# Assignment 2 - COSC 3320

Alex Bennett (ID: 1901408)

## Theory Problem 1

### (a)

To check if an element in $L_1$ is in $L_2$ we must iterate over each element of $L_1$ ($n$ elements), and then compare each element of $L_1$ with every element of $L_2$. Thus the lower bound is $O(n^2)$.

### (b)

## Theory Problem 2

## Theory Problem 3

To determine the average number of scalar multiplications for a sequence of $n$ matrices we will use the following informal algorithm:

$S[i, i] = 0$
$S[i, i+1] = p_i + p_{i+1} + p_{i+2}$
$S[i, j] = \text{avg}(S[i, k] + S[k+1, j] + p_i + p_{k+1} + p_{k+1}), \text{ for } i \leq k \leq j-1$

Where $S[i, j]$ is the matrix representing the average work for a parentheses configuration grouping every element from $i$ to $j$ and $p_i$ represents the $i$th dimension of the original matrix sequence. The average work is equal to the sum of scalar multiplications for possible k-values divided by the total number of k-values.

Since this algorithm will be iterated over 3 nested for-loops the time complexity is $O(n^3)$. Since $S$ is two dimensional our space complexity is $O(n^2)$

# Programming Problem 1

## Procedure

In this program we use three two-dimensional arrays $X, Y$, and $Z$, each of size $n$, with $X$ and $Y$ initialized to 1 for each entry. We then add corresponding entries of $X$ and $Y$ together via matrix addition to generate an entry in $Z$. We use two different algorithms to perform the addition, the first traversing the arrays in row-major order and the second traversing them in column-major order, and compare their timings.

## Hypothesis

Given that both algorithms have the same number of operations, one might expect that the timings for the two algorithms will be similar. However, as we've studied in class, if a multidimensional array is stored in memory using row-major order then row-major traversal is significantly faster than column-major traversal. Given that C++ must store these arrays in either row-major or column-major order, my hypothesis is that there will be significant difference in timing between the two algorithms.

## C++ Implementation

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

        char *arg = argv[1];
        int n = atoi(arg);

        int** X = new int*[n];
        int** Y = new int*[n];
        int** Z = new int*[n];

        //Initialize X, Y, Z
        for (int i = 0; i < n; i++) {
                X[i] = new int[n];
                Y[i] = new int[n];
```

```
            Z[i] = new int[n];
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                X[i][j] = 1;
                Y[i][j] = 1;
            }
        }

        //Version 1
        clock_t begin1 = clock();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                Z[i][j] = X[i][j] + Y[i][j];
            }
        }
        clock_t end1 = clock();

        //Version 2
        clock_t begin2 = clock();
        for (int j = 0; j < n; j++) {
            for (int i = 0; i < n; i++) {
                Z[i][j] = X[i][j] + Y[i][j];
            }
        }
        clock_t end2 = clock();

        int time1 = (double)(end1 - begin1)/(CLOCKS_PER_SEC/1000);
        int time2 = (double)(end2 - begin2)/(CLOCKS_PER_SEC/1000);

        printf("Version 1: %d milliseconds\n", time1);
        printf("Version 2: %d milliseconds\n", time2);

}
```

### Results

| Dimension of Array | Time of Vers. 1 (ms) | Time of Vers. 2 (ms) |
|---|---|---|
| 128 | 0 | 0 |
| 256 | 0 | 0 |
| 512 | 0 | 3 |
| 1024 | 5 | 22 |
| 2048 | 20 | 108 |
| 4096 | 81 | 545 |
| 8192 | 325 | 2609 |
| 16384 | 1421 | 15618 |
| 32768 | 11147 | 109160 |
| 65536 | 46973 | 824990 |

### Explanation

My hypothesis was correct: there was a substantial difference in timing between Version 1 and Version 2. As explained in the hypothesis depending on how programming languages store arrays in memory, row-major and column-major traversals of an array can vary quite significantly in their timings. Since Version 1 (row-major) was much faster than Version 2 (column-major), we can conclude that C++ stores multidimensional arrays in row-major order. In fact this is true for most languages of the C family.

## Programming Problem 2

### Procedure

### Hypothesis

### Java Implementation

```
import java.util.Random;

public class Prog2 {

        public static void main (String[] args) {

                int n = Integer.parseInt(args[0]);

                int[][] M = new int[n][n];
```

```java

        for (int k = 0; k < n; k++) {
            for (int q = 0; q < n; q++) {
                M[k][q] = 0;
            }
        }

        long start1 = System.currentTimeMillis();
        long m_1 = 1677721600;
        Random r = new Random();

        for (long k = 0; k < m_1; k++) {
            int x = r.nextInt(100) + 1;
            int j = r.nextInt(n);
            int i = r.nextInt(n);
            M[i][j] = M[i][j] + x;
        }

        long end1 = System.currentTimeMillis();

        long start2 = System.currentTimeMillis();
        long m_2 = 13421772800L;

        for (long k = 0; k < m_2; k++) {
            int x = r.nextInt(100) + 1;
            int j = r.nextInt(n);
            int i = r.nextInt(n);
            M[i][j] = M[i][j] + x;
        }

        long end2 = System.currentTimeMillis();

        System.out.println("Time elalpsed for n=" + n + " for m_1: " + (end1 - start1) + " milliseconds");
        System.out.println("Time elalpsed for n=" + n + " for m_2: " + (end2 - start2) + " milliseconds");


    }
}
```

### Results

| n | Time for m=1677721600 (ms) | Time for m=13421772800 (ms) |
|---|---|---|
| 16 | 55336 | 430628 |
| 64 | 55951 | 424845 |
| 256 | 59988 | 445868 |
| 1024 | 121313 | 795774 |
| 4096 | 232810 | 1858610 |
| 16384 | 286520 | 2290697 |

### Explanation

## Programming Problem 3

## Programming Problem 4

### Procedure

### Hypothesis

### Java Implementation

```java
import com.sun.management.OperatingSystemMXBean;
import java.lang.management.ManagementFactory;

public class Prog4 {

	public static void main (String[] args) {

		OperatingSystemMXBean opsys = ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
		System.out.println("Phys Mem: " + opsys.getFreePhysicalMemorySize());
		System.out.println("Virtual Mem: " + opsys.getCommittedVirtualMemorySize());
		System.out.println("Free swap space: " + opsys.getFreeSwapSpaceSize());

```

```java
                double[] C = {0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99, 1.0, 1.01, 1.1, 1.5, 2, 5, 10, 50};
                long freeBytes = opsys.getFreePhysicalMemorySize();

                for (int i = 0; i < 15; i++) {
                        System.out.println("-----------------------\n" + "Cache Size: " + C[i] + "*M");
                        long start = System.currentTimeMillis();

                        int numBytes = Math.abs((int)(C[i] * (freeBytes)));
                        int size = numBytes/4;
                        int[] testArr = new int[size];

                        System.out.println("Phys Mem: " + opsys.getFreePhysicalMemorySize());
                        System.out.println("Virtual Mem: " + opsys.getCommittedVirtualMemorySize());
                        System.out.println("Free swap space: " + opsys.getFreeSwapSpaceSize());

                        for (int j = 0; j < size; j++) {
                                testArr[j] = i + 1;
                        }
                        for (int j = 0; j < size; j++) {
                                testArr[j] -= 2;
                        }

                        System.out.println("Time elapsed: " + ((double) System.currentTimeMillis() - start) + " milliseconds");

                }

        }
}
```

### Results

| Cache Size | Free Physical Mem (bytes) | Free Virtual Mem (bytes) | Free Swap Space (bytes) | Time (ms) |
|---|---|---|---|---|
| Start | 292106240 | 10460585984 | 766246912 | N/A |
| 0.5*M | 142024704 | 10476437504 | 766246912 | 138 |
| 0.6*M | 105308160 | 10485899264 | 766246912 | 98 |
| 0.7*M | 92807168 | 10486038528 | 766246912 | 78 |
| 0.8*M | 78573568 | 10487099392 | 766246912 | 89 |
| 0.9*M | 78573568 | 10488160256 | 766246912 | 102 |
| 0.95*M | 78573568 | 10488160256 | 766246912 | 98 |
| 0.99*M | 78573568 | 10488160256 | 766246912 | 100 |
| 1.0*M | 78573568 | 10496548864 | 766246912 | 99 |
| 1.01*M | 78573568 | 10496548864 | 766246912 | 100 |
| 1.1*M | 78573568 | 10496548864 | 766246912 | 116 |
| 1.5*M | 21618688 | 10496548864 | 766246912 | 187 |
| 2.0*M | 22835200 | 10496548864 | 766246912 | 246 |
| 5.0*M | 23138304 | 10498646016 | 766246912 | 1072 |
| 10.0*M | 23121920 | 10499694592 | 766246912 | 824 |
| 50.0*M | 21336064 | 10499796992 | 766246912 | 663 |

### Explanation

## Programming Problem 5

### Description of algorithm

Our algorithm for generating an optimal Huffman code works as follows. We first take characters and their frequencies as input from the user. We create tree nodes for each character and its frequency, and store them in a priority queue sorted by frequency. We then take the two least frequent nodes in the queue and construct a new node connecting the two, whose frequency is derived from their two frequencies combined. This new node is added back into the queue. We continue this process until only one node remains. This resultant tree supplies our Huffman encoding.

### Java Implementation

```java
import java.util.Scanner;
import java.util.Comparator;
import java.util.ArrayList;
```

```java
import java.util.PriorityQueue;

class Node {
        char ch;
        int freq;
        Node left;
        Node right;
}

class HuffComparator implements Comparator<Node> {
        public int compare(Node a, Node b) {
                return a.freq - b.freq;
        }
}

public class Prog5 {

        public static void main(String[] args) {

                Scanner reader = new Scanner(System.in);
                ArrayList<Character> chars = new ArrayList<Character>();
                ArrayList<Integer> frequency = new ArrayList<Integer>();
                boolean check = true;

                while(check) {
                        System.out.println("Enter character (to exit enter '0')");
                        char elem = reader.nextLine().charAt(0);

                        if (elem == '0') {
                                check = false;
                        } else {
                                System.out.println("What is the frequency for the character?");
                                int freq = Integer.parseInt(reader.nextLine());
                                chars.add(elem);
                                frequency.add(freq);
                        }
                }
```

```java

        PriorityQueue<Node> queue = new PriorityQueue<Node>(chars.size(), new HuffComparator());

        for (int i = 0; i < chars.size(); i++) {
                Node newNode = new Node();

                newNode.ch = chars.get(i);
                newNode.freq = frequency.get(i);
                newNode.right = null;
                newNode.left = null;

                queue.add(newNode);
        }

        Node root = null;

        while (queue.size() > 1) {

                Node a = queue.peek();
                queue.poll();
                Node b = queue.peek();
                queue.poll();

                Node c = new Node();

                c.freq = a.freq + b.freq;
                c.ch = '-';
                c.left = a;
                c.right = b;
                root = c;

                queue.add(c);

        }

        printCode(root, "");

}
```

```java

    public static void printCode(Node root, String code) {
        if (root.left == null && root.right == null &&
        ↪  Character.isLetter(root.ch)) {
            System.out.println(root.ch + ": " + code);
            return;
        }

        printCode(root.left, code + "0");
        printCode(root.right, code + "1");
    }

}
```

### Sample Huffman Code

For character—frequency input:

a—8 v—1 z—1 b—5 c—6 p—4 e—10

Our algorithm generates the Huffman code:

c: 00 a: 01 e: 10 b: 110 v: 11100 z: 11101 p: 1111