Dynamic Programming

Cormen et. al. IV 15

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Operations research.

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Smith-Waterman for sequence alignment.
- Bellman-Ford for shortest path routing in networks.

Fibonacci numbers

$F(1) = F(2) = 1$

$F(n) = F(n-1) + F(n-2) \quad n>2$

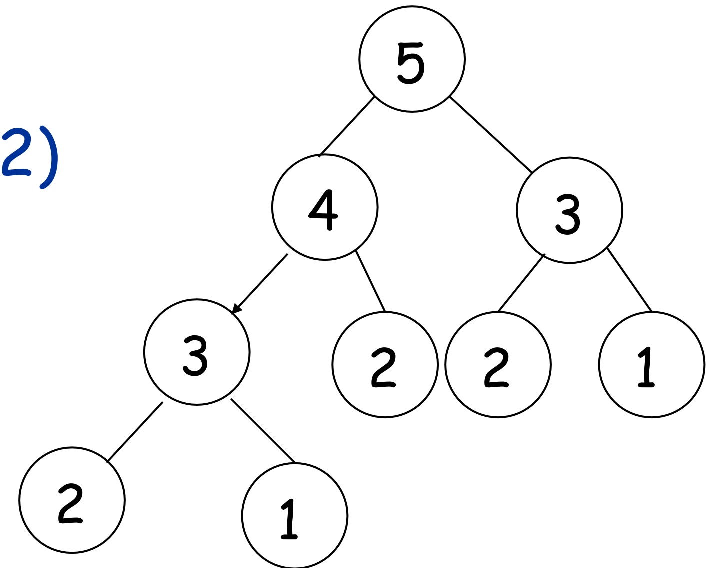
Fibonacci numbers

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?



Fibonacci numbers

$F(1) = F(2) = 1$
$F(n) = F(n-1) + F(n-2) \quad n>2$

Simple recursive solution:

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

Problem: exponential call tree

Can we avoid it?

Efficient computation 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 used a memo table, never computing the same value twice. How many calls now? Can we do better?

Look ma, no table

```
def fib(n) :
   if n<=2 : return 1
   a,b = 1
   c = 0
   for i in range(3, n+1) :
        c = a + b
        a = b
        b = c
    return c
```

Compute the values "bottom up"
Avoid the table, only store the previous two
same O(n) time complexity, constant space

Optimization Problems

In optimization problems a set of **choices** are to be made to arrive at an optimum, and sub problems are encountered.

This often leads to a **recursive** definition of a solution. However, the recursive algorithm is often **inefficient** in that it solves the **same sub problem many times.**

Dynamic programming avoids this repetition by solving the problem **bottom up** and **storing** sub solutions.

Dynamic vs Greedy, Dynamic vs Div&Co

Compared to Greedy, there is **no predetermined local best choice** of a sub solution, but a best solution is chosen by computing a set of alternatives and **picking the best**.

Dynamic Programming **builds on** the recursive definition of a divide and conquer solution, but **avoids re-computation** by storing earlier computed values, thereby often saving orders of magnitude of time.

Fibonacci: from exponential to linear

Dynamic Programming

Dynamic Programming has the following steps

- Characterize the **structure** of the problem, i.e., 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)?

Weighted Interval Scheduling

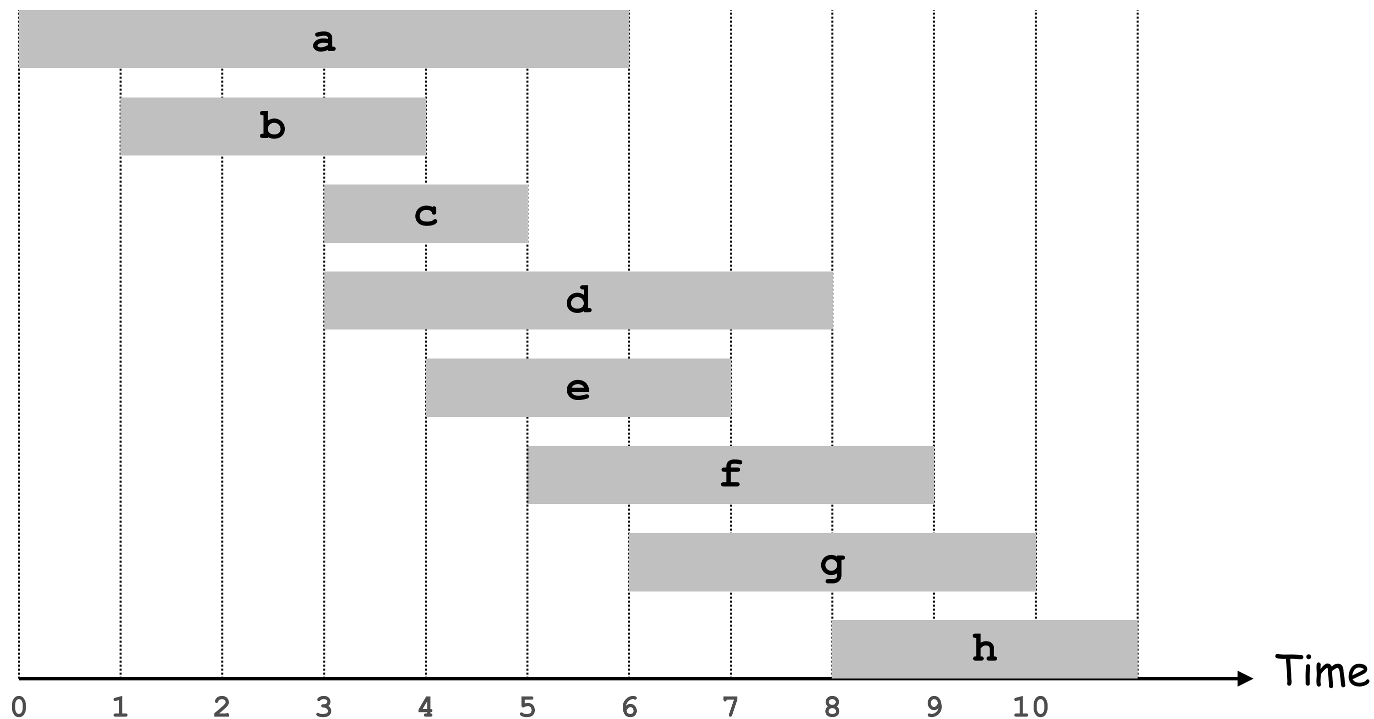
We studied a greedy solution for the interval scheduling problem, where we searched for the maximum number of compatible intervals.

If each interval has a weight and we search for the set of compatible intervals with the maximum sum of weights, no greedy solution is known.

Weighted Interval Scheduling

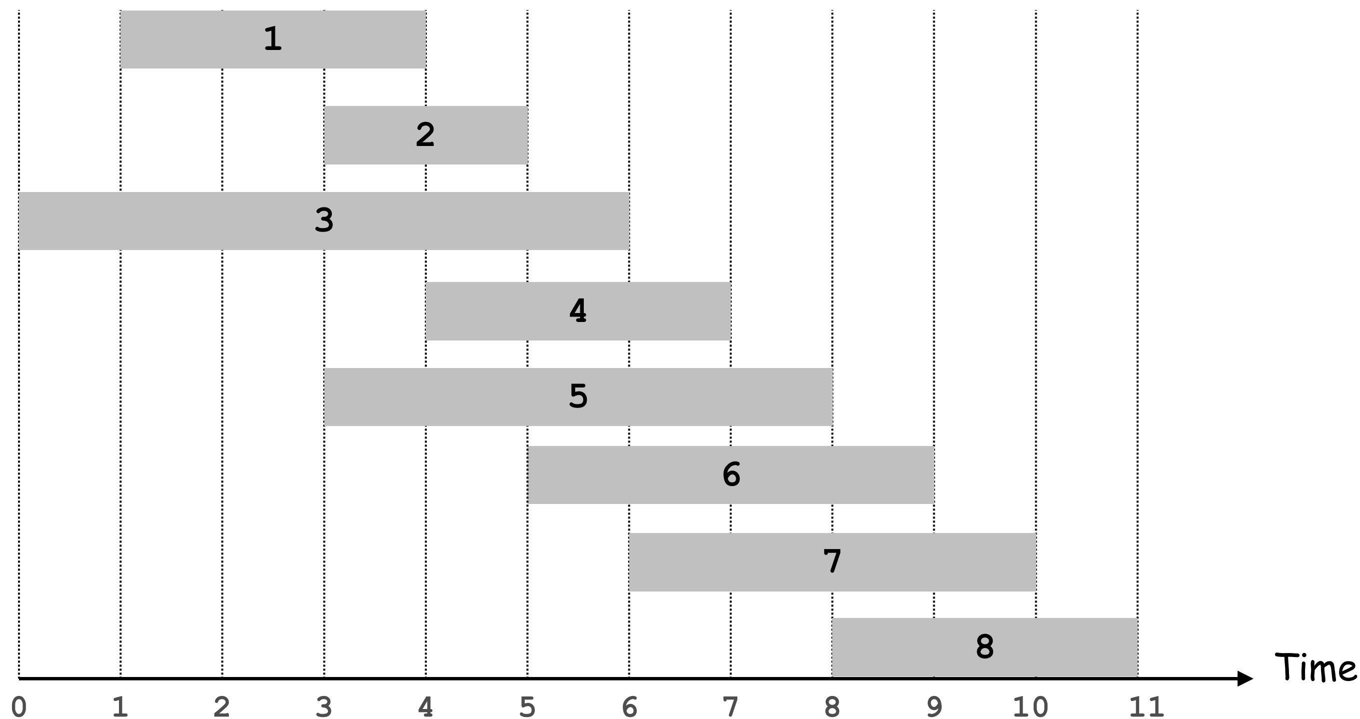
Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum value subset of compatible jobs.



Weighted Interval Scheduling

Assume jobs sorted by finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
p(j) = largest index i < j such that job i is compatible with j,
in other words: p(j) is j-s latest **predecessor**; p(j) = 0 if it has no predecessors. Example: p(8) = 5, p(7) = 3, p(2) = 0.
Using p(j) can you think of a recursive solution?



Dynamic Programming: Recursive Solution

Notation. OPT(j): optimal value to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT(j) includes job j.
  - add v_j to total value
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)
- Case 2: OPT(j) does not include job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Recursive Solution

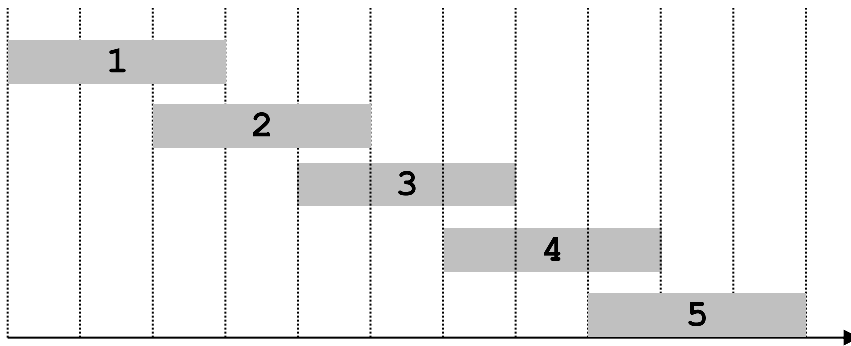
```
input: s1,…,sn , f1,…,fn , v1,…,vn

sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j == 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)),Compute-Opt(j-1))
}
```

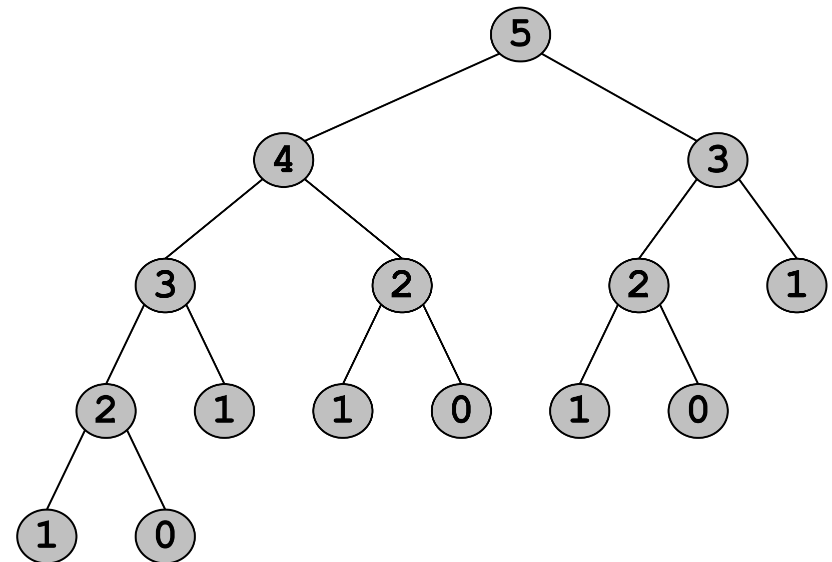
What is the size of the call tree here?
How can you make it as big as possible?

Analysis of the recursive solution

Observation. Recursive algorithm considers exponential number of (redundant) sub-problems.

Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.





p(1) = 0, p(j) = j-2
Code on previous slide becomes
Fibonacci: opt(j) calls
opt(j-1) and opt(j-2)

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
input: n, s1,…,sn , f1,…,fn , v1,…,vn

sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)),
                      M-Compute-Opt(j-1))
   return M[j]
}
```

Global array (annotation pointing to `M[j] = empty`)

Weighted Interval Scheduling: Running Time

Claim. Memoized version of `M-Compute-Opt(n)` takes O(n log n) time.

- `M-Compute-Opt(n)` fills in all entries of M ONCE in constant time
- Since M has n+1 entries, this takes O(n)

- But we have sorted the jobs

- So Overall running time is O(n log n).

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithm computes optimal value. What if we want the solution itself?
A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
input: n, s1,…,sn , f1,…,fn , v1,…,vn

sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

By going in bottom up order M[p(j)] and M[j-1] are present when M[j] is computed. This again takes O(nlogn) for sorting and O(n) for Compute, so O(nlogn)

Algorithmic Paradigms

Greedy. Build up a solution incrementally, optimizing some local criterion.

Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Memoization: follow divCo, but to avoid re-computing values, store computed values in memo table and check first if value was already computed.

Dynamic programming. Break up a problem into a series of sub-problems, and build up solutions to larger and larger sub-problems, store and re-use results.

Memoization

Remember Fibonacci: F(1) = F(2) = 1; F(n) = F(n-1) + F(n-2) n>2

Recursive solution, exponential call tree:

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

Memo table, follows the recursive solution but stores results to avoid recomputation, as in naïve recursive solution

```
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
```

Using a memo table, we never compute the same value twice. How many calls now?

At most n. So O(n) time and space bound

Dynamic Programming

- Characterize the structure of the problem: show how a problem can be solved using solutions to sub-problems

- Recursively define the optimum

- Compute the optimum bottom up, storing solutions of sub problems, optimizing the size of the stored data structure

- Construct the optimum from the stored data

Discrete Optimization Problems

Discrete Optimization Problem (S,f)

- S: space of feasible solutions (satisfying some constraints)
- $f : S \rightarrow R$
  - Cost function associated with feasible solutions
- Objective: find an optimal solution x_{opt} such that

  $f(x_{opt}) \leq f(x)$ for all x in S (minimization)

  or $f(x_{opt}) \geq f(x)$ for all x in S (maximization)

- Ubiquitous in many application domains
  - planning and scheduling
  - VLSI layout
  - pattern recognition

Example: Subset Sums

Given a set of n objects with weights w_i and a capacity W, find a subset S with the **largest sum of weights** such that total weight is less equal W

Does greedy work? How would you do it?

Largest first: No {3 2 2} W=4

Smallest first: No {1 2 2} W=4

Recursive Approach

Two options for object i:

- Either take object i or don't

Assume the current available capacity is W

- If we take object i, leftover capacity is $W - w_i$
  this gives rise to one recursive call
- If we don't, leftover capacity is W
- this gives rise to another recursive call

Then, after coming out of the two recursive calls, take the best of the two options

Recursive solution for Subset-Sum Problem

Notation: OPT(i, w) = weight of max weight subset that uses items 1, …, i with weight limit w.

- Case 1: item i is not included:
  - OPT includes best of { 1, 2, …, i-1 } using weight limit w
- Case 2: item i is included:
  - new weight limit = $w - w_i$
  - OPT includes best of { 1, 2, …, i-1 } using remaining capacity $w-w_i$

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

Subset Sum Problem: Bottom-Up Dynamic Programming

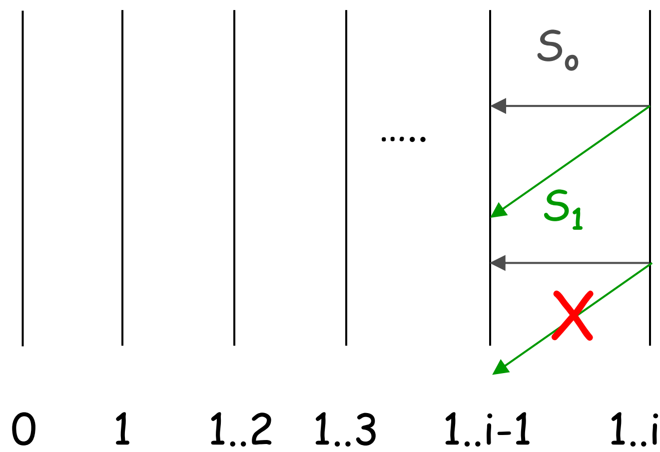
Approach: Fill an n-by-W array.

```
Input: n, W, w1,…,wN

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

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}

return M[n, W]
```

Subset Sum: Dynamic Programming

Go through the state space bottom-up

- select 1 object, then 2 objects, ..., all objects
  - What does M[0,*] look like? And M[1,*]?
- use solutions of smaller sub-problem to efficiently compute solutions of larger one



s0 = M[i-1,c]
if c>=W[i]:
  s1 = M[i-1,c-W[i]] + W[i]
else: s1=0
M[i,c] = max(s0,s1)
$0 \leq c \leq M$

Example

Capacity: 11
Object: 1 2 3 4
Weight: 2 6 4 3

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	0	1	2	3	4	5	6	7	8	9	10	11
{}:	0	0	0	0	0	0	0	0	0	0	0	0
{1}:												
{1,2}:												
{1:3}:												
{1:4}:												

Example

Capacity: 11

Object:	**1**	**2**	**3**	**4**
Weight:	**2**	**6**	**4**	**3**

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
{}:	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
{1}:	**0**	**0**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
{1,2}:												
{1:3}:												
{1:4}:												

Example

Capacity: 11

Object:	1	2	3	4
Weight:	2	6	4	3

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	0	1	2	3	4	5	6	7	8	9	10	11
{}:	0	0	0	0	0	0	0	0	0	0	0	0
{1}:	0	0	2	2	2	2	2	2	2	2	2	2
{1,2}:	0	0	2	2	2	2	6	6	8	8	8	8
{1:3}:												
{1:4}:												

Example

Capacity: 11

Object:	**1**	**2**	**3**	**4**
Weight:	**2**	**6**	**4**	**3**

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
{}:	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
{1}:	**0**	**0**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
{1,2}:	**0**	**0**	**2**	**2**	**2**	**2**	**6**	**6**	**8**	**8**	**8**	**8**
{1:3}:	**0**	**0**	**2**	**2**	**4**	**4**	**6**	**6**	**8**	**8**	**10**	**10**
{1:4}:												

Example

Capacity: 11

Object:	1	2	3	4
Weight:	2	6	4	3

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	0	1	2	3	4	5	6	7	8	9	10	11
{}:	0	0	0	0	0	0	0	0	0	0	0	0
{1}:	0	0	2	2	2	2	2	2	2	2	2	2
{1,2}:	0	0	2	2	2	2	6	6	8	8	8	8
{1:3}:	0	0	2	2	4	4	6	6	8	8	10	10
{1:4}:	0	0	2	3	4	5	6	7	8	9	10	11

Which objects? Choice vector?

Example

Capacity: 11

Object:	1	2	3	4
Weight:	2	6	4	3

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	0	1	2	3	4	5	6	7	8	9	10	11	
{}:	0	0	0	0	0	0	0	0	0	0	0	0	
{1}:	0	0	2	2	2	2	2	2	2	2	2	2	
{1,2}:	0	0	2	2	2	2	6	6	8	8	8	8	
{1:3}:	0	0	2	2	4	4	6	6	8	8	10	10	
{1:4}:	0	0	2	3	4	5	6	7	8	9	10	11	pick 4

Which objects? Choice vector?

Example

Capacity: 11

Object:	**1**	**2**	**3**	**4**
Weight:	**2**	**6**	**4**	**3**

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	0	1	2	3	4	5	6	7	8	9	10	11	
{}:	0	0	0	0	0	0	0	0	0	0	0	0	
{1}:	0	0	2	2	2	2	2	2	2	2	2	2	
{1,2}:	0	0	2	2	2	2	6	6	8	8	8	8	
{1:3}:	0	0	2	2	4	4	6	6	8	8	10	10	not 3
{1:4}:	0	0	2	3	4	5	6	7	8	9	10	11	pick 4

Which objects? Choice vector?

Example

Capacity: 11

Object:	1	2	3	4
Weight:	2	6	4	3

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	0	1	2	3	4	5	6	7	8	9	10	11	
{}:	0	0	0	0	0	0	0	0	0	0	0	0	
{1}:	0	0	2	2	2	2	2	2	2	2	2	2	
{1,2}:	0	0	2	2	2	2	6	6	8	8	8	8	pick 2
{1:3}:	0	0	2	2	4	4	6	6	8	8	10	10	not 3
{1:4}:	0	0	2	3	4	5	6	7	8	9	10	11	pick 4

Which objects? Choice vector?

Example

Capacity: 11

Object:	**1**	**2**	**3**	**4**
Weight:	**2**	**6**	**4**	**3**

Table: rows are sets of objects (as opposed to columns on previous slide)

cap:	0	1	2	3	4	5	6	7	8	9	10	11	
{}:	0	0	0	0	0	0	0	0	0	0	0	0	
{1}:	0	0	2	2	2	2	2	2	2	2	2	2	pick 1
{1,2}:	0	0	2	2	2	2	6	6	8	8	8	8	pick 2
{1:3}:	0	0	2	2	4	4	6	6	8	8	10	10	not 3
{1:4}:	0	0	2	3	4	5	6	7	8	9	10	11	pick 4

Which objects? Choice vector? 1101

Memo table vs Dynamic Programming

The dynamic programming solution computes ALL values BOTTOM UP, even ones that are not needed for the final result.

The memo table solution computes **only the necessary values** top down, **but needs to check** for each entry whether is has been computed yet.

How about the memory requirements for memo?

Knapsack Problem

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.

What would be a Greedy solution?

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)

What is the relation between Subset Sum and Knapsack?
Can you turn a Subset Sum problem into a knapsack problem?

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:
  - OPT includes best of { 1, 2, …, i-1 } using weight limit w
- Case 2: item i is included:
  - new weight limit = $w - w_i$
  - OPT includes best of { 1, 2, …, i-1 } using weight limit $w - w_i$

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

Knapsack Problem: Bottom-Up Dynamic Programming

Knapsack. Fill an n-by-W array.

```
Input: n, W, weights w1,…,wN, profits v1,…,vN

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

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

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	34	40

OPT: 40

How do we find the objects 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	34	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	34	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	34	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. $\Theta(nW)$.

- Not polynomial in input size!
  - W can be exponential in n
- "Pseudo-polynomial."

- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm.

- There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum.