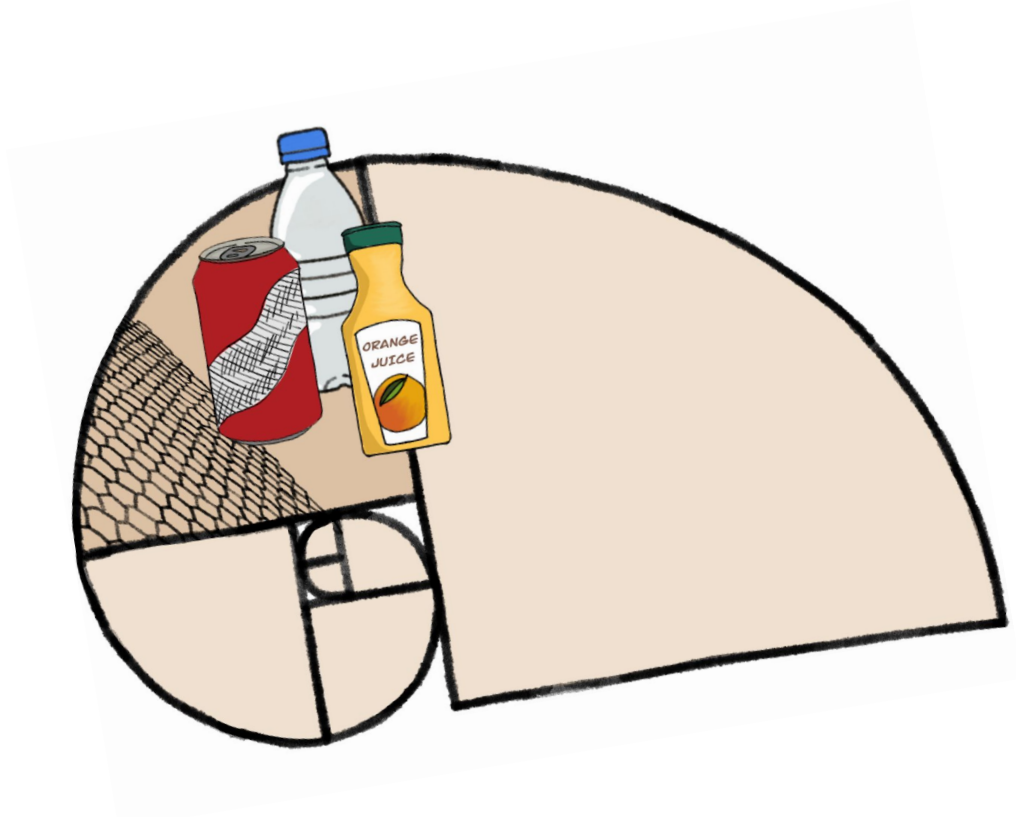# Dynamic Programming in Centermount Academy
Solving Drink Problem with Dynamic Programming

Day 2 Adventure: Centermount Academy

"You're going to get a traffic speeding ticket!!!" the zzz stand owner is shouting,
yy stand…for sure
x stand… certainly

[Todo: unwind recursion image]
[ToDo: knowledge on self-driving car inter-car communication, higher speed, closer distance, energy saving]

local world 1 time unit (second) = future world 1000 time unit (mili-second). Local world is 1000 times SLOWERRRRR!

"Welcome to the open house today!" the principle is greeting the guests at the entrance.

An open house day is an event held by school when its doors are open to the general public to allow people to look around their educational problems and learning environment in order to attract future students.

"To show our appreciation, each guest is granted some coupons to redeem at our school store."

"What good luck!" Banana Split hoorays, *"You can never be overdressed or overeducated. Let us get some school uniform accessories: bow ties."*
— Oscar Wilde

"A headband, a headhand!" Bubble Gum gets excited.

"And the most important, the map to the Gates!" Dark Knight reminds them.

Dashing to the store…

In the School Store

"I'm so thirsty!" Bubble Gum's eyes fix on the drink section in the store.
The store is offering 4 types of drinks served in bottles. Each guest can buy different types of drinks, but only one bottle per type due to the supply limitation.

Obviously, Bubble Gum likes some drinks over others. She wants to pick drinks with the highest satisfaction value.

Maximizing the satisfaction value is an optimization problem. So we can try Greedy approach.

Bubble Gum's Stomach Capacity: 5

	Water	Diet Water	Orange Juice	Gatorade
Satisfaction Value	3	1	4	5
Weight	2	1	2	4

Based on Banana Split Eating Nuts experience earlier today, we will use the same strategy. First sort the items by value per bottle.

Bubble Gum's Stomach Capacity: 5

	Gatorade	Orange Juice	Water	Diet Water
Satisfaction Value	5	4	3	1
Weight	4	2	2	1

First, pick the drink with the highest value: Gatorade.
Satisfaction value: 5
Remaining stomach capacity: 5 - 4 = 1

Now Bubble Gum still has 1 unit of empty space in her stomach. She can fill this space with 1 bottle of diet water.
Satisfaction value: 5 + 1 = 6
Remaining stomach capacity: 1 - 1 = 0

But wait! Seems like she can do better!

What if she drinks Orange Juice, Water and Diet Water? Hmm… seems like a *less-than-greedy* solution works better! **What goes wrong?**
Satisfaction value: 4 + 3 + 1 = 8!
Remaining stomach capacity: 5 - 2 - 2 - 1 = 0

"Hurry, our priority is to buy the map and remain focused on our mission!" Dark Knight urges everyone.

"Wait… Wait… Ha, I get it!" Banana Split's eyes light up, "the difference between the nuts and drink problems is that one allows fractional serving size and one not!"

Types of Knapsack Problems

Let us quickly review knapsack problem: There are different types of items (i) and each item i has its weight (wi) and value (vi). The bag has a capacity limitation (W). Our task is to take items within capacity to maximize the total value.

[ToDo: to decide whether to use W or w for the total capacity]

There are different types of knapsack problems:
- 0-1 knapsack problem: there is only one item of each type. So there are only two options for each item, either take it or leave it, e.g. 0 or 1 Gatorade bottle.
- Bounded knapsack problem: each item has more than one quantity, but has a quantity limitation, e.g. up to 3 chocolate bars
- Unbounded knapsack problem: there is no limitation on the quantity of the items, e.g. any number of water bottles when there is no supply constraint.
- Fractional knapsack problem: allow to take a fraction of any item, e.g. grain, powdered gold.

0-1 Knapsack problem

CalliLens shows the abstraction of our drink problem.
It is a 0-1 knapsack problem.

item(i) represent each item, and its value v(i) and weight w(i)
W capacity: 5

item(i)	1	2	3	4
v(i)	3	1	4	5
w(i)	2	1	2	4

Brute Force

Since Greedy is not working, we will try each drink combination and check which satisfies the problem requirement. This approach iis called **Brute Force**.
The total number of combinations is exponential. Clearly, this is not a good solution.

item(i)	1	2	3	4

pick or leave

pick or leave

pick or leave

pick or leave
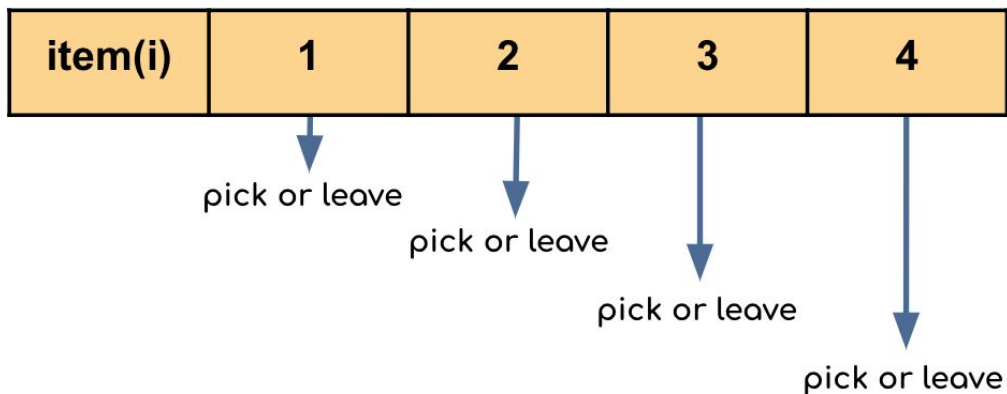
Total combinations = 2^n

2 x 2 x 2 x 2

Is there a better solution?

Dynamic Programming - Better Way to Solve 0-1 Knapsack Problem

Properties of the Problem

Optimal Substruture
Wearing the CalliLens, this monster problem is decomposed to some miniature problems.
From simplest problem, step by step, we work towards more complicated scenarios:
- increase capacity from low to high: 0, 1, 2, 3... to 5.
- Increase drink choices one at a time: no drink at all, water only, diet water, orange juice, etc.

Say that there is only the first drink, water, available. If its weight is greater than stomach capacity, we will have to leave it. Otherwise, we have two options: pick it or leave it.

Let's use formular f(i, w) to represent the optimal solution. "i" represents the first i number of items to consider. "w" represents the weight limit for the ith item (not the total stomach capacity W). For the first item, the solution is f(1, W).

Now let's increase the complexity a bit by adding another item, diet water. If the weight of this second item, item(2), is not within the capacity, we will have to leave it. Otherwise, we can either pick it or leave it. When picking it, we will decide for the optimal solution from item(1) using f(1, W-w2).

If we generalize the solution and consider item(i), we follow the same solution:
- If item(i)'s weight wi is greater than the remaining w capacity, we can not pick it. The optimal value will then be the same as the value up till item(i-1): f(i-1, w).
- Otherwise, we further have two options. We will choose the maximum of these two:
  - Pick item(i), we will add the current value vi to the optimal value for the first i-1 items. Since the current item(i) takes weight wi, the first i-1 items' max capacity will be w-wi. Optimal value will be f(i-1, w-wi) + vi
  - Leave item(i), we are left with the optimal value of the first i-1 items with w limit: f(i-1, w)

【ToDo， make sure to explain i-1 and w-w2 well]

item(i)	1	i	...	n

wi > w
can not pick

total value:
f(i-1, w)

wi <= w

pick

leave

value + v
solve f(i-1, w-wi)

value + 0
solve f(i-1, w)

total value:
f(i-1, w-wi) + vi

total value:
f(i-1, w)
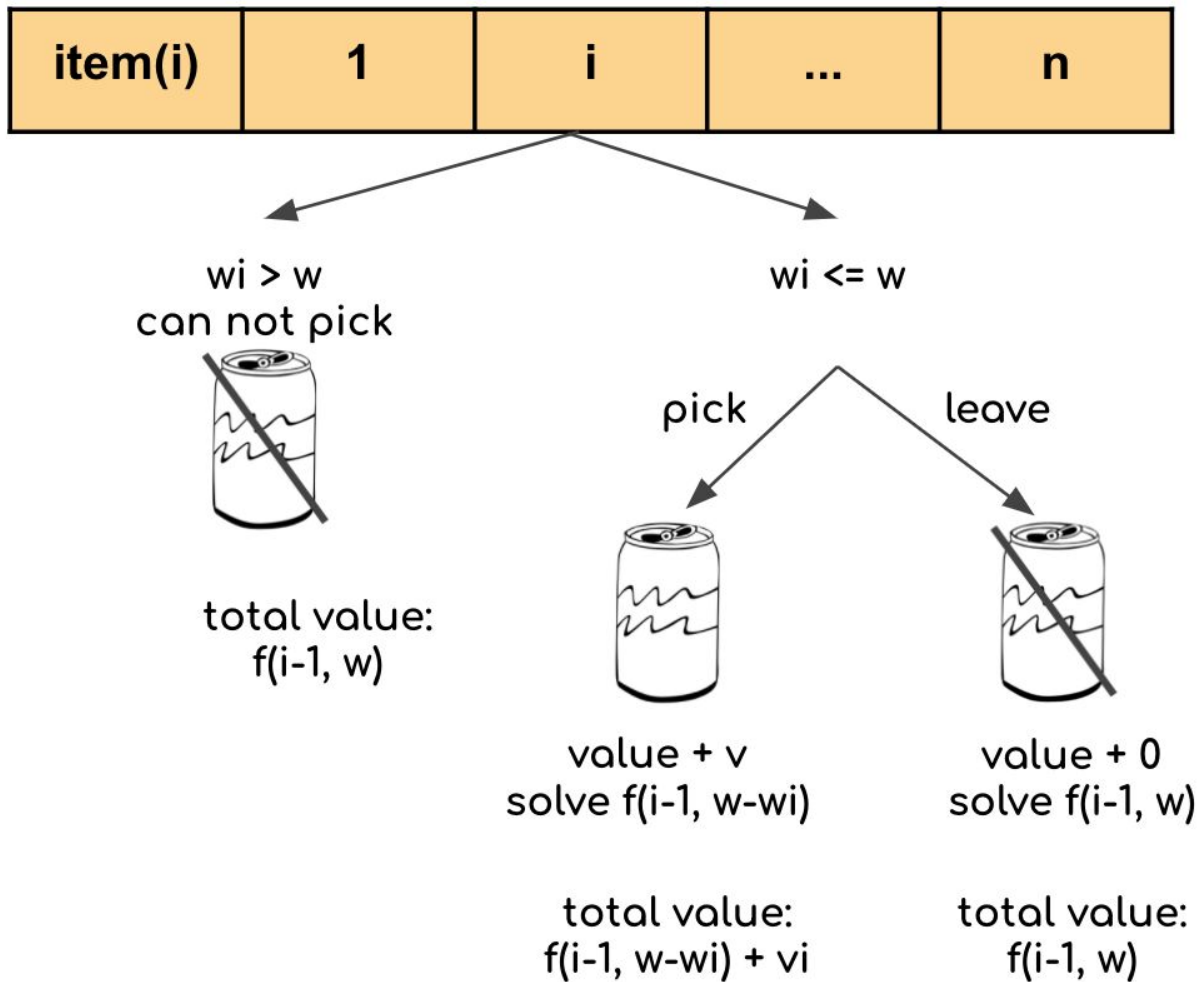
The formula looks like:

$$f(i, w) = \begin{cases} f(i-1, w) , & \text{if } wi > w \\ \max \{ f(i-1, w), f(i-1, w-wi)+vi \}, & \text{if } wi <= w \end{cases}$$

Clearly, **the solution to the subproblem is part of the solution to the overall problem. Same function used for subproblems, f , can be used to solve the overall problem.**

Overlapping Subproblems

On further look through the CalliLens, we see lots of overlapping subproblems.

capacity: 5

item(i)	1	2	3
vi	10	30	20
wi	1	1	1

optimal value (first 3 items, capacity 5)

$f(3,5)$

leave item(i)   pick item(i)

first 2 items, same weight limit $f(2,5)$ $f(2,4)$ weight limit = capacity - weight3

$f(1,5)$ $f(1,4)$ $f(1,4)$ $f(1,3)$

$f(0,5)$ $f(0,4)$ $f(0,4)$ $f(0,3)$ $f(0,4)$ $f(0,3)$ $f(0,3)$ $f(0,2)$
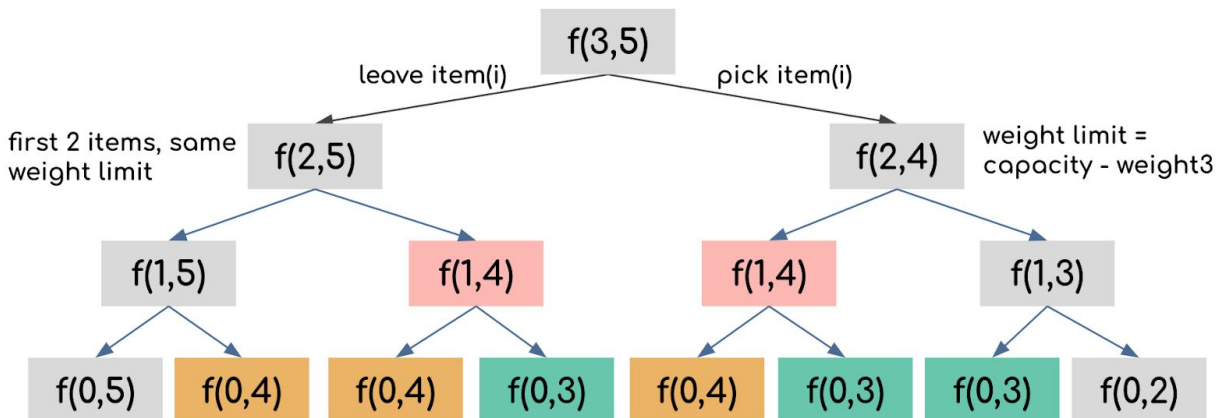
The overlapping subproblems in pink, brown and green occur multiple times while getting the final solution. To speed up the process, **we can store the results of subproblems to avoid computing the same subproblems again.**

The more complicated the problem is, **likely more overlapping subproblems are there.**

If a problem exhibits the above two properties, we can use **Dynamic Programming** approach to solve it!

Greedy and DP can be considered to solve optimization problems! If Greedy is not working, try with DP.

History:

Dynamic programming was invented by a person named Richard Bellman.

Richard Ernest Bellman[3] (August 26, 1920 – March 19, 1984) was an American applied mathematician, who introduced dynamic programming in 1953, and made important contributions in other fields of mathematics.

You may have heard of Bellman in the Bellman-Ford algorithm. So this is actually the precursor to **Bellman-Ford**. And we're going to see Bellman-Ford come up naturally in this setting. So here's a quote about him. It says, Bellman explained that he invented the name dynamic programming to hide the fact that he was doing mathematical research.

An entertaining and informative autobiography, *Eye of the Hurricane* (World Scientific Publishing Company, Singapore, 1984)
CHOICE OF THE NAME DYNAMIC PROGRAMMING "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. "An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he

felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

He was working at this place called RAND Corporation, and under a secretary of defense who had a pathological fear and hatred for the term research. So he settled on the term dynamic programming because it would be difficult to give a pejorative meaning to it. And because it was something not even a congressman could object to. Basically, it sounded cool.

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-19-dynamic-programming-i-fibonacci-shortest-paths/

https://en.wikipedia.org/wiki/Richard_E._Bellman

DP Key Concepts

Dynamic Programming approach solves a complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

Two Prerequisites

As we mentioned earlier, the problems solvable by DP have two properties:

1. This problem has optimal substructure.

The solution for the subproblem is part of the solution for the original problem.

2. This problem has overlapping subproblems

The total computation of this problem is not as big as the N^M combinations in brute-force algorithm because many subproblems are overlapping.

How to know there are overlapping subproblems?

We can draw the subproblems in graph. Each distinct subproblem is a node. If it is not a tree, it indicates overlapping subproblems.

States

The distinct subproblems are called the states. It is constructed by the required set of parameters that uniquely describe the subproblem. In our case $f(i, w)$, the state is the combination of i and w.

Transitions

The process of changing from one state (a.k.a. subproblem) to another is called transition.

Before transition to the next state, we want to make sure that the result of the current state is stored for later use. So $f(i, w)$ is stored.

Memoization

In DP we store the solution of these subproblems so that we do not have to solve them again, this is called **Memoization**. Please note, it is not "memorization"!

The term "memoization" was coined by Donald Michie in 1968 and is derived from the Latin word "memorandum" ("to be remembered"), usually truncated as "memo" in American English, and thus carries the meaning of "**turning [the results of] a function into something to be remembered**".

The Memoization table (memo table) should have the dimensions corresponding to the problem states. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state. Our table for the drink problem should have two dimensions representing i and w respectively.

Two Approaches

Bottom-Up Approach With Tabular Table

This is actually the original form of DP known as the 'tabular method' (computation technique involving a table).

Let us create the memoization table to store the optimal values and increase the complexity gradually.

- Row (i): the items to consider, e.g. row 2 represents the first 2 items to consider
- Column (w): the weight capacity, e.g. column 3 represents the capacity 3
- Value in cell (i, w): optimal value considering the first i number of items given capacity w, f(i, w)
- Simplest case:
  - Row 0: no items to pick, all optimal values are 0 irrespective to the capacity limit
  - Column 0: capacity 0, can not pick any item, all values are 0

capacity ⟶

item

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

when no items to pick, value is always 0 for any capacity
f(0, w)

optimal value for capacity 3 considering item 1 and 2
f(2, 3)

optimal value for max capacity 5 considering all items: 1, 2, 3 and 4
f(4, 5)

when capacity is 0, can not pick any item, so value is always 0
f(i, 0)
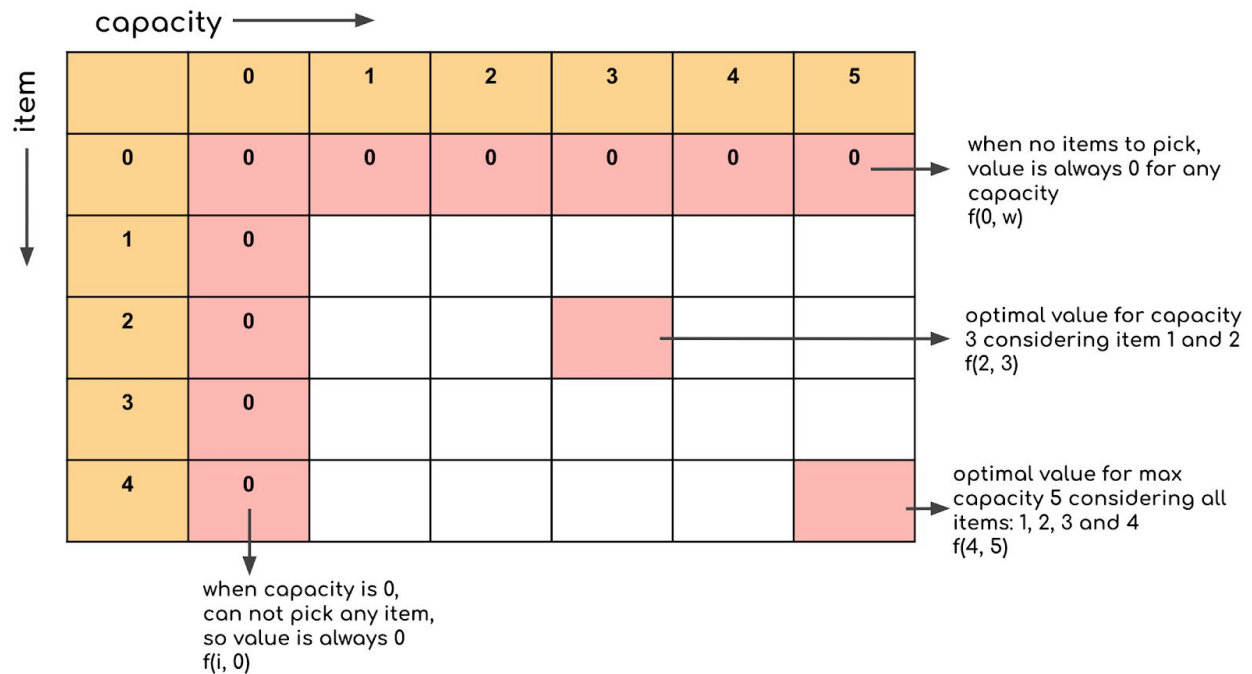
Gradually increase the complexity.

$$f(i, w) = \begin{cases} f(i-1, w), & \text{if } w_i > w \\ \max\{f(i-1, w), f(i-1, w-w_i)+v_i\}, & \text{if } w_i \leq w \end{cases}$$

Drink Types

	Water	Diet Water	Orange Juice	Gatorade
Satisfaction Value	3	1	4	5
Weight	2	1	2	4

Let us build row 1. item1 water, w1 = 2
w = 1, w1 > w, so f(i, w) = f(i-1, w)
f(1, 1) = f(0, 1) = 0

capacity ⟶

item

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				

w1>w, can not pick item(1)
optimal value same as above
f(i-1, w)

f(1, 2)
w=2, w1 <= w, so f(i, w) = max{ f(i-1, w), f(i-1, w-wi)+vi}
f(1,2) = max{f(0, 2), f(0, 0)+3} = max{0, 0+3} = 3

capacity ⟶

item

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3 0			

w1<=w, pick item(1)
optimal value: (above row
subtract w1) + v1
f(i-1, w-wi)+vi

w1<=w, leave item(1)
optimal value: same as above
f(i-1, w)

max of the two options

[meaning of each row, accumulation of the drink items]

Similarly, we get
f(1,3) = 3

f(1,4) = 3
f(1,5) = 3

capacity →	0	1	2	3	4	5
item ↓						
0	0	0	0	0	0	0
1	0	0	3	3 0	3	3

Let us build row 2 considering the first two types of drinks (introducing diet water).
diet water w2=1
w = 1, w2 <= w, so f(i, w) = max{ f(i-1, w), f(i-1, w-wi)+vi}
f(2, 1) = max{f(1, 1), f(1, 0)+1} = max{0, 1} = 1

Similarly, we get
f(2,3) = 4
f(2,4) = 4
f(2,5) = 4

capacity →	0	1	2	3	4	5
item ↓						
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	1 0	3	4	4	4

Similarly, we work on row 3 (introducing orange juice).

capacity →

item	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	1	3	4	4	4
3	0	1	4	5	7	8
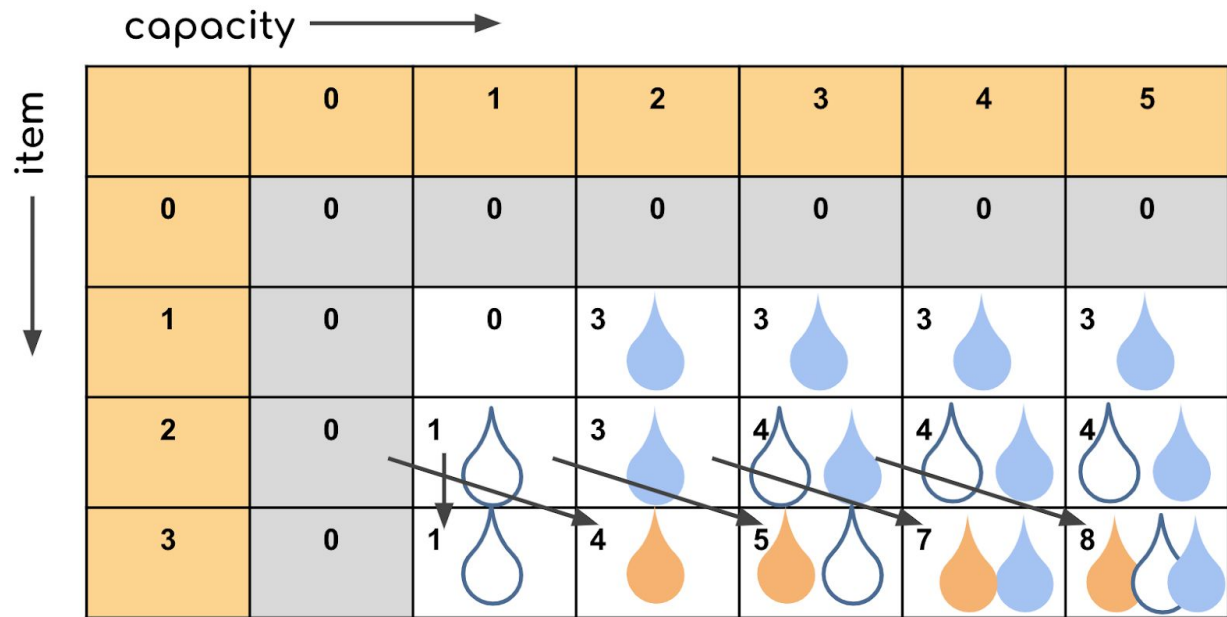
and row 4 (introducing Gatorade).

As we can see, the 4th item really does NOT contribute to the satisfaction value.

capacity →

item	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	1	3	4	4	4
3	0	1	4	5	7	8
4	0	1	4	5	7	8
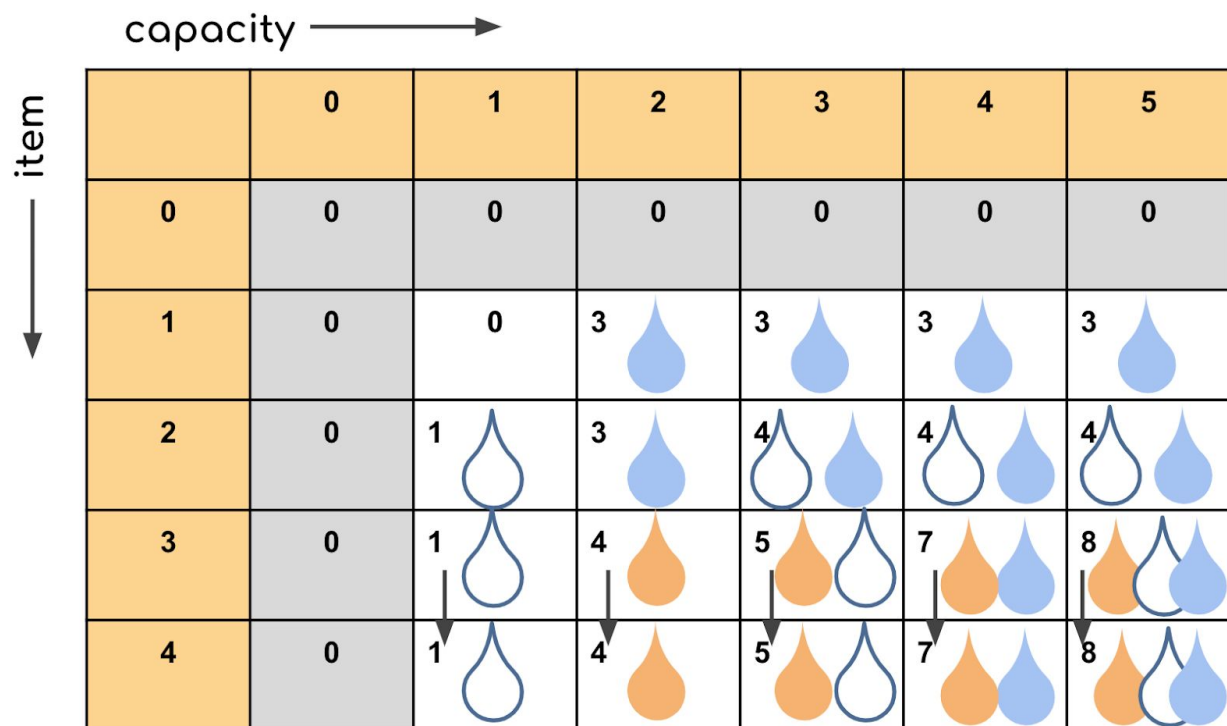
Every time when we include an additional drink type, we assess the value based on the above row without this drink type. This way we make sure to only include this drink type once.

Diagram Illustration

```
        ┌──────────┐      ╱ List of Items ╱  ╱ Capacity ╱
        │  Start   │─────▶╱ (Name, Value, ╱  ╱          ╱
        └──────────┘      ╱   Weight)     ╱  ╱          ╱
```

Initialize DP Table:
0 Capacity Column
0 Item Row

Loop Through Items

More Items? ──No──▶ *Output Max_Value* ──▶ **Stop**

Yes

Current item Weight <= Capacity

No ──▶ **Optimal Value: Same Capacity Without Current Item f(i-1, w)**

Yes

Pick Current item?

No ──▶ **Optimal Value: Same Capacity Without Current Item f(i-1, w)**

Yes ──▶ **Optimal Value: Current Item Value + Optimal Value of i-1 Items with Capacity - wi f(i-1, w-wi) +vi**

Max of the Two

```
 1 # Apple Pi Inc.
 2 # Algorithmic Thinking
 3 # 0-1 Knapsack Problem
 4 # DP Bottom Up Approach
 5
 6 class Item:
 7     def __init__(self, name: str, value: int, weight: int):
 8         self.name = name
 9         self.value = value
10         self.weight = weight
11
```

```
12 # return the maximum value of selected items within capacity
13 # argument items: choices
14 # argument capacity: knapsack capacity
15 def zero_one_knapsack(items, capacity):
16     # dp table
17     # dp[i][j] stores max value
18     # for capacity j (column) considering the first i (row) items
19     dp = []
20
21     # initialize max value to 0 for all capacities and drink types
22     total_item = len(items)
23     for i in range (total_item+1):
24         row = []
25         for j in range(capacity+1):
26             row.append(0)
27         dp.append(row)
28
29     # build dp table
30     # from simplest problem (0 capacity, no item),
31     # step by step, work towards more complicated scenarios.
32
33     # outer for loop to gradually increase item types
34     # innter for loop to gradually increase capacity
35     for i in range(1, total_item+1):
36         for j in range(capacity+1):
37             index = i-1
38             if (j < items[index].weight):
39                 # if current capacity < current item weight
40                 # then can not pick current item
41                 # optimal value same as the one without current item
42                 dp[i][j] = dp[i-1][j]
43             else:
44                 # pick current item
45                 value_with_current_item = dp[i-1][j-items[index].weight] + \
46                                           items[index].value
47
48                 # leave (do not pick) current item
49                 value_without_current_item = dp[i-1][j]
50
51                 # choose the higher value to store in dp table
52                 dp[i][j] = max(value_without_current_item, value_with_current_item)
53
54     return dp[total_item][capacity]
```

```
55
56 # initialize item list and capacity
57 items = [Item('Water', 3, 2),
58         Item('Diet Water', 1, 1),
59         Item('Orange Juice', 4, 2),
60         Item('Gatorade', 5, 4)]
61 capacity = 5
62
63 # call knapsack function
64 max_value = zero_one_knapsack(items, capacity)
65
66 # output result
67 print('The maximum value within capacity:', max_value)
```

Top-Down Approach With Recursion and Memo Table

It involves two key components:
- **Recursion** – Solve the sub-problems recursively
- **Memoization** – Store the solution of these sub-problems so that we do not have to solve them again

Workflow

1. Initialize a DP 'memo' table with dummy values that are not used in the problem, e.g. '-1'.

The purpose of this "memo" table is to "memoize" or store the solutions to the sub-problems in a table. When we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If so, we can use the result from the table directly, otherwise we solve the subproblem and add its solution to the table.

The memo table should have dimensions corresponding to the problem states. For our drink problem, the "memo" table should have two dimensions: one dimension representing available drink types, and another dimension representing weight limit.

2. At the start of the recursive function, check if this state has been computed before.

(a) If it has, simply return the value from the DP memo table.

(b) If it has not been computed, perform the computation as per normal (only once) and then store the computed value in the DP memo table so that further calls to this sub-problem (state) return immediately.

```
 1 # Apple Pi Inc.
 2 # Algorithmic Thinking
 3 # 0-1 Knapsack Problem
 4 # DP Top Down Approach
 5
 6 class Item:
 7     def __init__(self, name: str, value: int, weight: int):
 8         self.name = name
 9         self.value = value
10         self.weight = weight
11
```

```
12 # return the maximum value of given items list within weight_limit
13 # argument items: choices
14 # argument weight_limit: weight limit for each state
15 # recursive function
16 # top down approach
17 # return max value
18 def zero_one_knapsack(items, weight_limit):
19
20     # base cases
21     if (weight_limit < 0):
22         return 0
23
24     if len(items) == 0:
25         return 0
26
27     # if solution in memo table
28     # retrieve and use it
29     if (memo[len(items)-1][weight_limit] != -1):
30         return memo[len(items)-1][weight_limit]
31
```

```
32      # otherwise, compute and
33      # store the result in memo table
34      max_value = -1
35
36      # if current capacity < current item weight
37      # then can not pick current item
38      # optimal value same as the one without current item
39      if (weight_limit < items[-1].weight):
40          max_value = zero_one_knapsack(items[0:len(items)-1], weight_limit)
41
42      else:
43          # pick current item
44          value_with_current_item = \
45                          zero_one_knapsack(items[0:-1], \
46                          weight_limit - items[-1].weight) + \
47                          items[-1].value
48
49          # leave (do not pick) current item
50          value_without_current_item = zero_one_knapsack(items[0:-1], weight_limit)
51
52          # choose the higher value to store in dp table
53          max_value = max(value_without_current_item, value_with_current_item)
54
55      memo[len(items)-1][weight_limit] = max_value
56      return max_value
57
```

```
58 # one dimension of the problem state
59 # weight limit
60 capacity = 5
61
62 # the other dimension of the problem state
63 # max types of items
64 # initialize item list and capacity
65 items = [Item('Water', 3, 2),
66          Item('Diet Water', 1, 1),
67          Item('Orange Juice', 4, 2),
68          Item('Gatorade', 5, 4)]
69 max_items = len(items)
70
71 # initialize memo table to -1
72 memo = [[-1 for j in range(capacity+1)] for i in range(max_items)]
73
74 # alternative code to initialize memo table
75 # memo = []
76 # for i in range(0, max_items):
77 #     line = []
78 #     for j in range(0, capacity+1):
79 #         line.append(-1)
80 #     memo.append(line)
81
82 # call knapsack function
83 max_value = zero_one_knapsack(items, capacity)
84
85 # output result
86 print('The maximum value within capacity:', max_value)
```

Yay! We've solved the problem of getting the best juice in each scenario.
What are we going to do with the juice? Power up our **time tunnel**, of course!
We take the juice and run it through the batteries. The machine sputters… flickers a bit… and turns on!

Real World Connection
How can we use juice to make power and high-tech gadgets? Recently, scientists are researching a new and almost magical material called *graphene*.

Graphene is a very special material. It is extremely strong and conductive, yet flexible and transparent. This is amazing, since graphene sheets are only one atom thick. Graphene is over 100 times stronger than steel. Some people have even called graphene a "supermaterial"!

So what exactly is graphene? Well, it turns out that graphene is just a single layer of graphite — the same material used in pencil lead. By very carefully lifting away sheets from a piece of graphite, graphene is produced. Some graphene is probably hidden in the lead of your pencil!

The very first sample of graphene was produced in 2004 using a very simple technology: Scotch tape. Scientists repeatedly stuck a piece of tape to graphite and removed it until a single layer of graphene came off. This discovery won the Nobel Prize in Physics in 2010!

However, there is another way to produce graphene: by spraying sugar! By passing a sugary substance, like glucose or **juice**, and heating, scientists can produce almost pure sheets of graphene. That's how we make gadgets from juice!

Graphene has all sorts of applications, from flexible phone screens to solar panels. It's not hard to imagine phones using graphene's flexibility to create bendable or wearable gadgets. How do you think graphene will be used in the near future?

https://newatlas.com/making-graphene-from-table-sugar/16953/
https://www.digitaltrends.com/cool-tech/what-is-graphene/

Coin Change Problem with DP

With a satisfied stomach, Bubble Gum picks a headband, Banana Split picks a bow tie and Dark Knight picks the map with a total of $20. Ready to pay with store coupons with a variety of face values of $1, $5, $6.

| Total Value $15 | $1 | $5 | $6 |

The BestFour searches through the map and can not find any mention about the good invention and bad invention. "Would you please point us the locations of ...?" they seek help from the store owner.

The store owner looks around and says secretly, "If you can prove your intelligence by showing me the least number of coupons for $20, I will tell you. This is our top secret!"

Eyes lit up, with lightning speed and a grin, Dark Knight sorts the coupons and picks the higher face values by order: 3 $6 and 2 $1 worth coupons.

"Wait! Let us double check… well… 4 $5 coupons is clearly better. " Banana Split stops him right on time.

"But how come our greedy approach worked just fine when selling CalliLens in the market, but is no longer working now?!" Dark Knight puzzled.

Pause, can you spot the difference between the two problems?

"Well, please pay attention to the two sets of face values, " Banana Split hints to Dark Knight, "US paper bills have $1, $5, $10, $20, $50, and $100. Whereas the store coupons have $1, $5, and $6."

Which sequence of numbers works for Greedy and which does not?

[turn to the next page…]

Answer:

In cases where there is a bill whose value ($6), when added to the smallest bill ($1), is lower than twice that of the bill immediately less than it ($5), the Greedy approach will not work. In this case, $6 + $1 < 2 x $5. That's why!

"Wow, that's eye opening!" Dark Knight was really intrigued, "this is an optimization problem. If we can not go with Greedy, let us try with DP! Does this problem have the two DP properties?"

They look through CalliLens together...

[ToDo: change the following description to picture]

They see the abstraction of the problem:

If we have an unlimited number for each bill type S[]=[S1, S2, Sm], what's the solution to select the least number of bills to pay price N. Use a function to represent the least number of bills: f(N).

In our case, S[]=[$1, $5, $6], N = 15

1) Optimal Substructure

   Let Sj be the last bill added to the optimal solution. So the total number of bills is: 1 (last bill) + the number of bills to make up N-Sj.
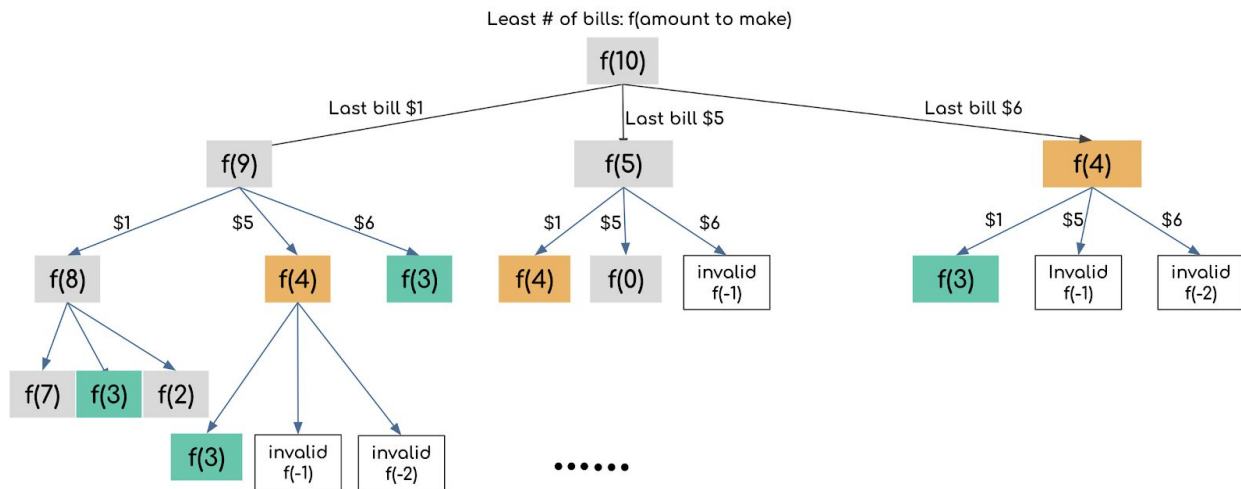
   f(N) = 1 + f(N-Sj)

   Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

2) Overlapping Subproblems

There are 3 possibilities for the last bill to make up N: $1, $5, $6.

   ● If the last bill is $1, the solution will be 1 + f(N-1).
   ● If the last bill is $5, the solution will be 1 + f(N-5).
   ● If the last bill is $6, the solution will be 1 + f(N-6).

If we continue to build the graph, we will see lots of overlapping subproblems highlighted in brown and green.

Least # of bills: f(amount to make)

f(10)

Last bill $1   Last bill $5   Last bill $6

f(9)   f(5)   f(4)

$1 $5 $6 $1 $5 $6 $1 $5 $6

f(8)   f(4)   f(3)   f(4)   f(0)   invalid f(-1)   f(3)   Invalid f(-1)   invalid f(-2)

f(7)   f(3)   f(2)

f(3)   invalid f(-1)   invalid f(-2)

• • • • • •

This Bill Change problem has the two DP properties, so we can safely go with DP solution!

State

The state is constructed by the required parameter that uniquely describes the subproblem. The function f(amount to make) indicates the state is constructed by amount to make. So we need to build a one dimension memo table.

Initialize memo table

amount	0	1	2	3	4	...
min # of bills	0					

when 0 amount to make, 0 bill needed

When amount = 1, we add $1 bill to the previous solution of f(1-1), making it 0+1=1 bills.

amount	0	1	2	3	4	...
min # of bills	0	1				

+ $1

When amount = 2, 3, 4, follow the logic above.

| amount	0	1	2	3	4	...
min # of bills	0	1	2	3	4	

+ $1   + $1   + $1

When amount = 5, we have more possibilities to select the last bill: $1 and $5.

- When the last bill is $1, we add 1 bill to the previous solution of f(5-1)), making it 4+1=5 bills.
- When the last bill is $5, we add 1 bill to the previous solution of f(5-5), make it 0+1=1 bill.

We choose the smaller number (5, 1) as the least number of bills for amount 5.

amount	0	1	2	3	4	5
min # of bills	0	1	2	3	4	5 1
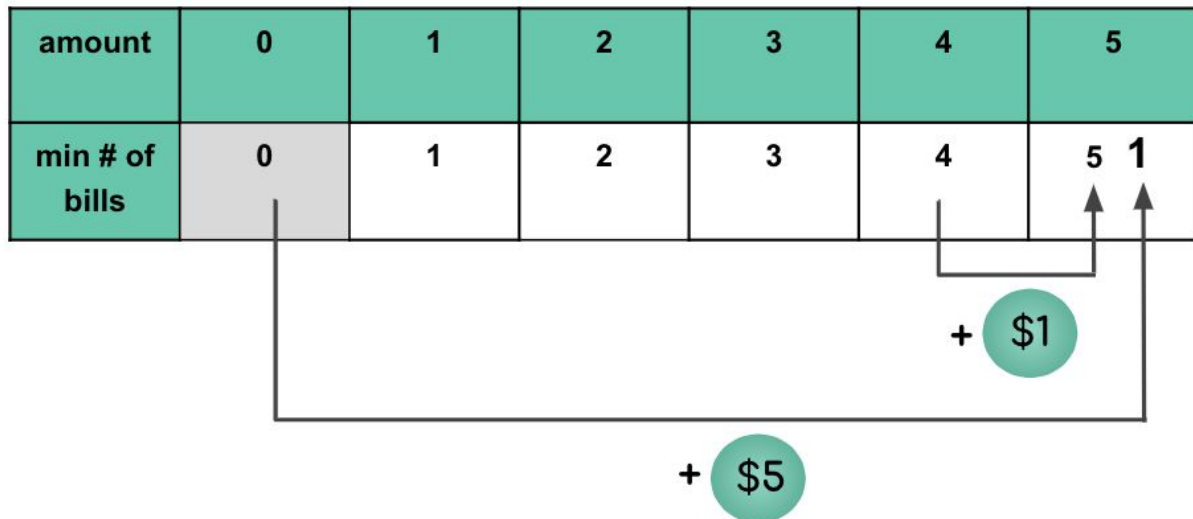
Similarly, we continue with f(6), f(7)...f(9), f(10).

When amount = 10, we have 3 possibilities to select the last bill: $1, $5 and $6.

- When the last bill is $1, we add 1 bill to the previous solution of f(10-1), making it 4+1=5 bills.
- When the last bill is $5, we add 1 bill to the previous solution of f(10-5), make it 1+1=2 bill.
- When the last bill is $6, we add 1 bill to the previous solution of f(10-6), make it 4+1=5 bill.

We choose the smallest number (5, 2, 5) as the least number of bills for amount 10.

amount	0	1	2	3	4	5	6	7	8	9	10
min # of bills	0	1	2	3	4	1	1	2	3	4	5 **2** 5

+ $1

+ $5

+ $6
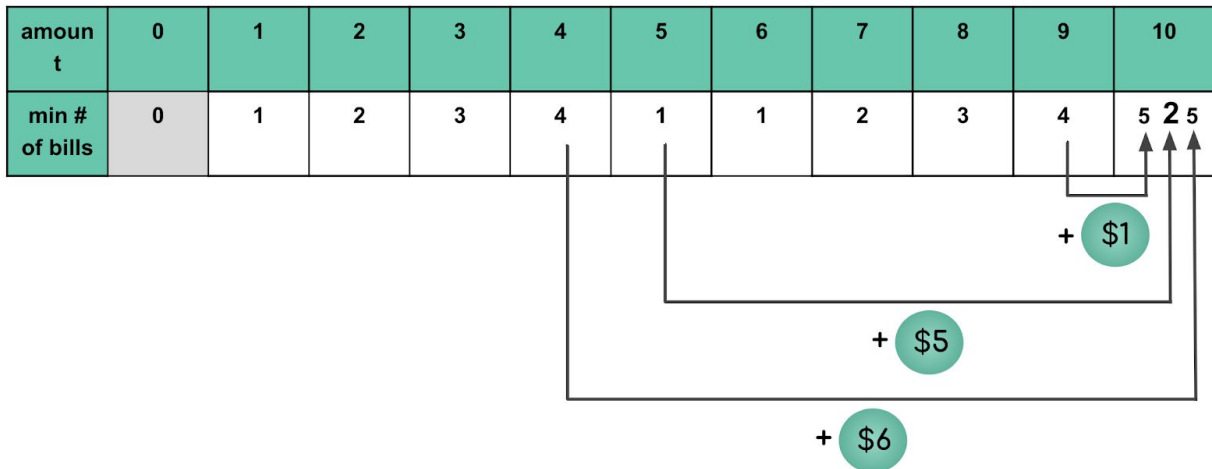
Ooh La La! Problem is solved!

Mighty Python

Bottom-Up Approach

```python
 1 # Apple Pi Inc.
 2 # Algorithmic Thinking
 3 # Coin Change Problem
 4 # DP Bottom Up Approach
 5
 6 # used to get the max number
 7 # to initialize dp table
 8 import sys
 9
10 # m is size of coins array (number of
11 # different coins)
12 def coin_change(amount):
13     # dp table
14     # dp[i] stores minimum number of coins required to
15     # make up amount i.
16     dp = []
17
18     # set initial values to infinite 1..N table cells
19     for i in range(N+1):
20         dp.append(sys.maxsize)
21
22     # alternative code
23     # dp = [sys.maxsize for i in range(N + 1)]
24
25     # base case: 0 amount needs 0 coin
26     dp[0] = 0
27
28     # build dp table
29     # from simplest problem (0 amount),
30     # step by step, work towards N.
31     for i in range(1, N + 1):
32
33         # Go through all coins smaller than i
34         for coin in coins:
35             if (coin <= i):
36                 previous_result = dp[i - coin]
37                 #if (previous_result != sys.maxsize and
38                 #    previous_result + 1 < dp[i]):
39                 if (previous_result + 1 < dp[i]):
40                     dp[i] = previous_result + 1
41     return dp[N]
42
43 coins = [1, 5, 6]
44 N = 10
45 print("Minimum number of coins to make up $10 is ",
46                 coin_change(N))
```

Top-Down Approach

[ToDo: add recursive code]

Fun Fact

You may think that all the coin systems in the world should be designed to work with Greedy algorithm so that people do NOT have to go through dynamic programming to find the coin change.

Well, country Bhutan is one of those exceptions.

How will you pay 40 chhertum? You can not first pick 25 chhertum coin. You should be less greedy and pick 20 chhertum coin. This is because

25 chhertum + 0 chhertum < 2 x 20 chhertum

Bhutanese Money

The monetary system of Bhutan is decimal based, with the primary unit of [Bhutanese money](#) being called the Ngultrum. The names and relative values of the coins depicted above are, from left to right:

- **Twenty Chhertum:** 20/100 of a Ngultrum

- **Twenty-Five Chhertum:** 25/100 of a Ngultrum

- **Fifty Chhertum:** 50/100 of a Ngultrum

- **One Ngultrum:** 100/100, 1 full Ngultrum

https://www.edupass.org/living-in-the-usa/money

US 3 cents piece

US half cent coin

US 2 cents piece

Canadian penny discontinued

Fibonacci Problem with DP

"Now I recall the Fibonacci sequence in nature. Does it qualify as DP problem?"
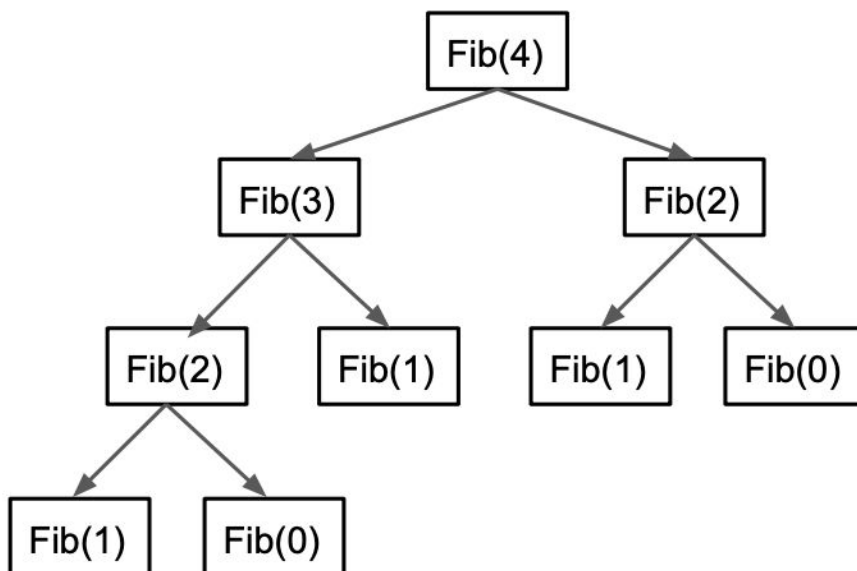
1. While we are calculating Fib(4), we need Fib(3) and Fib(2), Now for Fib(3), we need Fib(2) and Fib(1). This is a good demonstration of optimal substructure.
2. But we have calculated Fib(2) while calculating Fib(4). So we are solving many sub-problems again and again. This is a good representation of overlapping subproblems.
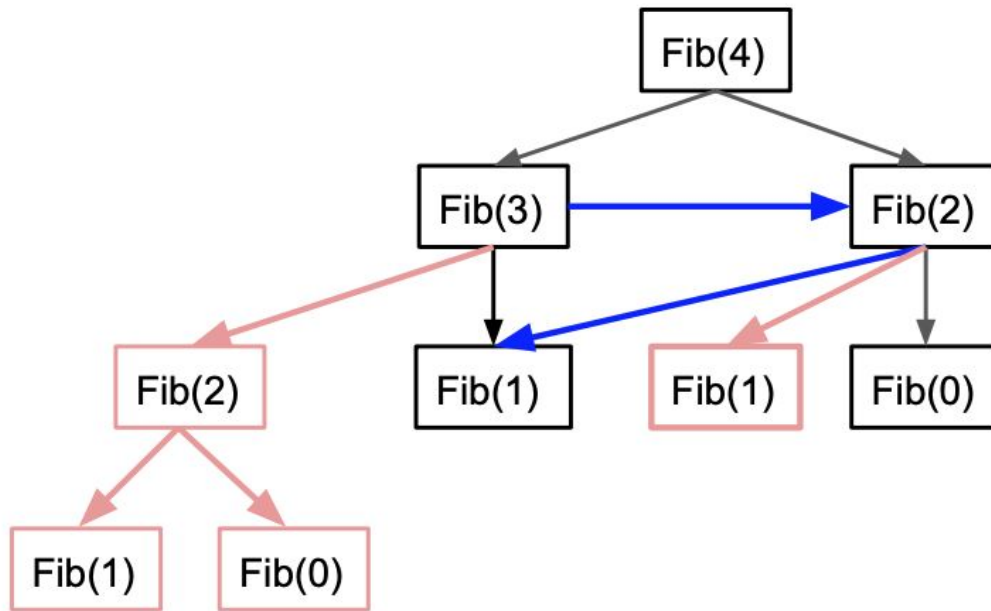
Overlapping subproblems are highlighted in red. The reusing subproblems are highlighted in blue. This graph is not a tree which indicates overlapping subproblems.

Therefore, Fibonacci is a good candidate to use DP approach.

Top down approach

```python
# initialize memo table
fib = [0 for i in range(100)]

# recursive function
# top down approach
def fibTopDown(n):
    if(n==0):
        return 0
    if(n==1):
        return 1

    # if solution in memo table
    # retrieve and use it
    if(fib[n]!=0):
        return fib[n]

    # otherwise, compute and
    # store the result in memo table
    fib[n] = fibTopDown(n-1) + fibTopDown(n-2);

    return fib[n];


# print the 10th fib number
print(fibTopDown(10))
```

Bottom up approach

```python
# bottom up approach
def fibBottomUp(n):

    # initialize memo table with
    # states of the base cases
    fib[0] = 0
    fib[1] = 1

    # transition to the next state
    # repeat the process until completion
    for i in range(2, n):
        fib[i] = fib[i-1] + fib[i-2]

    return fib[n]



# print the 10th fib number
print(fibBottomUp(10))
```

Bingo!

Bibliography

https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/#:~:text=Overlapping%20Subproblems%20Property%20in%20Dynamic%20Programming%20%7C%20DP%2D1,-Last%20Updated%3A%2003&text=Dynamic%20Programming%20is%20an%20algorithmic,computing%20the%20same%20results%20again.

https://en.wikipedia.org