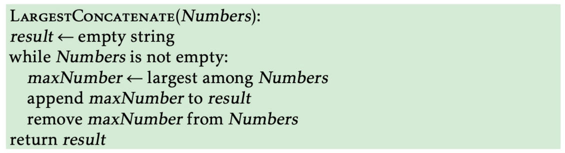
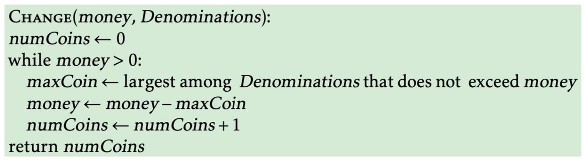
Week 3 Notes

Friday, January 27, 2023 1:10 PM

Main idea of Greedy Algorithms

- A Greedy algorithm builds a solution piece by piece each step of the way.
- 

```
LargestConcatenate(Numbers):
result ← empty string
while Numbers is not empty:
    maxNumber ← largest among Numbers
    append maxNumber to result
    remove maxNumber from Numbers
return result
```

- 

```
Change(money, Denominations):
numCoins ← 0
while money > 0:
    maxCoin ← largest among Denominations that does not exceed money
    money ← money − maxCoin
    numCoins ← numCoins + 1
return numCoins
```

Local Vs Global Optimum

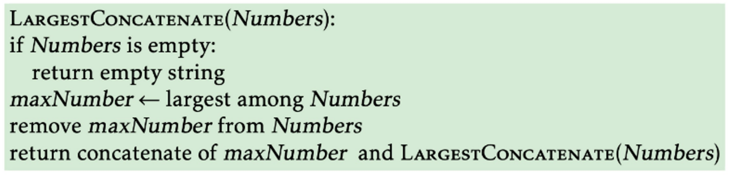
- If you use the same greedy strategy, then LargestConcatenate([2, 21]) returns 212 and Change(8, [1, 4, 6]) returns 3 because 8 = 6 + 1 + 1. But this strategy fails because the correct solutions are 221 (concatenating 2 with 21) and 2 because 8 = 4 + 4!
- Thus, in rare cases when a greedy strategy works, one should be able to prove its correctness: A priori, there should be no reason why a sequence of locally optimal moves leads to a global optimum!

Proving Correctness of Greedy Algorithms

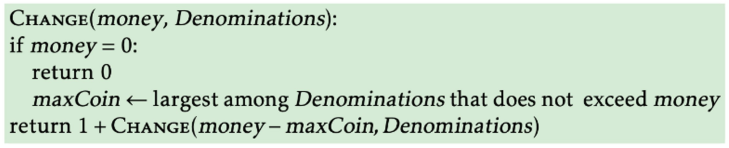
- At each step, a greedy algorithm restricts the search space by selecting a most "profitable" piece of a solution.
- instead of considering all possible concatenations, the LargestConcatenate algorithm only considers concatenations starting from the largest digit.
- Instead of all possible ways of changing money, the Change algorithm considers only the ones that include a coin with the largest denomination (that does not exceed money).

Implementation

- A greedy solution chooses the most profitable move and then continues to solve the remaining problem that usually has the same type as the initial one
- There are two natural implementations of this strategy:
    - iterative with a while loop or recursive
    - Below are their recursive variants.



```
LargestConcatenate(Numbers):
if Numbers is empty:
    return empty string
maxNumber ← largest among Numbers
remove maxNumber from Numbers
return concatenate of maxNumber and LargestConcatenate(Numbers)
```



```
Change(money, Denominations):
if money = 0:
    return 0
    maxCoin ← largest among Denominations that does not exceed money
return 1 + Change(money − maxCoin, Denominations)
```

Money Change

Money Change Problem
Compute the minimum number of coins needed to change the given value into coins with denominations 1, 5, and 10.

Input: An integer *money*.
Output: The minimum number of coins with denominations 1, 5, and 10 that changes *money*.

Solution: Use the Largest Denomination First

- While Money > 0, keep taking coins with the largest denomination that does not exceed money
-
```
Change(money):
numCoins ← 0
while money > 0:
   if money ≥ 10:
      money ← money − 10
   else if money ≥ 5:
      money ← money − 5
   else:
      money ← money − 1
   numCoins ← numCoins + 1
return numCoins
```
-
```
return ⌊money/10⌋ + ⌊(money mod 10)/5⌋ + (money mod 5)
```

Proving Correcting

- To prove that this greedy algorithms is correct, we show that taking a coin with the largest denomination is consistent with some optimal solution.
  - 1 <= money < 5 - D =1
  - 5 <= money < 10 - D = 5
  - 10 <= money - D=10

Maximizing the Value of the Loot Problem

Maximizing the Value of the Loot Problem
Find the maximal value of items that fit into the backpack.

Input: The capacity of a backpack W as well as the weights $(w_1,\ldots,w_n)$ and costs $(c_1,\ldots,c_n)$ of n different compounds.
Output: The maximum total value of items that fit into the backpack of the given capacity: i.e., the maximum value of $c_1 \cdot f_1 + \cdots + c_n \cdot f_n$ such that $w_1 \cdot f_1 + \cdots + w_n \cdot f_n \le W$ and $0 \le f_i \le 1$ for all i (f_i is the fraction of the i-th item taken to the backpack).

Solution: Take as Much of the Most Expensive Compound as You Can

- Take the maximum possible amount of the most expensive compound and iterate
- Stop when either there are no more compounds left or when the backpack is fully packed
-
```
MaximumLoot(W, Weight, Cost)
if W = 0 or Weight is empty:
   return 0
m ← the index of the most expensive item
amount ← min(Weight[m], W)
value ← Cost[m] · $\frac{amount}{Weight[m]}$
remove the m-th element from Weight and Cost
return value + MaximumLoot(W, Weight, Cost)
```

Summery
We go over examples that use greedy algorithms. Greedy algorithms are an algorithmic paradigm that builds the solution piece by piece to get the locally optimal choice with the hope of finding the global optimum. They typically are simple and easy to implement but come with the draw back of not always providing the globally optimum solution.