Module – 2 Combinatorial Optimization

- Dynamic Programming
- Branch and Bound

Dynamic Programming

- **The idea of DP**
  - Dynamic Programming is mainly an optimization over plain recursion.
  - Recursive solution that has repeated calls for same inputs,
  - This can be optimized using Dynamic Programming.
  - The idea is to simply store the results of sub-problems,
  - Need not re-compute them when needed later.
  - This simple optimization reduces time complexities from exponential to polynomial.
  - For example, a simple recursive solution for Fibonacci Numbers, is exponential time complexity and
  - DP is to optimize it by storing solutions of sub-problems, time complexity reduces to linear.

Linear Vs Exponential Complexity

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

RECURSION : Exponential Complexity

```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

DP: Linear time

0/1 Knapsack Problem

Dynamic programming with overlapping sub structures

Given n items of

integer weights: $w_1 \quad w_2 \; \ldots \; w_n$

values: $v_1 \quad v_2 \; \ldots \; v_n$

a knapsack of integer capacity W

find most valuable subset of the items that fit into the knapsack

Consider instance defined by first i items and capacity j ($j \leq W$).

Let V[i,j] be optimal value of such an instance. Then

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \geq 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

DP - Algorithm

Algorithm DPKnapsack($w[1..n]$, $v[1..n]$, W)
    var $V[0..n,0..W]$, $P[1..n,1..W]$: int
    for $j := 0$ to W do
        $V[0,j] := 0$
    for $i := 0$ to n do
        $V[i,0] := 0$
    for $i := 1$ to n do
        for $j := 1$ to W do
            if $w[i] \leq j$ and $v[i] + V[i-1,j-w[i]] > V[i-1,j]$ then
                $V[i,j] := v[i] + V[i-1,j-w[i]]$; $P[i,j] := j-w[i]$
            else
                $V[i,j] := V[i-1,j]$; $P[i,j] := j$
    return $V[n,W]$ and the optimal subset by backtracking

Longest Common Subsequence (LCS)

- A subsequence of a sequence/string *S* is obtained by deleting zero or more symbols from *S*. For example, the following are some subsequences of ***"president": pred, sdn, predent***. In other words, the letters of a subsequence of S appear in order in *S*, but they are not required to be consecutive.

- The longest common subsequence problem is to find a maximum length common subsequence between two sequences.

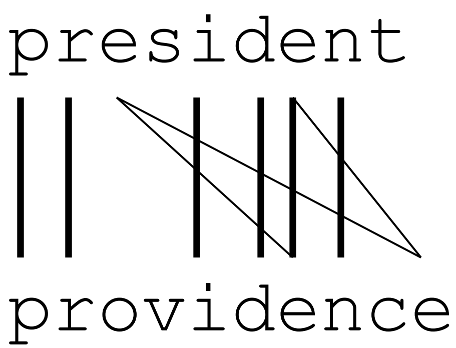
LCS

For instance,

Sequence 1: president

Sequence 2: providence

Its LCS is priden.



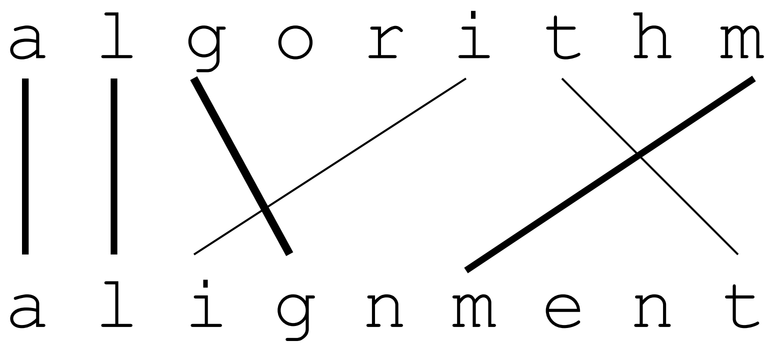
LCS

Another example:

Sequence 1: algorithm

Sequence 2: alignment

One of its LCS is algm.



How to compute LCS?

- Let A=$a_1a_2\ldots a_m$ and B=$b_1b_2\ldots b_n$.
- *len*(i, j): the length of an LCS between $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$
- With proper initializations, *len*(i, j) can be computed as follows.

$$len(i,j)=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ len(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i=b_j\ ,\\ \max(len(i,j-1),len(i-1,j)) & \text{if } i,j>0 \text{ and } a_i\neq b_j.\end{cases}$$

procedure *LCS-Length(A, B)*

1. **for** $i \leftarrow 0$ **to** m **do** $len(i,0) = 0$
2. **for** $j \leftarrow 1$ **to** n **do** $len(0,j) = 0$
3. **for** $i \leftarrow 1$ **to** m **do**
4.   **for** $j \leftarrow 1$ **to** n **do**
5.   **if** $a_i = b_j$ **then** $\begin{bmatrix} len(i,j) = len(i-1,j-1)+1 \\ prev(i,j) = "\nwarrow" \end{bmatrix}$
6.   **else if** $len(i-1,j) \geq len(i,j-1)$
7.   **then** $\begin{bmatrix} len(i,j) = len(i-1,j) \\ prev(i,j) = "\uparrow" \end{bmatrix}$
8.   **else** $\begin{bmatrix} len(i,j) = len(i,j-1) \\ prev(i,j) = "\leftarrow" \end{bmatrix}$
9. **return** *len* and *prev*

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Running time and memory: O(*mn*) and O(*mn*).

The backtracking algorithm

procedure *Output-LCS(A, prev, i, j)*

if $i = 0$ **or** $j = 0$ **then return**

if $prev(i, j) = \text{"}\nwarrow\text{"}$ **then** $\left[\begin{array}{l} Output-LCS(A, prev, i-1, j-1) \\ \text{print} \quad a_i \end{array}\right.$

else if $prev(i, j) = \text{"}\uparrow\text{"}$ **then** *Output-LCS(A, prev, i-1, j)*

else *Output-LCS(A, prev, i, j-1)*

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Output: *priden*

Branch and Bound

- **Idea of B&B**
  - Branch and bound is an algorithm design paradigm.
  - Generally used for solving combinatorial optimization problems.
  - Problems are typically exponential in terms of time complexity and
  - Require exploring all possible permutations in worst case.
  - The Branch and Bound Algorithm technique solves these problems relatively quickly.

Analysis

- Brute-force approach to evaluate every possible tour and select the best one.
- ***For n number of vertices in a graph, there are (n - 1)!*** number of possibilities.
- Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, ***though there is no polynomial time algorithm.***
- Let us consider a graph G = (V, E), where V is a set of cities and E is a set of weighted edges. An edge e(u, v) represents that vertices u and v are connected. Distance between vertex u and v is d(u, v), which should be non-negative.
- Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour.
- Need to know j, since this will determine which cities are most convenient to visit next.
- Need to know all the cities visited so far, so that no need to repeat any of them. Hence, this is an appropriate sub-problem.

Problem Formulation

For a subset of cities S Є {1, 2, 3, ... , n} that includes 1, and j Є S, let C(S, j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When |S| > 1, we define C(S, 1) = since the path cannot start and end at 1.

Now, let express C(S, j) in terms of smaller sub-problems.

We need to start at 1 and end at j.

We should select the next city in such a way that

$$C(S,j)=\min C(S-\{j\},i)+d(i,j),$$

Where $i \in S$ and $i \neq j$

Algorithm: Traveling-Salesman-Problem

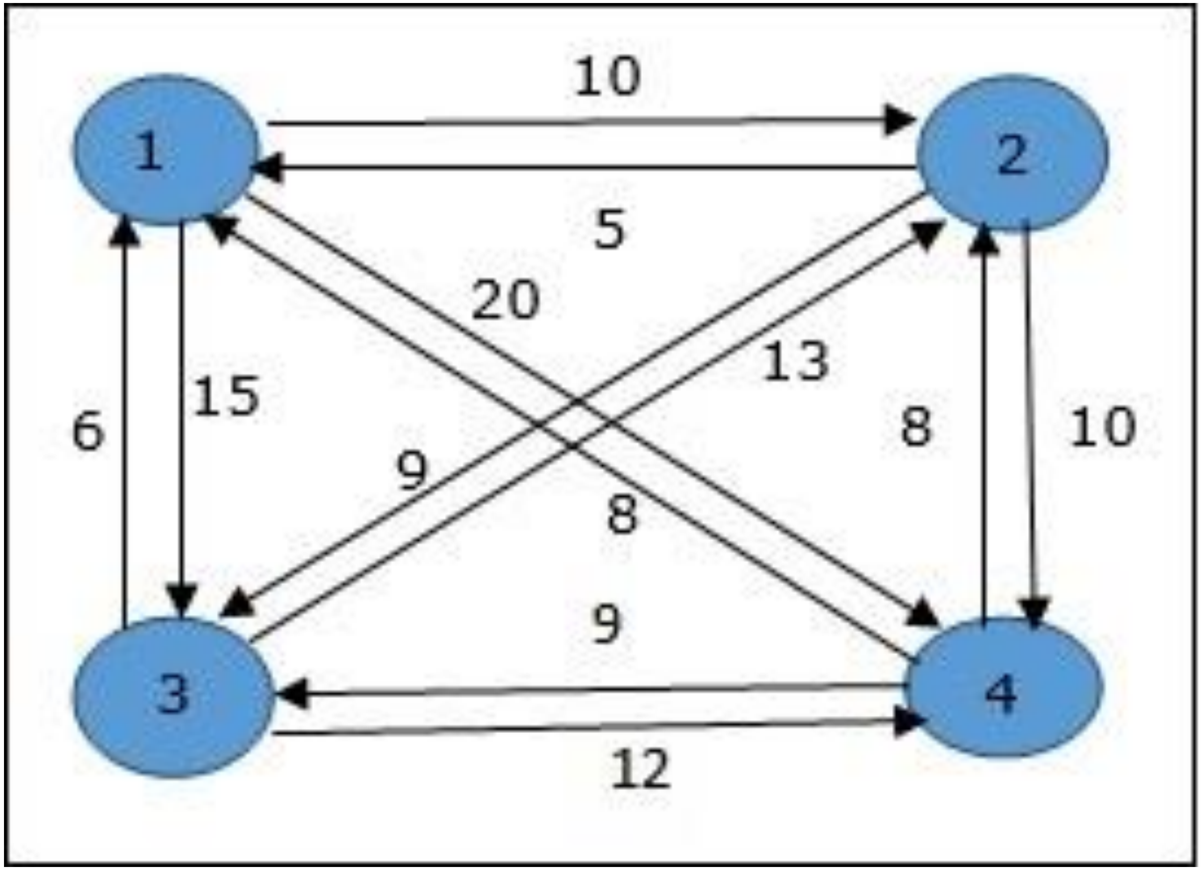
C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
    C (S, 1) = ∞
  for all j Є S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return min C ({1, 2, 3, …, n}, j) + d(j, i)

Analysis

There are at the most (2^n.n) sub-problems and each one takes linear time to solve.

Therefore, the total running time is O(2^n.n^2).

Travelling Salesman Problem(TSP)



TSP : Cost Matrix

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$S = \Phi$

$Cost(2,\Phi,1)=d(2,1)=5$

$Cost(3,\Phi,1)=d(3,1)=6$

$Cost(4,\Phi,1)=d(4,1)=8$

$S = 1$

$Cost(i,s)=min\{Cost(j,s-(j))+d[i,j]\}$

$Cost(2,\{3\},1)=d[2,3]+Cost(3,\Phi,1)=9+6=15$

$Cost(2,\{4\},1)=d[2,4]+Cost(4,\Phi,1)=10+8=18$

$Cost(3,\{2\},1)=d[3,2]+Cost(2,\Phi,1)=13+5=18$

$Cost(3,\{4\},1)=d[3,4]+Cost(4,\Phi,1)=12+8=20$

$Cost(4,\{3\},1)=d[4,3]+Cost(3,\Phi,1)=9+6=15$

$Cost(4,\{2\},1)=d[4,2]+Cost(2,\Phi,1)=8+5=13$

$S = 2$

$Cost(2,\{3,4\},1)=d[2,3]+Cost(3,\{4\},1)=9+20=29$

$d[2,4]+Cost(4,\{3\},1)=10+15=25$

$Cost(3,\{2,4\},1)=d[3,2]+Cost(2,\{4\},1)=13+18=31$

$d[3,4]+Cost(4,\{2\},1)=12+13=25$

$Cost(4,\{2,3\},1)=d[4,2]+Cost(2,\{3\},1)=8+15=23$

$d[4,3]+Cost(3,\{2\},1)=9+18=27$

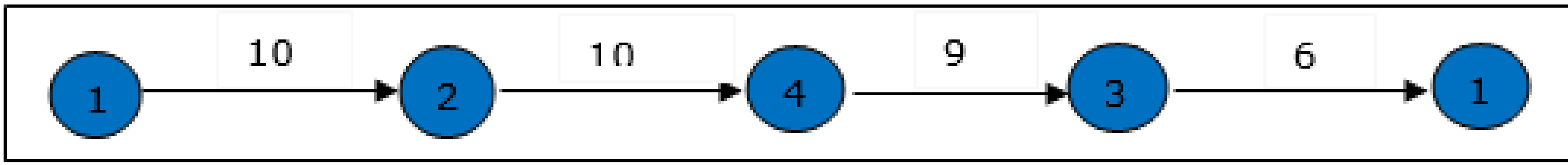
$S = 3$

$Cost(1,\{2,3,4\},1)=d[1,2]+Cost(2,\{3,4\},1)=10+25=35$

$d[1,3]+Cost(3,\{2,4\},1)=15+25=40$

$d[1,4]+Cost(4,\{2,3\},1)=20+23=43$

The minimum cost path is 35.



Start from cost {1, {2, 3, 4}, 1}, we get the minimum value for d [1, 2].
When s = 3, select the path from 1 to 2 (cost is 10) then go backwards.
When s = 2, we get the minimum value for d [4, 2].
Select the path from 2 to 4 (cost is 10) then go backwards.
When s = 1, we get the minimum value for d [4, 3].
Selecting path 4 to 3 (cost is 9), then we shall go to then go to s = Φ step.
We get the minimum value for d [3, 1] (cost is 6).

Assignment Question

Solve the TSP shown in figure