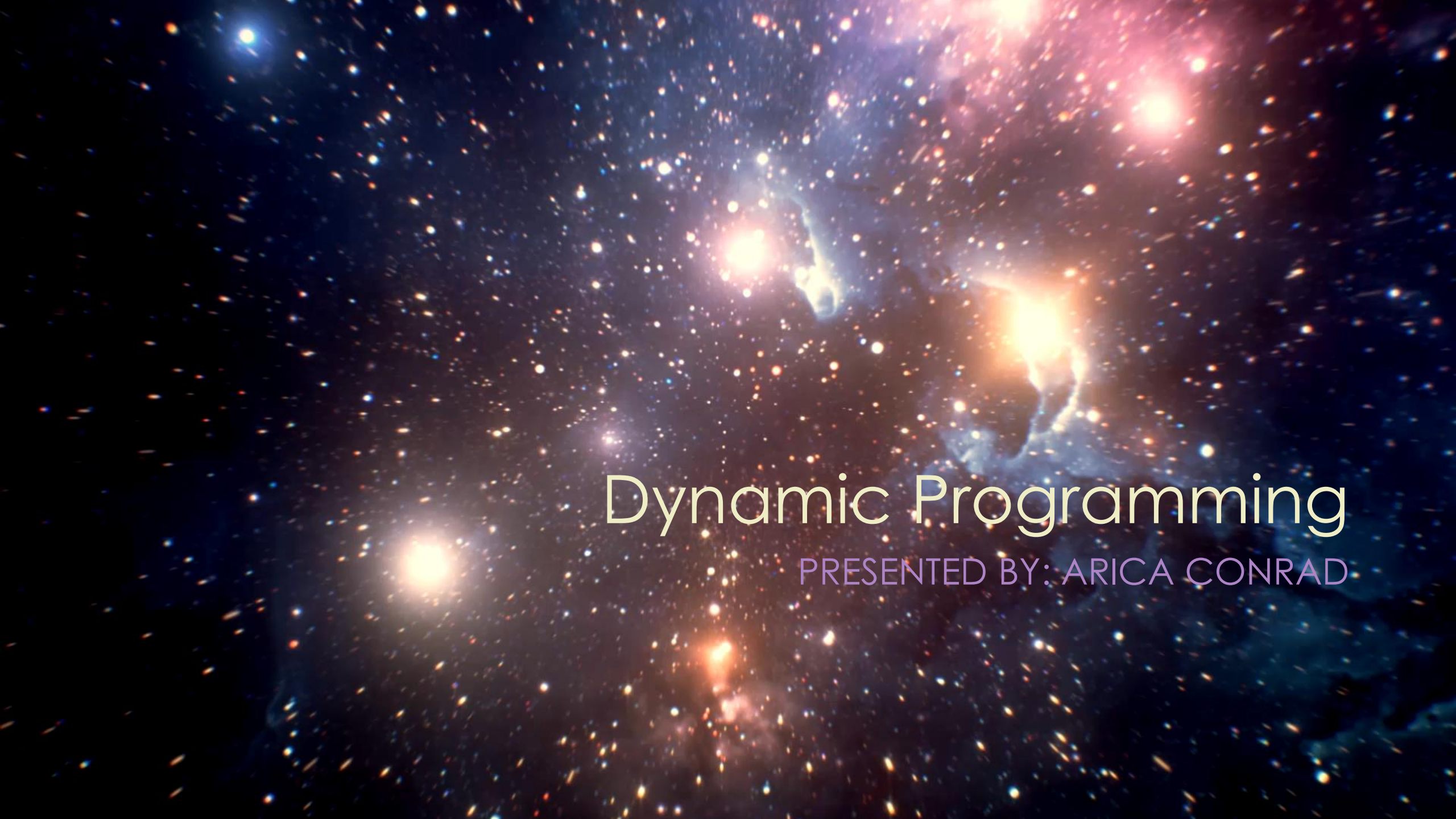


Dynamic Programming

PRESENTED BY: ARICA CONRAD

What is “Dynamic Programming”?

- Different than what we have seen so far in class, as it is not an algorithm!
- It is a way of optimizing your code over just using plain recursion
- The idea is to take a complex problem and break it down into simpler subproblems in a recursive manner
- You then store the results of those subproblems so you do not have to recompute them when they are needed later

Dynamic Programming (DP)

- Was developed by Richard Bellman in the 1950s
- Has found applications in numerous fields, from aerospace engineering to economics
  - For instance, flight control and robotics control
- DP can be tricky to master, though

Time and Space Complexities

- Since DP is not an algorithm, the time and space depends on which problem you are optimizing
  - I will mention specifically the time and space for each problem we go through
- Generally, though, using DP optimization can reduce time complexities from exponential to polynomial or linear

When to Use Dynamic Programming

- In order for a problem to be solved using DP, the problem must have two specific properties:
  - Overlapping Subproblems
  - Optimal Substructures

Overlapping Subproblems

- DP is mainly used when solutions of the same subproblems are needed again and again
- Solutions to subproblems are stored in a table so that they do not need to be recomputed later
- So, DP is not useful when there are no common ("overlapping") subproblems, because there is no point storing the solutions if they are not needed again
  - For instance, Binary Search does not have common subproblems

Optimal Substructures

- A given problem has an Optimal Substructure property if the problem can be solved using the solutions of the subproblems
- For instance, the Shortest Path problem in algorithms like Floyd-Warshall or Bellman-Ford follow the Optimal Substructure property
  - If a node x lies in the shortest path from a source node u to destination node v, then the shortest path from u to v is a combination of the shortest path from u to x and the shortest path from x to v
- This means the Longest Path problem does not have the Optimal Substructure property, as the longest path is not a combination of the longest simple paths between different nodes

Storing Subproblem Values

- There are two ways you can store values for a subproblem to be reused later
  - Memoization (Top Down)
  - Tabulation (Bottom Up)

Memoization (Top Down)

- The technique of storing and reusing previously computed results for subproblems so you do not need to recalculate them later
- You first check if the solution is already available
- If the solution is available, then use that solution
- Else calculate a new solution and store the solution for future use

Tabulation (Bottom Up)

- Similar to memoization, but the approach is different
- Tabulation focuses on filling the entries of the table until the target value has been reached
- We start solving the problem with the smallest possible inputs and store it for future use. Now as you calculate for the bigger values, use the stored solutions (the solutions for smaller subproblems)
- While DP problems are recursive in nature, a tabulation implementation is always iterative
- Complexity Bonus!
  - Since tabulation-based solutions fill values in a data structure using for loops, each value is typically computed in constant time (O(1))

Memoization vs. Tabulation

Memoization

- Code is easy and less complicated
- Slow due to a lot of recursive calls and return statements
- A memoized solution has the advantage of solving only the subproblems that are definitely required
- Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in the Memoized version (the table is filled on demand)

Tabulation

- Code gets complicated when a lot of conditions are required
- Fast, as we directly access previous solutions from the table
- If all subproblems must be solved at least once, a bottom-up DP approach usually outperforms a top-down memoized algorithm by a constant factor
- In Tabulated version, starting from the first entry, all entries in the table are filled one by one

Code Walkthroughs

- Fibonacci source: https://programming.guide/dynamic-programming-vs-memoization-vs-tabulation.html
- Knapsack source: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Example #1: Fibonacci Numbers

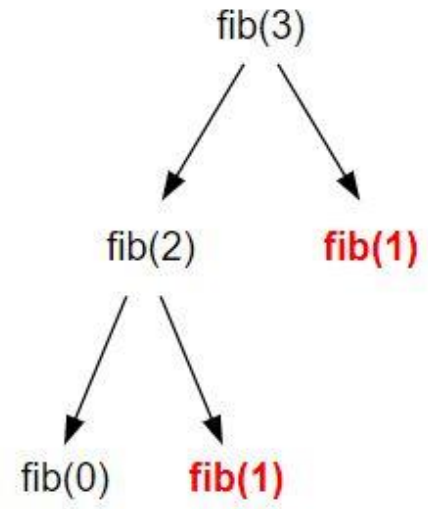
The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21
13+21=34
21+34=55
34+55=89
55+89=144
89+144=233
144+233=377

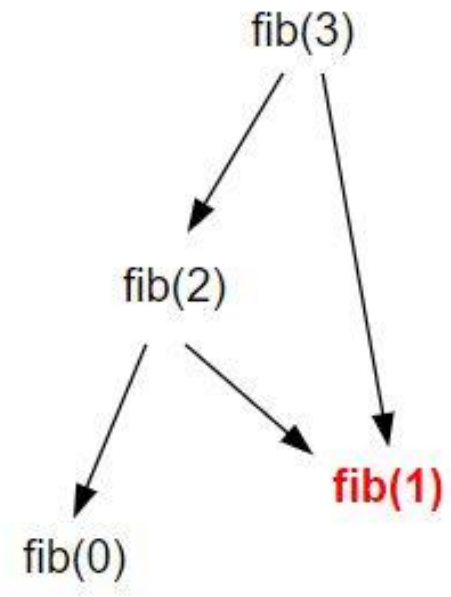
Basic Recursive Fibonacci Implementation

If you're computing for instance `fib(3)` (the third Fibonacci number), a naive implementation would compute `fib(1)` twice:



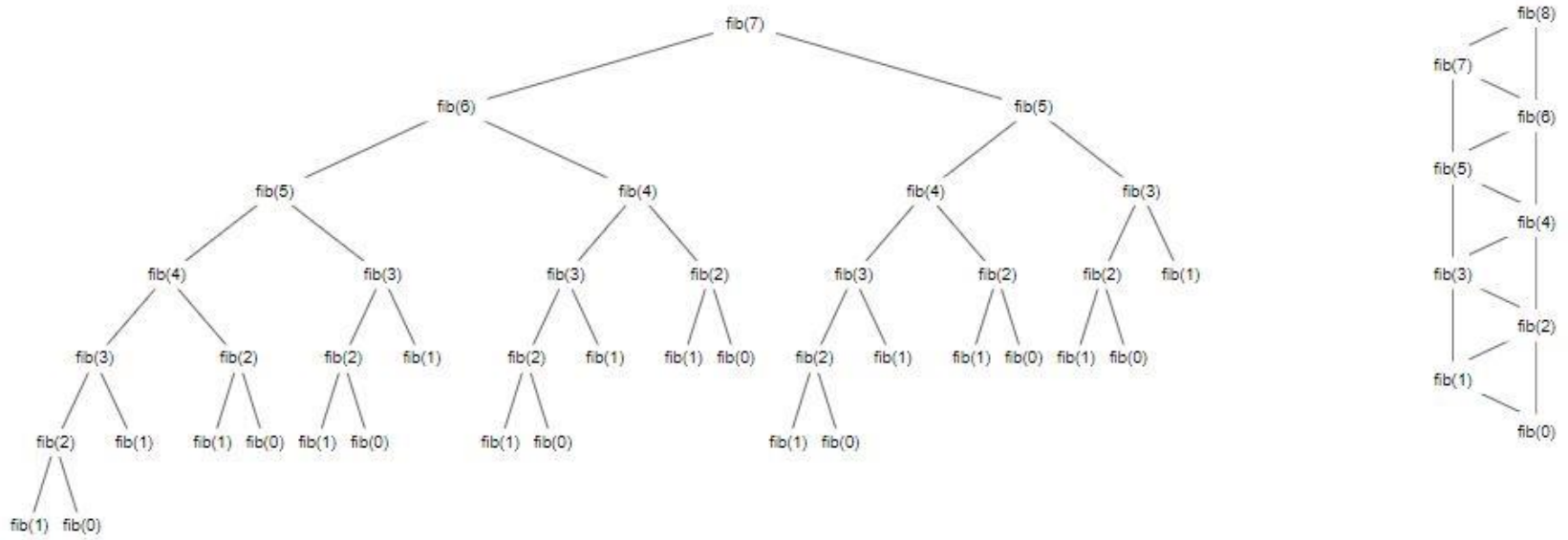
DP Implementation

With a more clever DP implementation, the tree could be collapsed into a graph (a DAG):



Why is DP better?

It doesn't look very impressive in this example, but it's in fact enough to bring down the complexity from $O(2^n)$ to $O(n)$. Here's a better illustration that compares the full call tree of `fib(7)` (left) to the corresponding collapsed DAG:



This improvement in complexity is achieved regardles of which DP technique (memoization or tabulation) is used.

Memoization Implementation

A memoized `fib` function would thus look like this:

```
fib_mem(n) {
    if (mem[n] is not set)
        if (n < 2) result = n
        else result = fib_mem(n-2) + fib_mem(n-1)
        mem[n] = result
    return mem[n]
}
```

Memoization Implementation

As you can see `fib_mem(k)` will only be computed **at most once** for each *k*. (Second time around it will return the memoized value.)

This is enough to cause the tree to collapse into a graph as shown in the figures above. For `fib_mem(4)` the calls would be made in the following order:

```
fib_mem(4)
    fib_mem(3)
        fib_mem(2)
            fib_mem(1)
            fib_mem(0)
        fib_mem(1)          already computed
    fib_mem(2)              already computed
```

This approach is **top-down** since the original problem, `fib_mem(4)`, is at the top in the above computation.

Tabulation Implementation

The tabulation version of `fib` looks like this:

```
fib_tab(n) {
    mem[0] = 0
    mem[1] = 1
    for i = 2...n
        mem[i] = mem[i-2] + mem[i-1]
    return mem[n]
}
```

Tabulation Implementation

The computation of `fib_tab(4)` can be described as follows:

```
mem[0] = 0
mem[1] = 1
mem[2] = mem[0] + mem[1]
mem[3] = mem[1] + mem[2]
mem[4] = mem[2] + mem[3]
```

As opposed to the memoization technique, this computation is **bottom-up** since original problem, `fib_tab(4)` , is at the bottom of the computation.

Fibonacci Numbers Implementations

Recursive

Running Time:

Exponential!

O(2^n)

Space:

O(N)

(Maybe O(1) depending on what implementation you use)

Memoization

Running Time:

O(N)

Space:

O(N)

Tabulation

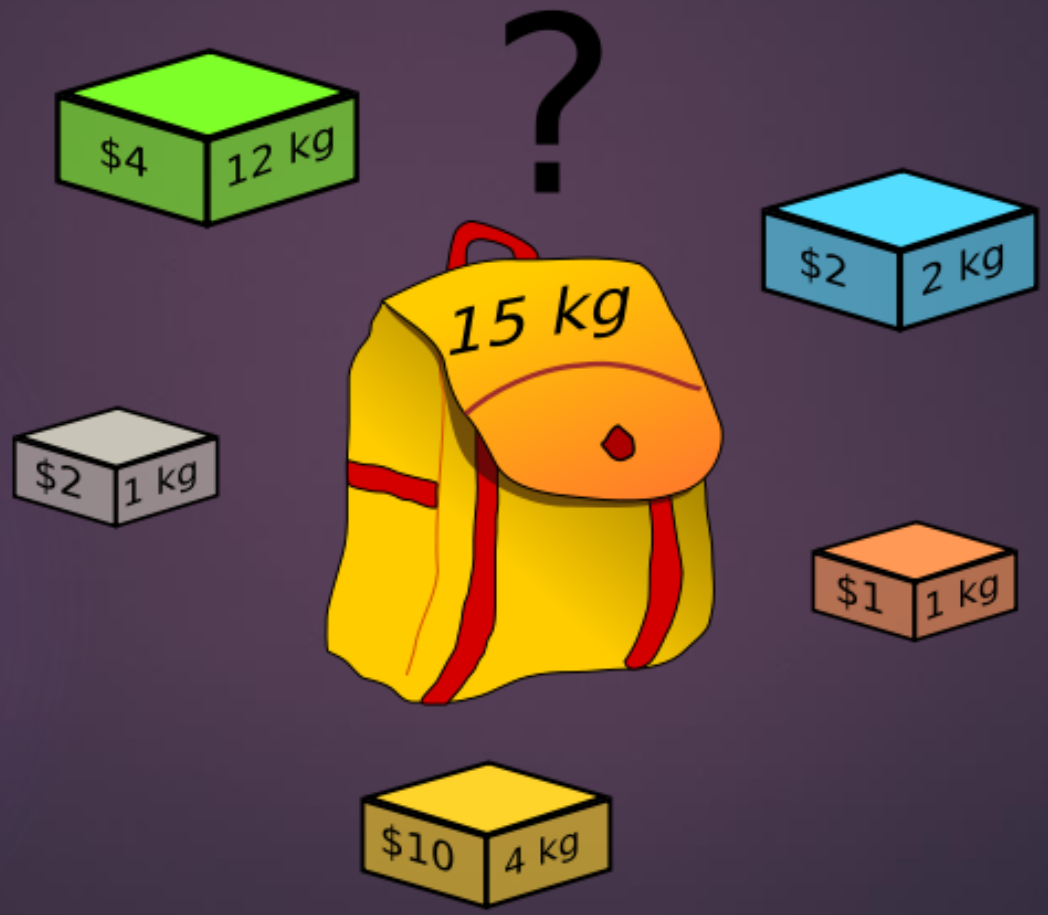
Running Time:

O(N)

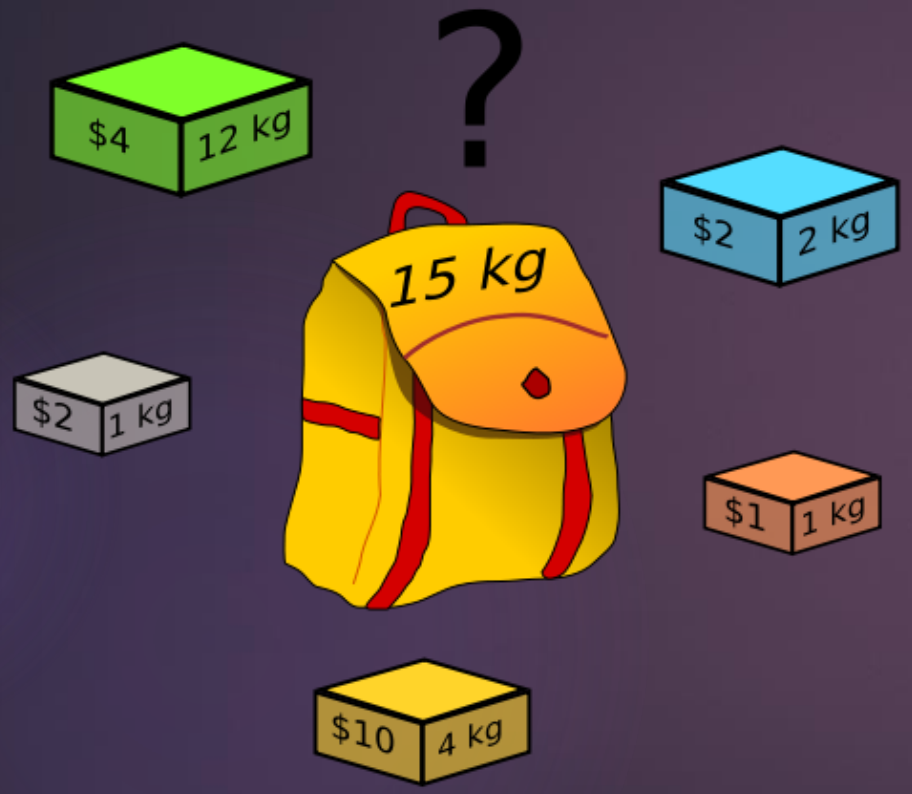
Space:

O(N)

Example #2: The Knapsack Problem (0-1)



What is the 0-1 Knapsack Problem?



- One of the most common DP problems
- You have a set amount of weight you can store in the knapsack
- You have items with varying weights and values you want to store in the knapsack
- What is the maximum value you can store in the knapsack?
- IMPORTANT: The "0-1" part means you cannot break an item!
  - You either pick the item or you don't
  - Different than the Fractional Knapsack Problem

Knapsack Problem Implementation

Recursive

- Time: O(2^n)
  - As there are redundant subproblems
- Space: O(1)
  - No extra data structure has been used to store values

Dynamic Programming

- Time: O(N*W)
  - Where ‘N’ is the number of items stored and ‘W’ is the capacity of the knapsack
- Space: O(N*W)
  - The use of a 2-D array of size ‘N*W’

Before We Finish…

Is There Anything Similar to Dynamic Programming?

- Yes! These are called algorithmic paradigms.

Algorithmic Paradigms

- Are general approaches to the construction of efficient solutions to problems
- In other words, a basic, commonly used approach in designing algorithms could be considered an algorithmic paradigm

Types of Algorithmic Paradigms

Divide and Conquer

- Divide a problem into small sub-instances of the same problem, solve these recursively, and then put the solutions together to be a solution of the given problem
- Examples: Merge Sort and Quick Sort

Greedy Algorithms

- Finds solutions by always making the choice that looks optimal at the moment – do not look ahead, never go back
- Examples: Prim's Minimum Spanning Tree and Kruskal's Minimum Spanning Tree

Dynamic Programming

- Recursively or iteratively solves subproblems and remembers the solutions to those subproblems
- Example: Floyd-Warshall Algorithm for the All Pairs Shortest Path problem

Final Thoughts

Conclusion

- Dynamic programming (DP) is a way of optimizing your code by breaking down a problem into smaller subproblems and remembering what you did
- A problem must have both optimal substructures and overlapping subproblems in order to be solved using DP
- You can store the solutions to subproblems using memoization (top down) or tabulation (bottom up)
- A useful skill to know, but tricky to master

References

- https://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/
- https://www.geeksforgeeks.org/dynamic-programming/
- https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/?ref=lbp
- https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/
- https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/
- https://www.geeksforgeeks.org/tabulation-vs-memoization/?ref=lbp
- https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
- https://www.javatpoint.com/dynamic-programming-introduction
- https://programming.guide/dynamic-programming-vs-memoization-vs-tabulation.html
- https://www.programminglogic.com/knapsack-problem-dynamic-programming-algorithm/
- https://softwareengineering.stackexchange.com/questions/168449/what-are-algorithmic-paradigms
- https://www.thecrazyprogrammer.com/2016/10/knapsack-problem-c-using-dynamic-programming.html
- https://www.tutorialspoint.com/data_structures_algorithms/dynamic_programming.htm
- https://en.wikipedia.org/wiki/Dynamic_programming

Photo References

- http://1.bp.blogspot.com/-RmAHD5rU3Gc/U6cjfkbPjHI/AAAAAAAAAOeg/9BYqLxmTSDQ/s1600/fibonacci+sequence.png
- https://programming.guide/dynamic-programming-vs-memoization-vs-tabulation.html
- https://www.thecrazyprogrammer.com/2016/10/knapsack-problem-c-using-dynamic-programming.html

Thank you for watching!