Data Structures & Algorithms Hyperdrive

Simplified data structures and algorithms for hyper-drivers

Sajidur Rahman

2021

Contents

Chapter 1

Data structures and algorithms

Examples are written in **Python**. **Go** examples will be added later. Main concepts are language independent.

1.1 Data structures

1.1.1 Linear data structures

1.1.1.1 Array

- ☒ Set
- ☐ Stack
- ☐ Queue
- ☒ Linked lists
- ☐ Hash table

1.1.2 Non-linear data structures

- ☐ Tree
- ☐ Graph

1.2 Algorithms

1.2.1 Searching

- ☒ Linear search
- ☒ Binary search

1.2.2 Sorting

- ☒ Bubble sort

- ☒ Selection sort
- ☒ Insertion sort
- ☐ Quick sort
- ☐ Merge sort

1.2.3 Performance

- ☒ Big O notation
- ☒ Time complexity
- ☐ Space complexity

1.3 Python installation

TODO: Installing and updating **pyenv** with automated script TODO: Installing latest python version TODO: Setting global python version

1.4 Setting up coding environment

TODO: Installing **vscode** (visual studio code) TODO: Installing python extensions

Chapter 2

Data structures

2.1 Data types

In the heart of computer there's only binary numbers, which is 1's and 0's. Solving real life problems in binary would have been extremely difficult if not impossible. To overcome this problem programming languages provides us data types. Integer, float, boolean, character etc are some of the most common data types.

2.1.1 Primitive data types

Data types that are defined by system are called primitive data types. Int, float, double, char, string, bool etc are the common primitive or system defined data types.

2.1.2 User-defined data types

For many situations only primitive data types are not enough. Most programming languages allow the users to define their own custom data types, which is called user-defined data types. Structures and classes are example of user-defined data types.

2.1.3 Abstract data types (ADT)

We combine the data structures with their operations to simplify the process of solving problem which is called Abstract data types (ADT). Two parts of ADT: 1. Declaration of data 2. Declaration of operations

2.2 Data structures

Data structure is a particular way of storing and organizing data, that can be used efficiently later on. Based on arrangement of the elements, data structures can be classified into two types:

2.2.1 Linear data structures

Elements are accessed in a sequential order but sequentially storing them is not essential. Example:

- Array, Set, Hash map, Linked list, Stack, Queue.

2.2.2 Non-linear data structures

Elements are stored and accessed in a non-linear order. Example:

- Tree, Graph

2.3 Array

The `array` is one of the most basic data structure. The `array` is simply a list of **data elements**. In **python** an array of animals would look like this:

```
animals = ["dog", "cat", "cow", "goat", "sheep"]
```

2.3.1 Size

The `size` of an array determined by how many items the array holds. In our case the length of **animals** array is **5**.

2.3.2 Index

The `index` of an *array item* is, it's position in the array expressed in number. Array index starts with the number `0`. So, the position of the first element/item of the array is `0`.

Last item index in an array is always (`n - 1`). Where the size of array is `n` *(n represents any integer)*. As array index starts with `0`.

2.3.3 Required steps for specific operation

2.3.3.1 Reading

Required step = 1. Item in any position

Reding from any position inside an array is 1 step.

2.3.3.2 Searching

Required steps best = 1, when it's the first item

Searching for a specific item in side an array is 1 step if the item is the first item of the array.

Required steps worst = n, when it's the last item

Searching for an item inside an array is n step in the array if the item is the last item of that array. Where n is the number of *total item count* inside that array.

2.3.3.3 Insertion

Required steps best = 1, when it's the last item
Required steps worst = n + 1, when it's the first item

2.3.3.4 Deletion

Required steps best = 1, when it's the last item
Required steps worst = n, when it's the first item

2.4 Ordered array

The `ordered array` is almost identical to the array. The only difference is ordered array has it's items sorted.

An `ordered array` might look like this:

```
numbers = [10, 20, 30, 40, 50]
```

2.4.1 Required steps for specific operation

2.4.2 Reading

Required step = 1. Item in any position

Reding from any position inside an array is 1 step.

2.4.3 Searching

Required steps best = 1, when it's the first item

Searching for a specific item in side an array is 1 step if the item is the firt item of the array.

Required steps worst = n, when it's the last item

Searching for an item inside an array is n step in the array if the item is the last item of that array. Where n is the number of *total item count* inside that array.

2.4.4 Insertion

Required steps best = 1, when it's the last item
Required steps worst = n + 1, when it's the first item

2.4.5 Deletetion

Required steps best = 1, when it's the last item
Required steps worst = n, when it's the first item

2.5 Set

A set is very similar to array, except set doesn't allow *duplicate values*. There are different types of sets, but we'll discuss about *array-based* set.

For this example we'll be using the same example from the array. This array is a set as all of it's items are unique.

```
animals = ["dog", "cat", "cow", "goat", "sheep"]
```

2.5.1 Required steps for specific operation on a set

2.5.2 Reading

Required step = 1. Item in any position

2.5.3 Searching

Required steps `best` = 1, when it's the first item
Required steps `worst` = n, when it's the last item

2.5.4 Insertion

Required steps `best` = n + 1, when it's the last item
Required steps `worst` = 2n + 1, when it's the first item

2.5.5 Deletion

Required steps `best` = 1, when it's the last item
Required steps `worst` = n, when it's the first item

2.6 Linked lists

Linked lists are collection of data like arrays. But they differ in some aspect like memory structure and operations. Find out more about them from the links below.

2.7 Singly linked list

Generally linked list refers to singly linked list. While arrays and linked list has many similarities, there are some big different under the hood. Items in a linked list is not contiguous block of memory like arrays. They can be scattered across different memory locations.

// TODO: add graphical explanation

Items of linked lists are called nodes. Each node consists of **value** and **link** to the next node. **Link** is the memory address of the next item. The last node has a **link** to *NULL* as it has no next element.

A linked list's first and last node can be also referred as head and tail respectively.

2.7.1 Creating linked list

```python
# creating linked list in python


class Node:
    """ Node class """

    def __init__(self, data=None):
        self.data = data
        self.next = None


class LinkedList:
    """ Linked list class"""

    def __init__(self):
        self.first_node = None


list = LinkedList()
list.first_node = Node("One")
node2 = Node("Two")
node3 = Node("Three")

list.first_node.next = node2
```

```
node2.next = node3

current_node = list.first_node
# print the node values
while current_node is not None:
    print(current_node.data)
    current_node = current_node.next
```

2.7.2 Results for code above:

```
One
Two
Three
```

2.7.3 Traversing linked list

```
# traversing linked list in python


class Node:
    """ Node class """

    def __init__(self, data=None):
        self.data = data
        self.next = None


class LinkedList:
    """ Linked list class"""

    def __init__(self):
        self.first_node = None

    def print(self):
        node = self.first_node

        while node is not None:
            print(node.data)
            node = node.next


list = LinkedList()
list.first_node = Node("One")
node2 = Node("Two")
node3 = Node("Three")
```

```
list.first_node.next = node2
node2.next = node3

list.print()
```

2.7.4 Results for code above:

```
One
Two
Three
```

2.7.5 Insertion at the beginning of a linked list

```
# insertion at the beginning in python


class Node:
    """ Node class """

    def __init__(self, data=None):
        self.data = data
        self.next = None


class LinkedList:
    """ Linked list class"""

    def __init__(self):
        self.first_node = None

    def set_first(self, data):
        node = Node(data)
        node.next = self.first_node
        self.first_node = node

    def print(self):
        node = self.first_node

        while node is not None:
            print(node.data)
            node = node.next


list = LinkedList()
list.first_node = Node("One")
node2 = Node("Two")
```

```
node3 = Node("Three")

list.first_node.next = node2
node2.next = node3

list.set_first("First")

list.print()
```

2.7.6 Results for code above:

```
First
One
Two
Three
```

2.7.7 Insertion at the end of a linked list

```
# insertion at the end in python


class Node:
    """ Node class """

    def __init__(self, data=None):
        self.data = data
        self.next = None


class LinkedList:
    """ Linked list class"""

    def __init__(self):
        self.first_node = None

    def set_first(self, data):
        node = Node(data)
        node.next = self.first_node
        self.first_node = node

    def set_last(self, data):
        node = Node(data)

        if self.first_node is None:
            self.first_node = node
            return
        last = self.first_node

        while last.next:
            last = last.next

        last.next = node

    def print(self):
        node = self.first_node

        while node is not None:
            print(node.data)
            node = node.next


list = LinkedList()
list.first_node = Node("One")
node2 = Node("Two")
node3 = Node("Three")

list.first_node.next = node2
node2.next = node3

list.set_first("First")
list.set_last("Last")

list.print()
```

2.7.8 Results for code above:

```
First
One
Two
Three
Last
```

2.7.9 Reading from a linked list

```
# reading from a linked list in python


class LinkedList:
    """ Linked list class"""

    def __init__(self):
        self.first_node = None

    def set_first(self, data):
        node = Node(data)
        node.next = self.first_node
        self.first_node = node

    def set_last(self, data):
        node = Node(data)

        if self.first_node is None:
            self.first_node = node
            return

        last = self.first_node

        while last.next:
            last = last.next

        last.next = node

    def read(self, index):
        current_node = self.first_node
        current_index = 0

        while current_index < index:
            current_node = current_node.next
            current_index += 1

            if not current_node:
                return

        return current_node.data

    def print(self):
        node = self.first_node

        while node is not None:
            print(node.data)
            node = node.next


list = LinkedList()
list.first_node = Node('One')
node2 = Node("Two")
node3 = Node("Three")
```

```
list.first_node.next = node2
node2.next = node3

list.set_first("First")
list.set_last("Last")

value = list.read(3)

print(value)
```

2.7.10 Results for code above:

```
Three
```

// TODO: add search
// TODO: add efficiency
// TODO: add performance table

2.8 Doubly linked list

Chapter 3

Algorithms

Algorithm is the step-by-step instructions to solve a specific problem.

3.1 Searching

3.1.1 Linear search

Linear search is the simplest way of searching for something inside a *collection*. It simply compares a value with every element inside a collection. It searches in a *linear direction*, until it finds the desired value. The searching gets terminated by either finding the value or reaching the end of the collection.

```python
# linear search example in python

animals = ["dog", "cat", "cow", "goat", "sheep"]


def linear_search(array, value):
    for item in array:
        if item == value:
            return array.index(item) # return the index of the item
    return False # return False if item not found


print(linear_search(animals, "cow"))
```

```
Output: 2
```

3.1.1.1 Required steps for specific operation

3.1.1.2 Best case

It takes just 1 step if the item is the first item of the collection.

3.1.1.3 Worst case

It takes n steps to find the item, if the item is the last element of that collection.

3.1.2 Binary search

Binary search is the search algorithm that eliminates half of possible items and works with the other half.

To perform binary search the array must be sorted.

```python
sorted_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


def binary_search(array, value):
    lower_bound = 0
    upper_bound = len(array) - 1

    while lower_bound <= upper_bound:
        mid_point = math.floor((upper_bound + lower_bound) / 2)
        mid_point_val = array[mid_point]

        if value == mid_point_val:
            return mid_point
        elif value < mid_point_val:
            upper_bound = mid_point_val - 1
        elif value > mid_point_val:
            lower_bound = mid_point + 1

    return False


binary_search(sorted_array, 4)
```

It took 3 steps to find value 4.
For a array / list with `100` ordered items it would take 7 steps maximum to find a value.

3.1.2.1 Required steps for specific operation

3.1.2.2 Best case

If the guess is just right, then only 1 step is needed. Which is rare scenario.

3.1.2.3 Worst case

3.2 Sorting

3.2.1 Bubble sort

// TODO: add intro

3.2.1.1 How it works

// TODO: add graphical explanation

3.2.1.2 Code example

```python
# bubble sort in python

def bubble_sort(array):
    # set upper bound
    upper_bound = len(array) - 1
    done = False

    while not done:
        done = True

        for i in range(upper_bound):
            # compare current index with its next index
            if array[i] > array[i+1]:
                # if current index is greater swap position
                array[i], array[i+1] = array[i+1], array[i]
                # we are not done yet
                done = False
        # after one iteration the largest item will go to the
        # last position and we'll exclude it every time
        upper_bound -= 1

    return array


print(bubble_sort([10, 30, 15, 11, 22]))
```

3.2.1.3 Time complexity of bubble sort

With a array of 5 elements we'd need a total of:

$4 + 3 + 2 + 1 = 10$ comparisons

For N number of elements that is:

$(n - 1) + (n - 2) + (n - 3)... + 1$ comparisons

As we got the comparisons count for the bubble sort. Let's look at the number of swaps we need. For the worst case scenario we'd need swap for every comparisons when the array component orders are exactly the opposite of a sorted array. In that case, swaps count would be 10 for an array with 5 elements. That is same as the swap count. So, the total steps for bubble sort would be:

10 comparisons $+10$ swaps $= 20$ steps for a 5 items array

3.2.1.4 Time complexity comparison table:

Data count n	Number of steps	n^2
5	20	25
10	90	100
20	380	400
30	870	900
40	1560	1600
50	2450	2500

For n amount of data step increment is approximately n^2. So, we can say the bubble sort algorithm has efficiency or time complexity of n^2.

3.2.2 Insertion sort

// TODO: add intro

3.2.2.1 How it works

// TODO: add graphical explanation

3.2.2.2 Code example

```python
# insertion sort in python

def insertion_sort(array):
    for index in range(1, len(array)):
        # put current element in temp variable
        temp = array[index]
        # set position to previous index
        position = index - 1

        # when previous element is bigger than current element
        while position >= 0 and array[position] > temp:
            # move previous element to current element position
            array[position + 1] = array[position]
            # move position backward
            position -= 1
            # move current element value to previous element position
            array[position + 1] = temp
    return array


print(insertion_sort([10, 30, 15, 11, 22]))
```

3.2.2.3 Efficiency of insertion sort

// TODO: add efficiency / time complexity

3.2.2.4 Efficiency comparison table:

n elements	Steps in bubble sort	Steps in selection sort	Steps in insertion sort
5	20	14	10
10	90	45	45
20	380	209	190
30	870	464	435
40	1560	819	780
50	2450	1274	1225

Insertion sort also has an efficiency of n^2.

3.2.3 Selection sort

// TODO: add intro

3.2.3.1 How it works

// TODO: add graphical explanation

3.2.3.2 Code example

```
# selection sort in python
```

```python
def selection_sort(array):

    # visit all array elements
    for i in range(len(array)):

        # find the minimum element
        min_idx = i
        for j in range(i+1, len(array)):
            if array[min_idx] > array[j]:
                min_idx = j

        # swap the found minimum element with the first element
        array[i], array[min_idx] = array[min_idx], array[i]
    return array


array = [5, 4, 3, 2, 1]
print(selection_sort(array))
```

3.2.3.3 Efficiency of selection sort

// TODO: add efficiency / time complexity

3.2.3.4 Efficiency comparison table:

N elements	Steps in bubble sort	Steps in selection sort
5	20	14
10	90	45
20	380	209
30	870	464
40	1560	819
50	2450	1274

For N items bubble sort takes almost N2 steps and for selection sort it’s N2/2.

Big O notation ignores constants.

That’s why efficiency of selection sort described as O(N2). It’s the same as bubble sort.

3.3 Big O notation

Big O notation is a mathematical family of notations to describe the efficiency of algorithms with the data growth. In programming it’s an asymptotic notation to represent the time or space complexity.

3.4 Time complexity

Big O notation simply describes not the number of steps an algorithm has, but how the steps increases as the data change (**growth rate**). If an algorithm always takes 2 steps, even if the data increases. The efficiency is O(1) not O(2). Because the steps are constant. After a certain amount data count O(1) algorithm is always efficient than O(N), if not fast initially.

For 100 data, if an O(1) algorithm takes 10 steps and O(N) takes 5 steps. For a 1000 data, O(1) will take less steps, as it's takes 10 steps every time for any amount of data. But for O(N), the steps will always increase with the increment of data count.

Big O notation generally refers to the worst case scenario, unless specified otherwise.

It describes exactly how inefficient an algorithm can get in a worst case scenario and helps to choose the better one.

3.5 Space complexity

Space complexity refers to determining how much memory an algorithm consumes during its runtime. It becomes a crucial factor where memory is limited. If an algorithm has an array of N items and that creates another array of the same size for its execution, the space complexity of that algorithm would be O(N), as the algorithm uses the same amount of additional memory as its input array.

3.6 Complexities table

Functions are arranged in ascending order according to their efficiency.

Position	Complexity	Name	Example
1	1	Constant	Adding and element to the front of a linked list
2	$logn$	Logarithmic	Finding and element in a sorted array
3	n	Linear	Finding an element in an unsorted array
4	$nlogn$	Linear Logarithmic	Sorting n items by divide-and-conquer Merge sort
5	n^2	Quadratic	Shortest path between two nodes in a graph
6	n^3	Cubic	Matrix multiplication
7	2^n	Exponential	The towers of hanoi problem

3.7 Logarithm

An algorithm like binary search can't be described as O(1) or O(N), as it takes much less steps than it's data count, and it's definitely not O(1). The worst case scenario for a binary search algorithm is 7 steps for 100 elements.

In Big O terms, this situation described as having time complexity of O(log N). Pronounce as "Oh of log N".

An algorithm that increases one step each time the data is doubled is O(log N) in term of Big O.

> Logarithm is the inverse function to exponentiation.

Here is an example:

If 2 has a exponent 3 the result is 8.

23 = 8

log is the opposite. If we have a value 8, what would be 2's exponent to get 8? The answer is 3.

log2 8 = 3

As we are working with binary logarithm, the base is 2. The above statement means.

If we have an array of 8 items, the worst case scenario to find an item will be 3 (2 x 2 x 2 = 8) steps, as we had to multiply 2 to itself 3 times.