Fun Fact: Golden Ratio

Golden Ratio: A universal law.

Golden ratio $\phi = \lim_{n \to \infty} \frac{a_n + b_n}{a_n} = \frac{1 + \sqrt{5}}{2}$

$a_{n+1} = a_n + b_n$, $b_n = a_{n-1}$
Recursion

Reduction: Reduce one problem to another

Recursion: Self-reduction

1. reduce problem to a smaller instance of itself
Recursion

Reduction: Reduce one problem to another

Recursion: Self-reduction

1. reduce problem to a *smaller* instance of *itself*

\[
\text{foo} (\text{instance } X) \\
\text{If } X \text{ is a base case then} \\
\text{solve it and return solution} \\
\text{Else} \\
\text{do some computation} \\
\text{foo}(X_1) \\
\text{do some computation} \\
\text{foo}(X_2) \\
\text{more computation} \\
\text{Output solution for } X
\]
Tail Recursion: *single* recursive call. Easy to convert algorithm into iterative or greedy algorithms.
Recursion in Algorithm Design

1. **Tail Recursion**: single recursive call. Easy to convert algorithm into iterative or greedy algorithms.

2. **Divide and Conquer**: Problem reduced to multiple independent sub-problems that are solved separately. Conquer step puts together solution for bigger problem.

Examples: Merge/Quick Sort, FFT
Recursion in Algorithm Design

1. **Tail Recursion**: single recursive call. Easy to convert algorithm into iterative or greedy algorithms.

2. **Divide and Conquer**: Problem reduced to multiple independent sub-problems that are solved separately. Conquer step puts together solution for bigger problem.
   Examples: Merge/Quick Sort, FFT

3. **Dynamic Programming**: problem reduced to multiple (typically) dependent or overlapping sub-problems. Use memoization to avoid recomputation of common solutions.
Part I

Recursion and Memoization
Recursion, recursion Tree and dependency graph

foo(*instance X*)

If *X* is a base case then
solve it and return solution

Else
do some computation
foo(*X*₁)
do some computation
foo(*X*₂)
more computation
Output solution for *X*
Recursion, recursion Tree and dependency graph

```
foo(instance X)
  If X is a base case then
    solve it and return solution
  Else
    do some computation
    foo(X₁)
    do some computation
    foo(X₂)
    more computation
  Output solution for X
```

Two objects of interest when analyzing foo(X)
- recursion tree of the recursive implementation
- a DAG representing the dependency graph of the distinct subproblems
Example: Fibonacci Numbers

Fibonacci (1200 AD), Pingala (200 BCE).
Numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \text{ and } F(0) = 0, F(1) = 1. \]
Example: Fibonacci Numbers

Fibonacci (1200 AD), Pingala (200 BCE). Numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \text{ and } F(0) = 0, F(1) = 1. \]

These numbers have many interesting and amazing properties. A journal *The Fibonacci Quarterly*!

1. \[ F(n) = (\phi^n - (1 - \phi)^n)/\sqrt{5} \text{ where } \phi \text{ is the golden ratio} \]
   \[ (1 + \sqrt{5})/2 \approx 1.618. \]
2. \[ \lim_{n \to \infty} F(n + 1)/F(n) = \phi \]
Recursive Algorithm for Fibonacci Numbers

Question: Given $n$, compute $F(n)$.

```python
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n - 1) + Fib(n - 2)
```

Running time? Let $T(n)$ be the number of additions in $Fib(n)$.

$$T(n) = T(n - 1) + T(n - 2) + 1$$

and

$$T(0) = T(1) = 0$$

Roughly same as $F(n)$

$$T(n) = \Theta(\phi^n)$$

The number of additions is exponential in $n$. 

Ruta (UIUC)
Question: Given $n$, compute $F(n)$.

\[
\text{Fib}(n) : \\
\begin{align*}
&\text{if } (n = 0) \\
&\quad \text{return } 0 \\
&\text{else if } (n = 1) \\
&\quad \text{return } 1 \\
&\text{else} \\
&\quad \text{return Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]

Running time? Let $T(n)$ be the number of additions in Fib(n).

\[
T(n) = T(n - 1) + T(n - 2) + 1 \text{ and } T(0) = T(1) = 0
\]
Recursive Algorithm for Fibonacci Numbers

Question: Given $n$, compute $F(n)$.

$Fib(n)$:
if $(n = 0)$
return 0
else if $(n = 1)$
return 1
else
return $Fib(n - 1) + Fib(n - 2)$

Running time? Let $T(n)$ be the number of additions in $Fib(n)$.

$T(n) = T(n - 1) + T(n - 2) + 1$ and $T(0) = T(1) = 0$

Roughly same as $F(n)$

$T(n) = \Theta(\phi^n)$

The number of additions is exponential in $n$. 
Recursion tree vs dependency graph

Fib(5)
An iterative algorithm for Fibonacci numbers

**FibIter**(\( n \)):

\[
\begin{align*}
\text{if } (n = 0) & \text{ then return 0} \\
\text{if } (n = 1) & \text{ then return 1} \\
F[0] & = 0 \\
F[1] & = 1
\end{align*}
\]

Running time: \( O(n) \) additions.
An iterative algorithm for Fibonacci numbers

\begin{verbatim}
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
\end{verbatim}

Running time: $O(n)$ additions.
What is the difference?

1. Recursive algorithm is recomputing same subproblems many time.

2. Iterative algorithm is computing the value of a subproblem only once by storing them: Memoization.
What is the difference?

1. Recursive algorithm is recomputing same subproblems many time.

2. Iterative algorithm is computing the value of a subproblem only once by storing them: Memoization.

Dynamic Programming:
Finding a recursion that can be effectively/efficiently memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.
Every recursive algorithm can be memoized by working with the dependency graph.
Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{Fib}(n-1) + \text{Fib}(n-2) & \text{if } \text{Fib}(n) \text{ was previously computed} \\
\text{stored value of Fib}(n) & \text{else} 
\end{cases}
\]

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure)
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

**Fib**(\(n\)):

1. if \((n = 0)\) return 0
2. if \((n = 1)\) return 1
3. if (\(\text{Fib}(n)\) was previously computed) return stored value of \(\text{Fib}(n)\)
4. else return \(\text{Fib}(n - 1) + \text{Fib}(n - 2)\)
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n): \\
\text{if } (n = 0) \quad \text{return } 0 \\
\text{if } (n = 1) \quad \text{return } 1 \\
\text{if } (\text{Fib}(n) \text{ was previously computed}) \\
\quad \text{return} \text{ stored value of } \text{Fib}(n) \\
\text{else} \\
\quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\]

How do we keep track of previously computed values?
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

\[
\text{Fib}(n): \\
\text{if } (n = 0) \quad \text{return } 0 \\
\text{if } (n = 1) \quad \text{return } 1 \\
\text{if } (\text{Fib}(n) \text{ was previously computed}) \quad \text{return } \text{stored value of Fib(n)} \\
\text{else} \\
\text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\]

How do we keep track of previously computed values?  
Two methods: explicitly and implicitly (via data structure)
Explicit memoization

Initialize table/array $M$ of size $n$ such that $M[i] = -1$ for $i = 0, \ldots, n$.

Fib($n$):

- if ($n = 0$) return 0
- if ($n = 1$) return 1
- if ($M[n] \neq -1$) (* $M[n]$ has stored value of Fib($n$) *)
  return $M[n]$
- $M[n] \leftarrow \text{Fib}(n-1) + \text{Fib}(n-2)$
  return $M[n]$

To allocate memory need to know upfront the number of subproblems for a given input size $n$. 
Explicit memoization

Initialize table/array $M$ of size $n$ such that $M[i] = -1$ for $i = 0, \ldots, n$.

```
Fib(n):
    if (n == 0)
        return 0
    if (n == 1)
        return 1
    if (M[n] != -1) (* M[n] has stored value of Fib(n) *)
        return M[n]
    M[n] ← Fib(n - 1) + Fib(n - 2)
    return M[n]
```

To allocate memory need to know upfront the number of subproblems for a given input size $n$
Implicit memoization

Initialize a (dynamic) dictionary data structure $D$ to empty

\[ \text{Fib}(n) : \]
\[ \quad \text{if } (n = 0) \]
\[ \quad \quad \text{return } 0 \]
\[ \quad \text{if } (n = 1) \]
\[ \quad \quad \text{return } 1 \]
\[ \quad \text{if } (n \text{ is already in } D) \]
\[ \quad \quad \text{return value stored with } n \text{ in } D \]
\[ \quad \text{val} \leftarrow \text{Fib}(n - 1) + \text{Fib}(n - 2) \]
\[ \text{Store } (n, \text{val}) \text{ in } D \]
\[ \text{return } \text{val} \]
Explicit vs Implicit Memoization

Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.

Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system. Need to pay overhead of data-structure.

Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.
Explicit vs Implicit Memoization

1. Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.

2. Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system.
   1. Need to pay overhead of data-structure.
   2. Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.
Saving space. Do we need an array of $n$ numbers?
Saving space. Do we need an array of $n$ numbers? Not really.

```python
FibIter(n):
    if (n = 0) then
        return 0
    if (n = 1) then
        return 1
    prev2 = 0
    prev1 = 1
    for i = 2 to n do
        temp = prev1 + prev2
        prev2 = prev1
        prev1 = temp
    return prev1
```
Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.
What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

**Dynamic Programming:**
A recursion that when memoized leads to an *efficient* algorithm.
What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

**Dynamic Programming:**
A recursion that when memoized leads to an *efficient* algorithm.

Key Questions:
- Given a recursive algorithm, how do we analyze complexity when it is memoized?
What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

**Dynamic Programming:**
A recursion that when memoized leads to an *efficient* algorithm.

**Key Questions:**
- Given a recursive algorithm, how do we analyze complexity when it is memoized?
- How do we recognize whether a problem admits a dynamic programming based efficient algorithm?
What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

Dynamic Programming:
A recursion that when memoized leads to an efficient algorithm.

Key Questions:
- Given a recursive algorithm, how do we analyze complexity when it is memoized?
- How do we recognize whether a problem admits a dynamic programming based efficient algorithm?
- How do we further optimize time and space of a dynamic programming based algorithm?
Part II

Edit Distance
Edit Distance

**Definition**

Edit distance between two words $X$ and $Y$ is the number of letter insertions, letter deletions and letter substitutions required to obtain $Y$ from $X$.

**Example**

The edit distance between FOOD and MONEY is at most 4:

\[
\text{FOOD} \rightarrow \text{MOOD} \rightarrow \text{MONOD} \rightarrow \text{MONED} \rightarrow \text{MONEY}
\]
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

F O O D
M O N E Y
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

```
FOOD
MONEY
```

Formally, an alignment is a sequence $M$ of pairs $(i, j)$ such that each index appears exactly once, and there is no “crossing”: if $(i, j), \ldots, (i', j')$ then $i < i'$ and $j < j'$. One of $i$ or $j$ could be empty, in which case no comparison. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (\ , 4), (4, 5)\}$. 
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

FOOOD
MONEY

Formally, an alignment is a sequence $M$ of pairs $(i, j)$ such that each index appears exactly once, and there is no “crossing”: if $(i, j), \ldots, (i', j')$ then $i < i'$ and $j < j'$. One of $i$ or $j$ could be empty, in which case no comparison. In the above example, this is

$M = \{(1, 1), (2, 2), (3, 3), (\ , 4), (4, 5)\}$.

Cost of an alignment: the number of mismatched columns.
Edit Distance Problem

Problem
Given two words, find the edit distance between them, i.e., an alignment of smallest cost.
Edit Distance

Basic observation

Let $X = \alpha x$ and $Y = \beta y$

$\alpha, \beta$: strings. $x$ and $y$ single characters.

Possible alignments between $X$ and $Y$

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$x$</th>
<th>or</th>
<th>$\alpha$</th>
<th>$x$</th>
<th>or</th>
<th>$\alpha x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta$</td>
<td>$y$</td>
<td></td>
<td>$\beta y$</td>
<td></td>
<td></td>
<td>$\beta$</td>
</tr>
</tbody>
</table>

Observation

Prefixes must have optimal alignment!

$\text{EDIST}(X, Y) = \min\begin{cases} \\
\text{EDIST}(\alpha, \beta) + [x \neq y]
\end{cases} + \text{EDIST}(\alpha, \beta y) + \text{EDIST}(\alpha x, \beta y)$
Let $X = \alpha x$ and $Y = \beta y$

$\alpha, \beta$: strings. $x$ and $y$ single characters.

Possible alignments between $X$ and $Y$

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$x$</th>
<th>or</th>
<th>$\alpha$</th>
<th>$x$</th>
<th>or</th>
<th>$\alpha x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta$</td>
<td>$y$</td>
<td></td>
<td>$\beta y$</td>
<td></td>
<td></td>
<td>$\beta$</td>
</tr>
</tbody>
</table>

Observation

*Prefixes must have optimal alignment!*
Let $X = \alpha x$ and $Y = \beta y$

$\alpha, \beta$: strings. $x$ and $y$ single characters.

Possible alignments between $X$ and $Y$

\[
\begin{array}{c|c}
\alpha & x \\
\beta & y \\
\end{array}
\]
or

\[
\begin{array}{c|c}
\alpha & x \\
\beta & y \\
\end{array}
\]
or

\[
\begin{array}{c|c}
\alpha x & \\
\beta & y \\
\end{array}
\]

Observation

Prefixes must have optimal alignment!

\[
EDIST(X, Y) = \min \left\{ \begin{array}{l}
EDIST(\alpha, \beta) + [x \neq y] \\
1 + EDIST(\alpha, \beta y) \\
1 + EDIST(\alpha x, \beta)
\end{array} \right\}
\]
Assume $X$ is stored in array $A[1..m]$ and $Y$ is stored in $B[1..n]$

$$EDIST(A[1..m], B[1..n])$$

If ($m = 0$) return $n$

If ($n = 0$) return $m$
Recursive Algorithm

Assume $X$ is stored in array $A[1..m]$ and $Y$ is stored in $B[1..n]$.

\[
\text{EDIST}(A[1..m], B[1..n])
\]

If $(m = 0)$ return $n$
If $(n = 0)$ return $m$
If $(A[m] = B[n])$ then
\[
m_1 = \text{EDIST}(A[1..(m - 1)], B[1..(n - 1)])
\]
Else
\[
m_1 = 1 + \text{EDIST}(A[1..(m - 1)], B[1..(n - 1)])
\]
\[
m_2 = 1 + \text{EDIST}(A[1..(m - 1)], B[1..n])
\]
\[
m_3 = 1 + \text{EDIST}(A[1..m], B[1..(n - 1)])
\]
return $\min(m_1, m_2, m_3)$
Example

DEED and DREAD
Subproblems and Recurrence

Each subproblem corresponds to a prefix of $X$ and a prefix of $Y$

**Optimal Costs**

Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\text{Opt}(i, j) = \min \begin{cases} 
[x_i \neq y_j] + \text{Opt}(i - 1, j - 1), \\
1 + \text{Opt}(i - 1, j), \\
1 + \text{Opt}(i, j - 1) 
\end{cases}$$

Base Cases: $\text{Opt}(i, 0) = i$ and $\text{Opt}(0, j) = j$
Memoizing the Recursive Algorithm

```
int M[0..m][0..n]
Initialize all entries of $M[i][j]$ to ∞
return $EDIST(A[1..m], B[1..n])$
```
Memoizing the Recursive Algorithm

```
int M[0..m][0..n]
Initialize all entries of M[i][j] to \infty
return EDIST(A[1..m], B[1..n])
```

**EDIST(A[1..m], B[1..n])**

If (M[i][j] < \infty) return M[i][j]  (* return stored value *)
If (m = 0)
  M[i][j] = n
ElseIf (n = 0)
  M[i][j] = m
Memoizing the Recursive Algorithm

```plaintext
int M[0..m][0..n]
Initialize all entries of M[i][j] to \( \infty \)
return EDIST(A[1..m], B[1..n])
```

```plaintext
EDIST(A[1..m], B[1..n])
If (M[i][j] < \( \infty \)) return M[i][j]  (* return stored value *)
If (m = 0)
    M[i][j] = n
ElseIf (n = 0)
    M[i][j] = m
Else
    If (A[m] = B[n])  \( m_1 = EDIST(A[1..(m-1)], B[1..(n-1)]) \)
    Else  \( m_1 = 1 + EDIST(A[1..(m-1)], B[1..(n-1)]) \)
    \( m_2 = 1 + EDIST(A[1..(m-1)], B[1..n]) \)
    \( m_3 = 1 + EDIST(A[1..m], B[1..(n-1)]) \)
    M[i][j] = \text{min}(m_1, m_2, m_3)
return M[i][j]
```
Matrix and DAG of Computation

Matrix M:

Figure: Dependency of matrix entries in the recursive algorithm of previous slide
Removing Recursion to obtain Iterative Algorithm

\[
EDIST(A[1..m], B[1..n])
\]

\[
\text{int } M[0..m][0..n]
\]

for \(i = 1\) to \(m\) do \(M[i, 0] = i\)

for \(j = 1\) to \(n\) do \(M[0, j] = j\)

for \(i = 1\) to \(m\) do
  for \(j = 1\) to \(n\) do
    \[
    M[i][j] = \min \left\{ \begin{array}{l}
    [x_i = y_j] + M[i - 1][j - 1], \\
    1 + M[i - 1][j], \\
    1 + M[i][j - 1]
    \end{array} \right. 
    \]

Analysis

Running time is \(O(mn)\).
Removing Recursion to obtain Iterative Algorithm

\[
\text{EDIST}(A[1..m], B[1..n])
\]

\[
\text{int } M[0..m][0..n]
\]

for \( i = 1 \) to \( m \) do \( M[i, 0] = i \)

for \( j = 1 \) to \( n \) do \( M[0, j] = j \)

for \( i = 1 \) to \( m \) do

  for \( j = 1 \) to \( n \) do

    \[
    M[i][j] = \min \begin{cases} 
    [x_i = y_j] + M[i - 1][j - 1], \\
    1 + M[i - 1][j], \\
    1 + M[i][j - 1]
    \end{cases}
    \]

Analysis

Running time is \( O(mn) \).
Removing Recursion to obtain Iterative Algorithm

\[
EDIST(A[1..m], B[1..n])
\]

\[
\text{int } M[0..m][0..n]
\]

for \( i = 1 \) to \( m \) do \( M[i, 0] = i \)

for \( j = 1 \) to \( n \) do \( M[0, j] = j \)

\[
\text{for } i = 1 \text{ to } m \text{ do } \text{for } j = 1 \text{ to } n \text{ do}
\]

\[
\begin{align*}
M[i][j] &= \min \left\{ [x_i = y_j] + M[i - 1][j - 1], \\
&\quad 1 + M[i - 1][j], \\
&\quad 1 + M[i][j - 1] \right\}
\end{align*}
\]

Analysis

1. Running time is \( O(mn) \).
2. Space used is \( O(mn) \).
Matrix M:

Figure: Iterative algorithm in previous slide computes values in row order.
Finding an Optimum Solution

The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

**Question:** Can we find a specific alignment which achieves the minimum?
Finding an Optimum Solution

The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

**Question:** Can we find a specific alignment which achieves the minimum?

**Exercise:** Show that one can find an optimum solution after computing the optimum value. Key idea is to store back pointers when computing $\text{Opt}(i,j)$ to know how we calculated it. See notes for more details.
Dynamic Programming Template

1. Come up with a recursive algorithm to solve problem
2. Understand the structure/number of the subproblems generated by recursion
3. Memoize the recursion → DP
   - set up compact notation for subproblems
   - set up a data structure for storing subproblem solutions
4. Iterative algorithm
   - Understand dependency graph on subproblems
   - Pick an evaluation order (any topological sort of the dependency dag)
5. Analyze time and space
6. Optimize
Part III

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.
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.
If $W = 11$, the best is $\{I_3, I_4\}$ giving value 40.
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.
For the following instance of Knapsack:

<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>16</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>

and weight limit $W = 15$. The best solution has value:

(A) 22  
(B) 28  
(C) 38  
(D) 50  
(E) 56
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 $v_3 = 3$;

Aside: 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 $W$. 

Ruta (UIUC)
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 $v_2 = 3$;

Aside: 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 $W$. 
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 $v_3 = 5$;

Aside: 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 $W$. 

Ruta (UIUC) CS473 36 Spring 2021 36 / 41
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 \( \{1, 2\} \)
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 $\{1, 2\}$
   - Aside: 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 $W$. 
Towards a Recursive Algorithms

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 I: item $i \not\in \mathcal{O}$ Then $\mathcal{O}$ is an optimal solution to items $1$ to $i - 1$

Case II: item $i \in \mathcal{O}$ Then $\mathcal{O} - \{i\}$ is an optimum solution for items $1$ to $i - 1$ in knapsack of capacity $W - w_i$. 
Towards a Recursive Algorithms

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 I: item \( i \notin \mathcal{O} \) \( \mathcal{O} \) is an optimal solution to items \( 1 \) to \( i - 1 \)

Case II: item \( i \in \mathcal{O} \) Then \( \mathcal{O} - \{i\} \) is an optimum solution for items \( 1 \) to \( i - 1 \) in knapsack of capacity \( W - w_i \).

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

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 I: item $$i \notin \mathcal{O}$$ Then $$\mathcal{O}$$ is an optimal solution to items $$1$$ to $$i - 1$$

Case II: item $$i \in \mathcal{O}$$ Then $$\mathcal{O} - \{i\}$$ is an optimum solution for items $$1$$ to $$i - 1$$ 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 $w$.

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

Number of subproblems generated by $\text{Opt}(n, W)$ is $O(nW)$. 
An Iterative Algorithm

\begin{algorithm}
\begin{align*}
&\text{for } w = 0 \text{ to } W \text{ do} \\
&\qquad M[0, w] = 0 \\
&\text{for } i = 1 \text{ to } n \text{ do} \\
&\quad \text{for } w = 1 \text{ to } W \text{ do} \\
&\qquad \text{if } (w_i > w) \text{ then} \\
&\qquad\qquad M[i, w] = M[i - 1, w] \\
&\qquad \text{else} \\
&\qquad\qquad M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)
\end{align*}
\end{algorithm}

Running Time

Time taken is \( O(nW) \); 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

- Time taken is \( O(nW) \)
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”!
Introducing a Variable

For the Knapsack problem obtaining a recursive algorithm required introducing a new variable, namely the size of the knapsack.

This is a key idea that recurs in many dynamic programming problems.
For the Knapsack problem obtaining a recursive algorithm required introducing a new variable, namely the size of the knapsack.

This is a key idea that recurs in many dynamic programming problems.

How do we figure out when this is possible?

Heuristic answer that works for many problems: Try divide and conquer or obvious recursion: if problem is not decomposable then introduce the “information” required to decompose as new variable(s). Will see several examples to make this idea concrete.
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.

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.

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.
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 \textit{pseudo-polynomial} time algorithm because running time is polynomial if \textit{numbers} in input are of size polynomial in the \textit{combinatorial size} of problem.

6. Knapsack is \textbf{NP-Hard} if numbers are not polynomial in \( n \).