Syntax of sort() Function
`List_name.sort(key,  reverse = False)`
Parameters
Key(optional): It is a function that works like a key and helps with custom sorting. The function named passed here can be built-in or custom-made. **Reverse(optional):** Its default value is False. For False, it sorts in the ascending order, and for True, it sorts in descending order. **Return Type:** None

Example:	**Output**

#1) Ascending sort of a list without the optional reverse
 	
`numbers = [10, 20, 70, 60, 40]`
`print("Before:", numbers)`
`numbers.sort()`
`print("After:", numbers)`
	`Before: [10, 20, 70, 60, 40]`
	
	`After: [10, 20, 40, 60, 70]`

#2) Descending sort of a list with the optional reverse.	`Before: [10, 20, 70, 60, 40]`

`numbers = [10, 20, 70, 60, 40]`
`print("Before:", numbers)`
`numbers.sort(reverse = True)`
`print("After:", numbers)`
	`After: [70, 60, 40, 20, 10]`

#3) Ascending sort of a list without the optional reverse. Here, the key is the built-in len function. So the sorting is done depending on the length of the string.
 	
`words = ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
`print("Before:", words)`
`words.sort(key = len)`
`print("After:", words)`
	`Before: ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
	
	`After: ['g', 'ef', 'pqr', 'abcd', 'hjklmno']`

#4) Descending sort of a list with the optional reverse. Here, the key is the built-in len function. So the sorting is done depending on the length of the string.
 	
`words = ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
`print("Before:", words)`
`words.sort(key = len, reverse = True)`
`print("After:", words)`
	`Before: ['abcd', 'ef', 'g', 'hjklmno', 'pqr']`
	
	`After: ['hjklmno', 'abcd', 'pqr', 'ef', 'g']`

Read the details about the built-in sorting from the provided link.
Link: https://docs.python.org/3/howto/sorting.html

Selection sort

For selection sort, in the ascending order, we need to find the minimum value of the given sequence and swap it with the first element of the sequence. Then, we need to find the second minimum from the rest of the data (starting from the 2nd index) and swap it with the 2nd element of the sequence. Then, we need to find the 3rd minimum from the rest of the data (starting from the 3rd index) and swap it with the 3rd element of the sequence. We need to continue this process until all the elements of the sequence have moved to their proper position.
At each iteration, the sequence is partitioned into two sections. The left section is sorted and the right section is being processed. Each iteration, we move the partition one step to the right, until the entire sequence has been processed.

The following shows the sequence of steps in sorting the sequence [17 3 9 21 2 7 5].
In the example below, the "|" symbol shows where the partition is at each step.

Input: 17 3 9 21 2 7 5

Step 1: | 17 3 9 21 2 7 5 << minimum is 2, exchange with 17
Step 2: 2 | 3 9 21 17 7 5 << minimum is 3, no exchange needed
Step 3: 2 3 | 9 21 17 7 5 << minimum is 5, exchange with 9
Step 4: 2 3 5 | 21 17 7 9 << minimum is 7, exchange with 21
Step 5: 2 3 5 7 | 17 21 9 << minimum is 9, exchange with 17
Step 6: 2 3 5 7 9 | 21 17 << minimum is 17, exchange with 21
Step 7 : 2 3 5 7 9 17 | 21 << STOP

For each iteration, we only consider the right section of the partition for finding the minimum value as the left side is already sorted. Lastly, in the last step, only one element remains which is the minimum. So sorting is unnecessary. So, for selection sort, size-1 times iteration is needed.

For descending order just find the maximum value instead of the minimum value.

Selection Sort

```
numbers = [17, 3, 9, 21, 2, 7, 5]
print("Before:", numbers)
#size/length is = 5
for index1 in range(0, len(numbers)-1): #0,1,2,3 (Iteration 4)
      min_val = numbers[index1]
      min_index = index1

      #finding the minimum value from partition to rest of data
      #partition/index is moving to right
      for index2 in range(index1+1, len(numbers)):
      if numbers[index2] < min_val:
            min_val = numbers[index2]
            min_index = index2
      #swapping partition's right value with the min value
      temp =  min_val
      numbers[min_index] =  numbers[index1]
      numbers[index1] = temp
```

```
print("After:", numbers)
```

For simulation(visualization) of the algorithms use this link below.
Link: https://visualgo.net/en/sorting

Bubble sort

For Bubble sort, in each iteration, adjacent elements are compared. If the adjacent elements are in the wrong order, then they are swapped. It has been named “Bubble sort” because of the way smaller or larger elements "bubble" to the top(Left side) of the list. As the number of iterations increase, the number of comparisons decreases.

The following shows the sequence of steps in sorting the sequence [17 3 9 21 2 7 5].

Input: 17 3 9 21 2 7 5

```
Before: [17, 3, 9, 21, 2]
Iteration: 0
Total compares: 4
[17, 3, 9, 21, 2] compare: 17 > 3 Swap places
[3, 17, 9, 21, 2] compare: 17 > 9 Swap places
[3, 9, 17, 21, 2] compare: 17 < 21 No swap
[3, 9, 17, 21, 2] compare: 21 > 2 Swap places
[3, 9, 17, 2, 21]
============================
Iteration: 1
Total compares: 3
[3, 9, 17, 2, 21] compare: 3 < 9 No swap
[3, 9, 17, 2, 21] compare: 9 < 17 No swap
[3, 9, 17, 2, 21] compare: 17 > 2 Swap places
[3, 9, 2, 17, 21]
============================
Iteration: 2
Total compares: 2
[3, 9, 2, 17, 21] compare: 3 < 9 No swap
[3, 9, 2, 17, 21] compare: 9 > 2 Swap places
[3, 2, 9, 17, 21]
============================
Iteration: 3
Total compares: 1
[3, 2, 9, 17, 21] compare: 3 > 2 Swap places
```

```
[2, 3, 9, 17, 21]
============================
After: [2, 3, 9, 17, 21]
```

Bubble Sort

```
numbers = [17, 3, 9, 21, 2]
print("Before:", numbers)

#size/length is = 5
for idx1 in range(0, len(numbers)-1): #0, 1, 2, 3 (Iteration 4)

     for idx2 in range(0, len(numbers)- idx1 - 1):
     #5-0-1=4 , 5-1-1=3 , 5-2-1=2, 5-3-1=1
          if (numbers[idx2] > numbers[idx2 + 1]):
          temp = numbers[idx2]
          numbers[idx2] = numbers[idx2 + 1]
          numbers[idx2 + 1] = temp
print("After:", numbers)
```

For simulation(visualization) of the algorithms use this link below.
Link: https://visualgo.net/en/sorting

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/