University of Hertfordshire UH

Data Structures and Algorithms 1

FINAL SUBMISSION - ASSIGNMENT

Trevor Gankarch |MSc in Computer Science with Artificial Intelligence| 10th March 2025

QUESTION 1:

Quick sort is a divide and conquer technique of sorting, that is widely used due to its efficiency, especially for large data sets. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, elements less than the pivot and elements greater than the pivot. The sub-arrays are then sorted recursively.

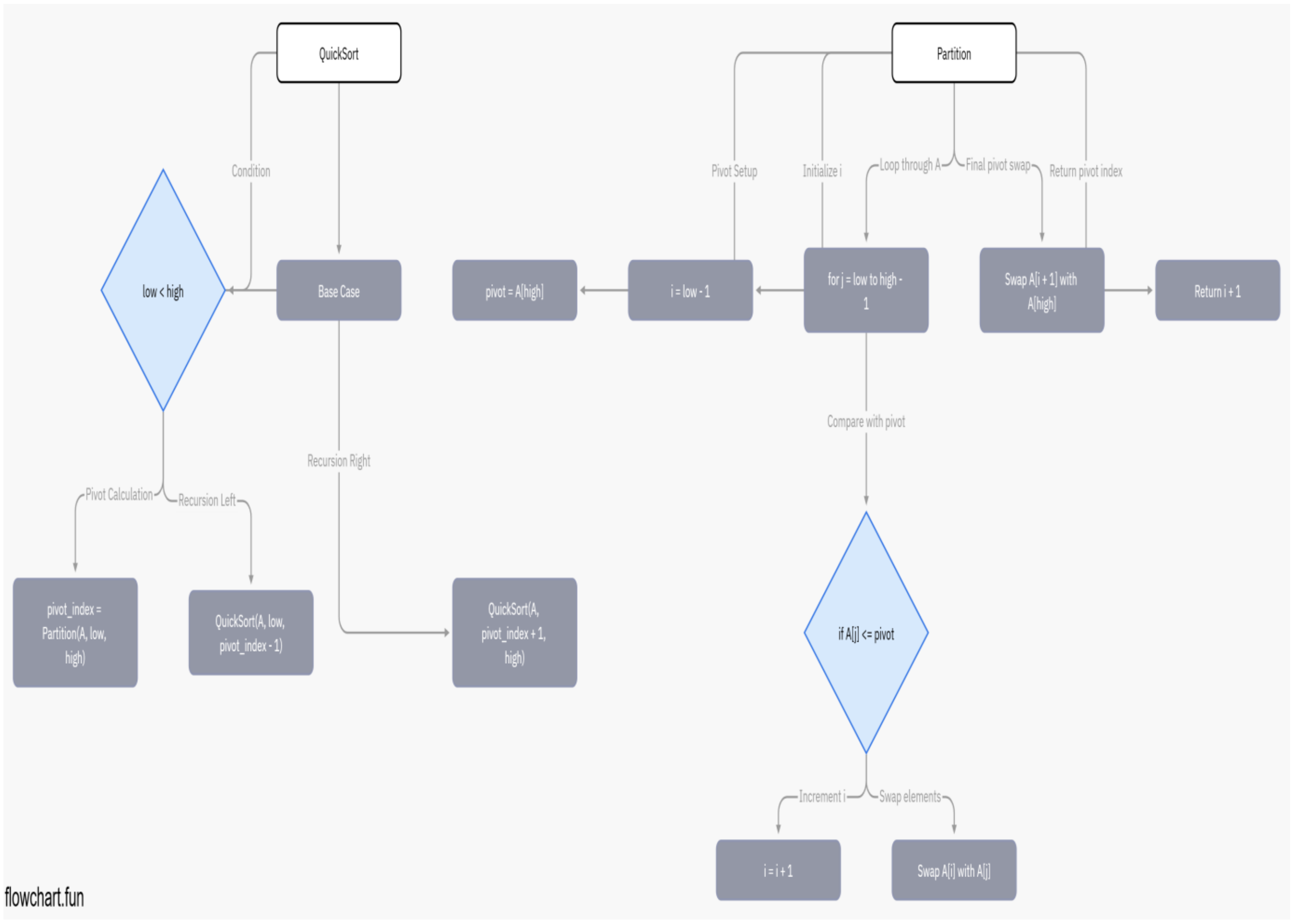
Here is the Pseudocode for Quick Sort;

```
QuickSort (A, low, high)
    if low < high then
    pivot_index = Partition (A, low, high)
    Quicksort( A, low, pivot_index – 1)      // Recursively sort the left sub-array'
    Quicksort (A, pivot_index + 1, high)  // Recursively sort the right sub-array

 Partition (A, low, high)
    pivot = A[high] // Choose the last element as the pivot
   i = low – 1          // Initialize i to be one less than the low index
   for j = low to high – 1 do
      if A[J] <= pivot then

      i = i + 1
      swap A[i] with A[j]
swap A [ i + 1] with A [ high]        // Move pivot to the correct position
return i + 1                          // Return the index of the pivot
```

Here is the flow chart;



When applying Quick Sort to the array of [9,2,4,11,2,5,9,7,8] which is the initial array

Step by step execution

1. First iteration – 8 selected as pivot, elements less than 8 will be [2,4,2,5,7], the pivot is placed where 5 was, the array after the first partition is [2,4,2,5,7,8,9,11,9]
 Recursively sort the left and right sub-arrays, we will get left sub-array [2,4,2,5,7] and right sub-array: [9,11,9]
2. Second Iteration (Left Sub- array) – Pivot is 7, elements less than 7 are [2,4,2,5], pivot 7 is placed at index 4. Array after partition is [2,4,2,5,7,8,9,11,9], recursively sort the left sub-array [2,4,2,5]
3. Third Iteration (Left Sub-array of the Left Sub-array), pivot is 5, portioning the elements less than 5 is [2,4,2] and pivot 5 is placed at index 3. Array after the partition [2,4,2,5,7,8,9,11,9], recursively sort the left sub-array [2,4,2].
4. Fourth Iteration (Left Sub-array of Left Sub-array of Left Sub-array), pivot is now 2, partitioning elements less than 2 is none, pivot now becomes 2 which is placed at index 1, array after partition now is [2,2,4,5,7,8,9,11,9]

5. Continuing to recursively sorting, the right sub-array [4] is already sorted, the right sub-array [7] is already sorted, and recursively apply sorting to the right sub-arrays [9,11,9].

   Therefore, the Final Sorted Array is [2.2.4,5,7,8,9,9,11]

TIME COMPLEXITY ANALYSIS

The time complexity analysis done on the array and the pivot selection resulted in reasonably balanced partitions so this has achieved O (n log n)

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.).
2. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
3. Geeksforgeeks. (2025). Quick Sort Algorithm. Retrieved from https://www.geeksforgeeks.org/quick-sort/

2. Provide the pseudocode and flow chart for the operation “**adding matrices**”. You must also include in it the appropriate checks.

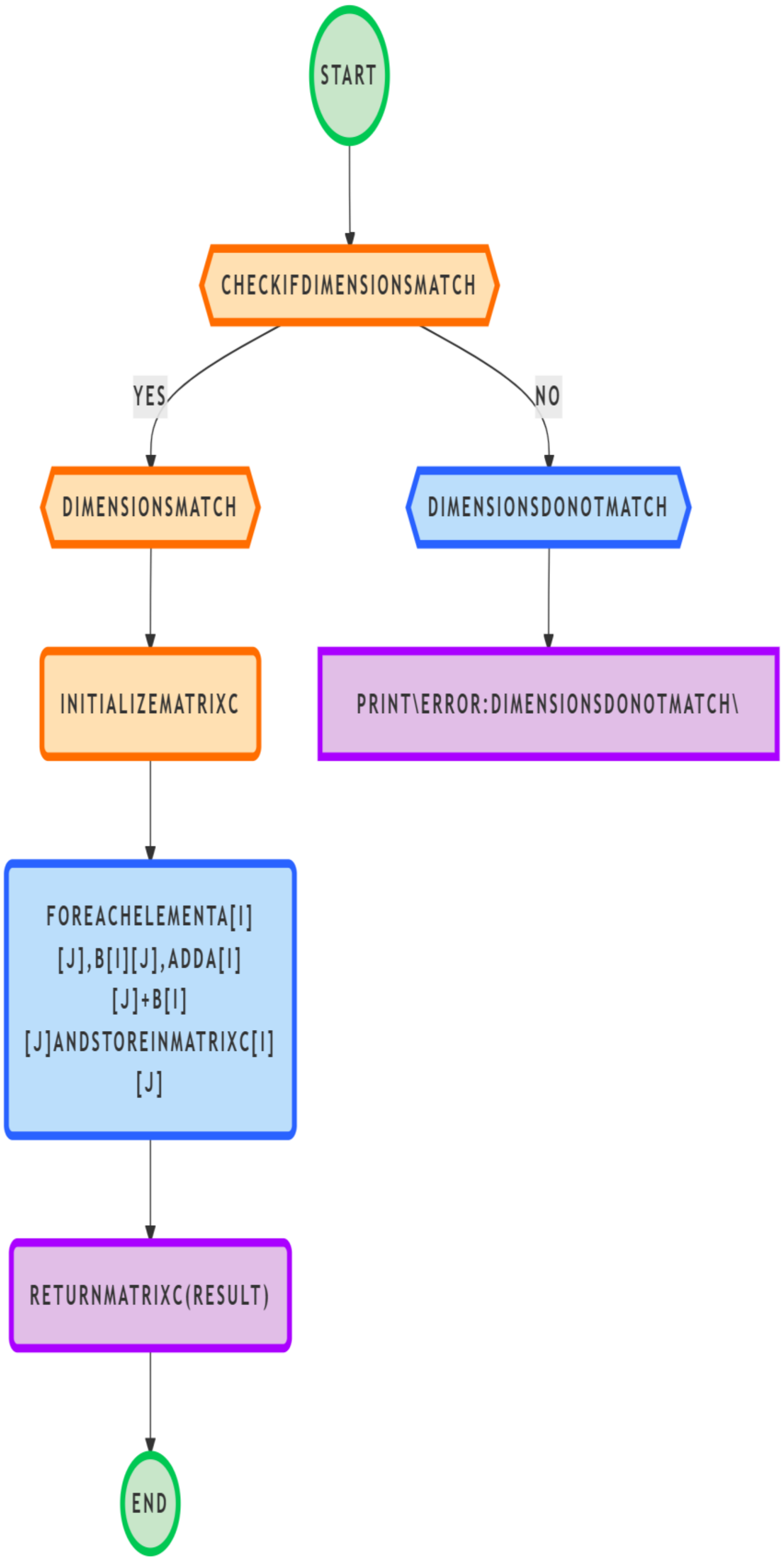
```
MatrixAddition (MatrixA, MatrixB)
    if number of rows of MatrixA ≠ number of rows of Matrix B or
       number of columns of MatrixA ≠ number of columns of MatrixB then
          print “Matrices cannot be added due to different dimensions. “
          return “Error”
end if

Initialize MatrixC with the same dimensions as MatrixA and MatrixB

For I = 1 to number of rows in MatrixA do
    For j = 1 to number of columns in MatrixA do
        MatrixC[i][j] = Matrix[i][j] + MatrixB[i][j]
    end for
end for

return MatrixC

End
```


START
CHECKIFDIMENSIONSMATCH
YES
NO
DIMENSIONSMATCH
DIMENSIONSDONOTMATCH
INITIALIZEMATRIXC
PRINT\ERROR:DIMENSIONSDONOTMATCH\
FOREACHELEMENTA[I][J],B[I][J],ADDA[I][J]+B[I][J]ANDSTOREINMATRIXC[I][J]
RETURNMATRIXC(RESULT)
END

Due to time constraints I did not represent it in Blocky but have done a python code to represent as below.

```
def add_matrices(matrix_a, matrix_b):
#Check if matrices have the same dimensions
if len(matrix_a) != len(matrix_b) or len(matrix_a[0]) != len(matrix-b[0]):
    raise ValueError(''Matrices must have the same dimensions")

#Initialize result matrix
matrix_c = []

# Element-wise addition
for i in range(len(matrix_a)):
    row = []
  for j in range(len(matrix_a[i])):
    row.append(matrix_a[i][j] + matrix_b[i][j])
  matrix_c.append(row)

return matrix_c

#Example usage of values to test the adding matrices concept
matrix_a = [[1, 2], [3,4]]
matrix_b = [[5,6], [7,8]]
result = add_matrices (matrix_a, ,matrix_b)
print ("Matrix C:" , result)
```

Result generated as Matrix C: [[6, 8], [10, 12]]

3. - Give the sequence of steps in plain English, i.e. the algorithm, for searching node 7 in both trees A and tree B below. What makes both trees different, and what does that imply for the differences between the algorithms? Have you used any of the previously studied algorithms? If yes, on top of giving the sequence of steps, mentioned which one and explain in 3 lines what it does in general. Warning: The answer must not be a pseudocode nor a flowchart, but a list of steps. You must give two sequences named A and B, even if some steps are similar or the same.

Sequence of steps for Searching Node 7 in Both Trees A and B;

Tree A:

Tree Structure:

- 1-> (11,6), 11-> (2,7), 2-> (9), 7-> (14),6-> (3) and 3-> (5,4)

  Sequence of steps for searching Node 7 in Tree A:

  1. Start at node 1
  2. Check if node 1 is node 7. It is not
  3. From node 1, the choice is to go to node 11 or node 6, but first we go to node 11
  4. Check if node 7 is node 7 and it is node 7
  5. Node 7 found

     Result for Tree A: Node 7 us found after traversing 1-> 11 ->7

     <u>**Tree B**</u>
     Tree Structure

     10-> (5,13), 5-> (2,7), 2-> (1), 7-> (6), 13-> (20), 20-> (18,22)

     Sequence of Steps for Searching Node 7 in Tree B:
     1. Start at node 10
     2. Check if node 10 is node 7 which it is not
     3. From node 10, I have two choices: go to node 5 or node 13. I go to node 5
     4. Check if node 5 is node 7. It is not
     5. From node 5, I have two choices either to go to node 2 or node 7. I go to node 7
     6. Check if node 7 is node 7 and yes it is.
     7. Node 7 found

<u>**Result for Tree B is Node 7 is found after traversing 10 -> 5 -> 7**</u>

Comparison of Tree Structures and Algorithm Implications:

- Tree A has node 7 as a child and node 11, which is directly connected to node 1. This means that to find node 7, we can quickly go through 1->11->7.
- Tree B has a node 7 as child of node 5, which is further down the tree. To reach node 7, you must traverse 10-> 5 -> 7.

The two trees are different because;

1. Tree A has a simpler, more direct route to node 7, whereas Tree B has a more complicated path with more branching nodes to go through.

2. In Tree A, the node 7 is only a few steps away from the root (1->11->7), whereas in Tree B, node 7 is further down the tree and requires passing through node 5 to reach it.

Implications for the Algorithms:

1. Tree A's simpler structure means that searching for node 7 would be faster in comparison to Tree B because there are fewer nodes to travers.
2. In Tree B, the traversal has more steps, as the search for node 7 passes through additional nodes like node 5 and node 10.

Algorithms Used:

Both of these search operations clearly shoe that Depth-First Search or DFS which are a common graph traversal algorithm.

1. **Depth –First Search** of DFS starts at the root node and explores as far as possible along each branch before backtracking. This algorithm is typically implemented using recursion or a stack. In these cases, the DFS starts at the root and explores each path unit it finds node 7 or exhaust the possibilities.

DFS generally works well for tree-like structures as it tries to go deep into one branch before moving to the next. If there is a path leading directly to node 7, it will be found quickly, but if the path is longer, it will take more steps. So the Tree A, the search for node 7 is quicker because the path is shorter 1->11->7. Tree B the search is bit longer, as it involves traversing through node 10 and node 5 before reaching node 7: 10->5->7.

4. **Dijkstra's Algorithm, to find out shortest path from A to Z**

Graphical representation:

- A: [(B,1), (G,10), (C,5)]
- B: [(A,1),(D,3)]
- C: [(A,5),(D,8), (E,6), (Z,9)]
- D: [(B,3), (F,1), (C,8)]
- E: [(G,3), (C,6), (Z,1)]
- F: [(D,1), (Z,6)]
- G: [(A,10), (E,3)]
- Z: [(E,1), (C,9), (F,6)]

Dijkstra's Algorithm:

We will use Dijkstra's algorithm to find the shortest path from node A to node Z. In Dijkstra's algorithm, we maintain a set of visited nodes and repeatedly select the node with the smallest tentative distance.

Step by step Workings for Dijkstras Algorithm.

1. Initialization:
   - Set the initial distance to A as 0 (dist[A] = 0)
   - Set the initial distance to all other nodes as infinity (dist[B] = inf, dist[C], etc.)
   - Start with node A.
2. First iteration (starting at A):
   - A has neighbor's B (distance 1), C (distance 5), and G (distance 10).
   - Update the distances
   - dist[B] = 1(A to B)
   - dist[C] = 5(A to C)
   - dist[G] = 10 (a to G)

Mark A as visited

3. Second iteration (next node with minimum distance, which is B):
   - B has neighbors A (distance 1) and D (distance 3).
   - A is already visited, so I ignore it.
   - Update dist[D]: dist[D] = dist[B] + 3 = 1 +3 =4.
   - **Mark B as visited.**
4. Third iteration (next node with minimum distance, which is C):
   - C has neighbors A (distance 5), D (distance 8), E (distance 6), and Z (distance 9).
   - **A is already visited.**
   - dist[D] is already 4, which is smaller than 8, so no update.
   - Update dist[E]: dist[E] = dist[C] + 6 = 5+6 = 11.
   - Update dist[Z]: dist[Z] =dist[C] +9 = 5+9 = 14
   - **Mark C as visited.**
5. Fourth Iteration (next mode with minimum distance, which is D):
   - D has neighbors B (distance 3), F (distance 1), and C (distance 8).
   - B and C are already visited.
   - Update dist[F]: dist[F] = dist[D] + 1 = 4+1 = 5
   - **Mark D as visited.**
6. Fifth iteration (next mode with minimum distance, which is F):
   - F has neighbors D (distance 1) and Z (distance 6).
   - D is already visited
   - Update dist[Z]: dist [Z] = dist[F] + 6 = 5 +6 =11.
   - **Mark F as visited.**
7. Sixth iteration (next node with minimum distance, which is E):
   - E has neighbor's G (distance 3), C (distance 6), and Z (distance 1).

- C is already visited.
- Update dist[Z]: dist[Z] = dist[E] + 1 = 11 +1 = 12
- **Mark E as visited.**

8. Seventh iteration (next node with minimum distance, which is Z):
   - Z has already been reached with the minimum distance.
   - No updates needed

Final Shortest Path from A to Z:

- **Shortest path: A -> B -> D -> F -> Z**
- **Total cost is 11**

<u>**Minimum Spanning Tree using Prim's Algorithm (Starting C)**</u>

Step by step workings for Prim's Algorithm:

Prim's algorithms build the **Minimum Spanning Tree** (MST) by starting at an arbitrary node and growing the MST one edge at a time, always adding the smallest edge that connects a new node to the MST.

1. Initialization
   - Starts at C.
   - MST contains only node C
   - The neighbors of C are A (5), D (8), E (6), and Z (9).

2. First iteration (add the edge with the smallest weight):
   - The smallest edge from C is (A, 5).
   - Add A to the MST
   - Now the MST contains C and A.
   - Updating the potential edges: A -> B (1), A- G (10), A_> C (5)
3. Second iteration (add the edge with the smallest weight):
   - The smallest edge is (B,1)
   - Add B to the MST.
   - Now, the MST contains C, A and B.
   - Updating the potential edges: B-> D (3), B-> A (1).
4. Third iteration (add the edge with the smallest weight):
   - The smallest edge is (B, D, 3)
   - Add D to the MST.
   - Now, the MST contains C, A, B, and D
5. Fourth iteration (add the edge with the smallest weight):
   - The smallest edge is (D, F,1).
   - Add F to the MST.

- Now, the MST contains C, A, B, D, and F.

6. Fifth iteration (add the edge with the smallest weight):
   - The smallest edge is (F,Z,6).
   - Add Z to the MST.
   - Now, the MST contains C, A, B, D, F, and Z.
7. Sixth iteration (add the edge with the smallest weight):
   - The smallest edge is (Z, E,1).
   - Add E to the MST.
8. **Completion: The MST is now complete. The edge in the MST are**:
   - (C, A, 5)
   - (A, B, 1)
   - (B, D, 3)
   - (D, F, 1)
   - (F, Z, 6)
   - (Z, E, 1)

New Minimum Shortest Path from A to Z (After B goes Down)

If B is down, all connection to and from B are unusable. This will affect the shortest path because we can no longer use B TO GET TO other nodes.

Graph without B:

- **A ->** (G,10), (C,5)
- C -> (A,5), (D,8), (E,6), (Z, 9)
- D -> (F, 1), (C, 8)
- E -> (G, 3), (C, 6), (Z,1)
- F -> (D, 1), (Z, 6)
- G - > (A, 10), (E, 3)
- Z -> (E, 1), (C, 9), (F, 6)

As per the result we can apply Dijkstra's algorithm again, but with B's connections removed. The new shortest path may be different from before.

Finally, the Minimum Spanning Tree using Kruskal's Algorithm.

Kruskal's algorithm is a different way of constructing the MST by adding edges in order of their weight, making sure no cycles are formed.

1. **Sort the edges by weight**.

2. **Add the smallest edge** to the MST that does not forma cycle.
3. Continue adding edges until the MST is complete.