overlapping subproblem

Introduction to DP

① Tabular Method → Bottom Upp.

② Memoization → Top to Bottom

Memoization:

fibonacci

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

convert

```
dp[n+1] = {-1}
f(n)
{
    if (n<=1)
        return;
    if (dp[n] != -1)
        return dp[n];
    else
        return dp[n] = f(n-1) + f(n-2);
}
```

T - O(N)

S - O(N) + O(N)

Tabular form

```
int n;
vector<int> dp(n+1, -1);
for (int i → n)
    dp[0] = 0
    dp[1] = 1
for (int i = 2; i <= n)
{   dp[i] = dp[i-1] + dp[i-2];
}
cout << dp[n] << endl;
```

T → O(N)
S → O(N)

Space optimized

```
int prev2 = 0
int prev = 1
for (i = 2; i <= n; i++)
    int curr = prev + prev2;
    prev2 = prev;
    prev = curr;
}
cout << prev << endl;
```

T → O(N)
S → O(1)

(3)

(Greedy won't work here, because values are not uniform)

0/1 Knapsack

Wt:	3	4	5	} Maximize
Val:	30	50	60	

→ Trying all combination, with recursion

Rules:

→ Express in every terms in index (ind, value)

→ Explore all possibilities. (pick, !pick)

→ Max of all possibilities

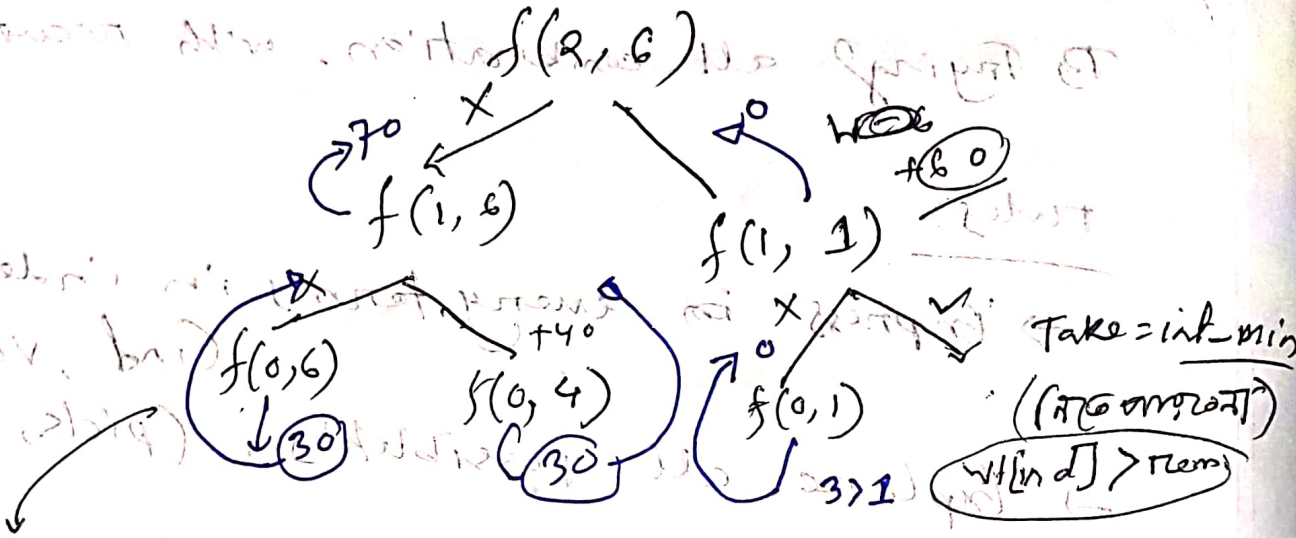
(4)

ind = 0 1 2

Wt → 3 2 5 } W = 6

Val → 30 40 60

f(ind, W)



Not Take = 0 + f(0,6) = 30

Take = 40 + f(0,4) = 40+30 = 70 } 70

for f(1,1)

Not take = 0 + f(0,1) = 0

Take = Int_min } 0

for f(2,6)

Not take = 0 + f(1,6) = 70

Take = 60 + f(1,1) = 60

70

(5)

Algo. (Recursive)

T → O(2^N)
S → O(N)

```
f(ind, W)
{
    if(ind == 0)
    {
        if(wt[0] <= W) return val[0];
        else return 0;
    }
    notTake = 0 + f(ind-1, W)
    Take = INT_MIN
    if(wt[ind] <= W)
        take = val[ind] + f(ind-1, W-wt[ind])

    return Max(notTake, Take);
}
```

(b)

T → O(N×W)
S → O(N×W) + O(N)

dp[N][W+1]

rec stackspace

Memoization

```
f(ind, W)
{   if (ind == 0)
    {   if (wt[0] <= W) return val[0];
        else return 0;
    }
    if (dp[ind][W] != -1)
        return dp[ind][W];

    NotTake = 0 + f(ind-1, W)

    Take = INT_MIN
    if (wt[ind] <= W)
    {   Take = val[ind] + f(ind-1, W-wt[ind])
        dp[ind][W] = Take;
    }
    return Max(Take, NotTake);
}
```

// driver

```
int knapsack(vi wt, vi val, int n, int W)
{   vector<vector<int>> dp(n, vector<int>(W+1, -1));
    return f(n-1, W, wt, val, dp);
}
```

Tabular Form

1. Base Case → all the weight will be considered
2. ind (1, n-1) wt (0, W)
3. copy the recurrence (bottom up)

ind dp [n] [W+1] = {0}

for (i = wt[0] → W) — W > wt[0]

dp [0] [i] = val [0]

(every w greater than wt[0], can be considered taken)

Tabular Form:

```
Knapsack (vi &wt, vi &val, int n,
                          int MaxW)

vector < vector <int>> dp (n, vector<int>
                          (MaxW+1, 0);

for(int w = wt[0]; w <= MaxW; w++)
    dp[0][w] = val[0];

for(int ind = 1; ind < n; ind++)
{   for(int w = 0, w <= MaxW; w++)
    {   int notTake = 0 + dp[ind-1][w];

        int Take = INT_MIN;

        if (wt[ind] <= w)
        {   Take = val[ind] + dp[ind-1]
                              [w - wt[ind]];
        }

    }   dp[ind][w] = max (Take, NotTake);
}
return dp[n-1][MaxW];

}
```

(9)

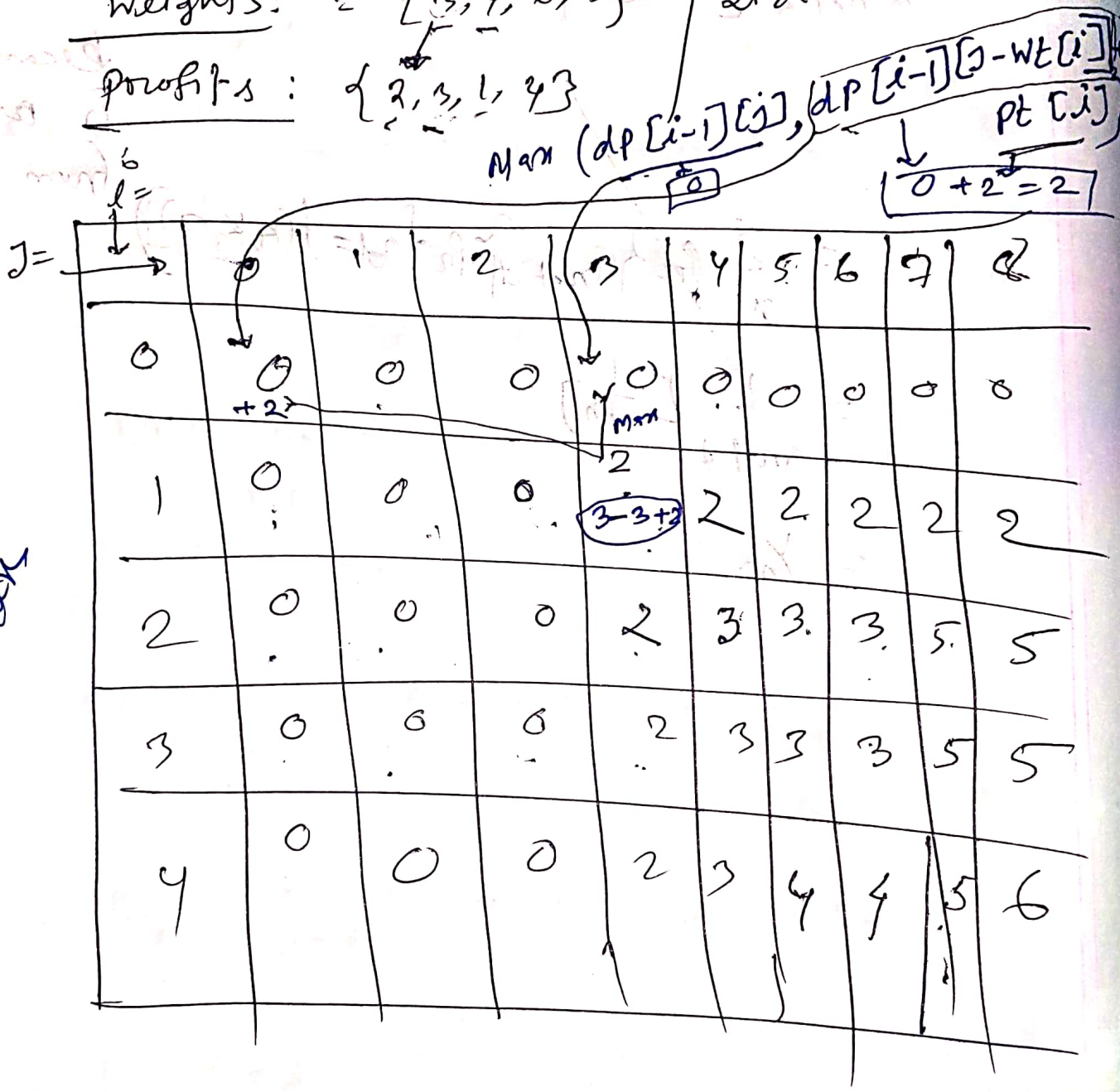
Tabulation for (space optimized)

```
knapsack (vi wt, vi val, int n, int MaxW)
{ vi prev (MaxW+1, 0), cur (MaxW+1, 0);

for (int w = wt[0]; w <= MaxW; w++)
        prev[w] = val[0];

for (int ind = 1; ind < n; ind++)
{
    for (int w = 0; w <= MaxW; w++)
    {
        int notTake = 0 + prev[w];
        int take = INT_MIN;
        if (wt[ind] <= w)
        {
            take = val[ind] + prev[w - wt[ind]];
        }
        cur[w] = max(take, notTake);
    }
    prev = cur;
}
return prev[MaxW];
}
```

Table:

Weights: = {3, 4, 6, 5} | Size = 8

Profits: {2, 3, 1, 4}



Longest Common Subsequence

Subsequence: Order maintain.

a b c d e

↳ ade, bce, acd ....

longest common subsequence:

S_1 = a b c d e f g h i j

S_2 = g c d g i l

} cdgi

Rules: (for recurrence)

1. Express in index
2. Explore all the possibility on the index
3. Take the best among them

indx → f(2) → String [0 .... 2]

4. do comparison characterwise

cases:

Case - 1: 2 characters are same:

$$lcs(\{1,2,3,4\}, \{1,2,3\})$$

$$= 1 + lcs(\{2,3,4\}, \{2,3\})$$

$$= 1 + lcs(\{3,4\}, \{3\})$$

General → $1 + lcs(indx1 - 1, indx2 - 1)$

Case - 2: 2 characters not same

lcs (YXA, AYZX)

lcs (YXA, AYZ) lcs (YXA, AYZX)

lcs (indx-1, indx2) lcs (indx1, ind2-1)

General:

$0 + lcs(Max($

$0 + Max(lcs(indx1-1, ind2),$

$lcs(indx1, indx2-1))$

code: (recursive)

lcs (X, Y, m, n) (string, index)

{
  if (m == n == 0) return 0;
  if X[m] == Y[n]
    return 1 + lcs (X, Y, m-1, n-1);
  return Max(Lcs(X, Y, m-1, n), Lcs(X, Y, m, n-1))
}

code (convert to DP).

{
  int Lcs (X, Y, m, n)
    if (n == 0 || m == 0) return 0;
    if (dp[m-1][n-1] != -1)
      return dp[m-1][n-1];
    if (x[m-1] == y[n-1])
      return dp[m-1][n-1] = 1 + Lcs(x, y, m-1, n-1, dp);
  return dp[m-1][n-1] = max(Lcs(x, y, m-1, n), Lcs(x, y, m, n-1))
}

Iterative: (Tabulation)

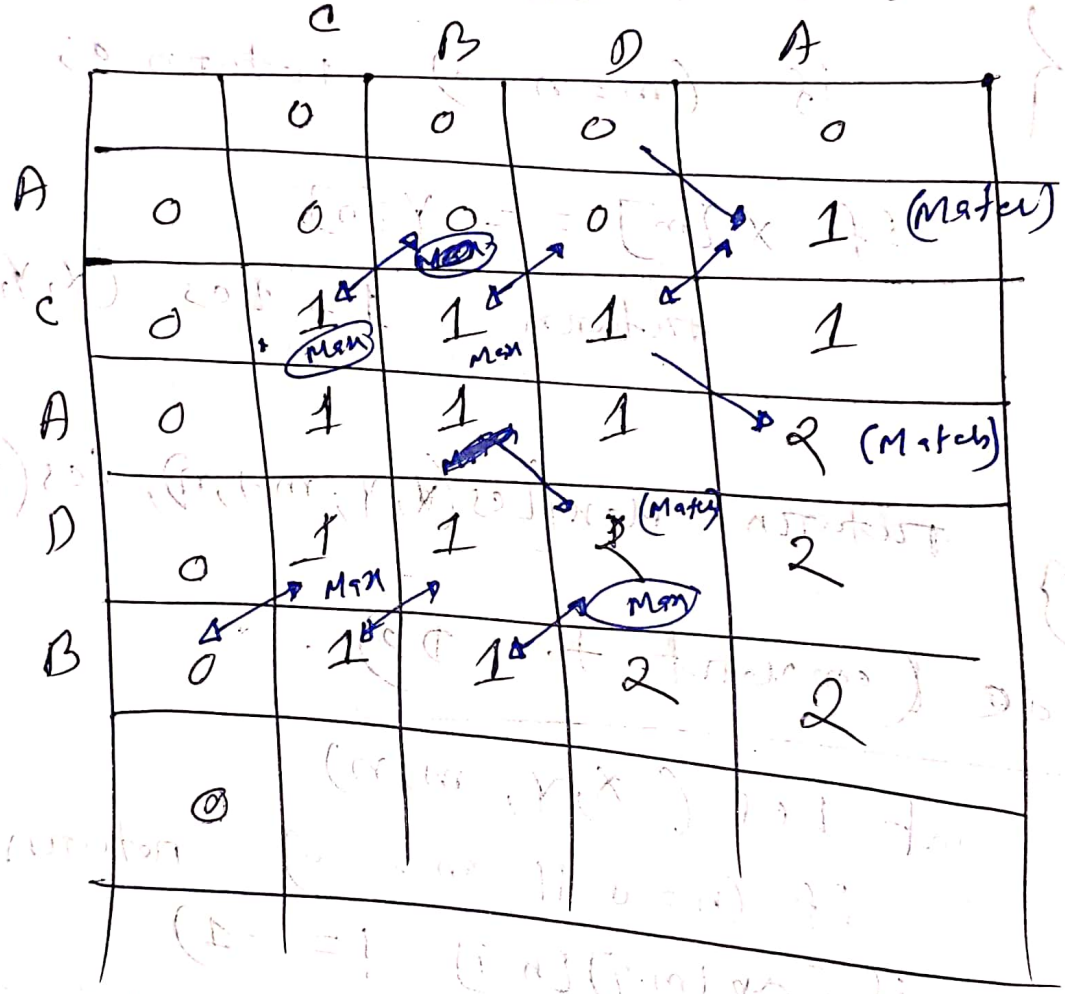
A C A D B

C B D A

if (match)

$S(i+1, j+1)+1$

else $\max(i+1, j), (i, j+1)$



code:

Technique:

1. copy the Base Case
2. i = 2, j = 1 (shift)
3. copy the recurrence

→ vector < vector <int>> dp (n+1, vector<int>(m+1, 0));

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
    {
        if (s[i-1] == s1[j-1])
            dp[i][j] = 1 + dp[i-1][j-1];
        else
            dp[i][j] = max(dp[i-1][j],
                           dp[i][j-1]);
    }
cout << dp[n][m]
```

Space optimization:

```
vector<int> prev(n+1, 0), cur(m+1, 0);

for (int i = 1; i <= n; i++)

    for (int j = 1; j <= m; j++)

    {   if (s[i-1] == s1[j-1])
            cur[j] = 1 + prev[j-1];

        else
            cur[j] = max(prev[j], cur[j-1]);

        prev = cur;
    }

cout << prev[m];
```

Knuth-Morris-Pratt (KMP)

prefix: a b c d

a, ab, abc, abcd..

suffix: a b c d

~~d c b a~~, d, cd, bcd. abcd.

printing LCS

```
{   int ln = dp[m][n];
    string ans = "";   Ans += "#" // dummy value
    int indx = ln-1;
    int i = n, j = m;
    while (i>0 && j>0)
    {   if (s[i-1] == s1[j-1])
        {   ans[index] = s[i-1];
            index--;
            i--;
            j--;
        }
        else if (dp[i-1][j] > dp[i][j-1])
            i--;
        else j--;
    }
```

String Matching Algorithm

Knuth-Morris-Pratt Algorithm (KMP)

```
vector<int> lpsf (string pattern)
{
    vector<int> lps(pattern.length());
    int index = 0;
    for (int i = 1; i < pattern.length(); )
    {
        if (pattern[indx] == pattern[i])
        {
            lps[i] = indx+1;
            ++index; ++i;
        }
        else if (index != 0)  indx = lps[indx-1]
        else { lps[i] = 0; i++; }
    }
    return lps;
}
```

```
void KMP (string text, string pattern)

vector<int> lps = lpsF (pattern);
i=0, j=0;
while (i < text.length())
{ if (text[i] == pattern[j])
  { i++; j++;
  }
  else if (j != 0) j = lps[j-1]
  else i++;
}
if (j == pattern.length())
  cout << "found" << endl;
}
```