More Dynamic Programming

Lecture 10
February 21, 2013
Part I

All Pairs Shortest Paths
Shortest Path Problems

Input  A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.
3. Find shortest paths for all pairs of nodes.
Single-Source Shortest Path Problems

**Input**
A (undirected or directed) graph \( G = (V, E) \) with edge lengths. For edge \( e = (u, v) \), \( \ell(e) = \ell(u, v) \) is its length.

1. Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
2. Given node \( s \) find shortest path from \( s \) to all other nodes.

Dijkstra’s algorithm for non-negative edge lengths. Running time: \( O((m + n) \log n) \) with heaps and \( O(m + n \log n) \) with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: \( O(nm) \).
Single-Source Shortest Path Problems

**Input**  A (undirected or directed) graph \( G = (V, E) \) with edge lengths. For edge \( e = (u, v) \), \( \ell(e) = \ell(u, v) \) is its length.

1. Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
2. Given node \( s \) find shortest path from \( s \) to all other nodes.

**Dijkstra’s algorithm** for non-negative edge lengths. Running time: \( O((m + n) \log n) \) with heaps and \( O(m + n \log n) \) with advanced priority queues.

**Bellman-Ford algorithm** for arbitrary edge lengths. Running time: \( O(nm) \).
All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and
   $O(nm + n^2 \log n)$ using advanced priority queues.

2. Arbitrary edge lengths: $O(n^2m)$.
   $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?
All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.

2. Arbitrary edge lengths: $O(n^2m)$. $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?
### All-Pairs Shortest Path Problem

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

**Apply single-source algorithms** $n$ times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.

2. Arbitrary edge lengths: $O(n^2m)$. $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?
Shortest Paths and Recursion

1. Compute the shortest path distance from $s$ to $t$ recursively?
2. What are the smaller sub-problems?

**Lemma**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges
Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

**Lemma**

Let G be a directed graph with arbitrary edge lengths. If s = v_0 → v_1 → v_2 → ... → v_k is a shortest path from s to v_k then for 1 ≤ i < k:

1. s = v_0 → v_1 → v_2 → ... → v_i is a shortest path from s to v_i

Sub-problem idea: paths of fewer hops/edges
1. Compute the shortest path distance from $s$ to $t$ recursively?

2. What are the smaller sub-problems?

**Lemma**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges
Hop-based Recur’: Single-Source Shortest Paths

Single-source problem: fix source $s$.

**$\text{OPT}(v, k)$**: shortest path dist. from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = \text{OPT}(v, n - 1)$. Recursion for $\text{OPT}(v, k)$:

$$\text{OPT}(v, k) = \min \left\{ \min_{u \in V} (\text{OPT}(u, k - 1) + c(u, v)), \text{OPT}(v, k - 1) \right\}$$

Base case: $\text{OPT}(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$

Leads to Bellman-Ford algorithm — see textbook.

$\text{OPT}(v, k)$ values are also of independent interest: shortest paths with at most $k$ hops
Hop-based Recur’: Single-Source Shortest Paths

**Single-source problem:** fix source $s$.

**OPT($v$, $k$):** shortest path dist. from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = \text{OPT}(v, n - 1)$. Recursion for $\text{OPT}(v, k)$:

$$\text{OPT}(v, k) = \min \begin{cases} 
\min_{u \in V} (\text{OPT}(u, k - 1) + c(u, v)) \\
\text{OPT}(v, k - 1) 
\end{cases}$$

**Base case:** $\text{OPT}(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$

Leads to Bellman-Ford algorithm — see text book.

$\text{OPT}(v, k)$ values are also of independent interest: shortest paths with at most $k$ hops.
Hop-based Recur’: Single-Source Shortest Paths

Single-source problem: fix source \( s \).

\( \text{OPT}(v, k) \): shortest path dist. from \( s \) to \( v \) using at most \( k \) edges.
Note: \( \text{dist}(s, v) = \text{OPT}(v, n - 1) \). Recursion for \( \text{OPT}(v, k) \):

\[
\text{OPT}(v, k) = \min \left\{ \min_{u \in V} (\text{OPT}(u, k - 1) + c(u, v)), \text{OPT}(v, k - 1) \right\}
\]

Base case: \( \text{OPT}(v, 1) = c(s, v) \) if \((s, v) \in E\) otherwise \( \infty \)
Leads to Bellman-Ford algorithm — see text book.

\( \text{OPT}(v, k) \) values are also of independent interest: shortest paths with at most \( k \) hops.
Hop-based Recur’: Single-Source Shortest Paths

Single-source problem: fix source \( s \).

\( \text{OPT}(v, k) \): shortest path dist. from \( s \) to \( v \) using at most \( k \) edges.

Note: \( \text{dist}(s, v) = \text{OPT}(v, n - 1) \). Recursion for \( \text{OPT}(v, k) \):

\[
\text{OPT}(v, k) = \min \left\{ \min_{u \in V} (\text{OPT}(u, k - 1) + c(u, v)), \text{OPT}(v, k - 1) \right\}
\]

Base case: \( \text{OPT}(v, 1) = c(s, v) \) if \( (s, v) \in E \) otherwise \( \infty \)

Leads to Bellman-Ford algorithm — see text book.

\( \text{OPT}(v, k) \) values are also of independent interest: shortest paths with at most \( k \) hops.
Hop-based Recur’: Single-Source Shortest Paths

Single-source problem: fix source $s$.

$OPT(v, k)$: shortest path dist. from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = OPT(v, n - 1)$. Recursion for $OPT(v, k)$:

$$OPT(v, k) = \min \begin{cases} \min_{u \in V}(OPT(u, k - 1) + c(u, v)) \quad \text{if } (s, v) \in E \text{ otherwise } \infty \\ OPT(v, k - 1) \end{cases}$$

Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$

Leads to Bellman-Ford algorithm — see text book.

$OPT(v, k)$ values are also of independent interest: shortest paths with at most $k$ hops
Hop-based Recur’: Single-Source Shortest Paths

Single-source problem: fix source $s$.

$\text{OPT}(v, k)$: shortest path dist. from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = \text{OPT}(v, n - 1)$. Recursion for $\text{OPT}(v, k)$:

$$\text{OPT}(v, k) = \min \begin{cases} 
\min_{u \in V} (\text{OPT}(u, k - 1) + c(u, v)), \\
\text{OPT}(v, k - 1) 
\end{cases}$$

Base case: $\text{OPT}(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$

Leads to Bellman-Ford algorithm — see text book.

$\text{OPT}(v, k)$ values are also of independent interest: shortest paths with at most $k$ hops
Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

**dist**(i, j, k): shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an *intermediate node* is at most $k$

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
\]
Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

$\text{dist}(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an intermediate node is at most $k$

$\text{dist}(i, j, 0) = 100$
$\text{dist}(i, j, 1) = 9$
$\text{dist}(i, j, 2) = 8$
$\text{dist}(i, j, 3) = 5$
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

2. $\text{dist}(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an *intermediate node* is at most $k$

```
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
```
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

2. $\text{dist}(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an \textit{intermediate node} is at most $k$

Distances:

- $\text{dist}(i, j, 0) = 100$
- $\text{dist}(i, j, 1) = 9$
- $\text{dist}(i, j, 2) = 8$
- $\text{dist}(i, j, 3) = 5$
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

2. $\text{dist}(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an *intermediate node* is at most $k$

![Graph with labeled edges and distances]

- $\text{dist}(i, j, 0) = 100$
- $\text{dist}(i, j, 1) = 9$
- $\text{dist}(i, j, 2) = 8$
- $\text{dist}(i, j, 3) = 5$
All-Pairs: Recursion on index of intermediate nodes

\[ \text{dist}(i, k, k - 1) \quad \text{dist}(k, j, k - 1) \]

\[ \text{dist}(i, j, k - 1) \]

\[
\text{dist}(i, j, k) = \min \left\{ \begin{array}{l}
\text{dist}(i, j, k - 1) \\
\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1)
\end{array} \right. 
\]

Base case: \( \text{dist}(i, j, 0) = c(i, j) \) if \( (i, j) \in E \), otherwise \( \infty \)

Correctness: If \( i \rightarrow j \) shortest path goes through \( k \) then \( k \) occurs only once on the path — otherwise there is a negative length cycle.
Floyd-Warshall Algorithm for All-Pairs Shortest Paths

Check if $G$ has a negative cycle // Bellman-Ford: $O(mn)$ time
if there is a negative cycle then return ‘‘Negative cycle’’

for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $\text{dist}(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, 0 if $i = j$ *)

for $k = 1$ to $n$ do
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            $\text{dist}(i, j, k) = \min \left\{ \begin{array}{l}
            \text{dist}(i, j, k - 1), \\
            \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1)
            \end{array} \right.$

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).
Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$. 
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

Check if $G$ has a negative cycle // Bellman-Ford: $O(mn)$ time
if there is a negative cycle then return ‘‘Negative cycle’’

for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $\text{dist}(i, j, 0) = c(i, j)$ (* $c(i, j) = \infty$ if $(i, j) \notin E$, $0$ if $i = j$ *)

for $k = 1$ to $n$ do
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            $\text{dist}(i, j, k) = \min \left\{ \text{dist}(i, j, k - 1), \right.$
            $\left. \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1) \right\}$

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).
Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$. 
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

Check if \( G \) has a negative cycle // Bellman-Ford: \( O(mn) \) time
if there is a negative cycle then return ‘‘Negative cycle’’

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    \( \text{dist}(i, j, 0) = c(i, j) \) (* \( c(i, j) = \infty \) if \( (i, j) \notin E \), \( 0 \) if \( i = j \) *)

for \( k = 1 \) to \( n \) do
  for \( i = 1 \) to \( n \) do
    for \( j = 1 \) to \( n \) do
      \( \text{dist}(i, j, k) = \min \left\{ \text{dist}(i, j, k - 1), \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1) \right\} \)

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).
Running Time: \( \Theta(n^3) \), Space: \( \Theta(n^3) \).
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

\[
\begin{align*}
\text{for } & i = 1 \text{ to } n \text{ do} \\
& \quad \text{for } j = 1 \text{ to } n \text{ do} \\
& \quad \quad \text{dist}(i, j, 0) = c(i, j) \quad (\ast \ c(i, j) = \infty \text{ if } (i, j) \notin E, \ 0 \text{ if } i = j \ \ast) \\
& \text{not edge, } 0 \text{ if } i = j \ \ast)
\end{align*}
\]

\[
\begin{align*}
\text{for } & k = 1 \text{ to } n \text{ do} \\
& \quad \text{for } i = 1 \text{ to } n \text{ do} \\
& \quad \quad \text{for } j = 1 \text{ to } n \text{ do} \\
& \quad \quad \quad \text{dist}(i, j, k) = \min(\text{dist}(i, j, k - 1), \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1))
\end{align*}
\]

\[
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do} \\
& \quad \text{if } (\text{dist}(i, i, n) < 0) \text{ then} \\
& \quad \quad \text{Output that there is a negative length cycle in } G
\end{align*}
\]
Do we need a separate algorithm to check if there is negative cycle?

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n \text{ do} \\
& \quad \text{for } j = 1 \text{ to } n \text{ do} \\
& \quad \quad \text{dist}(i, j, 0) = c(i, j) \quad (* c(i, j) = \infty \text{ if } (i, j) \notin E, 0 \text{ if } i = j *) \\
& \quad \text{not edge, 0 if } i = j *) \\
\text{for } k &= 1 \text{ to } n \text{ do} \\
& \quad \text{for } i = 1 \text{ to } n \text{ do} \\
& \quad \quad \text{for } j = 1 \text{ to } n \text{ do} \\
& \quad \quad \quad \text{dist}(i, j, k) = \min(\text{dist}(i, j, k - 1), \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1)) \\
\text{for } i &= 1 \text{ to } n \text{ do} \\
& \quad \text{if } (\text{dist}(i, i, n) < 0) \text{ then} \\
& \quad \quad \text{Output that there is a negative length cycle in } G
\end{align*}
\]

Correctness: exercise
Question:  Can we find the paths in addition to the distances?

1. Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices.

2. With array Next, for any pair of given vertices $i, j$ can compute a shortest path in $O(n)$ time.
Floyd-Warshall Algorithm: Finding the Paths

**Question:** Can we find the paths in addition to the distances?

1. Create a $n \times n$ array $\text{Next}$ that stores the next vertex on shortest path for each pair of vertices.

2. With array $\text{Next}$, for any pair of given vertices $i, j$ can compute a shortest path in $O(n)$ time.
Floyd-Warshall Algorithm

Finding the Paths

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    \( \text{dist}(i, j, 0) = c(i, j) \) (* \( c(i, j) = \infty \) if \((i, j)\) not edge, 0 if \( i = j \) *)
    \( \text{Next}(i, j) = -1 \)
  for \( k = 1 \) to \( n \) do
    for \( i = 1 \) to \( n \) do
      for \( j = 1 \) to \( n \) do
        if (\( \text{dist}(i, j, k - 1) > \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1) \)) then
          \( \text{dist}(i, j, k) = \text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1) \)
          \( \text{Next}(i, j) = k \)
    for \( i = 1 \) to \( n \) do
      if (\( \text{dist}(i, i, n) < 0 \)) then
        Output that there is a negative length cycle in \( G \)

Exercise: Given Next array and any two vertices \( i, j \) describe an \( O(n) \) algorithm to find a \( i-j \) shortest path.
### Summary of results on shortest paths

<table>
<thead>
<tr>
<th>Single vertex</th>
<th>No negative edges</th>
<th>Dijkstra</th>
<th>$O(n \log n + m)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Edges cost might be negative</td>
<td>Bellman Ford</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td></td>
<td>But no negative cycles</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### All Pairs Shortest Paths

<table>
<thead>
<tr>
<th>No negative edges</th>
<th>$n \times$ Dijkstra</th>
<th>$O(n^2 \log n + nm)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative cycles</td>
<td>$n \times$ Bellman Ford</td>
<td>$O(n^2m) = O(n^4)$</td>
</tr>
<tr>
<td>No negative cycles</td>
<td>Floyd-Warshall</td>
<td>$O(n^3)$</td>
</tr>
</tbody>
</table>
Part II

Knapsack
Knapsack Problem

Input  Given a Knapsack of capacity $W$ lbs. and $n$ objects with $i$th object having weight $w_i$ and value $v_i$; assume $W, w_i, v_i$ are all positive integers

Goal  Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.
Knapsack Problem

**Input**  Given a Knapsack of capacity $W$ lbs. and $n$ objects with $i$th object having weight $w_i$ and value $v_i$; assume $W, w_i, v_i$ are all positive integers

**Goal**  Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.
Knapsack Example

Example

<table>
<thead>
<tr>
<th>Item</th>
<th>$I_1$</th>
<th>$I_2$</th>
<th>$I_3$</th>
<th>$I_4$</th>
<th>$I_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

If $W = 11$, the best is $\{I_3, I_4\}$ giving value 40.

Special Case

When $v_i = w_i$, the Knapsack problem is called the Subset Sum Problem.
Greedy Approach

1. Pick objects with greatest value
   - Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$

2. Pick objects with smallest weight
   - Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick $\{1\}$, but the optimal is $\{2\}$

3. Pick objects with largest $v_i/w_i$ ratio
   - Let $W = 4$, $w_1 = w_2 = 2$, $w_3 = 3$, $v_1 = v_2 = 3$ and $v_3 = 5$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$
   - Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to $W$. 

Sariel, Alexandra  (UIUC)  
CS473  
18  
Spring 2013  
18 / 34
Towards a Recursive Solution

First guess: $\text{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$.

Observation

Consider an optimal solution $O$ for $1, \ldots, i$

Case item $i \notin O$, $O$ is an optimal solution to items $1$ to $i - 1$

Case item $i \in O$, then $O - \{i\}$ is an optimum solution for items $1$ to $n - 1$ in knapsack of capacity $W - w_i$.

Subproblems depend also on remaining capacity. Cannot write subproblem only in terms of $\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.

$\text{Opt}(i, w)$: optimum profit for items $1$ to $i$ in knapsack of size $w$

Goal: compute $\text{Opt}(n, W)$
Towards a Recursive Solution

First guess: $\text{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$.

Observation

Consider an optimal solution $\mathcal{O}$ for $1, \ldots, i$

Case item $i \notin \mathcal{O}$ \hspace{1em} $\mathcal{O}$ is an optimal solution to items $1$ to $i - 1$

Case item $i \in \mathcal{O}$ \hspace{1em} Then $\mathcal{O} - \{i\}$ is an optimum solution for items $1$
to $n - 1$ in knapsack of capacity $W - w_i$.

Subproblems depend also on remaining capacity. Cannot write subproblem only in terms of
$\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.

$\text{Opt}(i, w)$: optimum profit for items $1$ to $i$ in knapsack of size $w$

Goal: compute $\text{Opt}(n, W)$
First guess: $\text{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$. 

**Observation**

Consider an optimal solution $O$ for $1, \ldots, i$

- Case item $i \not\in O$: $O$ is an optimal solution to items $1$ to $i - 1$
- Case item $i \in O$: Then $O - \{i\}$ is an optimum solution for items $1$ to $n - 1$ in knapsack of capacity $W - w_i$.

Subproblems depend also on remaining capacity. Cannot write subproblem only in terms of $\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.

$\text{Opt}(i, w)$: optimum profit for items $1$ to $i$ in knapsack of size $w$

**Goal**: compute $\text{Opt}(n, W)$
Dynamic Programming Solution

**Definition**

Let $\text{Opt}(i, w)$ be the optimal way of picking items from 1 to $i$, with total weight not exceeding $w$.

$$\text{Opt}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{Opt}(i - 1, w) & \text{if } w_i > w \\
\max \begin{cases} 
\text{Opt}(i - 1, w) \\
\text{Opt}(i - 1, w - w_i) + v_i 
\end{cases} & \text{otherwise}
\end{cases}$$
An Iterative Algorithm

for \( w = 0 \) to \( W \) do
  \( M[0, w] = 0 \)

for \( i = 1 \) to \( n \) do
  for \( w = 1 \) to \( W \) do
    if \( (w_i > w) \) then
      \( M[i, w] = M[i - 1, w] \)
    else
      \( M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i) \)

Running Time

1. Time taken is \( O(nW) \)
2. Input has size \( O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)) \); so running time not polynomial but “pseudo-polynomial”!
An Iterative Algorithm

for \( w = 0 \) to \( W \) do
\[ M[0, w] = 0 \]
for \( i = 1 \) to \( n \) do
  for \( w = 1 \) to \( W \) do
    if \( w_i > w \) then
      \[ M[i, w] = M[i - 1, w] \]
    else
      \[ M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i) \]

Running Time

1. Time taken is \( O(nW) \)
2. Input has size \( O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)) \); so running time not polynomial but “pseudo-polynomial”!
An Iterative Algorithm

for \( w = 0 \) to \( W \) do
    \( M[0, w] = 0 \)
for \( i = 1 \) to \( n \) do
    for \( w = 1 \) to \( W \) do
        if \( (w_i > w) \) then
            \( M[i, w] = M[i - 1, w] \)
        else
            \( M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i) \)

Running Time

1. Time taken is \( O(nW) \)
2. Input has size \( O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)) \); so running time not polynomial but “pseudo-polynomial”!
Knapsack Algorithm and Polynomial time

1. **Input size for Knapsack:**
   \[ O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i). \]

2. **Running time of dynamic programming algorithm:** \( O(nW) \).

3. **Not a polynomial time algorithm.**

4. **Example:** \( W = 2^n \) and \( w_i, v_i \in [1..2^n] \). Input size is \( O(n^2) \), running time is \( O(n2^n) \) arithmetic/comparisons.

5. **Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if numbers in input are of size polynomial in the **combinatorial size** of problem.

6. **Knapsack is **NP-Hard** if numbers are not polynomial in \( n \).**
Knapsack Algorithm and Polynomial time

1. Input size for Knapsack:
   \[ O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i). \]

2. Running time of dynamic programming algorithm: \( O(nW) \).

3. Not a polynomial time algorithm.

4. Example: \( W = 2^n \) and \( w_i, v_i \in [1..2^n] \). Input size is \( O(n^2) \), running time is \( O(n2^n) \) arithmetic/comparisons.

5. Algorithm is called a pseudo-polynomial time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the combinatorial size of problem.

6. Knapsack is NP-Hard if numbers are not polynomial in \( n \).
Knapsack Algorithm and Polynomial time

1. Input size for Knapsack:
\[ O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i). \]

2. Running time of dynamic programming algorithm: \( O(nW) \).

3. Not a polynomial time algorithm.

4. Example: \( W = 2^n \) and \( w_i, v_i \in [1..2^n] \). Input size is \( O(n^2) \), running time is \( O(n2^n) \) arithmetic/comparisons.

5. Algorithm is called a pseudo-polynomial time algorithm because running time is polynomial if numbers in input are of size polynomial in the combinatorial size of problem.

6. Knapsack is NP-Hard if numbers are not polynomial in \( n \).
Knapsack Algorithm and Polynomial time

1. Input size for Knapsack:
   \[ O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i). \]
2. Running time of dynamic programming algorithm: \( O(nW) \).
3. Not a polynomial time algorithm.
4. Example: \( W = 2^n \) and \( w_i, v_i \in [1..2^n] \). Input size is \( O(n^2) \), running time is \( O(n2^n) \) arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if numbers in input are of size polynomial in the combinatorial size of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in \( n \).
Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i)$.

2. Running time of dynamic programming algorithm: $O(nW)$.

3. Not a polynomial time algorithm.

4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.

5. Algorithm is called a pseudo-polynomial time algorithm because running time is polynomial if numbers in input are of size polynomial in the combinatorial size of problem.

6. Knapsack is NP-Hard if numbers are not polynomial in $n$. 
Knapsack Algorithm and Polynomial time

1. Input size for Knapsack:
\[ O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i). \]

2. Running time of dynamic programming algorithm: \( O(nW) \).

3. Not a polynomial time algorithm.

4. Example: \( W = 2^n \) and \( w_i, v_i \in [1..2^n] \). Input size is \( O(n^2) \), running time is \( O(n2^n) \) arithmetic/comparisons.

5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.

6. Knapsack is **NP-Hard** if numbers are not polynomial in \( n \).
Knapsack Algorithm and Polynomial time

1. Input size for Knapsack:
   \[ O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i) \].

2. Running time of dynamic programming algorithm: \( O(nW) \).

3. Not a polynomial time algorithm.

4. Example: \( W = 2^n \) and \( w_i, v_i \in [1..2^n] \). Input size is \( O(n^2) \), running time is \( O(n2^n) \) arithmetic/comparisons.

5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the combinatorial size of problem.

6. Knapsack is **NP-Hard** if numbers are not polynomial in \( n \).
Part III

Traveling Salesman Problem
Traveling Salesman Problem

**Input** A graph $G = (V, E)$ with non-negative edge costs/lengths. $c(e)$ for edge $e$

**Goal** Find a tour of minimum cost that visits each node.

No polynomial time algorithm known. Problem is NP-Hard.
Traveling Salesman Problem

Input  A graph $G = (V, E)$ with non-negative edge costs/lengths. $c(e)$ for edge $e$

Goal  Find a tour of minimum cost that visits each node.

No polynomial time algorithm known. Problem is \textbf{NP-Hard}. 
Drawings using TSP
Drawings using TSP
Example: optimal tour for cities of a country (which one?)
An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling’s formula: $n! \sim \sqrt{n} \left(\frac{n}{e}\right)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?
An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling’s formula: $n! \sim \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?
An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling’s formula: $n! \simeq \sqrt{n} (n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?
An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling’s formula: $n! \sim \sqrt{n} (n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?
An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling’s formula: $n! \sim \sqrt{n} (n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?
Order vertices as $v_1, v_2, \ldots, v_n$.

**OPT(S)**: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to $S$. Want $\text{OPT}(V)$.

Can we compute $\text{OPT}(S)$ recursively?

1. Say $v \in S$. What are the two neighbors of $v$ in optimum tour in $S$?

2. If $u, w$ are neighbors of $v$ in an optimum tour of $S$ then removing $v$ gives an optimum *path* from $u$ to $w$ visiting all nodes in $S - \{v\}$.

Path from $u$ to $w$ is not a recursive subproblem! Need to find a more general problem to allow recursion.
Towards a Recursive Solution

1. Order vertices as $v_1, v_2, \ldots, v_n$

2. $OPT(S)$: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to $S$. Want $OPT(V)$.

Can we compute $OPT(S)$ recursively?

1. Say $v \in S$. What are the two neighbors of $v$ in optimum tour in $S$?

2. If $u, w$ are neighbors of $v$ in an optimum tour of $S$ then removing $v$ gives an optimum path from $u$ to $w$ visiting all nodes in $S - \{v\}$.

Path from $u$ to $w$ is not a recursive subproblem! Need to find a more general problem to allow recursion.
Towards a Recursive Solution

1. Order vertices as $v_1, v_2, \ldots, v_n$

2. $\text{OPT}(S)$: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to $S$. Want $\text{OPT}(V)$.

Can we compute $\text{OPT}(S)$ recursively?

1. Say $v \in S$. What are the two neighbors of $v$ in optimum tour in $S$?

2. If $u, w$ are neighbors of $v$ in an optimum tour of $S$ then removing $v$ gives an optimum path from $u$ to $w$ visiting all nodes in $S - \{v\}$.

Path from $u$ to $w$ is not a recursive subproblem! Need to find a more general problem to allow recursion.
A More General Problem: TSP Path

**Input** A graph $G = (V, E)$ with non-negative edge costs/lengths ($c(e)$ for edge $e$) and two nodes $s, t$

**Goal** Find a path from $s$ to $t$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

1. $OPT(u, v, S)$: optimum TSP Path from $u$ to $v$ in the graph restricted to $S$ (here $u, v \in S$).
A More General Problem: TSP Path

Input A graph $G = (V, E)$ with non-negative edge costs/lengths ($c(e)$ for edge $e$) and two nodes $s, t$

Goal Find a path from $s$ to $t$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

1. $OPT(u, v, S)$: optimum TSP Path from $u$ to $v$ in the graph restricted to $S$ (here $u, v \in S$).
A More General Problem: TSP Path

**Input** A graph \( G = (V, E) \) with non-negative edge costs/lengths \( c(e) \) for edge \( e \) and two nodes \( s, t \)

**Goal** Find a path from \( s \) to \( t \) of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

1. \( \text{OPT}(u, v, S) \): optimum TSP Path from \( u \) to \( v \) in the graph restricted to \( S \) (here \( u, v \in S \)).
What is the next node in the optimum path from $u$ to $v$? Suppose it is $w$. Then what is $\text{OPT}(u, v, S)$?

$$\text{OPT}(u, v, S) = c(u, w) + \text{OPT}(w, v, S - \{u\})$$

We do not know $w$! So try all possibilities for $w$. 

A More General Problem: TSP Path

Continued...
What is the next node in the optimum path from \( u \) to \( v \)? Suppose it is \( w \). Then what is \( \text{OPT}(u, v, S) \)?

\[
\text{OPT}(u, v, S) = c(u, w) + \text{OPT}(w, v, S - \{u\})
\]

We do not know \( w \)! So try all possibilities for \( w \).
A Recursive Solution

\[ \text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + \text{OPT}(w, v, S - \{u\}) \right) \]

What are the subproblems for the original problem \( \text{OPT}(s, t, V) \)?
\( \text{OPT}(u, v, S) \) for \( u, v \in S, S \subseteq V \).

How many subproblems?

1. number of distinct subsets \( S \) of \( V \) is at most \( 2^n \)
2. number of pairs of nodes in a set \( S \) is at most \( n^2 \)
3. hence number of subproblems is \( O(n^2 2^n) \)

Exercise: Show that one can compute TSP using above dynamic program in \( O(n^3 2^n) \) time and \( O(n^2 2^n) \) space.

Disadvantage of dynamic programming solution: memory!
A Recursive Solution

\[
\text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + \text{OPT}(w, v, S - \{u\}) \right)
\]

What are the subproblems for the original problem \( \text{OPT}(s, t, V) \)?

\( \text{OPT}(u, v, S) \) for \( u, v \in S, S \subseteq V \).

How many subproblems?

1. Number of distinct subsets \( S \) of \( V \) is at most \( 2^n \)
2. Number of pairs of nodes in a set \( S \) is at most \( n^2 \)
3. Hence number of subproblems is \( O(n^2 2^n) \)

**Exercise:** Show that one can compute TSP using above dynamic program in \( O(n^3 2^n) \) time and \( O(n^2 2^n) \) space.

Disadvantage of dynamic programming solution: memory!
A Recursive Solution

$$\text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + \text{OPT}(w, v, S - \{u\}) \right)$$

What are the subproblems for the original problem $\text{OPT}(s, t, V)$?

$\text{OPT}(u, v, S)$ for $u, v \in S, S \subseteq V$.

How many subproblems?

1. Number of distinct subsets $S$ of $V$ is at most $2^n$
2. Number of pairs of nodes in a set $S$ is at most $n^2$
3. Hence number of subproblems is $O(n^2 2^n)$

Exercise: Show that one can compute TSP using above dynamic program in $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Disadvantage of dynamic programming solution: memory!
A Recursive Solution

\[
\text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + \text{OPT}(w, v, S - \{u\}) \right)
\]

What are the subproblems for the original problem \text{OPT}(s, t, V)?
\text{OPT}(u, v, S) \text{ for } u, v \in S, S \subseteq V.

How many subproblems?

1. number of distinct subsets \( S \) of \( V \) is at most \( 2^n \)
2. number of pairs of nodes in a set \( S \) is at most \( n^2 \)
3. hence number of subproblems is \( O(n^2 2^n) \)

Exercise: Show that one can compute \text{TSP} using above dynamic program in \( O(n^3 2^n) \) time and \( O(n^2 2^n) \) space.

Disadvantage of dynamic programming solution: memory!
A Recursive Solution

\[ \text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + \text{OPT}(w, v, S - \{u\}) \right) \]

What are the subproblems for the original problem \( \text{OPT}(s, t, V) \)? \( \text{OPT}(u, v, S) \) for \( u, v \in S, S \subseteq V \).

How many subproblems?

1. Number of distinct subsets \( S \) of \( V \) is at most \( 2^n \)
2. Number of pairs of nodes in a set \( S \) is at most \( n^2 \)
3. Hence number of subproblems is \( O(n^2 2^n) \)

**Exercise:** Show that one can compute TSP using above dynamic program in \( O(n^3 2^n) \) time and \( O(n^2 2^n) \) space.

Disadvantage of dynamic programming solution: memory!
A Recursive Solution

\[ \text{OPT}(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + \text{OPT}(w, v, S - \{u\}) \right) \]

What are the subproblems for the original problem \( \text{OPT}(s, t, V) \)? \( \text{OPT}(u, v, S) \) for \( u, v \in S, S \subseteq V \).

How many subproblems?

1. number of distinct subsets \( S \) of \( V \) is at most \( 2^n \)
2. number of pairs of nodes in a set \( S \) is at most \( n^2 \)
3. hence number of subproblems is \( O(n^22^n) \)

Exercise: Show that one can compute TSP using above dynamic program in \( O(n^32^n) \) time and \( O(n^22^n) \) space.

Disadvantage of dynamic programming solution: memory!
Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?
Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?
Some Tips

1. Problems where there is a natural linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.

2. Problems involving trees: recursion based on subtrees.

3. More generally:
   1. Problem admits a natural recursive divide and conquer
   2. If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
   3. If optimal solution depends on all pieces then can apply dynamic programming if interface/interaction between pieces is limited. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.
Examples

1. Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?

2. Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?

3. Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?

4. Independent Set in a graph: break graph into two graphs. What is the interaction? Very high!

5. Knapsack: Split items into two sets of half each. What is the interaction?