# CMSC 351: Coin Changing

Justin Wyss-Gallifent

August 30, 2023

# 1 Introduction to Coin Changing

Suppose all you have are 1-cent, 5-cent and 10-cent coins but you have infinitely many of each. For any given (cent) total $n$ we wish to obtain $n$ cents out of our coins. Consider the following associated problems:

(a) How can we do this if we don't care how many coins we use?

(b) How can we do this if we wish to use the minimum number of coins?

(c) How many ways can we do this if we don't care about using the minimum number of coins?

In this case our coins can be thought of in a list $C = [1, 5, 10]$ and even though we'll change that we'll always assume that we have a 1-cent coin. This guarantees that it's possible to obtain any amount.

**Note 1.0.1.** The reason this is known as the *coin changing problem* is that the original premise is that the total $n$ is the amount of change being given for a purchase and the question was about how this can be done.

**Note 1.0.2.** The question can be asked even without a 1-cent coin but it gets more challenging. For example if $C = [3, 7]$ it's not clear at all which totals can be even be made.

# 2 The Greedy Method (a Minimization Attempt)

An intuitive approach (which doesn't always work, as we'll see) to using the minimum number of coins is to be *greedy*. Since we wish to use the minimum number of coins it seems sensible to use as many of the large coins as possible, followed by the next largest, and so on.

**Example 2.1.** Suppose $C = [1, 5, 10]$ and we wish to obtain $n = 27$ cents. We first grab two 10-cent coins (the most we can have) followed by one 5-cent coin (the most we can have) followed by two 1-cent coins. We have thus used 5 coins.

This is in fact optimal - it is the minimal number of coins, but this may not be obvious.

**Example 2.2.** Suppose $C = [1, 10, 25]$ and we wish to obtain $n = 30$ cents. We first grab one 25-cent coins (the most we can have), we can't grab any 10-cent coins, so we finish by grabbing five 1-cent coins. We have thus used 6 coins.

This solution is not optimal however since we could have grabbed three 10-cent coins instead.

# 3 An Algorithm for Minimization

## 3.1 A Dynamic Programming Idea

We've seen that our greedy approach is not guaranteed to give us an optimal (smallest) number of coins. However we know that there must be some optimal solution so we can ask - how might we go about finding it?

Before developing an algorithm let's make an observation. For now let's stick with $C = [1, 5, 10]$.

Let's suppose $A$ is a function such that $A[x]$ equals the minimum number of coins necessary to obtain $x$ cents. So for example it's easy to see that:

$$
\begin{aligned}
&A[0] = 0 \text{ (No coins.)}\\
&A[1] = 1 \text{ (One 1-cent coin.)}\\
&A[2] = 2 \text{ (Two 1-cent coins.)}\\
&A[3] = 3 \text{ (Three 1-cent coins.)}\\
&A[4] = 4 \text{ (Four 1-cent coins.)}\\
&A[5] = 1 \text{ (One 5-cent coin.)}\\
&A[6] = 2 \text{ (One 5-cent coin and one 1-cent coin.)}
\end{aligned}
$$

Suppose we know $A[0], ..., A[x - 1]$ for some $x$. Is there a sneaky way to obtain $A[x]$?

The answer is fairly easy! In the $C = [1, 5, 10]$ case there are three possibilities:

1. We could first select a 1-cent coin, then obtain $x - 1$, then combine. We can do this with $A[x - 1] + 1$ coins.
2. We could first select a 5-cent coin, then obtain $x - 5$, then combine. We can do this with $A[x - 5] + 1$ coins. Note that this is only a possibility if $x \geq 5$ because if $x < 5$ then $x - 5 < 0$ which we can't do.
3. We could first select a 10-cent coin, then obtain $x - 10$, then combine. We can do this with $A[x - 10] + 1$ coins. Note that this is only a possibility if $x \geq 10$ because if $x < 10$ then $x - 10 < 0$ which we can't do.

So what we'll do is assign $A[x]$ to be the minimum of these three (or rather the minimum of those that make sense).

*Proof.* It's fairly easy to see that this is optimal and here's the basic proof for the $C = [1, 5, 10]$ case which generalizes easily:

Assume by way of contradiction that:

$$A[x] < \min \{A[x - 1] + 1, A[x - 5] + 1, A[x - 10] + 1\}$$

Suppose the coin combination used to obtain the actual optimal solution for $x$ involves a $c$-cent coin where $c \in \{1, 5, 10\}$ (it has to involve at least one of these). Then $x - c$ cents may be obtained by removing a $c$-cent coin from this optimal solution for $x$ which implies that $A[x-c] \leq A[x] - 1$.

However the assumption tells us that $A[x] < A[x-c] + 1$ and so we have:

$$A[x] < A[x-c] + 1 \leq (A[x] - 1) + 1 = A[x]$$

This is a contradiction. $\mathcal{QED}$

Thus we have a way of obtaining $A[x]$ as long as we know all previous values. To jump-start this process observe that $A[0] = 0$ because it takes 0 coins to obtain 0 cents.

Let's see how this works for $A[0]$ through $A[20]$ with a table. We won't do all the values but rather point out some critical ones. We start with:

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A[x]$ | 0 | | | | | | | | | | | | | | | | | | | | |

For $A[1]$ we look at $1 + A[1-1] = 1 + A[0] = 1$, $1 + A[1-5] = 1 + A[-4] = BAD$, $1 + A[1-10] = 1 + A[-9] = BAD$ so we only have one value and so $A[1] = 1$.

For $A[2]$ we look at $1 + A[2-1] = 1 + A[1] = 2$, $1 + A[2-5] = 1 + A[-3] = BAD$, $1 + A[2-10] = 1 + A[-8] = BAD$ so we only have one value and so $A[2] = 2$.

If we do $A[3], A[4]$ (try them!) we have:

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A[x]$ | 0 | 1 | 2 | 3 | 4 | | | | | | | | | | | | | | | | |

For $A[5]$ we look at $1 + A[5-1] = 1 + A[4] = 5$, $1 + A[5-5] = 1 + A[0] = 1$, and $1 + A[5-10] = 1 + A[-5] = BAD$ so we have two values and take the minimum and so $A[5] = 1$.

For $A[6]$ we look at $1 + A[6-1] = 1 + A[5] = 2$, $1 + A[6-5] = 1 + A[1] = 2$, and $1 + A[6-10] = 1 + A[-4] = BAD$ so we have two values and take the minimum and so $A[6] = 2$.

For $A[7]$ we look at $1 + A[7-1] = 1 + A[6] = 3$, $1 + A[7-5] = 1 + A[2] = 3$, and $1 + A[7-10] = 1 + A[-3] = BAD$ so we have two values and take the minimum and so $A[7] = 3$.

If we do $A[8], A[9]$ (try them!) we have:

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A[x]$ | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 4 | 4 | 5 | | | | | | | | | | | |

For $A[10]$ we look at $1 + A[10-1] = 1 + A[9] = 6$, $1 + A[10-5] = 1 + A[5] = 2$, and $1 + A[10-10] = 1 + A[0] = 1$ so we have two values and take the minimum and so $A[10] = 1$.

If we continue this process to $A[20]$ we find:

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A[x]$ | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 4 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 3 | 4 | 5 | 6 | 7 | 2 |

### 3.2 An Algorithm

The preceeding idea can then give us a nice algorithm. Suppose $C$ is a list of coin denominations, so for example $C = [1, 5, 10]$ in the example we've been working through. Then we do the following to obtain the value for some $x$:

```
A = empty list which can grow as needed
A[0] = 0
C = list of coin denominations
for n = 1 to x:
    howmanycoins = infinity
    for each coin in C:
        if n - coin >= 0:
            howmanycoins = min(howmanycoins,1+A[n - coin])
        end if
    end for
    A[n] = howmanycoins
end for
```

Take a minute to digest how this works. For each $n$ up to and including $n = x$ we iterate through the coins looking at each $n - coin$ and for those that make sense we obtain the minimum of all the $1 + A[n - coin]$ values.

### 3.3 Time Complexity

While we haven't (yet) discussed time complexity in detail in this course there are two things we can observe:

- There are two loops - the outer loop iterates $x$ times and the inner loop iterates length($C$) times and so the innermost body iterates $x$length($C$) times.
- This is an approach known as *dynamic programming* which typically means that we use earlier solutions to efficiently calculate later ones. In this case we saw that to calculate $A[x]$ we use inputs smaller than $x$. Once we know the values up to $A[x - 1]$, finding $A[x]$ is quick.

## 4 Counting the Ways

### 4.1 Introduction

Now let's consider the problem of how many ways we can obtain $n$ cents. We'll go back to our initial currencies of 1, 5, and 10.

For specific values of $n$ we can brute-force this. For example for $n = 10$ we can do:

- Ten 1-cent coins.

- Two 5-cent coins.
- One 10-cent coin.
- One 5-cent coin and five 1-cent coins.

Thus we have a total of 4 ways.

Suppose we wished to write an algorithm which would give the answer for any given $n$. How might we go about this? Moreover what if we had a different set of coins rather than 1,5,10?

## 4.2 A Dynamic Programming Approach

Before creating a general approach (which will lead to an algorithm) consider the following observation:

Suppose we have two coin denominations 2 and 5. If $n = 12$ then it's easy to see that there is 1 way to obtain $n = 12$ using only 2-cent coins. How about if we also allow 5-cent coins? Observe that we have a disjoint sum:

$$\begin{aligned}\#\text{ ways to get 12 using 2 and/or 5} &= \#\text{ ways to get 12 using 2}\\ &+ \#\text{ ways to get 12 using 2 and at least one 5}\end{aligned}$$

We know the first summand is 1. For the second summand once we choose to use a single 5-cent coin we have 7 cents left to obtain and we can do that however we wish. Thus:

$$\begin{aligned}\#\text{ ways to get 12 using 2 and/or 5} &= \#\text{ ways to get 12 using 2}\\ &+ \#\text{ ways to get 7 using 2 and/or 5}\end{aligned}$$

Take a moment to see what we have observed here. As a general rule for some $n$ and two denominations $c_1$ and $c_2$ we have:

$$\begin{aligned}\#\text{ ways to get } n \text{ using } c_1 \text{ and/or } c_2 &= \#\text{ ways to get } n \text{ using } c_1\\ &+ \#\text{ ways to get } n - c_2 \text{ using } c_1 \text{ and/or } c_2\end{aligned}$$

This generalizes even futher with a set of coin denominations $[c_1, ..., c_r]$:

$$\begin{aligned}\#\text{ ways to get } n \text{ using } c_1, ..., c_r &= \#\text{ ways to get } n \text{ using } c_1, ..., c_{r-1}\\ &+ \#\text{ ways to get } n - c_r \text{ using } c_1, ..., c_r\end{aligned}$$

Before proceeding note that the above is only true if $n \geq c_r$ since otherwise we can't use a $c_r$ denomination coin at all. Thus more accurately:

- If $n \geq c_r$ then:

$$\begin{aligned}\# \text{ ways to get } n \text{ using } c_1, ..., c_r = \# &\text{ ways to get } n \text{ using } c_1, ..., c_{r-1} \\ + \# &\text{ ways to get } n - c_r \text{ using } c_1, ..., c_r\end{aligned}$$

- Otherwise we simply have:

$$\# \text{ ways to get } n \text{ using } c_1, ..., c_r = \# \text{ ways to get } n \text{ using } c_1, ..., c_{r-1}$$

## 4.3 An Algorithmic Idea

The above observation leads to the following. Suppose we have coin denominations $[c_1, ..., c_r]$ and wanted to know the number of ways to obtain $n$ cents using any combinations of these denominations.

Suppose we have an array $A$ indexed 0 through $n$ and we have some $k < n$ such that:

- $A[0]$,...,$A[k]$ contain the number of ways to obtain 0 through $k$ cents using any denominations from $c_1$ through $c_r$.
- $A[k+1]$,...,$A[n]$ contain the number of ways to obtain 0 through $k$ cents using any denominations from $c_1$ through $c_{r-1}$.

Suppose we wish to update $A[k+1]$ so that it contains the number of ways to obtain $k+1$ using any denominations from $c_1$ through $c_r$.

From the above rule we immediately see that:

- If $n \geq c_r$ then we update it as follows:

$$A[k+1] := A[k+1] + A[k+1-c_r]$$

- Otherwise we leave it alone.

## 4.4 An Actual Algorithm

Our algorithmic approach will emerge from this idea. We will start with such an array and pre-load it with the number of ways to achieve each of $0, 1, ..., n$ using no coins. Then we will update it with the number of ways to do so using just $c_1$, then we will update it with the number of ways to do so using $c_1$ and/or $c_2$, and so on, until we are done.

On to the algorithm!

Suppose we have coin denominations $C = [c_1, ..., c_r]$ and we wish to find the number of ways to obtain $n$ cents using any denomination from $c_1$ through $c_r$.

We first assign an array $A$ indexed 0 through $n$ as follows:

$$A = [1, 0, 0, ..., 0]$$

In this array, $A[i]$ tells us the number of ways to obtain 0 through $n$ cents using no coins at all.

We then iterate over the denominations and update accordingly. Here is the pseudocode:

```
function coincount(C,n):
  A = [1,0,0,...,0]
  for c in C:
    for i = 1 to n:
      if i >= c:
        A[i] = A[i] + A[i-c]
      end if
    end for
  end for
end function
```

**Example 4.1.** Let us walk through this with `n=10` and `C = [1,5,10]`.

We assign:

```
A = [1,0,0,0,0,0,0,0,0,0,0]
```

Observe that `A` now contains the number of ways to obtain 0 through 10 using no coins.

We assign `c = 1` and pass through `i = 1,2,...,10` yielding:

```
A = [1,1,1,1,1,1,1,1,1,1,1]
```

Observe that `A` now contains the number of ways to obtain 0 through 10 using just 1-cent coins.

We assign `c = 5` and pass through `i = 1,2,...,10` yielding:

```
A = [1,1,1,1,1,2,2,2,2,2,3]
```

Observe that `A` now contains the number of ways to obtain 0 through 10 using 1- and 5-cent coins.

We assign `c = 10` and pass through `i = 1,2,...,10` yielding:

```
A = [1,1,1,1,1,2,2,2,2,2,4]
```

Observe that `A` now contains the number of ways to obtain 0 through 10 using 1-, 5-, and 10-cent coins.

**Example 4.2.** Let us walk through this with `n=15` and `C = [2,5,7]`.

We assign:

```
A = [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

Observe that `A` now contains the number of ways to obtain 0 through 15 using no coins.

We assign `c = 2` and pass through `i = 1,2,...,15` yielding:

```
A = [1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0]
```

Observe that `A` now contains the number of ways to obtain 0 through 15 using just 2-cent coins.

We assign `c = 5` and pass through `i = 1,2,...,15` yielding:

```
A = [1,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1]
```

Observe that `A` now contains the number of ways to obtain 0 through 15 using 2- and 5-cent coins.

We assign `c = 7` and pass through `i = 1,2,...,15` yielding:

```
A = [1,0,1,0,1,1,1,2,1,2,1,2,2,2,3,2]
```

Observe that `A` now contains the number of ways to obtain 0 through 15 using 2-, 5-, and 7-cent coins.

## 4.5 Time Complexity

We have not formally started talking about time complexity but for now it is absolutely worth simply mentioning that the algorithm involves two nested loops such that the innermost body iterates $\text{length}(C)\,(n+1)$ time.

Thus intuitively the time required increases linearly as a function of either $n$ or the number of denominations available.