COMP285 – Data Structures & Algorithm Analysis

Spring 2018

Programming project #1

Maximilian Escutia & Patrick Walker

Dr. Eric Jiang

2/24/2018

Assertion:

- We believe the project is complete. We have saved proj1 on our terminal under the specified directory.
- We managed to successfully finish this project before the deadline.
- We started our project by writing methods for each algorithm we were going to use, and overall it went pretty smoothly. When it came to reading a file from a user, and turning a text file into an array, it required us to go back into our old COMP151 notes and double check that we didn't forget how to do such tasks. Initially, we had a couple of errors with System.nanoTime() since it was the first time we were using it to calculate the running time of an algorithm. We managed to learn how to implement it correctly within our methods through trial and error. Working with a partner allowed me to have a conversation about the subject in more detail hence, it helped me better understand each algorithm.
- Turning in our project through Linux allowed us to polish and practice our Linux skills.
- Overall this was a good first project to start the semester!

Summary:

proj1.java is a program that implements four different algorithms for solving the “Maximum Subsequence Sum” problem that we saw in class. Our program allows the user to enter a file name which it then reads from, in order to store and create a new array with all the values from the text file. This newly created array is then passed through all four different algorithms and also calculates how long it takes to execute for each algorithm. Once this is done, The user can choose to run a specific algorithm or just end the program.

Maximum Contiguous Subsequence Sum:

- Input: Array X of n integers (possibly negative)
- Output: Find a subsequence with maximum sum, that is, find $0 \le i \le j < n$ to maximize $\sum_{k=i}^{j} X[k]$
- Assumption: if all are negative, then output is 0
- Different algorithms with different running times

Algorithm 1:

- This algorithm is the cubic maximum contiguous subsequence sum.
- Using three nested loops: the 1st loop sets the index i (the left end of a subsequence); the 2nd loop sets the index j (the right end of the subsequence); the 3rd loop sums the subsequence.
- For every pair (i, j) with $0 \le i \le j < n$, compute the sum $\sum_{k=i}^{j} X[k]$
- The precise analysis is obtained from the sum $\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^{j} 1$
- Number of iterations of innermost loop is $j - i + 1$

- Therefore $\sum_{j=i}^{n-1} j - i + 1 = (n-i)(n-i+1)/2$
- The total running time is $\sum_{i=0}^{n-1} (n - i)(n-i+1)/2 = (n^3 + 3n^2 + 2n)/6$
- The running time is $O(n^3)$.

Algorithm 2:

- This algorithm is the quadratic maximum contiguous subsequence sum algorithm.
- Observation: Sum of $X[i..(j+1)]$ can be computed by adding $X[j+1]$ to the sum of $X[i..j]$. Essentially, we can eliminate a loop by combining the second and third loops from MSS1(Algorithm 1). Set the index j and sum the subsequence from i to j.
- Hence, two nested loops results in a run time of $O(n^2)$.

Algorithm 3:

- This algorithm is the linear-time maximum contiguous subsequence sum algorithm.
- We can see that an optimal subsequence happens when we carry the largest sum, which should start at a positive number, if there is one.
- In MSS2 (Algorithm 2) , i and j point to the start and end of the current sequence, resp.; if a[i] < 0, then it can not possible represent the start of an optimal subsequence (this is where we eliminate the possible subsequences).
- If we use the same logic, any subsequence with a negative sum can not be a prefix of an optimal subsequence. So, we do not need to maintain the index i. We have saved a loop!
- Therefore the time complexity is T(n) = O(n).

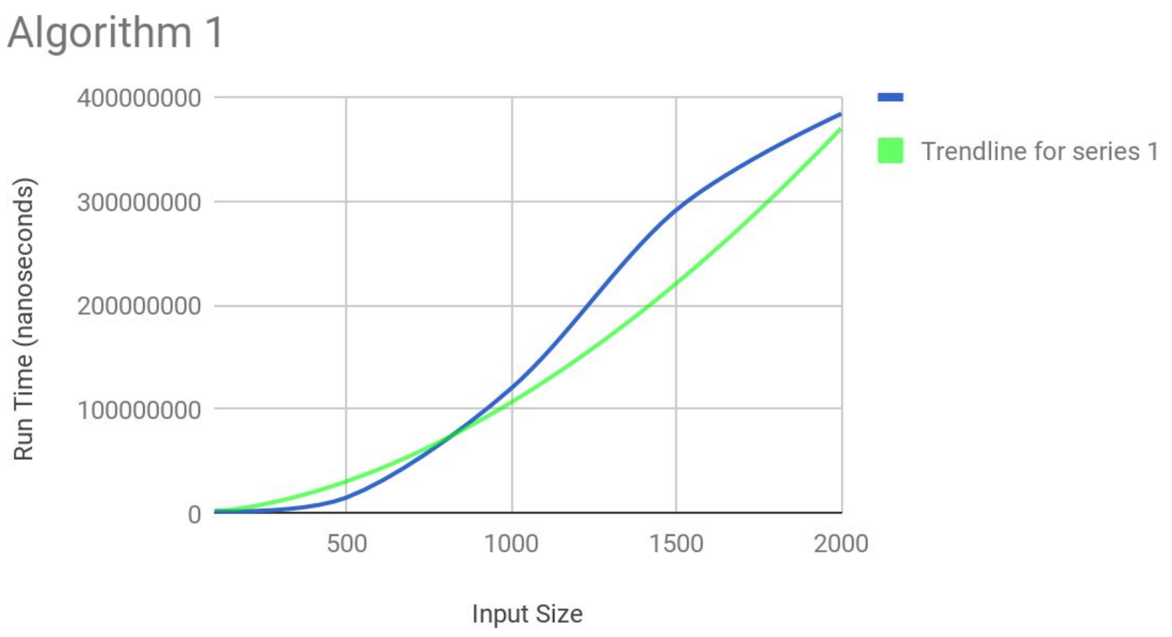
Algorithm 4:

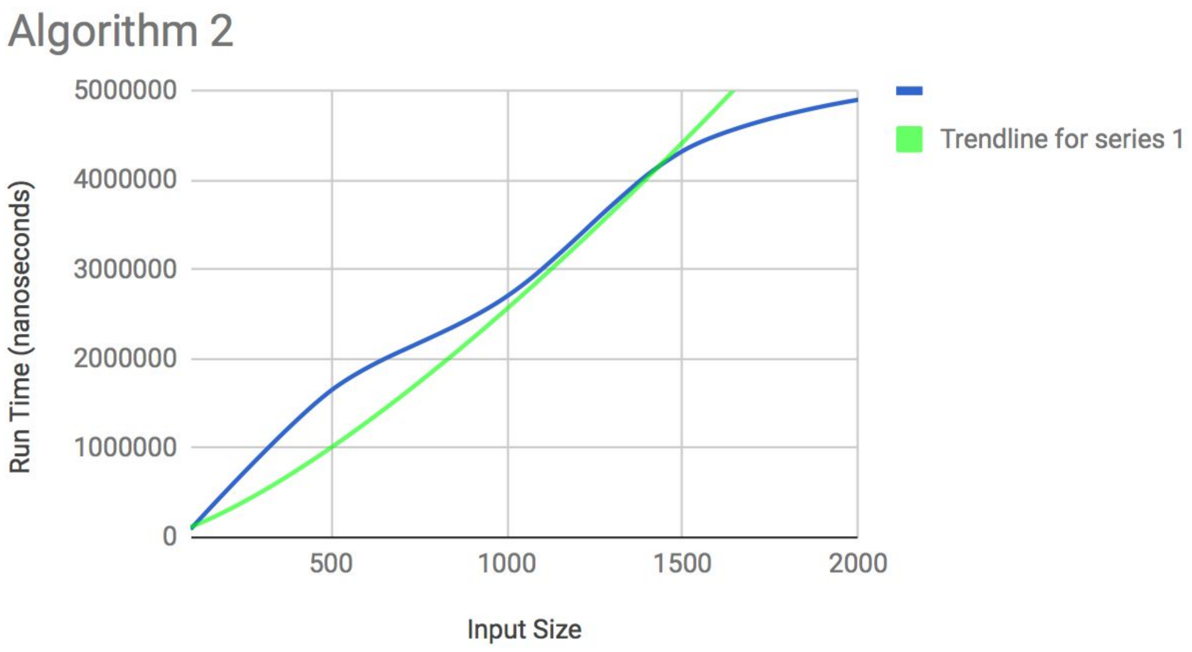
- This algorithm is known as the recursive maximum contiguous subsequence sum algorithm.
- First read the series of numbers into an array, then partition the array into 2 “equal” halves - left and right.
- Then compute the maxLeftSum and maxRightSum, recursively.
- Then return the max(maxLeftSum, maxRightSum), this does not however work for finding the subsequence that crosses the middle and in both halves.
- In the case that the maximum subsequence sum is in the middle: find the largest sum in the 1st half that includes its last element and the largest sum in the 2nd half that includes its first element.
- Then put them all together: Partition the array into 2 “equal”l halves - left and right.
- Then compute the maxLeftSum and maxRightSum, recursively, then compute the maxLeftBoundSum and maxRightBoundSum, and finally return the max(mexLeftSum, maxRightSum, maxLeftBoundSum + maxRightBoundSum)
- For example: 6, -5, 2, 3, -4, 7, -8, 1, 9, -10 → maxLeftSum = 6, maxRightSum = 10, maxLeftBoundSum = 2, maxRightBoundSum = 9
- Essentially, we are dividing the problem in two parts: find maximum subsequences of left and right halves, and take the maximum of the two.

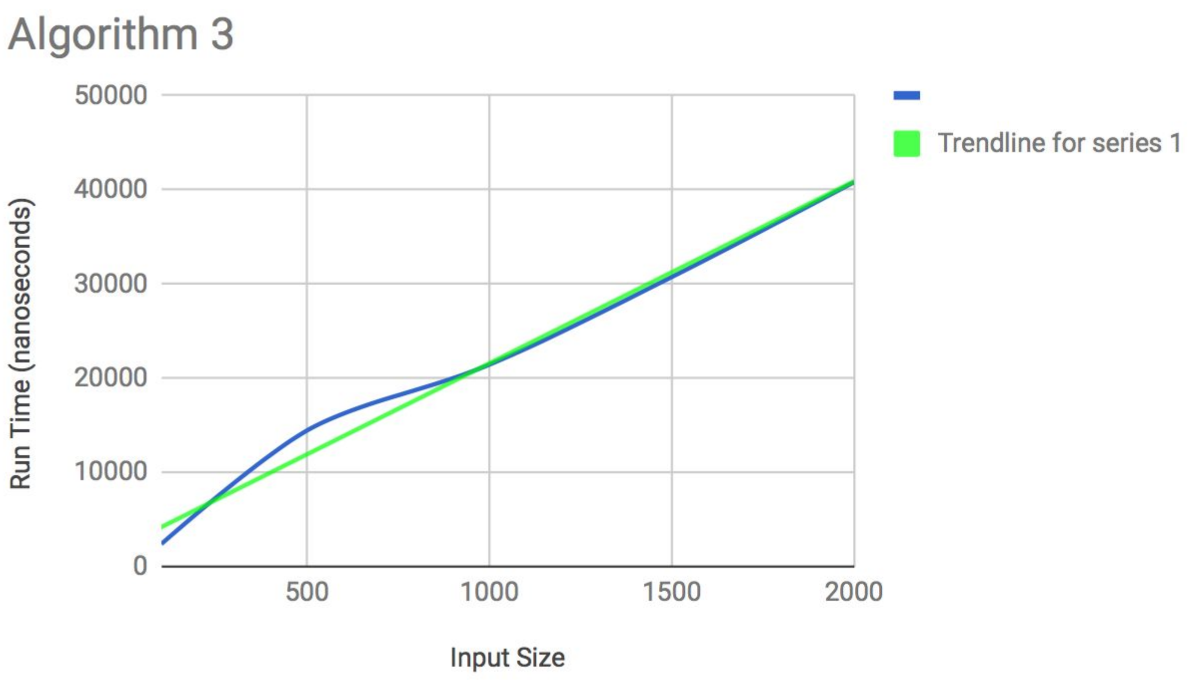
- Let T(n) be the running time of the recursive maximum subsequence sum from Left - Right +1 = N
- Base Case: T(1) = O(1) because if N = 1, then the algorithm takes some constant amount of time
- Recursive case: Two recursive calls of size n/2, plus O(N) for the additional steps
- This gives T(N) = 2T(N/2) + O(N)
- The pattern of the algorithm can be derived, is that if N = 2^k, then T(N) = N * (k + 1)
- Therefore the running time T(N) = N log (N) + N = O(N log N)

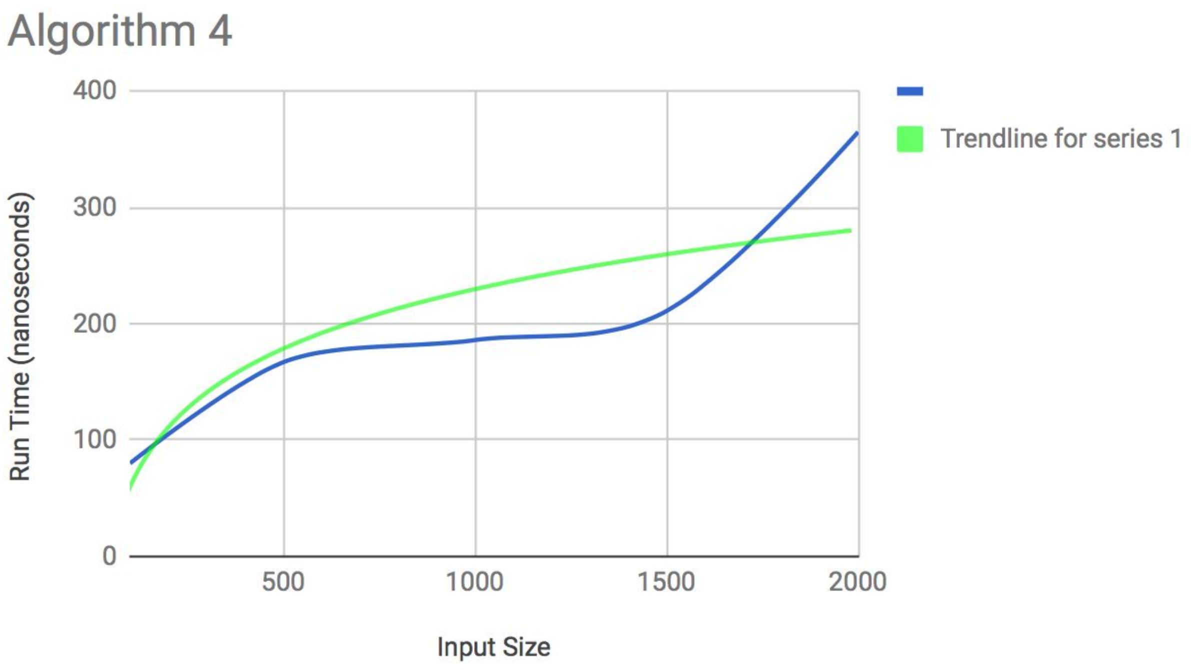
Data:

	Algorithm Running Time (nano seconds)			
Input Size	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
100	2262805	90339	2365	80
500	15271065	1644886	14433	167
1000	120411274	2699236	21422	186
1500	291763834	4326529	30729	211
2000	384134801	4900252	40804	365


Algorithm 1
400000000
300000000
200000000
100000000
0
500
1000
1500
2000
Run Time (nanoseconds)
Input Size
Trendline for series 1


Algorithm 2
5000000
4000000
3000000
2000000
1000000
0
500
1000
1500
2000
Run Time (nanoseconds)
Input Size
Trendline for series 1


Algorithm 3
Trendline for series 1
Run Time (nanoseconds)
Input Size
0
10000
20000
30000
40000
50000
500
1000
1500
2000


Algorithm 4
Trendline for series 1
Run Time (nanoseconds)
Input Size
0
100
200
300
400
500
1000
1500
2000

Conclusion:
Through thorough testing and algorithm analysis we can confidently say that Algorithm 4 was the fastest. In order of fastest to slowest, Algorithm 4 > Algorithm 3 > Algorithm 2 > Algorithm 1. Algorithm 4 was the fastest because it implemented recursive calls. As seen in the graph above, our conclusions are consistent with our data.

Sample runs:
proj1.java

```
:Script started on Sun 25 Feb 2018 09:24:39 PM PST
^[]0;mescutia@cslab5:~/COMP285/proj1^G^[[?1034h[mescutia@cslab5 proj1]$ java p^Groj1
Enter the file name you wish to read from:
testText.txt
The duration of the first algorithm is: 1419292874 nano seconds
The duration of the second algorithm is: 6349214 nano seconds
The duration of the third algorithm is: 48808 nano seconds
The duration of the fourth algorithm is: 75 nano seconds

Algorithms:
1) Algorithm 1
2) Algorithm 2
3) Algorithm 3
4) Algorithm 4
Choose a number to run:
1
The duration of the first algorithm is: 4768639442 nano seconds

Do you want to run the program again? (Type y or n)
y

Algorithms:
1) Algorithm 1
2) Algorithm 2
3) Algorithm 3
4) Algorithm 4
Choose a number to run:
2
The duration of the second algorithm is: 4099128 nano seconds

Do you want to run the program again? (Type y or n)
y
```

```
Algorithms:
1) Algorithm 1
2) Algorithm 2
3) Algorithm 3
4) Algorithm 4
Choose a number to run:
3
The duration of the third algorithm is: 58963 nano seconds

Do you want to run the program again? (Type y or n)
y

Algorithms:
1) Algorithm 1
2) Algorithm 2
3) Algorithm 3
4) Algorithm 4
Choose a number to run:
4
The duration of the fourth algorithm is: 96 nano seconds

Do you want to run the program again? (Type y or n)
y

Algorithms:
1) Algorithm 1
2) Algorithm 2
3) Algorithm 3
4) Algorithm 4
Choose a number to run:
4
The duration of the fourth algorithm is: 92 nano seconds

Do you want to run the program again? (Type y or n)
y

Algorithms:
1) Algorithm 1
2) Algorithm 2
3) Algorithm 3
4) Algorithm 4
Choose a number to run:
4
```

```
The duration of the fourth algorithm is: 200 nano seconds

Do you want to run the program again? (Type y or n)
n
^[]0;mescutia@cslab5:~/COMP285/proj1^G[mescutia@cslab5 proj1]$ exit
Exit
Script done on Sun 25 Feb 2018 09:25:48 PM PST
```

Source code:
proj1.java

```java
/**
 * Programming Project #1
 * Name: proj1.java
 * Last Modified: 02/24/2018
 * Author: Max Escutia & Patrick Walker
 * Description: Implementing 4 different algorithms for solving the "Maximum Subsequence Sum".
 *           This program reads input from a file while creating an array, and uses it to pass                    it through
 *           each algorithm while calculating the time it takes to execute.
 *
 * */

import java.io.File;
import java.util.*;
import java.util.Scanner;
import java.lang.*;


public class proj1 {
        public static void main(String[] args) {
        /*
         *  Read in file from user
         */
        System.out.println("Enter the file name you wish to read from: ");

        /*
         *  Read each line from the file and put values into an array
         */
        Scanner userInput = new Scanner(System.in);
        String fileName = userInput.nextLine();
        int[] a = readTextFile(fileName); //This creates an array from the text file


        /*
         *  Use created array and run it through each algorithm while
         *  calculating the time it takes to execute each algorithm
         */
        maximumSubSumOne(a);

        maximumSubSumTwo(a);

        maximumSubSumThree(a);
maximumSubSumFour(a);


/*
 *  Prompt the user with options about which algorithm he/she
 *  wants to run
 */
char ch;
do {
		System.out.println("\nAlgorithms: ");
		System.out.println("1) Algorithm 1");
		System.out.println("2) Algorithm 2");
		System.out.println("3) Algorithm 3");
		System.out.println("4) Algorithm 4");
		System.out.println("Choose a number to run: ");
		Scanner input = new Scanner (System.in);

		int choice = input.nextInt();
		switch (choice) {
		case 1 :
		maximumSubSumOne(a);
		break;
		case 2 :
		maximumSubSumTwo(a);
		break;
		case 3 :
		maximumSubSumThree(a);
		break;
		case 4 :
		maximumSubSumFour(a);
		break;
		default :
		System.out.println(choice + " is not an option");
		break;
		}
		//Loop to try run the whole program again
		System.out.println("\nDo you want to run the program again? (Type y or n)");
		ch = input.next().charAt(0);
} while (ch == 'Y'|| ch == 'y');
}


// This method will read in the integers from the text file and put them into an array
public static int[] readTextFile(String fileName) {
try {
		File file = new File (fileName);
		Scanner inputFile = new Scanner (file);
		int count = 0;
		while(inputFile.hasNextInt()) {
		count ++;
		inputFile.nextInt();
		}
		int[] arr = new int[count];
		Scanner s1 = new Scanner(file);
		for(int i = 0; i < arr.length; i ++)
		arr[i] = s1.nextInt();
```

```
            return arr;
} catch (Exception e) {
            return null;
}
}


//running time = O(n^3) --> Cubic
public static int maximumSubSumOne(int[] a) {
long startTime = System.nanoTime();
int maxSum = 0;

for (int i = 0; i < a.length; i++)
            for (int j = i; j < a.length; j++) {
            int sum = 0;

            for (int k = i; k <= j; k++)
            sum += a[k];

            if (sum > maxSum)
            maxSum = sum;
            }
long duration = System.nanoTime() - startTime;
System.out.println("The duration of the first algorithm is: " + duration + " nano seconds");
return maxSum;
}


//Running time = O(n^2) --> Quadratic
public static int maximumSubSumTwo(int[] a) {
long startTime2 = System.nanoTime();
int maxSum = 0;

for (int i = 0; i < a.length; i ++) {
            int sum = 0;

            for (int j = i; j < a.length; j++) {
            sum += a[j];

            if (sum > maxSum)
            maxSum = sum;
            }
}
long stopTime2 = System.nanoTime();
long duration2 = stopTime2 - startTime2;
System.out.println("The duration of the second algorithm is: " + duration2 + " nano seconds");
return maxSum;
}


//Running time = O(n) --> linear
public static int maximumSubSumThree(int[] a) {
long startTime3 = System.nanoTime();
int maxSum = 0;
int sum = 0;

for (int j = 0; j < a.length; j ++){
```

```
                sum += a[j];

                if (sum > maxSum)
                maxSum = sum;
                else if (sum < 0)
                sum = 0;
        }
        long stopTime3 = System.nanoTime();
        long duration3 = stopTime3 - startTime3;
        System.out.println("The duration of the third algorithm is: " + duration3 + " nano seconds");
        return maxSum;
        }


        //Running time = O(nlog(n)) --> Recursive maximum subsequence sum algorithm
        public static int maximumSubSumFour (int[] a) {
        long startTime4 = System.nanoTime();
        long stopTime4 = System.nanoTime();
        long duration4 = stopTime4 - startTime4;
        System.out.println("The duration of the fourth algorithm is: " + duration4 + " nano seconds");
        return maxSumRecursive(a, 0, a.length -1);
        }
        public static int maxSumRecursive(int[] a, int left, int right) {
        if (left == right) //Base case
                if (a[left] > 0)
                return a[left];
                else
                return 0;

        int middle = (left + right) / 2;
        int maxLeftSum = maxSumRecursive(a, left, middle);
        int maxRightSum = maxSumRecursive(a, middle + 1, right);

        int maxLeftBorderSum = 0;
        int leftBorderSum = 0;
        for (int i = middle; i >= left; i--) {
                leftBorderSum += a[i];
                if (leftBorderSum > maxLeftBorderSum)
                maxLeftBorderSum = leftBorderSum;
        }

        int maxRightBorderSum = 0;
        int rightBorderSum = 0;
        for (int i = middle + 1; i <= right; i++) {
                rightBorderSum += a[i];
                if (rightBorderSum > maxRightBorderSum)
                maxRightBorderSum  = rightBorderSum;
        }
        int maximum = Math.max(maxRightBorderSum + maxLeftBorderSum, Math.max(maxLeftSum, maxRightSum));
        return maximum;
        }
}
```