Making Change

Making Change

Goal. Given currency coin denominations, e.g., {100, 25, 10, 5, 1} devise a method to pay **an integer** amount using the **fewest coins.**

Example: 34¢.

25 5 1 1 1 1

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Example:
$2.89 = 289¢.

100, 100, 25, 25, 25, 10, 1, 1, 1, 1

Greedy Algorithm

Cashier's algorithm. Use the maximal number of the largest denomination coin

```
x – amount to be changed
Sort coins denominations by value: c1 < c2 < … < cn.
S ← empty  ← coins selected
while (x > 0) {
   let k be largest integer such that ck ≤ x
   if (k == 0) # all ck > x
      return "no solution found"
   x ← x - ck
   append(S,k)
}
return S
```

Does this Greedy algorithm always work?

Greedy doesn't always work

1. Greedy fails changing **30 optimally** with coin set **{25, 10, 1}** as it produces [25,1,1,1,1,1] instead of [10,10,10]

2. Greedy fails changing **30 at all** with coin set **{25, 10}** even though there is a solution: [10,10,10]

3. But the Greedy algorithm works for US coin set
   Proof: number theory (canonical coin systems)

Different problem: number of ways to pay

Given a sorted coin set coins = $\{c_0, c_1, ..., c_{d-1}\}$ c_0 the smallest coin value, and c_{d-1} the largest coin value, and an amount M

how many different ways can M be paid?

One possible recursive either / or solution: go backwards through coins and choose to use the largest remaining coin or not

mkCh(n, c):
 # n: amount still to be paid
 # c: index of coins value currently considered
Base:
if c == 0, how many ways? (is there always a way ?)
Step:
if c>0
if largest coin cannot be used: consider coin_{c-1}
else: # it can be used
either use one coin_c and keep considering coin_c
or don't use coin_c and thus consider coin_{c-1}

5

Make change vs. knapsack

Recurrence:

ways(amount, i) =
1. Base case?
2. If amount < coin[i]: ways(i-1, amount)
3. Else: ways(amount-coin[i],i) + ways(amount, i-1)

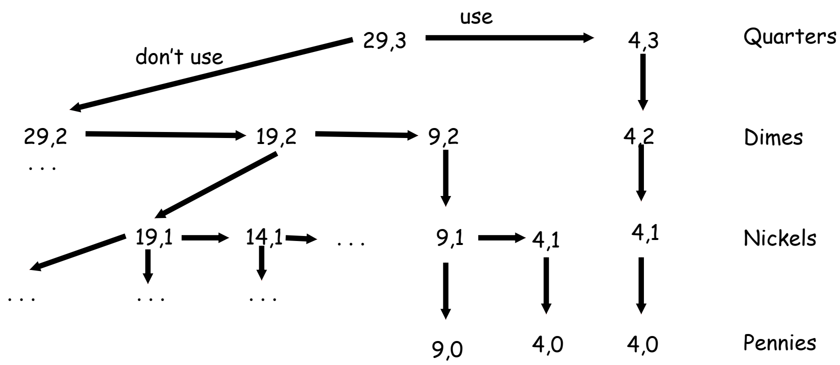
Making change is very similar to knapsack, but:

1. We take the sum, not the maximum, of the two options.
2. We must use the same coin value a number of times. How this is reflected in the recurrence?

6

Example of the recursive solution

coins = [1,5,10,25] M = 29

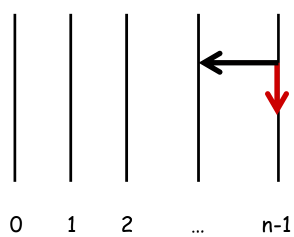


Complete this call tree

Making Change Dynamic Programming

Go through the state space bottom-up: i=0 to n-1

- select coin type
  - first 1 coin type, then 1&2, ..., finally all coin types
  - what does the first column look like?
- use solutions of smaller sub-problems to compute solutions of larger ones by storing previous values. Which values do you need to preserve?



In the recursive solution (DC) there are 2 (recursive) sub-problems. In the dynamic programming solution there are 2 reads:

don't use current coin ←

use current coin ↓

8

Programming Assignment

1. Write a recursive mkChange function based on the either or choices from slide 6, then turn it into a Dynamic Programming function.
   - Do you need a 2 D table here?

2. Determine the performance of the two algorithms. Later, in a written assignment, you will plot your data, and infer O complexity:
   - Recursive: count number of calls
   - Dynamic programming: count number of table reads

9