Dynamic Programming

Longest common subsequence problem

We are given 2 sequences $X = (x_1, x_2, \ldots x_m)$ and $Y = (y_1, y_2 \ldots y_n)$ and wish to find a maximum length common subsequence of X and Y.

eg. $X = (A, B, C, B, D, A, B)$
$Y = (B, D, C, A, B, A)$

Sequence (B, C, A) is a common subsequence of both X and Y and length = 3.

Sequence (B, C, A, B) is a common subsequence to both X and Y and length = 4.

* A simple algorithm to solve LCS problem is to enumerate all sequences of X and check each subsequence whether it is also a subsequence of Y, keeping track of LCS found.

There are 2^m subsequences of X, so this approach requires exponential time.

* LCS problem has an optimal substructure property.

Let $X = (x_1, x_2 \ldots x_m)$
$Y = (y_1, y_2 \ldots y_n)$

Let $Z = (z_1, z_2 \ldots z_k)$ be any LCS of X and Y.

(1) If $x_m = y_n$ then $z_k = x_m = y_n$ and z_{k-1} is an LCS of X_{m-1} and Y_{n-1}

(2) If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.

(3) If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

eg. (1) $X = (A, B, C, B, A, C, D)$
$Y = (E, C, F, A, D)$
$Z = (C, A, D)$

Now $x_6 = D = y_5 = z_3$
$\therefore Z_2 = (C, A)$ is an LCS of (A, C, B, A, C) and (E, C, E, A)

(2) Let $X = (A, C, B, A, C, D, E)$
$Y = (E, C, F, A, D)$
LCS $Z = (C, A, D)$
$x_7 = E \neq y_5$
but $z_3 \neq x_7$
∴ Z is LCS of (A, C, B, A, C, D) and (E, C, F, A, D)
$c[i-1, j]$

(3) Let $X = (A, C, B, A, C, D)$
$Y = (E, C, A, D, F)$
$Z = (C, A, D)$
$x_6 \neq y_5$ and $z_3 \neq y_5$
∴ Z is an LCS of (A, C, B, A, C, D) and (E, C, A, D)

Solving LCS problem using dynamic programming

We are finding LCS of $X = (x_1, x_2 \ldots x_m)$ and $Y = (y_1, y_2 \ldots y_n)$

If $x_m = y_n$, we must find LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields LCS of X and Y.

Let $c[i, j]$ be length of an LCS of sequences $X_i = (x_1, x_2 \ldots x_i)$ and $Y_j = (y_1, y_2 \ldots y_j)$. If last elements of X and Y are same, then:-

$c[i, j] = c[i-1, j-1] + 1$

if $i = 0$ or $j = 0$, then $c[i, j] = 0$

if $i, j > 0$ and $x_i \neq y_j$, we must solve 2 subproblems:-

(1) Finding an LCS of X_{m-1} and Y.
(2) Find an LCS of X and Y_{n-1}.

whichever of these 2 LCS is larger is an LCS of X and Y.

$$c[i, j] = \max \{ c[i, j-1], c[i-1, j] \}$$

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

we also maintain the table $b[1..m, 1..n]$ to simplify the construction of an optimal solution.

Algo

LCS_length (X, Y, m, n) — m: no. of elements in sequence X; n: no. of elements in sequence Y.

```
for i ← 0 to m
    do c[i,0] ← 0
for j ← 0 to n
    do c[0,j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if xi = yj
then c[i,j] ← c[i-1,j-1] + 1
     b[i,j] ← "↖"
else if c[i-1,j] ≥ c[i,j-1]
     then c[i,j] ← c[i-1,j]
          b[i,j] ← "↑"
     else c[i,j] ← c[i,j-1]
          b[i,j] ← "←"
return c and b
```

Running time $= O(mn)$

eg. Let $X = (D, A, C, B, E)$

$Y = (A, F, C, B)$

		0	A 1	F 2	C 3	B 4
	0	0	0	0	0	0
D	1	0	0↑	0↑	0↑	0↑
A	2	0	1↖	1←	1←	1←
C	3	0	1↑	1↑	2↖	2←
B	4	0	1↑	1↑	2↑	3↖
E	5	0	1↑	1↑	2↑	3↑

Length of LCS = 3

LCS = A C B

$c[1,1] = \max\{c[1,0], c[0,1]\}$
$= \max(0,0) = 0$

$c[1,2] = \max\{c[1,1], c[0,2]\}$
$= \max(0,0) = 0$

$c[2,1] = c[1,0] + 1$
$= 0 + 1 = 1$

For table

* Initially when $i=0$ and $j=0$, then $c[i,j]=0$ i.e 1st row & 1st column becomes 0.

* If $x_i \neq y_j$, then check top & left element & write max^m element (and put ↑ or ←) By default: "↑"

* If $x_i = y_j$, (eg. $x_2=y_1$, 1^{st} A's) $c[2,1] = c[1,0]+1 = 0+1 = 1$) ie if $x_i = y_j$, then 1 is added to diagonal element & put "↖"

* For length of LCS and to find LCS. Length of LCS is 3(4) ie last entry.

* To find LCS, walk from last entry in direction of arrows & write only diagonal arrow entries in reverse order.

Algo for constructing LCS:

```
Print_LCS (b, X, i, j)
  if i=0 or j=0
  then return
  if b[i,j] = "↖"
  then print_LCS (b, X, i-1, j-1)
  Print xi
  else if b[i,j] = "↑"
  then Print_LCS (b, X, i-1, j)
  else Print_LCS (b, X, i, j-1)
```

Running time = $O(m+n)$

Initial call is Print_LCS (b, X, m, n)

Find LCS of X = A B C B D A B
Y = B D C A B A

		0	B 1	D 2	C 3	A 4	B 5	A 6
	0	0	0	0	0	0	0	0
A	1	0	0↑	0↑	0↑	1↖	1←	1↖
B	2	0	1↖	1←	1←	1↑	2↖	2←
C	3	0	1↑	1↑	2↖	2←	2↑	2↑
B	4	0	1↖	1↑	2↑	2↑	3↖	3←
D	5	0	1↑	2↖	2↑	2↑	3↑	3↑
A	6	0	1↑	2↑	2↑	3↖	3↑	4↖
B	7	0	1↖	2↑	2↑	3↑	4↖	4↑

i ↓

Greedy algorithms

* A greedy algo always makes the choice that looks best at the moment.
* These makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
* The greedy algos do not always yield optimal solution, but for many problems they do.

Activity selection problem

* Suppose we have a set $S = \{1, 2, 3, \ldots n\}$ of n proposed activities that wish to use a resource such as a lecture hall, which can be used by only one activity at a time.
* Each activity i has a start time s_i and a finish time f_i where $s_i \leq f_i$.
* If selected, activity i takes place during half open time interval $[s_i, f_i)$
* Activities i and j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap ie if $s_i \geq f_j$ or $s_j \geq f_i$
* Activity selection problem is to select a maximum size set of mutually compatible activities.

We assume that input activities to this problem are in sorted order of increasing finishing time. ie

$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_n$.

```
Greedy-Activity-selector (s, f)
n ← length [s]
A ← {1}
j ← 1                          j = 1
for i ← 2 to n
    do if s_i ≥ f_j
        then A ← A ∪ {i}
             j ← i
return A
```

Set A collects selected activities.
j ← specifies most recent addition to A.
Initially, j = 1, ie A contains 1st activity.

Condition is that its start time s_i is not earlier than finish time f_j of activity most recently added to A.

Running time if not sorted by finish time = $O(n \lg n)$

Running time if sorted by finish time = $\Theta(n)$

Proving greedy algo correct

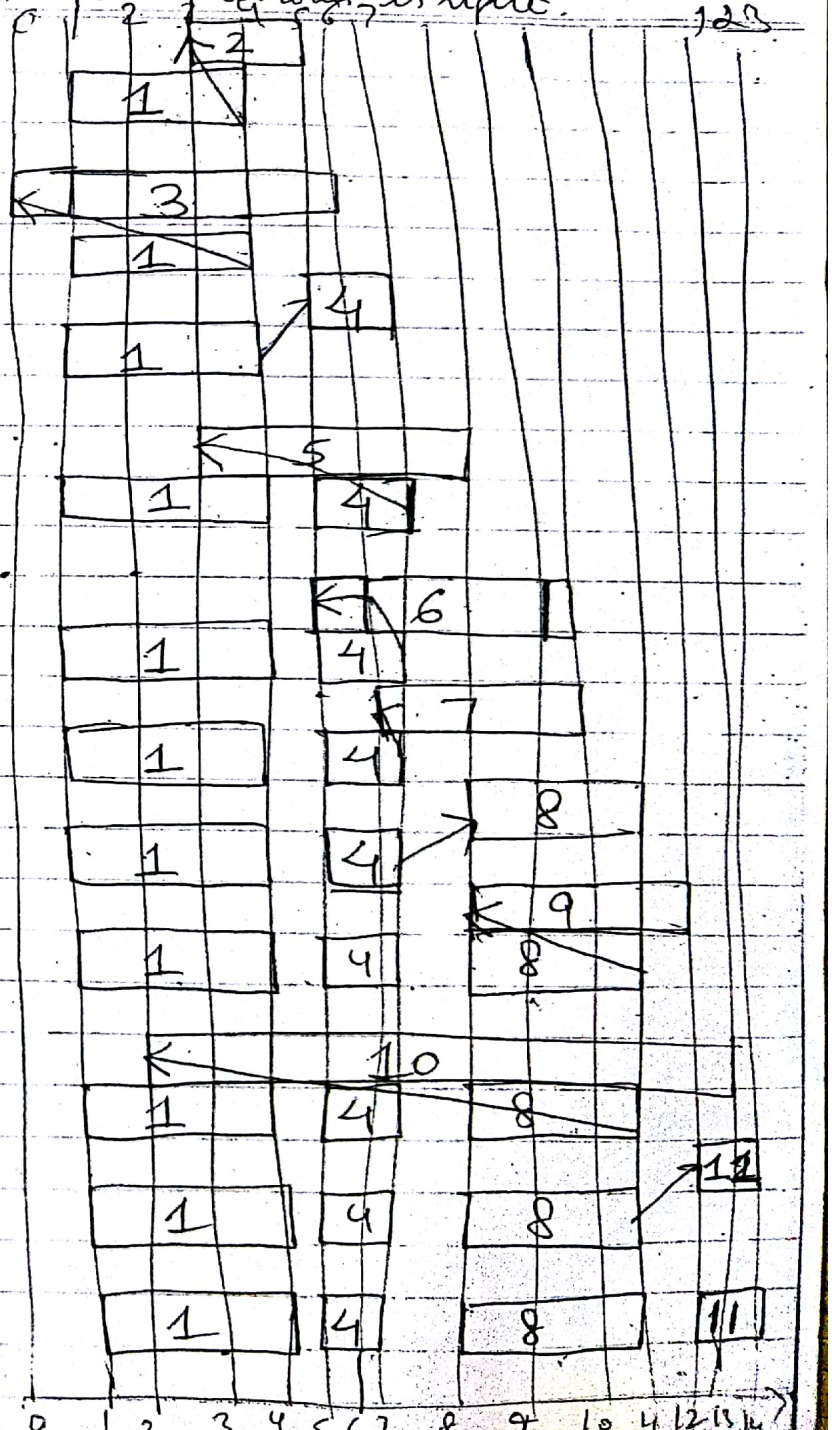
Greedy algo do not always produce optimal solution. However, Greedy-activity-selector always finds an optimal solution to an instance of activity selection problem.

Theorem:-

Algorithm Greedy-Activity-Selector produces solution of maximum size for activity selection problem.

Each row corresponds to an iteration of for loop. Activity is rejected if s_i occurs before finishing time f_j (arrows left). Activity accepted if arrow is right.

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



Proof: (1) Let $S = \{1, 2, \ldots n\}$ be set of activities to schedule (2) As we are assume that activities are in order of finish time. Activity 1 has earlier finish time.

We wish to show that there is an optimal solution that begins with a greedy choice i.e with activity 1.

(3) Suppose that $A \subseteq S$ is an optimal solution to given instance of activity selection problem + activities in A are ordered by finish time.

(4) Suppose 1st activity of A is activity k. If $k = 1$, then schedule A begins with a greedy choice.

(5) If $k \neq 1$, we want to show that there is another optimal solution B to S that begins with greedy choice, activity 1.

(6) Let $B = A - \{k\} \cup \{1\}$
Next $f_1 \leq f_k$ and the activities in B are disjoint and since B has the same no. of activities as A, it is also optimal.

Thus B is an optimal solution for S that contains greedy choice of activity 1.

If A is an optimal solution to original problem S, then $A' = A - \{1\}$ is an optimal solution to activity selection problem $S' = \{i \in S : s_i \geq f_1\}$

ie, when greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for activity-selection problem over those activities in S that are compatible with activity 1.

eg. If $k \neq 1$
eg. $A = \{2, 8, 14\}$
Let $B = \{1, 8, 14\}$
$f_1 \leq f_2$

For greedy strategy to be applicable problem must have:-

Elements of greedy strategy
(Two ingredients that are exhibited by most problems that lend themselves to greedy strategy are:-)

(1) Greedy choice property:- ie a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Diff' b/w DP + greedy

(a) But in dynamic programming we make a choice at each step + choice depends on solutions to subproblems. And in greedy we make choice that looks best at the moment (it does not depend on solution to subproblems)

(b) DP is bottom up technique
Greedy is top-up strategy where one greedy choice is made after another, iteratively reducing each given problem to smaller one.

(2) Optimal substructure:- If optimal sol" to problem contains within it optimal solutions to subproblems. → common to D.P + greedy.
Problem has optimal substructure

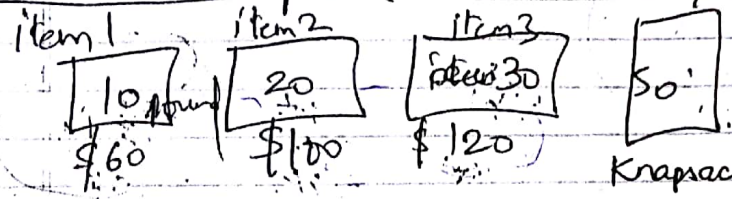
eg. activity selection problem if optimal solution A begins with activity 1, then set of activities $A' = A - \{1\}$ is an optimal solution to activity selection problem
$S' = \{i \in S : s_i \geq f_1\}$.

Greedy VS D.P.

0-1 knapsack problem:- A thief robbing a store finds n items, ith item is worth v_i dollars + weighs w_i pounds where v_i and w_i are integers. He can carry atmost W pounds. Thief can take either an item or not (not fractional amount).

In fractional knapsack problem, setup is same but thief can take fractions of items, rather than having to make binary choice for each item.

Both knapsack problems exhibit optimal-substructure property. But fractional knapsack problem is solvable by greedy strategy whereas 0-1 knapsack is not. 0-1 knapsack is solvable by D.P.



0-1 Knapsack problem

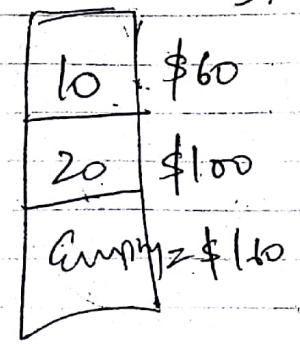
We calculate v_i/w_i per item

Item 1 = $\frac{60}{10}$ = 6 dollar/pound

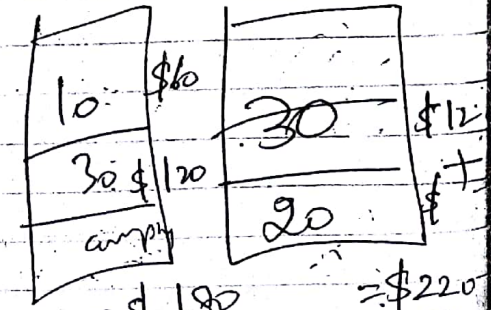
Item 2 = $\frac{100}{20}$ = 5 dollar/pound

Item 3 = $\frac{120}{30}$ = 4 dollar/pound

Greedy solution



'Other' solution

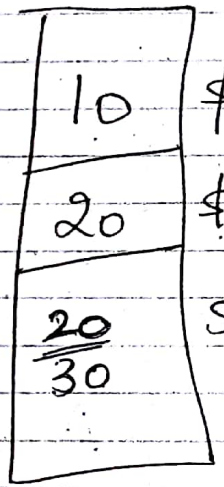


= $180

= $220

Greedy takes item 1 and 2, but optimal takes 2 and 3 leaving 1, so 0-1 knapsack cannot be solved using greedy.

Fractional Knapsack



= $240

(1, 1, 2/3)

$10 + 20 + \frac{2}{3} \times 30$

$= 30 + 20$

$= 50$

$\frac{2}{3} \times 30 = 20$

DP

0 1 KP

Fractional KP