|  |  |  |
| --- | --- | --- |
| **Module Title:** | Data Structures and Algorithms | |
| **Module Code:** | CMP4272 | |
| **Assessment Title:** | Technical Report | |
| **Assessment Type:** | CWRK | Weighting: **50%** |
| **College:** | College of Computing | |
| **Release date:** | 18/03/2024 | |
| **Submission date:** | 08/05/2024 | |

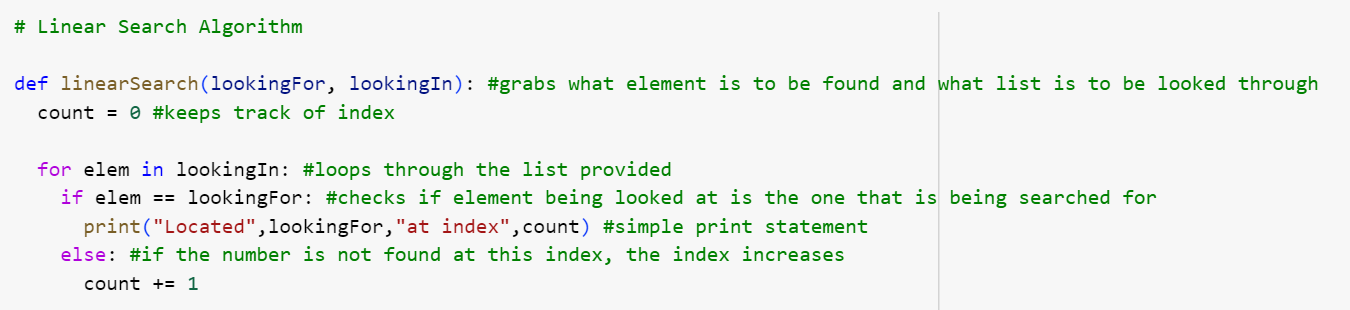
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number** | **Component** | **Description** | **Materials** | **Marks** |
| A1 | Data Structures and Algorithmic Solution | Effectve use of data structures which  suit the problem and appropriate.  Efficient algorithm implementation with consideration of algorithm ethics. | Refer to Lecture/Labs  Week-7, Week-8 and Week-9. | 20 |
| A2 | Algorithm Correctness | Test your implementation using sample input. | Refer to Lecture/Labs  Week-8, Week-9 and Week-12. | 10 |
| A3 | Algorithm Process, Plans and Strategy | Discuss about the algorithm process, strategy, plans with regards to each of the algorithms you have implemented. | Refer to Lecture/Labs  Week-7, Week-10 | 20 |
| B1 | Algorithm Efficiency- Running time computation using Dynamic Analysis | Computer running time for algorithms | Refer to Lecture/Labs  Week-9 and Week-11 | 20 |
| B2 | Algorithm Efficiency- Dynamic Analysis | Compare and discuss results of dynamic analysis. | Refer to Lecture/Labs  Week-9 and Week-11 | 10 |
| B3 | Algorithm Efficiency – Static Analysis | Use Big-Oh, Big-theta and Big-omega.  Sorting algorithms | Refer to Lecture/Labs  Week-9 and Week-11 | 10 |
| B4 | Algorithm Efficiency- Static Analysis | Big-Oh  Searching algorithms | Refer to Lecture/Labs  Week-8 and Week-11 | 10 |
|  |  | **TOTAL** |  | **100** |

# **SECTION A: ALGORITHM DESIGN, IMPLEMENTATION AND CORRECTNESS**

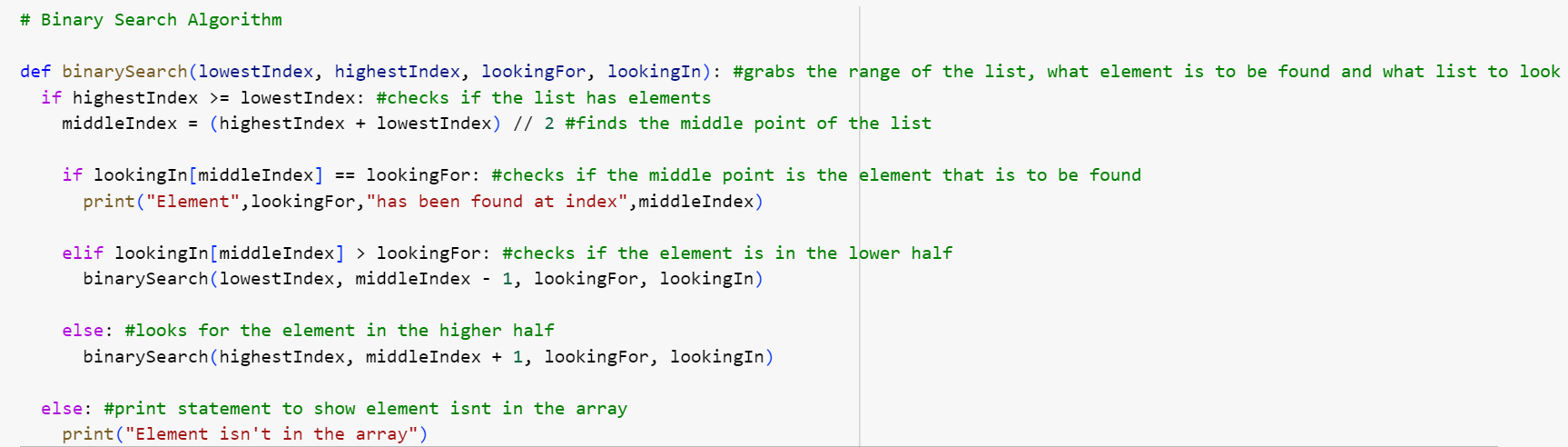
**A1.** Write python code to implement each of the following algorithms.

* 1. Linear Search
  2. Binary Search
  3. Selection Sort
  4. Quick Sort

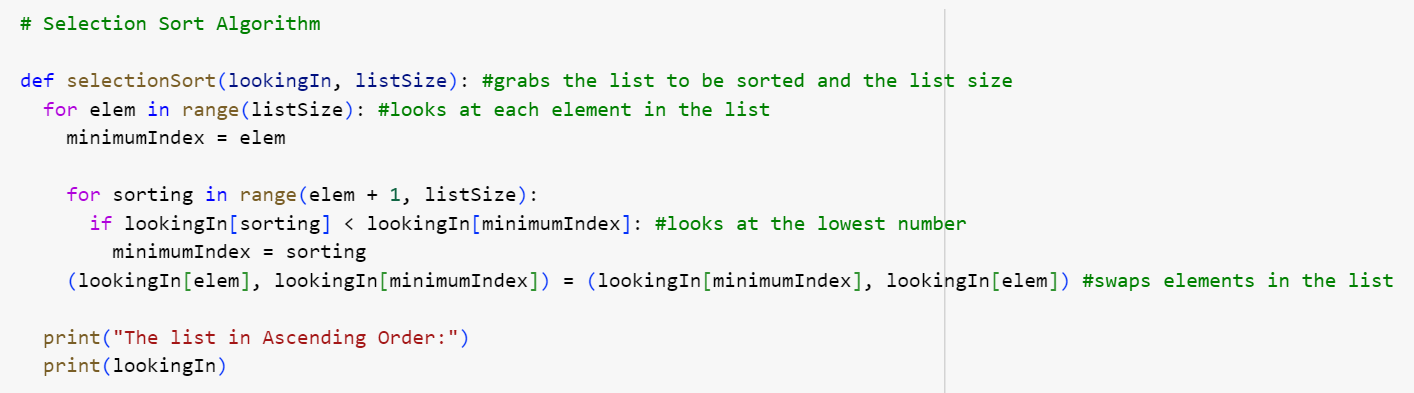
Linear Search

****

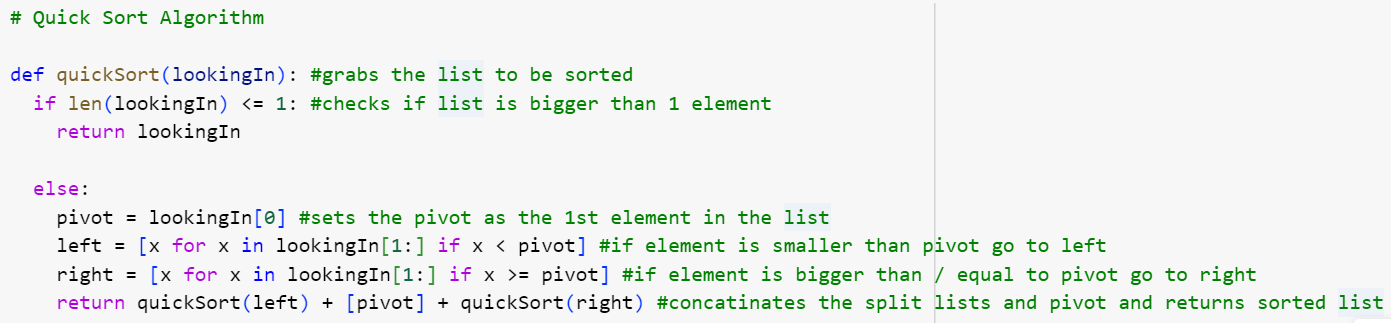
Binary Search



Selection Sort



Quick Sort



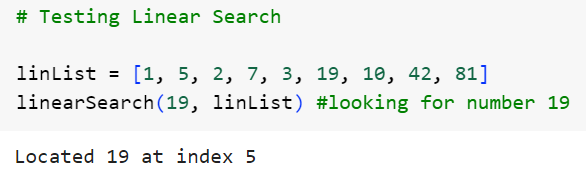
**[ 5 × 4 = 20 Marks]**

**[ 5 × 4 means 5 marks for each of the 4 algorithms]**

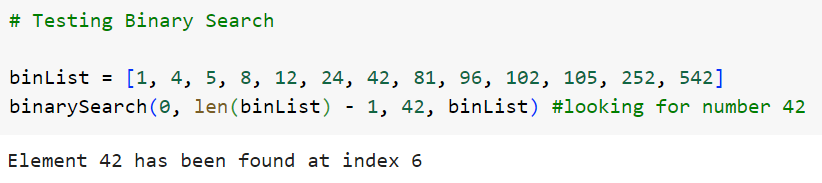
**A2.** You need to test the correctness of each of the algorithms implemented in **A1**. Testing is one of the steps to prove correctness of an algorithm.

Write Python code to test each of the algorithms implemented in **A1**.

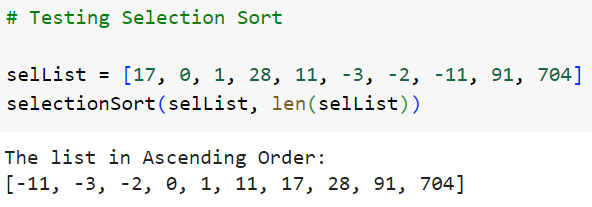
Linear Search



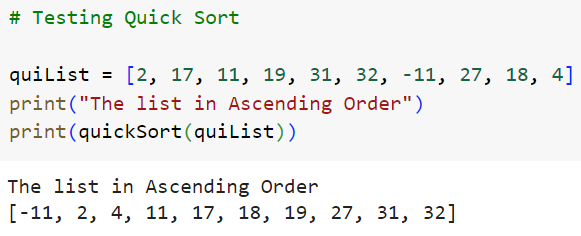
Binary Search



Selection Sort



Quick Sort



**[ 2.5 × 4 = 10 Marks]**

**A3.** Discuss about the algorithm process plans and strategy with regards to each of the algorithms you have implemented in **A1.**

Linear Search

The Problem – find a number in a list using linear search

The Input – a list and a number

The Output – the index of the number

Example – the index of the number 2 in the list [1,5,2,7,4] is 2

The main problem can be broken into 3 subproblems: looking at each value one by one, keeping a track of which element is being looked at and checking if the current value is the same as the one that is to be found.

The main plan used in this algorithm is find something with a linear search. However, during the 1st subproblem, the plan used is do something for each value, as it utilises a for loop. In the 2nd subproblem, the plan used is keeping a counter, in this instance it is a simple variable called count, which keeps track of which index is being look at in the list. The 3rd subproblem, the plan used is filter as the program has to compare each value in the list.

The approach used in this algorithm looks at each element one by one, hence it would be a brute force approach.­­

Binary Search

The Problem – find a number using binary search

The Input – a list, a number, lowest index, highest index

The Output – the index of the number

Example – the index of 7 in the list [1,2,4,5,7] is 4

The main problem can be broken into a couple subproblems; look at the middle element in the list, check if the value is the same as the number that is to be found, check if the number that is to be found is bigger/smaller than the middle number, create a new list based on the selected half of list, repeat until number is found

The plans that are used in the subproblems are: filter and collect results in a new list as the program will look at if the number that is being looked for is bigger or smaller than the middle point of the list. As such, the program also halves the original list and places it into a new list to then repeat the same process until the number has been found.

The approach used in this algorithm looks at the middle element and halves the list, hence it would be a divide and conquer approach.

Selection Sort

The Problem – sort a list in ascending order using selection sort

The Input – an unsorted list

The Output – a sorted list

Example – the list [2,1,6,4,7] sorted is [1,2,4,6,7]

The main problem can be broken into a couple subproblems; search through the list, find the lowest number, place the lowest number at the start of the list

The plan used in the 1st subproblem is do something for each value, in the 2nd subproblem the plan used is keep track of a best so far and in the 3rd subproblem the plan used collect results in a new list.

The approach used in this algorithm looks at all the elements of a list and compares each number to find the lowest once and places it at the start, hence it would be a brute force approach.

Quick Sort

The Problem – sort a list using quick sort

The Input – an unsorted list

The Output – a sorted list

Example – the list [7,1,8,2,9,4] sorted is [1,2,4,7,8,9]

The main problem can be broken into a couple subproblems; look at the middle element in the list, check if the value is the same as the number that is to be found, check if the number that is to be found is bigger/smaller than the middle number, create a new list based on the selected half of list, repeat until number is found

The plans that are used in the subproblems are: filter and collect results in a new list as the program will look at if the number that is being looked for is bigger or smaller than the middle point of the list. As such, the program also halves the original list and places it into a new list to then repeat the same process until the number has been found.

The approach used in this algorithm looks at the middle element and halves the list, hence it would be a divide and conquer approach.

**[ 5 × 4 = 20 Marks]**

# **SECTION B: ALGORITHM ANALYSIS**

**B1.** Compute the running time for the Selection sort and Quick sort algorithms and record it in the table given below. The input data need to be a list of randomly generated natural numbers.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Input Size** | | | | | | |
| **5** | **10** | **15** | **100** | **1000** | **10,000** | **Remarks** |
| **Selection Sort** | 0.001 sec | 0.001 sec | 0.002 sec | 0.002 sec | 0.133 sec | 13.904 sec |  |
| **Quick Sort (***First element as Pivot***)** | 0.001 sec | 0.001 sec | 0.001 sec | 0.001 sec | 0.006 sec | 0.039 sec |  |
| **Quick Sort (***Last element as Pivot***)** | 0.001 sec | 0.001 sec | 0.001 sec | 0.002 sec | 0.009 sec | 0.044 sec |  |
| **Quick Sort (***Randomly selected Pivot***)** | 0.0002 sec | 0.002 sec | 0.001 sec | 0.001 sec | 0.004 sec | 0.071 sec |  |

**[ 5 × 4 = 20 Marks]**

**B2.** Compare and discuss the results obtained in **B1**.

For selection sort, the processing goes up proportionately to the input size – the greater the input size the longer this method will take to fully sort a list. This method works well for small scale inputs as this algorithm uses a brute force approach and will look at each number one by one and place it in its corresponding place inside the sorted list.

Comparing the results of all methods for an input size of 10000, the fastest sorting algorithm would be quick sort with the 1st element as pivot as it took the least amount of time (0.039 seconds) on average, compared to other methods like selection sort which took around 13.904 seconds to fully sort the list.

Out of the 3 different quick sort algorithms, the fastest one across all input sizes would be quick sort with 1st element

**[ 10 Marks]**

**B3.** Discuss algorithm complexity of Selection Sort and Quick sort (with different choice of pivot) in terms of Big-O (O), Big-theta (Θ) and Big-omega (Ω).

Selection Sort

Big-O -

Big-theta -

Big-omega -

**[ 5 × 2 = 10 Marks]**

**B4.** Provide complete step-by-step workings to determine the worst-case complexity (Big-O) of Linear Search and the Binary search algorithms implemented in **A1.**

**[ 5 × 2 = 10 Marks]**

* Recorded time should be measured in second rounded to 5 decimal places.
* For each input size, the average time should be taken over 5 iterations.

For example, for input size 10; you need to measure time by taking average over 5 iterations.

* If any of the algorithms takes longer than 1 hour, you may stop it and record maximum running time (approximate) until you force it to stop. Add a relevant note in the remarks section accordingly.
* For QuickSort, you need to compute time for different selections of pivot.

**SAMPLE**

**Sum of a list of numbers**

# **SECTION A: ALGORITHM DESIGN, IMPLEMENTATION AND CORRECTNESS**

**A1. Implementation:**

Guidelines: Your source code should be well documented.

def list\_sum(numbers):

    """

    Function to find the sum of a list of numbers.

    Parameters:

    - numbers: List of numbers

    Returns:

    - sum: Sum of the numbers in the list

    """

    total\_sum = 0

    for num in numbers:

        total\_sum += num

    return total\_sum

**A2. Correctness:** Various functions created need to be tested. We can call the functions inside main (driver functions) and run our test cases to check for correctness for different scenarios.

# Test cases

if \_\_name\_\_ == "\_\_main\_\_":

    # Test case 1: List with positive integers

    numbers1 = [1, 2, 3, 4, 5]

    print("Sum of numbers1:", list\_sum(numbers1))  # Expected output: 15

    # Test case 2: List with negative integers

    numbers2 = [-1, -2, -3, -4, -5]

    print("Sum of numbers2:", list\_sum(numbers2))  # Expected output: -15

There is assert statement that is mainly used when running test cases. In your solutions, make use of assert and test your implementation using various test cases.

**A3. Algorithm Process, Plans and Strategy**

The problem is that of finding sum of numbers (integers) of a list.

Next, we identify input and output.

The input is a list and output is a number that is sum of all the values of list.

* Input: a list of numbers
* Output: the sum
* Example: the sum of [1,5,2,7,4] is 19

The given problem can be broken down into various smaller problems.

1. Look at each value in *lst* one by one

2. Add the value to total

For 1. Look at each value in *lst* one by one

for num in numbers:

The plan used is : Do something for each value.

Another plan this algorithm uses is Keep a running total Running which corresponds to code

total\_sum += num

The approach used in this algorithm adds up each element one by one, hence it would be a brute force approach.

# **SECTION B: ALGORITHM ANALYSIS**

**B1. Dynamic Analysis:**

For list size = 100

# Define the input size

input\_size = 100

# Create an empty list to store the random integers

input\_list = []

# Generate 100 random integers and append them to the list

for \_ in range(input\_size):

    random\_integer = random.randint(1, 100)

    input\_list.append(random\_integer)

# Measure the running time

start\_time = time.time()

result = list\_sum(input\_list)

end\_time = time.time()

elapsed\_time = end\_time - start\_time

# Output the running time

print(f"Input size: {input\_size}, Running time: {elapsed\_time:} seconds")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 100 | 1000 | 10,000 |  |
| *list\_sum* | 0.00011 |  |  |  |

Running time can be affected by various factors such as system load, hardware differences etc. Calculating the running time over many iterations and taking the average provides a more reliable and meaningful measurement of an algorithm's performance, enhancing its usefulness for analysis.

Recommended: Measure the running time over 5 or more iterations and record the average.

Likewise compute running time for different input size (given size) and record in the table.

**B2.** **Analysis, comparison, discussion of results**

Let us assume I am comparing list\_sum with another implementation (based on different approach).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 100 | 1000 | 10,000 | Remarks |
| *list\_su\_A* | 0.00011 | 0.00076 | 0.01239 |  |
| *list\_sum\_B* | 0.01451 | 0.00156 | 0.08156 |  |
| *list\_sum\_Slow* | 23 | 3800 | Not tested | For input size 1000, this algorithm takes around 1 hour to execute. Input size greater than 1000 were not tested. |

Based on the results, *list\_sum\_A* is faster than *list\_sum\_B* for all input ranges.

We need to report the values when providing the interpretation. Also broaden analysis check it from different angles like smaller range, larger range, different input size, variation, magnitude etc.

Report and refer to the values when providing the interpretation. Analysis needs to be thorough considering various factors for example different input size (small, large), order etc. For example, let us assume list\_sum function executes in .001 seconds for input size 5 whereas list\_sum\_B takes 0.0001 seconds, meaning list\_sum\_B is faster than list\_sum\_A. However, we cannot generalize this and conclude that it works the same for all input size.

**B3. Discussing complexity:**

Discuss the scenario for each algorithm. Include examples.

**Best Case**: It occurs when the list has only one element. In this case, the algorithm iterates through the list once and calculates the sum, which takes linear time. Therefore, the best-case time complexity is O(n), where n is the number of elements in the list.

Example code:

lst = []

**Average Case**: The average case time complexity remains O(n), as the algorithm always iterates through each element of the list once, regardless of the distribution of elements.

averge\_list = [3, -5, 2, 7, -1, 4, 0]

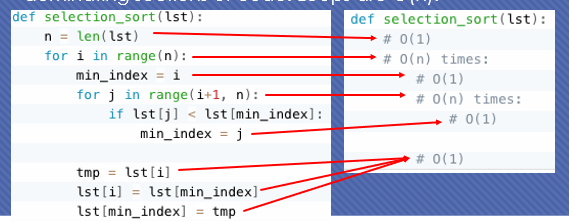
**Worst Case**: Similarly, the worst-case time complexity is O(n), which occurs when the list has a large number of elements. The algorithm still iterates through each element of the list once, leading to linear time complexity.

Example list would be a large list.

In case of list\_sum , the best case, average case and worst case follow the same time complexity which is O(n)

**B4. Static Analysis.**

Refer to Lecture/Labs related to Static Analysis [ Lecture -11] for detailed steps.



After working through each step, the complexity of this algorithm (selection sort) comes out to be O(n2)

Provide stepwise working for each algorithm.

**Submission Details:**

|  |
| --- |
| **What to Submit:**   1. *Technical Report:*  * You must submit electronic copies of your report to Moodle, in either Microsoft Word or PDF format. * The coding part (A1, A2 and B1) should be included as text, that needs to be well formatted and properly intended.  1. The coding part (A1, A2 and B1) should also be provided as .pynb file and submitted to the Moodle. |
| **Regulations**   * The minimum pass mark for a module is 40%. * Re-sit marks are capped at 40%.   **For detailed regulations, Late submission penalties etc., please refer to the assessment brief available on the Moodle.** |