# 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**. Submission is via Canvas as a single zip file.
- **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**:

- `findOptimalProfit` in `Knapsack01`
- `computeSum`, `computeSet`, and `computeSetHelper` methods in `MWIS`
- `longestCommonSubsequence` in `LCS`
- `longestIncreasingSubsequence` in `LIS`
- `findBestPath` in `VankinsMile`

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.

### 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 Multidimensional arrays

*A 2d array* is one which has fixed number of columns for each row.

- **In C++**, to create a 2d array having *numRows* rows and *numCols* columns, the syntax is

```
int **A = new int*[numRows];
for (int i = 0; i < numRows; i++)
    A[i] = new int[numCols];
```

  To access cell at row-index $r$ and column-index $c$, use `A[r][c]`

- **In JAVA**, to create a 2d array having *numRows* rows and *numCols* columns, the syntax is `int A[ ][ ] = new int[numRows][numCols]`

  To access cell at row-index $r$ and column-index $c$, use `A[r][c]`

- **In C#**, to create a 2d array having *numRows* rows and *numCols* columns, the syntax is `int[,] A = new int[numRows,numCols]`

  To access cell at row-index $r$ and column-index $c$, use `A[r,c]`

## 1.3 Operations on Strings

To get the length of a string *str*, use *str.length*() in C++, *str.length*() in Java, and *str.Length* in C#. To get hold of the character at index $i$ of a string *str*, use *str.at*($i$) in C++, *str.charAt*($i$) in Java, and *str*[$i$] in C#. To append a character $c$ to a string *str*, use $str = str + c$.

## 1.4 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⟨⟩()`. 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]`.

## 1.5 Sets and Maps

A Set is essentially a set in the typical Math terminology – contains unique elements (or more commonly referred to as *keys*). A Map on the other hand consists of map entries, where each map entry comprises of a *key* and *value* pair. You can search the map for a particular key, and if it exists, retrieve the corresponding value back. Thus a map essentially maps an element (called *key*) uniquely to another element (called *value*).

Sets are implemented as a balanced search tree (aka ordered sets) or as a hash-table (aka unordered sets). Maps can be implemented as a balanced search tree (aka ordered maps) or as a hash-table (aka unordered maps). In both cases, keys can be of any type (integers, floats, strings, characters, objects of a class); likewise, values (in case of maps) can also be of any type. For unordered sets and maps, we must be able to check whether or not two keys are the same. For ordered sets and maps, we must be able to order two keys (such as dictionary order for strings).

Operations on an ordered set/map are typically supported in $O(\log n)$ time, where $n$ is the number of items in the set/map. Operations on an unordered set/map are typically supported in $O(1)$ expected (average) time but in the worst-case it may take $O(n)$ time, where $n$ is the number of items in the set/map. For this assignment, we will assume $O(1)$ time for unordered sets/maps as the worst-case rarely happens.

An important thing to consider is that the *keys* that are stored in ordered sets and maps are, as the name suggests, *ordered*, such as in numeric order (for integers and floats), or in lexicographic order (for strings). Therefore, whenever an application demands that keys be stored in some well-defined order, we should consider ordered sets and maps. On the other hand, when the order is not important, then we should use unordered sets and maps because they are faster.

Over here, we will see a few applications of sets and maps. In the project folder you will find videos explanations of how some of these and related applications are implemented via sets and maps. Let's first see how they are implemented in C++ and Java.

### 1.5.1 C++

I will list some of the functions of sets and maps (both ordered and unordered). For more details, check out C++'s documentation.

#### Ordered and Unordered Set

- To create an integer ORDERED set: `set<int> mySet;`

  To create an integer UNORDERED set: `unordered_set<int> mySet;`

  Note that you may need to change the types of keys stored according to the application.

- To get the size of the set: `mySet.size();`

- To insert an integer $x$: `mySet.insert(x);` this will add $x$ if it is not present, else no change.

- To check if an integer $x$ is already in the set: `if(mySet.find(x) != mySet.end())`. The statement inside the if evaluates to true if and only if $x$ is already in the set.

- To create an iterator on an (ORDERED or UNORDERED) set and print all values:

  ```
  set<int>::iterator it = mySet.begin();
  while (it != mySet.end()) {
      cout << *it << " ";
      ++it;
  }
  ```

  For unordered set, just create `unordered_set<int>::iterator it`; rest is the same.

### Ordered and Unordered Map

- To create an ORDERED map with integer keys and character values: `map<int, char> myMap;`

  To create an UNORDERED map with integer keys and character values: `unordered_map<int, char> myMap;`

  Note that you may need to change the types of keys and values according to the application.

- To get the size of the map: `myMap.size();`

- To insert an integer *key-value* pair: `myMap[key] = value;` this will add the pair if it is not present, else it will update the existing value of *key* with the "new" value.

- To check if the map already contains a particular *key*: `if(myMap.find(key) != myMap.end())`

  The statement inside the if evaluates to true if and only if *key* is already in the map.

- To retrieve the value corresponding to a particular *key*: `char value = myMap[key];`

  If *key* is not present, since this method won't typically return a NULL, to avoid unexpected results, you should first check if *key* is present before using this.

- To create an iterator on an (ORDERED or UNORDERED) map and print all keys/values:

```
// create an iterator on the set of keys stored in the map
map<int, char>::iterator it = myMap.begin();
while (it != myMap.end()) { // as long as there is an entry
    int key  = it -> first; // obtain the key from the iterator
    char value = myMap[key]; // get the value corresponding to the key
                             // can also use: char value = it -> second;
    ++it; // move the iterator to the next entry (if exists)
    cout << key << ": " << value << endl;
}
```

  For unordered map, just create `unordered_map<int, char>::iterator it`; rest is the same.

### 1.5.2 Java

I will list some of the functions of sets and maps (both ordered and unordered). For more details, check out Java's documentation.

### Ordered and Unordered Set

- To create an integer ORDERED set: `TreeSet<Integer> mySet = new TreeSet<>();`

  To create an integer UNORDERED set: `HashSet<Integer> mySet = new HashSet<>();`

  Note that you may need to change the types of keys stored according to the application.

- To get the size of the set: `mySet.size();`

- To insert an integer $x$: `mySet.add(x);` this will add $x$ if it is not present, else no change.

- To check if an integer $x$ is already in the set: `if(mySet.contains(x))`. The statement inside the if evaluates to true if and only if $x$ is already in the set.
- To create an iterator on a set (ORDERED or UNORDERED) and print all values:

```
Iterator<Integer> it = mySet.iterator();
while (it.hasNext())
    System.out.print(it.next() + " ");
```

### Ordered and Unordered Map

- To create an ORDERED map with integer keys and character values: `TreeMap<Integer, Character> myMap = new TreeMap<>();`

  To create an UNORDERED map with integer keys and character values: `HashMap<Integer, Character> myMap = new HashMap<>();`

  Note that you may need to change the types of keys and values according to the application.
- To get the size of the map: `myMap.size();`
- To insert an integer *key-value* pair: `myMap.put(key, value);` this will add the pair if it is not present, else it will update the existing value of *key* with the "new" value.
- To check if the map already contains a particular *key*: `if(myMap.containsKey(key))`

  The statement inside the if evaluates to true if and only if *key* is already in the map.
- To retrieve the value corresponding to a particular *key*: `Character value = myMap.get(key);`

  The method returns *null* if *key* is not present.
- To create an iterator on a map (ORDERED or UNORDERED) and print all keys/values:

```
// create an iterator on the set of keys stored in the map
Iterator<Integer> it = myMap.keySet().iterator();
while (it.hasNext()) { // as long as there is a key
    Integer key = it.next(); // obtain the key from the iterator,
                             // and move iterator to the next key (if exists)
    Character value = myMap.get(key); // get the value corresponding to the key
    System.out.println(key+ ": " + value);
}
```

### 1.5.3 C#

I will list some of the functions of sets and maps (both ordered and unordered). For more details, check out C#'s documentation.

### Ordered and Unordered Set

- To create an integer ORDERED set: `SortedSet<int> mySet = new SortedSet<int>();`

  To create an integer UNORDERED set: `HashSet<int> mySet = new HashSet<int>();`

  Note that you may need to change the types of keys stored according to the application.

- To get the size of the set: `mySet.Count;`
- To insert an integer $x$: `mySet.Add(x);` this will add $x$ if it is not present, else no change.
- To check if an integer $x$ is already in the set: `if(mySet.Contains(x))`. The statement inside the if evaluates to true if and only if $x$ is already in the set.
- To create an iterator on a set (ORDERED or UNORDERED) and print all values:

```
IEnumerator<int> it = mySet.GetEnumerator();
while (it.MoveNext())
   Console.Write(it.Current + " ");
```

**Ordered and Unordered Map**

- To create an ORDERED map with integer keys and character values: `SortedDictionary<int, char> myMap = new SortedDictionary<int, char>();`

  To create an UNORDERED map with integer keys and character values: `Dictionary<int, char> myMap = new Dictionary<int, char>();`

  Note that you may need to change the types of keys and values according to the application.
- To get the size of the map: `myMap.Count;`
- To insert an integer *key-value* pair: `myMap[key] = value;` this will add the pair if it is not present, else it will update the existing value of *key* with the "new" value.
- To check if the map already contains a particular *key*: `if(myMap.ContainsKey(key))`

  The statement inside the if evaluates to true if and only if *key* is already in the map.
- To retrieve the value corresponding to a particular *key*: `char value = myMap[key];`

  The method returns *null* if *key* is not present.
- To create an iterator on a map (ORDERED or UNORDERED) and print all keys/values:

```
// create an iterator on the set of keys stored in the map
IEnumerator<int> it = myMap.Keys.GetEnumerator();
while (it.MoveNext()) { // as long as there is a key move to the next key
    int key = it.Current; // obtain the key from the iterator
    char value = myMap[key]; // get the value corresponding to the key
    Console.WriteLine(key+ ": " + value);
}
```

## 2 Subset Sum (no coding required)

The task is to understand how the space-efficient subset sum algorithm works. 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 remark that the set can be implemented as a balanced binary-search tree or a hash-table. We have implemented it using balanced BST. Below is an outline of the pseudo-code. Compare this to the code provided to see how it has been implemented.

Space-Friendly 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 (40 points)

You are going to implement 0-1 Knapsack, but in a space-efficient way similar to what has been done for subset-sum. The idea is essentially keep track of all weights less than or equal to the capacity that have been formed, and for each weight, keep the maximum profit. Keep track of the maximum profit and finally return it.

Implement the `findOptimalProfit` function using the following pseudo-code. Your algorithm must need space $O(\alpha)$ for any credit, where $\alpha$ is the number of unique weights less than or equal to the maximum capacity that can be formed.

Space-Friendly 0-1 Knapsack

- Create an integer-integer map called *weightsToProfitsPrev*. This will store all $\langle weight, profit\rangle$ key-value pairs that have been formed.
- For key = 0, add value = 0 to *weightsToProfitsPrev*. For weight 0, the profit is 0.
- Let $max = -\infty$. This will be used to keep track of the maximum profit found so far.

  In C++, use INT_MIN for $-\infty$. In Java, use Integer.MIN_VALUE for $-\infty$. In C#, use Int32.MinValue for $-\infty$.
- Run a loop from $i = 0$ to $i < numElements$
  - Create another integer-integer map called *weightsToProfitsCurr*. This will store all $\langle weight, profit\rangle$ pairs formed from the existing ones and the current item.
  - Iterate over *weightsToProfitsPrev*. Within this loop, we add all existing weight-profit pairs to *weightsToProfitsCurr*:
    * Let $w$ = iterator's key. We retrieve the weight.
    * Let $p$ = iterator's value. We retrieve the profit for the weight.
    * Add key = $w$ and value = $p$ to *weightsToProfitsCurr*.
  - Iterate over *weightsToProfitsPrev*. Within this loop
    * Let $w$ = iterator's key. We retrieve the weight.

* Let $p$ = iterator's value. We retrieve the profit for the weight.
    * Let $weightNew = w + weights[i]$. This is the weight that we are getting by adding the weight of the current item to the existing one.
    * if ($weightNew > capacity$) then continue. We exceed the Knapsack capacity and so it should not be considered.
    * Let $profitNew = p + profits[i]$. We are computing the profit if we include the current item.
    * if $weightsToProfitsCurr$ contains the key $weightNew$
        - There is already a profit corresponding to $weightNew$. Let $profitExisting$ be the profit (value) corresponding to $weightNew$ (key) in $weightsToProfitsCurr$
        - if ($profitExisting < profitNew$) then we have a higher profit. So, add key = $weightNew$ and value = $profitNew$ to $weightsToProfitsCurr$

      else
        - Profit for $weightNew$ has not been computed. So, add key = $weightNew$ and $value = profitNew$ to $weightsToProfitsCurr$
    * if ($max$ is less than $profitNew$) then we have a new highest profit. So, set $max = profitNew$
  - Set $weightsToProfitsPrev$ to $weightsToProfitsCurr$. We have the new set of $\langle weight, profit\rangle$ key-value pairs. So, we update.
- return max. We return the maximum profit.

## 4 Maximum Weighted Independent Set in a Tree (50 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.

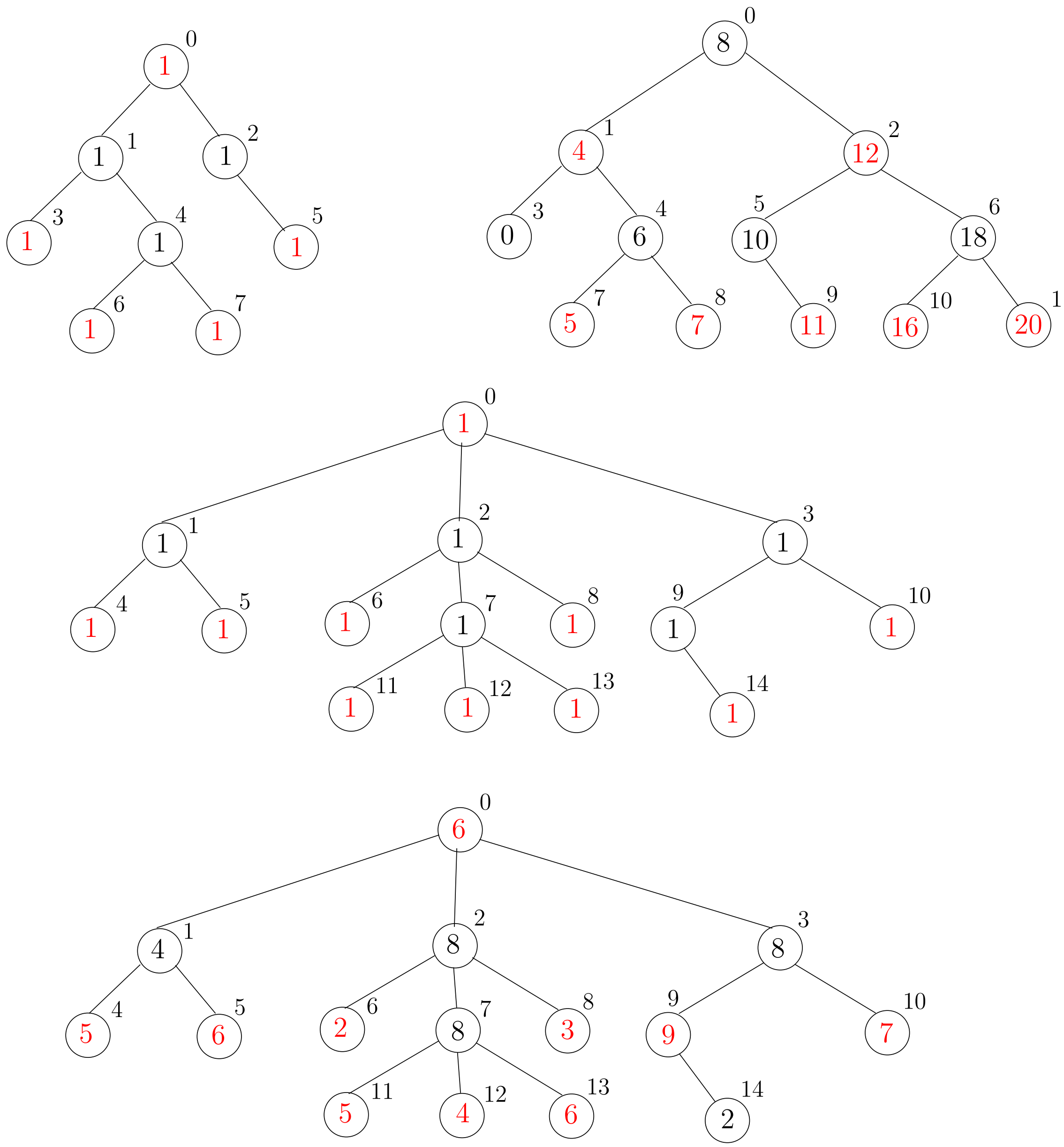

Figure 1: Trees used for Testing Weighted Maximum Set Algorithm. The numbers within the circle indicate the weight of a node, and on the top of the node is the id of the node. Red colors show the nodes that are included as the part of a weighted maximum independent set.

**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.

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 $computedSum[node]$.

  In C++, use INT_MIN for $-\infty$. In Java, use Integer.MIN_VALUE for $-\infty$. In C#, use Int32.MinValue for $-\infty$.

- Initialize $excl = 0$ and $incl = weights[node]$
- Let *children* be node's children, i.e., *children* is the row at index *node* of *adjList*
- for (each *child* in *children*), do the following:
  - increment *excl* by $computeSum(child)$
  - let *grandChildren* be the children of *child*, i.e., *grandChildren* is the row at index *child* of *adjList*
  - for (each *grandChild* in *grandchildren*), do the following:
    - increment *incl* by $computeSum(grandChild)$
- if $(incl > excl)$, then
  - set $computedSum[node] = incl$
  - set $isIncludedSumLarger[node] = true$

  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 (40 points)

We are going to implement LCS but in a space-efficient way, i.e., without using the `direction` matrix. The idea is that we can simulate the contents of the `direction` matrix just by looking at the two strings and the `length` matrix. Implement the `longestCommonSubsequence` method using the following pseudo-code. Your code must only use the `length` matrix for any credit.

**Compute Longest Common Subsequence without storing Directions**

- Let $lenx$ and $leny$ be the lengths of the strings $x$ and $y$ respectively.
- Create an integer 2d array $length$ having $(lenx + 1)$ rows and $(leny + 1)$ columns
- Using a loop, set $length[i][0] = 0$ '\0' for $0 \leq i \leq lenx$
- Using a loop, set $length[0][j] = 0$ '\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 $length[i][j] = length[i - 1][j - 1] + 1$
  - Else if $(length[i - 1][j] > length[i][j - 1])$
    - set $length[i][j] = length[i - 1][j]$
  - Else
    - set $length[i][j] = length[i][j - 1]$
- Initialize a string answer = "";
- As long as $(lenx > 0$ and $leny > 0)$, do the following:
  - If the character at index $lenx - 1$ of $x$ equals the character at index $leny - 1$ of $y$, we should be picking this character in our LCS and move diagonally. So,
    - append the character at index $(lenx - 1)$ of $x$ to $answer$
    - decrement both $lenx$ and $leny$ by one
  - Else if $(length[lenx - 1][leny] > length[lenx][leny - 1])$, we should move a cell up. So, decrement $lenx$ by one
  - Else we should move a cell to the left. So, decrement $leny$ by one
- reverse answer and then return it

## 6 Longest Increasing Subsequence (30 points)

Complete the `longestIncreasingSubsequence` method to find the longest increasing subsequence. Once again you should use a bottom-up dynamic program. Here's a sketchy pseudo-code:

**Compute Longest Increasing Subsequence**

- Create two integer arrays *length* and *pred* both of lengths *len*.
- For $i = 0, 1, 2, 3, \ldots, len - 1$, do the following:
  - Set $length[i] = 1$ and $pred[i] = -1$
  - Run a loop from $j = 0$ to $j < i$, Within this loop:
    - if ($arr[j] < arr[i]$ and $length[j] + 1 > length[i]$))
      - set $length[i] = length[j] + 1$
      - set $pred[i] = j$
- Find *lisIndex*, which is the index containing the maximum in the *length*[ ] array
- Create a dynamic integer array.
- while ($lisIndex \geq 0$)
  - add $arr[lisIndex]$ to the dynamic array
  - set *lisIndex* to $pred[lisIndex]$
- Reverse the dynamic array using the given helper function, and then return it.

## 7 Vankin's Mile (40 points)

Vankin's Mile is a board game.[1] As an input, we are given a two-dimensional matrix *board*[ ][ ] that has $R$ rows and $C$ columns. We are given a start cell in the board given by *startRow* and *startCol* as the the indexes. We can escape the board either to the right of the last column ($C - 1$) or below the last row ($R - 1$). Our movements are restricted to either a move right or a move down, both by a single cell. As we traverse through the matrix we accumulate the value stored in each cell, i.e., if we traverse $board[r][c]$, then $board[r + 1][c]$, and then $board[r + 1][c + 1]$, we will gain/lose the value ($board[r][c] + board[r + 1][c] + board[r + 1][c + 1]$).

The task is to escape the board by gaining the most wealth (or losing the least wealth) that is possible for the starting position. See Figure 2 for three examples at the top which shows in red the optimal path taken. The total value gained/lost is the sum of the red numbers; you can verify that if you take another path you cannot get anything better.

Now check the solution boards, which shows the paths taken in blue (notice that they are the exact same sequence of indexes as the red path in the figures above). If the solution board is given to us, then we can figure out the path as follows:

- Find the maximum value among the cells in the last row and last column. These are 23, 73, and 9 in the three figures respectively. Ties can be broken arbitrarily.
- Starting from this max cell use the direction markers to traverse: move up for $U$ and move left for $L$. Keep traversing until you reach the cell whose direction marker is $S$ (indicates start cell). Collect cells as you see.

[1] Thanks to Jeff Erickson of University of Illinois - Urbana Champaign for bringing this game to our attention as a classic example of Dynamic Programming.

- Reverse the sequence of cells, and that's your answer.

So, if we can obtain the solution board, then we can figure out the path. To this end, we are going to describe $Vankin(r, c)$ as a function which determines what is the most valuable way to go from the cell $[startRow, startCol]$ to the cell $[r, c]$. To solve $Vankin(r, c)$, we use the following recursion:

- **Base-case 1**: For $r < startRow$ and for $c < startCol$, we cannot traverse to the cells given by $board[r][c]$ given the movement restrictions. So, initially, for $r < R$ and for $c < C$ set (we will overwrite the "reachable" cells later):

  $$Vankin(r, c) = -\infty$$

  In C++, use INT_MIN for $-\infty$. In Java, use Integer.MIN_VALUE for $-\infty$. In C#, use Int32.MinValue for $-\infty$.

- **Base-case 2**: For $c \geq startCol$ and $c < C$ with the row fixed at $startRow$, we can only move right. That means at index $[startRow, c]$, we have gained/lost by summing up the cells $[startRow, startCol], [startRow, startCol + 1], \ldots, [startRow, c]$. Hence,

  $$Vankin(startRow, c) = \sum_{j=startCol}^{c} board[startRow][j]$$

- **Base-case 3**: For $r \geq startRow$ and $r < R$ with the column fixed at $startCol$, we can only move down. That means at index $[r, startCol]$, we have gained/lost by summing up the cells $[startRow, startCol], [startRow + 1, startCol], \ldots, [r, startCol]$. Hence,

  $$Vankin(r, startCol) = \sum_{i=startRow}^{r} board[i][startCol]$$

- **Recurrence**: For $r > startRow$ and for $c > startCol$ (nested), the max gain to cell $[r, c]$ is $board[r][c]$ plus the maximum of $Vankin(r-1, c)$ and $Vankin(r, c-1)$ Hence,

  $$Vankin(r, c) = board[r][c] + max\{Vankin(r-1, c), Vankin(r, c-1)\}$$

Notice that this recurrence is very similar to the LCS problem; in fact, this is slightly simpler as we do not have to worry about the diagonal traversal. So, you can use similar ideas here by creating the solution board split into two matrices – $values[\ ][\ ]$ and $directions[\ ][\ ]$. Here $values[r][c]$ represent the max gain to reach cell $[r, c]$. Similarly, $directions[r, c] = L$ or $directions[r, c] = U$ will determine the direction (left or up) from which $values[r, c]$ was computed in the recursion (or base-case). Additionally, $directions[r, c] = *$ when $values[r][c] = -\infty$ and $directions[r, c] = S$ when $[r, c] = [startRow, startCol]$ represents the starting cell.

When you have successfully computed these two matrices, just add the following code at the end to print the solution board and the path:

```
pathFinder(values, directions, numRows, numCols, startRow, startCol);
```

Your code must use a bottom-up dynamic program with complexity $O(RC)$ for any credit.

### *** Input Boards ***

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | -1 | **7** | **-2** | 10 | -5 |
| 1 | 8 | -4 | **3** | -6 | 0 |
| 2 | 5 | 1 | **5** | **6** | -5 |
| 3 | -7 | -4 | 1 | **-4** | **8** |
| 4 | 7 | 1 | -9 | 4 | 0 |

**Start at [0,1]**

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | -1 | 7 | -2 | 10 | -5 |
| 1 | **8** | -4 | 3 | -6 | 0 |
| 2 | **5** | 1 | 5 | 6 | -5 |
| 3 | **-7** | -4 | 1 | -4 | 8 |
| 4 | **7** | 1 | -9 | 4 | 0 |
| 5 | **4** | **43** | **13** | -12 | 4 |

**Start at [1,0]**

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | -1 | 7 | -2 | 10 | -5 |
| 1 | 8 | -4 | 3 | -6 | 0 |
| 2 | 5 | 1 | 5 | 6 | -5 |
| 3 | -7 | -4 | **1** | **-4** | **8** |
| 4 | 7 | 1 | -9 | 4 | **0** |
| 5 | 4 | 43 | 13 | -12 | **4** |

**Start at [3,2]**

### *** Solution (Value/Direction) Boards ***

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | $-\infty/*$ | **7/S** | **5/L** | 15/L | 10/L |
| 1 | $-\infty/*$ | 3/U | **8/U** | 9/U | 10/U |
| 2 | $-\infty/*$ | 4/U | **13/U** | **19/L** | 14/L |
| 3 | $-\infty/*$ | 0/U | 14/U | **15/U** | **23/L** |
| 4 | $-\infty/*$ | 1/U | 5/U | 19/U | 23/U |

**Max is 23 at [3,4]**

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ |
| 1 | **8/S** | 4/L | 7/L | 1/L | 1/L |
| 2 | **13/U** | 14/L | 19/L | 25/L | 20/L |
| 3 | **6/U** | 10/U | 20/U | 21/U | 29/L |
| 4 | **13/U** | 14/L | 11/U | 25/U | 29/U |
| 5 | **17/U** | **60/L** | **73/L** | 61/L | 65/L |

**Max is 73 at [5,2]**

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ |
| 1 | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ |
| 2 | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ | $-\infty/*$ |
| 3 | $-\infty/*$ | $-\infty/*$ | **1/S** | **-3/L** | **5/L** |
| 4 | $-\infty/*$ | $-\infty/*$ | -8/U | 1/U | **5/U** |
| 5 | $-\infty/*$ | $-\infty/*$ | 5/U | -7/L | **9/U** |

**Max is 9 at [5,4]**

Figure 2: Vankin's Mile Input and Solution Examples