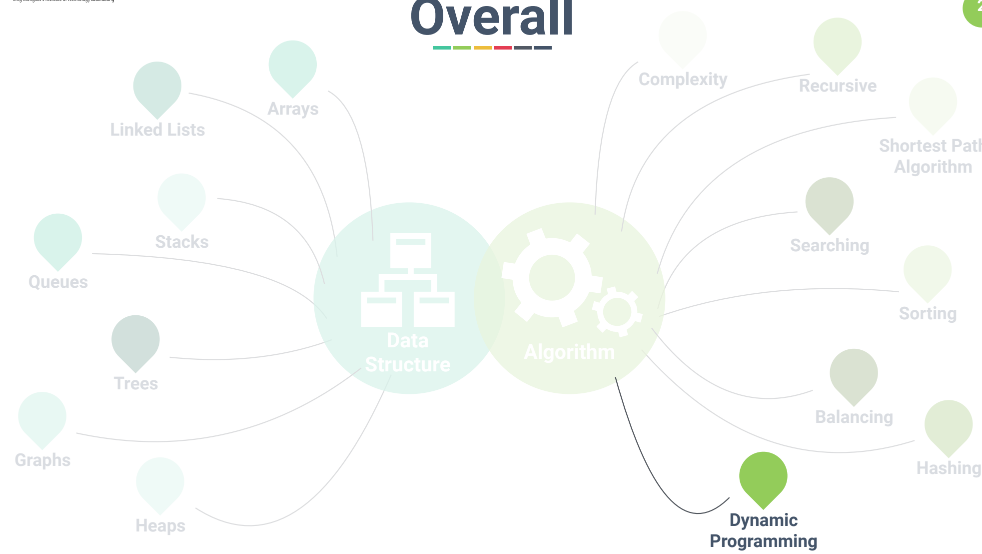
Chapter 12: Dynamic Programming Part 1

Dr. Sirasit Lochanachit

Overall



Outline

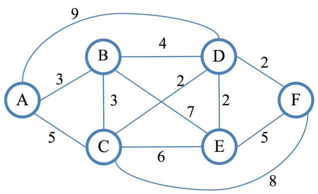
Optimisation Problem

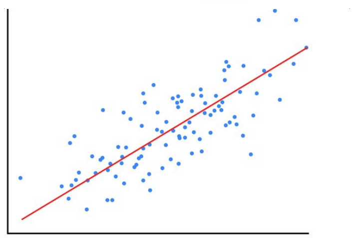
- Making Change using fewest coins Problem
  - Greedy Method
  - Greedy Method revised
  - Dynamic Programming Method

Introduction

- In an algorithm design, there is no one 'silver bullet' that is a cure for all computation problems.
- Different problems require the use of different kinds of techniques.
- A good programmer uses all these techniques based on the type of problem.

Optimisation



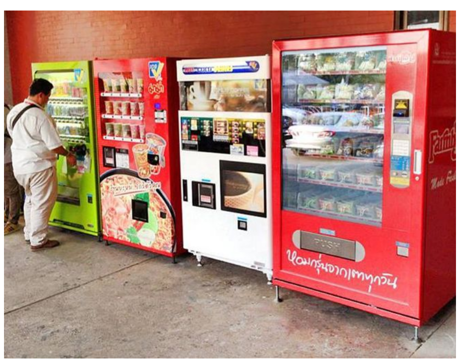


- Many problems involve optimisation of some value:
  - Find the shortest path between 2 points
  - Find the line that best fits a set of points
  - Find the smallest set of objects that satisfies some criteria
- Many strategies to solve these optimisation problems.
  - One of them is **Dynamic Programming.**

Dynamic Programming

- **Dynamic Programming (DP)** is a strategy similar to divide and conquer where a complicated problem is simplified by breaking it down into simpler sub-problems in a **recursive** manner.
- In divide and conquer, subproblems are independent of each other.
- In contrast, subproblems in DP are dependent of each other.

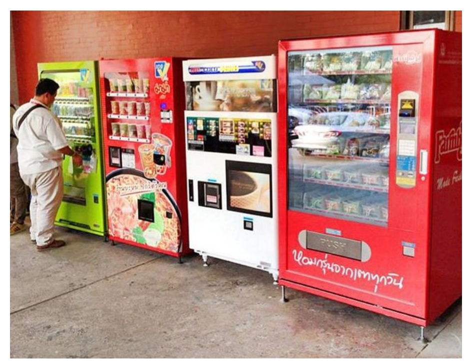
Optimisation Problem



Ref: http://www.thaismescenter.com/wp-content/uploads/2017/02/er10.jpg

- Suppose you are a programmer for a vending machine manufacturer.
- Your company wants to streamline effort by giving out the fewest possible coins in change for each transaction.
- For example, a customer puts in a 50 Baht bill to buy an item for 37 Baht.
- What is the smallest number of coins to make change?

Optimisation Problem



Ref: http://www.thaismescenter.com/wp-content/uploads/2017/02/er10.jpg

- How do we know that the answer is 3?
- We start with the largest coin and use as many of those as possible.
  - 10 Baht coin
- Then we go to the next second largest coin value and use as many of those as possible.
  - 2 Baht coin
  - Then 1 Baht coin

Greedy Method

- A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment.
- This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.
  - Keep in mind that local optimal is not always global optimal.

Greedy Method

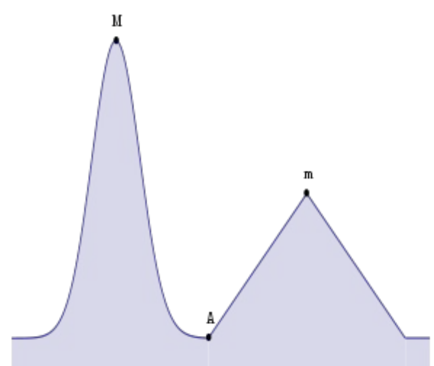
How do you decide which choice is optimal?

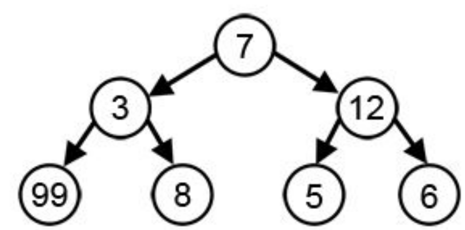
- Assume that you have an **objective function** that needs to be optimized (either maximized or minimized) at a given point.
- A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.
- The Greedy algorithm has only one shot to compute the optimal solution so that **it never goes back and reverses the decision**.

Greedy Method

- In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.
- Imagine there is a 7 Baht coin, a change for 14 Baht is needed.
  - Greedy method would find the solution to be 10 + 2 + 2, which are 3 coins
  - However, the optimal solution is two 7 Baht coins

Greedy Method





Making Change using Fewest Coins

A recursive solution is possible.

- Base case: If change is the same amount as the value of one of the coins, the answer is one coin.
- Recursive case: If the amount does not match, find the minimum of a coin as follows

$$\text{numCoins} = \min \begin{cases} 1 + \text{numCoins(originalAmount - 1)} \\ 1 + \text{numCoins(originalAmount - 2)} \\ 1 + \text{numCoins(originalAmount - 5)} \\ 1 + \text{numCoins(originalAmount - 10)} \end{cases}$$

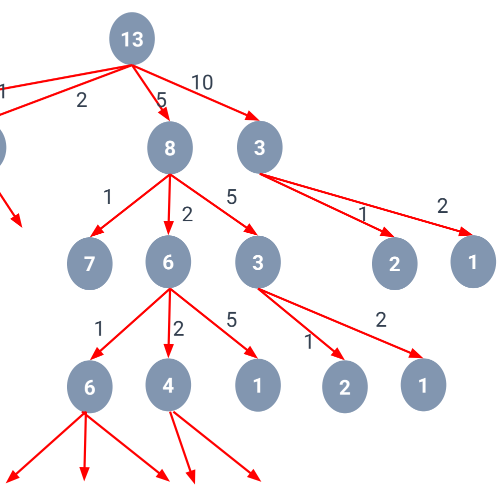
Making Change using Fewest Coins

Algorithm

```
def recMinCoin(coinValueList, change):
    minCoins = change
    if change in coinValueList:          #Base Case
        return 1
    else:                                #Recursive Case
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMinCoin(coinValueList, change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins
print(recMinCoin([1, 2, 5, 10], 13))
```

Efficient?

Making Change using Fewest Coins



Extremely inefficient

Each node corresponds to a call to recMinCoins.

Too many re-calculations.

Greedy Method Problems

- For example, the graph shows that the algorithm would recalculate the optimal number of coins to make change for 3 Baht > 1 times.
  - Wasting a lot of time and effort recalculating old results.
- Would be more efficient if the algorithm can remember some of the past results.
  - A simple solution is to store the results for the minimum of coins in a table for it to find later.
  - Before compute a minimum, the algorithm first check the table to see if a result is already known.
    - If exist already, then use the value directly from the table rather than computing.

Making Change using Fewest Coins

Algorithm with Table (Caching)

```
def recMinCoin(coinValueList, change, knownResults):
    minCoins = change
    if change in coinValueList:                    #Base Case
        knownResults[change] = 1
        return 1
    elif knownResults[change] > 0:#Case when the result in the table is found, so skip compute step
        return knownResults[change]
    else:                                          #Recursive Case
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMinCoin(coinValueList, change-i, knownResults)
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins
    return minCoins
print(recMinCoin([1, 2, 5, 10], 13, [0]*14))
```

Making Change using Fewest Coins

- The performance of the algorithm has improved by using a technique called “memoization” or “caching”.
- Is it a dynamic programming approach?

Greedy Choice Property

- Greedy method make whatever choice seems best at the moment and then solve the subproblems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.

Difference between Greedy and DP

- In other words, a greedy algorithm never reconsiders its choices.
- This is the main difference from dynamic programming, which is **exhaustive and is guaranteed** to find the solution.
- After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

Dynamic Programming Approach

- Dynamic Programming algorithm will start with making change for 1 Baht and systematically work its way up to the amount of change required.
- This guarantees that at each step of the algorithm, the minimum number of coins needed to make change for any smaller amount is known.

Dynamic Programming Approach

Change to make

1	2	3	4	5	6	7	8	9	10	11	12	13

Minimum number of coins needed to make change

Step of the Algorithm

1												
1	1											
1	1	2										
1	1	2	2									
1	1	2	2	1								
1	1	2	2	1	2							
						...						
1	1	2	2	1	2	2	3	3	1	2	2	
1	1	2	2	1	2	2	3	3	1	2	2	3

Dynamic Programming Approach

Change to make

1	2	3	4	5	6	7	8	9	10	11	12	13

Minimum number of coins needed to make change

1	1	2	2	1	2	2	3	3	1	2	2	3

3 Options to consider:

1. A Baht + the minimum number of coins to make change for 13 - 1 = 12 Baht (2)
2. A 2 Baht + the minimum number of coins to make change for 13 - 2 = 11 Baht (2)
3. A 5 Baht + the minimum number of coins to make change for 13 - 5 = 8 Baht (3)
4. A 10 Baht + the minimum number of coins to make change for 13 - 10 = 3 Baht (2)

Making Change using Fewest Coins

Algorithm (Dynamic Programming)

```
def recMinCoin_DP(coinValueList, change, minCoins):
    for bahts in range(change + 1):
        coinCount = bahts
        for j in [c for c in coinValueList if c <= bahts]:
            if minCoins[bahts - j] +1 < coinCount:
                cointCount = minCoins[bahts - j] + 1
        minCoins[bahts] = coinCount
    return minCoins[bahts]

print(recMinCoin([1, 2, 5, 10], 13, [0]*14))
```

No Recursive

Making Change using Fewest Coins

- minCoins is a list of the minimum number of coins needed to make change for each value.
- minCoins will contain the solution for all values from 0 to the value of change.
- Recursive solution is not always the best or most efficient solution.
- Although this algorithm provides the minimum number of coins, it does not tell what coin to change since it does not keep track of the coins used.

Making Change using Fewest Coins

- Solution: Remember the last coin added in the minCoins table

Algorithm with tracking (Dynamic Programming)

```
def recMinCoin_DP(coinValueList, change, minCoins, coinsUsed):
    for bahts in range(1, change + 1):
        coinCount = bahts
        newCoin = 1
        for j in [c for c in coinValueList if c <= bahts]:
            if minCoins[bahts - j] +1 < coinCount:
                cointCount = minCoins[bahts - j] + 1
                newCoin = j
        minCoins[bahts] = coinCount
        coinsUsed[bahts] = newCoin
    return minCoins[bahts]
print(recMinCoin([1, 2, 5, 10], 13, [0]*14, [0]*14))
```

Summary

- Dynamic Programming is mainly an optimization over plain recursion.
- Whenever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.
- This simple optimization reduces time complexities from exponential (2^n) to polynomial (n^2) or linear (n).

Fibonacci Number


```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)        # 2^n (Exponential)

def fibonacci_DP(n):
    f[0] = 0
    f[1] = 1
    for i in range(2, n+1)
        f[i] = f[i-1] + f[i-2]
    return f[i]                                        # n (Linear)
```