CS2220: Introduction to Computational Biology

Dynamic Programming Essence

Wong Limsoon

Outline

Dynamic programming in a nutshell

Knapsack problem

“Giving change” problem

Space-saving tricks in dynamic programming

Dynamic programming in a nutshell

Solve problems by breaking them down into overlapping subproblems

Solve each subproblem once

Reuse those solutions to build up the answer to the full problem

Two strategies

Top-down via memorization: Write a recursive algo and store results of subproblems in a cache so you don't recompute them

Bottom-up via tabulation: Start from the smallest subproblems and build up iteratively to the final answer

A helpful video

https://www.youtube.com/watch?v=piAlsJySUGE

Please watch it yourself at home

Packing for maximum benefits

Select items to maximize total value without exceeding a bag's capacity

Knapsack problem

Each item that can go into the knapsack has a size and a benefit

The knapsack has a certain capacity

What should go into the knapsack to maximize the total benefit?

An intuitive solution?

To fill a w-size knapsack, we must start by adding some item

If we add item j of size w_j, we end up with a knapsack k' of size $w - w_j$ to fill

$$g(w) = \max_j \{ b_j + g(w - w_j) \}$$

where w_j and b_j are size and benefit for item j

g(w) is max benefit that can be gained from a w-size knapsack

Source: http://mat.gsia.cmu.edu/classes/dynamic/node6.html

Exercise

Does g(w) produce the optimal benefit? Prove it

To fill a w-size knapsack, we must start by adding some item

If we add item j of size w_j, we end up with a knapsack k' of size $w - w_j$ to fill

$$g(w) = \max_j \{b_j + g(w - w_j)\}$$

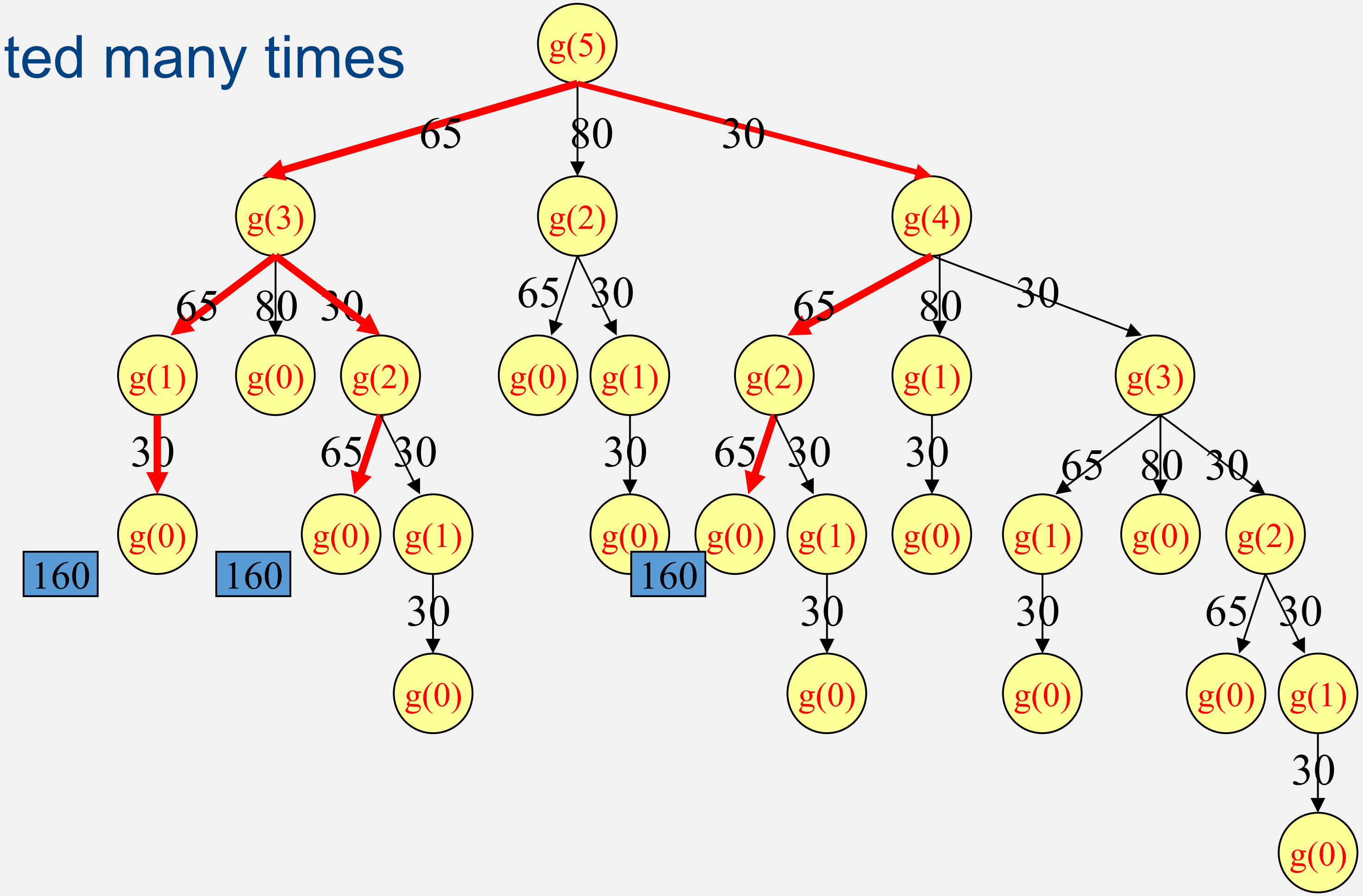
where w_j and b_j are size and benefit for item j
g(w) is max benefit that can be gained from a w-size knapsack

Direct recursive evaluation is inefficient

g(1), g(2), g(3) are computed many times

Item (j)	Weight (w_j)	Benefit(b_j)
1	2	65
2	3	80
3	1	30

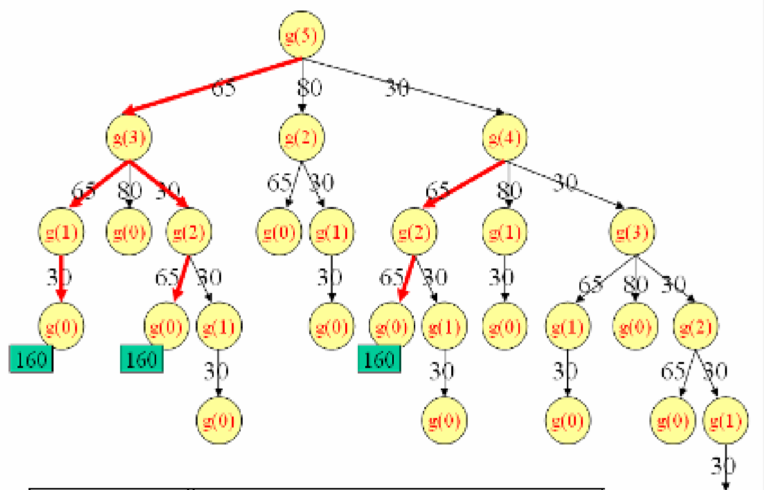
$$g(w) = \max_j \{b_j + g(w - w_j)\}$$



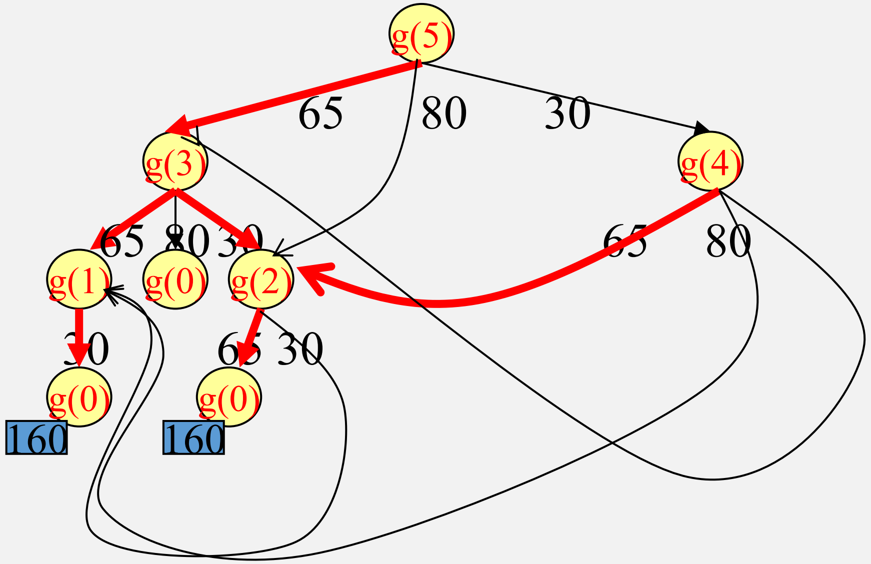
“Memoize” to avoid recomputation

$$g(w) = \max_j \{ b_j + g(w - w_j) \}$$

```
int s[]; s[0] := 0;
g’(w) = if s[w] is defined
    then return s[w];
    else {
        s[w] := max_j{b_j + g’(w – w_j)};
        return s[w]; }
```



Item (j)	Weight (w_j)	Benefit(b_j)
1	2	65
2	3	80
3	1	30



Exercise

What is the time and space complexity of the memorized solution?

```
int s[]; s[0] := 0;
g'(w) = if s[w] is defined
          then return s[w];
          else {
               s[w] := max_j{b_j + g'(w – w_j)};
               return s[w]; }
```

$$s[w] := \max_j\{b_j + g'(w - w_j)\};$$

Exercise

In what order do s[0], s[1], … get defined?

```
int s[]; s[0] := 0;
g'(w) = if s[w] is defined
            then return s[w];
           else {
                     s[w] := max_j{b_j + g'(w – w_j)};
                      return s[w]; }
```

Remove recursion: Dynamic programming

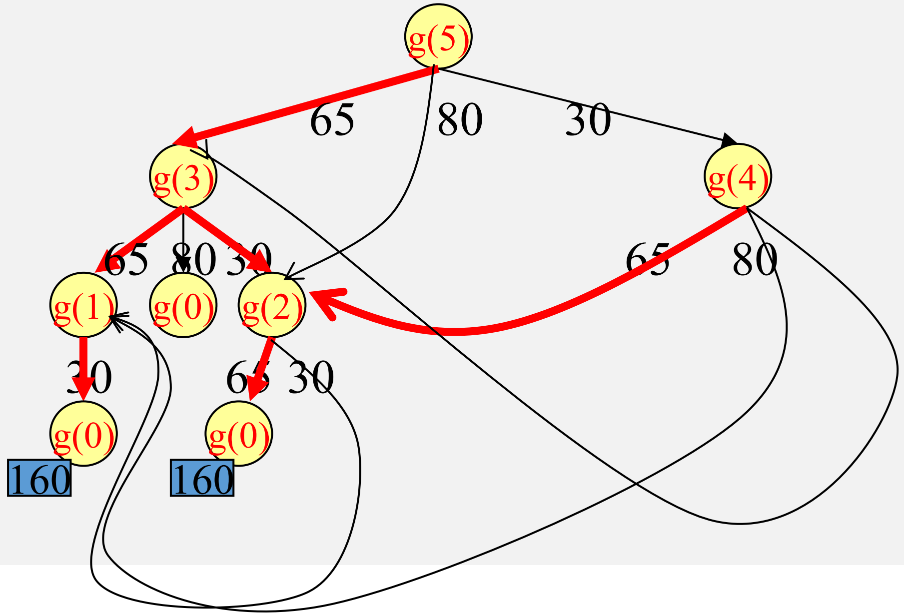
Item (j)	Weight (w_j)	Benefit(b_j)
1	2	65
2	3	80
3	1	30

$$g(w) = \max_j \{b_j + g(w - w_j)\}$$

```
int s[]; s[0] := 0;
g'(w) = if s[w] is defined
    then return s[w];
    else {
        s[w] := max_j{b_j + g'(w – w_j)};
        return s[w]; }
```

```
int s[]; s[0] := 0; s[1] := 30;
s[2] := 65; s[3] = 95;
for i := 4 .. w do
    s[i] := max_j{b_j + s[i – w_j]};
 return s[w];
```

g(0) = 0
g(1) = 30, item 3
g(2) = max{65 + g(0) =65, 30 + g(1) = 60} = 65, item 1
g(3) = max{65 + g(1) = 95, 80 + g(0) = 80, 30 + g(2) = 95} = 95, item 1/3
g(4) = max{65 + g(2) = 130, 80 + g(1) = 110, 30 + g(3) = 125} = 130, item 1/1
g(5) = max{65 + g(3) = 160, 80 + g(2) = 145, 30 + g(4) = 160} = 160, item 1/1/3



How to give change efficiently

Determine the fewest coins needed to make a given amount

Giving change

What algorithm would you propose to provide as few coins as possible when making change for a given set of denominations?

Roman Republic Denominations

(120, 40, 30, 24, 20, 10, 5, 4, 1)

A greedy solution

1. Given an amount *money*
2. Choose the largest *coin* with denomination less than or equal to *money*
3. Subtract *coin* from *money*.
4. If *money* = 0, STOP
5. Set *money* = *money* – *coin* and return to step 2

Does this greedy solution always give a change with the fewest coins?

The greedy solution is suboptimal

Roman Republic Denominations

(120, 40, 30, 24, 20, 10, 5, 4, 1)

When changing 48 Roman denarii, it would suggest five coins:

48 = 40 + 5 + 1 + 1 + 1

But we can make change with just two coins:

48 = 24 + 24

Looking for a better solution

Let MinNumCoins(*money*) denote the minimum number of coins needed to change an amount *money* for a collection of coin denominations

Is there anything that you can do to simplify the computation of MinNumCoins(76)?

Some observation and insight

Say that you need to change 76 denarii
you only have three denominations of coins: (5, 4, 1)

A minimum collection of coins totaling 76 denarii must be one of these:

A minimal collection of coins totaling 75 denarii, plus a 1-denarius coin

A minimal collection of coins totaling 72 denarii, plus a 4-denarius coin

A minimal collection of coins totaling 71 denarii, plus a 5-denarius coin

Intuitive and recursive definition of MinNumCoins

$$\text{MinNumCoins}(money) = \min \begin{cases} \text{MinNumCoins}(money - coin_1) + 1 \\ \vdots \\ \text{MinNumCoins}(money - coin_d) + 1 \end{cases}$$

Recurrence relation

An expression for a function f(x) in terms of values of f(y) where y < x

A recursive solution

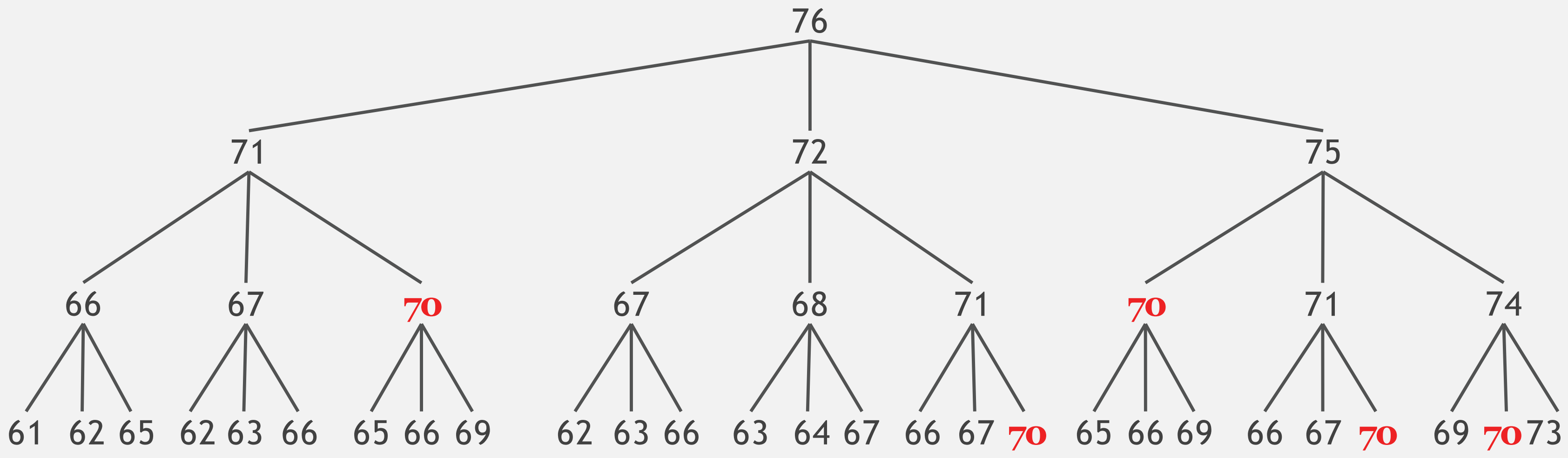
MinNumCoins(*money*) =

1. If *money* = 0, return 0
2. For each $coin_j \leq money$,
   set n_j = MinNumCoins($money - coin_j$)
3. Return 1 + min { n_j }

Does this solution always give a change with the fewest coins?

Is it computationally efficient?

Recursive calls grow exponentially



For *money* = 76 and *coins* = (5,4,1), MinNumCoins(70) is computed 6 times

How many times do you think MinNumCoins(30) is computed?

Exercise

Provide a memoization-based and a tabulation-based implementation for MinNumCoins

MinNumCoins(*money*) =

1. If *money* = 0, return 0
2. For each $coin_j \leq money$,
   set n_j = MinNumCoins($money - coin_j$)
3. Return 1 + min { n_j }

Exercise

State the assumption that underpins the correctness of MinNumCoins

MinNumCoins(*money*) =

1. If *money* = 0, return 0
2. For each $coin_j \leq money$,
   set n_j = MinNumCoins($money - coin_j$)
3. Return 1 + min { n_j }

Hint: Try it with coin denominations (2, 3, 5)

Exercise

Modify MinNumCoins''(*money*) to return also the coins to be given as change

MinNumCoins''(*money*) =

Set M(0) = 0

For m = 1 .. *money*

    For each $coin_j \leq$ m,

        Set n_j = M(m – $coin_j$)

    Set M(m) = 1 + min { n_j }

Return M(money)

Fibonacci numbers

Dynamic programming with less space

Fibonacci numbers

F(0) = 0
F(1) = 1
F(n) = F(n – 1) + F(n – 2)

Memoized version:
C(0) = 0
C(1) = 1
F'(n) = If C(n) is undefined
  Then C(n) = F'(n – 1) + F'(n – 2)
  Return C(n)

Tabulation version:
C(0) = 0
C(1) = 1
F''(n) = For j = 2 .. n
    C(j) = C(j – 1) + C(j – 2)
  Return C(n)

What space complexity do these implementations have?

O(n) space is needed in both implementations

Fibonacci numbers

F(0) = 0
F(1) = 1
F(n) = F(n – 1) + F(n – 2)

Memoized version:
C(0) = 0
C(1) = 1
F'(n) = If C(n) is undefined
    Then C(n) = F'(n – 1) + F'(n – 2)
    Return C(n)

Tabulation version:
C(0) = 0
C(1) = 1
F''(n) = For j = 2 .. n
        C(j) = C(j – 1) + C(j – 2)
    Return C(n)

What space complexity do these implementations have?

Can we do better?

When computing C(j)
C(j – 1) and C(j – 2) are needed
C(k) where k < j – 2 are not needed

Exploiting this gives:

```
Prev = 0; Curr = 1
F'''(n) = For j = 2 .. n
            Prev, Curr = Curr, Curr + Prev
          Return Curr
```

Space complexity is now O(1)

Exercise

Provide a space-efficient dynamic programming-based implementation for the “giving change” problem

MinNumCoins(*money*) =

1. If *money* = 0, return 0
2. For each $coin_j \leq money$,
   set n_j = MinNumCoins($money - coin_j$)
3. Return 1 + min { n_j }

Acknowledgement

The slides on the "giving change" problem are adapted from the slides of A/P Somayyeh Koohi