Programming: Fibonacci Numbers, Longest Increasing Subsequence, Knapsack Problem

General Dynamic Programming

C. Odhiambo[1]

[1]Institute of Mathematical Sciences
Strathmore University

28th April, 2025

Introduction

Programming theory

What is Theoretical Programming?
Fundamental problems help develop algorithmic thinking and problem-solving skills. Importance of Classic Problems:

- Appear in interviews, contests (e.g., ACM ICPC, LeetCode).
- Lay foundations for advanced topics (DP, greedy algorithms, recursion).

Key Concepts to be covered today:

- Fibonacci Numbers
- Longest Increasing Subsequence (LIS)
- Knapsack Problem

Fibonacci Numbers - Overview

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding numbers, starting with 0 and 1. In programming, it can be generated using either a recursive or iterative approach. Definition:

$$F(n) = F(n-1) + F(n-2), \text{with} F(0) = 0 \text{and} F(1) = 1$$

Sample Sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Real-Life Use Cases:

- Rabbit population modeling
- Financial algorithms
- Golden ratio approximation
- Nature (flower petals, pinecones)

Fibonacci Numbers - Types of approached

Recursive Approach

A recursive function is defined such that takes an integer n as input and returns the nth Fibonacci number. The base cases are F(0) = 0 and F(1) = 1. For larger values of n, the function recursively calls itself with n-1 and n-2 and returns the sum of the results.

Iterative/Loops Approach

An iterative approach uses a loop to calculate the Fibonacci numbers. It starts with the first two numbers, 0 and 1, and then calculates the subsequent numbers by adding the two previous numbers in the loop.

Efficacy

The recursive approach has a time complexity of $O(2^n)$, which is highly inefficient for larger values of n. The iterative approach has a time complexity of $O(n)$, making it significantly more efficient for large n.

Implementation Using a For Loop

List what the code must contain or do before programming it:

- Two variables to hold the previous two Fibonacci numbers
- A for loop that runs 18 times
- Create new Fibonacci numbers by adding the two previous ones
- Print the new Fibonacci number
- Update the variables that hold the previous two fibonacci numbers

Example:

```
prev2 = 0
prev1 = 1
print(prev2)
print(prev1)
for fibo in range(18):
newFibo = prev1 +prev2
print(newFibo)
prev2 = prev1
prev1 = newFibo
```

Implementation Using Recursion

Recursion is when a function calls itself.

- To implement the Fibonacci algorithm we need most of the same things as in the code example above, but we need to replace the for loop with recursion.
- To replace the for loop with recursion, we need to encapsulate much of the code in a function, and we need the function to call itself to create a new Fibonacci number as long as the produced number of Fibonacci numbers is below, or equal to, 19.

Implementation Using Recursion

```
print(0)
print(1)
count=2
def fibonacci(prev1, prev2):
global count
if count <=19:
newFibo = prev1+prev2
print(newFibo)
prev2 = prev1
prev1 = newFibo
count+ = 1
fibonacci(prev1, prev2)
else:
return
```

Longest Increasing Subsequence (LIS)

Given an array arr[] of size n, the task is to find the length of the Longest Increasing Subsequence (LIS) i.e., the longest possible subsequence in which the elements of the subsequence are sorted in increasing order.
Examples:
Input: arr[] = [3, 10, 2, 1, 20]
Output: 3
Explanation: The longest increasing subsequence is 3, 10, 20
Input: arr[] = [30, 20, 10]
Output:1
Explanation: The longest increasing subsequences are [30], [20] and [10]
Input: arr[] = [2, 2, 2]
Output: 1
Explanation: We consider only strictly increasing.
Input: arr[] = [10, 20, 35, 80]
Output: 4
Explanation: The whole array is sorted

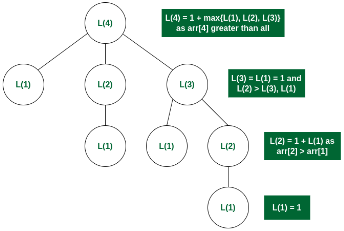
LIS Approach (Naive)- Using Recursion-Exponential Time and Linear Space

- The idea to do traverse the input array from left to right and find length of the Longest Increasing Subsequence (LIS) ending with every element arr[i].
- Let the length found for arr[i] be L[i]. At the end we return maximum of all L[i] values. Now to compute L[i], we use recursion, we consider all smaller elements on left of arr[i], recursively compute LIS value for all the smaller elements on left, take the maximum of all and add 1 to it. If there is no smaller element on left of an element, we return 1.
- Let L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS. Then, L(i) can be recursively written as:
- L(i) = 1 + max(L(prev)) where 0<prev < i and arr[prev]< arr[i]; or L(i) = 1, if no such prev exists.

  Formally, the length of LIS ending at index i, is 1 greater than the maximum of lengths of all LIS ending at some index prev such that arr[prev]< arr[i] where prev <i.

LIS Approach (Naive)- Using Recursion-Exponential Time and Linear Space

- After we fill the L array, we find LIS as maximum of all in L[]
- Overall LIS=max(L[i]) where $0 <= i < n$
- We can see that the above recurrence relation follows the optimal substructure property. Follow the below illustration to see overlapping subproblems.
- Consider arr[] = [3, 10, 2, 11]
- L(i): Denotes LIS of subarray ending at position 'i'


L(4)
L(4) = 1 + max{L(1), L(2), L(3)} as arr[4] greater than all
L(1)
L(2)
L(3)
L(3) = L(1) = 1 and L(2) > L(3), L(1)
L(1)
L(1)
L(2)
L(2) = 1 + L(1) as arr[2] > arr[1]
L(1)
L(1) = 1

LIS Approach-Using DP (Bottom Up Tabulation)-$O(n^2)$ Time and O(n) Space

- The tabulation approach for finding the Longest Increasing Subsequence (LIS) solves the problem iteratively in a bottom-up manner. The idea is to maintain a 1D array lis[], where lis[i] stores the length of the longest increasing subsequence that ends at index i. Initially, each element in lis[] is set to 1, as the smallest possible subsequence for any element is the element itself.
- The algorithm then iterates over each element of the array. For each element arr[i], it checks all previous elements arr[0] to arr[i-1]. If arr[i] is greater than arr[prev] (ensuring the subsequence is increasing), it updates lis[i] to the maximum of its current value or lis[prev] + 1, indicating that we can extend the subsequence ending at arr[prev] by including arr[i].
- Finally, the length of the longest increasing subsequence is the maximum value in the lis[] array.

LIS Approach- Using Binary Search-O(n Log n) Time and O(n) Space

- We can solve this in **O(n Log n)** time using Binary Search.
- The idea is to traverse the given sequence and maintain a separate list of sorted subsequence so far.
- For every new element, find its position in the sorted subsequence using Binary Search.

LIS Approach- Real Life Applications

- Stock Market Analysis: Identifying the longest sequence of increasing stock prices to find optimal buy-and-hold periods.
- Version Control Systems (like Git): Used in computing diffs or minimizing the number of edits between file versions.
- DNA/Protein Sequence Alignment: Helps in bioinformatics to find common patterns in genetic sequences (e.g., conserved gene regions).
- Data Compression: Run-length encoding and delta encoding often rely on detecting monotonic subsequences.
- Speech Recognition and Signal Processing: LIS helps match phoneme sequences or filter out noise by focusing on consistent signal increases.
- Computer Vision – Object Tracking: Used to identify consistent movement patterns of objects across frames in video.
- Human Resource or Academic Progression Analysis: Tracking employee performance scores or GPA progression to identify growth trends over time.

Knapsack Problem

- Given n items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.
- Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].
- Example Input: W = 4, profit[] = [1, 2, 3], weight[] = [4, 5, 1] Output: 3
- Explanation: There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.
- Input: W = 3, profit[] = [1, 2, 3], weight[] = [4, 5, 6] Output: 0

Knapsack Problem-(Naive Approach) Using Recursion $O(2^n)$ Time and O(n) Space

- A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W.
- Optimal Substructure: Consider this
  Case 1: The item is included in the optimal subset.
  Case 2: The item is not included in the optimal set.
- Follow the below steps to solve the problem:
  The maximum value obtained from 'n' items is the max of the following two values.
  - Case 1 (pick the nth item): Value of the nth item + maximum value obtained by remaining (n-1) items and remaining weight i.e. (W-weight of the nth item).
  - Case 2 (don't pick the nth item): Maximum value obtained by (n-1) items and W weight.
- If the weight of the 'nth' item is greater than 'W', then the nth item cannot be included and Case 2 is the only possibility.

Knapsack Problem-(Better Approach) Using Bottom-Up DP (Tabulation) – O(n x W) Time and Space

- There are two parameters that change in the recursive solution and these parameters go from 0 to n and 0 to W. So we create a 2D dp[][] array of size (n+1) x (W+1), such that dp[i][j] stores the maximum value we can get using i items such that the knapsack capacity is j.
  - We first fill the known entries when m is 0 or n is 0.
  - Then we fill the remaining entries using the recursive formula.
- For each item i and knapsack capacity j, we decide whether to pick the item or not.
  - If we don't pick the item: dp[i][j] remains same as the previous item, that is dp[i – 1][j].
  - If we pick the item: dp[i][j] is updated to val[i] + dp[i – 1][j – wt[i]].