General DP Remarks

Optimal Substructure

- Create optimal solution to problem using optimal solutions to subproblems.
- Can't use DP if optimal solution to a problem does not require subproblem solutions to be optimal.
  → Often happens when subproblems are *not independent* of each other.

Overlapping Subproblems

- For DP to be useful, recursive algorithm should require us to compute optimal solutions to the *same subproblems* over and over again.
- In total, there should be a small number of distinct subproblems (i.e. polynomial in input size).

LCS

$$LCS[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS[i-1, j-1] & \text{if } x_i = y_i \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

		j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8
			p	r	i	n	t	i	n	g
i=0		0	0	0	0	0	0	0	0	0
i=1	s	0	0	0	0	0	0	0	0	0
i=2	p	0	1↖	1←	1←	1←	1←	1←	1←	1←
i=3	r	0	1↑	2↖	2←	2←	2←	2←	2←	2←
i=4	i	0	1↑	2↑	3↖	3←	3←	3←	3←	3←
i=5	n	0	1↑	2↑	3↑	4↖	4←	4←	4←	4←
i=6	g	0	1↑	2↑	3↑	4↑	4↑	4↑	4↑	5↖
i=7	t	0	1↑	2↑	3↑	4↑	5↖	5←	5←	5←
i=8	i	0	1↑	2↑	3↖	4↑	5↑	6↖	6←	6←
i=9	m	0	1↑	2↑	3↑	4↑	5↑	6↑	6↑	6↑
i=10	e	0	1↑	2↑	3↑	4↑	5↑	6↑	6↑	6↑

OBST

- Any subtree of a BST contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.
- If T is an OBST, and T contains subtree T' with keys $k_i, \ldots, k_j$, then T' must be an OBST for keys $k_i, \ldots, k_j$.
- Examine all candidate roots k_r for $i \le r \le j$.
- Determine all OBSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$

$$e[i,j] = \begin{cases} 0 & \text{if } i = j-1 \\ \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i,j)\} & \text{if } i \le j \end{cases}$$

$root[i,j]$ = root of subtree with keys $k_i, \ldots, k_j$ for $1 \le i \le j \le n$

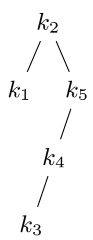
$$w[1, \ldots, n+1, 0, \ldots, n] = \text{ sum of probabilities}$$
$$w[i, i-1] = 0 \text{ for } 1 \le i \le n$$
$$w[i,j] = w[i, j-1] + p_j \text{ for } 1 \le i \le j \le n$$

Consider 5 keys with search probabilities $p_1 = 0.25, p_2 = 0.2, p_3 = 0.05, p_4 = 0.2, p_5 = 0.3$

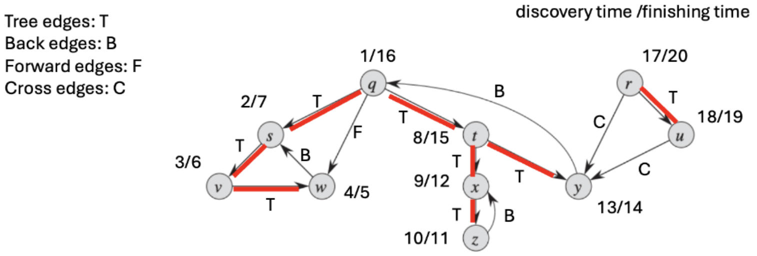
w	0	1	2	3	4	5
1	0	0.25	0.45	0.5	0.7	1.0
2		0	0.2	0.25	0.45	0.55
3			0	0.05	0.25	0.55
4				0	0.2	0.5
5					0	0.3
6						0

e	0	1	2	3	4	5
1	0	0.25	0.65	0.8	1.25	2.1
2		0	0.2	0.3	0.75	1.35
3			0	0.05	0.3	0.85
4				0	0.2	0.7
5					0	0.3
6						0

r	0	1	2	3	4	5
1		1	1	1	2	2
2			2	2	2	4
3				3	4	5
4					4	5
5						5
6						



DFS



$$(q\,[s\{v(ww)v\}s]\,[t\{x(zz)x\}yyt]q)\,(r[uu]r)$$

Tree Edges Are edges in depth-first forest G_π. Edge (u,v) is a tree edge if v was first discovered by exploring edge (u,v)
Back Edges Are edges (u,v) connecting a vertex u to an ancestor in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
Forward Edges Are nontree edges (u,v) connecting a vertex u to a descendant v in a depth-first tree.
Cross Edges Are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between verticies in different depth-first trees.

Knapsack

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w_i > w \\ \max\{v_i + c[i-1, w-w_i], c[i-1,w]\} & \text{if } i > 0 \text{ and } w \ge w_i \end{cases}$$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22

$c[2,1] = c[1,1]$ because $w_2 > w$
$c[2,2] = \max(v_2 + c[1,0], c[1,2]) \quad = 10$
$c[2,3] = \max(v_2 + c[1,1], c[1,3]) = \max(10+6, 6) \quad = 16$
$c[2,4] = \max(v_2 + c[1,2], c[1,4]) = \max(10+6, 6) \quad = 16$
$c[3,3] = \max(v_3 + c[2,0], c[2,3]) = \max(12, 16) \quad = 16$

Graphs

Handshaking Lemma

$$\sum_{v \in V} \deg(v) = 2|E|$$

Complexity

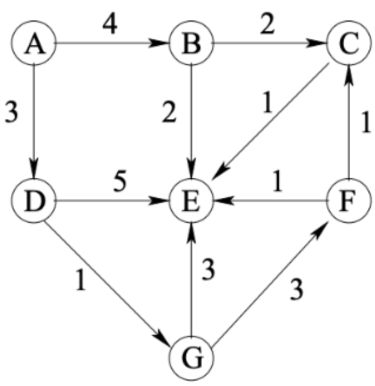
	Space	Check Edge	List Neighbors	List All Edges
Adjacency List	$\Theta(E+V)$	$O(degree(u))$	$\Theta(degree(u))$	$\Theta(V+E)$
Adjacency Matrix	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V)$	$\Theta(V^2)$

BFS vs DFS

- DFS is usually for finding relationship among vertices.
- BFS Is usually for finding shortest path from a given source.

Djikstra

Execute Djikstra's algorithm on the graph below starting at A. If there are ties, the vertex with the lowest letter comes first.

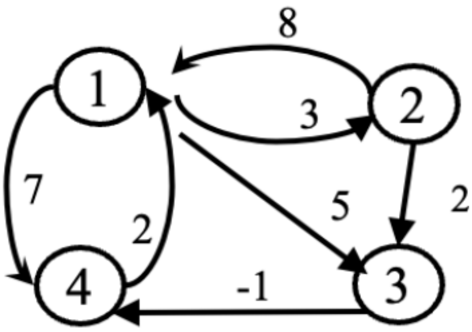


A: 0	D: 3	B: 4	G: 4	C: 6	E: 6
B: ∞	B: 4	G: 4	C: 6	E: 6	F: 7
C: ∞	C: ∞	E: 8	E:6	F: 7	
D: ∞	E: ∞	C: ∞	F: ∞		
E: ∞	F: ∞	F: ∞			
F: ∞	G: ∞				
G: ∞					

Floyd-Warshall

Let $d_{ij}^{(k)}$ be the weight of the shortest path from i to j with all intermediate vertices in the set $\{1, 2, \dots, k\}$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k > 0 \end{cases}$$



$D^0 =$

	1	2	3	4
1	0	3	5	7
2	8	0	2	∞
3	∞	∞	0	-1
4	2	∞	∞	0

$P =$

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

$D^1 =$

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	∞	∞	0	-1
4	2	5	7	0

$P =$

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

$D^2 =$

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	∞	∞	0	-1
4	2	5	7	0

$P =$

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

$D^3 =$

	1	2	3	4
1	0	3	5	4
2	8	0	2	1
3	∞	∞	0	-1
4	2	5	7	0

$P =$

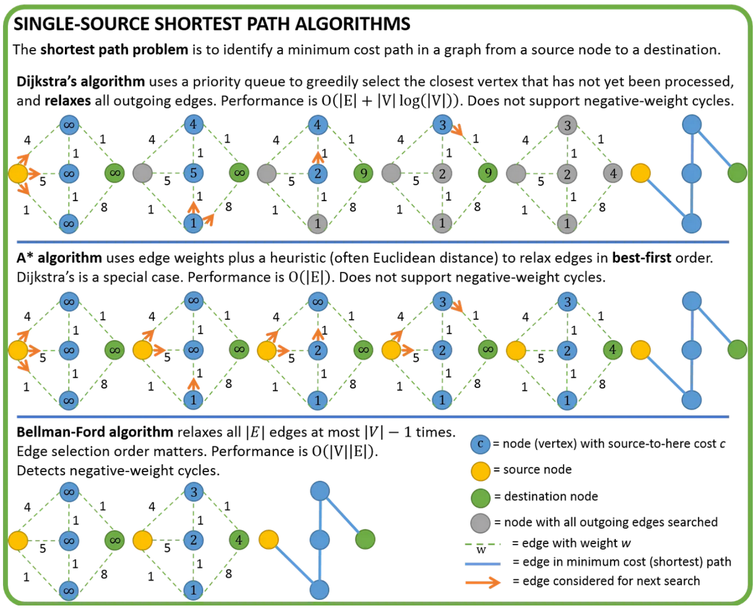
	1	2	3	4
1	0	0	0	3
2	0	0	0	3
3	0	0	0	0
4	0	1	1	0

$D^4 =$

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

$P =$

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

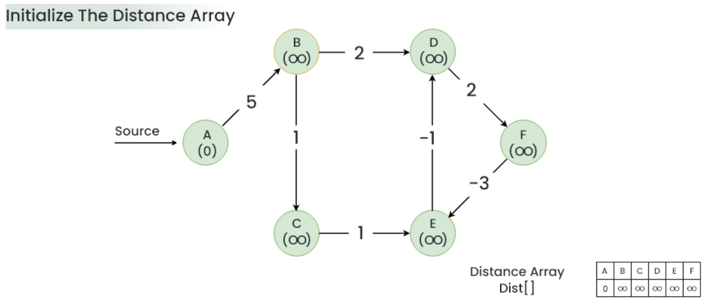


Knapsack

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w_i > w \\ \max(c[i-1, w], c[i-1, w-w_i] + v_i) & \text{otherwise} \end{cases}$$

Bellman-Ford

Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both weighted and unweighted graphs.



Bellman-Ford To Detect A Negative Cycle In A Graph

Base Case $\ell_{ss}^{(0)} = 0; \ell_{sv}^{(0)} = \infty$

Recurrence

$$\ell_{sv}^{(k)} = \min_{u \in V} \left\{ \ell_{su}^{(k-1)} + w_{uv} \right\}$$

Optimal

$$\ell_{sv}^{|V|-1} \text{ for all } v \in V$$

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
0	5	∞	∞	∞	∞
0	5	6	7	∞	∞
0	5	6	7	7	9
0	5	6	6	6	9
0	5	6	5	6	8

$$w_ij = \begin{cases} 0, & i = j \\ w(i,j), & i \neq j \text{ and } (i,j) \in E \\ \infty, & i \neq j \text{ and } (i,j) \notin E \end{cases}$$

Relaxation 1:

$$Dist(B) > Dist(A) + w(A, B)$$
$$\infty > 0 + 5 \Rightarrow Dist(B) = 5$$

Relaxation 2:

$$Dist(D) > Dist(B) + w(B, D)$$
$$\infty > 5 + 2 \Rightarrow Dist(D) = 7$$

$$Dist(C) > Dist(B) + w(B, C)$$
$$\infty > 5 + 1 \Rightarrow Dist(C) = 6$$

Relaxation 3:

$$Dist(F) > Dist(D) + w(D, F)$$
$$\infty > 7 + 2 \Rightarrow Dist(F) = 9$$

$$Dist(E) > Dist(C) + w(C, E)$$
$$\infty > 6 + 1 \Rightarrow Dist(E) = 7$$

Relaxation 4:

$$Dist(D) > Dist(E) + w(E, D)$$
$$7 > 7 + (-1) \Rightarrow Dist(D) = 6$$

$$Dist(E) > Dist(F) + w(F, E)$$
$$7 > 9 + (-3) \Rightarrow Dist(E) = 6$$

Relaxation 5:

$$Dist(F) > Dist(D) + w(D, F)$$
$$9 > 6 + 2 \Rightarrow Dist(F) = 8$$

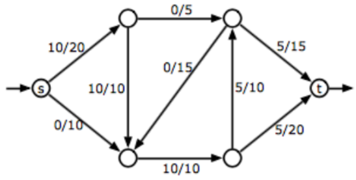
$$Dist(D) > Dist(E) + w(E, D)$$
$$6 > 6 + (-1) \Rightarrow Dist(D) = 6$$

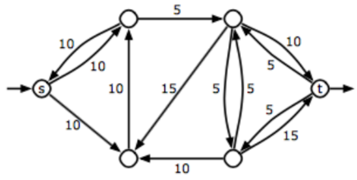
Relaxation 6:

$$Dist(E) > Dist(F) + w(F, E)$$
$$6 > 8 + (-3) \Rightarrow Dist(E) = 5$$

$$Dist(F) > Dist(D) + w(D, F)$$
$$8 > 5 + 2 \Rightarrow Dist(F) = 7$$

Network Flow





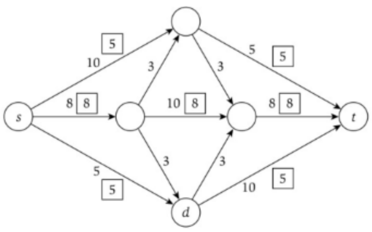
Flow Amount of material that can be transported between two nodes.

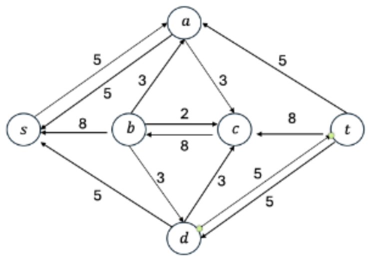
Capacity Maximum amount of material that can be transported between two nodes.

Residual Capacity Amount of material that can still be transported between two nodes.

Augmenting Path Path from source to sink where all edges have residual capacity.

Residual Graph Graph where edges have residual capacity.





Dyanmic Programming

1. Characterize structure of an optimal solution.
2. Recursively define value of an optimal solution.
3. Compute the value in a bottom-up fashion.
4. Construct an optimal solution from computed information.

1D Example

Let D_n be the number of ways to write n as the sum of 1, 3, and 4. Find the recurrence:

1. Consider one possible solution, $n = x_1 + x_2 + \cdots + x_m$
2. If $x_m = 1$, rest of the terms must sum to $n-1$
3. Thus, number of sums that end with $x_m = 1$ is D_{n-1}
4. Take other cases into account ($x_m = 3, x_m = 4$)

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

Rod Cutting

- Has optimal sub-structure property (must have optimal cut for each sub-problem to get global optimal)
- Has recursive exponential solution
- Has polynomial DP solution

Greedy Algorithm

Optimal Substructure The optimal solution to a problem incorporates the optimal solution to subproblems.

Greedy Choice Property Locally optimal choice leads to globally optimal solution.

Overlapping Subproblems Subproblems recur many times.

DP

- Used to solve optimization
- Has **optimal substructure**
- Make an 'informed choice' after getting optimal solutions to subproblems
- Bottom-up
- Dependent on overlapping subproblems

Greedy

- Used to solve optimization
- Has **optimal substructure**
- Make a 'greedy choice' before solving subproblem
- Top-down
  - Each round selects only one subproblem
  - Subproblem size decreases
- No overlapping subproblems

DFS Applications

- Finding connected components on an undirected graph
- Detecting cycles on a graph
- Topological sorting on a directed acyclic graph (DAG)
- Finding strongly connected components (SCC) in a directed graph

Lemma: A directed graph is acylic iff a DFS of the graph yields no back edges.

Strongly Connected Components

The SCC of a directed graph are the *equivalence classes of verticies* under the 'mutually reachable' relation. That is, a SCC is a maximal subset of mutually reachable nodes.

Spanning Tree

Spanning tree of an undirected graph is a tree that connects all verticies.

- Exactly $|V| - 1$ edges
- Acyclic
- Non-unique

Minimum Spanning Tree

Kruskal Consider edges in ascending order of weight. Each step, select next edge as long as it doesn't make a cycle.

Prim Start with any vertex and grow a tree from it. At each step, add edge of the least weight to connect an isolated vertex.

MST is unique if all edge weights are distinct.

String Matching

Rabin-Karp

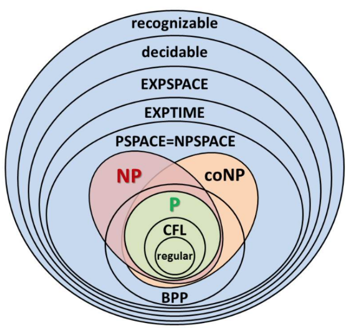
Compare a string's hash values.

Knuth-Morris-Pratt

- Linear time for exact matching
- Compares left to right, shifts more than one position
- Preprocessing approach to avoid trivial comparisons
- Conceived by Donald Knuth and Vaughan Pratt
- Worst-Case $\Theta((n - m + 1)m)$

`ABABCABCABAABABD` LPS=[0, 0, 1, 2, 0]

Complexity Theory



NP Verifiable in polynomial time

P Solvable in polynomial time

NP-Hard At least as hard has all NP problems

NP-Complete Both NP and NP-Hard

Psuedocodes

```
Optimal-BST(p, q, n)
    let e[1..n + 1, 0..n], w[1..n + 1, 0..n], and root[1..n, 1..n] be new tables
    for i = 1 to n + 1
        e[i, i - 1] = 0
        w[i, i - 1] = 0
    for l = 1 to n
        for i = 1 to n - l + 1
            j = i + l - 1
            e[i, j] = infty
            w[i, j] = w[i, j - 1] + p[j]
            for r = i to j
                t = e[i, r - 1] + e[r + 1, j] + w[i, j]
                if t < e[i, j]
                    e[i, j] = t
                    root[i, j] = r
    return e and root
```

```
floydWarshall(int dist[][])
for k=0 to V
    for i=0 to V
        for j=0 to V
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j]
```

```
LCS-Length(X, Y, m, n)
    let b[1..m, 1..n] and c[0..m, 0.. n] be new tables
    for i = 0 to m
        c[i, 0] = 0
    for j = 0 to n
        c[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x[i] == y[j]
                c[i, j] = c[i - 1, j - 1] + 1
                b[i, j] = "NW"
            else if c[i - 1, j] >= c[i, j - 1]
                c[i, j] = c[i - 1, j]
                b[i, j] = "N"
            else
                c[i,j] = c[i, j - 1]
                b[i, j] = "W"
    return c and b
```

```
DFS(G)
    for each vertex u in G.V
        u.color = WHITE
        u.pi = NIL
    time = 0
    for each vertex u in G.V
        if u.color == WHITE
            DFS-Visit(G, u)

DFS-Visit(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v in G.Adj[u]
        if v.color == WHITE
            v.pi = u
            DFS-Visit(G, v)
    u.color = BLACK
    time = time + 1
    u.f = time
```

```
KMP-Matcher(T, P)
    n = T.length
    m = P.length
    pi = Compute-Prefix-Function(P)
    q = 0
    for i = 1 to n
        while q > 0 and P[q + 1] != T[i]
            q = pi[q]
        if P[q + 1] == T[i]
            q = q + 1
        if q == m
            print "Pattern occurs with shift" i - m
            q = pi[q]
```

```
Compute-Prefix-Function(P)
    n = P.length
    let pi[0..n] be a new table
    for i = 0 to n
        j = pi[i - 1]
        while j > 0 and P[i] != P[j]
            j = pi[j - 1]
        if P[i] == P[j]
            j = j + 1
        pi[i] = j
    return pi
```

```
def knapsack(W, wt, val, n):
dp = [[0] * (W+1) for _ in range(n+1)]

for i in range(1, n+1):
    for w in range(1, W+1):
        if wt[i-1] <= w:
            dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]],
                dp[i-1][w])
        else:
            dp[i][w] = dp[i-1][w]

return dp[n][W]
```

```
def edit_distance(s1, s2):
m, n = len(s1), len(s2)
dp = [[0] * (n+1) for _ in range(m+1)]

for i in range(m+1):
    for j in range(n+1):
        if i == 0:
            dp[i][j] = j
        elif j == 0:
            dp[i][j] = i
        elif s1[i-1] == s2[j-1]:
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = 1 + min(dp[i-1][j],
                dp[i][j-1], dp[i-1][j-1])

return dp[m][n]
```

```
def coin_change(coins, amount):
dp = [float('inf')] * (amount+1)
dp[0] = 0

for i in range(1, amount+1):
    for coin in coins:
        if coin <= i:
            dp[i] = min(dp[i], dp[i-coin] + 1)

return dp[amount] if dp[amount] != float('inf') else -1
```

```
def tsp(graph, start):
n = len(graph)
visited = (1 << n) - 1
memo = {}

def dfs(node, visited):
    if visited == 0:
        return graph[node][start]

    if (node, visited) in memo:
        return memo[(node, visited)]

    ans = float('inf')
    for i in range(n):
        if visited & (1 << i):
            ans = min(ans, graph[node][i] +
                dfs(i, visited ^ (1 << i)))

    memo[(node, visited)] = ans
    return ans

return dfs(start, visited)
```