Dynamic Programming

Week 6 Objectives

- Subproblem Optimal structure
- Defining the dynamic recurrence
- Bottom up computation
- Tracing the solution

Subproblem Optimal Structure

- Divide and conquer - optimal subproblems
- divide PROBLEM into SUBPROBLEMS, solve SUBPROBLEMS
- combine results (conquer)
- critical/optimal structure: solution to the PROBLEM must include solutions to subproblems (or subproblem solutions must be combinable into the overall solution)
- PROBLEM = {DECISION/MERGING + SUBPROBLEMS}

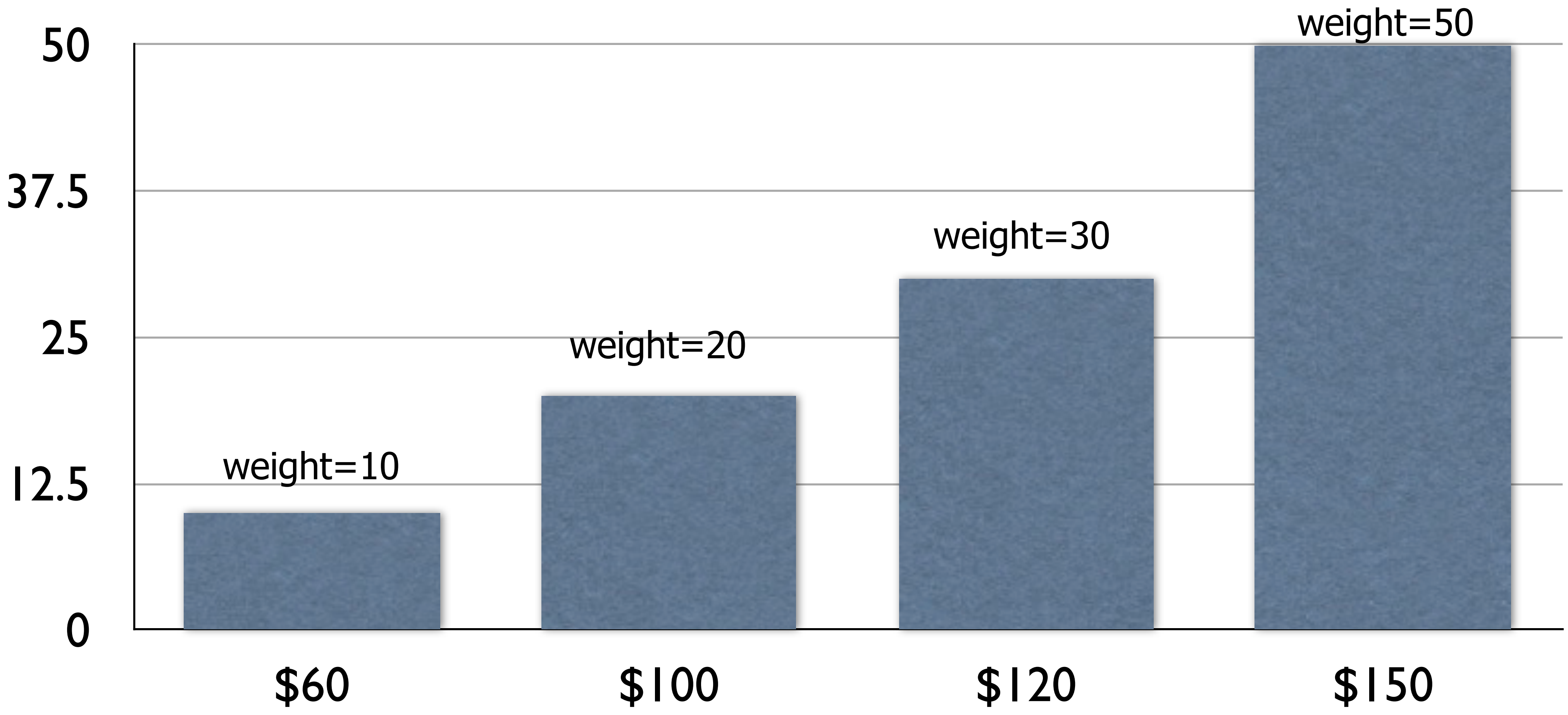
Optimal Structure - NON GREEDY

- Cannot make a choice decision/CHOICE without solving subproblems first
- Might have to solve many subproblems before deciding which results to merge.

Ex: Discrete 0/1 Knapsack

- objects (paintings) sold by item
- weights w1,w2,w3,w4...
- values v1,v2,v3,v4...
- knapsack capacity (weight) = W
- task : fill the knapsack to maximize value

Ex: Discrete Knapsack



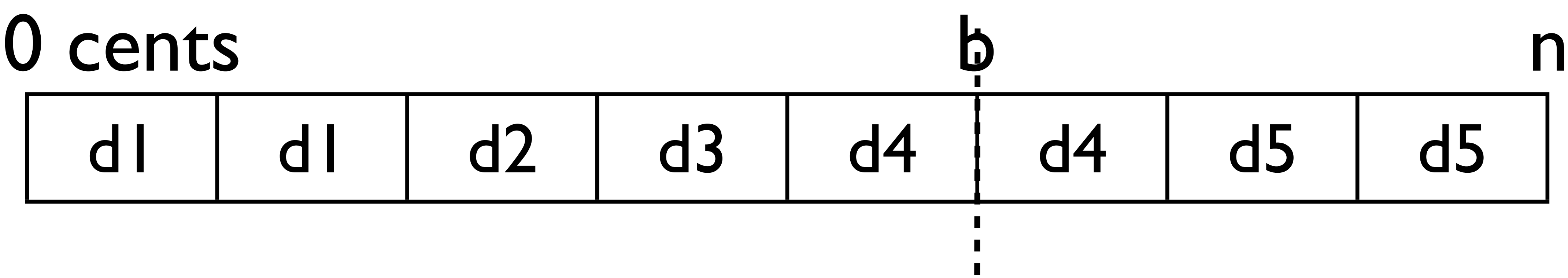
- naive approaches may lead to a bad solution
  - choose by biggest value - tea first
  - choose by smallest quantity - flour first
  - correct:

Dynamic Programming

- Characterize the structure of the optimal solution
- Define the dynamic recurrence
- Compute value bottom up (fill table)
  - or top-down using Memoization
  - will focus for now on "bottom-up", then discuss top-down Memoization as an alternative
- Trace the solution

Coin Change

- coin denominations $d_1, d_2, \ldots d_k$
- task: give change of n cents using as few as possible coins
  - denominations can be used multiple times
- 1) characterize optimal solution structure



- if above solution optimal, then
  - {d1,d1,d2,d3,d4} optimal solution for b cents
  - {d4,d5,d5} optimal solution for n-b cents

Coin change

- 2) value and dynamic recursion
- define $C[n]$ = minimum number of coins to make change of n cents (thus optimal solution)
- consider subproblems
  - if d1 is used to make change for n cents optimally (one of $C[n]$ coins) then $C[n]=1+C[n-d_1]$ ($C[n-d_1]$ is optimal solution for the rest of of the problem $n-d_1$)
  - if d_2 is used then $C[n]=1+C[n-d_2]$ etc
  - $C[n]$ is minimum, so $C[n] = \min_i \{1+ C[n-d_i]\}$. This requires that we have already computed values $C[n-d_i]$ for all i
- formally $C[n]$ =
  - 0, if $n=0$
  - 1, if $n=d_i$
  - $\min_{[i:d_i \leq n]} \{1+C[n-d_i]\}$, otherwise

Coin change

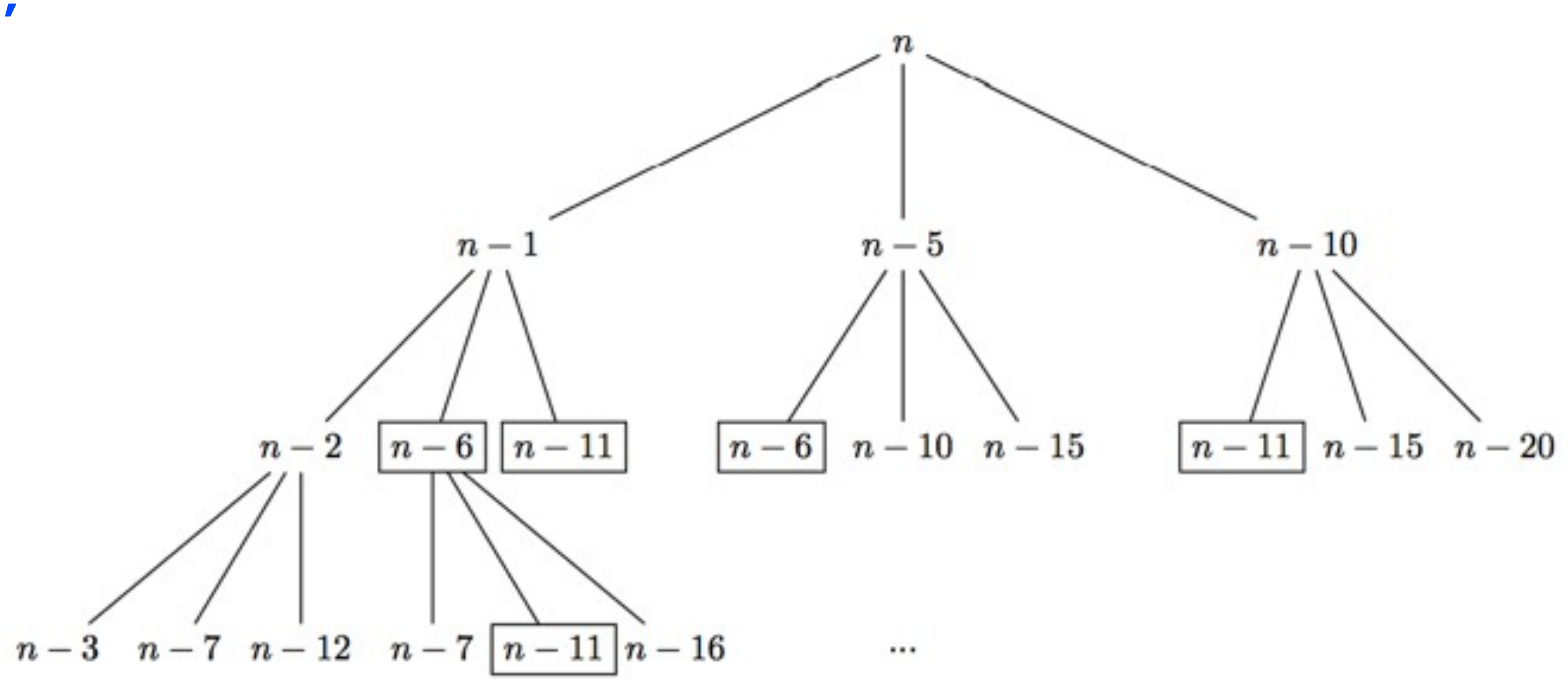
- 3) compute bottom-up the values C[]; also remember at each step the coin used to obtain the solution

```
C[0]=0;
for p=1:n
    min=∞
    for i=1:k
        if (p≥di && C[p-di]+1 < min) then
            min = C[p-di]+1
            coin=i
    C[p]=min
    S[p]=coin
return C[] and S[]
```

Coin Change

- naive way to solve the recursion top-down
  - exponential running time
  - same argument as with Fibonacci numbers top-down recursion

```
change(n, denominations d1=1,d2=5,d3=10)
  if(n==0) return 0;
  else
    val = 1+ min{change(p-10), change(p-5), change(p-1);
    return val;
```



Coin Change

- 4) Trace the solution
- at problem size=n the coin used was S[n]
  - we have used coin S[n], and then solved the problem $n-d_{S[n]}$
  - thus the next coin will be $S[n-d_{S[n]}]$, etc

```
Trace Solution (S[],d,n)
  while(n>0)
    print "coin S[n]"
    n= n-d_S[n]
```

Coin Change

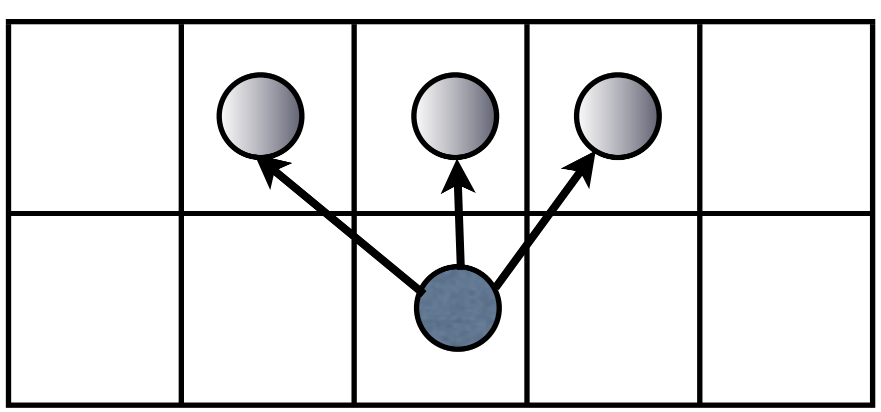
- Running time bottom up: for each step p=1:n
  - k comparisons
  - $\Theta(nk)$ total
- Tracing Solution : $O(n)$ steps
- Total $\Theta(nk)$

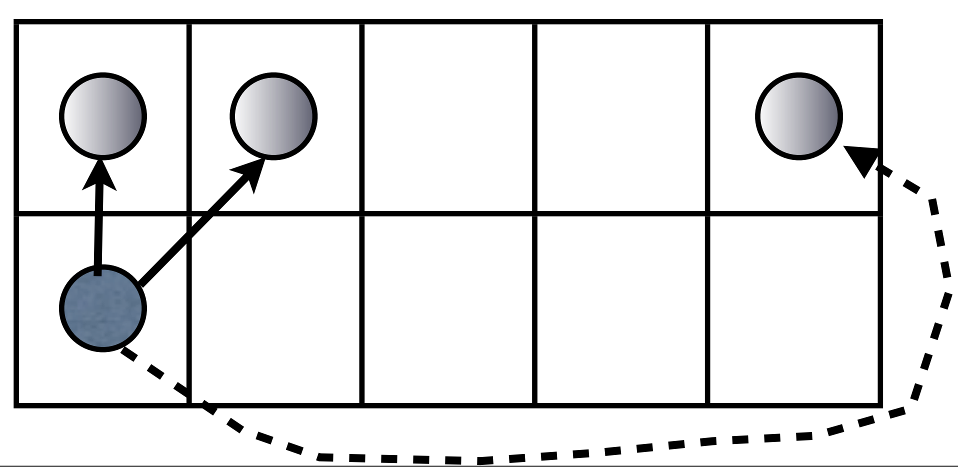
Check Board Pb

- Table of penalties given as a matrix P_{ij}; i=1:m; j=1:n
- Task: find the minimum path from anywhere-first-row to anywhere-last-row
  - always advance one row; can move straight, left, right
  - columns form a cylinder (left move from the left column ends up on the right column, and viceversa). Say column 0 is actually column n; column n+1 is column 1

illustrated path penalty= 1+3+1+1+2=8

m	7	1	2	6	6	1	0
	5	5	0	1	7	2	4
	3	2	9	1	1	3	7
2	0	1	5	3	8	6	2
1	1	3	6	3	1	7	6
	1	2	3				n





Check Board Pb

- 1)optimal solution structure
- if path $P=(i_1,j_1)(i_2,j_2)....(i_k,j_k)...(i_m,j_m)$ optimal overall, then
  - path $P'= (i_1,j_1)(i_2,j_2)....(i_k,j_k)$ is optimal to get from first row to cell (i_k,j_k)
  - path $P''= (i_k,j_k)..(i_m,j_m)$ is optimal to get from cell (i_k,j_k) to the last row
  - explain why (exchange argument)

Text

m	7	1	2	6	6	1	0
	5	5	0	1	7	2	4
	3	2	9	1	1	3	7
2	0	1	5	3	8	6	2
1	1	3	6	3	1	7	6
	1	2	3				n

(i_k,j_k)

CheckBoard

- 2)dynamic recurrence
- C[i,j]= minimum cost (penalty) from row 1 to cell [i,j]
- C[i,j] = Pij if i=1 (first row)
- Pij (that cell) + minimum of the path up to that cell
  - can come on cell [i,j] from any of the three cells below
  - Pij + min (C[i-1,j-1], C[i-1,j], C[i-1,j+1])

CheckBoard

- 3) Bottom up computation (fill array C)
  - `c[1,j]=P`$_{1j}$ `for all j`
  - `for i=2:m`
    - `for j=1:n`
      - `c[i,j]= P`$_{ij}$ `+ min (C[i-1,j-1], C[i-1,j], C[i-1,j+1])`
  - `return array C[]`

CheckBoard

- 4)Trace the solution
  - array C computed
    - `find the minimum column j = argmin C[m,:] on the last row; output cell (m,j)`
    - `i=m; while i>1`
      - `j_below = argminj (C[i-1,j-1], C[i-1,j], C[i-1,j+1]); output cell (i-1,j_below)`
      - `i=i-1; j=j_below`

CheckBoard – Running Time

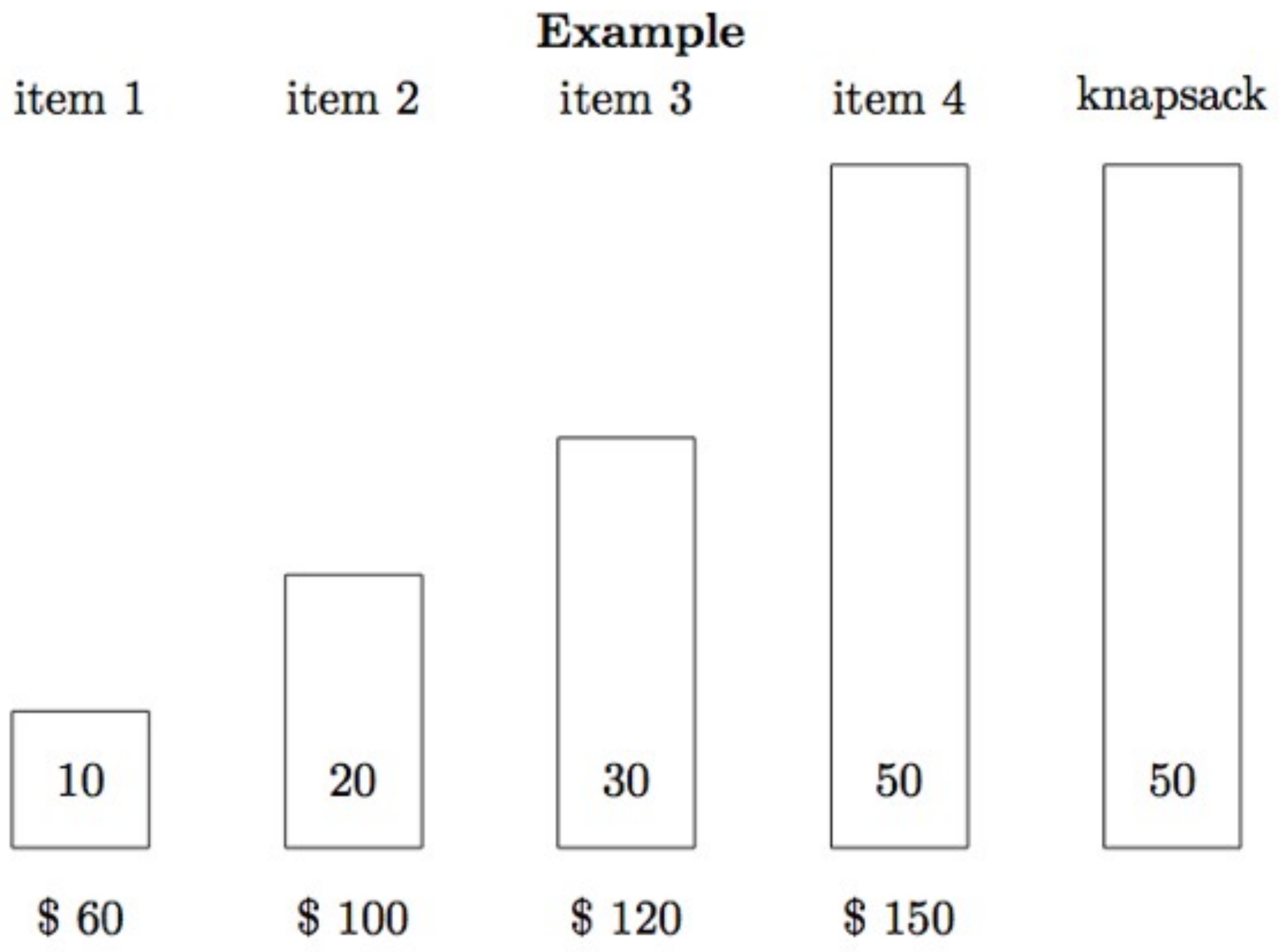
- Outer loop – n iterations
- inner loop – m iteration
- \- constant time (3 comparisons)
- Total Θ(mn)

Discrete Knapsack

- given a knapsack of max-weight W
- and a set of items
  - item weights w_1, w_2, ..., w_n
  - item values v_1,v_2, ...,v_n
- select the items that fit in the knapsack and maximize the total value.
  - difference to discrete knapsack: an item can be selected or not, no fractions allowed

Discrete Knapsack

- Greedy ideas dont work - lead to not-optimal selection of items:
  - select maximum value
  - select minimum weight



Discrete Knapsack - trick

- Before we proceed to steps 1-4, solution need to fix an order of the items.
- any order works, but it fas to be fixed
- will use the order given by the input : items 1, 2, 3, ...,n

Discrete Knapsack

- 1) characterize the optimal solution structure
- say i is the highest number item (by our fixed order) included in the optimal solution SOL
  - SOL contains some items in the set {1,2,...i}
  - so item i+1, i+2, ... , n not used
- then SOL\{i} is the optimal solution for the Knapsack problem (knapsack = $W-w_i$, items {1,2,3..,i-1})
  - why ? use an exchange argument

Discrete Knapsack

- 2) dynamic recursion
- C[i,W] = maximum value to the Knapsack problem (knapsack=W, items ={1,2,3...i})
- does C[i,W] includes the item i?
  - not if wi>W
  - if no, C[i,W] = C[i-1,W]
  - if yes, C[i,W] = C[i-1,W-w_i] + vi
  - we dont know yes or no above, so we solve both subprobelms, choose max

$$c[i,w] = \begin{cases} 0 & \textit{if } i = 0 \textit{ or } w = 0, \\ c[i-1,w] & \textit{if } w_i > w, \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \textit{if } i > 0 \textit{ and } w \geq w_i. \end{cases}$$

Discrete Knapsack

- 3) bottom up computation of C[]
  - `for w=0:W {C[0,w]=0}`
  - `for i=1:n`
    - `C[i,0]=0`
    - `for w=1:W`
      - `if wi>w C[i,w]=C[i-1,w]`
      - `else C[i,w] = max(vi+C[i-1,w-w`$_i$`], C [i-1,w])`

Discrete Knapsack

- 4) Trace the solution
- computed C[], weights w[], number of items n , knapsack capacity W

```
Items(C[],w[],n,W)
while (n>0 and W>0)
    if(C[n,W]>C[n-1,W])
        output n
        W=W-wn
    n=n-1
```

Discrete Knapsack – running time

- Outer for loop – n iterations
- Inner for loop – W iterations
- – inside step : constant time
- Overall $\Theta(nW)$