Coin Change

From Algorithmist

Coin Change is the problem of finding the number of ways of making changes for a particular amount of cents, n , using a given set of denominations $d_1 \ldots d_m$. It is a general case of Integer Partition, and can be solved with dynamic programming. (The Min-Coin Change is a common variation of this problem.)

Contents

Overview

The problem is typically asked as: If we want to make change for N cents, and we have infinite supply of each of $S = \{S_1, S_2, \ldots, S_m\}$ valued coins, how many ways can we make the change? (For simplicity's sake, the order does not matter.)

It is more precisely defined as:

Given an integer N and a set of integers $S = \{S_1, S_2, \ldots, S_m\}$, how many ways can one express N as a linear combination of $S = \{S_1, S_2, \ldots, S_m\}$ with non-negative coefficients?

Mathematically, how many solutions are there to $N = \sum_{k=1 \ldots m} x_k S_k$ where $x_k \geq 0, k \in \{1 \ldots m\}$

For example, for $N = 4, S = \{1, 2, 3\}$, there are four solutions: $\{1, 1, 1, 1\}, \{1, 1, 2\}, \{2, 2\}, \{1, 3\}$.

Other common variations on the problem include decision-based question, such as:

Is there a solution for $N = \sum_{k=1 \ldots m} x_k S_k$ where $x_k \geq 0, k \in \{1 \ldots m\}$ (Is there a solution for integer N and a set of integers $S = \{S_1, S_2, \ldots, S_m\}$?)

Is there a solution for $N = \sum_{k=1 \dots m} x_k S_k$ where $x_k \geq 0, k \in \{1 \dots m\}, \sum_{k=1 \dots m} x_k \leq T$ (Is there a solution for integer N and a set of integers $S = \{S_1, S_2, \dots, S_m\}$ such that $\sum_{k=1 \dots m} x_k \leq T$ - using at most T coins)

Recursive Formulation

We are trying to count the number of distinct sets.

The set of solutions for this problem, $C(N, m)$, can be partitioned into two sets:

- There are those sets that do not contain any S_m and
- Those sets that contain at least 1 S_m

If a solution does not contain S_m, then we can solve the subproblem of N with $S = \{S_1, S_2, \dots, S_{m-1}\}$, or the solutions of $C(N, m-1)$.

If a solution does contain S_m, then we are using at least one S_m, thus we are now solving the subproblem of $N - S_m$, $S = \{S_1, S_2, \dots, S_m\}$. This is $C(N - S_m, m)$.

Thus, we can formulate the following:

$$C(N, m) = C(N, m-1) + C(N - S_m, m)$$

with the base cases:

- $C(N, m) = 1, N = 0$ (one solution -- we have no money, exactly one way to solve the problem - by choosing no coin change, or, more precisely, to choose coin change of 0)
- $C(N, m) = 0, N < 0$ (no solution -- negative sum of money)
- $C(N, m) = 0, N \geq 1, m \leq 0$ (no solution -- we have money, but no change available)

Pseudocode

```
def count( n, m ):
    if n < 0 or m <= 0: #m < 0 for zero indexed programming languages
        return 0
    if n == 0: # needs be checked after n & m, as if n = 0 and m < 0 then it would return 1, which shoulc
        return 1

    return count( n, m - 1 ) + count( n - S[m], m )
```

Dynamic Programming

Note that the recursion satisfies the weak ordering $R(n, m) < R(x, y) \iff n \leq x, m \leq y, (n, m) \neq (x, y)$. As a result, this satisfies the optimal-substructure property of dynamic programming.

The result can be computed in $O(nm)$ time - the above pseudocode can easily be modified to contain memoization. It can be also rewritten as:

```
func count( n, m )

  for i from 0 to n
    for j from 0 to m
      if i equals 0
         table[i, j] = 1
      else if j equals 0
         table[i, j] = 0
      else if S_j greater than i
         table[i, j] = table[i, j - 1]
      else
         table[i, j] = table[i - S_j, j] + table[i, j-1]

  return table[n, m]
```


Categories: Dynamic Programming | Integer Partition