# Change Making

Our next problem involves something we all do every day, and most of us do it instinctively: making change using the fewest possible coins.

For example, suppose our set of coin values is {1,5,10,25,100,200} - similar to Canada's coins before the penny was phased out. (Canada does have 50 cent coins but they are not often seen so I have left them out.)

If we need to make a total of $1.73, we naturally use 1 loonie, 2 quarters, 2 dimes and 3 pennies, using a total of 8 coins. There is no solution for $1.73 that uses fewer than 8 coins.

This - as we now recognize - is a greedy algorithm: repeatedly use the largest coin we can. We can prove that it gives the optimal solution using the standard method: proof by induction.

It's worth looking at this proof in some detail because it involves something we don't see all that often – multiple base cases. We will apply induction to the target value $n$.

Base cases:

$1 \leq n < 5$: the only solution is n pennies, which is what the algorithm does

$n = 5$: the algorithm solves this case with $1$ coin, which is clearly optimal

$5 < n < 10$: the only solutions are $n$ pennies, or $1$ nickel + $n - 5$ pennies. The second option clearly uses fewer coins, and this is what the algorithm does.

$n = 10$: the algorithm solves this case with $1$ coin

$10 < n < 25$: Note that no optimal solution can contain more than 4 pennies (because $5$ pennies can be replaced by $1$ nickel, reducing the number of coins). Similarly an optimal solution can contain at most $1$ nickel. Thus for $n > 9$ it is not possible to have an optimal solution consisting only of pennies and nickels. From this we see that the algorithm's first choice of a dime is correct. We can verify that the algorithm's subsequent choices are also optimal.

$n = 25$: the algorithm solves this case with $1$ coin

Continuing in this fashion we can show that the algorithm finds an optimal solution for all values of $n \leq 200$ ... and we are finally finished with the base cases!

Inductive Hypothesis: Suppose the algorithm finds an optimal solution for all values of $n \leq k$ for some value of $k \geq 200$

Let $n = k + 1$

Suppose there is no optimal solution containing a toonie (ie the algorithm's first choice is wrong). Then the optimal solution can contain only pennies, nickels, dimes, quarters, and loonies. In fact it can contain at most $1$ loonie and $3$ quarters and $4$ pennies. It can contain either $2$ dimes and $0$ nickels, or $1$ dime and $1$ nickel (try to figure out why all these conditions hold). Within these constraints, the maximum total that can be achieved is $199$, which contradicts the fact that $n > 200$. Thus the algorithm's first choice is correct – it is contained in an optimal solution.

From this point the proof proceeds in the standard fashion.

But our coin values are carefully (or luckily) chosen, and the greedy algorithm happens to work for us. Suppose the set of coin values is $\{1, 4, 9\}$ and the target value is $12$. Now the greedy algorithm will start with $1$ "niney" then $3$ pennies, using $4$ coins in total. But $3$ "foursies" also equals $12$ ... a better solution. So in this situation the greedy algorithm fails to find an optimal solution.

A natural question to ask is ***why*** the greedy algorithm works for the first set of coin values but not for the second. It turns out this is an incredibly deep question! It has given rise to a specialized branch of mathematics called *greedoid theory,* with the goal of finding a general characterization of optimization problems that can be solved with greedy algorithms. For an introduction see the Wikipedia article https://en.wikipedia.org/wiki/Greedoid

Our base cases are

```
   Min_Coins(ci) = 1 for each ci in the coin set
   Min_Coins(x) = infinity if x < 0        (this may seem strange but it simplifies our algorithm)
```

and the recursive part is

```
   Min_Coins(n) = min{1 + Min_Coins(n-c1),
                      1 + Min_Coins(n-c2),
                        ...
                      1 + Min_Coins(n-ck)
                     }
```

We see that Min_Coins(n) depends on the solution of k smaller problems. If k is fixed (it usually is) then only a constant amount of work is needed to compute Min_Coins(n) ... assuming the smaller problems have already been solved. We can guarantee this by solving all smaller problems from 1 up to n-1 - this guarantees that we will have all the required subproblems already solved when we get to solving for n.

**Dynamic Programming Paradigm**

Constructing a dynamic programming solution to an optimization problem always involves the same steps. I'll describe them in a sequence, but in practice they usually proceed in parallel.

1. Find a recurrence relation that defines an optimal solution in terms of optimal solutions to subproblems, and establish the base case(s).

2. Determine one or more parameters that define each subproblem. In the path problem, these were the row and column of the vertex. In the other problems, they were the size of the bar, and the target money value.

3. Define a table to hold the values of optimal solutions for the subproblems. If each subproblem is defined by 2 parameters, this will typically be a two-dimension table. If the subproblems are defined by a single parameter, the table is typically one-dimensional.

4. Determine the order in which the elements of the table will be filled in. There may be alternatives - the essential requirement is that when we want to fill in a particular element of the table, the subproblems on which it depends have already been solved and their table elements have been filled in.

5. Determine how we will use the completed table to extract the details of the optimal solution. There are two popular methods:

- store information in the table that indicates which particular subproblems provide the optimal solutions to larger problems, so that when we get to the optimal solution for the original problem, we can easily trace back the steps that get us there
- work backwards from the final entry in the table, re-examining the different subproblems that might have contributed to it, and determine which subproblem(s) were actually chosen ... and then work backwards from there in the same way.

Now let's look at an application of dynamic programming to a problem that has haunted us throughout the course:

**Subset-Sum Problem**

Let S = { $s_1, s_2, \ldots, s_n$ } and let the target value be k.

I'm going to present the dynamic programming solution very briefly - you should be able to fill in any gaps and complete the steps outlined above.

We can look at the process of finding a solution as a sequence of decisions: first we decide whether or not to include $s_1$ , then $s_2$, etc. This suggests constructing a table S_S in which we list the elements of S on one axis, and possible target values on the other. For x, we will use 1 .. k. Then S_S(i,x) = True iff $\{s_1 \ldots s_i\}$ contains a subset that sums to x.

We can establish the necessary recurrence by observing that

S(1,x) = T iff $s_1$== x for all x ( base cases)

S(i,x) = T iff S(i-1, x) - there is a subset of $\{s_1 \ldots s_i)$ } that sums to x

or

S(i-1, x - $s_i$) - there is a subset of $\{s_1 \ldots s_{i-1}$ } that sums to x - $s_i$

Thus each value in the table is based on two values in the previous row. If the table has a T anywhere in the column for k, then the answer to Subset_Sum(S,k) is yes, and if not then the answer is no.

This gives us a concise and simple-seeming algorithm that will always correctly solve Subset Sum ... and yet earlier we learned that Subset Sum is NP-Complete. Does our new algorithm show that **P** = **NP** ?