**CS 2420**

**Program 8 (Dynamic Programming)**

**Part 1: Understanding the problem**

You need to sell a collection of matching teacups. You know how much net income you will receive (after accounting for shipping and handling) for each quantity of teacups. You need to figure out which combination will give you the most profit. You will do this using dynamic programming.

A group of teacups

Description automatically generated with medium confidence

|  |  |
| --- | --- |
| # of Teacups | Profit for set |
| 1 | 1 |
| 2 | 3 |
| 3 | 5 |
| 4 | 9 |
| 5 | 10 |
| 6 | 15 |
| 7 | 17 |
| 8 | 18 |
| 9 | 19 |
| 10 | 22 |
| 11 | 25 |
| 12 | 27 |

So, if you had 5 teacups, your choices for dividing are:

* 1 + 4 ( $10) (one set of size 1 and one set of size 4)
* 1 +1 + 3 ($7)
* 1+1+1 +2 ($6)
* 1+1+1+1+1 ($5)
* 1+ 2 + 2 ($7)
* 3+2 ($8)
* 5 ($10)

With 10 matching teacups, there would be many ways to divide up the sets. Using recursion, write the code to itemize all possible ways of dividing the teacups into various sized sets.

Consider a teaset size of 10. Show all the ways of dividing the sets. Recursion works great for this.

**Output**

**Teaset Size=10**

1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 2

1 1 1 1 1 1 1 3

1 1 1 1 1 1 2 2

1 1 1 1 1 1 4

1 1 1 1 1 2 3

1 1 1 1 1 5

1 1 1 1 2 2 2

1 1 1 1 2 4

1 1 1 1 3 3

1 1 1 1 6

1 1 1 2 2 3

1 1 1 2 5

1 1 1 3 4

1 1 1 7

1 1 2 2 2 2

1 1 2 2 4

1 1 2 3 3

1 1 2 6

1 1 3 5

1 1 4 4

1 2 7

1 3 3 3

1 1 8

1 2 2 2 3

1 2 2 5

1 2 3 4

1 3 6

1 4 5

1 9

2 2 2 2 2

2 2 2 4

2 2 3 3

2 2 6

2 3 5

2 4 4

2 8

3 3 4

3 7

4 6

5 5

10

**Hint:** for part 1, my prototype looked like:

// amt: Number of teacups left to dividie

// soFar is the teaset sizes I’ve already decided to use

// currentSize minimum additional teaset size I’m considering (helps to make progress)

**void** printAll(**int** amt, String soFar, **int**  currentSize)

So printAll(6, “2 2”, 3) is asking me to print all ways of dividing the 10 teacups when I have 6 more teacups to divide, I’ve decided to divide the original set by breaking them into two sets of 2, and I only want to consider additional sets of at least size 3.

The initial call looks like printAll(10,””,1), but after I make some decisions I want to force the method to only consider set sizes I haven’t yet considered.

**Part 2: Dynamic Programming**

**Dynamic Programming** (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.  If there are N teacups to sell and S number of teacup set sizes, the complexity is SxN as you fill in the 2D space.

Set 1

Best Sum for (1 teacup) :$1 1

Best Sum for (2 teacup) :$3 2

…

Best Sum for (23 teacup) :$56 7 6 6 4

Best Sum for (24 teacups): $60 6 6 6 6

You will need a table with 24 columns, but the first half of the table is shown below.

Table

Description automatically generated

Consider another data set:

|  |  |
| --- | --- |
| # of Teacups | Profit for set 2 |
| 1 | 2 |
| 2 | 5 |
| 3 | 8 |
| 4 | 9 |
| 5 | 10 |
| 6 | 15 |
| 7 | 19 |
| 8 | 23 |
| 9 | 24 |
| 10 | 29 |
| 11 | 30 |
| 12 | 32 |

Set 2

Best Sum for (1 teacup) :$2 1

Best Sum for (2 teacup) :$5 2

…

**Output:**

For each data set:

For each teacup size (1 through 24), show the best dollar amount possible from the sale and the way the cups were divided for sale.

So for the first data set: Best Sum for (24 teacups): $60 6 6 6 6 indicates you could get $60 by dividing the 24 cups into four sets each of size 6.

**Bonus (2 points)**

There are three techniques that are similar. This assignment uses the dynamic programming solution.

Here are other solutions approaches:

1. **Recursion**

int bestSolution(int itemSize, int amt) where you recursively consider (a) using the item of size itemSize or (b) not  using the item.  The complexity is 2^m where m is the number of possible items (which may be more than  N if you are allowed to use the item more than once ).  We would expect this to be much slower as the subproblems are computed multiple times.

1. Memoizing

Memoizing is a lot like the recursive solution, only you store all the solutions in a 2D array ( like you used in dynamic programming).  The difference is in the order in which you fill in the array.  In memoizing, you only store smaller solutions you actually use (instead of everything).   The complexity is still A\*N as you end up using almost all of the array.  In the examples I ran, everything was filled in except the lower left-hand triangle.  The runtime appeared to be similar to dynamic programming.

For the bonus points, solve the problem all three ways and compare the run time. Here is one way to measure runtime.

import java.util.concurrent.TimeUnit;

long startTime = System.nanoTime();

//WORK GOES HERE

long endTime = System.nanoTime();

long timeElapsed = endTime - startTime;

System.out.println("Execution time in nanoseconds: " + timeElapsed);

System.out.println("Execution time in milliseconds: " + timeElapsed / 1000000);