cs475
Knapsack Problem and Dynamic Programming

Wim Bohm, CS, CSU

sources: Cormen,Leiserson; Kleinberg, Tardos, Vipin Kumar et.al.

Dynamic Programming Applications

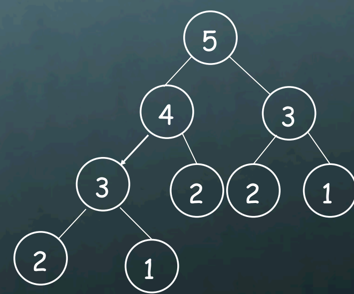
- Areas.
  - Search
  - Bioinformatics
  - Control theory
  - Operations research
- Some famous dynamic programming algorithms.
  - Unix diff for comparing two files.
  - Knapsack
  - Smith-Waterman for sequence alignment.

Fibonacci numbers

- F(1) = F(2) = 1, F(n) = F(n-1) + F(n-2) n>2

Simple recursive solution

```
def fib(n) :
  if n<=2: return 1
  else: return fib(n-1) + fib(n-2)
```



What is the size of the call tree?

exponential

Using a memo table

```
def fib(n, table) :
  # pre: n>0, table[i] either 0 or contains fib(i)
  if n <= 2 : return 1
  if table[n] > 0 : return table[n]
  result = fib(n-1, table) + fib(n-2, table)
  table[n] = result
  return result
```

We use a memo table, never computing the same value twice. How many calls now?

Look ma, no table

```
def fib(n) :
    if n<=2 : return 1
    else
      a = 1; b=1, c=0
      do n-2 times : c = a + b;  a = b; b = c
      return c
```

Compute the values "bottom up"
Only store needed values: the previous two
Same O(n) time complexity, constant space

5

Dynamic Programming

- Characterize the structure of the problem, ie show how a larger problem can be solved using solutions to sub-problems
- Recursively define the optimum
- Compute the optimum bottom up, storing values of sub solutions
- Construct the optimum from the stored data

Optimal substructure

- Dynamic programming works when a problem has optimal substructure: we can construct the optimum of a larger problem from the optima of a "small set" of smaller problems.
  - small: polynomial
- Not all problems have optimal substructure.
- Travelling Salesman Problem (TSP)?

Knapsack Problem

- Given n objects and a "knapsack" of capacity W
- Item i has a weight $w_i > 0$ and value $v_i > 0$.
- Goal: fill knapsack so as to maximize total value.
- Is there a Greedy solution?
  - What's Greedy again?
  - What would it do here?

repeatedly add item with maximum v_i / w_i ratio …

Does Greedy work?

Capacity M = 7, Number of objects n = 3
w = [5, 4, 3]
v = [10, 7, 5] (ordered by v_i / w_i ratio)

Recursion for Knapsack Problem

- Notation: OPT(i, W) = optimal value of max weight subset that uses items 1, …, i with weight limit W.
  - Case 1: item i is not included:
    - Take best of { 1, 2, …, i-1 } using weight limit W : OPT(i-1,W)
  - Case 2: item i with weigh w_i and value v_i is included:
    - only possible if W >= w_i
    - new weight limit = W – w_i
    - Take best of { 1, 2, …, i–1 } using weight limit W-w_i and add v_i:
      OPT(i-1,W-w_i)+v_i

$$OPT(i, W) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, W) & \text{if } w_i > W \\ \max\{OPT(i-1, W),\ \ v_i + OPT(i-1, W-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up Dynamic Programming

- Knapsack. Fill an n-by-W array.

```
Input: n, W, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if w_i > w :
         M[i, w] = M[i-1, w]
      else :
         M[i, w] = max (M[i-1, w], v_i + M[i-1, w-w_i ])

return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT: 40

How do we find the choice vector x, in other words the objects picked in the optimum solution?

Walk back through the table!!

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT: 40
n=5 Don't take object 5

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT: 40
n=5 Don't take object 5
n=4 Take object 4

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT: 40
n=5 Don't take object 5
n=4 Take object 4
n=3 Take object 3

and now we cannot take anymore,
so choice set is {3,4}

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

- Running time. Θ(nW).
  - Not polynomial in input size!
    - W can be exponential in n
  - "Pseudo-polynomial."
- Knapsack approximation: there exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum

DIY: another example

W = 5 4 6 1   P = 7 8 9 4   M = 10

Example

W = 5 4 6 1   P = 7 8 9 4   M = 10

obj \ cap	0	1	2	3	4	5	6	7	8	9	10
{}	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	7	7	7	7	7	7

Example

W = 5 4 6 1   P = 7 8 9 4   M = 10

obj \ cap	0	1	2	3	4	5	6	7	8	9	10
{}	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	7	7	7	7	7	7
1:2	0	0	0	0	8	8	8	8	8	15	15

Example

W = 5 4 6 1   P = 7 8 9 4   M = 10

cap

obj	0	1	2	3	4	5	6	7	8	9	10
{}	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	7	7	7	7	7	7
1:2	0	0	0	0	8	8	8	8	8	15	15
1:3	0	0	0	0	8	8	9	9	9	15	17

Example

W = 5 4 6 1   P = 7 8 9 4   M = 10

cap

obj	0	1	2	3	4	5	6	7	8	9	10		
{}	0	0	0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	7	7	7	7	7	7		
1:2	0	0	0	0	8	8	8	8	8	15	15		
1:3	0	0	0	0	8	8	9	9	9	15	17		
1:4	0	4	4	4	8	12	12	13	13	15	19	take 4	WHY?

Example

W = 5 4 6 1 P = 7 8 9 4 M = 10

cap

obj	0	1	2	3	4	5	6	7	8	9	10		
{}	0	0	0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	7	7	7	7	7	7		
1:2	0	0	0	0	8	8	8	8	8	15	15		
1:3	0	0	0	0	8	8	9	9	9	15	17	not 3	WHY?
1:4	0	4	4	4	8	12	12	13	13	15	19	take 4	WHY?

Example

W = 5 4 6 1 P = 7 8 9 4 M = 10

cap

obj	0	1	2	3	4	5	6	7	8	9	10		
{}	0	0	0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	7	7	7	7	7	7		
1:2	0	0	0	0	8	8	8	8	8	15	15	take 2	WHY?
1:3	0	0	0	0	8	8	9	9	9	15	17	not 3	WHY?
1:4	0	4	4	4	8	12	12	13	13	15	19	take 4	WHY?

Example

W = 5 4 6 1   P = 7 8 9 4   M = 10

cap

obj	0	1	2	3	4	5	6	7	8	9	10		
{}	0	0	0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	7	7	7	7	7	7	take 1	WHY?
1:2	0	0	0	0	8	8	8	8	8	15	15	take 2	WHY?
1:3	0	0	0	0	8	8	9	9	9	15	17	not 3	WHY?
1:4	0	4	4	4	8	12	12	13	13	15	19	take 4	WHY?

Example

W = 5 4 6 1   P = 7 8 9 4   M = 10

cap

obj	0	1	2	3	4	5	6	7	8	9	10		
{}	0	0	0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	7	7	7	7	7	7	take 1	WHY?
1:2	0	0	0	0	8	8	8	8	8	15	15	take 2	WHY?
1:3	0	0	0	0	8	8	9	9	9	15	17	not 3	WHY?
1:4	0	4	4	4	8	12	12	13	13	15	19	take 4	WHY?

Solution vector: 1101, optimum: 19