Dynamic Programming

0/1 knapsack, Recursive

How to identify either it is DP problem or not

- choice
- optimal
  - Max
  - Min
  - Larger
  - Smaller.

Recursive ⇒ memoize ⇒ ~~Top Down~~ Bottom up
(DP)

Recursive Code
⇓
make choice Diagram

After writing choice Diagram Code (writing will be easy)

- Dynamic programming is used to solve a wide variety of discrete optimization problems such as scheduling string-editing, packaging and inventory management
- Break problems into sub-problems and combine their solutions to larger problem.
- In contrasts to divide and Conquer these may be relationship accross subproblems.

0/1 knapsack problem.

- We are given a knapsack of Capacity c and a set of n objects numbered 1,2.---n. Each object i has w_i and profit P_i.

Steps to Design a Dynamic programming Algorithm

1. characterize optimal Structure
2. Recursively define the value of a optimal solution
3. Compute the value bottom up.
4. If needed Construct an optimal solution.

Knapsack

- Given n objects and a "Knapsack"
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$
- Knapsack has Capacity of W Kilograms.
- Goal: fill Knapsack So as to maximize total Value.

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

knapsack weight
$\boxed{W = 11}$

Ex: {3,4} has value 40

Greedy:- Repeatedly add item with maximum (v_i/w_i)

Ex {5,2,1} achieves only value of 35 ⇒

Greedy not optimal.

0-1 knapsack problems.

1) Subset Sum
2) Equal sum partition
3) Count of subset sum
4) minimum subset sub Diff
5) Target sum
6) # of subset & given d/f.

• all problems are inter connected each others

knapshake problem

- Fractional knapshake → Greedy
- 0/1
- Unbounded knapsack

Problem

	I_1	I_2	I_3	I_4
wt	1	3	4	5
val	1	4	5	7

Max profit.

• keep item such that profit should be max.

Bag (knapshak)

Constraints ⇒ Bag fix → fixed weight.

P_1	P_2	P_3	P_4
W_1	W_2	W_3	W_4

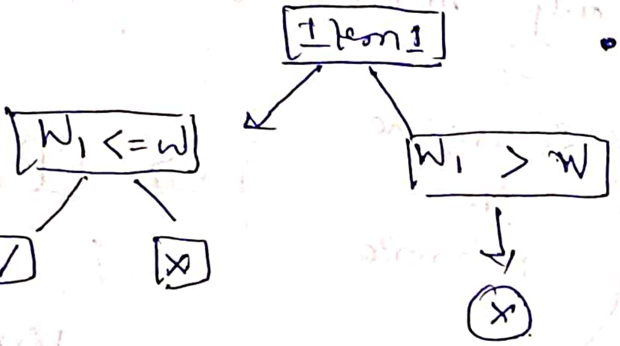
knapshak

- Fractional (half/divide) 0.5, 0.6 → Greedy
- 0/1 → keep/don't keep

• Unbounded.

item → Unlimited supply

Choice Diagram



{
- Base Cond is drop in recursive solutions
- Choice diagram
  ↳ code
}

<u>Algorithm</u>

→ return Max profit

int knapsack(int wt[ ], int val [ ], int W, int n)

wt Array · val Array · Array size

{

// think of Base Condition.

// Think of the Smallest Valid Input for getting BC

// array size may be zero, weight may be zero.

Base Cond:

```
if (n==0 || W==0)   // smallest Valid output
    return 0;
```

Choice Diagram:

```
if (wt[n-1] <= W)   // choice diagram.     (n-1 → last item)
return max( val[n-1] + knapsack( wt, val, W-wt[n-1]
    , n-1), knapsack( wt, val, W, n-1))
else if (wt[n-1] > W)
    return knapsack( wt, val, W, n-1);
```

- To memoize after writting recursive code it needs only 2 or more line of code.

Recursive + memoize ⇒ DP.

- W and n changes all the time, so we need to apply matrix (Bottom up to those (W and n))

knapsack (wt[ ], val[ ], W, n)

n → n-1

W → wt - wt[n-1]

int t[n+1][W+1]

↳ matrix initialize with -1.

memset (t, -1, size of(t))

-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

- Dp with Bottom up and memoize will have same time complexity

- apply memoization only for the parameter which changes.