Dynamic Programming Exercises

Ibrahim Albluwi Sufyan Almajali

Exercise 1 (General DP)

A. Consider the following recursive definition of function **F**:

```
F(n) = 0                 if n = 0
F(n) = 1                 if n = 1
F(n) = F(n-2) + F(n/2)  if n > 1
```

1. Show that this function has overlapping subproblems.

2. Assuming that a *memoized* dynamic programming solution was implemented for this function, how many subproblems will be solved if **F(9)** is called?

B. The Useless Pair Cost (**UPC**) of two numbers **n** and **m** is defined as follows:

```
UPC(n, m) = 1                               if n <= 1 or m <= 1
UPC(n, m) = UPC(n-1, m-1) + UPC(n/2, m/2)   Otherwise
```

If you were asked to write the code for computing the **UPC** of two numbers, would you use dynamic programming or not? Justify your answer and draw diagrams if you need.

Exercise 2 (Know Thy Problems)

A. Tracing

ToDo

B. Reverse Engineering.

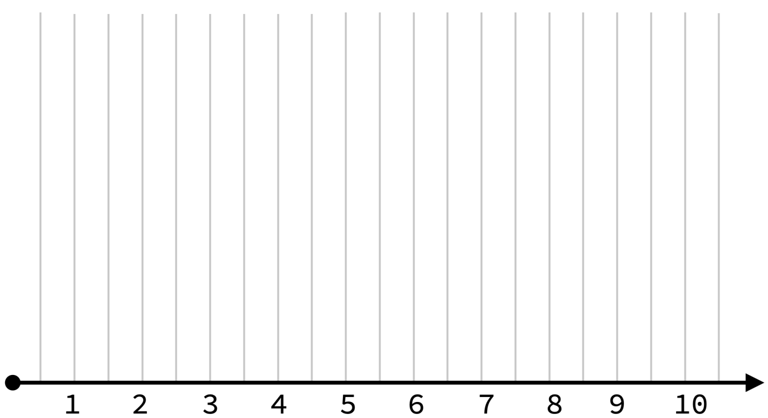
The following exercises assume the dynamic programming solutions discussed in class. Assuming other valid DP solutions that were not covered in class will most likely not be helpful.

1. The following dynamic programming array resulted from solving an instance of the **0-1 Knapsack** problem using bottom-up dynamic programming. What was the capacity of the knapsack? What were the values and weights of the items?

0	0	0	0
0	0	5	5
0	0	5	10

2. The following dynamic programming array resulted from solving an instance of the Weighted **Activity Selection** problem using bottom-up dynamic programming. Assuming that the activity values are `v[] = {4, 3, 2, 1}`, draw on the timeline below four activities that could lead to the array.

5	3	2	1	0



3. The following dynamic programming array resulted from solving an instance of the **Longest Common Subsequence** problem using bottom-up dynamicprogramming. What were the two strings?(provide any two strings that could produce this array)

0	0	0	0
0	0	0	1
0	0	1	1
0	1	1	1

Exercise 3 (Design)

A. The **Infinite 0-1 Knapsack** Problem is a variant of the 0-1 Knapsack problem studied in class, where it is allowed to take an item an infinite number of times (but taking a fraction of an item is not allowed).

Design a dynamic programming solution for this problem. Your task is only to complete the following recursive definition for **`KNAPSACK(i, j)`** which captures the optimal substructure of the problem, where **`KNAPSACK(i, j)`** is the optimal solution considering the items **`0`** to **`i`** and a knapsack of capacity **`j`**.

(**`w[i]`** = weight of item **`i`**, **`v[i]`** = value of item **`i`**, **`W`** = knapsack capacity, **`N`** = number of items)

```
                     ⎧                                    if (i = 0 or j = 0)
                     ⎪
                     ⎪                                    if (w[i] > W)
KNAPSACK(i, j) =     ⎨
                     ⎪                                    otherwise
                     ⎪
                     ⎩
```

B. Consider the 0-1 Knapsack problem studied in class. Assume that our goal is to maximize the ***total weight*** of items in the knapsack instead of the total value (profit), but the other constraints are the same.

Assume that **`OPT(i, j)`** is the maximum sum of weights possible for items that fit in a knapsack of capacity **`j`**, considering the first **`i`** items only. Provide a recursive definition for **`OPT(i, j)`** that captures the optimal substructure of the problem. (Do not forget the base cases!)

(**`w[i]`** = weight of item **`i`**, **`v[i]`** = value of item **`i`**, **`W`** = knapsack capacity, **`N`** = number of items)

C. Consider three strings, **X**, **Y**, and **Z**. Assume that **`LCS(i, j, k)`** is the length of the longest common subsequence between the first **`i`**, **`j`** and **`k`** characters of **X**, **Y** and **Z** respectively. Provide a recursive definition for **`LCS(i, j, k)`** that captures the optimal substructure of the problem. (Do not forget the base cases!)

$$\texttt{LCS(i, j, k)} = \begin{cases} \\ \\ \\ \end{cases}$$

D. Consider the following recurrence equation capturing the optimal substructure for the **collecting apples** problem we studied in class (base cases ignored).

```
MAX_APPLES(i, j) = apples[i][j] + max(MAX_APPLES(i-1, j),
                                      MAX_APPLES(i, j-1))
```

Write a new recurrence equation capturing the optimal substructure of a new version of the problem. The new version has the following changes:

1. Your goal is to collect the **least** number of apples from the top-left corner to the bottom-right corner in the grid.
2. You have 3 types of moves: **`DOWN`**, **`RIGHT`**, **`2RIGHT`**.
   `2RIGHT` allows moving 2 cells to the right (jumping over the right cell without taking the apples).

(Do not forget the base cases)

E. Consider **N** houses next to each other on a road. House **i** is willing to pay charity **c[i]** provided that you did not collect charity from the house directly before it. What is the maximum amount of money that you can collect for your charity campaign?

Examples.

`c[] = {10, 2, 3}`	maximum amount = **13** (pick 10 and 3)
`c[] = {2, 1, 1, 5, 1}`	maximum amount = **7** (pick 1 and 5)
`c[] = {1, 5, 1, 1, 7, 10}`	maximum amount = **16** (pick 5, 1 and 10)

Design a dynamic programming solution for the problem above, following the steps below.

1. Assume that **OPT(i)** is the optimal solution for the problem considering houses **i** to **N-1**. Provide a recursive definition for **OPT(i)** that captures the optimal substructure. Indicate also what the base case(s) is (are).

2. Show that your recursive definition leads to overlapping sub-problems.

3. Draw the table that can be used by the dynamic programming algorithm. Fill the table if the houses are willing to pay the following charities: **c[] = {1, 2, 3, 4}**.
   Clearly indicate where the final answer is in the table.

Exercise 4 (Implementation)

A. Consider a variant of the ***weighted activity selection*** problem, where taking more than **`m`** activities (out of the given **`n`** activities) is not allowed. Consider also the following recursive definition for the optimal solution **`OPT(i, m)`** considering the activities :

```
OPT(i, m)  =  0                                    if m <= 0 or i >= n
           =  MAX(OPT(i+1, m),
                  OPT(NEXT(i), m-1) + v[i])       otherwise
```

Convert the above recursive definition of **`OPT(i, m)`** to a bottom-up dynamic programming implementation. Assume the following:

- The activities are sorted according to their start times.
- The value of activity i is stored at **`v[i]`**, which is accessible globally.
- **`NEXT(i)`** returns the index **`j`** of the first activity compatible with **`i`**, where **`j > i`**.

B. Consider the following definition of the Binomial Coefficient:

$$\binom{n}{k} = \left\{ \begin{array}{ll} 1 & \text{if } k = 0 \text{ or } n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \end{array} \right\}$$

Write the pseudocode of the function **`BinCoeff(n, k)`**, which uses bottom-up dynamic programming to compute and return the binomial coefficient. Assume that **n** and **k** are guaranteed to be positive.

C. The ***Edit Distance*** between two strings **X** (of size **N**) and **Y** (of size **M**) is defined as the minimum number of operations required to make **X = Y**. The operations that can be performed are: removing a character, inserting a character or substituting a character.

For example:

- The edit distance between **X=abc** and **Y=abb** is **1**, since we can substitute the **c** in **X** with a **b** to make **X=Y**.
- The edit distance between **X=ab** and **Y=abb** is **1**, since we can either add a **b** to **X** or remove a **b** from **Y** to make **X=Y**.

The following is a recursive algorithm that correctly finds the edit distance between two strings:

```
DIST(X, Y, N, M)
    IF (N == 0)       // if X is empty, we can make X=Y by
        RETURN M      // removing all of the characters in Y
    IF (M == 0)       // if Y is empty, we can make X=Y by
        RETURN N      // removing all of the characters in X

    // if the two characters are the same, no changes are needed.
    IF (X[N] == Y[M]) RETURN DIST(X, Y, N-1, Y-1)

    // if the characters are not the same an operation is needed.
    RETURN 1 + MIN( DIST(X, Y, N-1, M),     // Insert
                    DIST(X, Y, N, M-1),     // Remove
                    DIST(X, Y, N-1, M-1))   // Replace
```

Implement a bottom-up dynamic programming algorithm for the problem above.

D. Consider a new problem known as **Subset Sum**. Given an array **`data`**`[]` of **n** strictly positive integers, and an integer **k**, the goal is to return the number of subsets in the `data[]` array that sum to exactly **k**.

Consider the following recursive definition capturing the optimal solution of the problem.

```
OPT(k, i) = 1                                       IF k = 0
            0                                       IF k < 0 OR if i = n
            OPT(k-data[i], i+1) + OPT(k, i+1)      otherwise
```

Convert the above recursive definition of `OPT(k, i)` to a bottom-up dynamic programming implementation.

z

E. Solve the above implementation exercises again using *memoization* instead of bottom-up iteration.

Solutions

Exercise 1 (General DP)

A.1. `F(12)` requests `F(10)` and `F(6)`
`F(10)` requests `F(8)` and `F(5)`
`F(8)` requests `F(6)` and `F(4)` ← Bingo! `F(6)` was requested twice!

A.2. `F(9)` requests `F(7)` and `F(4)`
`F(7)` requests `F(5)` and `F(3)`
`F(5)` requests `F(3)` and `F(2)`
`F(3)` requests `F(1)` and `F(1)`
`F(4)` requests `F(2)` and `F(2)`
`F(2)` requests `F(0)` and `F(1)`

All subproblems F(0) → F(9) will be solved except F(6) and F(8).
Answer = 8 subproblems will be solved.

B. **ToDo**

Exercise 2 (Know Thy Problems)

A. Trace, Fill Table.

B. Reverse Engineering.

The following exercises assume the dynamic programming solutions discussed in class. Assuming other valid DP solutions that were not covered in class will most likely not be helpful.

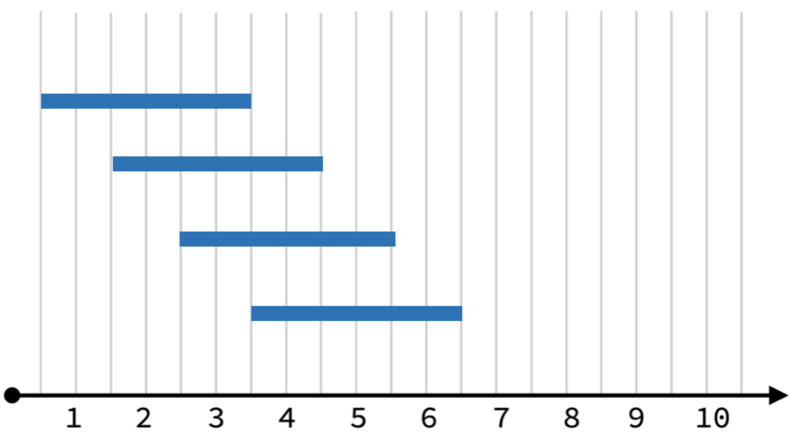
4. The following dynamic programming array resulted from solving an instance of the **0-1 Knapsack** problem using bottom-up dynamic programming. What was the capacity of the knapsack? What were the values and weights of the items?

0	0	0	0
0	0	5	5
0	0	5	10

```
items       0     1
weights = {2,     3}
values  = {5,    10}
```

5. The following dynamic programming array resulted from solving an instance of the Weighted **Activity Selection** problem using bottom-up dynamic programming. Assuming that the activity values are `v[] = {4, 3, 2, 1}`, draw on the timeline below four activities that could lead to the array.

5	3	2	1	0



6. The following dynamic programming array resulted from solving an instance of the **Longest Common Subsequence** problem using bottom-up dynamicprogramming. What were the two strings?(provide any two strings that could produce this array)

0	0	0	0
0	0	0	1
0	0	1	1
0	1	1	1

```
X = ABC
Y = CBA
```

Exercise 3 (Design)

A. The **Infinite 0-1 Knapsack** Problem is a variant of the 0-1 Knapsack problem studied in class, where it is allowed to take an item an infinite number of times (but taking a fraction of an item is not allowed).

Design a dynamic programming solution for this problem. Your task is only to complete the following recursive definition for **`KNAPSACK(i, j)`** which captures the optimal substructure of the problem, where **`KNAPSACK(i, j)`** is the optimal solution considering the items **`0`** to **`i`** and a knapsack of capacity **`j`**.

(**`w[i]`** = weight of item **`i`**, **`v[i]`** = value of item **`i`**, **`W`** = knapsack capacity, **`N`** = number of items)

$$\text{KNAPSACK(i, j)} = \begin{cases} 0 & \text{if (i = 0 or j = 0)} \\ \text{KNAPSACK(i - 1, j)} & \text{if (w[i] > W)} \\ \begin{array}{l}\text{max(KNAPSACK(i - 1, j),} \\ \text{v[i] + KNAPSACK(i - 1, j - w[i]),} \\ \text{v[i] + KNAPSACK(i, j - w[i]))}\end{array} & \text{otherwise} \end{cases}$$

B. Consider the 0-1 Knapsack problem studied in class. Assume that our goal is to maximize the ***total weight*** of items in the knapsack instead of the total value (profit), but the other constraints are the same.

Assume that **`OPT(i, j)`** is the maximum sum of weights possible for items that fit in a knapsack of capacity **`j`**, considering the first **`i`** items only. Provide a recursive definition for **`OPT(i, j)`** that captures the optimal substructure of the problem. (Do not forget the base cases!)

(**`w[i]`** = weight of item **`i`**, **`v[i]`** = value of item **`i`**, **`W`** = knapsack capacity, **`N`** = number of items)

```
           ⎧ 0                              if (i = 0 or j = 0)
           ⎪
OPT(i, j) = ⎨ OPT(i-1, j)                    if (w[i] > W)
           ⎪
           ⎪ max(w[i] + OPT(i-1, j-w[i],   otherwise
           ⎩            OPT(i-1, j))
```

C. Consider three strings, **X**, **Y**, and **Z**. Assume that **`LCS(i, j, k)`** is the length of the longest common subsequence between the first **`i`**, **`j`** and **`k`** characters of **X**, **Y** and **Z** respectively. Provide a recursive definition for **`LCS(i, j, k)`** that captures the optimal substructure of the problem. (Do not forget the base cases!)

```
              ⎧ 0                             if (i = 0 or j = 0 or k = 0)
              ⎪
              ⎪ 1 + LCS(i-1, j-1, k-1)        if (X[i] == Y[j] == Z[k])
LCS(i, j, k)= ⎨
              ⎪ max(LCS(i-1, j, k),           if (X[i] != Y[j] or
              ⎪     LCS(i, j-1, k),               X[i] != Z[k] or
              ⎩     LCS(i, j, k-1))               Y[j] != Z[k])
```

D. Consider the following recurrence equation capturing the optimal substructure for the **collecting apples** problem we studied in class (base cases ignored).

```
MAX_APPLES(i, j) = apples[i][j] + max(MAX_APPLES(i-1, j),
                                      MAX_APPLES(i, j-1))
```

Write a new recurrence equation capturing the optimal substructure of a new version of the problem. The new version has the following changes:

3. Your goal is to collect the **least** number of apples from the top-left corner to the bottom-right corner in the grid.

4. You have 3 types of moves: **`DOWN`**, **`RIGHT`**, **`2RIGHT`**.
   `2RIGHT` allows moving 2 cells to the right (jumping over the right cell without taking the apples).

(Do not forget the base cases)

E. Consider **`N`** houses next to each other on a road. House **`i`** is willing to pay charity **`c[i]`** provided that you did not collect charity from the house directly before it. What is the maximum amount of money that you can collect for your charity campaign?

Examples.

`c[] = {`**10**`, 2, `**3**`}`	maximum amount = **13** (pick 10 and 3)
`c[] = {`**2**`, 1, 1, `**5**`, 1}`	maximum amount = **7** (pick 1 and 5)
`c[] = {1, `**5**`, 1, `**1**`, 7, `**10**`}`	maximum amount = **16** (pick 5, 1 and 10)

Design a dynamic programming solution for the problem above, following the steps below.

4. Assume that **`OPT(i)`** is the optimal solution for the problem considering houses **`i`** to **`N-1`**. Provide a recursive definition for **`OPT(i)`** that captures the optimal substructure. Indicate also what the base case(s) is (are).

5. Show that your recursive definition leads to overlapping sub-problems.

6. Draw the table that can be used by the dynamic programming algorithm. Fill the table if the houses are willing to pay the following charities: **`c[] = {1, 2, 3, 4}`**.
   Clearly indicate where the final answer is in the table.

Exercise 3 (Implementation)

A. Consider a variant of the ***weighted activity selection*** problem, where taking more than **m** activities (out of the given **n** activities) is not allowed. Consider also the following recursive definition for the optimal solution **OPT(i, m)** considering the activities :

```
OPT(i, m)  =  0                                      if m <= 0 or i >= n
           =  MAX(OPT(i+1, m),
                  OPT(NEXT(i), m-1) + v[i])          otherwise
```

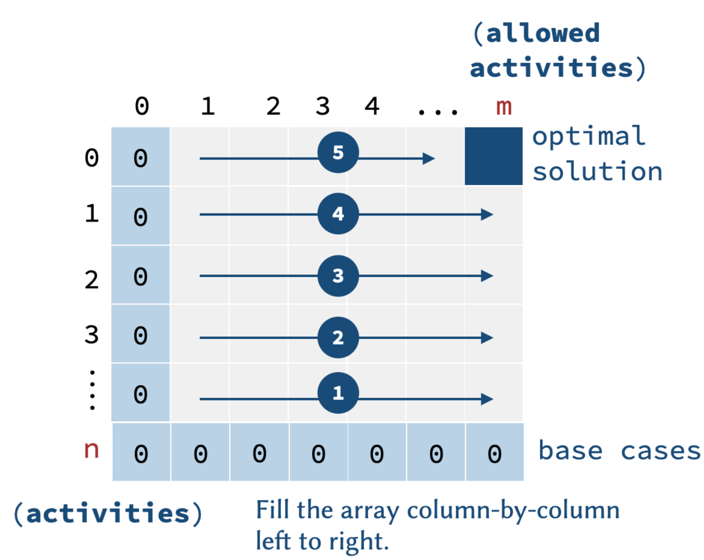
Convert the above recursive definition of **OPT(i, m)** to a bottom-up dynamic programming implementation. Assume the following:

- The activities are sorted according to their start times.
- The value of activity i is stored at **v[i]**, which is accessible globally.
- **NEXT(i)** returns the index **j** of the first activity compatible with **i**, where **j > i**.

```
OPT(i, m):
     CREATE result[n+1][m+1]
     Initialize result[n][] to 0
     Initialize result[][0] to 0

     FOR (i = n-1 TO 0):
          FOR (j = 1 to m):
               choice1 = result[i+1][j]
               choice2 = result[NEXT(i)][j-1] + v[i]
               result[i][j] = MAX(choice1, choice2)

     RETURN result[0][m]
```



Memoized Solution:

```
OPT():
      CREATE result[n][m]
      Initialize result[][] to -1
      OPT(0, m)
      RETURN result[0][m]

OPT(i, m):
      IF (m <= 0 OR i >= n)   RETURN 0
      IF (result[i][m] != -1) RETURN result[i][m]
      result[i][m] = MAX(OPT(i+1, m), OPT(NEXT(i), m-1) + v[i])

      RETURN result[i][m]
```

B. Consider the following definition of the Binomial Coefficient:

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \end{cases}$$

Write the pseudocode of the function **BinCoeff(n, k)**, which uses bottom-up dynamic programming to compute and return the binomial coefficient. Assume that **n** and **k** are guaranteed to be positive.

```
BinCoeff(n, k):
      CREATE result[n + 1][k + 1]
      FOR (i = 0 to n):
            FOR (j = 0 to k):
                  if(j > i):
                        break
                  else if(j == 0 || j == i):
                        result[i][j] = 1
                  else:
                        result[i][j] = result[i - 1][j - 1] + result[i - 1][j]
      return result[n][k]
```

C. The ***Edit Distance*** between two strings **X** (of size **N**) and **Y** (of size **M**) is defined as the minimum number of operations required to make **X = Y**. The operations that can be performed are: removing a character, inserting a character or substituting a character.

For example:

- The edit distance between **X=abc** and **Y=abb** is **1**, since we can substitute the **c** in **X** with a **b** to make **X=Y**.
- The edit distance between **X=ab** and **Y=abb** is **1**, since we can either add a **b** to **X** or remove a **b** from **Y** to make **X=Y**.

The following is a recursive algorithm that correctly finds the edit distance between two strings:

```
DIST(X, Y, N, M)
    IF (N == 0)         // if X is empty, we can make X=Y by
        RETURN M        // removing all of the characters in Y
    IF (M == 0)         // if Y is empty, we can make X=Y by
        RETURN N        // removing all of the characters in X

    // if the two characters are the same, no changes are needed.
    IF (X[N] == Y[M]) RETURN DIST(X, Y, N-1, Y-1)

    // if the characters are not the same an operation is needed.
    RETURN 1 + MIN( DIST(X, Y, N-1, M),     // Insert
                    DIST(X, Y, N, M-1),     // Remove
                    DIST(X, Y, N-1, M-1))   // Replace
```

Implement a bottom-up dynamic programming algorithm for the problem above.

D. Consider a new problem known as **Subset Sum**. Given an array **`data`**`[]` of **n** strictly positive integers, and an integer **k**, the goal is to return the number of subsets in the `data[]` array that sum to exactly **k**.

Consider the following recursive definition capturing the optimal solution of the problem.

```
OPT(k, i) = 1                                       IF k = 0
            0                                       IF k < 0 OR if i = n
            OPT(k-data[i], i+1) + OPT(k, i+1)       otherwise
```

Convert the above recursive definition of `OPT(k, i)` to a bottom-up dynamic programming implementation.

E. Solve the above implementation exercises again using *memoization* instead of bottom-up iteration.