Dynamic Programing

Dynamic Programming

- Dynamic programming like the divide and conquer method, solves problem by combining the solutions of sub problems

- Divide and conquer method partition the problem into independent sub problems, solves the sub problems recursively and then combine their solutions to solve the original problem.

Dynamic Programming

- Dynamic programming is applicable, when the sub-problems are NOT independent, that is when sub-problems share sub sub-problems.

- It is making a set of choices to arrive at optimal solution.

- If sub-problems are not independent, we have to further divide the problem.

- In worst case, we may end-up with an exponential time algorithm.

Dynamic Programming

- Frequently, there is a polynomial number of sub-problems, but they get repeated.

- A dynamic programming algorithm solves every sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time the sub-problem is encountered

- So we end up having a polynomial time algorithm.

- Which is better, Dynamic Programming or Divide & conquer?

Optimization Problems

- Dynamic problem is typically applied to <u>Optimization Problems</u>

- In optimization problems there can be many possible solutions. Each solution has a value and the task is to find the solution with the optimal (Maximum or Minimum) value. There can be several such solutions.

4 steps of Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution bottom-up.

4. Construct an optimal solution from computed information

Often only the value of the optimal solution is required so step-4 is not necessary.

Overlapping Sub problems

When a recursive algorithm revisits the same problem, over and over again, we say that the optimization problem has overlapping sub problems.

Typically the total number of distinct sub problems, is polynomial in the input size.

Divide & Conquer approach is suitable when brand new problems are generated at each step of the recursion.

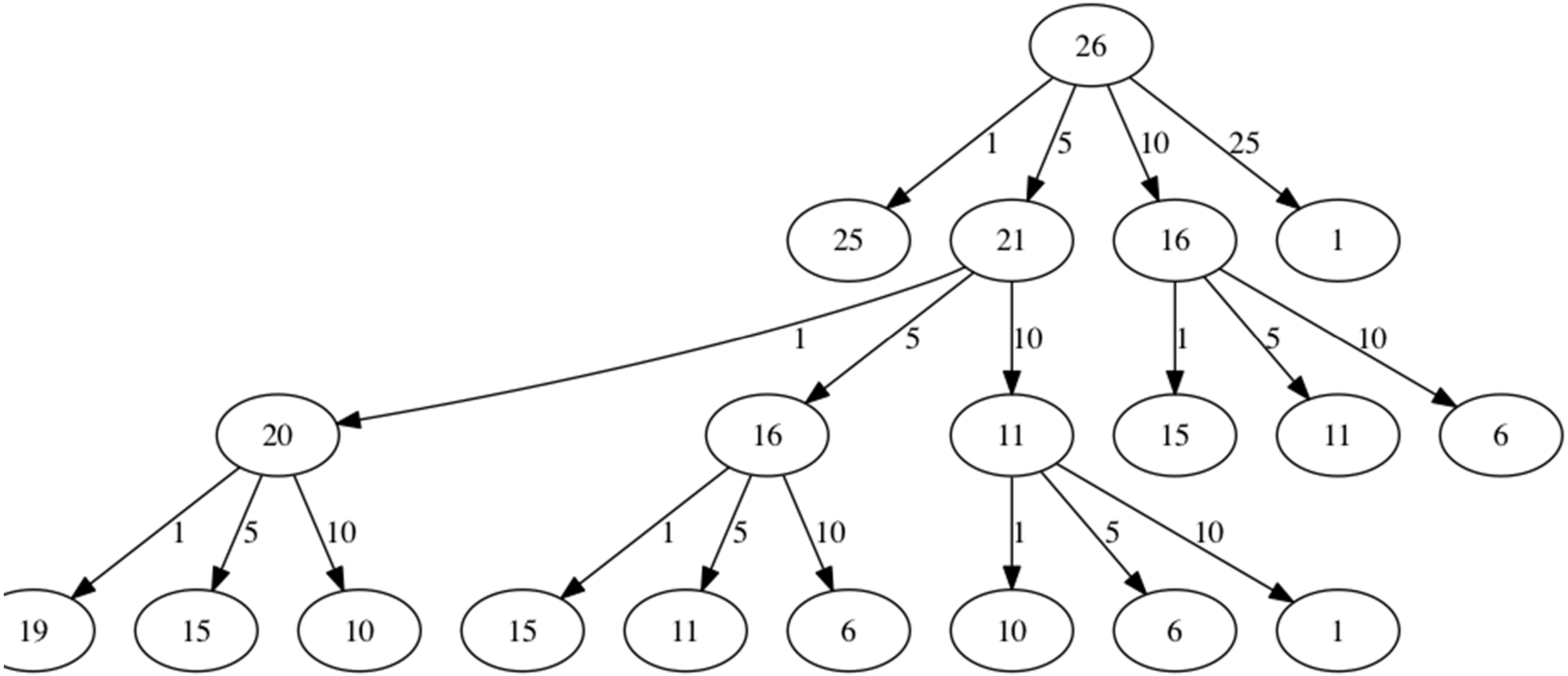
Coin Change Problem

The smallest number of coins you can use to make change

Recursive Solution

```
def recMC(coinValueList,change):
   minCoins = change
   if change in coinValueList:
     return 1
   else:
      for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recMC(coinValueList,change-i)
         if numCoins < minCoins:
            minCoins = numCoins
   return minCoins

print(recMC([1,5,10,25],63))
```


26
1
5
10
25
25
21
16
1
1
5
10
1
5
10
20
16
11
15
11
6
1
5
10
1
5
10
1
5
10
19
15
10
15
11
6
10
6
1

Caching/ memoization

```
def recDC(coinValueList,change,knownResults):
   minCoins = change
   if change in coinValueList:
      knownResults[change] = 1
      return 1
   elif knownResults[change] > 0:
      return knownResults[change]
   else:
       for i in [c for c in coinValueList if c <= change]:
         numCoins = 1 + recDC(coinValueList, change-i,
                              knownResults)
         if numCoins < minCoins:
            minCoins = numCoins
            knownResults[change] = minCoins
   return minCoins

print(recDC([1,5,10,25],63,[0]*64))
```

Dynamic Programing

```
def dpMakeChange(coinValueList,change,minCoins):
   for cents in range(change+1):
      coinCount = cents
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
      minCoins[cents] = coinCount
   return minCoins[change]
```

https://runestone.academy/runestone/books/published/pythonds/Recursion/DynamicProgramming.html

Recursive Definition of the Fibonacci Numbers

The Fibonacci numbers are a series of numbers as follows:

fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
...

$$\mathrm{fib}(n) = \begin{cases} 1, & n <= 2 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2), & n > 2 \end{cases}$$

fib(3) = 1 + 1 = 2
fib(4) = 2 + 1 = 3
fib(5) = 2 + 3 = 5

Memoization\Caching

```
Dictionary m;
m[0] = 0, m[1] = 1
Integer fib(n)
    if m[n] == null
        m[n] = fib(n-1) + fib(n-2)
    return m[n]
```

Bottom Up Approach

```
int fib(int n)
{
  /* Declare an array to store Fibonacci numbers.
*/
  int f[n+2];   // 1 extra to handle case, n = 0
  int i;

  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }

  return f[n];
}
```