Computer Science 331 — Winter 2016

Solution for Assignment #1

Instructions

This assignment concerns lecture material that has been introduced in this course on or before Wednesday, January 29.

Please **read the entire assignment** before you begin work on any part of it.

This assignment is due by 11:59 pm on Monday, February 8.

Total marks available: 100 (plus 10 bonus marks)

Additional Requirements

- **Start this assignment now!** While the programming required for this assignment is not particularly difficult, the process of performing the analysis that the above questions require will all take thought and time.
- Please make sure to read the information about "How to Succeed in Computer Science 331" as well as "Things to Avoid in Computer Science 331" on the course web site. Note, in particular, the information about academic misconduct that is included there.
- Please read the information about how you should prepare written answers for questions and about how you should submit your material that is provided on the course web page.

Questions

Consider the following computational problem:

Problem: Maximum subsequence sum

Precondition: A is an array (not `null`) of possibly negative integers

Postcondition: S is the maximal value of the sum $\sum_{k=i}^{j} A[k]$ for all possible values of i and j, i.e.,

$$S = \max_{0 \le i,j < n} \sum_{k=i}^{j} A[k] \ .$$

If all entries in A are negative, the maximum subsequence sum is defined to be zero.

An an example, consider the following array:

	0	1	2	3	4	5	6	7	8	9	10	11
A	25	-5	-12	-9	14	12	-13	5	8	-2	18	-8

The values of i and j that maximize the subsequence sum are $i = 4$ and $j = 10$, for which

$$\sum_{k=4}^{10} A_k = 14 + 12 - 13 + 5 + 8 - 2 + 18 = 42 \ .$$

One solution to this problem is to compute the subsequence sum for every possible pair of indices (i, j) and keep track of the maximum value obtained as follows:

```
maxS = 0
n = A.length
for j from 0 to n − 1 do
  for i from 0 to j do
    S = 0
    for k from i to j do
      S = S + A[k]
    end for
    if S > maxS then
      maxS = S
    end if
  end for
end for
```

The file `MaxSubsequenceSum.java` contains an implementation of this algorithm, along with two other methods (one recursive) for solving this problem. You can obtain this file from the course web page. The code in this file is thoroughly documented. It uses exceptions for testing adherence to preconditions for public functions and several assertions in key parts of the code as described in the document "Programming With Assertions" available through the java page on the course web site. You are expected to adhere to this standard of documentation for subsequent assignments in the course.

You should be able to compile and run this program as-is. To run the program with assertion testing enabled, use

`java -enableassertions MaxSubSequenceSum` *command-line args*

This program takes three command-line arguments. Run the program without any to see a description of these required arguments.

1. **(5 marks)** State (no proof required) a loop invariant and loop variant for the outer-most for loop of the algorithm described above.

   Solution:

   Loop invariant:

   - $0 \leq j < n$;

- $maxS = \max_{0 \le a,b < j} \sum_{k=a}^{b} A[k]$ (the max. subsequence sum for the first j entries of A)
- A has not been modified

Loop variant: $f(n, j) = n - j$

2. **(20 marks)** Prove that the inner-most loop of the algorithm described above is correct. (Hint: writing pre- and post-conditions specifically for the inner-most loop, as well as re-writing the loop as a while-loop, will be helpful).

Solution: The purpose of the inner loop is to compute the sum $A[i] + \cdots + A[j]$. Expressing the loop as a while-loop is convenient, as this makes it clear where the initialization and updating of k occur:

$S = 0$
$k = i$
while $k <= j$ **do**
  $S+ = A[k]$
  $k = k + 1$
end while

This also makes the loop guard G clear: $k \le j$.

Now, the precondition p and postcondition q for this loop can be expressed as follows:

Precondition (p):

- A is an integer array of length n
- $0 \le i < n$;
- $i \le j < n$;
- $k = i$;
- $S = 0$

Postcondition (q):

- A is an integer array of length n;
- A, i, and j are unchanged;
- $S = A[i] + A[i + 1] + \cdots + A[j]$

Note that the precondition as stated is assumed to hold immediately before the while-loop. It contains statements about the effects of the two initializations before the loop.

We also require a loop invariant r :

- A is an integer array of length n;
- $0 \le j < n$;
- $0 \le i \le j$;
- $i \le k \le j$;
- $S = A[i] + \cdots + A[k - 1]$

Notice that A, i, and j are not modified during the loop, so we only need to handle the conditions on k and S throughout the proof.

We first prove that r is a valid loop invariant by induction:

(a) (Base case): True before first iteration: $k = i$ and the sum $A[i] + \ldots A[k-1]$ contains no terms (thus equal to zero).

(b) If True before an iteration and $k < j$ then True after the iteration:

- Before the iteration: $i \leq k_{before} < n$, and $S_{before} = \sum_{l=i}^{k_{before}} A[l]$
- After the iteration: $k_{after} = k_{before} + 1 \Rightarrow 1 \leq k_{after} \leq j$ and

$$S_{after} = S_{before} + A[k_{after}] = \sum_{l=i}^{k_{before}} A[l] + A[k_{after}] = \sum_{l=i}^{k_{after}} A[l] \ .$$

Proof of partial correctness:

- precondition p and initial assignment statements imply the loop invariant.
- upon termination: $k = j$ and $S = \sum_{l=i}^{j} A[l] \Rightarrow Q$ □

To prove termination, note that the function $f(k, j) = (j + 1) - k$ is a loop variant for this loop. To see this, note that:

- the function is integer-valued,
- the function decreases by one after every iteration (k increases, so $(j+1)-k$ decreases),
- if $f(k, j) \leq 0$, implying that $k \geq j + 1$, then the loop terminates.

Thus, $f(k, j)$ satisfies the propoerties of a loop variant, and its existence implies that the loop must terminate.

The inner most loop is partially-correct and terminates, and is therefore correct.

3. **(15 marks)** Give an *upper bound* for a function $T_1(n)$ that describes the *worst-case* number of steps executed by the algorithm described above when the input array has n elements. Consider one step to be one assignment, comparison, or arithmetic operation (you may ignore array subscripting). Justify your answer.

Solution: To find the worst-case run-time of this algorithm, we first determine the worst-case run-time of the inner-most loop and work outwards. The loop variant for this loop is $f(j, k) = j + 1 - k$. The initial value $k = i$ tells us that the number of iterations of this loop is equal to $j + 1 - i$. The number of operations done per execution is at most 3 (adding $A[k]$ to S, assigning new value to S, and incrementing k). The termination condition $k > j$ is checked $j + 1 - i + 1 = j + 2 - i$ times, and 1 additional step is required to initialize k before the loop begins. Thus, $T_k(i, j)$, the worst-case number of steps executed by the inner loop, satisfies

$$T_k(i, j) = 3(j + 1 - i) + (j + 2 - i) + 1 = 4j - 4i + 6 \ .$$

Note that the worst-case for this loop occurs when $i = 0$ and $j = n - 1$, implying that $T_k(i, j) \leq 4n + 2$.

Next, we analyze the middle loop (index j). A loop variant for this loop is $f(i,j) = j+1-i$, implying that this loop iterates at least $j+1$ times. The number of steps in the loop body is $T_k(i,j)+4$ (executing the inner loop costs $T_k(i,j)$ steps, initializing S, comparison of S and $maxS$, updating $maxS$, and incrementing i). The termination condition $i > j$ is tested $j+2$ times, and 1 additional step is required to initialize i. Thus, $T_i(j)$, the worst-case number of steps executed by the middle loop, satisfies

$$\begin{aligned} T_i(j) &\leq (j+1)(4n+2+4) + (j+2) + 1 (\text{because } T_k(i,j) \leq 4n+2) \\ &\leq (j+1)(4n+6) + (j+3) \ . \end{aligned}$$

Note that the worst-case for this loop occurs when $j = n-1$, implying that $T_i(j) \leq n(4n+6) + n - 2 = 4n^2 + 7n - 2$.

Finally, we analyze the outer loop. The loop variant $f(n,j) = n - j$ implies that this loop iterates at most n times. The number of steps in the loop body is $T_i(j)$ plus 1 step to increment j. The termination test $j > n-1$ costs two steps and is executed $n+1$ times, and 3 steps are required to initialize $maxS$, n, and j. Thus, $T_1(n)$, the worst-case number of steps executed by the entire algorithm, is

$$\begin{aligned} T_1(n) &\leq n(4n^2+7n-2) + 2(n+1) + 3 \qquad (\text{because } T_i(j) \leq 4n^2+7n-2) \\ &= 4n^3 + 7n^2 + 5 \ . \end{aligned}$$

Note: It is possible to write a precise function for $T_1(n)$ as opposed to an upper bound using summations. 5 bonus marks were awarded for a successful application of this approach.

4. **(5 marks)** Prove that your function $T_1(n) \in O(n^d)$ for some positive integer d. Your value of d should satisfy $2 \leq d \leq 3$.

   Solution: We will show that there exists a positive real constant $c > 1$ and a real constant $N_0 \geq 0$ such that

   $$T_1(n) = 4n^3 + 7n^2 + 5 \leq cn^3 \quad \text{for all } n \geq N_0.$$

   Then, by definition we will have shown that $T_1(n) \in O(n^3)$.

   We will show that $c = 16$ and $N_0 = 1$ satisfy the required inequality. To see this, note that if $n \geq 1$ we have $n^3 \geq n^2 \geq 1$. Thus, if $n \geq 1$ we have

   $$4n^3 + 7n^2 + 5 \leq 4n^3 + 7n^3 + 5n^3 = 16n^3$$

   as required.

5. **(10 marks)** The following pseudocode describes the recursive algorithm used by `maxSubSum2` to solve the maximum subsequence sum problem:

   ```
   public int maxSumRec(int [] A)
   n = A.length
   if n = 0 then
     return 0
   else if n = 1 then
     if A[0] > 0 then
       return A[0]
   ```

```
    else
      return 0
    end if
  else
    S_L = maxSumRec(A[0], ..., A[n/2])
    S_R = maxSumRec(A[n/2 + 1], ..., A[n − 1])
    S_1 = max(A[n/2], A[n/2] + A[n/2 − 1], ..., A[n/2] + A[n/2 − 1] + ··· + A[0])
    S_2 = max(A[n/2+1], A[n/2+1]+A[n/2+2], ..., A[n/2+1]+A[n/2+2]+···+A[n−1])

    return max(S_L, S_R, S_1 + S_2)
  end if
```

Give a recurrence relation $T_2(n)$ that describes the *worst-case* number of steps executed by this algorithm when the input array has n elements. Consider one step to be one assignment, comparison, or arithmetic operation (you may ignore array subscripting). You may express functions involved in the statement of the recurrence relation using asymptotic notation, for example, use $\Theta(n)$ in place of any linear functions of n that arise.

Solution:

$$T_2(n) \leq \begin{cases} 3 & \text{if } n = 0 \\ 5 & \text{if } n = 1 \\ T_2(\lfloor n/2 \rfloor) + T_2(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Explanation: A recurrence relation is a function defined in terms of values of the same function applied to smaller inputs, eg. $T(n) = T(n-1) + T(n-2)$. In order to use a recurrence relation to describe the number of steps of a recursive function, we define it piecewise as follows:

$$T(n) = \begin{cases} exp_1 & \text{describes number of steps for base case(s)} \\ exp_2 & \text{describes number of steps for the recursive case(s)} \end{cases}$$

Note that the expressions describing the running time for the recursive cases will be defined in terms of the same function $T(n)$ applied to smaller values of n. For example, if the function makes a recursive call on an input of size $n/2$, then the expression describing the number of steps required for that recursive case will have a term $T(n/2)$ in it, because the number of steps required by the algorithm for inputs of size $n/2$ is precisely $T(n/2)$.

The algorithm `maxSumRec` has two base cases. The first occurs when $n = A.length$ is equal to zero. In this case, 3 operations are required (initializing n, testing whether $n = 0$, and returning 0), and we have that

$$T_2(n) = 3 \quad \text{if } n = 0.$$

The second base case, when $n = 1$, requires 5 operations (initializing n, testing whether $n = 0$, testing whether $n = 1$, testing whether $A[0] > 0$, and invoking the appropriate return statement), so we have

$$T_2(n) = 5 \quad \text{if } n = 1.$$

The recursive case is analyzed as follows. Notice that there are two recursive calls in the algorithm.

- The first is called on the left half of the input array, namely the elements

$$A[0], \ldots, A[n/2],$$

  The size of the input in this case is $\lfloor \frac{n}{2} \rfloor$, so, the cost of evaluating the first recursive call is less than or equal to $T_2(\lfloor n/2 \rfloor)$. Notice that the floor is required here in the event that n is odd in order to have an integer value.

- The second recursive call is called on the elements

$$A[n/2+1], \ldots, A[n-1],$$

  and the size of the input is $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, so the cost of evaluating the second recursive call is less than or equal to $T_2(\lceil n/2 \rceil)$.

The remainder of the work in the general case is to compute S_1 and S_2. Each of these involves adding roughly $n/2$ terms and computing the maximum of $n/2$ values. The number of steps to compute these will be some linear function in n of the form $an + b$ (a, b constants). All the remaining steps can be done in constant time, so in addition to the two recursive calls, the rest of the recursive case can be done in $\Theta(n)$ steps. Thus, the total cost for the recursive case ($n > 1$) is

$$T_2(n) \leq T_2(\lfloor n/2 \rfloor) + T_2(\lceil n/2 \rceil) + \Theta(n) \ .$$

6. **(10 marks)** Using mathematical induction, it is possible to prove that the correct recurrence $T_2(n) \in \Theta(n \log n)$. It is also easy to see that $T_3(n) \in \Theta(n)$, where $T_3(n)$ describes the worst-case running time of function `maxSubSum3`. Using asymptotic notation as described in the lectures, relate $T_1(n)$ to $T_2(n)$, $T_1(n)$ to $T_3(n)$ and $T_2(n)$ to $T_3(n)$. You do not need to justify these answers — just state the *most precise* relationship that is appropriate.

   Solution: Both

$$T_1(n) \in \omega(T_2(n)), \quad T_2(n) \in \omega(T_3(n))$$

   and

$$T_3(n) \in o(T_2(n)), \quad T_2(n) \in o(T_1(n))$$

   are acceptable. Both of these indicate that, asymptotically, algorithm `maxSubSum1` is the slowest and `maxSubSum3` is the fastest.

   Explanation: Your task for this question is to compare the three functions $T_1(n)$, $T_2(n)$, and $T_3(n)$ using the asymptotic notations described in class. We assume that $T_1(n) \in \Theta(n^3)$ (answer to a previous question). By "relating" these functions using asymptotic notation, we mean providing statements such as $T_1(n) \in \omega(T_2(n))$, allowing us to compare the asymptotic growth rates of the runtimes for the three corresponding algorithms.

   - For the first case, we can write $T_1(n) \in \omega(T_2(n))$ because, intuitively, a function who's runtime has growth-rate n^3 grows strictly faster than one with growth-rate $n \lg n$. You can prove this formally using the definitions by arguing that
     - $T_1(n) \geq c_1 n^3$ for all $n \geq N_1$ for constants $c_1 > 0$ and $N_1 \geq 0$ (because $T_1(n) \in \Theta(n^3)$)
     - $c_2 n \lg n \geq T_2(n)$ for all $n \geq N_2$ for constants $c_2 > 0$ and $N_2 \geq 0$ (because $T_2(n) \in \Theta(n \lg n)$)

- $c_1 n^3 \in \omega(c_2 n \lg n)$ (using the limit test), implying that for any constant $c_3 > 0$ there exists a constant N_3 such that $c_1 n^3 \geq c_3 c_2 n \lg n$ for all $n \geq N_3$.
- putting these results together yields

$$\begin{aligned} T_1(n) &\geq c_1 n^3 \quad \text{for all } n \geq N_1 \\ &\geq c_3 c_2 n \lg n \quad \text{for any constant } c_3 > 0 \text{ and all } n \geq \max(N_1, N_3) \\ &\geq c T_2(n) \quad \text{for any constant } c_3 \text{ and all } n \geq \max(N_1, N_3, N_2) \end{aligned}$$

so $T_1(n) \in \omega(T_2(n))$ by definition.

- We can write $T_1(n) \in \omega(T_3(n))$ and $T_2(n) \in \omega(T_3(n))$ for similar reasons.

7. **(10 marks)** Describe a set of black-box tests for the maximum subsequence sum problem based on the problem description given at the beginning of the assignment description.

Solution: As described in class, you need to specify the following for each individual test case:

- input
- expected output
- rationale/purpose of the test

Here are some sample black-box tests covering boundary conditions and typical inputs:

Input (Array A)	Output	Purpose
`null`	exception	boundary (invalid input)
[]	0	boundary (min. number of elements)
[4]	4	boundary (array with 1 element)
$[-5]$	0	boundary (array with 1 negative element)
$[-11, -5, 2, -8]$	2	boundary (only 1 positive entry)
$[3, 10, 3]$	16	boundary (all pos. entries, max is sum of all elements)
$[-2, -5, -1]$	0	boundary (all negative entries)
$[-3, -10, -3, 15]$	15	boundary (max sum equlas last element)
example array from above	42	typical case (mixed positive and neg. entries)

8. **(10 marks)** Describe a set of white-box tests for the function `maxSubSum3`.

Solution: White-box test cases:

Input (Array A)	Output	Purpose
example from above	42	typical case (covers all code)
$[-2, -5, -1]$	0	typical case for which if-statement condition is never true (all neg. entries)
[4]	4	boundary (min. array size that covers all code)
$[-5]$	0	boundary (min. array size for which if-statement condition is never true)

Notice that complete code coverage is ensured by having an input array with at least one positive entry. All for-loops execute the loop body at least once if there is one or more elements in the array, and the condition in the if-statement will be true the first time a sum larger than zero is computed by the inner for-loop (one non-zero entry ensures this).

9. **(10 marks)** Implement a JUnit test harness called `MSSTester.java` that applies your black box test cases from Problem 7 for the MaxSubSequenceSum problem to all three implementations (`maxSubSum1`, `maxSubSum1`, `maxSubSum1`) and your white-box test cases from Problem 8 to the `maxSubSum3` function. Be sure to include this file with your submission of the assignment.

   Solution: See the file `MSSTester.java`, available on the course web page.

10. **(5 marks)** Your tests should have uncovered (at least) two errors, one in each of the `maxSubSum2` and `maxSubSum3` functions. Describe these errors and correct them in the `MaxSubsequenceSum.java` file. Be sure to include the corrected version with your submission of the assignment.

    Solution: The errors are:

    - In function `maxSubRec`, the termination condition of the for-loop that computes $rightBorderSum$ should be $i \leq right$ instead of $i < right$. The test case $[3, 10, 3]$ catches this error.
    - In `maxSubSum3`, the termination condition of the for-loop should be $j < A.length$ as opposed to $j < A.length - 1$. This error is caught by any test case for which the last element in the array is included in the maximum subsequence sum.

Bonus Questions

11. **(10 marks)** Prove that the function `maxSubSum2` runs in worst-case time $O(n \log n)$.

Solution: For simplicity, we approximate the general term of $T_2(n)$ by $2T(n/2) + \Theta(n)$ — for justification as to why the floors and ceilings can be omitted, see Section 4.4.2 of "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein. The proof can then be completed essentially as described in Section 4.1.