Java Algorithms

Dynamic Programming

1. **Fibonacci**
    a. Original solution:

```
int count(int x) {
    if (x == 0) {
        return 0;
    } else if (x == 1) {
        return 1;
    }

    return count(x-1) + count(x-2);
}
```

    b. The original solution has runtime of **$O(2^N)$**, an **inefficient** algorithm.
    c. Dynamic solution:

```
int dpCount(int x) {

    int[] fArray = new int[x+1];
    fArray[0] = 0;
    fArray[1] = 1;

    for (int i = 2; i <= x; i++) {
        fArray[i] = fArray[i-1] + fArray[i-2];
    }

    return fArray[x];
}
```

        i. DP: no recursion in algorithm.
        ii. Other method: memoization.
2. **Longest Increasing Subsequence (LIS)**
    a. Input: **n** numbers **$a_1,a_2,...,a_n$**
    b. Goal: find the **length** of LIS in **$a_1,a_2,...,a_n$**
    c. Difference of substring and subsequence
        i. Substring: set of consecutive elements
        ii. Subsequence: subset of elements in order (can skip)
    d. Dynamic programming logic

i. For each element of the array, find possible array from previous elements to append.
ii. The arrays to be appended need to end in the number less than the current number.
iii. The dynamic solution has runtime **$O(N^2)$**

e. Code:

```
int dpFindLongestLength(int[] array) {
    int max = 1;
    int[] c = new int[array.length];

    // O(N)
    for (int i = 0; i < array.length; i++) {
        c[i] = 1;
        // O(N)
        for (int j = 0; j < i; j ++) {
            if (array[j] < array[i] && c[i] < c[j] + 1)
            {
                c[i] = 1+c[j];
            }
        }
    }

    // O(N)
    for (int i = 0; i < c.length; i++) {
        max = Math.max(c[i], max);
    }

    return max;
}
```

3. **Longest Common Subsequence (LCS)**
   a. Input: **2** strings **$X = x_1, x_2,...,x_n$** and **$Y = y_1,y_2,...,y_n$**
   b. Goal: find the length of longest string which is a subsequence of both **X** and **Y.**
   c. Subproblems
      i. Recursive
      ii. Conditions
         - If $X_n == Y_n$:

- If X_n != Y_n:

3. Code:

```
int lengthOfLCS(String s1, String s2, int idx1, int idx2) {
    if (idx1 < 0 || idx2 < 0) {
        return 0;
    }

    if (s1.charAt(idx1) == s2.charAt(idx2))
        return 1 + lengthOfLCS(s1, s2, idx1-1, idx2-1);
    else
        return Math.max(lengthOfLCS(s1, s2, idx1, idx2-1),
                lengthOfLCS(s1, s2, idx1-1, idx2));
}
```

4. **Knapsack problem**
   a. Input: **n** objects with weights $\mathbf{W_1,...,W_n}$ and values $\mathbf{V_1,...,V_n}$.
   b. Goal: find subset **S** of objects that have maximum values and can be fit in backpack with total weight **≤ B.**
   c. DP design
      i. An array **K** that defines the max value achievable using a subset of objects **1,...,i** with total weight <= **b** where 0<=**b**<=**B**.
      ii. Express **K(i)** in terms of **K(1),...,K(i-1)**.
      iii. At index i:
      If $\mathbf{W_i \leq b}$:
      K(i, b) = max { $\mathbf{v_i}$ + **K(i-1,b-**$\mathbf{w_i}$**)**, **K(i-1,b)**}
      Else:
      K(i, b) = **K(i-1,b)**
      Base cases:
      K(0, b) = 0, **K(i, 0)** = 0
   d. Code:

```
int knapSack(int[] wt, int[] val, int idx, int w) {
    // Base case
    if (w == 0 || idx == wt.length) {
        return 0;
    }

    if (wt[idx] <= w)
        return Math.max(val[idx] + knapSack(wt, val, idx+1,
```

```
                    w-wt[idx]), knapSack(wt, val, idx+1, w));
        else
            return knapSack(wt, val, idx+1, w);
    }
```

e. Runtime
   i. The runtime complexity is **O(NW)** where **N** is the number of objects and **W** is the total weight allowed in the backpack.
   ii. The runtime is <u>not polynomial</u> in the input size since **W** is just a constant.

5. Knapsack problem with repetition of items allowed

a. Unlimited supply: can use an object as many times as you'd like.
b. The rest of definition is the same as previous version.
c. DP design
   i. An array **K** with the weight **b** where 0<=**b**<=**B**. **K(b)** is the max value attainable using weight <= **b**.
   ii. Recurrence:
   If $\mathbf{W_i \leq b}$:
   $\mathbf{K(b)} = \max \{ \mathbf{v_i} + \mathbf{K(b-w_i)}: 1 \leq i \leq n, w_i \leq b\}$
   Base cases:
   $\mathbf{K(0, b)} = 0$
d. Code:

```
int knapSackSol2(int[] wt, int[] val, int w) {
    if (w == 0) {
        return 0;
    }

    int[] maxVals = new int[w+1];
    for (int b = 0; b <= w; b++) {
        for (int i = 0; i < val.length; i++) {
            if (wt[i] <= b) {
                maxVals[b] = Math.max(maxVals[b],
                                      maxVals[b-wt[i]] + val[i]);
            }
        }
    }
    return maxVals[w];
}
```

6. **Chain Matrix Multiply**
    a. Input: **4** matrices **A**, **B**, **C**, **D**
    b. Goal: compute **A** x **B** x **C** x **D**
    c. Standard way:
        i. (((**A** x **B**) x **C**) x **D**)
        ii. (**A** x **B**) x (**C** x **D**)
        iii. (**A** x (**B** x **C**)) x **D**
        iv. **A** x (**B** x (**C** x **D**))
    a. Graphical view
        i. Represent as binary tree
    b. DP attempt
        i. Find the minimum cost for computing.
        ii. To compute **C(i)** = min cost for computing $\mathbf{A_1,...A_i}$.
        iii. Try to split **C(n)** into binary tree in order to minimize the computing cost. For example, a split point i where $\mathbf{1 \le i \le n}$, the calculation becomes $\mathbf{A_1,...,A_i}$ and $\mathbf{A_{i+1},...,A_n}$, which looks like a substring.
    c. DP design
        i. Use substrings.
        ii. For **i** and **j** where $\mathbf{1 \le i \le j \le n}$.
        iii. Let **C(i, j)** = min cost for computing $\mathbf{A_i \text{ x } A_{i+1} \text{ x } ... x A_j}$.
        iv. Recurrence for **C(i, j)**
            - **C(i, i)** = 0
            - **C(i, j)** = min { **AC(i, l)**+**C(l+1, j)**: $i \le l \le j-1$ }
            - Runtime: $\mathbf{O(N^3)}$
    d. Code:

```
int chainMatrixMultiply(int[] m, int i, int j) {
    if (i == j)
        return 0;

    int min = Integer.MAX_VALUE;
    for (int l = i; l < j; l ++) {
        int count = chainMatrixMultiply(m, i, l) +
                    chainMatrixMultiply(m, l+1, j) +
                    m[i-1]*m[l]*m[j];

        min = Math.min(min, count);
    }
    return min;
}
```

vi. Input:

```
int[] m = new int[]{1, 2, 3, 4, 3};
int answer = chainMatrixMultiply(m, 1, m.length-1);
```

7. **Balanced binary tree**
   a. Code:

```
boolean isBalancedBinaryTree(TreeNode root) {
   if (root == null)
       return true;

   int leftHeight = getTreeHeight(root.left);
   int rightHeight = getTreeHeight(root.right);

   return Math.abs(leftHeight - rightHeight) <= 1
           && isBalancedBinaryTree(root.left)
           && isBalancedBinaryTree(root.right);
}

int getTreeHeight(TreeNode tree) {
   if (tree == null)
       return 0;
   return 1 + Math.max(getTreeHeight(tree.left),
                       getTreeHeight(tree.right));
}
```

8. **Shortest path in graph**
   a. Dijkstra's algorithm: requires all edge weight ≥ 0.
   b. Input: Giver a graph **G** with edge weights which can be negative.
   c. Goals: Find the shortest path from **s** to every other vertices.
   d. DP design
      i. If there is no negative cycles, the shortest path **P** from **s** to **z** visits every vertex only once. |P| ≤ n-1 edges.
      ii. Let **D(i, z)** = length of shortest path from **s** to **z** using ≤ **i** edges.
      iii. Assume **y** is the vertex between s and z.
         - **D(i, z)** = min {D(i-1,z), min{ D(i-1, y) + w(y,z) }
   e. Floyd Warshall
      i. Find the negative cycles anywhere in the graph.
   f. Bellman-Ford
      i. Slower than Dijkstra's algorithms but allows negative edge weights.

ii. Pseudocode:

```
for all z ∈ V, D(0, z) = ∞ // Initial states
D(0,s) = 0 // Distance from source to source = 0
for i=1→n-1 // number of vertices allowed.
    for all edges z ∈ V:
        // Get the shortest distances.
        if D(i,z) > D(i-1, y) + w(y, z):
        then D(i, z) = D(i-1, y) + w(y, z)
// The above guarantee shortest distances if graph doesn't
// contain negative cycle.

// If there're still shorter path, it means there's nagative
// cycle.
for all edges z ∈ V:
    if D(i,z) != ∞ and D(i,z) > D(i-1, y) + w(y, z):
    then there is negative cycle.
```

9. Examples to try on:
    a. Balanced Binary Tree
    b. Palindromic Substrings