CSCI-UA 480.4: APS
Algorithmic Problem Solving

Dynamic Programming

Instructor: Joanna Klukowska

created based on materials for this class by
Bowen Yu and materials shared by the authors of
the textbook Steven and Felix Halim

Dynamic Programming

Dynamic programming is a technique of solving problems by means of breaking a larger problem into smaller/simpler sub-problems.

Dynamic programming solutions gain their speed over the complete-search types of problem solving techniques by making sure that each sub-problem is solved only once and the result is stored for reuse later one.

Making Change

Making change - revisited

Task: Given a set of coin denominations constructe a given value using as few coins as possible.

- `C = {c1, c2, c3, ..., cK}` set of coin denomination (assume we have unlimited number of each)
- `N` amount of money that we need to come up with

Making change - revisited

Task: Given a set of coin denominations constructe a given value using as few coins as possible.

- `C = {c1, c2, c3, ..., cK}` set of coin denomination (assume we have unlimited number of each)
- `N` amount of money that we need to come up with

Solution

- Recall that greedy strategy does not always work.
- We need to perform a complete search (with backtracking) to find the solution to this problem.
    - consider a function `coins` that given the amount of money and a set of denominations returns the mininum number of coins that one could use to make the change
    - calling `coins(N)` will solve our problem

```
coins( n )
    if  n < 0   return INF   // solution is not possible
    if  n == 0  return 0     // all money used, we found a solution

    best = INF
    for all c in C
        best = min ( best ,  coins(n-c) + 1)

    return best
```

    - this function may make a lot of repeated calls with the same parameter

Making change with Dynamic Programming

- in dynamic programming we will save the results of each of the recursive calls in a table to avoid making repeated computation

```
answer = a 1D array of (N+1) elements intialized to -1

coins( n )
    if  n < 0   return INF   // solution is not possible
    if  n == 0              // all money used, we found a solution
        answer[0] = 0
        return 0

    if answer[n] >= 0  return answer[n]

    //otherwise, compute it ...
    best = INF
    for all c in C
        best = min ( best ,  coins(n-c) + 1)

    //... and store it in the answer table
    answer[n] = best

    return best
```

- this guarantees that the function is called recursively at most N times (each call fills in one of the values in the `answer` array)

Making Change Iteratively

- the recursive DP solution that we have so far is a top-down approach

```
answer = a 1D array of (N+1) elements intialized to -1

coins( n )
    if  n < 0   return INF   // solution is not possible
    if  n == 0              // all money used, we found a solution
        answer[0] = 0
        return 0
    if answer[n] >= 0  return answer[n]

    best = INF
    for all c in C
        best = min ( best ,  coins(n-c) + 1)

    answer[n] = best

    return best
```

- we can compute the same using an iterative approach that constructs the solution from the bottom (bottom-up approach)

```
answer = a 1D array of (N+1) elements
answer[0] = 0

for n in 1 .. N
    answer[n] = INF
    for all c in C
        if n-c >= 0
            answer[n] = min ( answer[n] ,  answer[n-c] + 1)
```

Making Change: which coins to use

- in the previous solutions we only determined how many coins we can use
- what if we need to know which ones to use as well?
    - this means that we need to keep track of the information about the denominations of coins that went into the optimal solution

Making Change: which coins to use

- in the previous solutions we only determined how many coins we can use
- what if we need to know which ones to use as well?
  - this means that we need to keep track of the information about the denominations of coins that went into the optimal solution

```
answer = a 1D array of (N+1) elements
answer[0] = 0

first_coin = a 1D array of (N+1) elements, it indicates for each
            element the first coin used in the solution for that
            amount of money
first_coint[0] = -1

for n in 1 .. N
    answer[n] = INF
    for all c in C
        if n-c >= 0  AND  answer[n-c]+1 < answer[n]
            answer[n] = answer[n-c] + 1
            first_coin[n] = c
```

  - and then based on the `first_coin` array we can calculate an optimal solution

```
while n > 0
    print first_coin[n]
    n = n - first_coin[n]   //decrement the amount of money by the used coin
```

Making change: in how many ways

- another version of this problem is not to minimize the number of coins, but rather calculate the total number of possible ways in which we can make the requested amount
- recall that the original solution picked the smallest value returned by the recursive calls
    - in this new case, we actually want to calculate the sum of all the possibilities

```
answer = a 1D array of (N+1) elements
answer[0] = 0

for n in 1 .. N
    answer[n] = 0
    for all c in C
        if x-c >= 0
            answer[n] = answer[n] +  answer[n-c] + 1
```

Wedding Shopping

Challenge: Wedding Shopping

- Wdding Shopping

Given different options for each garment (e.g. 3 shirt models, 2 belt models, 4 shoe models, . . .) and a certain limited budget, our task is to buy one model of each garment.

We cannot spend more money than the given budget, but we want to spend the maximum possible amount.

The **input** consists of two integers $1 \leq M \leq 200$ and $1 \leq C \leq 20$, where M is the budget and C is the number of garments that you have to buy, followed by some information about the C garments. For the garment $g \in [0..C-1]$, we will receive an integer $1 \leq K \leq 20$ which indicates the number of different models there are for that garment g, followed by K integers indicating the price of each model $\in [1..K]$ of that garment g.

The **output** is one integer that indicates the maximum amount of money we can spend purchasing one of each garment without exceeding the budget. If there is no solution due to the small budget given to us, then simply print “no solution”.

Challenge: Wedding Shopping

Example 1

M = 20, C = 3:

```
3:   6 4 8
2:   5 10
4:   1 5 3 5
```

Solution: 19

3: 6 4 **8**
2: 5 **10**
4: **1** 5 3 5

3: **6** 4 8
2: 5 **10**
4: 1 5 **3** 5

Challenge: Wedding Shopping
Greedy Attempt Fails

Idea: Go through each garment and pick the most expensive item within the remaining budget.

Problem: The each decision influences the possible future decisions, making some of them impossible. In many cases, this approach will lead to wrong answers (sub-optimal solution or no solution).

Example:

M = 12, C = 3:

```
3:   6 4 8
2:   5 10
4:   1 5 3 5
```

Greedy approach picks 8 for the first garment, and returns no solution since the remaining budget of 4 does not allow us to pick anything else.

Optimal solution picks 6, 5, 1 and uses the entire budget.

BUT greedy approach works for the original example when the budget was 20.

Challenge: Wedding Shopping
Divide and Conquer

Subproblems (selecting of each type of the garment) are not independent - the solution depends on the choices made for other subproblems.

The divide and conquer approach is not suitable here.

Challenge: Wedding Shopping
Complete Search / Recursive Backtracking

Algorithm:

```
- start with money M and garment category i (assume indexing starting at zero),
call this function shop( M, i )

- if i == C (number of categories)
    record the total price if it is larger than the largest price calculated so far

- for each garment g in this category
    if price of g is <= M
        use this item and make a rerusive call to select a garment from catergory
        i+1 and the budget of (M - price of g), shop(M-p[g], i+1)
    else
        selecting g exceeds the budget, so do not continue on this path
```

Challenge: Wedding Shopping
Complete Search / Recursive Backtracking

Algorithm:

```
- start with money M and garment category i (assume indexing starting at zero),
call this function shop( M, i )

- if i == C (number of categories)
    record the total price if it is larger than the largest price calculated so far

- for each garment g in this category
    if price of g is <= M
        use this item and make a rerusive call to select a garment from catergory
        i+1 and the budget of (M - price of g), shop(M-p[g], i+1)
    else
        selecting g exceeds the budget, so do not continue on this path
```

Produces correct answer, but slow.

- in the largest case, each garment category has 20 different items and there are 20 garment categories
- this gives recursive calls to solve the sub-problems in the worst case - too many

Challenge: Wedding Shopping

But, the good news is that some of those subproblems are overlapping and we really have many fewer unique subproblems to solve.

Challenge: Wedding Shopping

But, the good news is that some of those subproblems are overlapping and we really have many fewer unique subproblems to solve.

- if any garment category has more than one item with the same price, then the two subproblems (selecting one or the other of the garments) are the same, example:

  C = 3:

  ```
  3:   6 1 8
  3:   5 10 5
  4:   1 5 3 5
  ```

  selecting either of the 5's in the second row, will lead to the same solutions

Challenge: Wedding Shopping

But, the good news is that some of those subproblems are overlapping and we really have many fewer unique subproblems to solve.

- if any garment category has more than one item with the same price, then the two subproblems (selecting one or the other of the garments) are the same, example:

  C = 3:

  ```
  3:   6 1 8
  3:   5 10 5
  4:   1 5 3 5
  ```

  selecting either of the 5's in the second row, will lead to the same solutions

- at the time that we look at garment cagory i, we might be starting with the same amount of money even though we made different choices in the previous categories, example:

  C = 3:

  ```
  3:   6 1 8
  3:   5 10 5
  4:   1 5 3 5
  ```

  selecting either 6 and 5, or 1 and 10 will lead to the same *best* option for the last row (in both cases, we already spend 11 out of the budget)

Challenge: Wedding Shopping

consider the following functions and quantities

- `shop(m, i)`, `0 <= m <= M` and `0 <= i <= C`
  returns the amount of money we can spend using `m` dollars for the garments in categories `i, i+1, ..., C-1`
- `p[i][g]` is the price of garment g in garment category i
- count[i] is the number of garments in garment category i

Challenge: Wedding Shopping

consider the following functions and quantities

- `shop(m, i)`, `0 <= m <= M` and `0 <= i <= C`
  returns the amount of money we can spend using `m` dollars for the garments in categories `i, i+1, ..., C-1`
- `p[i][g]` is the price of garment g in garment category i
- count[i] is the number of garments in garment category i

then the following are true

- if `m < 0`, then `shop(m, i) = negative infinity`
- if `i = C` (an item in last garment category has been purchased), then `shop(m,i) = M-m`
- for all other cases,
  `shop(m, i) = max( shop(m - p[i][g]), i+1) )` for all values of `0 <= g <= count[i]`

Challenge: Wedding Shopping

consider the following functions and quantities

- `shop(m, i)`, `0 <= m <= M` and `0 <= i <= C`
  returns the amount of money we can spend using `m` dollars for the garments in categories `i, i+1, ..., C-1`

- `p[i][g]` is the price of garment g in garment category i

- count[i] is the number of garments in garment category i

then the following are true

- if `m < 0`, then `shop(m, i) = negative infinity`

- if `i = C` (an item in last garment category has been purchased), then `shop(m,i) = M-m`

- for all other cases,
  `shop(m, i) = max( shop(m - p[i][g]), i+1) )` for all values of `0 <= g <= count[i]`

In this problem, there are

- 201 options for M (since `0 <= M <= 200`)
- 20 options for item category (since `1 <= C <= 20`)

so there are 201*20 = 4020 possible subproblems (in the worst case).

Challenge: Wedding Shopping

DP - take 1 (top-down)

```
answer = an M+1 by C 2D array initialized to -1

shop( m, i )

    if m < 0  return -INF           //run out of money
    if i == C return (M - m)        //finished and spent M-m

    // !!! without this line below, we are simply doing backtracking
    if answer[m][i] != -1   return answer[m][i]

    best = -1
    for g in  0 .. count[i]
        best  =  max ( best,  shop(m - price[i][g], i+1) )

    answer[m][i] = best

    return best
```

Challenge: Wedding Shopping
DP - take 2 (bottom-up)

- for a bottom-up approach to solving this problem see the actual code in C++ and Java from the textbook

Best Path in a Grid

Challenge: Best Path in a Grid

Task

Given an NxN grid find the best path from the upper left corner to the lower right corner.

Best = the one whose values add up to the highest number

Restrictions: you can only move down or right

Challenge: Best Path in a Grid

Task

Given an NxN grid find the best path from the upper left corner to the lower right corner.

Best = the one whose values add up to the highest number

Restrictions: you can only move down or right

Example

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Challenge: Best Path in a Grid

Task

Given an NxN grid find the best path from the upper left corner to the lower right corner.

Best = the one whose values add up to the highest number

Restrictions: you can only move down or right

Example

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

3	7	9	2	7
9	**8**	3	5	5
1	**7**	**9**	**8**	**5**
3	8	6	4	**10**
6	3	9	7	**8**

Challenge: Best Path in a Grid

Solution

- assume that rows and columns are numbered using indexes `1..n`, so the value of the upper left corner is `value[1][1]` and the value of the lower right corner is `value[n][n]`
- we have the following relationship
- we store the `sum[y][x]` as a two dimensional matrix equal in size to the given grid