Pseudocode Examples

- This is it, folks: using pseudocode as a first step toward expressing *ordered*, *unambiguous*, and *executable* steps that define a *terminating* process (i.e., an algorithm) initiates you into the realm of computer science
- While the textbook contains a more thorough discussion of pseudocode, this is one of those areas where you can't have too many examples
- This handout works through pseudocode approaches to a "make change" algorithm, as well as "Russian peasant multiplication," an alternative multiplication algorithm

Making Change (in US Currency)

- While there are many ways to express a "make change" algorithm, we'll take an approach that illustrates how algorithms can be reused within other algorithms
- For this example, we focus on US currency: quarters, dimes, nickels, and pennies
- As an aside, think about whether you can:
  - Modify the algorithm to accommodate some other currency/coinage system
  - Modify the algorithm to work with *any* coinage system

Expressing "Make Change" in Human Terms

- As mentioned before, knowing how to *perform* an algorithm does not necessarily mean that we know how to *express* it in a manner that can be used by a machine
- Still, that doesn't mean we can't express this algorithm at all — it's just that human brains can manipulate and infer things that machines cannot (for now)
- In natural language, one might say: "Determine the number of quarters to use. Then, from what is left, determine the number of dimes. Do the same for the number of nickels and pennies."

- We may also express the algorithm *by example*: "Suppose you're making change for 67 cents. A maximum of 2 quarters 'fits' into 67 cents, for a total of 50 cents. With 17 cents remaining, a maximum of 1 dime leaves 7 cents. 1 nickel leaves 2 cents, which finally results in 2 pennies."
- People can handle these expressions of algorithms because we are capable of *inductive reasoning* — we can infer *general principles* either from *limited information* or *patterns and examples*
    - ◇ Given the phrase "determine the number to use," humans can choose the correct arithmetic operations that make this determination
    - ◇ Given the phrase "do the same thing as before," humans can figure out which steps form the "same thing," and whether these steps may vary slightly (by denomination, in this example)
    - ◇ Given a specific example such as 67 cents, humans can generalize to other amounts

The “Top-Down” Approach

- One approach to a make-change algorithm is to break it down into the individual coin counts
- If we know how to count the number of quarters, dimes, nickels, and pennies, respectively, within some amount, then the overall make-change algorithm can use these “sub-”algorithms to help find the answer
- We don’t worry about the details for these “sub-”algorithms until later — an approach that is known as *top-down* (since we start at the “top” and work our way “down” the overall algorithm)

We use indentation to indicate what steps are “within” other steps

*To eliminate ambiguity, we give **names** to items that we care about and make sure that we use them with absolute consistency; in our pseudocode, we use the word “let” to indicate that we’re giving something a name*

The “:=” symbol is used to “assign” a value to something that we’ve named

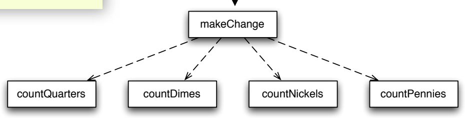
```
makeChange(amount)
   let currentAmount := amount
   let quarters := countQuarters(currentAmount)
   currentAmount := currentAmount – (25 × quarters)
   let dimes := countDimes(currentAmount)
   currentAmount := currentAmount – (10 × dimes)
   let nickels := countNickels(currentAmount)
   currentAmount := currentAmount – (5 × nickels)
   let pennies := countPennies(currentAmount)

   // Our answer consists of a list of four numbers, one for quarters,
   // dimes, nickels, and pennies, respectively.
   return [quarters, dimes, nickels, pennies]
```

*We sometimes rely on the existence of **built-in** operations: activities that we assume are known without further elaboration — in this case, we assume that subtraction (–) and multiplication (×) are available to us*

We use “return” to express an algorithm’s conclusion, delivering its result — in this case, a list of coin counts, enclosed in “[ ]” brackets and separated by commas

*Sometimes, it’s useful to give “notes” to ourselves (or others) in plain English — these are called **comments** and may show up here and there, indicated via the “//” symbol*

“Top-down,” see?



Going Down a Level

- Once we've convinced ourselves that the "top-level" make change algorithm should work *if* we knew how to count quarters, dimes, nickels, and pennies, then we "go down" a level by working on *those* algorithms next
- Note how the top-down approach allows us to break a bigger problem down into smaller chunks, *without* having to worry about those pieces right away
- Not a bad approach, if you think about it — both in terms of algorithms and with regard to some situations that we may encounter in real life

__Repetition__ is another key element of many algorithms; in pseudocode, we use "while" to indicate a conditional repetition — something that we repeat as long as some condition is true or false

Another "built-in" assumption: we assume that we can compare one value to another, using a combination of equality, greater than, or less than (and many more, where applicable)

```
countQuarters(amount)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= 25)
    coinCount := coinCount + 1
    currentAmount := currentAmount - 25
  return coinCount
```

Indentation again — this time, it tells us what part is repeated "while" currentAmount ≥ 25

Note how sometimes, our algorithms are determined by the operations that we presume to be "built-in" — for example, would this change in any way if we can do some kind of integer division?

Avoiding Redundancy

Using *countQuarters()* as a pattern, it isn't too hard to figure out the rest of the *count()* algorithms:

```
countDimes(amount)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= 10)
    coinCount := coinCount + 1
    currentAmount := currentAmount - 10
  return coinCount
```

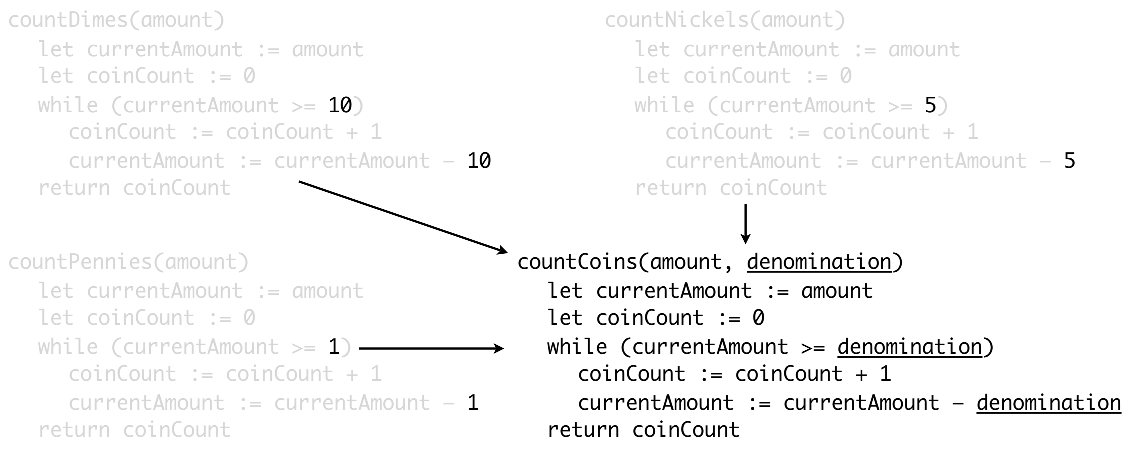
```
countNickels(amount)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= 5)
    coinCount := coinCount + 1
    currentAmount := currentAmount - 5
  return coinCount
```

```
countPennies(amount)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= 1)
    coinCount := coinCount + 1
    currentAmount := currentAmount - 1
  return coinCount
```

…but wait, this looks pretty repetitive!

If you're also thinking that countPennies() *looks like overkill, you would be right; for the moment, though, that isn't the point that we're making*

- Note how only the darker sections of each algorithm are different — the overall structure and sequence are otherwise the same
- After some staring, we realize that this difference is based on the *denomination* of the coin — so we can turn *that* into part of the algorithm's input!



```
countDimes(amount)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= 10)
    coinCount := coinCount + 1
    currentAmount := currentAmount - 10
  return coinCount
```

```
countNickels(amount)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= 5)
    coinCount := coinCount + 1
    currentAmount := currentAmount - 5
  return coinCount
```

```
countPennies(amount)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= 1)
    coinCount := coinCount + 1
    currentAmount := currentAmount - 1
  return coinCount
```

```
countCoins(amount, denomination)
  let currentAmount := amount
  let coinCount := 0
  while (currentAmount >= denomination)
    coinCount := coinCount + 1
    currentAmount := currentAmount - denomination
  return coinCount
```

Expanding to Other Currencies

At this point, our algorithm looks like this:

```
countCoins(amount, denomination)
   let currentAmount := amount
   let coinCount := 0
   while (currentAmount >= denomination)
      coinCount := coinCount + 1
      currentAmount := currentAmount − denomination
   return coinCount

makeChange(amount)
   let currentAmount := amount
   let quarters := countCoins(currentAmount, 25)
   currentAmount := currentAmount − (25 × quarters)
   let dimes := countCoins(currentAmount, 10)
   currentAmount := currentAmount − (10 × dimes)
   let nickels := countCoins(currentAmount, 5)
   currentAmount := currentAmount − (5 × nickels)
   let pennies := countCoins(currentAmount, 1)
   return [quarters, dimes, nickels, pennies]
```

Not bad, but it still assumes US currency; can we change this?

- As one last tweak, we observe that the notion of "US currency" is just a list of denominations: in this case, [25, 10, 5, 1]
- If our algorithm can accept any list of denominations, then we can handle *any* type of currency!

```
// countCoins() is the same as before.
makeChange(amount, denominationList)
   let currentAmount := amount
   let answerList := [ ]
   for each (denomination in denominationList)
      coinCount := countCoins(currentAmount, denomination)
      currentAmount := currentAmount − (denomination × coinCount)
      answerList[denomination] := coinCount
   return answerList
```

Expanding on our use of "[ ]" brackets to express lists, we show that we can put nothing in between those brackets to indicate an empty list

Next, we have a new type of repetition: we use a "for each" phrase to indicate that we're repeating some set of steps once for each element of the list

We re-use "[ ]" brackets to designate individual items in a list — in this case, we want the list of answers to correspond to the list of denominations

Russian Peasant Multiplication

- Now let's try the reverse — for *makeChange()*, we started with an algorithm that we know intuitively, and showed how it can be expressed in pseudocode
- This time, let's look at an algorithm that is already expressed in pseudocode, and see if we can follow its instructions to perform the desired computation
- The algorithm is for *Russian peasant multiplication* — it's the same multiplication that you know and love (or loathe?), but just done in a different way…and available in two versions…

◇ The version on the left builds a list of numbers to sum up in the end (making it easier to do by hand), while the version on the right adds the numbers up right away (making it easier to translate for a computer)…in the end, the result is the same — can you see why?

◇ Both versions use one more major tool in specifying algorithms: a **conditional** statement that does one thing if some condition is true, but does something else (or nothing at all) if that condition is false, indicated by an "if – then" or "if – then – else" construct

```
listRussianPeasantMultiply(factor1, factor2)
  if (factor1 > factor2) then
    let term1 := factor2
    let term2 := factor1
  else
    let term1 := factor1
    let term2 := factor2

  let addendList := [ ]
  while (term1 > 0)
    if (term1 is odd) then
      add term2 to addendList
    term1 := halveWithoutRemainder(term1)
    term2 := double(term2)

  let product := 0
  for each (number in addendList)
    product := product + number
  return product
```

```
russianPeasantMultiply(factor1, factor2)
  if (factor1 > factor2) then
    let term1 := factor2
    let term2 := factor1
  else
    let term1 := factor1
    let term2 := factor2

  let product := 0
  while (term1 > 0)
    if (term1 is odd) then
      product := product + term2
    term1 := halveWithoutRemainder(term1)
    term2 := double(term2)

  return product
```