アルゴリズムの設計と解析

教授:　黄　潤和　（W4022）
rhuang@hosei.ac.jp

SA:　広野　史明（A4/A10）
fumiaki.hirono.5k@stu.hosei.ac.jp

- 高度なアルゴリズム設計・解析手法
  - 分割統治法　　Divide and Conquer
  - 動的計画法　　Dynamic Programming

クイックソートを通して，分割統治法の概念を理解する

L3. 動的計画法
Dynamic Programming

What is dynamic programming?

D*ynamic Programming* is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub-instances. Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS

動的計画法（どうてきけいかくほう、英: Dynamic Programming, DP）は、計算機科学の分野において、アルゴリズムの分類の1つである。対象となる問題を複数の部分問題に分割し、部分問題の計算結果を記録しながら解いていく手法を総称してこう呼ぶ。

ボトムアップである（つまり、部分問題を解き終わるまで問題全体に手を出してはいけない）

Main idea?

1. set up a recurrence relating a solution to a larger instance to solutions of some smaller instance
2. solve smaller instances once
3. record solutions in a table
4. extract solution to the initial instance from that table

直接計算すると大きな時間がかかってしまう問題に対し、途中の計算結果をうまく再利用することで計算効率を上げる手法のこと。

- 「途中の計算結果を再利用」＝「同じ計算をしない」ということ
- 難しいように見えて考え方自体は単純

Some examples

- 部分和問題 - Fibonacci numbers
- コイン両替問題 – counting coins
- 最長増加部分列
- 連鎖行列積
- 巡回セールスマン

The Fibonacci numbers problem

Example: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n-2) =$
$F(n-1) =$
$F(n) = F(n-1) + F(n-2)$

0	1	1	. . .	$F(n-2)$	$F(n-1)$	$F(n)$

Efficiency:
- time n
- space n

Q: What if we solve it recursively?

Example: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- **Computing the n^{th} Fibonacci number recursively (top-down):**

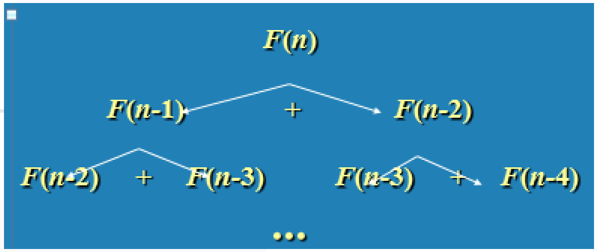
$$F(n)$$

$$F(n\text{-}1) \quad + \quad F(n\text{-}2)$$

$$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$$

...

A naïve implementation of a function




```python
def fib_recursive(n):
    if n < 2: return 1
    return fib_recursive(n - 1) + fib_recursive(n - 2)
```

Below is one of the execution image

```
  fib(5)
= fib(4) + fib(3)
= (fib(3) + fib(2)) + (fib(2) + fib(1))
= ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
= (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
```

The time complexity is $O(2^n)$

このように最終的に fib(0) と fib(1) の呼び出しに収束し、fib(0) と fib(1) の呼び出し回数の和が結果の値となる。この方法を用いたフィボナッチ数列の計算量は $O(2^n)$ の指数関数時間となる。

Q: Efficiency:

- time N?

- space N?

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$

$F(1) = 1$

$F(2) = 1+0 = 1$

…

$F(n\text{-}2) =$

$F(n\text{-}1) =$

$F(n) = F(n\text{-}1) + F(n\text{-}2)$

If we use dynamic programming (bottom-up)

We calculate f(n-2) and f(n-1), save and store the results and them calculate f(n)
This bottom-up approach method uses O(n) time since it contains a loop that repeats n − 1 times, but it only takes constant (O(1)) space.

下記のフィボナッチ数列を表示するプログラムは、動的計画法の具体的な例らしい。

```c
int fib(unsigned int n) {
    int memo[1000] = {0, 1}, i;
    for (i = 2; i <= n; i++) {
        memo[i] = memo[i - 1] + memo[i - 2];
    }
    return memo[n];
}
```

Python version

```python
def fib(n):
    memo = [0] * n
    memo[0:2] = [0, 1]
    for i in range(2, n):
        memo[i] = memo[i - 1] + memo[i - 2]
    return memo[0:n]
print(fib(100))
```

The coin change problem

Work in class:

Please find out
- Japanese coin types
- US coin types?

Please find out

- Japanese coin types

  1¥, 5¥, 10¥, 50¥, 100¥, 500¥

- US coin types?

  5 ¢, 10 ¢, 25 ¢, 50 ¢, 1$

  enough coin types?

  →try to find the minimum number of coin types

  for example, 31¢, 61¢

To find the minimum number of US coins to make any amount

Try to count 31c

?

Try to count 63c

?

Count coins – the minimum number

To find the minimum number of US coins to make any amount

?

The greedy method always works

- At each step, just choose the largest coin that does not overshoot the desired amount: 31¢=25+?

- **The greedy method would not work** if we did not have 5¢ coins
  - For 31 cents, the greedy method gives seven coins (25+1+1+1+1+1+1), but we can do it with four (10+10+10+1)
- **The greedy method also would not work** if we had a 21¢ coin
  - For 63 cents, the greedy method gives six coins (25+25+10+1+1+1), but we can do it with three (21+21+21)

? How can we find

the minimum number of coins for any given coin set?

Coin set for examples

- For the following examples, we will assume coins in the following denominations:
  1¢   5¢   10¢   21¢   25¢
- We'll use 63¢ as our goal

Coin set for examples

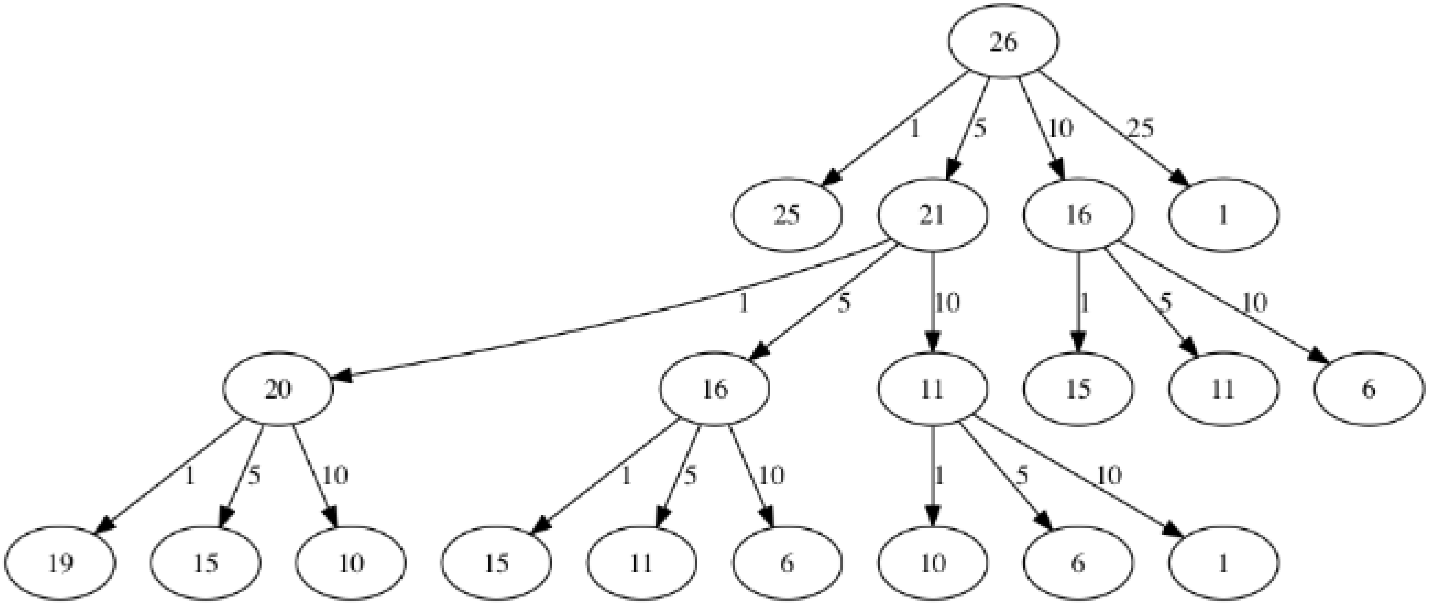
- For the following examples, we will assume coins in the following denominations:
  1¢ 5¢ 10¢ 21¢ 25¢
- We'll use 63¢ as our goal
  (work in class: Everyone thinks about it,
- how to solve it?)

A simple solution

- We always need a 1¢ coin, otherwise no solution exists for making one cent
- To make K cents:
  - If there is a K-cent coin, then that one coin is the minimum
  - Otherwise, for each value i < K,
    - Find the minimum number of coins needed to make i cents
    - Find the minimum number of coins needed to make K - i cents
  - Choose the i that minimizes this sum
- This algorithm can be viewed as divide-and-conquer, or as brute force (by exhaustion, a method of mathematical proof)
  - This solution is very recursive
  - It requires exponential work
  - It is *infeasible* to solve for 63¢

Another solution

- We can reduce the problem recursively by choosing the first coin, and solving for the amount that is left
- For 63¢:
  - One 1¢ coin plus the best solution for 62¢
  - One 5¢ coin plus the best solution for 58¢
  - One 10¢ coin plus the best solution for 53¢
  - One 21¢ coin plus the best solution for 42¢
  - One 25¢ coin plus the best solution for 38¢
- Choose the best solution from among the 5 given above
- Instead of solving 62 recursive problems, we solve 5 (62, 58, 53, 42, 38) using 1, 5, 10, 21, 25
- This is still a very expensive algorithm



Work in class:

Refer to the above, to draw the case of 63 using 1¢ 5¢ 10¢ 21¢ 25¢

A dynamic programming solution

- Idea: Solve first for one cent, then two cents, then three cents, etc., up to the desired amount
  - *Save each answer in an array !*
- For each new amount N, compute all the possible pairs of previous answers which sum to N
  - For example, to find the solution for 13¢,
    - First, solve for all of 1¢, 2¢, 3¢, ..., 12¢
    - Next, choose the best solution among:
      - Solution for 1¢ + solution for 12¢
      - Solution for 2¢ + solution for 11¢
      - Solution for 3¢ + solution for 10¢
      - Solution for 4¢ + solution for 9¢
      - Solution for 5¢ + solution for 8¢
      - Solution for 6¢ + solution for 7¢

Example

- Suppose coins are 1¢, 3¢, and 4¢
  - There's only one way to make 1¢ (one coin)
  - To make 2¢, try 1¢+1¢ (one coin + one coin = 2 coins)
  - To make 3¢, just use the 3¢ coin (one coin)
  - To make 4¢, just use the 4¢ coin (one coin)
  - To make 5¢, try
    - 1¢ + 4¢ (1 coin + 1 coin = 2 coins)
    - 2¢ + 3¢ (2 coins + 1 coin = 3 coins)
    - The first solution is better, so best solution is 2 coins
  - To make 6¢, try
    - 1¢ + 5¢ (1 coin + 2 coins = 3 coins)
    - 2¢ + 4¢ (2 coins + 1 coin = 3 coins)
    - 3¢ + 3¢ (1 coin + 1 coin = 2 coins) – best solution
  - Etc.

In Python

```
def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)

    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)

main()
```

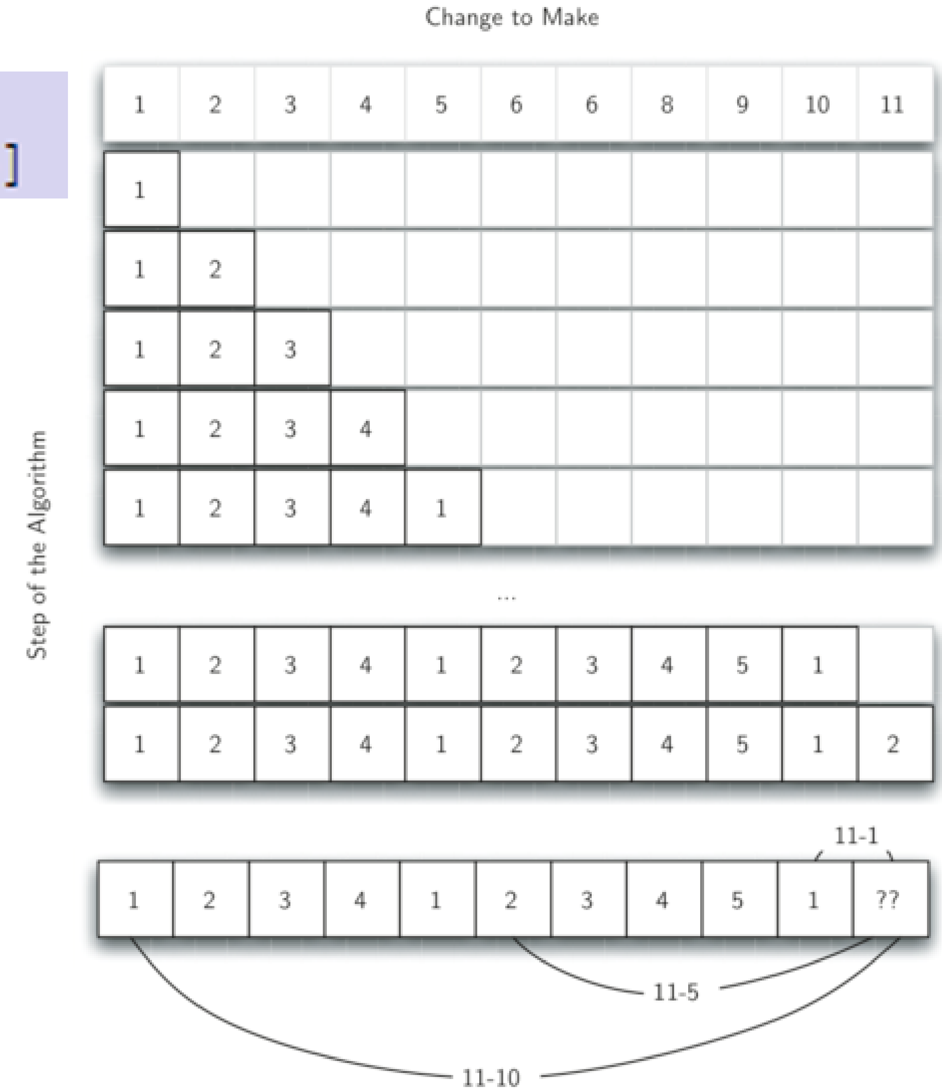
In Python (continue…)

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
   for cents in range(change+1):
      coinCount = cents
      newCoin = 1
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
               newCoin = j
      minCoins[cents] = coinCount
      coinsUsed[cents] = newCoin
   return minCoins[change]

def printCoins(coinsUsed,change):
   coin = change
   while coin > 0:
      thisCoin = coinsUsed[coin]
      print(thisCoin)
      coin = coin - thisCoin
```

```
amnt = 63
clist = [1,5,10,21,25]
```

Change to make for 11




```
j= 1 cents= 11
j= 5 cents= 11
j= 10 cents= 11
coinCount 2
coinsUsed [1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

“j” is the coin types can be used
e,g, coin 7 uses 1, 5 (1+1+5)

```
Making change for 63 requires
j= 1 cents= 1
j= 1 cents= 2
j= 1 cents= 3
j= 1 cents= 4
j= 1 cents= 5
j= 5 cents= 5
j= 1 cents= 6
j= 5 cents= 6
j= 1 cents= 7
j= 5 cents= 7
j= 1 cents= 8
j= 5 cents= 8
j= 1 cents= 9
j= 5 cents= 9
j= 1 cents= 10
j= 5 cents= 10
j= 10 cents= 10
j= 1 cents= 11
j= 5 cents= 11
j= 10 cents= 11
j= 1 cents= 12
j= 5 cents= 12
j= 10 cents= 12
j= 1 cents= 13
j= 5 cents= 13
j= 10 cents= 13
```

x2

```
j= 1 cents= 14
j= 5 cents= 14
j= 10 cents= 14
j= 1 cents= 15
j= 5 cents= 15
j= 10 cents= 15
j= 1 cents= 16
j= 5 cents= 16
j= 10 cents= 16
j= 1 cents= 17
j= 5 cents= 17
j= 10 cents= 17
j= 1 cents= 18
j= 5 cents= 18
j= 10 cents= 18
j= 1 cents= 19
j= 5 cents= 19
j= 10 cents= 19
j= 1 cents= 20
j= 5 cents= 20
j= 10 cents= 20
j= 1 cents= 21
j= 5 cents= 21
j= 10 cents= 21
j= 21 cents= 21
```

```
j= 1 cents= 22
j= 5 cents= 22
j= 10 cents= 22
j= 21 cents= 22
j= 1 cents= 23
j= 5 cents= 23
j= 10 cents= 23
j= 21 cents= 23
j= 1 cents= 24
j= 5 cents= 24
j= 10 cents= 24
j= 21 cents= 24
j= 1 cents= 25
j= 5 cents= 25
j= 10 cents= 25
j= 21 cents= 25
j= 25 cents= 25
j= 1 cents= 26
j= 5 cents= 26
j= 10 cents= 26
j= 21 cents= 26
j= 25 cents= 26
j= 1 cents= 27
j= 5 cents= 27
j= 10 cents= 27
j= 21 cents= 27
j= 25 cents= 27
j= 1 cents= 28
j= 5 cents= 28
j= 10 cents= 28
j= 21 cents= 28
j= 25 cents= 28
```

```
j= 1 cents= 29
j= 5 cents= 29
j= 10 cents= 29
j= 21 cents= 29
j= 25 cents= 29
j= 1 cents= 30
j= 5 cents= 30
j= 10 cents= 30
j= 21 cents= 30
j= 25 cents= 30
j= 1 cents= 31
j= 5 cents= 31
j= 10 cents= 31
j= 21 cents= 31
j= 25 cents= 31
j= 1 cents= 32
j= 5 cents= 32
j= 10 cents= 32
j= 21 cents= 32
j= 25 cents= 32
j= 1 cents= 33
j= 5 cents= 33
j= 10 cents= 33
j= 21 cents= 33
j= 25 cents= 33
j= 1 cents= 34
j= 5 cents= 34
j= 10 cents= 34
j= 21 cents= 34
j= 25 cents= 34
j= 1 cents= 35
j= 5 cents= 35
j= 10 cents= 35
j= 21 cents= 35
j= 25 cents= 35
```

......

```
j= 1 cents= 58
j= 5 cents= 58
j= 10 cents= 58
j= 21 cents= 58
j= 25 cents= 58
j= 1 cents= 59
j= 5 cents= 59
j= 10 cents= 59
j= 21 cents= 59
j= 25 cents= 59
j= 1 cents= 60
j= 5 cents= 60
j= 10 cents= 60
j= 21 cents= 60
j= 25 cents= 60
j= 1 cents= 61
j= 5 cents= 61
j= 10 cents= 61
j= 21 cents= 61
j= 25 cents= 61
j= 1 cents= 62
j= 5 cents= 62
j= 10 cents= 62
j= 21 cents= 62
j= 25 cents= 62
j= 1 cents= 63
j= 5 cents= 63
j= 10 cents= 63
j= 21 cents= 63
j= 25 cents= 63
```

```
3 coins
They are:
21
21
21
The used list is as follows:
[1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 21, 1, 1,
1, 25, 1, 1, 1, 1, 5, 10, 1, 1, 1, 10, 1, 1, 1, 1, 5, 10, 21, 1, 1, 10, 21,
1, 1, 1, 25, 1, 10, 1, 1, 5, 10, 1, 1, 1, 10, 1, 10, 21]
```

```
j= 1 cents= 60
j= 5 cents= 60
j= 10 cents= 60
j= 21 cents= 60
j= 25 cents= 60
coinCount 3
j= 1 cents= 61
j= 5 cents= 61
j= 10 cents= 61
j= 21 cents= 61
j= 25 cents= 61
coinCount 4
j= 1 cents= 62
j= 5 cents= 62
j= 10 cents= 62
j= 21 cents= 62
j= 25 cents= 62
coinCount 4
j= 1 cents= 63
j= 5 cents= 63
j= 10 cents= 63
j= 21 cents= 63
j= 25 cents= 63
coinCount 3
```

Point:

Use what are in the coin used before

```
 RESTART: C:¥Huang-Start2011¥teaching¥Algorithms¥algorithm-2015¥2017¥lecture
-notes¥dp-algo¥dp63-coins.py
Making change for 63 requires
coinCount 0
coinsUsed [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j= 1 cents= 1
coinCount 1
coinsUsed [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j= 1 cents= 2
coinCount 2
coinsUsed [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j= 1 cents= 3
coinCount 3
coinsUsed [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j= 1 cents= 4
coinCount 4
coinsUsed [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j= 1 cents= 5
j= 5 cents= 5
coinCount 1
coinsUsed [1, 1, 1, 1, 1, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j= 1 cents= 6
j= 5 cents= 6
coinCount 2
coinsUsed [1, 1, 1, 1, 1, 5, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
j= 1 cents= 7
j= 5 cents= 7
coinCount 3
coinsUsed [1, 1, 1, 1, 1, 5, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

How good is the algorithm?

- The first algorithm is recursive, with a branching factor of up to 62
  - Possibly the average branching factor is somewhere around half of that (31)
  - The algorithm takes exponential time, with a large base
- The second algorithm is much better—it has a branching factor of 5
  - This is exponential time, with base 5
- The dynamic programming algorithm is O(N*K), where N is the desired amount and K is the number of different kinds of coins

http://www.geocities.jp/m_hiroi/light/pyalgo23.html

http://ailaby.com/dynamic/

Other problems

Knapsack problem ナップザック問題

work in class

資料を調べてください

All-pairs shortest paths problem

Optimal Binary Search Trees

Comparison with divide-and-conquer

- Divide-and-conquer algorithms split a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem
  - Example: Quicksort
  - Example: Mergesort
  - Example: Binary search
- Divide-and-conquer algorithms can be thought of as top-down algorithms
- In contrast, a dynamic programming algorithm proceeds by solving small problems, remembering the results, then combining them to find the solution to larger problems
- Dynamic programming can be thought of as bottom-up

Exercises

Ex 3.1

Understand the dynamic programming approach to solve the coin problem and other problems.

Ex 3.2

Divide-and-conquer is a top-down technique while dynamic programming is a bottom-up technical. Both can be applied to solve coin change problem.

3.2.1 Please run dynamic program in in Python to solve coin 63 cents problem.

3.2.2 Please make Divide-and-conquer approach to solve the coin change problem, in Python, please refer to next three pages.

3.2.3 Compare their performance to see which is faster.

The divide-and-conquer approach

The key observation is that we can split the potential solutions into two disjoint classes, those solutions that use c_0 at least once and those that do not:

  - When a is zero then we need no coins.
  - When a is non-zero and we have no coins to offer change with – in this case the answer should be infinity. In our implementation we just use `Integer.MAX_VALUE`.

- Guarantee that the solution does not loop endlessly, that is, that the basis cases are evenually reached. For this, consider the parameter $(a + k)$. We can see that it always decreases for either of the steps (use c_0 or not), and that it cannot decrease for ever without reaching one of the basis cases.

The class `Change` below implements this solution in Java. The values of a coin set are stored in an array `coins`. For instance, the UK set corresponds to `coins[0]=100`, `coins[1]=50`,`coins[2]=20`, `coins[3]=10`, `coins[4]=5`,`coins[5]=2`, `coins[6]=1`. The method `change()` accepts an amount and a coin set as input and returns the minimal number of coins whose values add up to the amount.

- When a is zero then we need no coins.
- When a is non-zero and we have no coins to offer change with – in this case the answer should be infinity. In our implementation we just use `Integer.MAX_VALUE`.

- Guarantee that the solution does not loop endlessly, that is, that the basis cases are evenually reached. For this, consider the parameter $(a+k)$. We can see that it always decreases for either of the steps (use c_0 or not), and that it cannot decrease for ever without reaching one of the basis cases.

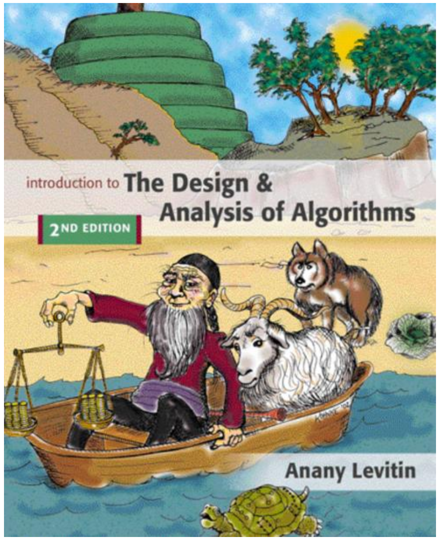
The class `Change` below implements this solution in Java. The values of a coin set are stored in an array `coins`. For instance, the UK set corresponds to `coins[0]=100`, `coins[1]=50`,`coins[2]=20`, `coins[3]=10`, `coins[4]=5`,`coins[5]=2`, `coins[6]=1`. The method `change()` accepts an amount and a coin set as input and returns the minimal number of coins whose values add up to the amount.

Java source code

```
class Change {
  private static int[] c;

  public static int change(int amount, int[] coins){
     c = coins;
     return change(amount,0);
  }
  private static int change(int amount, int j) {
    if (amount == 0) return(0);
    if (j == c.length) return(Integer.MAX_VALUE);
    if (amount < c[j]) return(change(amount,j+1));
    else {
      int c1 = change(amount,j+1);
      int c2 = 1 + change(amount-c[j],j);
      if (c1 < c2) return(c1);
      else return(c2);
    }
  }
}
```

References:



http://interactivepython.org/courselib/static/pythonds/Recursion/DynamicoPrgramming.html#lst-change2

https://www.cis.upenn.edu/
(30-dynamic-programming.ppt)

https://github.com/OSU-CS-325/Project_Two_Coin_Change