Data Structures & Algorithms: Technical Report

Anisa Uddin 22128321

CMP4272

SECTION A: ALGORITHM DESIGN, IMPLEMENTATION AND CORRECTNESS

Algorithms enable a way to perform ways to organise substantial quantities of data in different ways. In this report, the algorithms explored are: linear search, binary search, selection sort and quick sort. Along with different forms of quick sort from different pivots (emphasised further on B1). Each algorithm have different complexities and subset of data that are performed and recorded the running times which is further explained on B1 and B2. The output information retrieved from the algorithms enables to provide understanding in research analysis, specifically static and dynamic showcasing it's prominence on the impact in real-time computing.

For the full access to the algorithms, visit the CMP4272 Technical Report.ipynb file that is submitted separately along with the code. Run the code blocks in order to maximise efficiency of the performance.

A1. Implementation of the following programmes

a. Linear Search

```
# linear search function

def linear_search(lst, target):

  for i, value in enumerate(lst):
    # for each value, it checks if value is equivalent to target
    if value == target:
      # returns the index of target if it is found, loop and function
ends
      return i
  # returns value (-1) if target is not found
  return -1

# test case below

list1 = [1, 2, 3, 4]
print(linear_search(list1, 2))
```

b. Binary Search

```
# binary search function

# begin with an interval which covers the whole array
def binary_search(lst, target):
  left = 0
  right = len(lst) - 1

# if value of the search key is less than the item in the middle of
interval, the integer division rounds down narrowing the interval to
the lower half
```

```
# if value is not less than the item in the middle of the interval,
narrow it to the upper half
  while left <= right:
    middle = (left + right)//2
    value = lst[middle]

    if value == target:
# return the index (target found)
      return middle
    elif value < target:
# narrows lower half further (to the left)
      left = middle + 1
    else:
# increases upper half (to the right)
      right = middle - 1

#returns value (-1) if x is not found
  return -1

# test case below

list2 = [20, 30, 2, 4, 87, 96, 43, 1]
print(binary_search(list2, 4))
```

c. Selection Sort

```
# selection sort algorithm

def selection_sort(lst):
# set min_index to location of i (0)
  n = len(lst)
  for i in range(n):
    min_index = i
# search for the minimum element in the list
    for j in range(i+1, n):
      if lst[j] < lst[min_index]:
        min_index = j

# swap with value at location min_index
    tmp = lst[i]
    lst[i] = lst[min_index]
# increment min_index to point to the next element
    lst[min_index] = tmp
  return lst

# test case below

list3 = [8907, 670, 1, 3, 37, 56, 87, 7, 928, 34, 13]
```

```
print(selection_sort(list3))
```

d. Quick Sort

```
# quick sort algorithm

def partition(lst,low,high):
# index of smaller pivot
  i = ( low-1 )
  pivot = lst[high]

  for j in range(low, high):
# if current element is smaller than pivot, then increment index of
smaller element
    if lst[j] < pivot:

      i = i+1
      lst[i],lst[j] = lst[j],lst[i]
  lst[i+1],lst[high] = lst[high],lst[i+1]
  return ( i+1 )

# quick sort function

def quick_sort(lst,low,high):
  if low < high:
    # pi is partitioning index, lst[p] is now at correct location
    pi = partition(lst,low,high)

    # sort the elemtns before partition and after partition
    quick_sort(lst, low, pi-1)
    quick_sort(lst, pi+1, high)
  return lst

# test case below

list4 = [675, 23, 6, 54, 33, 2, 11, 9, 2138, 98, 113]
size = len(list4)

print(quick_sort(list4, 0, size - 1 ))
```

A2. Algorithm correctness and testing

a. Linear Search test

Precondition:

- array must be compatible with the search i.e cannot be a string

Postcondition:

- if target is found in the array, return index of target
- if target is not found in the array, return -1
- array and target remains unchanged after the execution of linear search

```
# test linear search (precondition)

"""What must be true before executing linear search is the number to be
searched
   for is compatible with data in the array"""

# successful pass

def start_program(target: int):
    assert isinstance(target, int), 'Invalid data entry, target must be
an integer'
    assert int, 'No target found...'

    print('Entered sucessfully')

if __name__ == '__main__':
  test_list = [78, 32, 4, 2, 89, 0]
  num = test_list[0]
  start_program(target=num)

# unsuccessful pass
  test_list2 = [87, 'c', 'b', 'd', 6]

  try:
      num2 = test_list['c']
      start_program(target=num2)
  except TypeError:
    print('target Must be integer')
```

```
# test linear search (postcondition)

linear_list1 = [9, 4, 5, 32, 1, 3]
linear_list2 = [4, 2, 124, 43, 21]
linear_list3 = [123, 23, 1, 2, 3]

# succesful pass of linear_list1
```

```
# set the target to the index() function, which returns true
try:
  linear_search(linear_list1, 4) == linear_list1.index(4)
  #set the condition through assert and error message to debug
  assert (linear_search(linear_list1, 4) == linear_list1.index(4))
  #test
  print('The index of 4 is: ', linear_search(linear_list1, 4),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 4')

# successful pass of linear_list1

try:
  linear_search(linear_list1, 9) == linear_list1.index(9)
  assert (linear_search(linear_list1, 9) == linear_list1.index(9))
  print('The index of 9 is: ', linear_search(linear_list1, 9),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 9')

# unsuccesful pass of linear_list1

# condition is false and raises AssertionError

try:
  linear_search(linear_list1, 2) == linear_list1.index(5)
  assert (linear_search(linear_list1, 2) == linear_list1.index(5))
  print('The index of 2 is: ', linear_search(linear_list1, 2),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 5')
```

```
# successful pass of linear_list2
try:
  linear_search(linear_list2, 21) == linear_list2.index(21)
  assert (linear_search(linear_list2, 21) == linear_list2.index(21))
  print('The index of 21 is: ', linear_search(linear_list2, 21),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 21')

# successful pass of linear_list2

try:
  linear_search(linear_list2, 124) == linear_list2.index(124)
  assert (linear_search(linear_list2, 124) == linear_list2.index(124))
```

```
  print('The index of 124 is: ', linear_search(linear_list2, 124),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 124')

# unsuccessful pass of linear_list2

# raises AssertionError, prompts user to fix

try:
  linear_search(linear_list2, 3) == linear_list2.index(21)
  assert (linear_search(linear_list2, 3) == linear_list2.index(21))
  print('The index of 2 is: ', linear_search(linear_list2, 3),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 21')
```

```
# successful pass of linear_list3

try:
  linear_search(linear_list3, 123) == linear_list3.index(123)
  assert (linear_search(linear_list3, 123) == linear_list3.index(123))
  print('The index of 123 is: ', linear_search(linear_list3, 123),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 123')

# successful pass of linear_list3

try:
  linear_search(linear_list3, 2) == linear_list3.index(2)
  assert (linear_search(linear_list3, 2) == linear_list3.index(2))
  print('The index of 2 is: ', linear_search(linear_list3, 2),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 2')

# unsuccesful pass of linear_list3

try:
  linear_search(linear_list3, 23) == linear_list3.index(23)
  assert (linear_search(linear_list3, 2) == linear_list3.index(23))
  print('The index of 23 is: ', linear_search(linear_list3, 23),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 23')
```

b. Binary Search

Precondition:

- array must be already sorted

Postcondition:

- if target is found in the array, return index of target
- if target is not found in the array, return -1
- array and target remains unchanged after the execution of binary search

```
# test binary search (postcondition)

binary_search_list1 = [345, 76, 38, 1, 23, 543, 72]
binary_search_list2 = [64, 2432, 12, 8, 90, 43]
binary_search_list3 = [99, 321, 67, 83, 2, 5, 9, 17]

# successful pass of binary_search_list1

try:
  binary_search(binary_search_list1, 23) ==
binary_search_list1.index(23)
  assert (binary_search(binary_search_list1, 23)) ==
binary_search_list1.index(23)
  print('The index of 23 is: ', binary_search(binary_search_list1, 23),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 23')

# unsuccessful pass of binary_search_list1

try:
  binary_search(binary_search_list1, 345) ==
binary_search_list1.index(345)
  assert (binary_search(binary_search_list1, 345)) ==
binary_search_list1.index(345)
  print('The index of 345 is: ', binary_search(binary_search_list1,
345), '(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 345')

# unsuccessful pass of binary_search_list1

try:
  binary_search(binary_search_list1, 72) ==
binary_search_list1.index(72)
  assert (binary_search(binary_search_list1, 72)) ==
binary_search_list1.index(72)
```

```
  print('The index of 72 is: ', binary_search(binary_search_list1, 72),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 72')
```

```
# successful pass of binary_search_list2

try:
  binary_search(binary_search_list2, 90) ==
binary_search_list2.index(90)
  assert (binary_search(binary_search_list2, 90)) ==
binary_search_list2.index(90)
  print('The index of 90 is: ', binary_search(binary_search_list2, 90),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 90')

# unsuccessful pass of binary_search_list2

try:
  binary_search(binary_search_list2, 64) ==
binary_search_list2.index(64)
  assert (binary_search(binary_search_list2, 64)) ==
binary_search_list2.index(64)
  print('The index of 64 is: ', binary_search(binary_search_list2, 64),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 64')

# successful pass of binary_search_list2

try:
  binary_search(binary_search_list2, 12) ==
binary_search_list2.index(12)
  assert (binary_search(binary_search_list2, 12)) ==
binary_search_list2.index(12)
  print('The index of 12 is: ', binary_search(binary_search_list2, 12),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 12')
```

```
# unsuccessful pass of binary_search_list3

try:
  binary_search(binary_search_list3, 99) ==
binary_search_list3.index(99)
```

```
  assert binary_search(binary_search_list3, 99) ==
binary_search_list3.index(99)
  print('The index of 99 is: ', binary_search(binary_search_list3, 99),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 99')

# successful pass of binary_search_list3

try:
  binary_search(binary_search_list3, 83) ==
binary_search_list3.index(83)
  assert binary_search(binary_search_list3, 83) ==
binary_search_list3.index(83)
  print('The index of 83 is: ', binary_search(binary_search_list3, 83),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 83')

# unsuccessful pass of binary_search_list3

try:
  binary_search(binary_search_list3, 9) == binary_search_list3.index(9)
  assert binary_search(binary_search_list3, 9) ==
binary_search_list3.index(9)
  print('The index of 9 is: ', binary_search(binary_search_list3, 9),
'(Pass!)')
except AssertionError:
  print('Incorrect value/target for index, it should be 9')
```

c. Selection Sort

Precondition:

- Elements in array must compatible with each other
- Array cannot be empty

Postcondition:

- After execution, original elements unchanged
- Length of list remains unchanged

```
# testing selection sort

selection_list1 = [34, 54, 2, 34, 21, 435475]
selection_list2 = [4354, 32, 1, 3, 432, 76]
selection_list3 = [321431, 45, 43,2 , 14, 47, 83, 35, 8, 3]

#testing selection_list1
```

```
# successful lists

def test_selection_sort():
    # Test case 1
    selection_list1 = [34, 54, 2, 34, 21, 435475]
    selection_sort(selection_list1)
    assert selection_list1 == [2, 21, 34, 34, 54, 435475]

    # Test case 2
    selection_list2 = [4354, 32, 1, 3, 432, 76]
    selection_sort(selection_list2)
    assert selection_list2 == [1, 3, 32, 76, 432, 4354]

    # Test case 3
    selection_list3 = [321431, 45, 43,2 , 14, 47, 83, 35, 8, 3]
    selection_sort(selection_list3)
    assert selection_list3 == [2, 3, 8, 14, 35, 43, 45, 47, 83, 321431]

    print("Pass!")

# Run the test cases
test_selection_sort()
```

d. Testing Quick Sort

Precondition:

- array must be comparable and compatible elements

Postcondition:

- all elements to the left of pivot is less than or equal to pivot
- sublists formed by array are recursively sorted
- correctly placed in its final position after execution

```
# testing quick sort

# successful quick sort lists

def test_quick_sort():
    # Test case 1
    quick_sort1 = [3, 1, 24, 1, 324315, 9, 2, 6, 211235, 34, 5]
    size1 = len(quick_sort1)
    sorted_list1 = quick_sort(quick_sort1, 0, size1 - 1)
    assert sorted_list1 == [1, 1, 2, 3, 5, 6, 9, 24, 34, 211235,
324315]

    # Test case 2
    quick_sort2 = [5, 324, 3, 2423, 1]
```

```
    size2 = len(quick_sort2)
    sorted_list2 = quick_sort(quick_sort2, 0, size2 - 1)
    assert sorted_list2 == [1, 3, 5, 324, 2423]

    # Test case 3
    quick_sort3 = [3241, 2, 989084, 4214, 50]
    size3 = len(quick_sort3)
    sorted_list3 = quick_sort(quick_sort3, 0, size3 - 1)
    assert sorted_list3 == [2, 50, 3241, 4214, 989084]

    print("All tests pass.")

# Run the test cases
test_quick_sort()
```

A3. Discussion of algorithm process plans and strategies in regards to algorithms in A1

The algorithmic process linear search sequentially checks the list or array given and identifies the target. It checks through each element until it reaches at the end of the list and is most efficient with unsorted data or smaller lists. By searching through the array and returning the index, it provides simplicity.

Walkthrough of how linear search will work through an example scenario:

- Input: Unsorted list of numbers/ integers
- Output: The index of target value
- Example: list of numbers [9,38,4,643,1] and target = 38. It will output an index of 1.

To further explain:

1. Check each item in the lst separately and adjacently
2. If target equal to input value return the index (position of the value) as output
3. Not value will return -1

For 1. Checking each item:

```
for i, value in enumerate(lst):
```

For 2. Target equal to input value? Return index if true

```
if value == target:
return i
```

For 3. No target found, return -1

```
  return -1
```

Plan used is: Linear Search – finding specific value based on the input and return

Binary Search searches an item in a sorted list of items. By repeatedly dividing in half the portion of the list that may contain the item, until the list is narrowed down to one item. This is based on the concept of "divide and conquer", improving the search through recursively dividing.

Walkthrough of how binary search will work through an example scenario:

- Input: Sorted list of numbers
- Output: The index of target value
- Example: list of numbers [1,2,3,4,5] and target = 5 It will output an index of 4

To further explain:

1. Begin with sorted list
2. Calculate middle index and use as current interval
3. Compare and contrast middle element with target
4. If middle element equals target, return index
5. If target less than middle element, search the left of the list or right
6. Repeat recursively until target is found or list is empty

For 1. Beginning with input of sorted list

For 2. Calculation of the middle index when left is smaller than right and divide by 2

```
  while left <= right:
    middle = (left + right)//2
    value = lst[middle]
```

For 3. Comparing and contrast the target == value

```
    if value == target:
```

For 4. Return index if target == value

```
      return middle
```

For 5. Target is less than middle element, continue searching on left or right

```
    elif value < target:
# narrows lower half further (to the left)
      left = middle + 1
    else:
# increases upper half (to the right)
      right = middle - 1
```

For 6. Repetition of search until target is found or no more intervals, returns -1 if not found

```
  return -1
```

Plan for binary search: filtering that does not meet target value

Selection Sort is a sorting algorithm that organises unsorted lists or arrays by repeating the smallest element and swapping the first element.

Walkthrough of how Selection Sort will work through an example scenario:

- Input: Unsorted list of numbers
- Output: A sorted list of numbers
- Example: list of numbers [45,6,43,3,654, 87] and output [3,6,43,45,87,654]

To further explain:

1. Create an empty list called sorted list and unsorted list is input
2. Iterate across the unsorted list
3. Depict smallest element in unsorted list
4. Swap it with first element of unsorted list, increasing sorted list
5. Repeat process recursively until no more elements are found and list is sorted

For 1. Creation of empty list to implement sorted data and the input list is unsorted

```
    tmp = lst[i]
```

For 2. Iterate across the unsorted list

```
# swap with value at location min_index
    tmp = lst[i]
    lst[i] = lst[min_index]
```

For 3. Finding the smallest element in the unsorted list

```
    for j in range(i+1, n):
      if lst[j] < lst[min_index]:
        min_index = j
```

For 4. Swapping with first element of unsorted list, increasing sorted list

```
    lst[min_index] = tmp
```

For 5. Repeating the process

```
  return lst
```

Plan for Selection Sort: collecting a new ‘sorted’ list

Quick Sort algorithm is another sorting algorithm that also uses ‘divide-and-conquer’ sorting that repeatedly partitions the list in smaller lists based on the pivot element. The sublists are sorted singularly, separating each time.

Walkthrough of how Quick Sort will work through an example scenario:

- Input: Unsorted list of numbers
- Output: A sorted list of numbers
- Example: list of numbers [45,6,11,3,654, 8732] and output [3,6,11,45,654,8732,]

To further explain:

1. Choose pivot element from unsorted list
2. Partition list into two sublists. Elements greater than pivot and elements less than pivot

3. Repeatedly apply quick sort to the sublists
4. Split the sublists with a pivot in between

For 1. Choosing pivot from unsorted list

```
 pivot = lst[high]
```

For 2. Parting the list into two sublists. A list for less than pivot and a list for greater than pivot

```
    if lst[j] < pivot:

      i = i+1
      lst[i],lst[j] = lst[j],lst[i]
  lst[i+1],lst[high] = lst[high],lst[i+1]
```

For 3. Recursively applying quick sort to the sublists

```
    quick_sort(lst, low, pi-1)
    quick_sort(lst, pi+1, high)
```

For 4. Splitting the sublists with pivot in between

```
    pi = partition(lst,low,high)
```

Plan for Quick Sort: Collecting results in a new list and doing something for each value

SECTION B: ALGORITHM ANALYSIS

Part of programming is conducting thorough research and analysis to produce results that can optimise the efficiency of coding and overall performance. In algorithms analysis, it encompasses many constituents such as storage, time, and adequate resources for final execution. These factors are considered in functions that can explain time complexity or space complexity (discussed further in B3 and B4).

Programming algorithms from A1 and B1, a running time was produced and analysed. Results are explained further in B1 and B2. For full display of the algorithm, view the separate CMP4272 Technical Report.pynb file.

B1. **Algorithm Efficiency – Running time computation using Dynamic Analysis**

The algorithms: Selection Sort, Quick Sort (first element as pivot), Quick Sort (last element as pivot) and Quick Sort (randomly selected pivot) is performed on the data entries (5, 10, 15, 100, 1000, 10000) which the running time is recorded in seconds, rounded to 5 decimal places. The input data are randomly generated natural numbers.

Programmes extracted below from CMP4272 Technical Report.pynb file:

Selection Sort

```
# selection sort algorithm
```

```
def selection_sort(lst):
# set min_index to location of i (0)
  n = len(lst)
  for i in range(n):
    min_index = i
# search for the minimum element in the list
    for j in range(i+1, n):
      if lst[j] < lst[min_index]:
        min_index = j

# swap with value at location min_index
    tmp = lst[i]
    lst[i] = lst[min_index]
# increment min_index to point to the next element
    lst[min_index] = tmp
  return lst

# test case below

list3 = [8907, 670, 1, 3, 37, 56, 87, 7, 928, 34, 13]
print(selection_sort(list3))
```

Quick Sort (*Last element as Pivot)*

```
# quick sort algorithm

def partition(lst,low,high):
# index of smaller pivot
  i = ( low-1 )
  pivot = lst[high]

  for j in range(low, high):
# if current element is smaller than pivot, then increment index of
smaller element
    if lst[j] < pivot:

      i = i+1
      lst[i],lst[j] = lst[j],lst[i]
  lst[i+1],lst[high] = lst[high],lst[i+1]
  return ( i+1 )

# quick sort function
```

```
def quick_sort(lst,low,high):
  if low < high:
    # pi is partitioning index, lst[p] is now at correct location
    pi = partition(lst,low,high)

    # sort the elemtns before partition and after partition
    quick_sort(lst, low, pi-1)
    quick_sort(lst, pi+1, high)
  return lst

# test case below

list4 = [675, 23, 6, 54, 33, 2, 11, 9, 2138, 98, 113]
size = len(list4)

print(quick_sort(list4, 0, size - 1 ))
```

Quick Sort *(First element as Pivot)*

```
# quick sort (first element as pivot)

def partition(lst, low, high):

    # First Element as pivot
    pivot = lst[low]

    # The start points to the starting item of list
    start = low + 1

    # end points to the last item of the list
    end = high

    while True:
        # an indication all items/elements have been shifted to the
correct side of pivot
        while start <= end and lst[end] >= pivot:
            end = end - 1

        # opposing process
        while start <= end and lst[start] <= pivot:
            start = start + 1

        # Case in where loop is exits
        if start <= end:
            lst[start], lst[end] = lst[end], lst[start]
            # loop continues
```

```
        else:
            # exit out of loop
            break

    lst[low], lst[end] = lst[end], lst[low]
    # pivot element is retrieved, index is end
    # pivot element is at sorted position
    # return pivot element index (end)
    return end

# The main function that implements QuickSort
# lst[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index
def quick_sort_first(lst, start, end):

    # If low is lesser than high
    if start < end:

        # idx is index of pivot element which is at its
```

Continuation of Quick Sort (first element as pivot)

```
        # sorted position
        idx = partition(lst, start, end)

        # Separately sort elements before
        # partition and after partition
        quick_sort_first(lst, start, idx-1)
        quick_sort_first(lst, idx+1, end)

# Function to print the list
def print_lst(lst, n):
    for i in range(n):
        print(lst[i], end=" ")
    print()

#
list12 = [213, 32, 12132, 76, 0, 2, 213, 21, 3, 9]
quick_sort_first(list12, 0, len(list12)-1)
print_lst(list12, len(list12))
```

Quick Sort (*Randomly selected Pivot*)

```
# quick sort (random pivot)

import random

'''
The function which implements QuickSort.
lst :- list required to be sorted
start :- starting index of the list
stop :- ending index of the list
'''
def quicksort_random(lst, start , stop):
    if(start < stop):

        # pivotindex is the index where
        # the pivot lies in the list
        pivotindex = partitionrand(lst,\
                              start, stop)

        # the list is partially sorted around pivot
        # sort the left and right of the list seperately
        quicksort_random(lst , start , pivotindex-1)
        quicksort_random(lst, pivotindex + 1, stop)

# function to generate random pivot

# swaps first element of the pivot and calls partition function
def partitionrand(lst , start, stop):

    # Generate a random number in the list between starting and ending
index
    #  of list
    randpivot = random.randrange(start, stop)

    # Swapping the starting element of
    # the array and the pivot
    lst[start], lst[randpivot] = \
        lst[randpivot], lst[start]
    return partition(lst, start, stop)

# test case
if __name__ == "__main__":
    quick_random = [90, 38, 7585, 3, 432, 4, 6, 473, 86, 9, 1038]
    quicksort_random(quick_random, 0, len(quick_random) - 1)
    print(quick_random)
```

Programme used to record running times of data entries

```
# Generation of natural numbers

import time
import random

# calculate sum of list of numbers
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 sublist in numbers:
        for num in sublist:
            total_sum += num
    return total_sum

"""Below are randomly generated numbers covering numbers from the
table.

Input size adn lsit are made for each data entry. This is
sorted through selection sort and quick sort functions.

For each list, time is recorded and output. This is recorded in the
table
of the report."""

# Define the input size

input_sizes = [5, 10, 15, 100, 1000, 10000]

# Measure running time for Selection Sort
print("Selection Sort:")
print("Input Size\tRunning Time (seconds)")
for size in input_sizes:
    input_list = random.sample(range(size * 10), size)  # Generate
random input list
    start_time = time.time()
    selection_sort(input_list)
    end_time = time.time()
```

Continuation of programme used for production of running time

```
    elapsed_time = end_time - start_time
    print(f"{size}\t\t{elapsed_time:.6f}")

# Measure running time for Quick Sort (First element as pivot)
print("\nQuick Sort (First element as Pivot):")
print("Input Size\tRunning Time (seconds)")
for size in input_sizes:
    input_list = random.sample(range(size * 10), size)  # Generate
random input list
    start_time = time.time()
    quick_sort_first(input_list, 0, len(input_list)-1)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"{size}\t\t{elapsed_time:.6f}")

# Measure running time for Quick Sort (Last element as pivot)
print("\nQuick Sort (Last element as Pivot):")
print("Input Size\tRunning Time (seconds)")
for size in input_sizes:
    input_list = random.sample(range(size * 10), size)  # Generate
random input list
    start_time = time.time()
    quick_sort(input_list, 0, len(input_list)-1)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"{size}\t\t{elapsed_time:.6f}")

# Measure running time for Quick Sort random
print("\nQuick Sort (Randomly selected Pivot):")
print("Input Size\tRunning Time (seconds)")
for size in input_sizes:
    input_list = random.sample(range(size * 10), size)  # Generate
random input list
    start_time = time.time()
    quicksort_random(input_list, 0, len(input_list)-1)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"{size}\t\t{elapsed_time:.6f}")
```

Running time table

	Input Size						
	5	**10**	**15**	**100**	**1000**	**10,000**	**Remarks**
Selection Sort	0.00001	0.00001	0.00002	0.00057	0.05263	5.27577	Least efficient for large input data
Quick Sort (*First element as Pivot***)**	0.00001	0.00002	0.00002	0.00014	0.00201	0.02982	Least efficient out of the quick sorts, but better than Selection Sort
Quick Sort (*Last element as Pivot***)**	0.00001	0.00002	0.00003	0.00014	0.00202	0.02682	Performs the best and handles large data efficiently
Quick Sort (*Randomly selected Pivot***)**	0.00002	0.00003	0.00003	0.00022	0.00267	0.03555	More efficient at handling large data than first element at pivot

B2. Algorithm Efficiency Discussion – Dynamic Analysis

Based on the performances of the algorithms, it is clear that Selection Sort is the least efficient amongst the algorithms. It takes 5.27577 seconds to process 10,000 suggesting that it cannot handle large mass of data. In contrast, to the most efficient at handling large data Quick Sort (last element as pivot), processing at 0.2682 seconds. The least amount of time taken compared to the other algorithms. Selection Sort is of around 5x more than the Quick Sort (last element as pivot) when processing 10,000 in which if doubled the data to 20,000, it may reach to around 10x the time taken to sort the data (~10 – 12 seconds). The data provides further evidence that Selection Sort is unable to process large inputs when the time taken to process an input of 100 is significantly increased to 0.00057 from 0.00002 (sorting a list of 15 entries).

Quick Sort, selected from last pivot, is able to process at faster rate in comparison to Selection Sort. When the input is increased from 15 to 100, of about 6.67 times bigger, there is a small increase in running time of 0.00011 seconds. Across all data entries, the algorithm takes the least amount of time with larger entries such as 100, 1000 and 10000 but is in tie with other algorithms with lower inputs. Conversely, Quick Sort (pivot selected from the first element) is also is able to grasp large input methodically as it only has a difference of 0.03 seconds processing list of 10,000. Based on the running time, Quick Sort (selected from first element) and Quick Sort (selected from final element) behave similarly. It's mannerisms and procession has similar running time against Quick Sort (selected from last pivot) on all other entries, with even being able to operate 15 and 1000 rapidly with only a difference of 0.00001 second. Quick Sort, when selected from the first element as pivot, outperforms Selection Sort and Quick Sort (randomly selected pivot).

Quick Sort (pivot randomly selected) takes the least amount of time in comparison to Selection Sort when handling larger quantities by 4.92027 seconds which is significantly much larger. For other data

entries, such as 100 and 1000, it falls behind other Quick Sort algorithms and does not behave similarly. However, the it follows almost as the same running time with the other algorithms when operating smaller data, with only a difference of 0.00001 seconds. Although, this algorithm is requires longer time to process the smaller data amongst other algorithms.

Across all algorithms, the consistency of running times are adjacent to each other with the Quick Sort algorithms however, Selection Sort fails to behave similarly and the procession is dramatically increased when addressing substantial input.

B3. Algorithm Efficiency – the discussion of algorithm complexity of Selection Sort and Quick Sort (random pivot selection) in terms of Big-O(O), Big-theta(O) and Big-omega(Ω).

Algorithm complexities require notations to express an algorithm's time and space complexity. Regulating the speed of performance of algorithms is imperative for time-sensitive applications which time complexity expresses in contrast to space complexity affecting the quantity of memory usage, the volume of 'space' taken to run the algorithm. It is crucial for memory-constrained devices. Arguably, space complexity is not emphasised as much as time complexity any longer as most machinery today have an appreciable memory storage. Nevertheless, both are take into consideration when testing performances.

The notations:

- Big-O(O)/ $O(n^2)$ represents the worst-case performance. Sets an upper bound on the code.
- Big-theta(Θ)/ $\Theta(n \times p)$ represents the average case performance
- Big-omega(Ω)/ $\Omega(n)$ represents the best case performance, setting a lower bound

Selection Sort sorts through an array in ascending or descending order through processing through the unsorted array, extracting the smallest element (initialisation stage) and swapping them, iterating through the process. It used nested loops; outer loops define the boundary of the unsorted array and inner loop identifies the smallest element to be swapped with first unsorted item in the array. It can be optimised in situations where costs of swapping items honourably surpasses the cost of comparing the items or in memory-constraint due to Selection Sort listed as in-place sorting algorithm (does not require additional spacing beyond what it is given). It is also preferred in situations where algorithms require a predicted time of task, which Selection Sort simplicity is employed and is predictable – an increase in certainty in real-time processing systems.

The simplicity of Selection Sort is stressed when operating with smaller datasets as it spends less time however, struggles to manage larger datasets which increases time taken. The limitations of Selection Sort is what it's strengths are, that it can only handle less memory input efficiently.

In terms of notations, Selection Sort has a time complexity of Big-O $[O(n^2)]$ in all cases. This is due to the outer loop altering its scale with each addition to the size of dataset, with second loop comparison with elements to identify the smallest and consider a swap. It always performs $O(n^2)$ comparisons and $O(n)$ swaps.

Best case of Selection Sort: A sorted array or list is the most effective for best performance of Selection Sort.

An example list below:

```
arr = [1, 2, 3, 4, 5]  # Sorted array
```

Average case of Selection Sort: Generation of random list

An example list below:

```
def generate_random_list(size):
    return [random.randint(1, 1000) for _ in range(size)]

# Test the average case scenario
size = 10  # Adjust the size of the array as needed
arr = generate_random_list(size)
```

Worst case of Selection Sort: generation of the list in descending order, forcing the Selection Sort to perform the maximum number of swaps and comparisons

An example list below:

```
def generate_descending_list(size):
    return list(range(size, 0, -1))

# Test the worst-case scenario
size = 10  # Adjust the size of the array as needed
arr = generate_descending_list(size)
```

On the other hand, Quick Sort with a randomly selected pivot performs better than Selection Sort. A random pivot is selected in the partition function. Initialising the first part of partitioning in such that all elements located to the left of pivot is smaller in contrast to elements located at the right of pivot are greater than pivot. Calling and iterating the same procedure for left and right subarrays.

Factors to include when constructing the best-case time complexity are:

- Position of array
- Size of array
- State of array (sorted, reverse or randomly organised)

It can be used in real-time computing in scientific computing, search engines and databases. It's speed on average is rapid in comparison to Selection Sort. Additionally, it has great cache performance due to its in-place sorting algorithm. It is also universally applicable on both arrays and linked lists, operating with any type of data. The scalability of Quick Sort is also effective as it can grasp increasing lists or datasets, significantly benefiting on large-scale data.

Due to a random pivot being selected during the partition phase, it will have the best chances of performing efficiently if the pivot randomly chooses an array that does not fall to smallest or largest pivot. An imbalance is caused by selection of largest or smallest, and produces the worst-case time complexity when this occurs. A randomised partition is selected as it is more likely to choose a balanced partition in contrast to using first or final elements in an array. It has the best and average case time complexity when randomised with notations of: $O(n \log n)$, $\Theta(n \log n)$, $\Omega(n \log n)$.

Best case for Quick Sort (random pivot): Division of array is consistently balances partitions

An example below:

```
def quick_sort(arr):
```

```
    if len(arr) <= 1:
        return arr
    else:
        pivot = random.choice(arr)  # Choose a random pivot
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quick_sort(left) + middle + quick_sort(right)

# Test the best-case scenario
arr = [random.randint(1, 1000) for _ in range(10)]  # Generate a random
array
```

Emphasis on:

```
pivot = random.choice(arr)  # Choose a random pivot
```

Average case scenario for Quick Sort (random pivot): Generation of random lists of numbers

```
# Test the average case scenario
size = 10  # Adjust the size of the array as needed
arr = generate_random_list(size)
```

Worst case scenario for Quick Sort (random pivot): A list in sorted order causing imbalance in recursive calls

```
def generate_sorted_list(size):
    return list(range(1, size + 1))

# Test the worst-case scenario
size = 10  # Adjust the size of the array as needed
arr = generate_sorted_list(size)
```

B4. Static analysis of worst-case complexity (Big-O) of Linear Search and the Binary search algorithms implemented in A1

Calculation of the notations in worst-case:

Linear Search: Element not present

```
def linear_search(lst, target):
    for i in range(len(lst)):  ------------->#O(n)
        if lst[i] == target:   ------------># O(n)
            return i
    return -1  # Element not found

# Create an array with elements 1 to 75
lst = list(range(1, 76))       -------------> #O(n)
print("Original array:", lst)
```

```
# Perform linear search for an element not present in the list
target = 76                    -----------→ # O(1)
index = linear_search(lst, target)   ---------→ O(n)
if index != -1
    print(f"Element {target} found at index {index}.")
else:
    print(f"Element {target} not found in the list.")
```

```
def linear_search(lst, target):
  # O(n) times:
  # O(1)

# Create an list with elements 1 to 10
# O(1)
# O(1)
# O(n)
```

Overall: O(n)

Binary Search: Element not present

```
def binary_search(lst, target):
    low = 0 ---------------------→ # O(1)
    high = len(lst) - 1 -----------→ # O(n)

    while low <= high: --------------→ # O(n)
        mid = (low + high) // 2 ----------→ #O(log on)
        if lst[mid] == target: ---------→ #O (n)
            return mid
        elif lst[mid] < target:
            low = mid + 1   -------------→ #O(n)
        else:
            high = mid - 1 -------------→ # O(1)

    return -1  # Element not found

# Test the worst-case scenario
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  # Sorted list  ----→ # O(1)
target = 23  # Element not present in the list   --------→ # O(1)
index = binary_search(lst, target)
if index != -1:
    print(f"Element {target} found at index {index}.")
else:
    print(f"Element {target} not found in the list.")
```

```
def binary_search(lst, target):
  # O(1)
```

```
# O(n)
# O(n)
# O(log on)
# O(1)
# O(n)
# O(1)

#Test worst-case scenario
# O(1)
# O(1)
```

Overall answer: O(log on)

Conclusion

Analysis of algorithms is prominent to understanding the performances processes and how each algorithm is best suited to different situations depending on its strength and weaknesses. By providing raw data and implementing in a running table, this report intends to introduce data to evident the findings of the time complexities and support our understanding of algorithms in real-time.

Attributions:

These are not directly sourced from the websites however, intended used for modifying the algorithms and sourcing information.

For A1, the algorithms were modified from different sources: The CMP4272 module presentation, (Lecture 8-slide 5, Lecture 8-slide 9, Lecture 9-slide 18, lecture 9, slide 36). Testing conditions were sources from YouTube channel Real Python and Idently, in combination with website GeeksforGeeks. Modification of the technical report examples were used specifically for B1 (sum_list). Selection Sort algorithm was modified from lecture 9, slide 18. Quick Sort (last element as pivot) was modified from lecture 9, slide 36. Quick Sort (first element as pivot) & Quick Sort (Randomly generated as pivot) are modified from GeeksforGeeks. General information of selection sort and quick sourced from websites used in B3:

https://www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/selection-sort/#:~:text=Selection%20Sort%20has%20a%20time,the%20size%20of%20the%20input

and

https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

Most of the code blocks were modified from GeeksforGeeks or OverStack Flow