**COSC 3320  
Leiss  
Saim Ali  
ID: 2100224  
Assignment 1**  
  
1. **TOH**

(a) To design an algorithm to solve this unique version of the Towers of Hanoi game we must first consider the rules to the game. For this problem we will be considering disks as the value “n” and the range of n is 1-10 meaning we can solve for up to 10 disks at a time. For our specific given game, a given disk n can be moved from S to D but not back to S once it has left. As for the pegs D, A1, A2, A3, A4 this is where most of our game will take place in order to solve this problem. Our goal is to get all of the disks to peg D while also not breaking the rules of the game. We also cannot place a larger disk on top of a smaller one.

In order to solve this problem in f(n) moves we must first start with sending n-1 disks to peg A1. Once the n-1 disks are at A1 we can then move the largest disk n from S to D. Now we have our largest disk in the place we need, and it will not move for the rest of the game. Now using A1, A2, A3, A4 we send the n-1 disks parked at A1 to D. We can do this by using A2, A3, A4 to park disks while we send the largest disk n from the stack from A1 to D one at a time. This function that I have just described will be the main function that will be used to solve this while also using helper functions to achieve tasks such as moving n-1 from S to A1.  
  
Using the pseudocode below the time complexity I compiled was: **O(3^n)** since the main function to solve this would be:   
*Void sTOd() {*

*If (n==1) move from s to d*

*If (n>=2) move n-1 from s to a1 (void sTOa1)   
move n from s to d  
move n-1 from a1 to d (void a1TOd)*

*}*

**Pseudocode for my algorithm:**

*Void sTOd() {*

*If (n==1) move from s to d*

*If (n>=2) move n-1 from s to a1 (void sTOa1)  
move n from s to d  
move n-1 from a1 to d (void a1TOd)*

*}*

*Void sTOa1() {*

*If n == 1 move from s to d then d to a1  
if n>=2 move n-1 from s to a1 (sTOa1)  
move n from s to d  
move n-1 from a1 to a4 (a1TOa4)  
move n from d to a1  
move n-1 from a4 to a1 (oneHOP)*

*}*

*Void a1Toa4{*

*If n == 1 from n to a4*

*If n>=2 move n-1 to a4  
move n to a3*

*Move n-1 to a1*

*Move n to a4*

*Move n-1 from a1 to a4  
}*

*Void oneHop{*

*If n==1 move a4 to a1*

*If n>=2 move n-1 from a4 to a2*

*Move n to a1*

*Move n-1 from a2 to a4*

*Move n-1 from a4 to a1*

*}*

*Void a1TOd{*

*If n==1 move n from a1 to d*

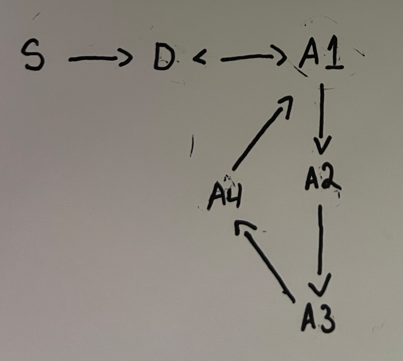
*If n>=2 move n-1 to a1*

*Move n from a1 to d*

*Move n-1 from a4 to a1*

*Move n-1 from a1 to d*

*}*

**Picture example of the Hanoi problem given**  
  
  
  
  
  
  
  
  
  
  
  
  
  
(b)   
**C++ Code**Text

Description automatically generated**Text

Description automatically generatedText

Description automatically generated  
  
Text

Description automatically generated  
  
  
Output Moves:**Text

Description automatically generatedA picture containing text, electronics

Description automatically generatedA screen shot of a computer

Description automatically generated with low confidence **Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated**Table

Description automatically generated **Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated with low confidence  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated  
Table

Description automatically generated**

2. **Row/col major**(a) Row Major Order:  
For A[I, J] = A[I, J] \* B[I, J]  
A read = 4pg, A write = 4pg  
B read = 4pg, B write = 4pg

Total = 16pg  
  
C read = (C[N – I +1,J]) = 4pg  
C read = (C[J, I])  
2 reads each column = 4000pg  
  
So we get: 4 + 4 + 4 + 4 + 4 + 4000 = 4020pg for each given I  
So finally the answer is: 4020 \* 4000 = **16,080,000 total**  
(b) Column Major Order:  
For column major: 4000 + 4000 + 4000 + 4000 + 4000 + 4 = 20,004pgfor each given   
The answer will be calculated this way: 20,004 \* 4000 = **80,016,000 total**

3. **Quicksort**

(a) In order to do quicksort on an array of size n assuming that the pivot element x is the penultimate (2nd to last) element of the array we can use this information to compute a best-case scenario. The most optimal execution would occur when the second to last element of the array (x) is the median of the array. Since the element x will be the median of the list it will result in the array of size n being split evenly on both sides of the pivot x. The left side will consist of elements <x and the right side will compose of elements >x making this the best possible case for our quicksort algorithm with the time complexity of O(nlogn).

(b) When looking for possible slowest ways to execute quicksort when our 2nd to last element is the pivot we can come up with the following cases:  
1) The array is already sorted.  
2) All the elements of the array are of the same value.  
3) The array is sorted in a reversed order – When our array is sorted in a reversed order this is the worst case because you will have to do the highest number of iterations to your array and move every element in order to get the correctly ordered array. The result for these cases would be O(n^2).

4. **VMM**  
**Hypothesis:**For this programming experiment my hypothesis is that out of the ten n values given the timing on the first five should be very low but the last five should be far greater but still should have similar timings between Version 1 and Version 2. I believe these versions will have similar timings because they both state the same number of functions in each respective program leading me to believe they will also perform the same. **Experiment:**

I used C++ for my programming language and compiled in Visual Studio Code. First, I started with creating a program that executes Versions 1 and 2 for all ten values of n stated: (100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, and 51200). The program is shown below. Then I compiled the program to run for all of the sizes of n and record the timings needed for each. The results are also shown below. Also note that the time required to initialize the given matrices A and B will not be part of my timing measurements as stated in the homework problem.  
  
**Results**

(C++)  
A screenshot of a computer

Description automatically generated with medium confidence  
Text

Description automatically generated  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**(Output)**  
Text

Description automatically generated  
  
**Findings:**

When looking at the timings gathered for all of the ten n values, we can see there is normality for up to the fourth n value of 100 as the timings (in milliseconds) only differs by 2 when comparing both versions ran. However, when we start looking at the higher values of n we can see there is a clear and shocking distinction between the timings of Version 1 and Version 2. Version 1 seems to be performing more than 3x as fast as Version 2 by the time our n reaches 25600 (1479 and 6979 ms respectively). After some research I found the reasoning for Version 1’s success in timings was due to the way the version accessed the elements of the given matrix. Version 1 used a row-major approach while Version 2 uses a column-major approach. C++ allocates its memory in row-major order which means that Version 1 will always have the upper hand when compiled in C++ making it exceptionally faster than Version 2 in this case. I also noticed that my computer could not reach the 10th n value of 51200 as my program would throw a termination error as it failed to allocate memory. This is due to the high size of the n value which is impossible to run on the hardware I was conducting this experiment on.

5. **Memory fragmentation in C**  
**Hypothesis:**My Hypothesis that for any given m (array size) I choose to execute my C++ program with, the situation where the timing for the allocations of the m arrays of size 1.4 MB each will be the most time consuming. I believe this because the array size of 1.4 MB is greater than the first two scenarios resulting in a larger time. For (a) I was able to use the malloc function to allocate 1 MB sized m arrays and the clock\_t function to record the timings. For (b) I used the free function to free all even numbered arrays and recorded the timings. Finally, for (c) I used malloc again but this time around for 1.4 MB sized m arrays. The results are shown below as well as the code.  
 **Experiment:**

For this experiment I used 1,048,576 (1024 \* 1024) as the size of 1 MB and 1,468,006.4 (1024 \* 1024 \* 1.4) as the size for 1.4 MB to represent the sizes of the m arrays more accurately. As for the m value I will try to positively increment the number until most of my computers main memory is exhausted leading to a crash or error appearing. In order to come up with the actual code to execute the three given actions I started with researching how to allocate memory in C. I realized that the use of pointers and a loop with the array size was necessary.

**C++ Code**Text

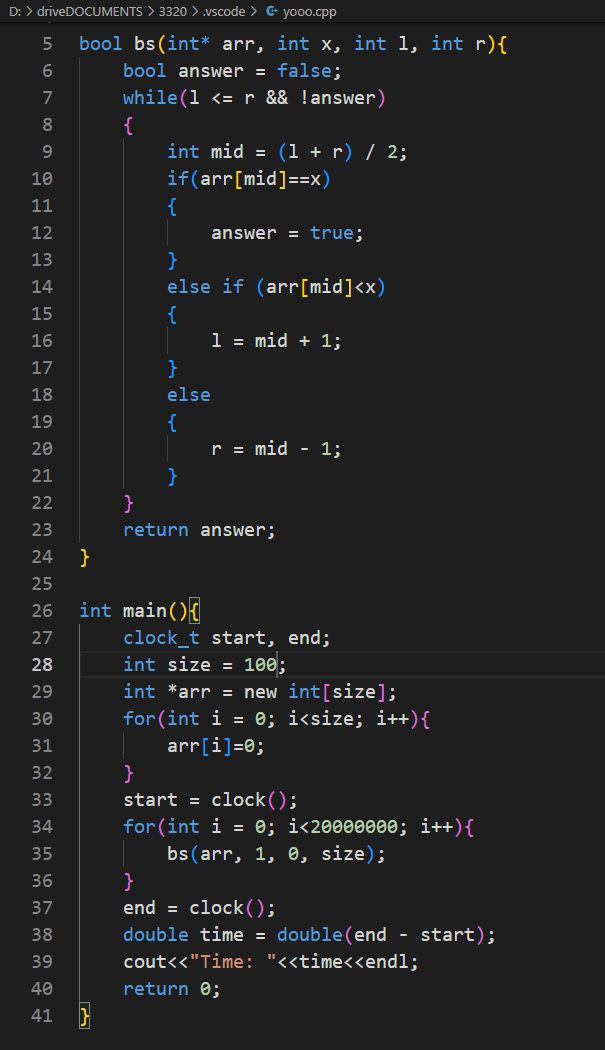
Description automatically generated  
**Results:**After experimenting with countless different m values I found that my computers main memory capped off at m = 1,000,000 and any larger value would result in a black screen and the need of a full system restart. The timings for the three actions are given below and I can gladly say that in this problem my hypothesis was correct based on the results collected. To allocate 3m arrays of size 1MB it took me approximately 3.6 seconds at m = 1,000,000 while the deallocation of the even numbered arrays took approximately 2.6 seconds. This shows that the initial allocation took more amount of time than the deallocation of the even numbered arrays and I can inference that this is due to only the even numbered arrays being deallocated resulting in only about half of the arrays being deallocated out of 3\*m arrays. On the other hand, the allocation of m arrays of size 1.4 MB took a hefty 615.7 seconds which is beyond my expectations in the difference in timings between the three situations. This was a shock to me to say the least even though I predicted this one would take the longest I did not expect it to show this significant jump in timing. I believe the most obvious reason for this would be the change in the size of the arrays from 1 MB to 1.4 MB which results in the huge spike in execution timings. Nonetheless, I can determine from this experiment that the array size plays a huge factor in the execution time for memory allocation/deallocation.  
  
Text

Description automatically generated

6. **binary search in 3 diff languages**  
**Hypothesis:**My Hypothesis to test three different languages all running the same program which is binary search of an array in this case is that since binary search is a pretty low-level program with basic recursion, the time needed for execution of 20,000,000 unsuccessful searches will be relatively the same in the three programming languages (C++, Python, Java). I believe that although these programming languages all looks substantially different, they will perform the same when it comes to their execution time. **Experiment:**

To conduct this experiment, I first started with the basic implementation of the binary search algorithm in each of the three languages (C++, Python, Java). After creating the basic algorithm for binary search, I then started formulating a way I can direct my programs to execute 20,000,000 unsuccessful binary searches of an algorithm. In order for my programs to do the task I needed I did the following:  
  
1. Declared an array of the eight respective sizes given (100, 400, 1600, 6400, 25600, 102400, 409600, and 1638400)

2. Then I set the array declared in the previous set to be filled with all values of 0.  
3. After the test array had been setup, I formulated a loop that ran 20,000,000 unsuccessful searches by calling to my binary search function and recorded the time it took to execute all of the searches.

4. I ran each respective program with each size 5 times and calculated the average to record for my experiment.  
  
  
  
  
  
**Results:**  
  
**(1) C++  
**

**Size Time (ms)**100 283

400 353

1600 423

6400 494

25600 565

102400 635

409600 708

1638400 776  
  
**(2) Python**  
**Text

Description automatically generated  
  
Size Time (ms)**100 12689

400 18341

1600 25094

6400 27151

25600 29638

102400 34077

409600 37051

1638400 41032

**(3) Java**  
**Text

Description automatically generated**  
  
**Size Time (ms)**100 202

400 280

1600 360

6400 445

25600 531

102400 612

409600 696

1638400 781 **Table for all 3 languages with times + log(N) \*worst times\***Table

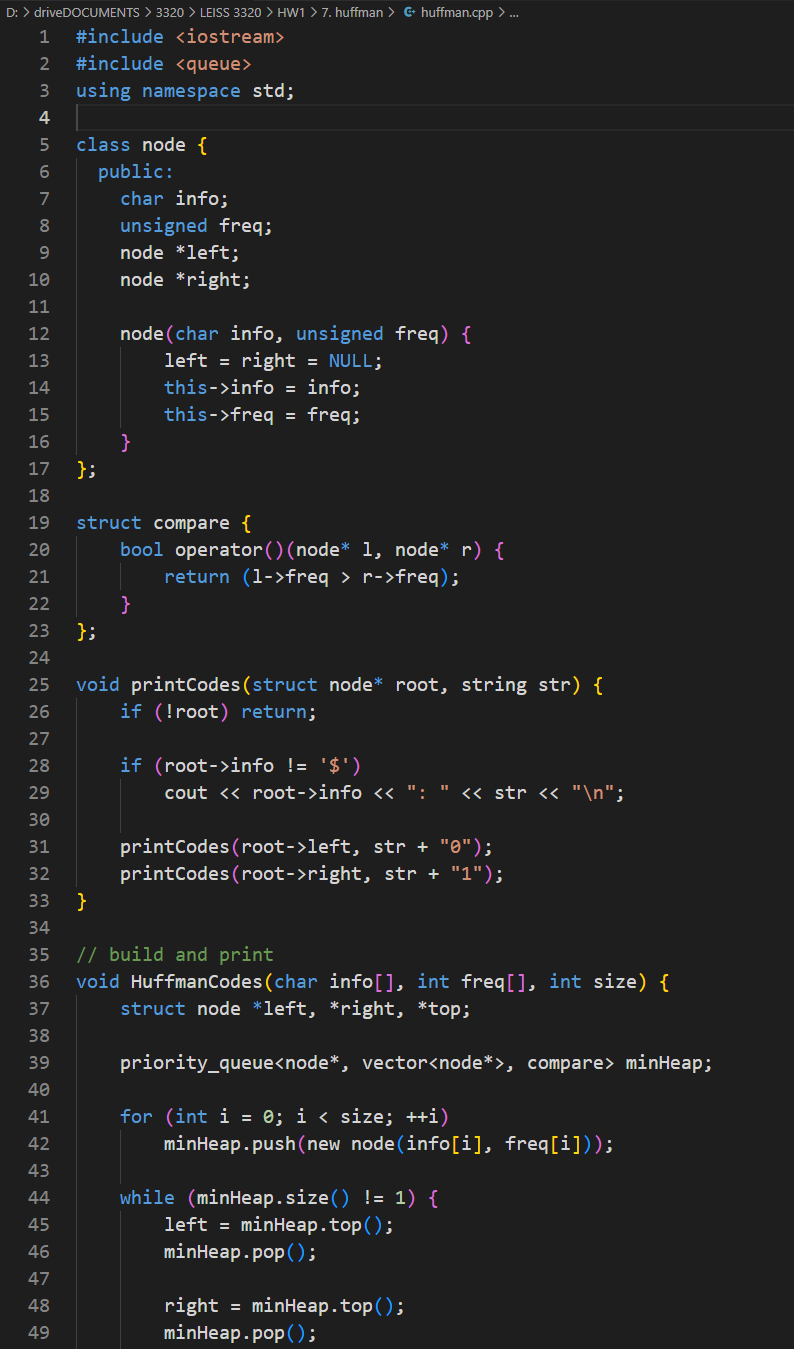
Description automatically generated with low confidence **Findings:**

If we examine the table above which contains all of the resulting times for my three programs, we can see there is a clear distinction in times. It is clear that Python execute times were substantially higher than Java and C++’s times respectively. I was shocked to even be able to notice the speed difference in my tests without a timer present, it was clear that Python was taking a long time to execute the same program that completed in under a fraction of a second in C++ or Java. After this experiment I evaluated the differences in the three languages and what differed so greatly in Python. Since Python evaluates its written code at runtime rather than compiling the code at compile time this makes Python the slower language in this experiment. Although slower, Python is undoubtably the fastest language out of the three to write code in as seen above the code I wrote was short and concise while the Java and C++ codes were slightly longer. This makes Python a great pick to write code faster and efficiently but also adds to the fact that Python will execute functions such as binary search way slower than languages such as C++ and Java. Now that we have determined Python to be the slowest amongst the three, we can compare C++ and Java. Since C++ is compiled to binaries it runs almost immediately which therefore gives great execution times for the program. Java on the other hand had the lowest runtimes for many of the array sizes in the above table. I believe this can be explained by evaluating the compiler I was running and comparing the versions of each language I was executing. I believe my Java version is the latest as I just redownloaded it, but my C++ version is the same version I have been using for quite some time now, so it is outdated. Factors such as these can come into play when timings are so close between languages and therefore, I cannot determine whether Java is always going to be faster than C++ in binary search execution times. To address my hypothesis that all three programs will have similar timings, I was clearly wrong from the data I collected. I proved my hypothesis to be false by showing the clear difference in timings between the languages most notably how slow Python was compared to C++ and Java.

7. **Huffman code program**  
**Optimal Algorithm Implementation**

To follow the Huffman Code Algorithm (O(nlog2n)), we must take the following steps:

1. We will be creating a leaf node for each character in our example (a, b, c, d, e f, g) and then use them to create a min heap of all the nodes we created. The min heap we create is used as a priority queue and the value of frequency is sued to compare two nodes in the heap. To start the least frequency character will be the root.
2. Next, we remove two nodes from the min heap and these two will be chosen based on the lowest frequency values.
3. Now, we will create an internal node with frequency equal to the sum of the two nodes picked in the earlier steps frequencies. The first node picked will serve as the left child and the second node picked will be the right child. Then add this node to the min heap.
4. We will repeat steps 2 and 3 until the heap is down to only one node and that last node will be the root and the tree will be complete indicating we have successfully executed the Huffman Algorithm.

**Code Source: Geeks4Geeks.com // Aditya Goel**  
 Text

Description automatically generated