Programming Assignment 3 (Dynamic Programming)

Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

Instructions For Submissions

- **This assignment is to be completed individually. If you are stuck with something, consider asking the instructor for help.**
- **Submit code and a brief report**. Submission is via Canvas as a single zip file. No need to include the algorithm description in the report.
- **Any function with a compilation error will receive a zero, regardless of how much it has been completed.**

---

1 Overview

We are going to implement a few dynamic programming algorithms. To this end, **your task is to implement the following methods**:

- `isSumPossible` in `SubsetSum`
- `findOptimalProfit` in `Knapsack01`
- `computeSum`, `computeSet`, and `computeSetHelper` methods in `MWIS`
- `longestCommonSubsequence` in `LCS`
- `numberOfBinaryStringsWithNoConsecutiveOnes` in `BinaryStrings`
- `longestIncreasingSubsequence` in `LIS`

The project also contains additional files which you do not need to modify (but need to use).

Use `TestCorrectness` file to test your code. For each part, you will get an output that you can match with the output I have given to verify whether or not your code is correct. Output is provided in the `ExpectedOutput` file. You can use `www.diffchecker.com` to tally the output.

You will also write a report for the Maximum Subarray Sum problem.

1.1 C++ Helpful Hints

For C++ programmers, you must use DYNAMIC ALLOCATION to return an array. Thus, to return an array x of length 10, declared it as: **int *x = new int[10];**

1.2 Dynamic Arrays

Here, you will use their C++/Java/C# implementations of dynamic arrays, which are named respectively **vector**, **ArrayList**, and **List**. Typically, this encompasses use of generics, whereby you can create dynamic arrays of any type (and not just integer). However, you will create integer dynamic arrays here; the syntax is pretty self explanatory on how to extend this to other types (such as char, string, or even objects of a class).

- **In C++**, the syntax to create is `vector⟨int⟩ name`. To add a number (say 15) at the end of the vector, the syntax is `name.push_back(15)`. To remove the last number, the syntax is `name.pop_back()`. To access the number at an index (say 4), the syntax is `name.at(4)`.
- **In Java**, the syntax to create is `ArrayList⟨Integer⟩ name = new ArrayList⟨Integer⟩()`. To add a number (say 15) at the end of the array list, the syntax is `name.add(15)`. To remove the last number, the syntax is `name.remove(name.size() - 1)`. To access the number at an index (say 4), the syntax is `name.get(4)`.
- **In C#**, the syntax to create is `List⟨int⟩ name = new List⟨int⟩()`. To add a number (say 15) at the end of the vector, the syntax is `name.Add(15)`. To remove the last number, the syntax is `name.RemoveAt(name.Count - 1)`. To access the number at an index (say 4), the syntax is `name[4]`.

1.3 Set and Iterator

We will use set for the Subset Sum problem; the set is essentially an implementation a balanced binary search tree (a Red-Black tree to be precise). As with AVL trees, we can insert a number (or a key) at most once; then, we can search or delete it. I'll provide some of the functions and a sample example; the code will need pretty much the same ideas. I'll only focus only on integers because that's what we will store, but other data types can be accommodated using generics.

- **In C++**, the syntax to create is `set⟨int⟩ name`. The number of items in the set is given by `name.size()`. To add a number (say 15), the syntax is `name.insert(15)`. To remove a number (say 15), the syntax is `name.erase(15)`. To check if a number (say 15) exists, the syntax is `if (name.find(15) != name.end())`; the statement inside the if evaluates to true if the number is present.
- **In Java**, the syntax to create is `TreeSet⟨Integer⟩ name = new TreeSet⟨Integer⟩()`. The number of items in the set is given by `name.size()`. To add a number (say 15), the syntax is `name.add(15)`. To remove a number (say 15), the syntax is `name.remove(15)`. To check if a number (say 15) exists, the syntax is `if (name.contains(15))`; the statement inside the if evaluates to true if the number is present.
- **In C#**, the syntax to create is `SortedSet⟨int⟩ name = new SortedSet⟨int⟩()`. The number of items in the set is given by `name.Count`. To add a number (say 15), the syntax is `name.Add(15)`. To remove a number (say 15), the syntax is `name.Remove(15)`. To check if a number (say 15) exists, the syntax is `if (name.Contains(15))`; the statement inside the if evaluates to true if the number is present.

For this assignment, you will need to also read all the values that are in the set. There are multiple ways of doing this; we are going to use an an iterator (in C++/Java) or an enumerator (in C#). Starting from the minimum, the iterator points to a value in the set and can be moved to the next highest value in the set. Details:

- **In C++, the syntax is:**

```
set<int>::iterator it = name.begin(); // create iterator
while (it != name.end()) { // as long as there is a value
    int currentVal = (*it); // reads the value the iterator is pointing at
    it++; // moves to the next value
}
```

- **In Java, the syntax is:**

```
Iterator<Integer> it = name.iterator(); // create iterator
while (it.hasNext()) // as long as there is a value
    int currentVal = it.next(); // reads the value the iterator is pointing at
                                // and moves to the next value
```

- **In C#, the syntax is:**

```
IEnumerator<int> it = name.GetEnumerator(); // create enumerator
while (it.MoveNext()) // as long as there is a value, move to the next value
    int currentVal = it.Current; // reads the value at which the iterator is pointing
```

2 Subset Sum (20 points)

You are going to implement the space-efficient algorithm for solving the subset-sum problem. Recall the idea is to keep all the sums less than or equal to the target in a set; then, add each new number to the sums and add the new sums to the set. We remarked that the set can be implemented as a balanced binary-searcg tree, which is what you are going to do over here.

Implement the `isSumPossible` method using the following pseudo-code.

Space-Friendly Floating Subset Sum

- Create a set *sums*. Insert 0 into *sums*.
- for ($i = 0$ to $i < numElements$), do the following:
  - create an integer array *values* having length equaling the size of the set
  - use an iterator to read the numbers from the set and fill up the array
  - for ($j = 0$ to $j <$ the length of *values*), do the following:
    * let $val = elements[i] + values[j]$
    * if (*val equals target*) return *true*
    * else if (*val* is less than *target*), insert *val* into the *sums*
- return $false$

3 0-1 Knapsack (20 points)

You are going to implement 0-1 Knapsack, but in a space-efficient way. Recall the idea is to compute and keep only two rows at a time. You use the previous row to compute the current one, and then set the previous to be the current row; thus, in the next iteration, current becomes previous.

Implement the `findOptimalProfit` method using the following pseudo-code.

Space-Friendly 0-1 Knapsack

- Create two integer arrays *currentRow* and *prevRow* of size $(capacity + 1)$.
- Set all cells of *prevRow* to 0
- for $(i = 0$ to $i < numElements)$, do the following:
  - if $weights[i]$ is more than *capacity*, then continue
  - assign $currentRow[j] = prevRow[j]$, where $0 \leq j < weights[i]$
  - for $(j = weights[i]$ to $j <= capacity)$,
    - set $currentRow[j]$ to the maximum of $prevRow[j]$ and $\big(prevRow[j - weights[i]] + profits[i]\big)$
  - copy *currentRow* cell-by-cell to *prevRow*
- Finally, return $currentRow[capacity]$

4 Maximum Weighted Independent Set in a Tree (45 points)

Recall the maximum (weighted) independent set problem that we discussed. Here, we are going to implement it. The structure is pretty much the same, with the following minor modifications:

- The vertices in the tree are numbered 0 through $n - 1$, where n is the number of vertices.
- Instead of augmenting *incl* and *excl* values at every node of the tree, we maintain an array $computedSum[\,]$, where $computedSum[i]$ stores the maximum between *incl* and *excl* values of node i in the tree.

  Recall that when we want to create the independent set, we need to know whether *incl* value at a node is larger or smaller than *excl* value. To that end, we use another boolean array $inIncludedSumLarger[\,]$ to indicate the same, i.e., $inIncludedSumLarger[i]$ is set to *true* if the *incl* value of node i is larger than the *excl* value of the node.

  Let's see the purpose of this implementation. A boolean (1 byte) takes less space than an integer (4 bytes). Thus, $computedSum[\,]$ and $inIncludedSumLarger[\,]$ need 5 bytes per node, whereas maintaining *incl* and *excl* values need 8 bytes per node.
- Finally, we have another boolean array $isInSet[\,]$, where $isInSet[i]$ is set to *true* if node i is included in the final independent set.

Representing the Tree in Memory: We use a two-dimensional jagged array **adjList** (called *adjacency list*) to represent the tree. Specifically, row index i in the array corresponds to the node v_i, i.e., row 0 corresponds to v_0, row 1 corresponds to v_1, and so on. Each cell in row i stores the children of v_i, and each row is an integer dynamic array. Also, *adjList* is a dynamic array of these integer dynamic arrays. This implementation makes it easier to read the tree. In C++, we implement *adjList* as a vector of integer vectors. In Java, we implement *adjList* as an ArrayList of integer ArrayLists. In C#, we implement *adjList* as a List of integer Lists.

The tree contains the weights associated with each node in $weights[\,]$ array. For unweighted trees, each entry in this array is 1.

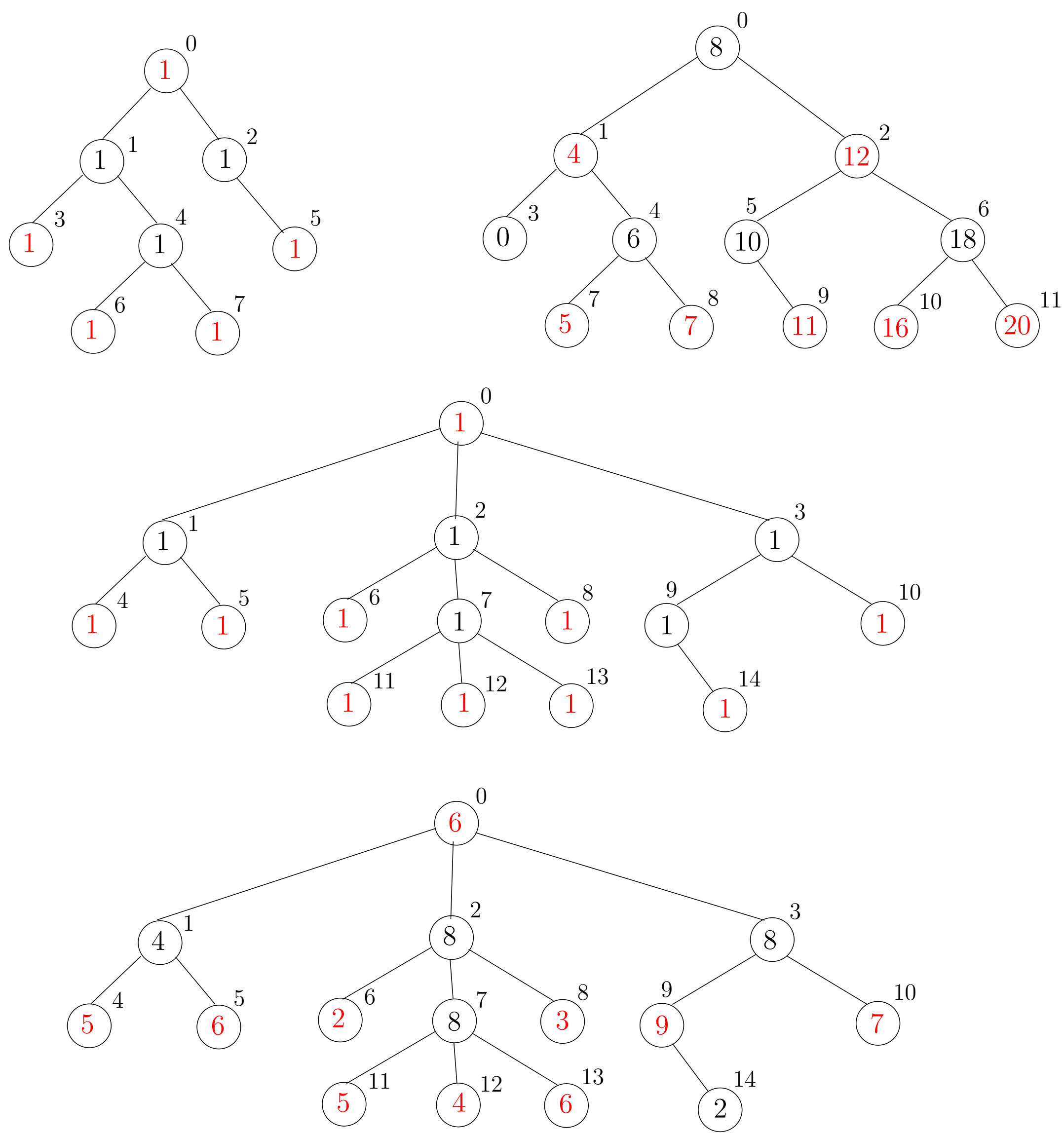

Figure 1: Trees used for Testing Weighted Maximum Set Algorithm. The numbers within the circle indicate the weight of a node, and on the top of the node is the id of the node. Red colors show the nodes that are included as the part of a weighted maximum independent set.

As an example, consider the last tree in Figure 2. Row 0 of adjList contains the following nodes: $\langle 1, 2, 3\rangle$; this is to be interpreted as vertex v_0 has 3 children – v_1, v_2, and v_3. Likewise, row 1 contains the nodes $\langle 4, 5\rangle$, row 2 contains the nodes $\langle 6, 7, 8\rangle$, and so on. In this example, $weights[0] = 6, weights[1] = 4, weights[2] = 8$, and so on.

To test the MWIS methods, I have included 4 sample files: (`mis1.txt`, `mis2.txt`, `mis3.txt`, and `mis4.txt`). For the MWIS methods to work, you MUST fill in the paths (in `TestCorrectness`) for the folder where the tree files are stored. The corresponding trees are shown above.

Implement the `computeSum`, `computeSet`, and `computeSetHelper` methods using the following.

Compute Sum

- if $computedSum[node] \neq -\infty$, then return $computedSum[node]$.
- Initialize $excl = 0$ and $incl = weights[node]$
- Let $children$ be the children of node. Specifically, $children$ is the dynamic array at index $node$ of $adjList$
- for ($i = 0$ to $i <$ size of children), do the following:
  - let $child$ be the value at index i of children
  - increment $excl$ by $computeSum(child)$
  - let $grandChildren$ be the children of $child$
  - Use the same idea above for finding $child$ to find each $grandChild$ in $grandChildren$, and do the following:
    - increment $incl$ by $computeSum(grandChild)$
- if $incl$ is more than $excl$, then set $computedSum[node] = incl$ and $isIncludedSumLarger[node] = true$, else set $computedSum[node] = excl$
- return $computedSum[node]$

Compute Set

- if included sum of $root$ is larger than excluded sum, then set $isInSet[root]$ to $true$
- for each $child$ of $root$, call $computeSetHelper(child, root)$

Compute Set Helper

- if included sum of $node$ is larger than excluded sum and parent is not included in the set, then set $isInSet[node] = true$
- for each $child$ of $node$, call $computeSetHelper(child, node)$

5 Longest Common Subsequence (35 points)

Implement the `longestCommonSubsequence` method using the following pseudo-code.

Compute Longest Common Subsequence

- Let $lenx$ and $leny$ be the lengths of x and y respectively.
- Create an integer 2d array $length$ and a char 2d array $direction$ both having $(lenx + 1)$ rows and $(leny + 1)$ columns
- Set $length[i][0] = 0$ and $direction[i][0] =$ '\0' for $0 \leq i \leq lenx$
- Set $length[0][j] = 0$ and $direction[0][j] =$ '\0' for $0 \leq j \leq leny$

- Run two nested for-loops from $i = 1$ to $i \leq lenx$, and $j = 1$ to $j \leq leny$. Within the inner loop, do the following:
  - If the character at index $i-1$ of x equals the character at index $j-1$ of y, set
    - $length[i][j] = length[i-1][j-1] + 1$
    - $direction[i][j] =$ 'D'
  - Else if $(length[i-1][j] > length[i][j-1])$, set
    - $length[i][j] = length[i-1][j]$
    - $direction[i][j] =$ 'U'
  - Else set
    - $length[i][j] = length[i][j-1]$
    - $direction[i][j] =$ 'L'
- Initialize a string answer = "";
- As long as $(direction[lenx][leny] \neq$ '\0'), do the following:
  - If $(direction[lenx][leny]$ equals 'D'),
    - append the character at index $(lenx-1)$ of x to $answer$
    - decrement both $lenx$ and $leny$ by one
  - Else if $(direction[lenx][leny]$ equals 'U'), decrement $lenx$ by one
  - Else decrement $leny$ by one
- reverse answer and then return it

6 Binary Strings with No Consecutive Ones (15 points)

Complete the `numberOfBinaryStringsWithNoConsecutiveOnes` method to find the number of binary strings of length n with no consecutive ones. If $B(n)$ is the answer for an n-length string, then $B(1) = 2$, $B(2) = 3$, and for any $n > 2$, we have $B(n) = B(n-1) + B(n-2)$.

Here's a video on how the algorithm works: https://drive.google.com/file/d/1TA2GMo_tRgBwGqxDjTnkpm2g6iOkl1bq/view?usp=sharing

You MUST write a bottom-up dynamic program that uses only $O(1)$ space, i.e., it does not use an array for storage, but a few variables is fine. You won't receive any credits if your code has exponential complexity, and will get partial credits if it uses more than constant space and/or does not use a bottom-up approach.

7 Longest Increasing Subsequence (25 points)

Complete the `longestIncreasingSubsequence` method to find the longest increasing subsequence. Once again you should use a bottom-up dynamic program. Here's a video on how the algorithm works: https://drive.google.com/file/d/1DjYYLLe5nU-03dRaIb0OsVqLaBWo5goV/view?usp=sharing. Here's a sketchy pseudo-code (I expect you to fill in the details):

Compute Longest Increasing Subsequence

- Create two integer arrays LIS and $PRED$ both of lengths len.
- For $i = 0, 1, 2, 3, \ldots, len - 1$, do the following:
  - Set $LIS[i] = 1$ and $pred[i] = -1$.
  - Using a loop, among the indexes $0, 1, 2, \ldots, i - 1$, find the index $maxIndex$ such that $arr[maxIndex] < arr[i]$ and $LIS[maxIndex]$ is the maximum of all the values among $LIS[0], LIS[1], \ldots, LIS[i - 1]$.

    If the values $arr[0], arr[1], \ldots, arr[i - 1]$ are all greater than $arr[i]$, then let $maxIndex = -1$
  - If $maxIndex \neq -1$, set $LIS[i] = LIS[maxIndex] + 1$ and $PRED[i] = maxIndex$
- Find $lisIndex$, which is the index containing the maximum value in the $LIS[\,]$ array
- Create a dynamic integer array.
- Starting from $lisIndex$ and by using the $PRED$ array, add the values in the longest increasing subsequence to the dynamic array
- Reverse the dynamic array using the given helper function, and then return it.

8 Maximum Subarray Sum (15 points)

You don't need to write any code for this part. Instead, write a brief report on the following.

- Read the notes on the Maximum Subarray Sum problem. [**2 points**]
- In light of the numbers obtained by running `TestTime`, which of the following improvements is more – Method 1 to Method 2, or Method 2 to Method 3? [**1 point**]
- Under what programming paradigm (Brute-force, Greedy, Divide & Conquer, and Dynamic Programming) does each of the three methods fall under? [**3 points**]
- What are the complexities of these 3 algorithms? Explain briefly. [**6 points**]
- Do the complexities align with the numbers from TestTime? Explain briefly. [**3 points**]