Tutorial for Paying Up

REVISIONS VIEW

Hello,

The problem "Paying Up" was one of the easy ones in the March 2009 contest on Codechef. It is considered an easy problem, because it has a couple of approaches that work. The problem statement boils down to finding a subset of banknotes such that their sum is exactly equal to a certain value. Now, this problem is somewhat similar to the knapsack problem which asks for ways to fill up a knapsack of a certain size optimally with the given blocks. There is a solution based on dynamic programming for this problem, but we will be taking up a solution which makes good use of the way integers are represented in binary to solve this problem.

Now, the limit on the number of banknotes is 'n'. Let us see how many subsets exist for these banknotes. For finding the number of subsets, we see that for every banknote, we have two choices, either to choose it in our calculation for the sum of notes or to ignore it in calculating the sum. Thus, we have 2 options per banknote and 'n' banknotes. So, the total number of subsets thus becomes 2^n where ^ represents the power operation. The number of subsets are small enough for us to bruteforce through them and the program to run in time.

An interesting way to generate all subsets of 'n' objects when 'n' is considerably small (n <= 20) is to use the properties of how unsigned integers are stored. Consider the number 2^n. This number is represented in binary form as '10000...0', that is, 1 followed by n 0s. Also, any number containing a 1 at the same position will surely be greater than 2^n. Thus, all numbers below 2^n do not have a 1 in a position greater than or equal to 'n' starting from the LSB. By induction, we can do the same for values of n = n-1 and n-2 and so on. Thus, we can see that any number between 2^(n-1) and 2^n will have a 1 in the position n-1. Extending this logic, we can say that if we consider the numbers from 1 to 2^n, we would be considering all possible ways in which we can choose some objects from 'n' objects.

For example, consider n = 3, so 2^n = 8. Let us list 'i' and its binary representation

i	Binary representation using 'n' bits
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

As you can see, if we consider that 1 means that we have selected object at that position and 0 means that we have not selected the object at that position, we can get all possible subsets of 'n' objects by looping over numbers from 1 to 2^n.

For calculating the sum of the subset represented by 'i', we loop from 0 to 'n' and we check whether the corresponding bit is set in the value for 'i' or not. If it is, we include that object in calculating our sum else we don't. In this way, we can get the sum for all possible subsets of the 'n' objects.

This is exactly what we need for this problem. After taking in the number of objects, we loop 'i' from 1 to 2^n incrementing the value of 'i' by 1 at every stage to give us a new subset. Then for a particular value of 'i', we loop 'j' from 0 to n and check if the bit at that particular value is set or not. Languages like C / C++ provide bit-wise operators for leftshift and rightshift. For checking if the bit is set at position 'j'(starting from 0) we can just check if the value of (i & (1<<j)) is 0. If it is 0, then the bit is not set, while if it is greater than 0, then the bit is set. Alternatively, we can also loop from 0 to n and at each stage check whether 'i' modulo 2 is equal to 1 or not. If it is 1, then the bit at that position is set, else it's not. Then we divide 'i' by 2 and proceed. At the end of the 'n' iterations, 'i' will equal 0. The problem with this appraoch is that the modulo operations take much more time compared to the bitwise operations. Thus, now that we know how to check if the bit is set, we initialize a value 'sum' equal to 0 at the start of the 'n' iterations for a value of 'i' and if the bit at position 'j' is set, we add the corresponding banknote value to 'sum' else we don't. At the end of these iterations, we check if the value of 'sum' equals the required value. If it does, then we have found a subset with the required sum and so we print a "Yes" and exit. Else, if at the end of 2^n iterations of 'i' we don't have a subset with the required sum, then we print a "No" and exit.

The program should look something like this :

```
Start

Take in the value of 'n' and the required value of sum 'm'

Take in all the values for the banknotes in array 'd[]'

For i = 1 and i < (2^n)

sum = 0

For j = 0 and j < n

if jth bit of i is set

 sum = sum + d[j]

 if sum equals m

 print Yes and return

 Print No and return
```

Gaurav Gaba *- 31st Dec,2009 17:13:41.*

can smone fnd error in my code i dnt why it's giving wrong answer.....#include<stdio.h>