DYNAMIC PROGRAMMING

PROBLEM OF THE DAY

There are given N ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. The task is to connect the ropes with minimum cost.

Example:

n = 4 ;arr[] = {4, 3, 2, 6} Output: 29

DYNAMIC PROGRAMMING

Generally used for solving optimization problems

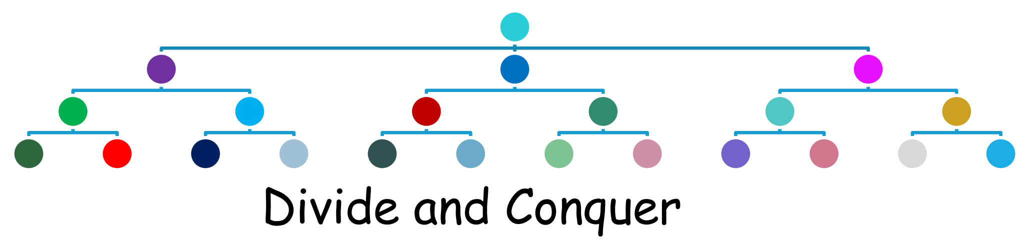
- Finding the largest, smallest, highest, etc.

The main characteristic of dynamic programming is that we solve several overlapping subproblems

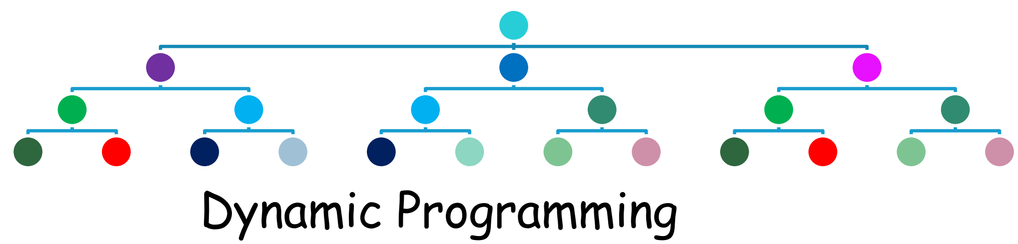
The results of these subproblems are combined to find solutions of larger subproblems, and so on, until we can solve the entire problem

DYNAMIC PROGRAMMING VS DIVIDE AND CONQUER

Both divide and conquer and dynamic programming use recursion



Divide and conquer also solves non-overlapping subproblems



Dynamic programming is used to solve overlapping subproblems

FIBONACCI SERIES

The Fibonacci series is such that the each number is the sum of the last two numbers occurring before it in the series

1,1,2,3,5,8,13,21,34,….

Problem: Given an integer n find the n^{th} Fibonacci number

- N=0 => 1
- N=1 => 1
- N=2 => 2
- N=3=> 3
- N=4 => 5
- N=5=> 8

NAÏVE METHOD

Fibonacci(n)

- If n=0 or n=1
  - Result=1
- Else
  - Result=Fibonacci(n-1)+Fibonacci(n-2)
- Return Result

- What is the complexity of this algorithm ?

NAÏVE METHOD

Time to solve Fibonacci (n) is $O(2^n)$

Complexity is $T(n)=T(n-1)+T(n-2)$; $T(0)=1$

- $T(n)=T(n-1)+T(n-2)$
- $< 2T(n-1)$
- $<2^2T(n-2)\ldots< 2^kT(n-k)$

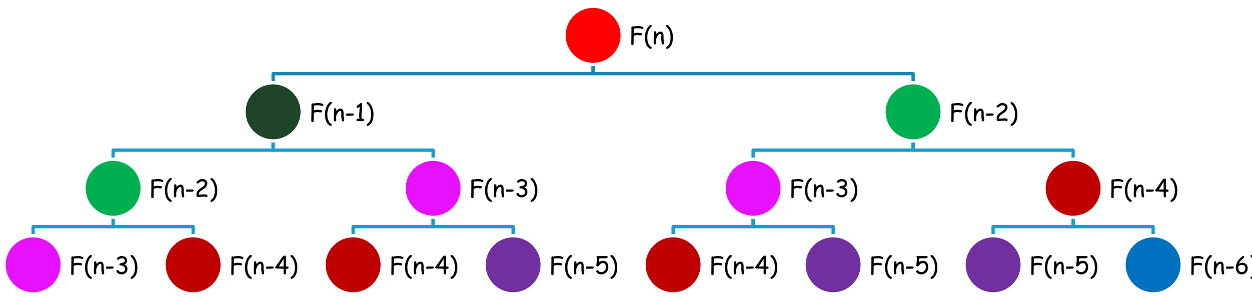
Recursion ends when $k=n$

$T(n))< 2^nT(n-n)=O(2^n)$

Can we do better ?

To do so, first understand why the complexity is high

FIBONACCI(N)



Note that the same Fibonacci value is used multiple times, and has to be recomputed

This increases the time complexity

Solution: store the results as they are computed

DYNAMIC PROGRAMMING METHOD

Fibonacci(n)

- F[0]=0;
- F[1]=1;
- For(i=2; i<=n;i++)
  - F[i]=F[i-1]+F[i-2]
- Return F[n]

- What is the complexity of this algorithm ?
  - One for loop of n-1 iterations
  - But also requires O(n) memory
  - How can you optimize the space ?

TO RECAP

Divide and Conquer: non-overlapping subproblems

Dynamic Programming: overlapping subproblems

The same problem can be written using divide and conquer as well as dynamic programming (the complexity differs)

MAXIMUM SUBSET SUM

Given an array of numbers find the maximum sum of contiguous numbers

-2	-3	4	-1	-2	1	5	-3

Maximum subarray sum: 7

MAXIMUM SUBSET SUM: DYNAMIC PROGRAMMING

-2	-3	4	-1	-2	1	5	-3

Local_max= 0
Global_max=0

Local_max= 0
Global_max=0

Local_max= 4
Global_max=4

Local_max= 3
Global_max=4

Local_max= 1
Global_max=4

Local_max= 2
Global_max=4

Local_max= 7
Global_max=7

Local_max= 4
Global_max=7

Complexity:
T(n)=O(n)
But need to store intermediate sums

Local max= Maximum Subset sum for the current sequence
Global max= Maximum subset sum across all sequences

MAXIMUM SUBSET SUM: DYNAMIC PROGRAMMING

Kadane's Algorithm

```
int maxSumSubArray(vector<int> &A)
{
    int n = A.size(); // Size of the array
    int local_max = 0;
    int global_max = INT_MIN; // -Infinity

    for (int i = 0; i < n; i++)
    {
        local_max = max(A[i], A[i] + local_max);
        if (local_max > global_max)
        {
            global_max = local_max;
        }
    }

    return global_max;
}
```

EXAMPLES

Given a set of non negative numbers, for example:

- {1,4,9,3, 7,6,5,2}

Determine whether divide and conquer or dynamic programming would be most efficient for the following problems

- Finding the maximum
- Finding if a given number, i.e. 3, is in the set of numbers
- Finding whether a subset of the numbers can add to a given number. For example, can a subset of the numbers add to 10 (yes 4+6; 9+1; 5+2+3)

CHARACTERISTICS OF DYNAMIC PROGRAMMING

Have to consider all possible subsets/options

Recall all possible subsets of n items is 2^n

However if we store the smaller subsets and the results, we can update for larger subsets

So we need

- A method to update the results as we increase size of subsets
- Memory to store previous results that need to be updated

SUBSET SUM PROBLEM

Given a set of positive, non repeating numbers

Given the sum which is a positive number

Is there a subset of numbers from the set that when added will produce the sum ?

Example:

- Set: {4,6,5,3,7} Sum:13
- Answers: Yes {4,6,3}

NAÏVE METHOD

Check all subsets, and see if they add to the given sum

- {4}, {6}, {5}, {3}, {7}
- {4,6}, {4,5}, {4,3}, {4,7}, {6,5}, {6,3}, {6,7}, {5,3}, {5,7}, {3,7}
- {4,6,5}, {4,6,3},{4,6,7}, {6,5,3}, {6,5,7}, {5,3,7}
- {4,6,5,3},{4,6,5,7},{6,5,3,7}
- {4,6,5,3,7}

What is the complexity ?

- Time to add = $1*{}^nC_1 + 2*{}^nC_2 + 3*{}^nC_3 + \ldots + n*{}^nC_n = n*2^{(n-1)}$

IDENTIFYING OVERLAPPING SUBPROBLEMS

Lets check the overlapping subproblems in the list

- {4}, {6}, {5}, {3}, {7}
- {4,6}, {4,5}, {4,3}, {4,7}, {6,5}, {6,3}, {6,7}, {5,3}, {5,7}, {3,7}
- {4,6,5}, {4,6,3},{4,6,7}, {6,5,3}, {6,5,7}, {5,3,7}
- {4,6,5,3} ,{4,6,5,7}, {6,5,3,7}
- {4,6,5,3,7}

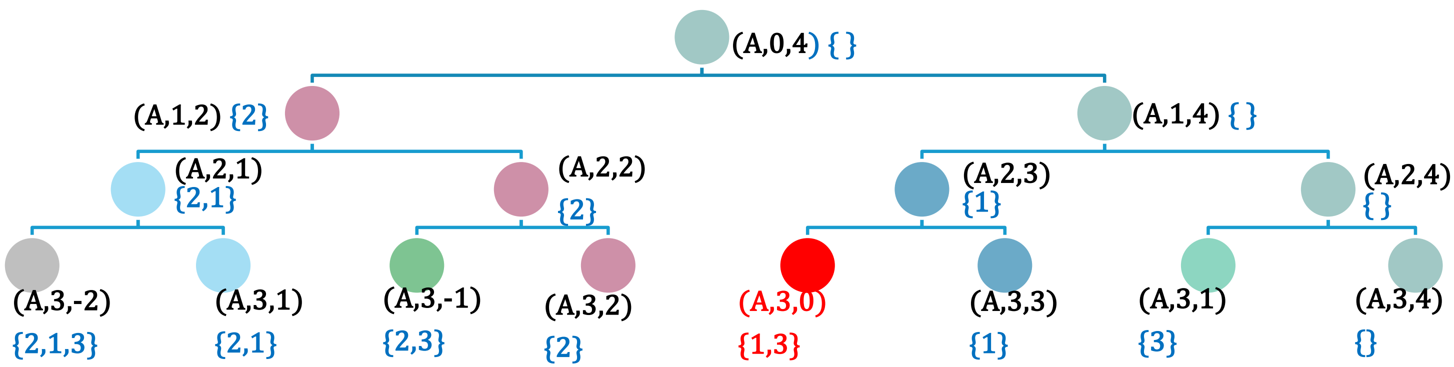
RECURSION

SubsetSum(Array,Start_Index,Sum)

- If sum=0;
  - then return true; (since ϕ is a subset)
- Else
  - (i) We include the current element in the sum
    - Sum=Sum-Array[Start_Index];
    - S1=SubsetSum(Array,Start_Index-1,Sum)
  - (ii) We do not include the current element in the sum
    - S2=SubsetSum(Array,Start_Index-1,Sum)
  - Return S1||S2

Set={2,1,3}
Sum=4

Complexity
T(n)=2T(n-1)



SUBSET SUM: DYNAMIC PROGRAMMING

We will need a two dimensional array (called subset), to keep track of the intermediate sum and the index

Complexity=O(Sum*n)

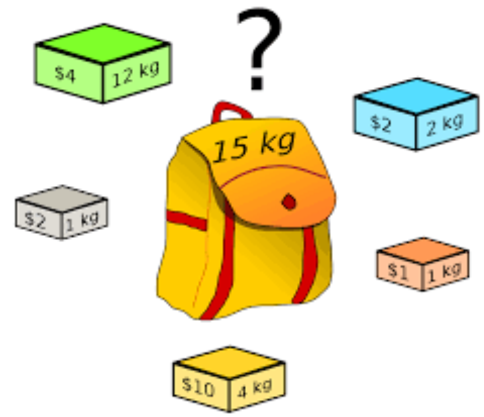
Set={2,1,3}
Sum=4

```
// Returns true if there is a subset of set[] with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{  // Base Cases
    if (sum == 0) return true;
    if (n == 0)  return false;
    // If last element is greater than sum,  then ignore it
    if (set[n - 1] > sum)
        return isSubsetSum(set, n - 1, sum);
    /* else, check if sum can be obtained by any of the following:
      (a) including the last element
      (b) excluding the last element   */
    return isSubsetSum(set, n - 1, sum)
           || isSubsetSum(set, n - 1, sum - set[n - 1]);
}
```

Code from:https://www.geeksforgeeks.org/subset-sum-problem-dp-25/

	0	1	2	3	4
{2}					
{2,1}					
{2,1,3}					

0-1 KNAPSACK PROBLEM



Given a set of n items ;

Items cannot be subdivided

Value of ith item is vi and its weight is wi

Your Knapsack can hold at most W weight

Which items should you pick to maximize the value

0-1 KNAPSACK PROBLEM

Notice the similarity with the subset sum problem

Naïve method can be checking every subset---which is expensive

Recursive method is for each item,

- either select the ith item, and subtract the weight
- Or do not select the ith item and maintain the weight

Initial Settings: Set

$$\begin{array}{lll} V[0,w] = 0 & \text{for } 0 \le w \le W, & \text{no item} \\ V[i,w] = -\infty & \text{for } w < 0, & \text{illegal} \end{array}$$

Recursive Step: Use

Don't include item i and maintain weight

Include item i and subtract its weight

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

for $1 \le i \le n$, $0 \le w \le W$.

Let V[i,w] store the maximum value of any set of items from 1 to I, with weight W

Let $W = 10$ and

Complexity O(nW)

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
$i = 0$											
1											
2											
3											
4											

PSEUDO-POLYNOMIAL COMPLEXITY

Subset sum and 0-1knapsack are examples of problems with pseudopolynomial complexity

A numeric algorithm runs in **pseudo-polynomial time** if its running time is a polynomial in the *numeric value* of the input (the largest integer present in the input) — but not in the *length of the input* (the number of bits representing it).

The subset sum and 0-1 knapsack have complexity O(nW);

- This means order of W steps
- But representing W only takes log(W)
- So number of steps exponential to length of the input

ALL PAIR SHORTEST PATH

APSP finds the shortest path between all pairs of vertices

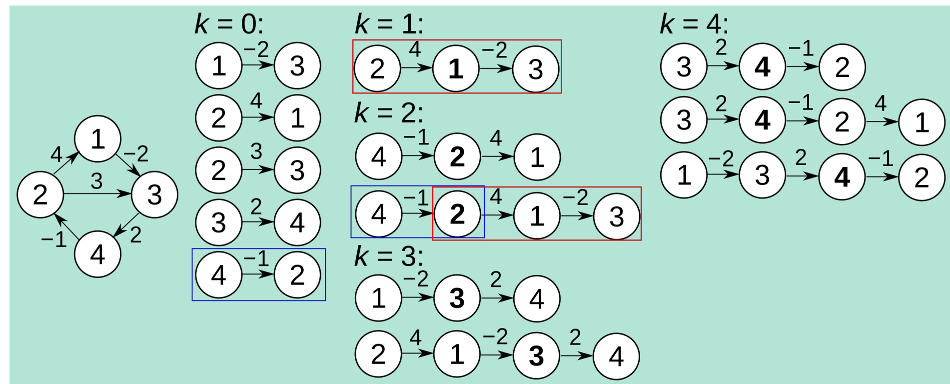
Uses dynamic programming

Can handle negative weights, but not negative cycles

Floyd Warshall Algorithm (1959, Roy; 1962 Floyd, Warshall)

- Initialize all entries in |V|x|V| matrix d to INF
- For each edge(u,v)
  - d[u][v]=w(u,v)
- For each vertex v
  - d[v][v]=0
- For k=1 to|V|
  - For j=1 to|V|
    - For i=1to |V|
      - If d[i][j]>d[i][k]+d[k][j]
      - d[i][j]=d[i][k]+d[k][j]

Complexity O(|V|^3)

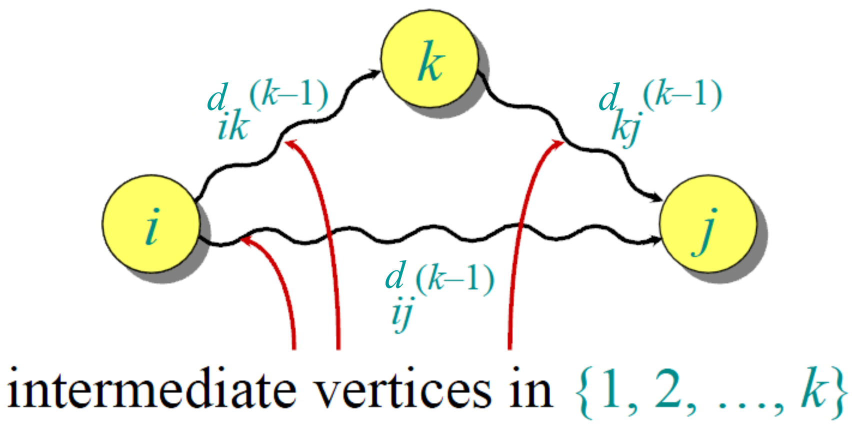


By Dcoetzee - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=23230210

RECURSIVE STRUCTURE IN THE ALGORITHM

$d_{ij}(k)$ = weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$.

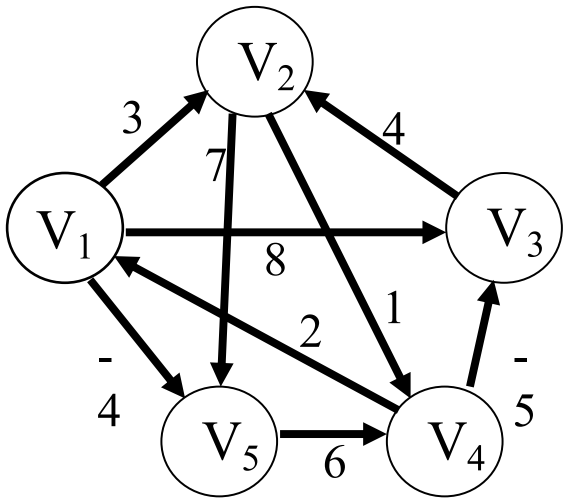
$$d_{ij}^{(k)} = \min_k \{ d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \}$$



How to find All Pairs Shortest Paths?

Compute the $d_{ij}(k)$ values in order of increasing values of k.

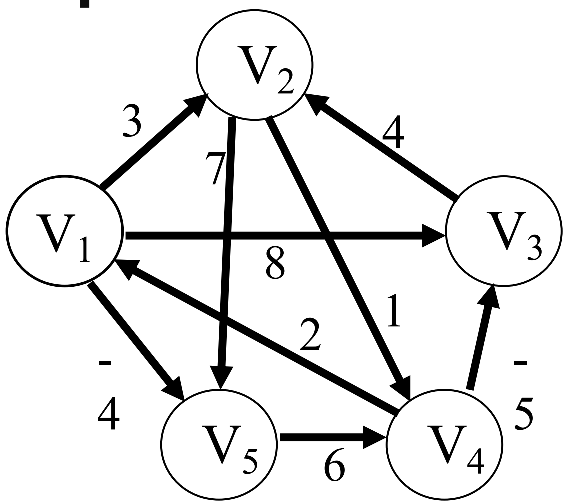
EXAMPLE



	V_1	V_2	V_3	V_4	V_5
V_1	0	3	8	∞	-4
V_2	∞	0	∞	1	7
V_3	∞	4	0	∞	∞
V_4	2	∞	-5	0	∞
V_5	∞	∞	∞	6	0

	V_1	V_2	V_3	V_4	V_5
V_1	-	1	1	-	1
V_2	-	-	-	2	2
V_3	-	3	-	-	-
V_4	4	-	4	-	-
V_5	-	-	-	5	-

K=1



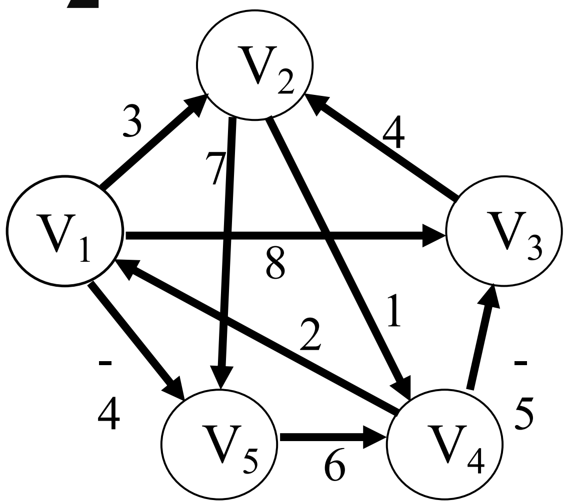
$D_{i,j}^{(1)}=\min(D_{i,j}^{(0)}, D_{i,1}^{(0)}+D_{1,j}^{(0)})$

Allowed intermediate vertices: subset of {V1}

	V_1	V_2	V_3	V_4	V_5
V_1	0	3	8	∞	-4
V_2	∞	0	∞	1	7
V_3	∞	4	0	∞	∞
V_4	2	5	-5	0	-2
V_5	∞	∞	∞	6	0

	V_1	V_2	V_3	V_4	V_5
V_1	-	1	1	-	1
V_2	-	-	-	2	2
V_3	-	3	-	-	-
V_4	4	1	4	-	1
V_5	-	-	-	5	-

K=2



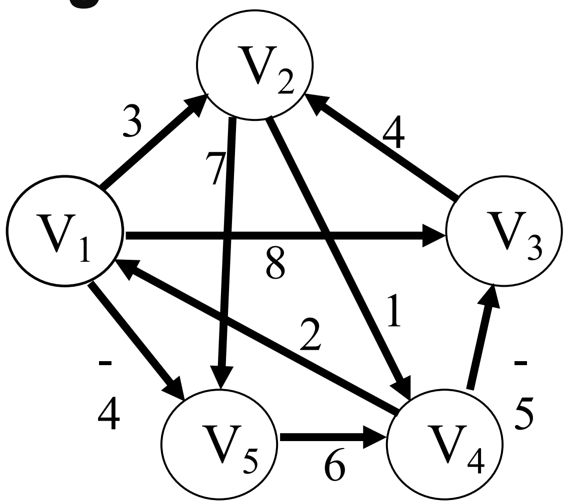
$D_{i,j}^{(2)}=\min(D_{i,j}^{(1)}, D_{i,2}^{(1)}+D_{2,j}^{(1)})$

Allowed intermediate vertices: subset of {V1,V2}

	V_1	V_2	V_3	V_4	V_5
V_1	0	3	8	4	-4
V_2	∞	0	∞	1	7
V_3	∞	4	0	5	11
V_4	2	5	-5	0	-2
V_5	∞	∞	∞	6	0

	V_1	V_2	V_3	V_4	V_5
V_1	-	1	1	2	1
V_2	-	-	-	2	2
V_3	-	3	-	2	2
V_4	4	1	4	-	1
V_5	-	-	-	5	-

K=3



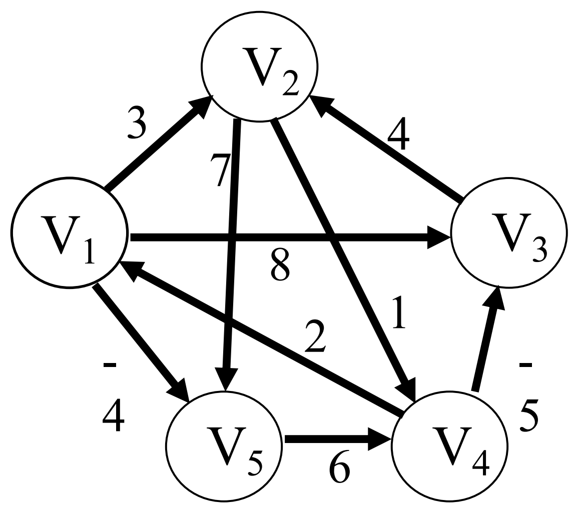
	V_1	V_2	V_3	V_4	V_5
V_1	0	3	8	4	-4
V_2	∞	0	∞	1	7
V_3	∞	4	0	5	11
V_4	2	-1	-5	0	-2
V_5	∞	∞	∞	6	0

	V_1	V_2	V_3	V_4	V_5
V_1	-	1	1	2	1
V_2	-	-	-	2	2
V_3	-	3	-	2	2
V_4	4	3	4	-	1
V_5	-	-	-	5	-

$D_{i,j}^{(3)}=\min(D_{i,j}^{(2)}, D_{i,3}^{(2)}+D_{3,j}^{(2)})$

Allowed intermediate vertices: subset of {V1,V2,V3}

K=4



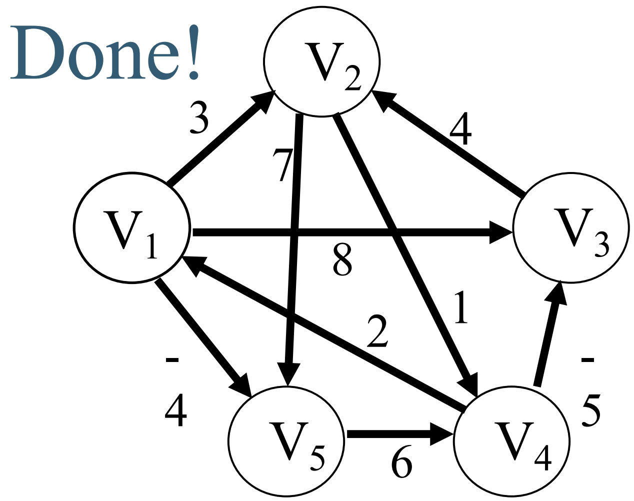
$D_{i,j}^{(4)}=\min(D_{i,j}^{(3)}, D_{i,4}^{(3)}+D_{4,j}^{(3)})$

Allowed intermediate vertices: subset of {V1,V2,V3,V4}

	V_1	V_2	V_3	V_4	V_5
V_1	0	3	-1	4	-4
V_2	3	0	-4	1	-1
V_3	7	4	0	5	3
V_4	2	-1	-5	0	-2
V_5	8	5	1	6	0

	V_1	V_2	V_3	V_4	V_5
V_1	-	1	4	2	1
V_2	4	-	4	2	4
V_3	4	3	-	2	4
V_4	4	3	4	-	1
V_5	4	3	4	5	-

K=5



$D_{i,j}^{(5)}=\min(D_{i,j}^{(4)}, D_{i,5}^{(4)}+D_{5,j}^{(4)})$

Allowed intermediate vertices: subset of {V1,V2,V3,V4,V5}

Trace back the shortest path between v3,v5: V3→v2→v4→v1→v5

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	-3	2	-4
V_2	3	0	-4	1	-1
V_3	7	4	0	5	3
V_4	2	-1	-5	0	-2
V_5	8	5	1	6	0

	V_1	V_2	V_3	V_4	V_5
V_1	-	5	5	5	1
V_2	4	-	4	2	4
V_3	4	3	-	2	4
V_4	4	3	4	-	1
V_5	4	3	4	5	-

SUBSEQUENCE OF A STRING

Given a sequence X=<x1,x2, …xm>, the sequence Z= <z1,z2,…zk> is a subsequence of X if there is a strictly increasing sequence <i1, i2,…ik> such that for all j=1,2,…k $x_{i_j}=z_j$

X=<A,B,C,B, D,A,B> then a subsequence is

Z=<B,C, D,B> and the sequence index is <2,3,5,7>

Y=<B,D,C> is not because B,D,C do not appear in that order in X

LONGEST COMMON SUBSEQUENCE

Given two sequences X and Y, Z is a common subsequence of X and Y if Z is a subsequence of both X and Y

X=< A,B,C,B, D, A, B>

Y=<B,D,C,A,B,A>

Common Subsequence <B,C,B>

Longest common subsequence (LCS) <B,C, B,A>

Goal : To find one of the longest common subsequences of a pair of sequences

This is used in comparing between two genetic/protein sequences.

RECURSION

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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}.

Start from the last letters

If the last two letters match then LCS length is increased by 1

LCS of X (-last entry), and Y(-last entry)

Else. Max of

LCS of X, and Y(-last entry)

LCS of X(-last entry) and Y

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

ALGORITHM FOR COMPUTING LCS

C[i,j]= length of LCS of X(0,i) and Y(0,j)

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 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 x_i = y_j
                      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
```

If last two entries match, increase LCS by 1

LCS of X(-last entry),Y

LCS of X,Y(-last entry)

For loop on m and For loop on n=> O(nm)

	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						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Strings being compared:
X[0,i] and Y[0,j]

At entry 2,3
We are comparing
X={A,B}
Y={B,D,C}

BACK TO FIBONACCI

Fibonacci(n)

Set F[1:n]=0;

- If n=0 or n=1
  - Result=1
- If F[n]>0
  - Result=F[n]
- Else
  - Result=Fibonacci(n-1)+Fibonacci(n-2)
  - F[n]=Result
- Return Result

Memoization:
Bottom up

Fibonacci(n)

- F[0]=0;
- F[1]=1;
- For(i=2; i<=n;i++)
  - F[i]=F[i-1]+F[i-2]
- Return F[n]

Tabulation:
Top Down

STORING RESULTS

Memoization

Only computes the values that are needed

Goes through the recursion tree and stores results as they occur

Tabulation

Computes all values

Iteratively fills all the values

EXAMPLES

Given below are returns on investments for the amounts given in the top row. For example, investing $500 in INV2 gives back return of $30. What is optimal investment for $600 for get back the maximum return

	100	200	300	400	500	600
INV 1	5	11	16	23	29	35
INV 2	4	12	18	23	30	34
INV 3	4	5	5	30	30	35
INV 4	6	12	17	24	30	31

EXAMPLES

Alex can work upto 6hrs He has a choice of jobs to choose from in the Table given below. The start and end time of the jobs are flexible, but the duration is fixed. If he starts a job he has to finish it and cannot leave it in-between. He can work on multiple jobs one after another, but can work on only one job at a time (no multitasking). He cannot repeat a job. Give an algorithm to select which jobs should he do to get the most payment within the 6 hrs limit, and obtain the best value of money for time.

	Duration in hours	Salary per hrs
J1	5	5.4
J2	4	5
J3	3	6
J4	2	4

SUMMARY

Dynamic programming solves optimization problems

The optimization of subproblems can be combined to optimize the larger problem

Unlike divide and conquer, we have to check all the possible partitions to find the optimal solution