Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab

ONLINE SESSION 13-PLAN

innovate achieve lead

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
13	**Dynamic Programming - Design Principles and Strategy, Matrix Chain Product Problem, 0/1 Knapsack Problem, All-pairs Shortest Path Problem**	**T1: 5.3, 7.2**

Dynamic Programming

- Invented by a prominent U.S. mathematician, Richard Bellman
- The word “programming” in the name of this technique stands for “planning” and does not refer to computer programming.

Dynamic Programming

- Dynamic programming is a technique for solving problems with overlapping subproblems.
- Typically, given problem's solution can be related to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Dynamic Programming

- A straightforward application of dynamic programming can be interpreted as a special variety of space-for-time trade-off.
- Optimisation of plain recursion.

Dynamic Programming-Example

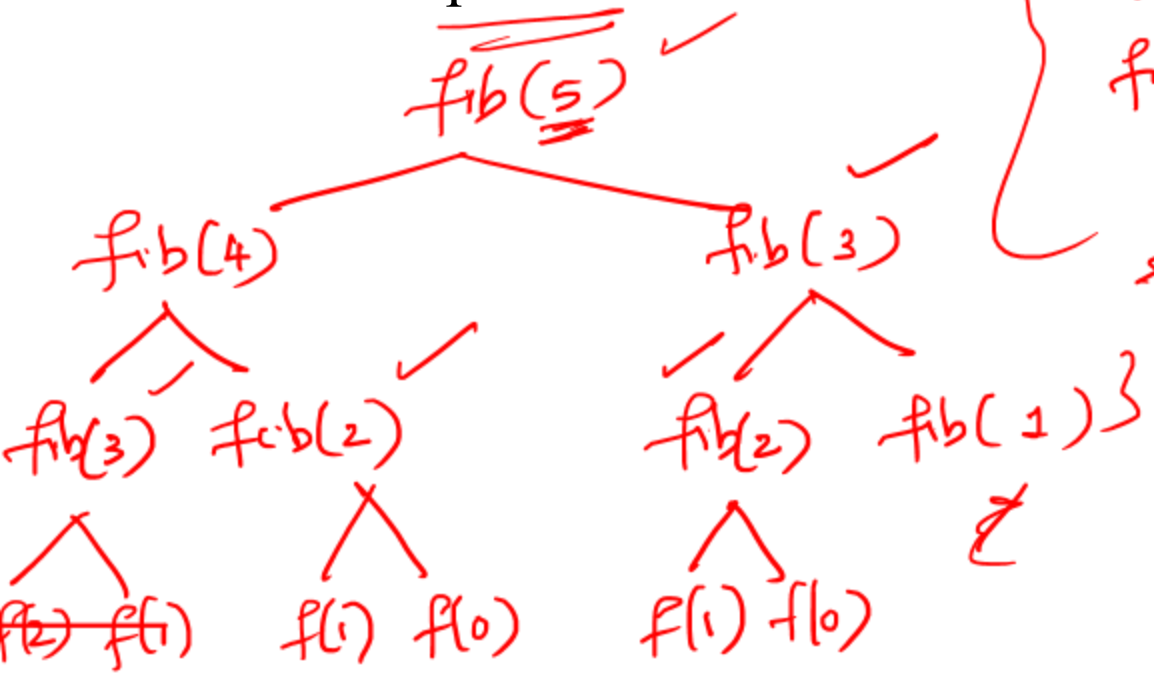
- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…
- F(n) = F(n-1) + F(n-2), F0 = 0 and F1 = 1

```
int fib(int n)
{
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```

$$T(n) = T(n-1) + T(n-2)$$

Dynamic Programming-Example

- Time complexity:
- T(n) = T(n-1) + T(n-2)
- which is exponential.



DP

O(n)

```
{
  f[0] = 0
  f[1] = 1
  for(i = 2; i <= n; i++)
    f[i] = f[i-1] + f[i-2]
  return f[n]
}
```

Dynamic Programming-Properties

Overlapping Sub-problems:

- Sub-problems needs to be solved again and again.
- In recursion we solve those problems every time and in dynamic programming we solve these sub problems only once and store it for future use

Optimal Substructure:

- A problem can be solved by using the solutions of the sub problems

Dynamic Programming-Example

```
Algorithm DynamicFibonacci(n)
{
 f[0]=0,f[1]=1
 for(i =2;i<=n;i++)
      f[i]=f[i-1]+f[i-2]
return f[n];
}
```

Time complexity??

Dynamic Programming-Example

- A man put a pair of rabbits in a place surrounded by a wall. How many pairs of rabbits will be there in a year if the initial pair of rabbits (male and female) are newborn and all rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male-female pair at the end of every month?

Dynamic Programming

- The circumstances and restrictions are not realistic.
- Still, this isn't THAT unrealistic a situation in the short term.

Dynamic Programming

- Similar to Fibonacci problem.
- To solve Fibonacci's problem, we'll let f(n) be the number of pairs during month n.
- By convention, f(0) = 0, f(1) = 1 for our new first pair.
- f(2) = 1 as well, as conception just occurred.
- The new pair is born at the end of month 2, so during month 3, f(3) = 2.
- Only the initial pair produces offspring in month 3, so f(4) = 3.

Dynamic Programming

- In month 4, the initial pair and the month 2 pair breed, so f(5) = 5. We can proceed this way, presenting the results in a table. At the end of a year, Fibonacci has 144 pairs of rabbits.

Month	0	1	2	3	4	5	6	7	8	9	10	11	12
Pairs	0	1	1	2	3	5	8	13	21	34	55	89	144

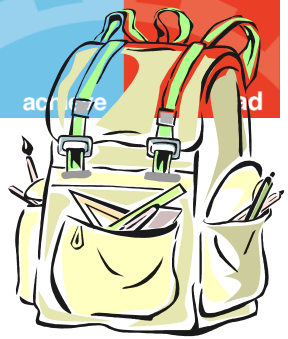
The General Dynamic Programming Technique

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

The 0/1 Knapsack Problem

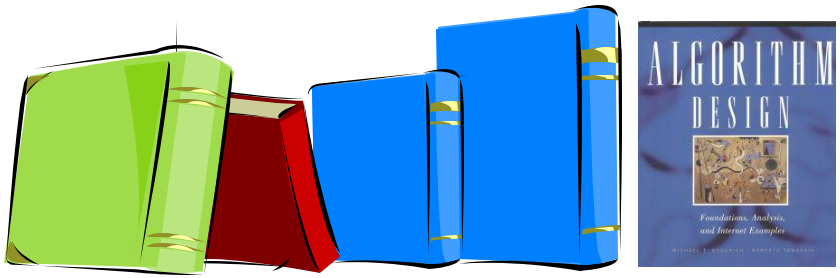
- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - p_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- This problem is called **a "0- 1 " problem**, because each item must be entirely accepted or rejected.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum p_i x_i$
  - Constraint: $\sum_{i \varepsilon T} w_i x_i \leq W$

Example



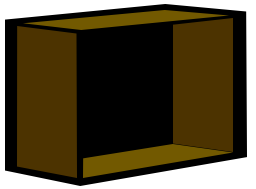
- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	4 kg	2 kg	2 kg	6 kg	2 kg
Benefit:	$20	$3	$6	$25	$80

"knapsack"



box of width 9 in

Solution:

- item 5 ($80, 2 kg)
- item 3 ($6, 2 kg)
- item 1 ($20, 4 kg)

Example

4 Items

W = 8 P = {1, 2, 5, 6}

n = 4 w = {2, 3, 4, 5}

B[3, 6-5] + Pk

B[3, 1] + Pk

B[3, 5-5]

B[3, 0] + Pk

0 + Pk

W →

k	n	0	1	2	3	4	5	6	7	8
P	w	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0	1	2	5	5	6	7	7
6	5	4	0	1	2	5	6	6		

B[4,2] B[4,3] B[4,4] B[4,6]

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

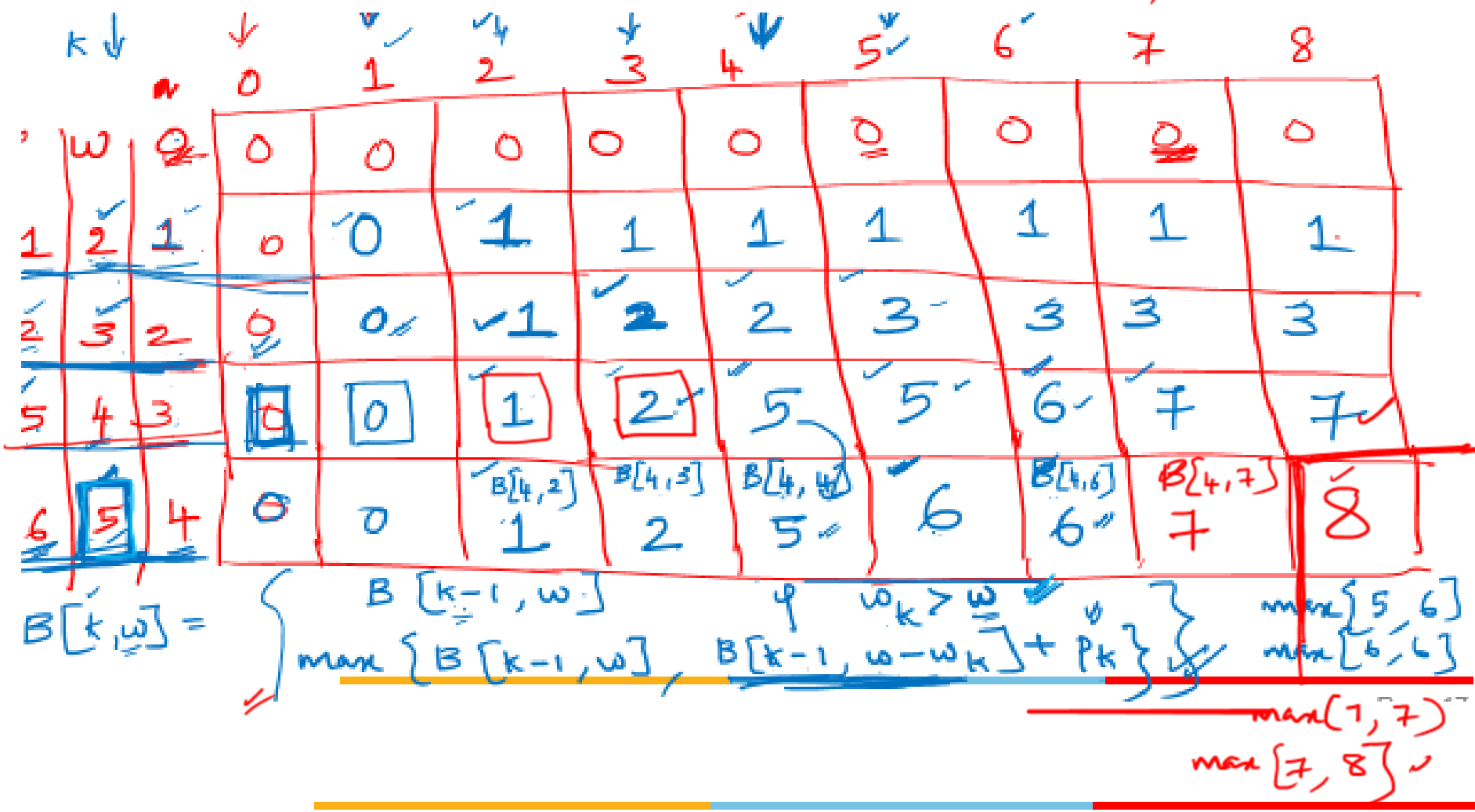
max{5, 6}

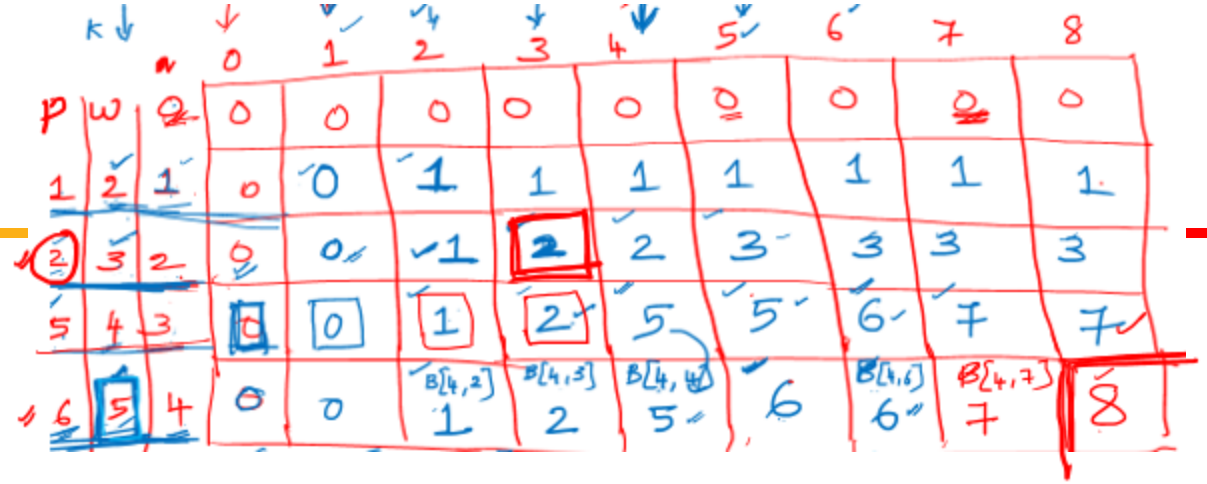
max{6, 6}

Example $O(nW)$

$B[3,8-5]+P_k$
$B(3,3)+6$
$2+6=8$

$B[3,7-5]+P_k$
$B[3,2]+6$
$1+6$





k \ n	0	1	2	3	4	5	6	7	8
p w	0	0	0	0	0	0	0	0	0
1 2 1	0	0	1	1	1	1	1	1	1
2 3 2	0	0	1	2	2	3	3	3	3
5 4 3	0	0	1	2	5	5	6	7	7
6 5 4	0	0	B[4,2] 1	B[4,3] 2	B[4,4] 5	6	B[4,6] 6	B[4,7] 7	8

$8 - 6 = 2$

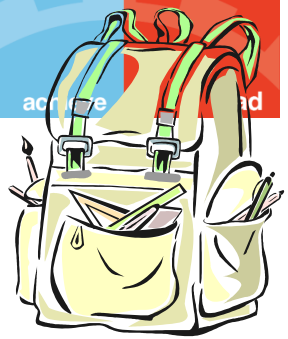
$2 - 2 = 0$

x_1, x_2, x_3, x_4

0 1 0 1

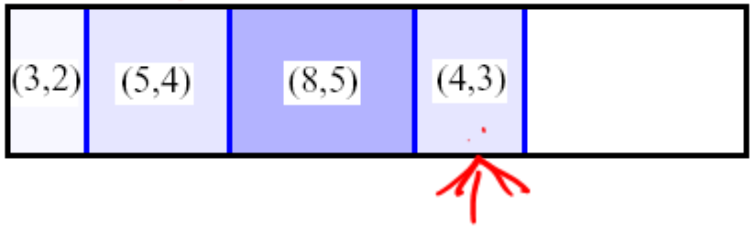
(1)

A 0/1 Knapsack Algorithm, First Attempt

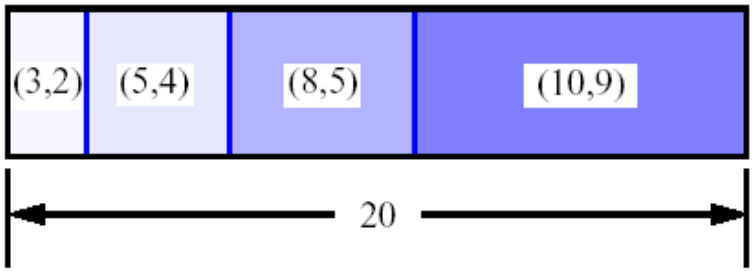


- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:



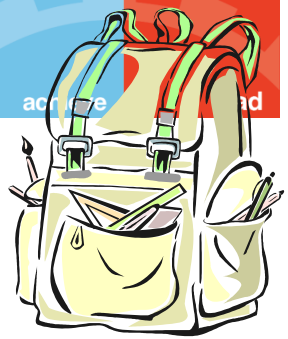
Best for S_5:



A 0/1 Knapsack Algorithm, First Attempt

- Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots I_k\}$.
- The solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

A 0/1 Knapsack Algorithm, Second Attempt

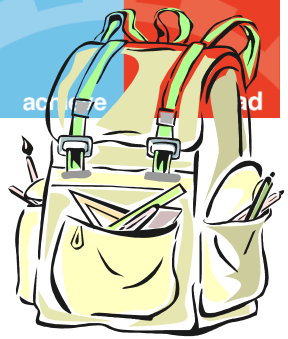


- S_k: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from S_k with weight at most w
- Good news: this does have subproblem optimality.

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

- I.e., the best subset of S_k with weight at most w is either
  - the best subset of S_{k-1} with weight at most w or
  - the best subset of S_{k-1} with weight at most $w-w_k$ plus item k
  -

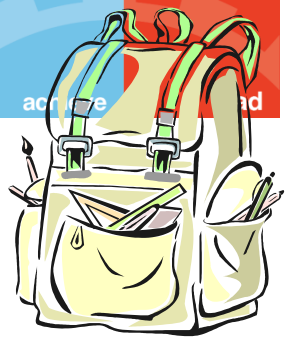
A 0/1 Knapsack Algorithm, Second Attempt



- **Case 1**
- The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight w, if item k weighs greater than w.

- Basically, you can NOT increase the value of your knapsack with weight w if the new item you are considering weighs more than w – because it WON'T fit!!!

A 0/1 Knapsack Algorithm, Second Attempt



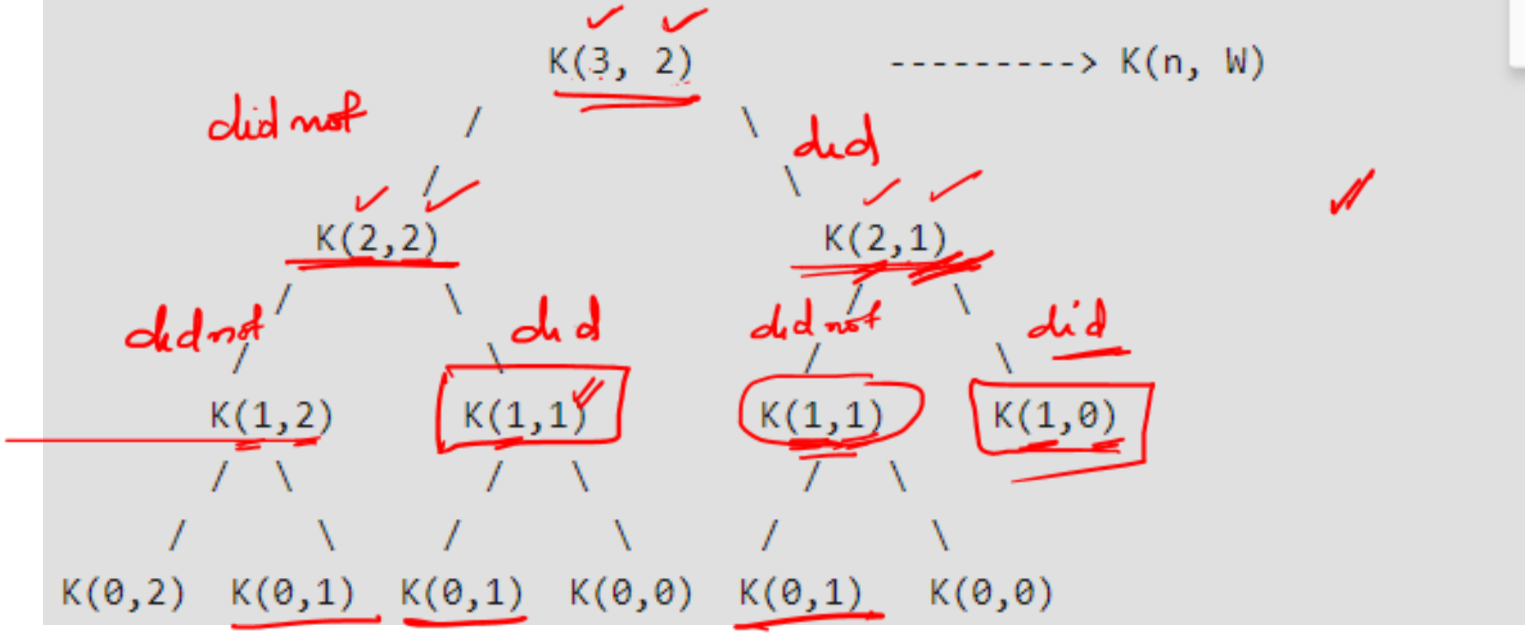
- **Case 2**
- The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight $w-w_k$, plus item k.

- You need to compare the values of knapsacks in both case 1 and 2 and take the maximal one.

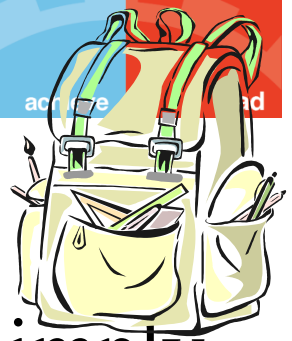
A 0/1 Knapsack Algorithm, Second Attempt

- Recursively, we will STILL have an $O(2^n)$ algorithm.

In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W. The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

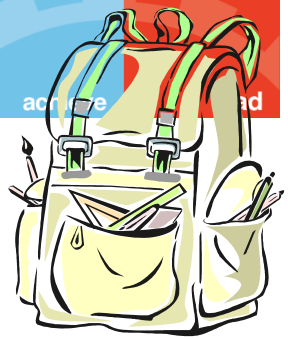


A 0/1 Knapsack Algorithm, Second Attempt



- But, using dynamic programming, we simply have to do a double loop - one loop running n times and the other loop running W times.

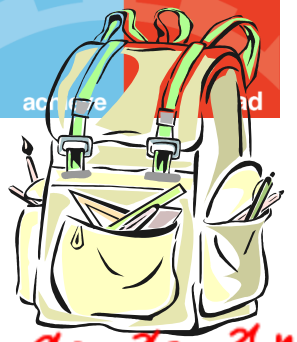
0/1 Knapsack Algorithm



0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n
    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
```

0/1 Knapsack Algorithm



pseudopolynomial time alg.

Running time

```
for w = 0 to W                    O(W)
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n                    Repeat n times
    for w = 0 to W                O(W)
        < the rest of the code >
```

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

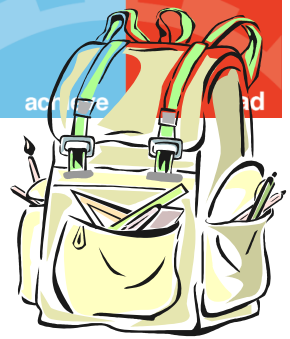
x_1	x_2	x_3	x_4
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
⋮			
1	1	1	1

$O(2^n)$

0/1 Knapsack Algorithm

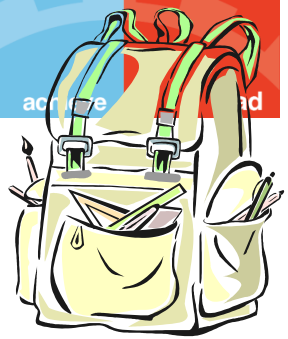
- Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops.
- Depending on W, either the dynamic programming algorithm is more efficient or the brute force **$O(2^n)$,** algorithm could be more efficient.
- (For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable.)

0/1 Knapsack Algorithm



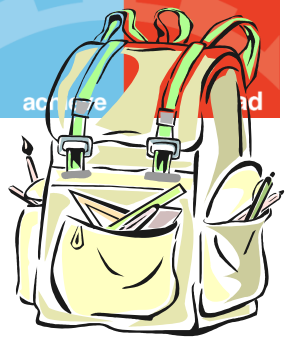
- **Pseudo-polynomial time algorithm**
- Running time of the algorithm depends on a parameter W that, strictly speaking, is not proportional to the size of the input (the n items, together with their weights and benefits, plus the number W).
- If W is very large (say $W = 2^n$), then this dynamic programming algorithm would actually be asymptotically slower than the brute force method.
- Thus, technically speaking, this algorithm is not a polynomial-time algorithm, for its running time is not actually a function of the size of the input

0/1 Knapsack Algorithm



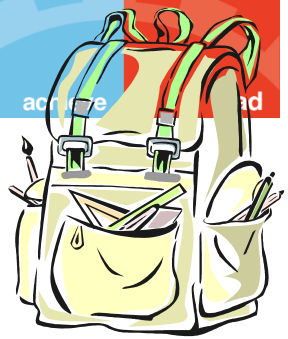
- **Pseudo-polynomial time algorithm**
- Running time depends on the magnitude of a number given in the input, not its encoding size

0/1 Knapsack Algorithm



Knapsack
Example Traced

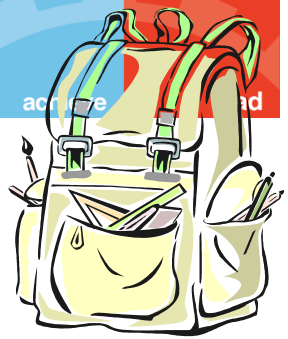
0/1 Knapsack Algorithm-Example2



- **W = 10**

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

0/1 Knapsack Algorithm-Example:Ans



Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

THANK YOU!

BITS Pilani
Hyderabad Campus