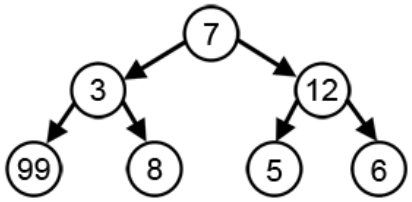
NOTES

Greedy Algorithm

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. However, in many problems, a greedy strategy does not produce an optimal solution.



Take a look at the diagram above and the greedy algorithm seeks to find the path with the largest sum. It does this by selecting the largest available number at each step. The greedy algorithm fails to find the largest sum, however, because it makes decisions based only on the information it has at any one step, without regard to the overall problem. The greedy algorithm here will take longest path based on maximum value at each level so it will pick 7->12->6 whose sum will be 25. But the actual longest path is 7->3->99 whose sum is 109.

A greedy algorithm will be able to solve the problem if it satisfies following properties-

1. **Greedy choice property:** A global (overall) optimal solution can be reached by choosing the optimal choice at each step means greedy algorithms work on problems for which it is true that, at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem.

2. **Optimal substructure:** A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.

Fractional Knapsack

```
def fractional_knapsack(value, weight, capacity):
        # index = [0, 1, 2, ..., n - 1] for n items
        index = list(range(len(value)))
        # contains ratios of values to weight
        ratio = [v/w for v, w in zip(value, weight)]
        # index is sorted according to value-to-weight ratio in decreasing order
        index.sort(key=lambda i: ratio[i], reverse=True)

        max_value = 0
        fractions = [0]*len(value)
        for i in index:
                if weight[i] <= capacity:
                        fractions[i] = 1
                        max_value += value[i]
                        capacity -= weight[i]
                else:
                        fractions[i] = capacity/weight[i]
                        max_value += value[i]*capacity/weight[i]
                        break

        return max_value, fractions

value = [100, 120, 60]
weight = [20, 30, 10]
capacity = 50

max_value, fractions = fractional_knapsack(value, weight, capacity)
print('The maximum value of items that can be carried:', max_value)
print('The fractions in which the items should be taken:', fractions)
```

Problem: Minimum Steps to One
Given a positive integer n, find the minimum number of steps that takes n to 1. Such that you can perform any one of the following 3 steps-
Subtract 1 from it. (n = n - 1)
If it's divisible by 2, divide by 2. (if n % 2 == 0 , then n = n / 2)
If it's divisible by 3, divide by 3. (if n % 3 == 0 , then n = n / 3).

Sample Input
n = 1
output: 0

Sample Input
n = 4
output: 2 (4/2=2, 2/2=1)

Sample Input
For n = 7
output: 3 (7-1=6, 6/3=2, 2/2=1)

One can think that make n as low as possible and continue the same, till it reaches 1.

```
def stepscount(n):
        steps = 0
        while  n > 1:
                        if n % 3 == 0:
                                        n//=3;
                        elif n % 2 == 0:
                                        n//=2;
                        else:
                                        n-=1
                        steps+=1

        return steps

print(stepscount(10))
```

Output: 4 as
10/2 = 5 (step-1)
5-1 = 4 (step-2)
4/2 = 2 (step-3)
2/2 = 1 (step-4)

If you observe carefully, the greedy strategy doesn't work here look at the other possible combination-
10-1 = 9 (step-1)
9/3 = 3 (step-2)
3/3 = 1 (step-3)

Here the approach is-

f(n) = 1 + f(n-1)
f(n) = 1 + f(n/2) // if n is divisible by 2
f(n) = 1 + f(n/3) // if n is divisible by 3

```
def getMinSteps(n, memo):
   if n == 1:
      return 0
   if memo[n] != -1:
      return memo[n];

   res = getMinSteps(n-1, memo)
```

```
    if n%2 == 0:
        res = min(res, getMinSteps(n//2, memo))
    if n%3 == 0:
        res = min(res, getMinSteps(n//3, memo))

    memo[n] = 1 + res;
    return memo[n];

n = 10
memo = [0]*(n+1)
for i in range(0,n+1):
            memo[i] = -1;

print(getMinSteps(n, memo))
```

Output: 3

Caution

In dynamic Programming all the subproblems are solved even those which are not needed, but in recursion only required subproblem are solved. So solution by dynamic programming should be properly framed to remove this ill-effect.