CS146: Data Structures and Algorithms
Lecture 16

DYNAMIC PROGRAMMING
BRUTE FORCE
BACKTRACKING

INSTRUCTOR: KATERINA POTIKA
CS SJSU

Algorithmic Paradigms

- Greed. Build up a solution incrementally, myopically optimizing some local criterion.

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

- **Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

- Brute Force: Naïve, try all possible (slow)
- Backtracking: Consider sub-solutions.

Brute Force Algorithm

- Based on trying all possible solutions
- Approach
  - Generate and evaluate possible solutions until
    - Solution is found
    - Best solution is found (if can be determined)
    - All possible solutions found
      - Return best solution
      - Return false if no
- Generally the most expensive approach

- Example brute force sorting: try all permutations (n!) and check if any is sorted.

Backtracking Algorithm

- Based on depth-first recursive search of a tree
  - easy case: consider for each variable take it or not take it
- Intuition: It is often possible to reject a solution by looking at just a small portion of it. Incrementally grow a tree of partial solutions.
- Tree of alternatives → search tree

Steps of backtracking

- More abstractly, a backtracking algorithm requires a test that looks at a subproblem and quickly declares one of three outcomes:

    1. Failure: the subproblem has no solution.
    2. Success: a solution to the subproblem is found.
    3. Uncertainty, i.e. expand sub solution

Backtracking example: Sudoku

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

1. Find row, col of an unassigned cell
2. If there is none, return true
3. For digits from 1 to 9
4. a) If there is no conflict for digit at row,col assign digit to row,col and recursively try fill in rest of grid
5. b) If recursion successful, return true
6. c) Else, remove digit and try another
7. If all digits have been tried and nothing worked, return false

Dynamic Programming History

- Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ...

- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

Dynamic Programming – 3rd technique (Ch 15)

- Fibonacci numbers
- 0-1 Knapsack Problem
- Longest Common Subsequence (LCS)
- All pairs shortest paths

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory (table) for later use
- More efficient than *Brute-Force*, which solve the same subproblems over and over again

DP: three-step method

1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

Example: Fibonacci numbers

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- Computing the n^{th} Fibonacci number recursively (top-down):

$$F(n)$$

$$F(n\text{-}1) \quad + \quad F(n\text{-}2)$$

$$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad\qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$$

…

Example: Fibonacci numbers (cont.)

bottom-up iteration and recording results:

$F(0) = 0$

$F(1) = 1$

$F(2) = 1+0 = 1$

…

$F(n\text{-}2) =$

$F(n\text{-}1) =$

$F(n) = F(n\text{-}1) + F(n\text{-}2)$

0	**1**	**1**	**. . .**	$\boldsymbol{F(n\text{-}2)}$	$\boldsymbol{F(n\text{-}1)}$	$\boldsymbol{F(n)}$

Efficiency:

- time O(n)
- space O(n)

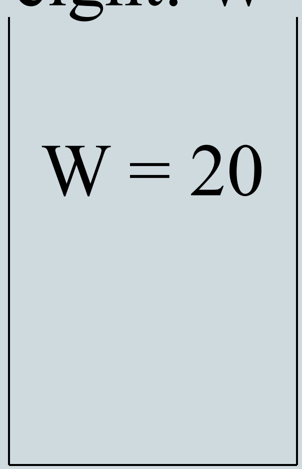
0-1 Knapsack problem (Ch 16.2): a picture

n items with weight & benefit

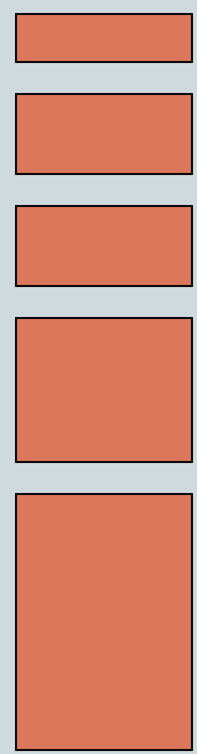
Pack knapsack:

- Holds up to W weight
- Maximum total benefit

Max weight: $W = 20$



Items



Weight w_i	Benefit value b_i
2	3
3	4
4	5
5	8
9	10

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

  - The problem is called a *"0-1" (or "discrete")* problem, because each item must be entirely accepted or rejected.
  - Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

The Knapsack Problem: Greedy Vs. Dynamic Programming

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - Greedy strategy: take in order of dollars/pound

- The optimal solution to the 0-1 problem **cannot** be found with the same greedy strategy but by ad dynamic programming

Counter Example of greedy algorithm for

Consider the 0-1 knapsack problem. What is a good example, with 3 items and W=50, that shows that being greedy does not provide the optimum profit Take the ratio b/w and sort take the two highest (only these fit) is it the best you can do? Or can you choose another pair with higher b's?

a. w=(10,20,30) and b=(40,100,150)
b. w=(10,20,30) and b=(60,100,120)

The 0-1 Knapsack Problem And Optimal Substructure

- Consider the most valuable load with at most W pounds
  - *If we remove item j from the load of the Knapsack, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most W - w_j that was packed, excluding item j

0-1 Knapsack problem: brute-force approach

- Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items (possible subsets).
  - 1 BIT for each element: 1 take it or 0 don't take it.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$
- *Is that fast?*

0-1 Knapsack problem: subproblem

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:
If items are labeled *1..n*, then a subproblem would be to find an optimal solution for
S_k = {*items labeled 1, 2, .. k*}

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that. Explanation follows....

Defining a Subproblem

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	?

Max weight: W = 20

For S_4:

Total weight: 14;

total benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 9$ $b_4 = 10$

For S_5:

Total weight: 20

total benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem (continued)

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the <u>exact</u> weight for each subset of items
- **The subproblem then will be to compute *B[k,w]***

Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**

2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k>w$. Item k can't be part of the solution, since if it was, the total weight would be > w, which is unacceptable
- Second case: $w_k <= w$. Then the item k <u>can</u> be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm (Exer 16.2-3)


```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
       if w_i <= w // item i can be part of the solution
           if (b_i + B[i-1,w-w_i] > B[i-1,w])
               B[i,w] = b_i + B[i-1,w- w_i]
           else
               B[i,w] = B[i-1,w]
       else B[i,w] = B[i-1,w]  // w_i > w
```

What is the running time of this algorithm?

$O(n*W)$

Constructing the optimal solution

- The previous algorithm does not keep track of which subset of items gives the optimal solution.
- To compute the actual subset
  - add an auxiliary boolean array **keep[i,w]**
  - keep[i,w]=1 if we take item i and 0 if we don't take it

Algorithm becomes…

if $(b_i + B[i-1,w-w_i] > B[i-1,w])$

$B[i,w] = b_i + B[i-1,w- w_i]$

keep[i,w]=1

else

$B[i,w] = B[i-1,w]$

keep[i,w]=0

Constructing the solution

How to we use keep?

- If keep[n,W]=1 then n-th item is taken and we repeat in keep[n-1, W-w_n] to find the remaining.
- If keep[n,W]=0 the n-th is not taken and we repeat to keep[n-1,W]

K=W

for i=n downto 1

if keep[i,K]=1

print i

K=K-w_i

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

item	weight	profit
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity j

		0	1	2	3	4	5
	0	0	0	0			
$w_1 = 2, v_1 = 12$	1	0	0	12			
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

Backtracing finds the actual optimal subset, i.e. solution.

Example Knapsack

- Let B = [1, 4, 3] and W = [1, 3, 2] be the array of profits and weights the 3 items respectively. Total Weight of knapsack is 4

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				

Example Knapsack

- Let B = [1, 4, 3] and W = [1, 3, 2]
- W=4

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

B[1,1]=max{B[0,1],B[0,0]+1}=1

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0				
3	0				

Example Knapsack

- Let B = [1, 4, 3] and W = [1, 3, 2]
- W=4

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

B[2,3]=max{B[1,3],B[1,3-3]+4}=4

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	4	
3	0				

Example Knapsack

- Let B = [1, 4, 3] and W = [1, 3, 2]
- W=4

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

B[2,4]=max{B[1,4],B[1,4-3]+4}=5

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	4	5
3	0				

Example Knapsack

- Let B = [1, 4, 3] and W = [1, 3, 2]
- W=4

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

B[3,2]=max{B[2,2],B[2,2-2]+3}=2

B[3,3]=max{B[2,3],B[2,3-2]+3}=max{4,1+3}=4

B[3,4]=max{B[2,4],B[2,4-3]+3}=max{5,1+3}=5

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	4	5
3	0	1	3	4	5

Example Knapsack

- Let B = [1, 4, 3] and W = [1, 3, 2]
- W=4

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

Solution tracking how the values changed

	0	**1**	**2**	**3**	**4**
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1	4	5
3	0	1	3	4	**5**