Minh Duc Nguyen

ID: 2069407

**COSC 3320 - Assignment 1**

1. **Towers of Hanoi game**
2. Algorithm

The algorithm for this game would require a group of recursive functions, in order to move every disk from S to D, using a cycle with 4 A(s) (A1->A2->A3->A4->A1).

We will have to move disks from S to A4, then move the disks back to A1, A2, and keep going until the largest disk will be the only one in A4. We can move that disk from A4 to D, then keep doing again and again for the other disks.

Time complexity: O(2^n)

Space complexity: O(n)

1. Implement the algorithm

*Output:*

Enter the number of disks: 1

Total moves for 1 disk: 5

Enter the number of disks: 2

Total moves for 2 disks: 14

Enter the number of disks: 3

Total moves for 3 disks: 35

Enter the number of disks: 4

Total moves for 4 disks: 92

Enter the number of disks: 5

Total moves for 5 disks: 257

Enter the number of disks: 6

Total moves for 6 disks: 746

Enter the number of disks: 7

Total moves for 7 disks: 2207

Enter the number of disks: 8

Total moves for 8 disks: 6584

Enter the number of disks: 9

Total moves for 9 disks: 19709

Enter the number of disks: 10

Total moves for 10 disks: 59078

1. **Page transfers**
2. In row-major order

for I := 1 to 4000:

for J := 1 to 4000:

A[I, J] = A[I, J] \* B[I, J] (1)

B[I, J] = C[J, N-I+1] \* A[J, I] (2)

(1): Request A[I, J], read A[I,…]

Request B[I, J], read B[I,…]

Update A[I, J] 🡪 write

(2): Request C[J, N-I+1], read C[J,…]

Request A[J, I], read A[J,…]

Update B[I, J] 🡪 write

Request A[I, J+1]

Request B[I, J+1]

Update A[I, J+1]

Request C[J+1, N-I+1], read C[J+1,…]

Request A[J+1, I], read A[J+1,…]

Update B[I, J+1]

Then in total it needs 2x4000 + 2 = 8002 transfers per I value.

Add the first and second statement totals will be 6 + 8002 = 8008 transfer per I value.

So in row-major order, there will be 4000 x 8008 = 32,032,000 transfers.

1. In column-major order
2. **QuickSort** on the array A[1:n] and assume that the pivot element x is the last element of the array to be split (A[hi])
3. Execute optimally (A[lo:m] and A[m:hi] are always of equal size). The best case is when the pivot is the middle element of the array. Time complexity: O(nlog(n))
4. Execute in the slowest possible way: when the array is already sorted. Time complexity: O(n^2)
5. **Matrix addition:**

Below is my code:

|  |  |
| --- | --- |
| *Version 1* | *Version 2* |
| Text  Description automatically generated | Text  Description automatically generated |

Timings:

|  |  |
| --- | --- |
| *Version 1* | *Version 2* |
| Text  Description automatically generated | Text  Description automatically generated |

When the value of n is double, the time will be approximately 4 times as the previous one. It is not exact as theory since there are several factors affected the results, such as computer hardware, software, other running background tasks in the computer, etc.… Also, from my results, if we use version 2, as the size is increasing, the time will also be more than the version 1. The reason might be because of my testing process. I ran the second version 2 after the first one, so the computer might not release the full memory for the next test (some running backgrounds left), and it could be the reason for the big difference.

1. **Memory fragmentation in C:**

C-program:

Text

Description automatically generated

And here is the result:

A screenshot of a computer

Description automatically generated with medium confidence

Explain the timings:

m = 15000

Total number of arrays = 15,000 x 3 = 45,000

Time needed for the allocation memory of the 45,000 arrays of size 1MB = 10^6 bytes each is approximately 0.003516 Sec. Because the amount of memory needed for 45,000 arrays and each of them is 1,000,000 bytes, we can see that it’s a large number, so it takes time.

Time needed for the allocation memory of the 15,000 arrays of size 1.25MB = 1.25 x 10^6 bytes each is approximately 0.001455 Sec. As we can see here, even the size of each element in the arrays is bigger, however, we just need to allocate memory for m arrays (15,000 arrays) this time. Therefore, the time needed will be less than the time required for the allocation of the 3m arrays of size 1MB each.

1. **Binary Search:**

Python code:

|  |  |
| --- | --- |
| Text  Description automatically generated | A screenshot of a computer  Description automatically generated with medium confidence |

C++ code:

|  |  |
| --- | --- |
| Text  Description automatically generated | A screenshot of a computer  Description automatically generated with medium confidence |

Java code:

|  |  |
| --- | --- |
| Text  Description automatically generated | A screenshot of a computer  Description automatically generated with medium confidence |

Timings and observations:

From my results, Java takes less time than C++ and Python. There is a big different between C++ and Python, because Python is an interpreted language, it takes longer to execute.

However, both are approximately comparable to log(n) time.

1. **Huffman algorithm:**

Huffman Node class:

Text

Description automatically generated

Main class:

Text

Description automatically generated

Output:

A screenshot of a computer

Description automatically generated with medium confidence

Time complexity:

O(nlogn) where n is the number of unique characters.

If there are n nodes, extract min is called 2\*(n – 1) times. Extract min takes O(logn) time as it calles minHeapify().

So, overall complexity is O(nlogn).