bottom up is always more efficient cause top down has the overhead of every recursive call

top down is seen because it is an extension of recursive divide and conquer appraoch but we always prefer bottom up approach

top down needs initialisation of dictionary before execution, but not bottom up

Time complexity of both are same

In some cases top down could be faster than bottom up but not by time complexity

DP

- better divide and conquer where the value of each subproblem may be computed multiple times by storing values of already solved subproblems.

Fibonacci

- no of recursive calls : $2^{n/2+1} - 1 < R < 2^n - 1$
  - exponential time algorithm : $O(2^n)$

DP replaces exponential time complexity w/ polynomical time computation

- similar to DFS where we only explore edges to undiscovered vertex

DP Top Down

1. formulate P in terms of smaller version of the problem (recursively) : Q_1, Q_2
2. make recursive function to solve P
3. use dictionary to store solutions to subproblems
4. Before recursive call on Q_i, check if soln exists
   Before returning, store soln

(dict)

Soln must be initialised (all to 1) before running

↓

```
int fibDP(n) {
    int f1, f2;
    if (n == 0 || n == 1) {
        store(Soln, n, n);          → soln[n] = n
        return n;
    } else {
        if (notmember(Soln, n-1))   → soln[n-1] = -1?T
            f1 = fibDP(n-1);          O(1)
        else
            f1 = retrieve(Slon, n-1);   return soln[n-1]
                                        O(1)

        if (not member(Soln, n-2))  → O(1)
            f2 = fibDP(n-2);
        else
            f2 = retrieve(Soln, n-2);   O(1)
    }
    ft t = f2;
    store(Soln, n, f1);             → O(1)
    return f1;
}
```

time complexity ∝ no of times fibDP is called

↓

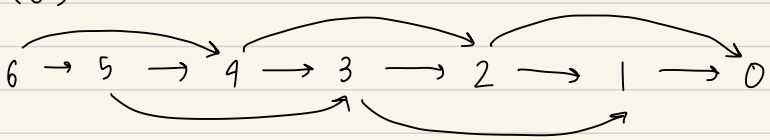
O(n+1) times

DP Bottom Up

- subproblem graph : for recursive algorithm A, the graph for A is a directed graph whose vertices are the instances for this problem.

  for directed edge (I, J) is A is called on I, it makes a recursive call directly on instance J

fib(6)



1. formulate problem
2. turn formulation into a recursive function to solve problem P
3. Draw subproblem, find dependencies.
4. use dictionary to store solutions
5. in iterative fn, solve P

   ① compute solutions of subproblem of the problem first

   ② solution to P is computed based on the solutions to it's subproblems and is stored into the dictionary.

```
soln[0] = 0
soln[1] = 1
for j = 2 to n
    soln[j] = soln[j-1] + soln[j-2];
```

n-1 loop.

Time complexity : $O(n)$

Longest common sequence

- given a sequence $s = \langle s_1, s_2, \dots, s_n \rangle$ a subsequence is any sequence $\langle s_{i_1}, s_{i_2}, \dots, s_{i_m} \rangle$ with i_j strictly increasing

- example : S : ACTTGCG
  → always pick increasing
  subsequence : ACT, AG, CC
  TCA ← not.

- given 2 sequences x and y, a common subsequence is a subsequence of both x and y

- LCS is the longest such sequence ← may not be unique

- a sequence of n elements can have 2^n subsequence and we check if they are subsequences of y

- time complexity = $O(m2^n)$
  ↑
  optimize

DP solves optimisation problems that satisfy the principal of optimality
↓
if subsolutions of an optimal solution of the problem are themselves optimal solutions of their subproblems

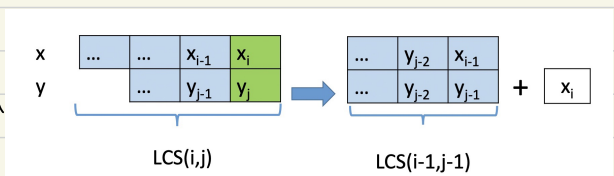
DP selection rule : given a choice, compute all, take best

Does LCS satisfy the principle

Step 1. formulate P in terms of smaller versions of the problem

$X = \langle x_1, x_2, \ldots x_i \rangle$
$Y = \langle y_1, y_2 \ldots y_j \rangle$

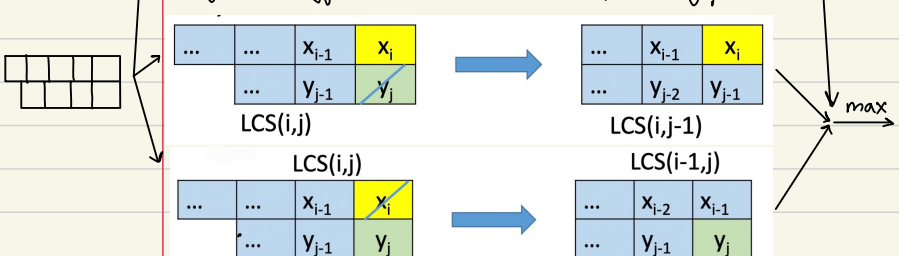
(A) if $x_i = y_j$ then that element belongs in LCS.
So, LCS is the LCS of $\langle x_1 \ldots x_{i-1} \rangle$ and $\langle y_1 \ldots y_{j-1} \rangle$ + x_i



subsolution of LCS(i,j) when $x_i = y_j$ and
LCS(i-1, j-1) is an optimal solution

(B) if $x_i \neq y_j$ then either one of them (or both) are not in the LCS

if x_i is not in the LCS, just find LCS for $\langle x_1 \ldots x_{i-1} \rangle$ & $\langle y_1 \ldots y_j \rangle$; same for y_j



Step 2 formulate recursive function

$LCS(i,j) = 0$ $\quad i=0$ or $j=0$

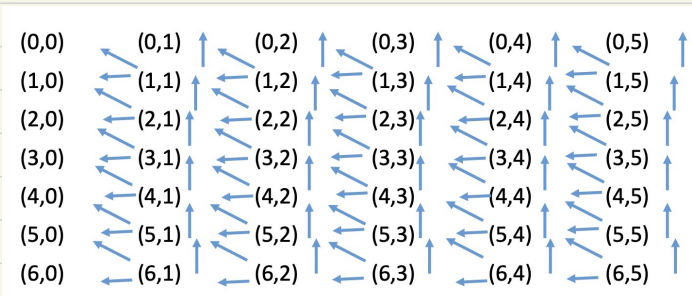
$LCS(i,j) = LCS(i-1, j-1) + 1$ $\quad$ if $i,j > 0$, $x_i = y_j$

$LCS(i,j) = \max(LCS(i-1,j), LCS(i,j-1))$ $\quad$ if $i,j > 0$, $x_i \neq y_j$

top down using recursion will be very inefficient

∴ decide to do bottom up (iterative)

Step 3 Bottom up, draw subproblem graph and find dependencies.



Step 4 $n+1$ by $m+1$ array

initialise $(0,0)$

compute row 1 to row n, column 1 to column m per row

code →

```
int LCS (n, m) {

    for i = 0 to n     c[i][0] = 0;
    for j = 1 to m     c[0][j] = 0;
    for i = 1 to n
        for j = 1 to m
            if x[i] == y[j]
                c[i][j] = c[i-1][j-1] + 1;
            else if c[i-1][j] >= c[i][j-1]
                c[i][j] = c[i-1][j];
            else c[i][j] = c[i][j-1];
    return c[n][m];
}
```

mn iterations

constant time O(1)

space : O(mn) [n+1 x m+1]

time : O(mn)

hint array : indicates where LCS(i,j-1), LCS(i-1,j) or LCS(i-1, j-1) the optimal soln comes from

eg. LCS(i,j) = LCS(i-1, j-1) + 1 → h[i][j] = '\'

LCS(i,j) = LCS(i-1, j) → h[i][j] = '|'

LCS(i,j) = LCS(i, j-1) → h[i][j] = '—'

first row : h[0][i] = '—'

first column : h[i][0] = '|'

```
getSequence(n, m){   // get LCS from hint array
    s = empty stack;          // cause backwards
    i = n
    j = m
    while(i ≠ 0 and j ≠ 0){   ← max n+m iterations
      if( h[i][j] == '\'){
            s.push(X[i]);
            i--;
            j--;
      }
      else if( h[i][j] == '|')
            i--;
      else  j--;
    }
  pop and output from s }
```

constant time

Time complexity : O(n+m)

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

		A	C	T	G
	←	–	–	–	–
A	|	↖	–	–	–
C	|	|	↖	←	–
G	|	|	|	↑	\
G	|	|	|	|	↖
A	|	\	|	|	↑

x	A	C	G	G	A
y	A	C	T	G	

h(5,4) = '|'
h(4,4) = '\' → G
h(3,3) = '|'
h(2,3) = '—'
h(2,2) = '\' → C
h(1,1) = '\'
end → A

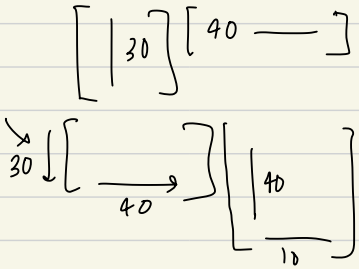
The sub sequence: A C G

Chain Matrix Multiplication

- The order problem

$A_1 \times A_2 \times A_3 \times A_4$

$30\times1 \quad 1\times40 \quad 40\times10 \quad 10\times25$



$((A_1A_2)A_3)A_4 = 30\times1\times40 + 30\times40\times10 + 30\times10\times25$

$= 20700$ multiplications

$A_1(A_2(A_3A_4)) = 40\times10\times25 + 1\times40\times25 + 30\times1\times25$

$= 11750$ multiplications

$A_1((A_2A_3)A_4) = 1\times40\times10 + 1\times10\times25 + 30\times1\times25$

$= 1400$ multiplications

↑ how do you choose optimal order to reduce multiplications?

problem: given matrices $A_1, A_2, \ldots A_n$ where dimensions of A_i are $d_{i-1} \times d_i$ (for $1 \le i < n$) in what order should the matrix multiplication be computed so as to incur minimum cost.

does it satisfy principle of optimality?

↓

Yes! because to find the most optimal solution, you need the most optimal solutions for all it's subproblems

eg. at level 1, i decide to split B & C

A B C D E

↓

(A B)(C D E)

↙ ↘

A B C D E

for this to work, the split determined for CDE also needs to be optimal.

Step 1. formulate into smaller sub-problems

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$ matrix

$d_0 \times d_1$ $d_1 \times d_2$ $d_2 \times d_3$ $d_3 \times d_4$ $d_4 \times d_5$ $d_5 \times d_6$ dimensions

① B_1 $d_0 \times d_3$

↓

$A_1 \times A_2 \times A_3$

② B_2 $d_3 \times d_6$

↓

$A_4 \times A_5 \times A_6$

or can be any where.

$\text{cost} = \text{cost}(①) + \text{cost}(②) + \text{cost}(B_1 \times B_2)$

Optimal cost $(A_1 A_2 A_3 A_4 A_5 A_6)$

$= \min ($ Optcost $(A_1 A_2 A_3 A_4 A_5)$ + Optcost (A_6) + $d_0 d_5 d_6$,
Optcost $(A_1 A_2 A_3 A_4)$ + Optcost $(A_5 A_6)$ + $d_0 d_4 d_6$,
Optcost :
:(A_3)
:(A_2)
Optcost (A_1) + Optcost $(A_2 A_3 A_4 A_5 A_6)$ + $d_0 d_1 d_6$)

base case : Optcost $(A) = 0$

Step 2 recursive function

↳ det input to Optcost

↓

needs to know matrix → start and end

need array of matrix to infer from dimensions:

0	1	2	3	4	5	6	7	8
d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8

Optcost (A_1, A_3)

Optcost $(0, 3)$

$d_0 \times d_1$, $d_1 \times d_2$, $d_2 \times d_3$ } ✓ indexes of d_i

calc starts from d_i to d_K

Optcost (i, k) ⟶ $A_{i+1} \times A_{i+2} \times \dots \times A_k$ first non inclusive

Optcost $(i, j) = 0$ if $k - i = 1$

$$\text{Optcost}(i,k) = \min_{i<j<k+1} \left(\text{Optcost}(i,j) + \text{Optcost}(j,k) + d_i \times d_j \times d_k \right)$$

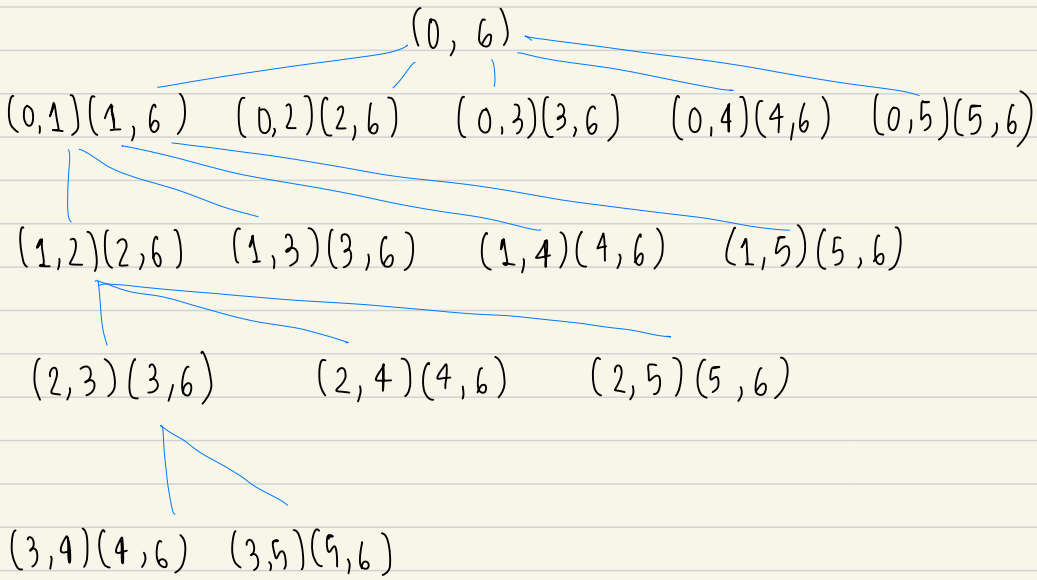
for matrix A_i ⟶ $d_{i-1} \times d_i$

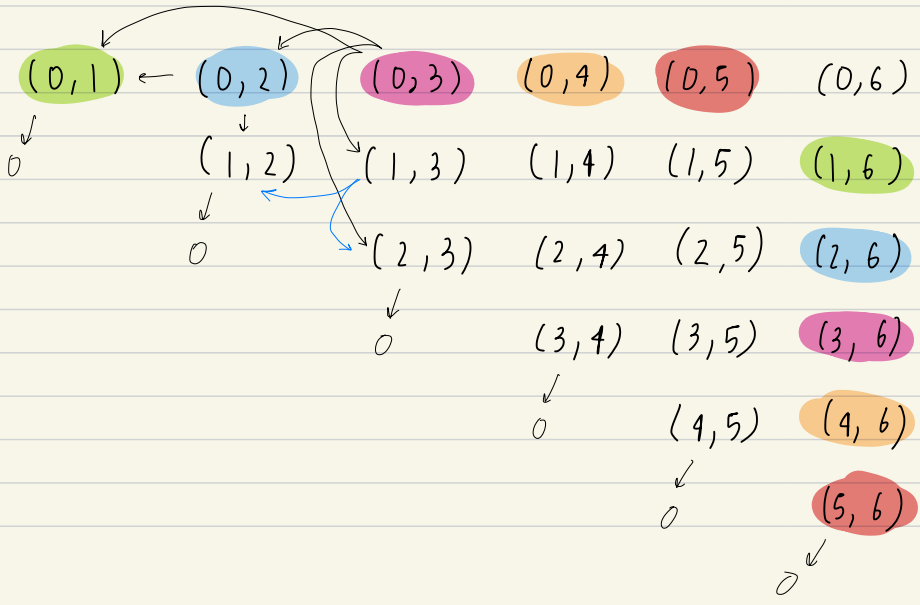
for all : Optcost $(0, n)$

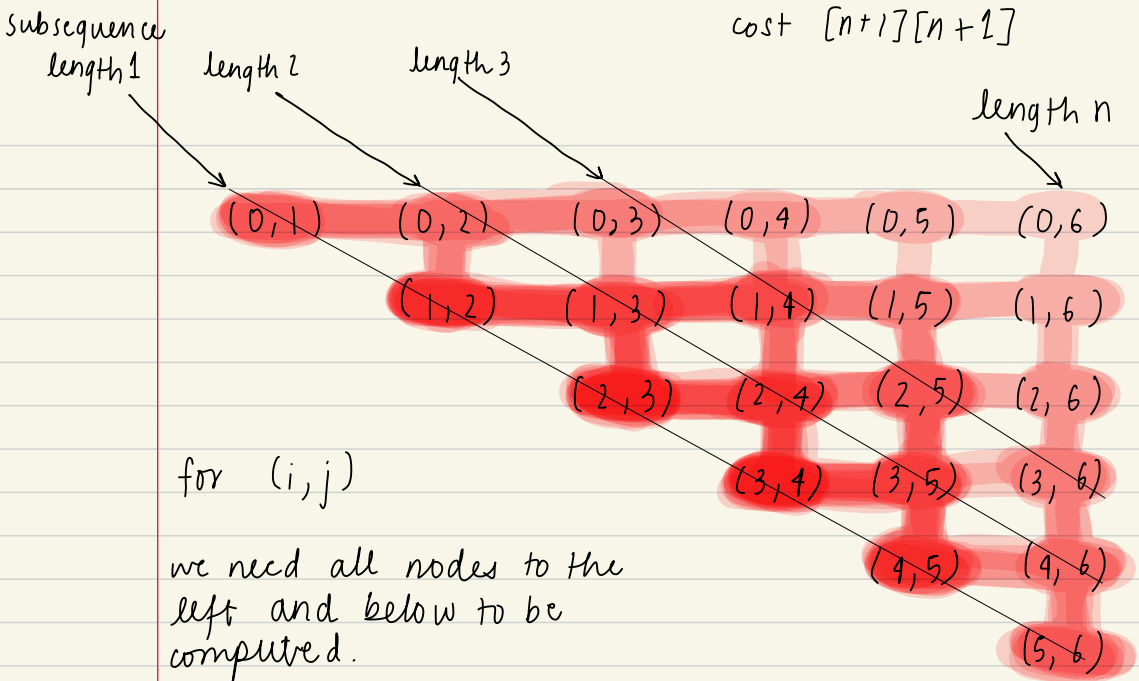
↓

n matrix

Step 3 Draw subproblem graph.







for (i, j)

we need all nodes to the left and below to be computed.

algo to fill the matrix

- we need to solve all subproblems:
- from length l : $1 \to n$
- for each length l, (i, j) ranging : $i : 0 \to n-l$, $j = i + l$

—x—

- approx $\frac{n^2}{2}$ subproblems $\rightsquigarrow$ $O(n^2)$ subprob calculation
- each subproblem (i,j) needs the computation of other immediate subproblems. $\rightsquigarrow O(n)$ (max: $2n$)

$\therefore$ total time comp : $O(n^3)$

Step 4 bottom up approach

```
int cost(ind[]d, int n) {

    for i = 0 to n-1
        cost[i][i+1] = 0;

for every l → for l = 2 to n {                          n
for every      for i = 0 to n-l {                       n
subproblem
of length l        j = i+l;
                   cost[i][j] = min( cost(subproblem 1) +
                                     cost(subproblem 2) +
                                     d[i]*d[ind1]*d[j] )

                   cost[i][j] = ∞
                   for k = i+1 to j-1 {                 n

                       c = cost[i][k] + cost[k][j] + d[i]*d[j]*d[k];
                       if ( c < cost[i][j]) {
                           cost[i][j] = c;

can have last[n+1][n+1] to store where we split
                           last[i][j] = k;
                       }
                   }
               }
           }
```

$O(n^3)$

d

30	1	40	10	25
0	1	2	3	4

cost

	0	1	2	3	4
0		0	1200	700	1400
1			0	400	650
2				0	10000
3					0
4					

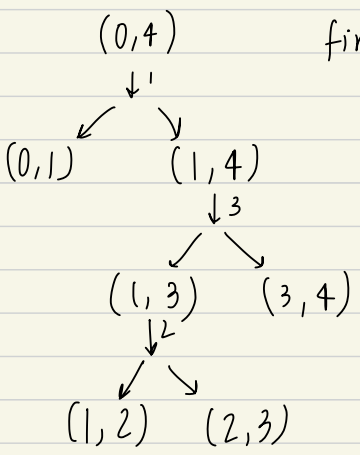
last

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

```
for l = 2 to n
    for i = 0 to n-l
        j = i + l;
        cost[i][j] = ∞;
        for k = i+1 to j-1
            c = cost[i][k] + cost[k][j]
                + d[i]*d[k]*d[j];
            if (c < cost[i][j])
                cost[i][j] = c;
                last[i][j] = k;
```

k=1:
Cost[0][1]+
Cost[1][4] + 750
k=2:
Cost[0][2]+
Cost[2][4] + 30000
k=3:
Cost[0][3]+
Cost[3][4] + 7500

l=4, i=0, j=4, k=1,2,3



final sum

0/1 Knapsack Problem

problem: we have a knapsack of capacity weight C (+ve int) and n objects with weights $w_1, w_2 \dots w_n$ and profits $p_1, p_2, \dots p_n$ (all w_i & all p_i are +ve). find the largest total profit of any subset of the objects that fit in the knapsack

- theres 2^n subsets of n objects, examining all takes $O(2^n)$ time

Step 1 formulate into subproblems

n objects w/ weights $w_1, w_2, \dots w_n$

n^{th} obj w/ w_n.

① chosen : cap available = $C - w_n$
prof = p_n + prof of rest
↳ subprob $(n-1)$

② not chosen : cap available = C
prof = prof of rem.
↳ sub prob $(n-1)$

Principle of optimality

↳ for optimal soln you choose max profit subproblem and for that subprob you need it's subproblem to have the optimal solution.

Step 2 recursive function definition

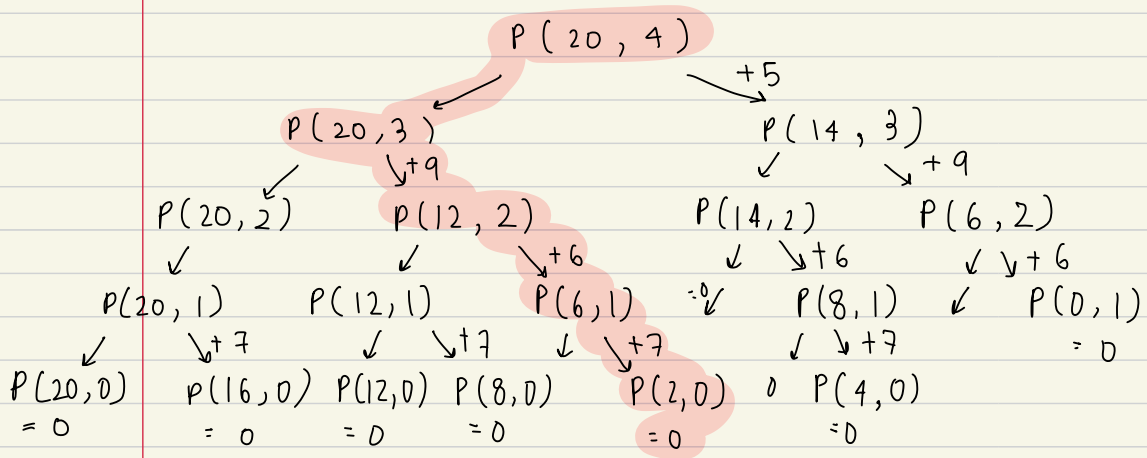
$$Profit(c,n) = \max(p[n] + Profit(c - w[n], n-1),\ Profit(c, n-1))$$

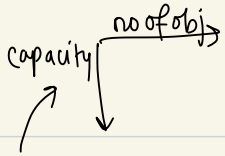
$$Profit(c,0) = Profit(0,j) = 0$$

Step 3 subproblem graph

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

C = 20.





Step 4 Dictionary : profit[C+1][n+1]

knapsack (int[] w, int[] p, int C, int n)

```
for i = 0 → n      profit[0][i] = 0;
for i = 0 → C      profit[i][0] = 0;

for c = 1 → C                                          C
    for obj = 1 → n                                    n
        profit[c][obj] = profit[c][obj-1];

        if ( w[obj] ≤ c )
            if ((profit[c - w[obj]][obj-1] + p[obj]) >
                profit[c][obj])
                profit[c][obj] = profit[c-w[obj]][obj-1]
                                 + p[obj];
```

constant

- time complexity : O(nC)

psuedo - polynomial

a polynomial fn of the value of input

size of input

DP review

- prob solving paradigm
- overlapping subproblem
- optimization problems
- expo → polynomial

- top down → problem-dictionary
- bottom up → dictionary

Principle of optimality :

- a problem satisfies it if the subsolutions of an optimal solution of the problems are themselves optimal solutions for their subproblems.

- shortest distance → yes
- longest path problem → no : 

a→d : a→b→c→d
a→b : a→c→d→c
making change

- 0/1 knapsack
- LCS → O(mn)
- chain multiplication →