1 Problem Description

We have introduced and studied the so called Maximum Subsequence Sum Problem. The problem is used as an example to illustrate the study of algorithms. In general, there are many ways or algorithms to solve a particular problem and some are more efficient than others. Once we have understand the idea of an algorithm, we need to express the idea in a program using a programming language.

In this lab, we will learn two algorithms to solve this problem, express the two algorithms as two functions in C++, and empirically evaluate the performance of the two algorithms by clocking how long it takes for each algorithm to run on the same data. Our data will have different sizes, that is different values of n (see definition below), which is referred to as input size.

Here is a more precise definition of the problem. Given a sequence of integers (which could be positive, negative, or 0), \(a_1 a_2 \ldots a_n\), determine the maximum subsequence sum of \(a_1 a_2 \ldots a_n\). Here a subsequence is any sequence \(a_i a_{i+1} \ldots a_j\), where \(1 \leq i \leq j \leq n\). The subsequence sum of \(a_i a_{i+1} \ldots a_j\) is \(a_i + a_{i+1} + \cdots + a_j\). The maximum subsequence sum is the largest subsequence sum among all possible subsequences. When all values in the sequence are negative, which is a possibility based on the definition of the problem, the maximum subsequence sum is considered 0 and the empty sequence, that is the sequence that does not contain any element, produces 0.

Here is a concrete instance, served as our example, to illustrate the definition. Let us consider the sequence 31, -41, 59, 26, -53, 58, 97, -93, -23, 87. In this example, \(n = 10\), and \(a_1 = 31\), \(a_2 = -41\), and so on. For this sequence, the maximum subsequence sum is 187 (convince yourself it is the case) and the sequence that produces this value is \(a_3 \ldots a_7\) (or 59, 26, -53, 58, 97).

In our lecture we have illustrated and shown that the number of all subsequences is \(2^n\) (see you can derived it again). So our first algorithm enumerates all possible nonempty subsequences. For each subsequence enumerated, it computes its sum and compares the sum with the current best value (What do you think we should initialize the current best value?). If the sum is larger than the current best value, the current best value is updated to the sum. The correctness of the algorithm follows from the fact we have considered all the possible subsequences and kept the largest value.

Here is how we enumerate the subsequences: we first enumerate all subsequences that begin with \(a_1\) and within those subsequences we enumerate the subsequences with increasing length. We will repeat the above with \(a_2\), \(a_3\), and so on. The following is the enumeration based on the above idea for \(a_1 a_2 a_3 a_4\):

- \(a_1\)
- \(a_1 a_2\)
- \(a_1 a_2 a_3\)
- \(a_1 a_2 a_3 a_4\)
- \(a_2\)
- \(a_2 a_3\)
- \(a_2 a_3 a_4\)
- \(a_3\)
- \(a_3 a_4\)

In expressing the algorithm in C++, we can imagine that the sequence is
stored in an array of integers, seqArray. Once the array and the two indices \( i \) and \( j \) are known, the subsequence that starts with \( i \) and ends with \( j \) is known. Since our array may have different sizes, we need to have a variable to keep track of the size. Keeping the above in mind may help us to understand the prototypes in the design section.

Read the following after you have finished and tested the first algorithm above. Now you have successfully solved a challenging problem and congratulations! Let us see if we might improve the algorithm above to be more efficient. In fact, one of the reasons for studying data structures, which we will do in this course, is for efficiency.

Observe that in the enumeration scheme of the above algorithm, when we start to compute the sum of \( a_i \ldots a_j \), we have just finished the sum for \( a_i \ldots a_{j-1} \) (verify that with \( a_1 a_2 a_3 a_4 \) enumeration above). If we can save the prefix sum, that is the sum of \( a_i \ldots a_{j-1} \), we may simply add \( a_j \) to it to get the sum for \( a_i \ldots a_j \). Now instead of using a loop, we just use one operation. This is our second algorithm, which is based on the idea of the first algorithm but with better performance.

In fact there are even better algorithms to solve this problem and we will come back to it. Our first algorithm has time complexity of \( n^3 \), the second one has time complexity of \( n^2 \). The even better ones have time complexity of \( n \log n \) and \( n \) (Are you familiar with these functions and the growth rate of them?).

2 Purpose

Review C++ functions, function overloading, array, nested loops, and basic programming (expressing algorithmic ideas in a programming language). Learn the brute force idea of algorithm development (algorithm 1). Learn the importance of refinement and observation in improving an algorithm (the result of which is algorithm 2). See and experience how efficiency (or complexity) of an algorithm relates to run time. Understand conceptually that a problem is a set of instances, each instance has a size among other properties (later labs and lectures will deal with them), and the run time of our program or algorithm is a mathematical function of the size of an input instance and other properties of the instance (later labs and lectures will deal with them).

3 Design

The following lists the prototype of the functions:

- int Algo1MSS(int * seqArray, int size);
- int Algo2MSS(int * seqArray, int size);
- int subseqSum(int * seqArray, int i, int j);
- int subseqSum(int prefixToaj, int aj);

The main program as well as additional support functions (for initializing the SeqArray) will be provided for you to use in the Lab. The usage of those
functions will be discussed in the lab as well.

4 Implementation

1. Implement int subseqSum(int * seqArray, int i, int j) function using a single loop to add from seqArray[i] to seqArray[j] and unit test the function. Note that the function returns the sum obtained.

2. Next implement int Algo1MSS(int * seqArray, int size) function using two nested for loops for enumerating all possible subsequences and for each subsequence call int subseqSum(int * seqArray, int i, int j) to evaluate the subsequence sum.

3. Finish everything about our first algorithm before working on the second algorithm.

4. Implement int subseqSum(int prefixToaj, int aj), which is really easy and no loop!

5. Next implement int Algo2MSS(int * seqArray, int size) function. The code should be almost identical to that of Algo1MSS(int * seqArray, int size). The only difference is that now we will call int subseqSum(int prefixToaj, int aj) to evaluate the subsequence sum.

5 Test and evaluation

1. Unit testing each function using the instance given in the problem description.

2. Clocking the run time of the first algorithm, that is Algo1MSS function, with $n = 1000, 10000, 100000, 1000000$. If it takes too long to get the results in the lab, then try to complete the evaluation outside the lab. Also if it takes too long for the first algorithm to handle size 100000 and 1000000, you may estimate the run time.

3. Clocking the run time of the second algorithm, that is Algo2MSS, as in the step above.

4. Observe the relationship between input size $n$ and the measured run time (should be $n^3$ for the first algorithm and $n^2$ for the second one).

6 Report and documentation

A short report about things observed and things learned and understood. Properly document and indent the source code.
7 Lab submission

Get instructions from the Lab instructor.