1. **Theory**

1. **Lower Bound:**  
   Since we need to check if list A has an equal element in list B, we will need to check through every element in A so it would n comparisons (iterating once). Since there is n comparisons occurring the lower bound time would be order of **O(n)**.
2. **Algorithm:**  
   1. Copy List A to a new List C.  
   for(int i = 0; i<n; i++){

A[i] = C[i]

};

2. Increment all values in List B by 1  
for(int i = 0; i<n; i++){

B[i] += 1;

};

3. Compare List A to List C. If A does not equal C, that means that there is at least one element in List A that is also present in List B. On the other hand, if A does equal C, this will mean that no elements are shared between List A and List B.

for(int i = 0; i<n; i++){

if(A[i] != C[i];

cout<<”share an element”};

4. Revert List B back to its original form.

for(int i = 0; i<n; i++){

B[i] -= 1;

};

Conclusion: There is four nested for loops present in this algorithm so the time complexity will be O(4n) which when reduced to our lower bound matches **O(n).**

2. **On-line Median**To improve this function we must make the insert/delete function shift the nodes according to the given value of k (n/2). The functions are implemented like an AVL tree method which we learned about in class but differs with a shift node function at the end of each insert/delete function.  
  
**Algorithm  
Text

Description automatically generated  
  
Conclusion**The insert/delete function will be slower than O(log2n) but the find function now has a time complexity of O(1) since we are only directly accessing the root of the tree to get the value which is an improvement from the previous time complexity of O(log2n). On the other hand, the space complexity for the insert/delete functions are still the same as they were at **O(log2n**) respectively as well as the find function having the same complexity of **O(1).**  
3. **Multiplying rectangular matrices**To multiply rectangular matrices in the most efficient way we must first construct a n x n matrix. The average work of completing the multiplication would be the total sum of the scalar multiplications of the k values described from the test book divided by the total number of k values used in the operation.  
  
**To get k values:**S[i,j] = S[i,j] = S[i,k]+S[k+1,j]  
  
**Algorithm for multiplying rectangular matrices**A screenshot of a computer

Description automatically generated with medium confidence

**Time Complexity: O(n^3), Space Complexity: O(n^2)**

4.   
For this programming question I will be using C++ to implement the algorithm stated on p. 126. For n values I will use 16,64, 256, 1024, 4096, and 16384 and for m values I will have m1 = 1,677,721,600 and m2 = 13,421,772,800. My code is shown below for the algorithm:  
  
**C++ Code in VSCode**  
Text

Description automatically generated **A screenshot of a computer

Description automatically generated with medium confidence**  
**Results for m1 = 1,677,721,600**Text

Description automatically generated

**Results for m2 = 13,421,772,800**Text, chat or text message

Description automatically generated

**Findings**I found that this program was unable to run on my personal computer which shocked me. My computer is pretty up to date with 16gb of DDR4 ram, so I was surprised that this program took even more memory than that to run. Instead, I opted to run this program on an online compiler which supports VMM to carry out this experiment. After analyzing my timings above for m1 and m2 I found that the time calculated by the program is relatively similar between the different values of m. For the first three n sizes m2 resulted in shorter times but next two n sizes m1 showed it calculated the program faster. This so far showed me that regardless of the size of the matrix m which translates to the number of calculations, the time does not linearly vary in either direction. As for n = 16,384 I was surprised to see my program ran faster for m2. I would like to also note the huge jump in timings from n = 256 to n = 1024. Almost doubling in time, I believe this is a dir3ect result of the physical memory of the server being used fully and not being enough to run the calculations so for the larger values, the server used VMM to temporarily transfer some data from the ram to the disk storage. This resulted in the much larger timings we can see in the second half of the findings. This is also supported by the values of the first three n’s being small, so they are able to be computed using only physical memory resulting in drastically reduced timings. The complexity of this program would be **O(m)** where m is the m value of calculations because each instance has to go through the loop m times to calculate the sum in the end.

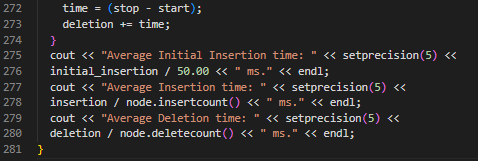
5. **AVL Tree Timings**

**C++ Code in VSCode**Text

Description automatically generated **Text

Description automatically generated** Text

Description automatically generatedText

Description automatically generated  
  
**C++ Output  
  
  
Conclusion**As seen from the C++ output above, my initial insertion time which is the average time to insert te first 50 nodes into the tree were insanely low and only showed me 0 milliseconds. The average insertion time was the time on average to insert one node from/into the tree; on the other hand, the average deletion time was the time on average to delete one node from/into the tree. Since memory fragmentation occurs when the program engages in either insertion/deletion it results as the time for insertion/deletion being higher than the average initial insertion time. The cause of this is because the program must engage in memory compaction to close gaps in the memory which takes up more time than the program just inserting the first 50 nodes into the tree. These timings were to be expected therefore, I believe that my program is running correctly and displays accurat4e timings for the three output timings shown above.

6. **VMM**

**Hypothesis**I believe that as our memory size grows that our timing will increase with it linearly. As more memory is needed in these tasks our timing through VMM is kept reasonable but will increase nonetheless and I expect to see the growth through the program below:  
  
**C++ Code in VSCode  
Text

Description automatically generated  
  
Text

Description automatically generated**

**Results**  
A picture containing text, monitor, black, screenshot

Description automatically generatedA black screen with white text

Description automatically generated with low confidence

**Findings**For this program there was many timings and memory numbers to evaluate so I will break up my explanation into parts. Firstly, I would like to acknowledge that there is a clear positive linear relationship between the memory size and the time elapsed. As seen in the pictures above, as our C goes up our time also goes up without every dipping below the previous value. Secondly, from my findings I can see that from C = 0.5M to C = 2M there was very minimal timing difference almost unnoticeable. Once we pass C = 2M and enter C = 5M we can see that there is more than double the time needed to complete this than the previous test. We can see around this size we increase in larger and larger values in timings. The largest jump in timing occurs from C = 10M to C = 50M which is to be expected as this is the biggest jump in our program in memory size. This experiment has shown me that VMM is very useful as it shows a linear growth rate in timings instead of an exponential increase.

7. **Dijkstra’s Algorithm**

**Explanation**Dijkstra’s algorithm is used to determine the shortest path/distance between an initialized starting node to the rest of the node in the graph presented. The algorithm calculates shortest distances from the starting node and excludes the longer paths calculated when making an update. The steps to this algorithm as listed below:  
  
**Algorithm**  
1. Create an adjacency matrix for a given graph V x V, V being vertices.  
2. Initialize a starting node you will use to calculate distance to other nodes from.  
3. Set the distances in the adjacency matrix to INFINITY.   
4. Calculate the distances to all neighboring nodes by summing its distance with the weights of the edges.  
5. If the calculated distance is less than the current one, replace the distance already present and set the new smaller distance into the adjacency matrix.  
6. The node with minimal temporary distance we will set as active and mark its distance as permanent.  
7. We repeat steps 4-7 until there are no nodes in our matrix left with a permanent distance which have neighboring nodes with temporary distances.   
  
**Time Complexity: O(V2) Where V is the number of vertices in our adjacency matrix/ graph.**

**C++ Code in VSCode**Text

Description automatically generatedText

Description automatically generated

**Example 1 Graph**Chart, line chart

Description automatically generated **Example 1 Adjacency Matrix  
Text

Description automatically generated  
Example 1 Output**Text

Description automatically generated **Testing If Paths are Correct  
Chart, line chart

Description automatically generated** **Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated**Chart, line chart

Description automatically generated  
   
**Chart, line chart

Description automatically generated**

**Example 2 Graph**Chart, line chart

Description automatically generated **Example 2 Adjacency Matrix  
Text

Description automatically generated  
  
  
Example 2 Output**Text

Description automatically generated with medium confidence **Testing If Paths are Correct  
Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated  
Chart, line chart

Description automatically generated** **Chart, line chart

Description automatically generated**  
**Chart, line chart

Description automatically generated  
  
Conclusion**My code implementation of Dijkstra’s algorithm initializes a matrix which I have included above with example graphs. Once it goes through the matrix the shortest path from the starting node (1 in the graph, 0 in the output) to every other node with a valid path is shown in my output images above. Using the logic, I showed in my steps listed above I was able to implement this algorithm using C++ and was able to successfully carry out Dijkstra’s algorithm finding the shortest path for any sized graph thrown at my algorithm.

**Thank you for grading my homework!**