Dynamic Programming Strategy

Dynamic Programming

- Dynamic Programming is mainly an improvement over plain recursion.
- Recursion breaks down a complex problem into simpler subproblems, but ends up repeatedly solving the same problems again and again.
- For example when we want to obtain Fibonacci(10), we simplify it
- Fib(10) = Fib(9) + Fib(8)
- = **Fib(8)** + Fib(7) + Fib(8)
- so this involves solving for Fib(8) two times separately.
- This happens at each step, so we end up doing lots of computations.

- Dynamic programming solves a complex problem by first breaking into a collection of simpler subproblems,

- solving each subproblem *just once*,
- and then storing their solutions to avoid repetitive computations.

- It somewhat resembles divide and conquer strategy where we know how to divide a problem.
- But often in D.P. the best way to divide a problem is not known beforehand.
- DP divides the problems in many ways Then it solves all sub-problems.
- Solves the simplest sub-problem first, Then it solves all sub-problems.
- and works its way up to the original problem.
- solutions are stored in a table.
- When a solution is needed, it is not re-computed , but simply taken from the table.

Fibonacci Numbers using recursion

- $f_n = f_{n-1} + f_{n-2}$
- The recurrence relation for Fibonacci sequence is
- $T(n) = T(n-1) + T(n-2) + 1$
- Let us simplify it $T(n) = T(n-1) + T(n-1) + 1$
- $= 2^* T(n-1) + 1$
- Note $T(n-1) = 2^* T(n-2) + 1$
- Substituting
- $T(n) = 4^* T(n-2) + 3$
- $T(n) = 8^* T(n-3) + 7$
- $T(n) = 16^* T(n-4) + 15$

Fibonacci Numbers using recursion

- $T(n) = 16*T(n\text{-}4) + 15$ $= 2^4 * T(n-4) + (2^4\text{-}1)$
- General case
- $T(n) = 2^k * T(n-k) + (2^k\text{-}1)$
- Boundary condition $T(0) = 1$.
- Put $(n - k) = 0$, $n = k$
- Substituting
- $T(\mathrm{n}) = 2^n * T(0) + (2^n - 1)$ $= 2^n + 2^n - 1$ $= O(2^n)$

- **$T(\mathrm{n}) = O(2^n)$**

Fibonacci Numbers using recursion

- ***T*(n) = O(2^n)**

- How large is it? let us say for n = 20
- $T(n) = 2^{20}$ = nearly 1 million.

- Most recursive solutions have very high time complexities.

Fibonacci Numbers using DP

- $f_n = f_{n-1} + f_{n-2}$
- DP method breaks down the problem into subproblems
- Starts from smaller solutions and builds up final solution based on these.

- Recursion Method starts from n and goes down to 3.

- DP method starts from lowest possible value 3, and builds up all solutions for 4,5,6,7, ... , n
- stores the solution to various subproblems in an array.

Fibonacci algo. using DP

- fib1 (n) {

  f[1] = 1

  f[2] = 2

  for i = 3 to n

  f [i] = f [i – 1] + f [i – 2]

  return f[n] }
- It stores all values 3 to n in the array
- Because of the for loop, the algorithm runs in $\Theta(n)$.

Dyn Prog Solution

- [f(3)   f(4)   f(5)   f(6)   f(7)   f(8)   f(9)   f(10)   f(11) .. . . . .]

- **Complexity** of Fib. Seq. generation using D.P. is $\Theta(n)$.

- Dynamic Programming should be considered

  if a problem can be divided into subproblems *that overlap*

- so that info from one subproblem can be used for another subproblem

Memoization

- In programming, **memoization is an optimization technique** that makes applications more efficient and hence faster.

- stores computation results in cache, and retrieves that information from the cache next time it's needed instead of computing it again.

- In simpler words, it consists of storing in **cache** the output of a function. Function checks if each computation is already in the cache before computing it.

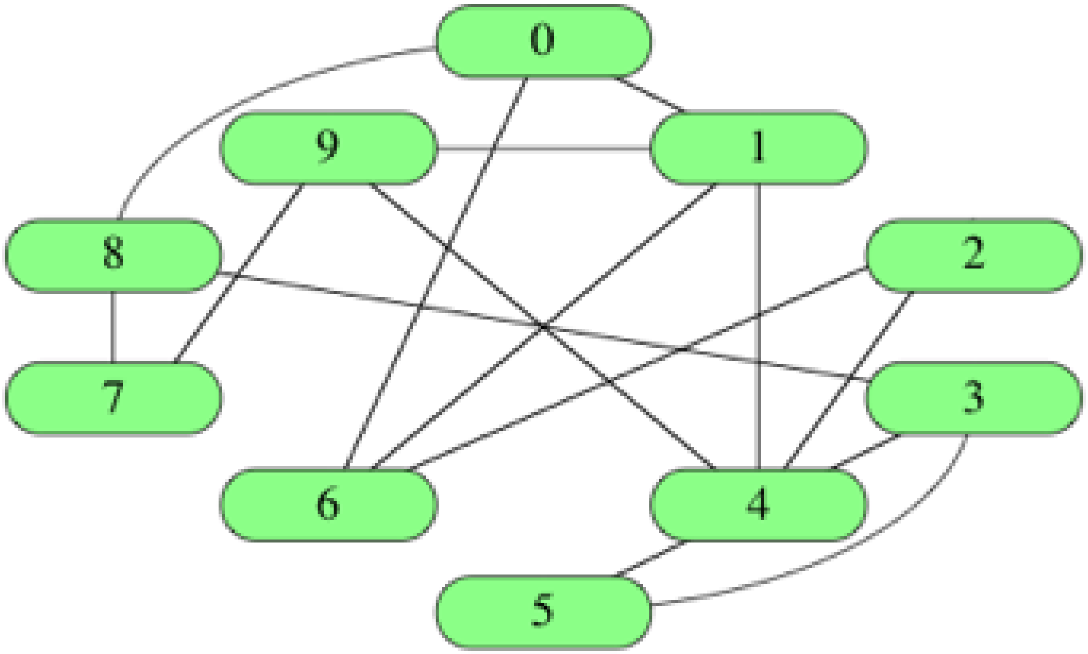
- Memoization is a simple but powerful trick

Memoization in D.P.

- Memoization is storing in memory
- It is a common strategy for dynamic programming problems,
- **where the solution is composed of solutions to the same problem with smaller inputs**
- Like in Fibonacci sequence, f(10) depends on solutions to f(9) and f(8)

Principle of Optimality

- It is core principle of D.P.
- An optimal sequence is feasible if and only if its sub-sequences are optimal.



- The path 9 – 2 – 6 will be optimal only if path 9 – 2 is optimal and path 2 – 6 is also optimal.

Problems which can be divided into sub-problems

- 0/1 Knapsack Problem
- Travelling Salesman problem

- Largest Common Subsequence problem
- Handling chain of matrices

- All pairs shortest path in graphs
- Single source shortest path in graphs (extension of Dijkastra’s algo)
- Optimal Binary search trees

<u>Prob.1.</u>

Longest Common Subsequence problem

Longest Common Subsequence problem

- A subsequence is a sequence that appears in the same relative order but is not necessarily contiguous.

- Consider sequence “abcdefg”. Here are some subsequences
- “abc”,
- “abg”,
- “bdf”,
- “aeg”, ‘
- ”acefg”, .. etc.

Longest Common Subsequence problem

- Consider a sequence “abtdiekf”
- and another sequence “abcdefg”.

- Note subsequence “abde” is common to both the sequences

LCS problem in protein matching

- One Application of LCS:
- To figure out if two proteins are similar.

- Proteins are linear chains of amino acids (around 20 more frequent)

- To understand an unknown protein is to compare its amino acid sequence with proteins whose functions are known.

- KVLWKTIGETLTWSRIITGGAMHDQVMITG
- GMILLETNPGWYNSKRNMDRCSWTINTDMD
- HKQVDHTNHKLWCIEPGFFGVHSMQANYFV
- MSLGWTVTLPVGNHHGTWHKITQCNQGNSQ
- FLRGISTEITACTYKPCDQAMRNVAQLAGA

- Two proteins are “similar” if they have a long common subsequence.

- we need to first know the number of subsequences with lengths ranging from 1,2,..n-1.
- number of combinations with 1 element is ${}^{n}C_{1}$.
- A number of combinations with 2 elements are ${}^{n}C_{2}$ and so on.
- We know that ${}^{n}C_{0} + {}^{n}C_{1} + {}^{n}C_{2} + \ldots {}^{n}C_{n} = 2^{n}$.
- So a string of length n has $2^{n} – 1$ different possible subsequences .
- This implies that the time complexity of the brute force approach will be $O(n * 2^{n})$.
- Note that it takes O(n) time to check if a subsequence is common to both strings. This time complexity can be improved using dynamic programming.

- The D.P. approach uses *array to store solutions to smaller subproblems* and build up solutions from there.
- Create a 2D array "c" with rows and columns equal to the length of each input string plus 1.
- c[i, j] is used to store length of LCS of a[1],. . . ,a[i] and b[1], ... , b[j]
- First column and first row are set to 0.
- If a[i] = b[j], we have found a common member of LCS, so we increment previous value

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

- Otherwise, we retain the max of earlier comparisons
- $c[i, j] = \max\{ c[i-1, j], c[i, j-1] \}$

- For every c[i,j] we examine
- the element at its top and
- the element at its left

- Find the common subsequence of

- G V C E K S T
- and
- G D V E G T A

- Find the common subsequence of

- G V C E K S T
- and
- G D V E G T A

- Common subsequence is G V E T
- Let us extract the subsequence using Dynamic Programming
- Form a 2D array with one sequence row-wise and other column wise.

- Let us name the strings

- b[i] = [G V C E K S T]
- and
- a[j] = [G D V E G T A]

Values of c[i, j]

- If a[i] = b[j], same c[i, j] = c[i – 1 , j – 1] + 1
- If not same , c[i, j] = max { c[i – 1 , j] , c[i , j – 1] }

			G	V	C	E	K	S	T
		0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
G	1	0	1						

- a[1] = G, b[1] = G same
- so c[1 ,1] = c[1-1, 1-1] + 1 = 1 *increment diagonal value*

Values of c[i, j]

- If a[i] = b[j], same c[i, j] = c[i – 1 , j – 1] + 1
- If not same , c[i, j] = max { c[i – 1 , j] , c[i , j – 1] }

			G	V	C	E	K	S	T
		0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
G	1	0	1	1					

- for c[1,2], a[1]= G and b[1]= V not same
- c[1,2] = max{ c[0,2], c[1,1] } = 1 *take max of top and left values*

Values of c[i, j]

- If a[i] = b[j], same $\quad$ c[i, j] = c[i – 1 , j – 1] + 1
- If not same , $\quad$ c[i, j] = max { c[i – 1 , j] , c[i , j – 1] }

			G	V	C	E	K	S	T
		0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
G	1	0	1	1	1	1	1	1	1
D	2	0	1	1	1	1	1	1	1

- for c[2,1], a[2]= D and b[1]= G $\quad$ not same
- c[2,1] = max{ c[1,1], c[2,0] } = 1 $\quad$ *take max of top and left values*

			G	V	C	E	K	S	T
		0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
G	1	0	1	1	1	1	1	1	1
D	2	0	1	1	1	1	1	1	1
V	3	0	1	2	2	2	2	2	2
E	4	0	1	2	2	3	3	3	3
G	5	0	1	2	2	3	3	3	3
T	6	0	1	2	2	3	3	3	4
A	7	0	1	2	2	3	3	3	4

- length of LCS is lowermost right corner value = 4

The name of the subsequence

			G	V	C	E	K	S	T
		0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0
G	1	0	**1**	1	1	1	1	1	1
D	2	0	1	1	1	1	1	1	1
V	3	0	1	**2**	2	2	2	2	2
E	4	0	1	2	2	**3**	3	3	3
G	5	0	1	2	2	3	3	3	3
T	6	0	1	2	2	3	3	3	4
A	7	0	1	2	2	3	3	3	**4**

Time Complexity of LCS

- Common subsequence: whenver C[i,j] gets incremented

- If m is length of one sequence, and n is length of the other sequence
- complexity : **O(mn)**

<u>prob 2:</u>
0/1 Knapsack Problem

- We have already seen Greedy strategy can provide Optimal solution to continuous knapsack problem.
- But not for 0/1 Knapsack problem, where either the item has to be loaded or dropped.
- One cannot take fraction of an item.

Thief in a museum

What should he take?

- The burglar wishes to carry away the most valuable items subject to the weight constraint.
- No point taking fraction of an object, so he must make a decision to take the object entirely or leave it

- A thief wants to maximize his profit by stealing items from a museum. He has got a bag which can hold 60 Kg of goods.

items	Price	weight	ratio p/w
1	1000	10	100
2	2800	40	70
3	1300	20	65

The Greedy approach selects the items in order of 1, 2, 3

Knapsack Capacity = 60.

Knapsack Capacity after picking item 1 = 60 – 10 = 50.

Knapsack Capacity after picking item 2 = 50 – 40 = 10.

So only item 1 and item 2 can be picked up.

Profit after selling the items : 1000+2800 = 3800.

- A thief wants to maximize his profit by stealing items from a museum. He has got a bag which can hold 60 Kg of goods.
  - items | Price | weight | ratio p/w

items	Price	weight	ratio p/w
1	1000	10	100
2	2800	40	70
3	1300	20	65

The Greedy approach selects the items in order of 1, 2, 3

However, if items 2 and 3 are picked up.

total weight is 60, and both can be put in the bag.

Now profit is: 2800+ 1300 = 4100.

while profit using *Greedy approach* was only 3800.

So we did not get optimal solution.

Brute force approach

- Suppose 4 items with weights {2, 3, 4, 5} are to be loaded in a knapsack of capacity 5. The profit associated with the items is the set { 30,40, 50, 60}.
- The brute force approach to maximize the profit would be to try all possible combinations of items
- {0 0 0 0}, {0 0 0 1} , {0 0 1 0}, {0 0 1 1}, {0 1 0 0}, ... {1 1 1 0}, {1 1 1 1}
- and work out profit for each combination.

- For n items , there are $\mathbf{2}^{n}$ possible combinations of collecting the items
- so this approach involves O($\mathbf{2}^{n}$) operations.
- We study use of *Dynamic programming* approach for this

0/1 Knapsack Problem using DP

- The Dyn. Prog. approach involves working out the smaller subproblems first, and reusing the solutions to solve bigger subproblems.
- We try to solve multiple problems by considering different knapsack sizes
- sizes starting with 0, 1, 2, 3, 4. . ., and going all the way to GIVEN CAPACITY

- At each stage we figure out, as to which items can be loaded in the knapsack and then which combinations will give maximum profit.

- Solve the 0/1 knapsack problem given 4 items with

- item Weights { 2, 3, 4, 5 }
- corresponding Profits { 3, 4, 5, 6 }

- Let Knapsack capacity be 5 Kg.

- DP approach needs filling up a 2D PROFIT array

Form a 2 D table for P[i ,j], i is item and j is total weight of knapsack.

- Let us first do it without using DP formula
- Weights { 2, 3, 4, 5 } Profits { 3, 4, 5, 6 }
- Each column represents a subproblem with increasing knapsack capacity

	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	(no weight, no profit)

Form a 2 D table for P[i ,j], i is item and j is total weight of knapsack.

- Let us first do it without using DP formula
- Weights { 2, 3, 4, 5 } Profits { 3, 4, 5, 6 }
- Consider that thief considers just item 1.

	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	(no weight, no profit)
1	0	0	3	3	3	3	(Only item 1)

Form a 2 D table for P[i,j], i is item and j is total weight of knapsack.

- Now consider both items 1 and 2.
- Weights { 2, 3, 4, 5 }, Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	(Only item 1)
2	0	0	3	4	4	7	(Only items 1 and 2)

Form a 2 D table for P[i,j], i is item and j is total weight of knapsack.

- Now consider items 1, 2, and 3.
- Weights { 2, 3, 4, 5 }, Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	(Only item 1)
2	0	0	3	4	4	7	(Only items 1 and 2)
3	0	0	3	4	5	7	(items 1 , 2 and 3)

Form a 2 D table for P[i,j], i is item and j is total weight of knapsack.

- Now consider all the items. You can put either item 4, or items 2 & 3.
- Weights { 2, 3, 4, 5 }, Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	(Only item 1)
2	0	0	3	4	4	7	(Only items 1 and 2)
3	0	0	3	4	5	7	(items 1 , 2 and 3)
4	0	0	3	4	5	7	(all 4 items)

- We filled up all elements of array by simple mental reasoning.
- Do we really need D.P. formulation?

- Given a knapsack of 5 Kg , work out 0/1 Knapsack for 4 items
- Weights { 2, 3, 4, 5 }

  Profits { 3, 4, 5, 6 }

Start with knapsack of 1 kg, 2 kg, 3 kg…. . . .

- 4 Kg

- 5 Kg

Dyn Prog approach for knapsack problem

- Let us know use Dynamic Programming to solve the same problem
- Create a 2D array for profit.
- items along each row and knapsack size along columns.

- $P[i, j] = P[i-1, j]$ if $w_i > j$ if item weight can go in Knapsack size
- *(Keep the left value)*
- $P[i, j] = \max\{$ **$P[i-1, j]$,** **$P_i + P[i-1, j - w_i$** $]\}$ if $w_i < j$
- *(choose max value between left*
- *and current profit +<u>previous best </u>entries)*

- .
- P[i, j] = P[i-1, j]   if $w_i > j$ if item weight can go in Knapsack size
- *(Keep the left value)*
- P[i, j] = max{ **P[i-1, j] ,**   **P_i + P[i-1, j – w_i]**}   if $w_i < j$

Weights { 2, 3, 4, 5 }
Profits { 3, 4, 5, 6 }

0	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	
1	0	0	3				
2							
3							
4							

- .

- P[i, j] = P[i-1, j]   if $w_i > j$ if item weight can go in Knapsack size
- *(Keep the left value)*
- P[i, j] = max{ **P[i-1, j]** , **P_i + P[i-1, j – w_i]**}   if $w_i < j$

- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

(Knapsack capacity)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2						
3						
4						

- Let us know use Dynamic Programming to solve the same problem
- $P[\,i, j\,] = P[\,i\text{-}1, j]$ if $w_i > j$
- $P[\,i, j\,] = \max\{\,P[\,i\text{-}1, j\,],\quad P_i + P[i\text{-}1, j{-}w_i\,]\}$ if $w_i < j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

0	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	
1	0	0	3	3			$\{P_1 = 3,\ w_1 = 2\}$

- In P[1,3], $i = 1, j = 3$ $\quad w_1 < j$ so $P[1,3] = \max\{P[0,3],\ 3+P[0,1]\} = 3$

- Dynamic Programming equations
- $P[i, j] = P[i-1, j]$ if $w_i > j$
- $P[i, j] = \max\{P[i-1, j], \quad P_i + P[i-1, j-w_i]\}$ if $w_i < j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }
- 0 0 1 2 3 4 5 (Knapsack capacity)
- 0 | 0 0 0 0 0 0
- 1 | 0 0 3 3 3 $\{P_1 = 3, w_1 = 2\}$
- In P[1,3], i = 1, j = 3 $w_1 < j$ so P[1,3] = max{P[0,3] , 3+P[0,1]} = 3
- In P[1,4], i = 1, j = 4 so P[1,4] = max{P[0,4] , 3+P[0,2]} = 3

- Dynamic Programming equations
- $P[i, j] = P[i-1, j]$ if $w_i > j$
- $P[i, j] = \max\{ P[i-1, j], \quad P_i + P[i-1, j-w_i] \}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			

(Knapsack capacity)

$\{ P_2 = 4, w_2 = 3 \}$

In $P[2,2]$, $i = 2$, $j = 2$ $\quad w_2 > j$ $\quad$ so $P[2,2] = P[1,2] = 3$

- Dynamic Programming equations
- $P[\ i, j\ ] = P[\ i\text{-}1, j]$ if $w_i > j$
- $P[\ i, j\ ] = \max\{\ P[\ i\text{-}1, j\ ],\ \ P_i + P[i\text{-}1, j - w_i\ ]\}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		

(Knapsack)

$\{ P_2 = 4 ,\ w_2 = 3\}$

- In P[2,2], i = 2, j = 2 $w_2 > j$ so P[2,2] = P[1,2] = 3
- In P[2,3], i = 2, j = 3 so P[2,3] = max{P[1,3] , 4+P[1,0]} = 4

- Dynamic Programming equations
- $P[\ i, j\ ] = P[\ i\text{-}1, j]$ if $w_i > j$
- $P[\ i, j\ ] = \max\{\ P[\ i\text{-}1, j\ ],\quad P_i + P[i\text{-}1, j–w_i\ ]\}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	

(Knapsack)

$\{\ P_2 = 4\ ,\ w_2 = 3\}$

- In P[2,2], i = 2, j = 2 $w_2 > j$ so P[2,2] = P[1,2] = 3
- In P[2,3], i = 2, j = 3 so P[2,3] = max{P[1,3] , 4+P[1,0]} = 4
- In P[2,4], i = 2, j = 4 so P[2,4] = max{P[1,4] , 4+P[1,1]} = 4

- Dynamic Programming equations
- $P[i, j] = P[i-1, j]$ if $w_i > j$
- $P[i, j] = \max\{ P[i-1, j], \; P_i + P[i-1, j-w_i] \}$ if $w_i < j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }
- 0 0 1 2 3 4 5 (Knapsack capacity)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7

$\{ P_2 = 4 , w_2 = 3 \}$

- In P[2,2], $i = 2, j = 2$ $w_2 > j$ so $P[2,2] = P[1,2] = 3$
- In P[2,3], $i = 2, j = 3$ $w_2 = j$ so $P[2,3] = \max\{P[1,3], 4+P[1,0]\} = 4$
- In P[2,4], $i = 2, j = 4$ $w_2 < j$ so $P[2,4] = \max\{P[1,4], 4+P[1,1]\} = 4$
- In P[2,5], $i = 2, j = 5$ $w_2 < j$ so $P[2,5] = \max\{P[1,5], 4+P[1,2]\} = 7$

- Dynamic Programming equations
- $P[\, i, j\,] = P[\, i\text{-}1, j]$ if $w_i > j$
- $P[\, i, j\,] = \max\{ P[\, i\text{-}1, j\,],\quad P_i + P[i\text{-}1, j\text{–}w_i\,]\}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }
- 0 0 1 2 3 4 5 (Knapsack capacity)
- 0 | 0 0 0 0 0 0
- 1 | 0 0 3 3 3 3
- 2 | 0 0 3 4 4 7 $\{ P_3 = 5 , w_2 = 4\}$
- 3 | 0 0 3 4 5
- P[3,4] i=3, j =4 $w_i = j$ so P[3,4] = max{ P[2,4], 5+P[2,0] } = 5

- Dynamic Programming equations
- $P[\,i, j\,] = P[\,i\text{-}1, j]$ if $w_i > j$
- $P[\,i, j\,] = \max\{\, P[\,i\text{-}1, j\,],\ P_i + P[i\text{-}1, j\!-\!w_i\,]\}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7

(Knapsack capacity)

$\{ P_3 = 5 , w_2 = 4\}$

- P[3,4] i=3, j =4 $w_i = j$ so P[3,4] = max{ P[2,4], 5+P[2,0] } = 5
- P[3,5] i=3, j =5 $w_i < j$ so P[3,5] = max{ P[2,5], 5+P[2,1] } = 7

- Dynamic Programming equations
- $P[i, j] = P[i-1, j]$ if $w_i > j$
- $P[i, j] = \max\{ P[i-1, j], \quad P_i + P[i-1, j-w_i] \}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

0	0	1	2	3	4	5	(Knapsack capacity)
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	
2	0	0	3	4	4	7	
3	0	0	3	4	5	7	
4	0	0	3	4	5	7	

- The table gives total profit for the problem.
- To figure out which particular items get selected, see the last part of the solution to this problem in

https://codecrucks.com/knapsack-problem-using-dynamic-programming/

- Now solve the same 0/1 knapsack problem with a Knapsack capacity of **6 Kg.**

- item Weights { 2, 3, 4, 5 }
- corresponding Profits { 3, 4, 5, 6 }

- work out

- Dynamic Programming equations
- $P[i, j] = P[i-1, j]$ if $w_i > j$
- $P[i, j] = \max\{ P[i-1, j],\ P_i + P[i-1, j-w_i]\}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }

	0	1	2	3	4	5	**6** (Knapsack capacity)
0	0	0	0	0	0	0	**0**
1	0	0	3	3	3	3	**3**
2	0	0	3	4	4	7	**7**

- $\{ P_2 = 4 , w_2 = 3\}$
- In P[2,2], $i = 2, j = 2$ $w_2 > j$ so $P[2,2] = P[1,2] = 3$
- In P[2,6], $i = 2, j = 6$ so $P[2,6] = \max\{P[1,3], 4+P[1,3]\} = 7$

- Dynamic Programming equations
- $P[i, j] = P[i-1, j]$ if $w_i > j$
- $P[i, j] = \max\{P[i-1, j], P_i + P[i-1, j-w_i]\}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }
- 0 0 1 2 3 4 5 6 (Knapsack capacity)

0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3
2	0	0	3	4	4	7	7
3	0	0	3	4	5	7	

- $\{P_3 = 5, w_3 = 4\}$
- In P[3,5], i = 3, j = 5 $w_2 < j$ so $P[3,5] = \max\{P[2,5], 5+P[2,1]\} = 7$

- Dynamic Programming equations
- $P[i, j] = P[i-1, j]$ if $w_i > j$
- $P[i, j] = \max\{ P[i-1, j],\ P_i + P[i-1, j-w_i]\}$ if w_i not $> j$
- Weights { 2, 3, 4, 5 }
- Profits { 3, 4, 5, 6 }
- 0 0 1 2 3 4 5 6 (Knapsack capacity)

	0	1	2	3	4	5	6	
0	0	0	0	0	0	0	0	
1	0	0	3	3	3	3	3	
2	0	0	3	4	4	7	7	
3	0	0	3	4	5	7	8	$\{ P_3 = 5, w_3 = 4\}$

- In P[3,5], $i = 3, j = 5$ $w_2 < j$ so $P[3,5] = \max\{P[2,5], 5+P[2,1]\} = 7$
- In P[3,6], $i = 3, j = 6$ $w_2 < j$ so $P[3,6] = \max\{P[2,6], 5+P[2,2]\} = 8$

Complexity of 0/1 Knapsack

- Brute force technique is all possible combinations: $O(2^n)$
- Dynamic Programming solution needs $O(n\ W)$
- where n is number of items and
- W is knapsack capacity.

- So if W is 10, complexity looks small, but
- if W is 1000, the solution becomes almost exponential.

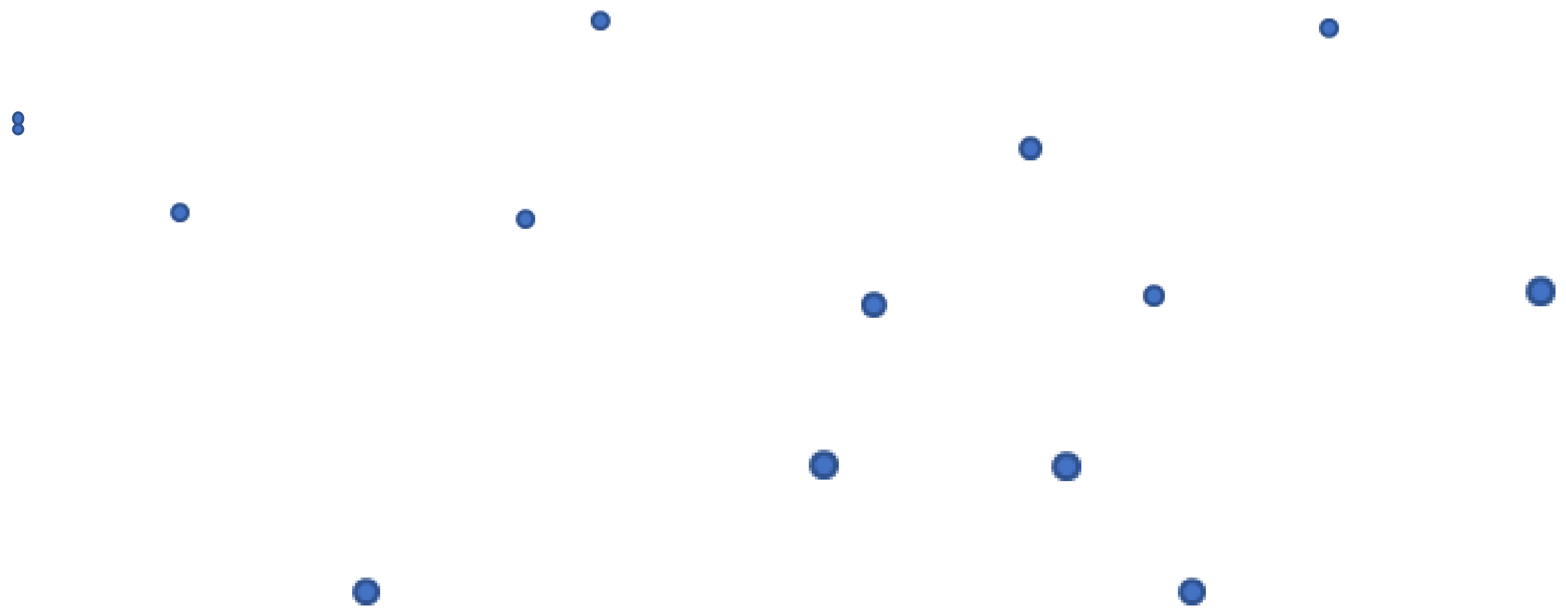
<u>prob.3.</u>

Travelling Salesman Problem

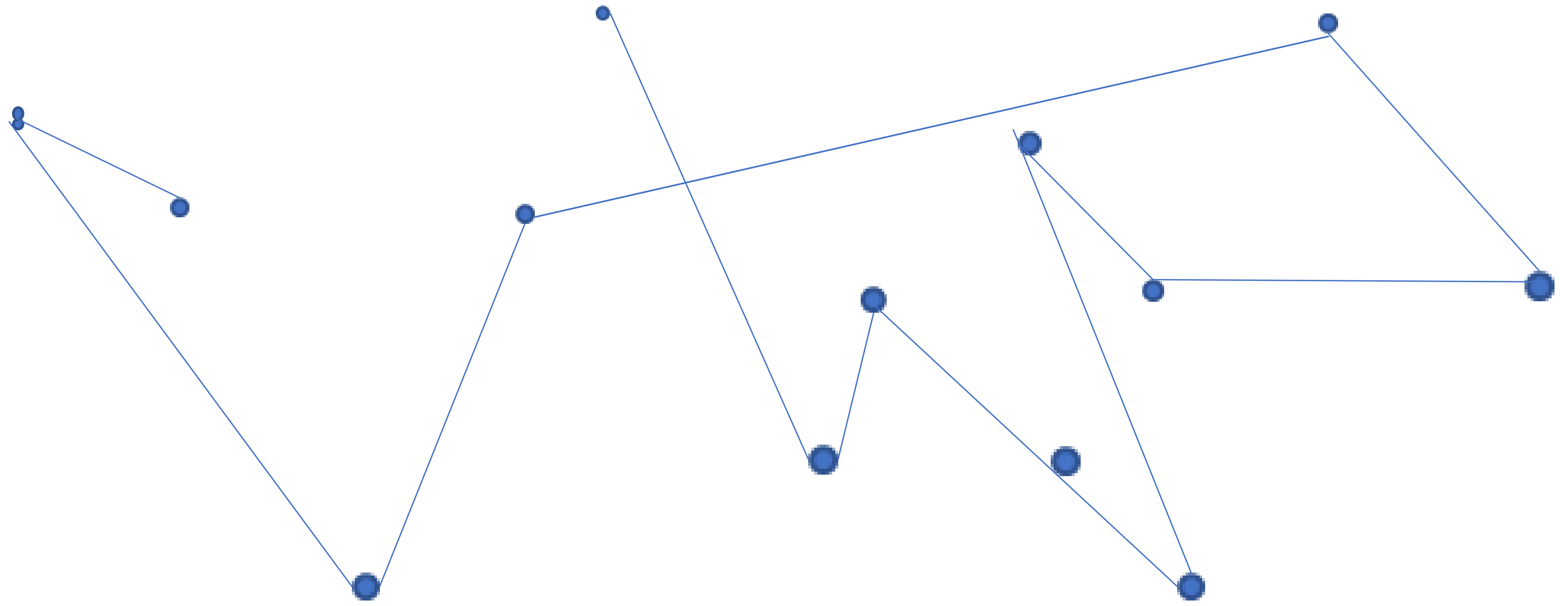
7. Travelling salesman Problem

- A salesman needs to visit number of cities in connection with his work.
- He can base himself in one city, say A
- visit one new city everyday, and come back to A
- However, in order to save travelling cost, he decides to find the shortest tour, so that in one go , he visits all the cities one by one and finally return back to A.
- *Constraint:* Every city to be visited only once

Cities

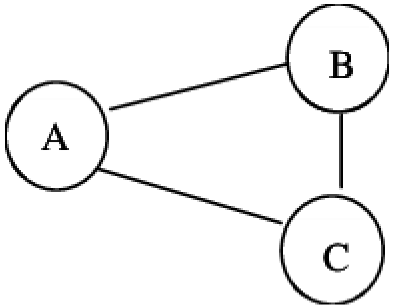


One possible Route



- 3 city network
- A B C
- A C B

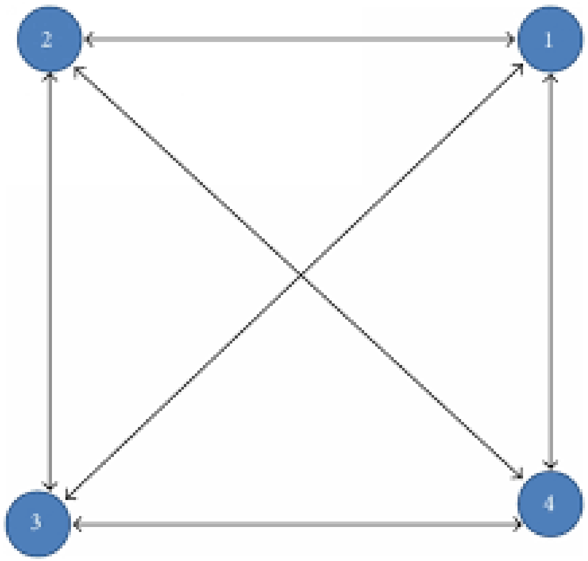
- 2 paths



- 4 city network . Consider all paths starting from city 1

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

- 6 paths



How Hard?

Number of possible tours:

Suppose there are N cities.

Salesman can start from any of N cities:

Once he visits first city, he can choose any of (N-1) cities.

Then from there, he can choose further any of (N-2) cities.

Number of possible tours: $(N-1) \times (N-2) \times \ldots\ldots.$

How Hard?

Number of possible tours:

N-1 ! = (N-1) ×(N-2)×…..3×2×1

For just 11 cities, starting from city 1, possible tours

10! = 3,628,200

Around 3 million possible tours…..

route calculation by Salesman

- The salesman has got a table
- which lists distance between each pair of 11 cities

- He sits down and works out cost for all possible paths,
- so that he can choose the tour with the smallest cost.

- If it involves 1 minute of work for each possible tour,
- how soon can he finish up his calculations so that
- he can start the actual trip?

- Figure out on your notebooks !
- Take 7 hours per day, and 5 days per week…..
- ??

- number of paths = 10 !
- = 3628800 minutes
- = 60480 hours
- Suppose he works for 7 hours a day
- = 8640 days
- = 1728 weeks (5 days a week)
- = 33.23 years

- So he will start his trip after 33 years.

- 11 is a very very small numbr
- Suppose there are 21 cities

For 21 cities

$20! \sim 2.43\times10^{18}$ (2.43 quadrillion tours)

- So brute force technique will not work.
- However, we can try Dynamic Programming approach
- Approach needs a 2D array of possible path lengths.
- To start with, distance matrix d[i, j] between node i and node j is computed from graph data

$$d[i, j] = 0 \qquad \text{if } i = j$$

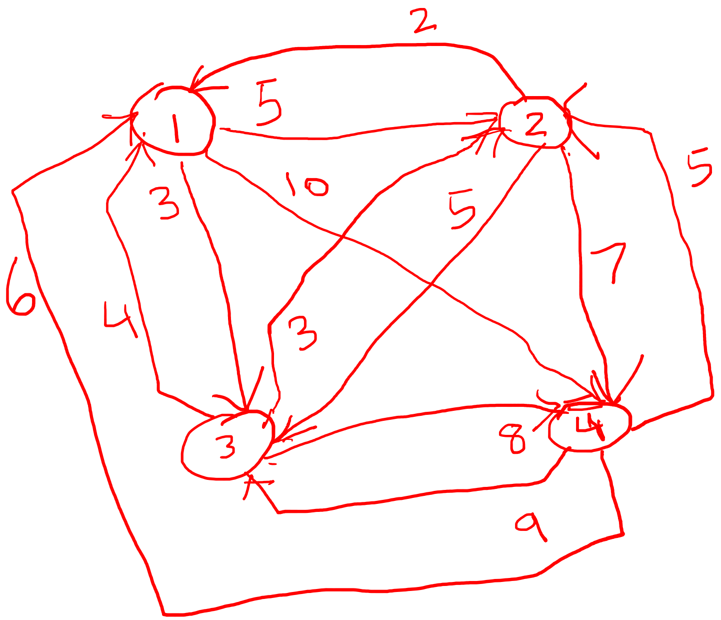
$$= w_{ij} ,$$

- Assume starting node is 1.
- Let cost (i, {S})= shortest path from node i to node 1, using nodes in set S.
- d[i,j] = distance of node i to node j.
- Consider a 4 node directed graph
- cost of going from i to j may be different from cost of going from j to i

- The DP approach is to first compute smaller best paths
- and slowly increase the number of cities for going back to starting node..

- A 4 node graph with
- d[i, j]

	1	2	3	4
1	0	5	3	10
2	2	0	5	7
3	4	3	0	8
4	6	5	9	0



Cost of going back directly to node 1

- cost of going back from node 2 to node 1.
- cost of going back from node 3 to node 1.
- cost of going back from node 4 to node 1.

- A 4 node graph with
- d[i, j]

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & 3 & 10 \\ 2 & 2 & 0 & 5 & 7 \\ 3 & 4 & 3 & 0 & 8 \\ 4 & 6 & 5 & 9 & 0 \end{array}$$

- GOING BACK TO 1 DIRECTLY
- SMALLEST PATHS

- cost of going back from node 2 to node 1.
- Cost(2, { Φ}) = d[2,1] = 2

- cost of going back from node 3 to node 1.
- Cost(3, { Φ}) = d[3,1] = 4

- cost of going back from node 4 to node 1.
- Cost(4, { Φ}) = d[4,1] = 6

Going indirectly through one more node

- Next we find out if 2-3-1 is cheaper or 2-4-1 is cheaper.
- Similarly , we compare costs of 3-2-1 and 3-4-1
- and compare costs of 4-2-1 with 4-3-1

- A 4 node graph with
- d[i, j]

	1	2	3	4
1	0	5	3	10
2	2	0	5	7
3	4	3	0	8
4	6	5	9	0

- USING ONE INTERMEDIATE NODE
- Cost of going from 2 to 1 through 3 / 4
- 2 – 3 – 1
- cost (2,{3}) = d[2,3] +cost(3,1) = 5+4 = 9
- 2 – 4 – 1
- cost (2,{4}) = d[2,4] +cost(4,1) = 7+6 = 13

- Cost of going from 3 to 1 through 2 / 4
- 3 – 2 – 1
- cost (3,{2}) = d[3,2] +c (2,1) = 3+2 = 5
- 3 – 4 – 1
- cost (3,{4}) = d[3,4] +c (4,1) = 8+6 = 14

- A 4 node graph with
- d[i, j]

	1	2	3	4
1	0	5	3	10
2	2	0	5	7
3	4	3	0	8
4	6	5	9	0

- USING ONE INTERMEDIATE NODE

- Cost of going from 4 to 1 through 2 /3
- 4 – 2 – 1
- cost (4,{2}) = d[4,2] +c (2,1) = 5+2 = 7
- 4 – 3 – 1
- cost (4,{3}) = d[4,3] +c (3,1) = 9+4 = 13

- Now we consider 2 intermediate nodes for going back to 1.
- we try to find out if 2-3-4-1 is cheaper or 2-4-3-1 is cheaper.
- Similarly , we compare costs of 3-2-4-1 and 3-4-2-1
- and compare costs of 4-2-3-1 with 4-3-2-1

- 4 node graph with
- d[i, j]

- 1 2 3 4
- 1 0 5 3 10
- 2 2 0 5 7
- 3 4 3 0 8
- 4 6 5 9 0

	1	2	3	4
1	0	5	3	10
2	2	0	5	7
3	4	3	0	8
4	6	5	9	0

- USING 2 INTERMEDIATE NODES

- Is it cheaper to do 2-3-4-1 or 2-4-3-1?
- cost (2,{3,4}) = min { d[2,3] + cost (3,{4}) ,
  d[2,4] +cost (4,{3}) }
  = min{ 5 +14, 7 +13 } = 19

- Is it cheaper to do 3-2-4-1 or 3-4-2-1?
- cost (3,{2,4}) = min { d[3,2] + cost (2,{4}) ,
  d[3,4] +cost (4,{2}) }
  = min { 3+13, 8+7 } = 15

- cost (4,{2,3}) = min {d[4,2] + cost (2,{3}) ,
  d[4,3] cost (3,{2}) }
  = min{ 5 + 9, 9+5} = 14

- USING 3 INTERMEDIATE NODES
- Finally, cost of total tour
- min { 1-2 and best path to return, 1-3 and return, 1-4 and return}
- = min { d[1,2] + cost (2, {3,4}), // using cheaper path 1-2-3-4-1
- d[1,3] + cost { 3, {2,4}), // using cheaper path 1-3-2-4-1

  d[1, 4] + cost {4, {2,3}) } // using cheaper path 1-4-2-3-1
- = min { 5 +19, 3+15, 10+14 }
- = 18
- So the best order is 1 – 3 – 2 – 4 – 1

Complexity of TSP

- Brute force TSP of n cities needs $O(\mathbf{n!})$
- In DP approach there are at most $O(n\ \mathbf{2}^{n})$ sub problems.
- Each can be solved in linear time.

- So overall complexity of TSP using Dynamic Prog. is $O(\mathbf{n^2\ 2}^{n})$
- which is better?
- For n = 10, TSP using brute force needs   3,628,200 operations
- and using Dynamic Prog it needs order of   100,000 operations

- How do the salesmen handle this problem?

Drilling holes on PCB

- To connect a conductor on one layer with a conductor on another layer, or to position the pins of ICs, holes have to be drilled.
- To drill two holes of different diameters consecutively, the head of the machine has to move to a tool box and change the drilling equipment.
- This is quite time consuming.
- Thus it is clear that one has to choose some diameter, drill all holes of the same diameter, change the drill, drill the holes of the next diameter, etc.
- Thus, this drilling problem can be viewed as a series of TSPs, one for each hole diameter

Order-picking problem in warehouses

- Assume that a warehouse receives an order for certain items.
- Some vehicle has to collect all items of this order to ship them to the customer.

- The storage locations of the items correspond to the nodes of the graph.
- The distance between two nodes is the time needed to move the vehicle from one location to the other.

- The problem of finding a shortest route for the vehicle with minimum pickup time can now be solved as a TSP.

Vehicle Routing problem

- Suppose n customers require certain amounts of some commodities and a supplier has to satisfy all demands with a FIXED fleet of trucks.
- The problem is to find an assignment of customers to the trucks

  and a delivery schedule for each truck

  so that the capacity of each truck is not exceeded and the total travel distance is minimized.
- This problem is solvable as a TSP if there are no time and capacity constraint and number of trucks are fixed.