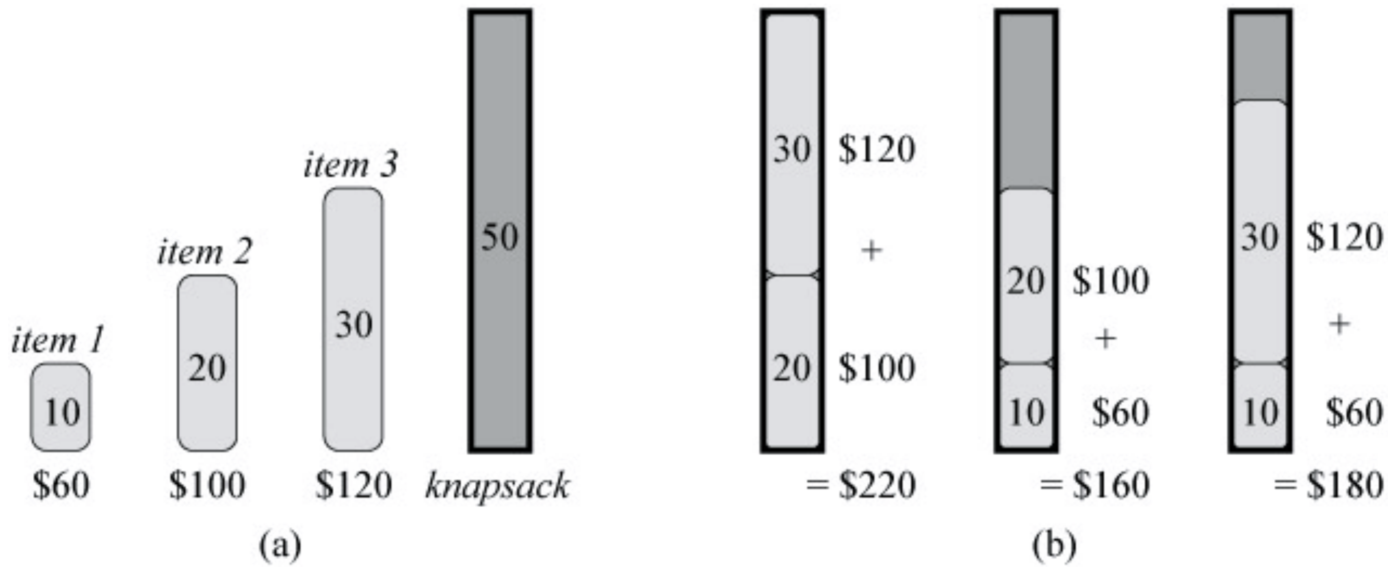
2D Dynamic Programming

2D because we use two-dimensional array to store solutions

The 0/1 Knapsack Problem



Input: A set of n items,

where item i has weight w_i and value v_i,

and a knapsack with capacity W.

Goal: Find $x_1, \ldots, x_n \in \{0,1\}$ satisfying $\sum_{i=1}^{n} x_i w_i \leq W$

that maximizes $\sum_{i=1}^{n} x_i v_i$.

Recall: Greedy doesn't provide optimal solution.

The Recurrence

First Attempt Definition: Let $V[w]$ be the largest obtainable value for a knapsack with capacity w.

First Attempt Recurrence:

If Optimal Solution for knapsack of size w chooses item i, remainder of optimal solution is optimal solution for subproblem of filling knapsack of size $w - w_i$ (1D solution coin denominations)

$$V[w] = \max(0,\ v_1 + V[w - w_1],\ v_2 + V[w - w_2], \ldots,\ v_n + V[w - w_n])$$
$$V[j] = 0, j \le 0$$

WRONG: This may pick the same item more than once! Non-legal Solution!

New 2D definition: Let $V[i, w]$ be the largest obtained value for a knapsack with capacity w, choosing ONLY from the first i items.

Recurrence:

Doesn't choose i

$$V[i, w] = \max(V[i-1, w],\ \ v_i + V[i-1, w - w_i])$$
$$V[i, w] = 0, i = 0\ or\ w = 0$$

Chooses i

So Far

Input: **A set of n items; item i has weight w_i and value v_i; a knapsack with capacity W.**
Goal: Find $x_1, \ldots, x_n \in \{0,1\}$ such that $\sum_{i=1}^{n} x_i w_i \leq W$ and $\sum_{i=1}^{n} x_i v_i$ is maximized.

Subproblem:

$V[i, w]$ is the largest obtained value for knapsack with capacity w, choosing ONLY from items $1, \ldots, i$

Recurrence: $V[i, w] = \max(V[i-1, w],\ \ v_i + V[i-1, w - w_i])$

With initial condition, $\forall i, \quad V[i, 0] = 0$

Find Order for filling in table: For i = 1 to n
For w = 1 to W

Required Solution: $V[n, W]$

DP is 2-Dimensional (2 variables) and not 1-D.

The Algorithm

```
let V[0..n, 0..W] be a new 2D array of all 0
for i ← 1 to n do
    for w ← 1 to W do
        if w[i] ≤ w and v[i] + V[i − 1, w − w[i]] > V[i − 1, w] then
            V[i, w] ← v[i] + V[i − 1, w − w[i]]
        else
            V[i, w] ← V[i − 1, w]
return V[n, W]
```

Running time: $\Theta(nW)$

Space: $\Theta(nW)$, but can be improved to $\Theta(n + W)$

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $keep[i, w]$

```
let V[0..n, 0..W] and keep[0..n, 0..W] be a new array of all 0
for i ← 1 to n do
    for w ← 1 to W do
        if w[i] ≤ w and v[i] + V[i − 1, w − w[i]] > V[i − 1, w] then
            V[i, w] ← v[i] + V[i − 1, w − w[i]]
            keep[i, w] ← 1
        else
            V[i, w] ← V[i − 1, w]
            keep[i, w] ← 0
K ← W
for i ← n downto 1 do
    if keep[i, K] = 1 then
        print i
        K ← K − w[i]
```

Running time: $\Theta(nW)$

Space: $\Theta(nW)$, cannot be improved to $\Theta(n + W)$ due to the $keep$ array.

Longest Common Subsequence

Problem: Given two sequences $X = (x_1, x_2, \dots, x_m)$ and $Y = (y_1, y_2, \dots, y_n)$, we say that $Z = (z_1, z_2, \dots, z_k)$ is a common subsequence of X and Y if $x_{i_p} = y_{j_p} = z_p$ for all $p = 1, 2, \dots, k$

where $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$.

The goal is to find the **longest common subsequence** of X and Y.

Example:

X: **A B A C B D A B**

Y: **B D C A B A**

Z: **B C B A**

The Recurrence

Observation: The problem is equivalent to finding the maximum matching between X and Y such that matched pairs don't cross.

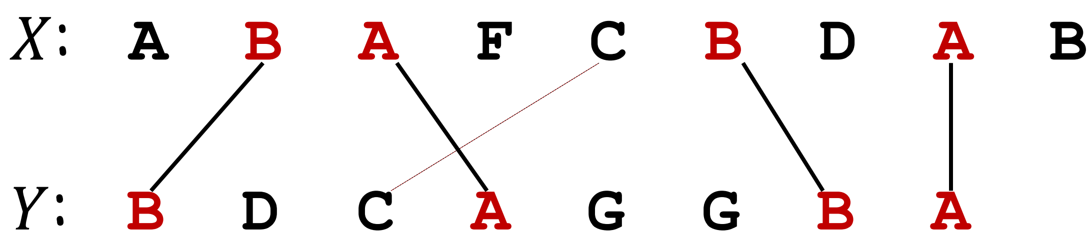
X: **A B A F C B D A B**

Y: **B D C A G G B A**

Z: **B C B A** is a solution

The Recurrence

Observation: The problem is equivalent to finding the maximum matching between X and Y such that matched pairs don't cross.



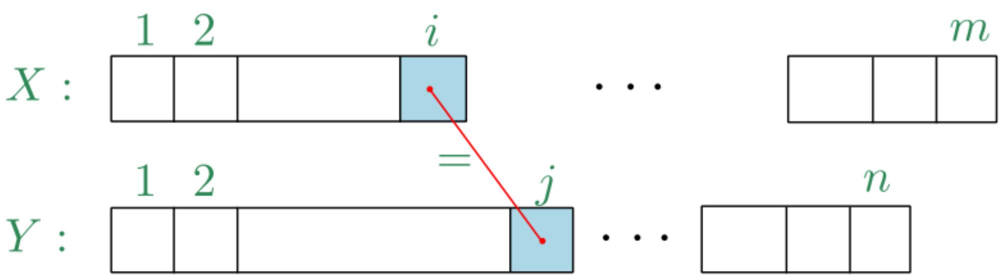
Z: **B C B A** is a solution

Z': **B A B A** is another legal solution

The Recurrence

Def: $c[i,j]$ is length of the longest common subsequence of $X[1..i]$ and $Y[1..j]$.

Observations: The problem is equivalent to finding the maximum matching between X and Y such that matched pairs don't cross.



The recurrence:

- Case 1: If $x_i = y_j$, then we match x_i and y_j.
- Case 2: If $x_i \neq y_j$, then either x_i or y_j is not matched.
  Optimal solution reduces to either $c[i-1,j]$ or $c[i,j-1]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

The Recurrence and Algorithm

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

```
let c[0..m, 0..n] and b[0..m, 0..n] be new arrays of all 0
for i ← 1 to m
    for j ← 1 to n
        if x_i = y_j then
            c[i,j] ← c[i-1, j-1] + 1
            b[i,j] ← "↖"                          MATCH x_i, y_j
        else if c[i-1, j] ≥ c[i, j-1] then
            c[i,j] ← c[i-1, j]                    x_i not matched
            b[i,j] ← "↑"
        else
            c[i,j] ← c[i, j-1]
            b[i,j] ← "←"                          y_j not matched
Print-LCS(b, m, n)
```

Running time: $\Theta(mn)$

Space: $\Theta(mn)$, can be improved to $\Theta(\min(m,n))$ if we only need to return the optimal length.

Reconstruct the Optimal Solution

```
Print-LCS(b,i,j):
    if i = 0 or j = 0 then return
    if b[i,j] = "↖" then
        Print-LCS(b,i − 1,j − 1)
        print x_i
    else if b[i,j] = "↑"
        Print-LCS(b,i − 1,j)
    else Print-LCS(b,i,j − 1)
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Value of $b[i,j]$ indicates whether

↖: x_i, y_j matched:
then write x_i and return LCS$(i-1, j-1)$

↑: x_i not matched
skip x_i and return LCS$(i-1, j)$

←: y_j not matched
skip y_j and return LCS$(i, j-1)$

Longest Common Substring

Problem: Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$, we wish to find their longest common substring Z, that is, the largest k for which there are indices i and j with $x_i x_{i+1} \dots x_{i+k-1} = y_j y_{j+1} \dots y_{j+k-1}$.

Ex:

X : **DEADBEEF**

Y : **EATBEEF**

Z : **BEEF** //pick the longest contiguous substring

Note: Brute-force algorithm takes $O(n^4)$ time.

Different from LCS problem because, in this problem, letters have to be together.

The Recurrence

Def: $d[i,j]$ = the length of the longest common substring of $X[1..i]$ and $Y[1..j]$. (Does this work?)

Def: $d[i,j]$ = the length of the longest common substring of $X[1..i]$ and $Y[1..j]$ that ends at x_i and y_j.

Q: Wait, are we changing the problem?

A: Yes, but it's OK. Optimal solution to the original is just $\max_{i,j}\{d[i,j]\}$

Recurrence:

- If $x_i = y_j$, then the LCS of $X[1..i]$ and $Y[1..j]$
  is just the LCS of $X[1..i-1]$ and $Y[1..j-1]$, plus $x_i = y_j$
- If $x_i \neq y_j$, then there can't be a common substring ending at x_i and y_j!

$$\boxed{d[i,j] = \begin{cases} d[i-1,j-1]+1 & \text{if } x_i = y_j \\ 0 & \text{if } x_i \neq y_j \end{cases}}$$

The Algorithm

```
let d[0..m, 0..n] be a new array of all 0
l_m ← 0, p_m ← 0
for i ← 1 to m
    for j ← 1 to n
        if x_i = y_j then
            d[i, j] ← d[i − 1, j − 1] + 1
            if d[i, j] > l_m then
                l_m ← d[i, j]
                p_m ← i
for i ← p_m − l_m + 1 to p_m
    print x_i
```

Note: For this problem, reconstructing the optimal solution just needs the location of the LCS.

Running time: $\Theta(mn)$

Space: $\Theta(mn)$ but can be improved to $\Theta(\min(m, n))$.

Exercise on Edit Distance

Given two strings s and t, the edit distance $edit(s,t)$ is the smallest number of following edit operations to turn s into t:

Insertion: add a letter

Deletion: remove a letter

Substitution: replace a character with another one.

Example: $s =$ abode and $t =$ blog.

Then, $edit(s,t) = 4$ operations

Start from abode

1 delete a ⇒ bode

2 insert l after b ⇒ blode

3 delete d ⇒ bloe

4 substitute e with g ⇒ blog

Impossible to do so with at most 3 operations.

Exercise on Edit Distance (cont)

Let s and t be two strings with lengths m and n, respectively.
Case 1 If $m = 0$, then $edit(s, t) = n$.
Case 2 If $n = 0$, then $edit(s, t) = m$.
Case 3 If $m > 0, n > 0, \text{and } s[m] = t[n]$, then

$$edit(s,t) = min\begin{cases} 1 + edit(s[1..m], t[1..n-1]) \\ 1 + edit(s[1..m-1], t[1..n]) \\ edit(s[1..m-1], t[1..n-1]) \end{cases}$$

Explanation of Case 3

- Delete $t[n]$, and use the least number of edit operations to change $s[1..m]$ into $t[1..n-1]$. The total number of edit operations is therefore $1 + edit(s[1..m], t[1..n-1])$. (example: $s = abc$, $t = abcc$)
- Delete $s[m]$, and use the least number of edit operations to change $s[1..m-1]$ into $t[1..n]$. The total number of edit operations is therefore $1 + edit(s[1..m-1], t[1..n])$. (example: $s = abcc$, $t = abc$)
- Simply change $s[1..m-1]$ into $t[1..n-1]$. The total number of edit operations is therefore $edit(s[1..m-1], t[1..n-1])$. (example: $s = abcc$, $t = abcc$)

Exercise on Edit Distance (cont2)

Case 4 If $m > 0, n > 0, \text{and } s[m] \neq t[n]$, then

$$edit(s,t) = min \begin{cases} 1 + edit(s[1..m], t[1..n-1]) \\ 1 + edit(s[1..m-1], t[1..n]) \\ 1 + edit(s[1..m-1], t[1..n-1]) \end{cases}$$

Lets store $edit(i,j)$ in an array $E[i,j]$. Then

$$E[i,j] = min \begin{cases} 1 + E[i, j-1] \\ 1 + E[i-1, j] \\ \begin{cases} E[i-1, j-1], & \text{if } s[i] = t[j] \\ 1 + E[i-1, j-1], & \text{if } s[i] \neq t[j] \end{cases} \end{cases}$$

DP Algorithm for filling array E

1 Fill in row 0 and column 0.
2 Fill in the cells of row 1 from left to right.
3 Fill in the cells of row 2 from left to right.
4 ...
5 Fill in the cells of row m from left to right.