SER 501: Adv Data Struct and Algorithms

Name : **Sai Swaroop Reddy Vennapusa**

Assignment 4(A)

Instructor : **Dr. Ajay Bansal**

Due Date : 1st Dec 2023, 11:59PM

Problem 1:
Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.
a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies.
b) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

Solution:

Part (a): Greedy Algorithm for Quarters, Dimes, Nickels, and Pennies

The greedy algorithm for making change with the least number of coins, given the denominations of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), works as follows:

1. Initialize: Start with 'n', the total amount of cents you need to make change for.

2. Quarters: Use as many quarters as possible. Subtract the value of these quarters from 'n'. That is, use 'q = floor(n / 25)' quarters and update 'n = n - 25 * q'.

3. Dimes: Repeat the process with dimes. Use 'd = floor(n / 10)' dimes and update 'n = n - 10 * d'.

4. Nickels: Next, use nickels. Use 'ni = floor(n / 5)' nickels and update 'n = n - 5 * ni'.

5. Pennies: Finally, use pennies for the remaining amount. The number of pennies needed will be equal to the updated value of 'n'.

The algorithm always chooses the largest coin denomination that is less than or equal to the remaining amount. This works optimally for the given set of coin denominations.

```
Algorithm MakeChange(n)
   Input: An integer n representing the total amount of cents
   Output: The number of quarters, dimes, nickels, and pennies to make the change

   Initialize coin counts:
      quarters = 0
      dimes = 0
      nickels = 0
      pennies = 0

   While n > 0
      If n >= 25
         quarters = floor(n / 25)
         n = n - (quarters * 25)
      Else If n >= 10
         dimes = floor(n / 10)
```

```
            n = n - (dimes * 10)
        Else If n >= 5
            nickels = floor(n / 5)
            n = n - (nickels * 5)
        Else
            pennies = n
            n = 0

    Return quarters, dimes, nickels, pennies
```

Part (b): Example 1 Where Greedy Algorithm Fails

A set of coin denominations where the greedy algorithm does not yield an optimal solution could be:

- 1 penny (1 cent)
- 1 nickel (5 cents)
- 1 non-standard coin, say 12 cents

In this case, let's consider making change for 15 cents:

- The Greedy Approach would choose one 12-cent coin and then three pennies, totaling four coins.
- The Optimal Solution is actually three 5-cent nickels, totaling three coins.

The greedy algorithm fails to find the optimal solution because it always picks the largest denomination first, without considering combinations that might use more coins of smaller denominations but result in fewer coins overall.

Example 2:

For the coin denominations of 1, 10, 20, and 25 cents

1. 65 Cents:
   - Greedy: 2 × 25 cents + 1 × 10 cents + 5 × 1 cent = 8 coins.
   - Optimal: 2 × 20 cents + 1 × 25 cents = 3 coins.

2. 80 Cents:
   - Greedy: 3 × 25 cents + 5 × 1 cent = 8 coins.
   - Optimal: 4 × 20 cents = 4 coins.

3. 90 Cents:
   - Greedy: 3 × 25 cents + 1 × 10 cents + 5 × 1 cent = 9 coins.
   - Optimal: 4 × 20 cents + 1 × 10 cents = 5 coins.

In this set of denominations, the greedy algorithm fails at 65, 80, and 90 cents. At these amounts, the greedy approach, which prioritizes the largest coin possible at each step, results in using more coins than the optimal solution.

Problem 2:
Given a rod of length n inches and an array of prices (p_i, for i = 1…n) of all pieces of size smaller than or equal to n, that is p_i is the price for a piece of length i inches. The Rod Cutting problem is to determine the maximum value obtainable by cutting up the rod and selling the pieces.
a) Describe a greedy algorithm to determine a set of piece sizes to cut up a rod to maximize the value when selling the pieces. For example, given p = [1, 3, 4, 5], the optimal solution would be [2, 2]. That is, a 4-inch rod is divided into two 2-inch pieces worth 3 each, for a total of 6.
b) Give a set of prices for which the greedy algorithm does not yield an optimal solution. Show the solution your algorithm yields along with an optimal solution.

Solution:
Part (a):
Pseudo code

```
function GreedyRodCutting(prices, rodLength):
    n = length(prices)
    totalValue = 0
    remainingLength = rodLength

    while remainingLength > 0:
        maxPricePerInch = 0
        maxPricePerInchIndex = 0

        for i in range(1, min(n, remainingLength) + 1):
            pricePerInch = prices[i-1] / i
            if pricePerInch > maxPricePerInch:
                maxPricePerInch = pricePerInch
                maxPricePerInchIndex = i

        if maxPricePerInchIndex == 0:
            break

        totalValue += prices[maxPricePerInchIndex - 1]
        remainingLength -= maxPricePerInchIndex

    return totalValue
```

Applying the above pseudo code for the given example:

p = [1, 3, 4, 5] with a rod length of 4 inches, let's look at the detailed steps:

- Calculate price per inch for each possible piece:
  - For length 1: Price = 1, Price per inch = 1/1 = 1.0.
  - For length 2: Price = 3, Price per inch = 3/2 = 1.5.
  - For length 3: Price = 4, Price per inch = 4/3 = 1.33.

  - For length 4: Price = 5, Price per inch = 5/4 = 1.25.
- The highest price per inch is 1.5 for the 2-inch piece.
- Cut a 2-inch piece, and the remaining rod length is 2 inches.

- Repeating the same calculation for the remaining 2 inches:
  - For length 1: Price = 1, Price per inch = 1.0.
  - For length 2: Price = 3, Price per inch = 1.5.
- Again, the highest price per inch is 1.5 for the 2-inch piece.
- Cut another 2-inch piece.
- Details of the cut: Length = 2 inches, Price = 3, Price per inch = 1.5.

- No remaining length.
- Total value obtained is 3 + 3 = 6.

Part(b):
p = [1, 5, 8, 9]

Greedy Algorithm Solution:

- First Iteration:
  - Price per inch for 1-inch piece = 1/1= 1.
  - Price per inch for 2-inch piece = 5/2= 2.5.
  - Price per inch for 3-inch piece = 8/3 =2.67.
  - Price per inch for 4-inch piece = 9/4= 2.25.
  - The algorithm chooses a 3-inch piece (price = 8).
  - Remaining length = 1 inch.

- Second Iteration (Remaining Length: 1 inch):
  - The only option is the 1-inch piece (price = 1).

- Greedy Solution: [3, 1], Total Value = 8 + 1 = 9.

Optimal Solution:

In contrast, the optimal solution would be to cut two 2-inch pieces:

- Two 2-inch pieces, each priced at 5, give a total value of 5 + 5 = 10.

- Optimal Solution: [2, 2], Total Value = 10.

Conclusion:

In this example, the greedy algorithm fails to yield the optimal solution. It selects cuts based on the highest price per inch at each step, leading to a total value of 9, whereas the optimal approach of cutting two 2-inch pieces results in a higher total value of 10.

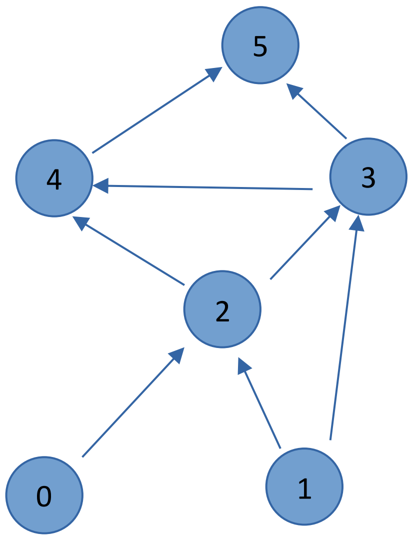
Problem 3:
The Fibonacci numbers are defined by recurrence F_0 = 0, F_1 = 1 and F_i = F_i-1 + F_i-2 . Give an O(n) time dynamic-programming algorithm to compute the nth Fibonacci number i.e. F_n . Draw the sub-problem graph. How many vertices and edges are in the graph?

Solution:

```
Algorithm: Fibonacci BottomUp(n)
Input: A non-negative integer n
Output: The nth number in the Fibonacci Sequence

if n == 0
   return 0
if n == 1
   return 1

Create an array A[0...n]
A[0] = 0
A[1] = 1

for i = 2 to n do
   A[i] = A[i-1] + A[i-2]

return A[n]
```

```
Algorithm : Fibonacci TopDown(n)
Input: A non-negative integer n
Output: The nth number in the Fibonacci Sequence

Create an array memo[0...n] and fill with -1 indicating uncomputed values

Function Fibonacci(n)
   if n == 0
      return 0
   if n == 1
      return 1
   if memo[n] != -1
      return memo[n]
   memo[n] = Fibonacci(n-1) + Fibonacci(n-2)
   return memo[n]
return Fibonacci(n)
```

Sub-problem graph:



Vertices: There are n+1 vertices in the graph, each representing the computation of a Fibonacci number (F_0 to F_n).

Edges: For bottom up approach, each vertex F_i for i >= 2 is connected with two edges from F_(i-1) and F_(i-2). The total number of edges is 2*(n-1).

For top-down approach, at each level n, the number of vertices is 2^n, the total number of vertices across all levels up to level k is the sum of a geometric series:

S_k = 2^0+2^1+2^2+⋯+2^k

The sum of this geometric series is given by:

S_k = 2^(k+1)-1

Now, considering the graph structure for the Fibonacci sequence, every node (except for the root) will have exactly one incoming edge. Therefore, the total number of edges is equal to the total number of vertices minus one (since the root vertex does not have an incoming edge). So, the total number of edges is approximately:

Edges = 2^(k+1)−1−1 = 2^(k+1)−2

Problem 4:
Give a dynamic-programming solution to the 0-1 knapsack problem that runs in O(nW) time where n is the number of items and W is the maximum weight of items that can be put in the knapsack.
Provide a sample run of your algorithm for 4 items (w_i , v_i) as (1,3), (2,4), (3,5), (4,8) where w_i is the weight of i-th item and v_i is the value of i-th item and also the maximum weight that can be put in the knapsack(W) is 6. Give the maximum value of items that can be put in the knapsack.

Solution:

```
for w = 0 to W
   B[0, w] = 0

for i = 0 to n
   B[i, 0] = 0

for i = 1 to n
   for w = 1 to W
      if w_i <= w
         if b_i + B[i-1, w-w_i] > B[i-1, w]
            B[i, w] = b_i + B[i-1, w-w_i]
         else
            B[i, w] = B[i-1, w]
      else
         B[i, w] = B[i-1, w]
```

w i	0	1	2	3	4
0	0	0	0	0	0
1	0	3	3	3	3
2	0	3	4	4	4
3	0	3	7	7	7
4	0	3	7	8	8
5	0	3	7	9	11
6	0	3	7	12	12

Problem 5:
You are climbing a staircase. It takes n (> 0) steps to reach to the top.
Each time you can either climb 1, 2 or 3 steps. In how many distinct ways can you climb to the top?
Example:
Number of stairs: 3
Number of ways: 4
Distinct ways to climb to the top: {(1,1,1), (1, 2), (2, 1), (3)}
Note: your algorithm only needs to return the number of distinct ways and not the actual ways to climb to the top.

Solution:

To get just the count:

```
Function CountWays(n: Integer): Integer
   if n == 0
      return 1
   if n == 1
      return 1
   if n == 2
      return 2

   ways = Array of size n + 1
   ways[0] = 1  // There is 1 way to stay at the bottom (i.e., do nothing)
   ways[1] = 1  // There is 1 way to reach the first step
   ways[2] = 2  // There are 2 ways to reach the second step: (1, 1) and (2)

   for i from 3 to n
      ways[i] = ways[i-1] + ways[i-2] + ways[i-3]

   return ways[n]
```

To get distinct ways as well along with the count:

```
Function CountAndListWays(n: Integer): Tuple[Integer, Set]
   if n == 0
      return (1, {()} )
   if n == 1
      return (1, {(1)} )
   if n == 2
      return (2, {(1, 1), (2)} )


   ways = Array of size n + 1
   distinctWays = Array of sets, each of size n + 1
   ways[0] = 1
   distinctWays[0] = {()}
   ways[1] = 1
```

```
    distinctWays[1] = {(1)}
    ways[2] = 2
    distinctWays[2] = {(1, 1), (2)}

    for i from 3 to n
        ways[i] = ways[i-1] + ways[i-2] + ways[i-3]
        distinctWays[i] = Set()

        for sequence in distinctWays[i-1]
            distinctWays[i].add(sequence + (1))

        for sequence in distinctWays[i-2]
            distinctWays[i].add(sequence + (2))

        for sequence in distinctWays[i-3]
            distinctWays[i].add(sequence + (3))

    return (ways[n], distinctWays[n])
```

Problem 6:
Describe an efficient algorithm that, given a set {x_1 , x_2 , …., x_n } of points on the real line, determines the size of the smallest set of unit-length closed intervals that contains all of the given points. Also give the time complexity of your algorithm.
For example: Given points {0.8, 4.3, 1.7, 5.4}, the size of the smallest set of unit-length closed intervals to cover them is 3.
Run your algorithm on the following set of points:
{0.8, 2.3, 3.1, 1.7, 3.6, 4.0, 4.2, 5.5, 5.2, 1.0, 3.9, 4.7} and determine the size of the smallest set of unit-length closed intervals that contains all of the given points.

Solution:

```
Function FindMinimumIntervals(points: Set of Real): Integer
    Sort(points)

    intervalCount = 0
    currentIntervalEnd = -Infinity

    for each point in points
        if point > currentIntervalEnd
            intervalCount = intervalCount + 1
            currentIntervalEnd = point + 1

    return intervalCount
```

Example:

Set of points = {0.8, 2.3, 3.1, 1.7, 3.6, 4.0, 4.2, 5.5, 5.2, 1.0, 3.9, 4.7}

1. Sort the Points:
   After sorting, we have:
   '{0.8, 1.0, 1.7, 2.3, 3.1, 3.6, 3.9, 4.0, 4.2, 4.7, 5.2, 5.5}'

2. Initialize Variables:
   'interval_count = 0'
   'current_interval_end = -Infinity'

3. Iterate and Create New Intervals:
   - Start with the first point '0.8'. Since it's not covered, we create a new interval '[0.8, 1.8]'. 'interval_count = 1'.
   - Point '1.0' is within '[0.8, 1.8]'.
   - Point '1.7' is within '[0.8, 1.8]'.
   - Point '2.3' is not within '[0.8, 1.8]', so we start a new interval '[2.3, 3.3]'. 'interval_count = 2'.
   - Point '3.1' is within '[2.3, 3.3]'.
   - Point '3.6' is not within '[2.3, 3.3]', so we start a new interval '[3.6, 4.6]'. 'interval_count = 3'.
   - Point '3.9' is within '[3.6, 4.6]'.
   - Point '4.0' is within '[3.6, 4.6]'.
   - Point '4.2' is within '[3.6, 4.6]'.
   - Point '4.7' is not within '[3.6, 4.6]', so we start a new interval '[4.7, 5.7]'. 'interval_count = 4'.
   - Point '5.2' is within '[4.7, 5.7]'.
   - Point '5.5' is within '[4.7, 5.7]'.

4. Final Count:
   The final count of intervals needed to cover all points is '4'.

5. Time Complexity:
   - The sorting step takes 'O(n log n)' time.
   - The iteration and interval creation is 'O(n)' since each point is considered once.
   - Thus, the overall time complexity is 'O(n log n)' due to the sorting step being the most significant factor.