Today

logistics:

pset0 due R5, (aka, tomorrow) — submit individually!
pset1 out tomorrow, due R5 (next week)
piazza signup

last lecture:
divide and conquer
triangle detection
matrix multiplication
today:
recursion
dynamic programming
logistics:
logistics:
- pset0 due R5,
logistics:
- pset0 due R5, (aka, tomorrow)
logistics:
- pset0 due R5, (aka, tomorrow) — submit *individually*!
logistics:
- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow,
logistics:

- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow, due R5 (next week)
logistics:

- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow, due R5 (next week)
- piazza signup
Today

**logistics:**
- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow, due R5 (next week)
- piazza signup

**last lecture:**
- divide and conquer
- triangle detection
- matrix multiplication
- recursion
- dynamic programming
logistics:

- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow, due R5 (next week)
- piazza signup

last lecture:

- divide and conqueror
logistics:

- pset0 due R5, (aka, tomorrow) — submit individually!
- pset1 out tomorrow, due R5 (next week)
- piazza signup

last lecture:

- divide and conqueror
  - triangle detection
logistics:
- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow, due R5 (next week)
- piazza signup

last lecture:
- divide and conqueror
  - triangle detection
  - matrix multiplication
logistics:
- pset0 due R5, (aka, tomorrow) — submit individually!
- pset1 out tomorrow, due R5 (next week)
- piazza signup

last lecture:
- divide and conqueror
  - triangle detection
  - matrix multiplication

today:
Today

**logistics:**
- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow, due R5 (next week)
- piazza signup

**last lecture:**
- divide and conqueror
  - triangle detection
  - matrix multiplication

**today:**
- recursion
logistics:
- pset0 due R5, (aka, tomorrow) — submit *individually*!
- pset1 out tomorrow, due R5 (next week)
- piazza signup

last lecture:
- divide and conqueror
  - triangle detection
  - matrix multiplication

today:
- recursion
- dynamic programming
Recursion

Definition

A reduction transforms a given problem into a yet another problem, possibly into several instances of another problem. Recursion is a reduction from one instance of a problem to instances of the same problem. example (Karatsuba, Strassen, ...): reduce problem instances of size $n$ to problem instances of size $n/2$, terminate recursion at $O(1)$-size problem instances, solve straightforwardly as a base case.
Recursion

Definition

A reduction transforms a given problem into a yet another problem, possibly into several instances of another problem. Recursion is a reduction from one instance of a problem to instances of the same problem. Example (Karatsuba, Strassen, ...): reduce problem instances of size $n$ to problem instances of size $n/2$. Terminate recursion at $O(1)$-size problem instances, solve straightforwardly as a base case.
Recursion

Definition

A **reduction** transforms a given problem into a yet another problem,
A reduction transforms a given problem into a yet another problem, possibly into several instances of another problem.
Recursion

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A <strong>reduction</strong> transforms a given problem into a yet another problem, possibly into <em>several instances</em> of another problem. <strong>Recursion</strong> is a reduction from one instance of a problem to instances of the <em>same</em> problem.</td>
</tr>
</tbody>
</table>

*(example (Karatsuba, Strassen, ...): reduce problem instances of size $n$ to problem instances of size $n/2$, terminate recursion at $O(1)$-size problem instances, solve straightforwardly as a base case)*
Definition

A **reduction** transforms a given problem into a yet another problem, possibly into several instances of another problem. **Recursion** is a reduction from one instance of a problem to instances of the same problem.

**example**
Recursion

Definition

A reduction transforms a given problem into a yet another problem, possibly into several instances of another problem. **Recursion** is a reduction from one instance of a problem to instances of the same problem.

deep

example (Karatsuba,
Recursion

Definition

A **reduction** transforms a given problem into a yet another problem, possibly into **several instances** of another problem. **Recursion** is a reduction from one instance of a problem to instances of the **same** problem.

example (Karatsuba, Strassen, ...):
Recursion

Definition

A reduction transforms a given problem into a yet another problem, possibly into several instances of another problem. Recursion is a reduction from one instance of a problem to instances of the same problem.

example (Karatsuba, Strassen, ...):

- reduce problem instances of size \( n \) to problem instances of size \( n/2 \)
Recursion

**Definition**

A *reduction* transforms a given problem into a yet another problem, possibly into *several instances* of another problem.  
**Recursion** is a reduction from one instance of a problem to instances of the *same* problem.

**example (Karatsuba, Strassen, ...):**
- reduce problem instances of size $n$ to problem instances of size $n/2$
- terminate recursion at $O(1)$-size problem instances,
Recursion

Definition

A **reduction** transforms a given problem into a yet another problem, possibly into several instances of another problem. **Recursion** is a reduction from one instance of a problem to instances of the same problem.

**example (Karatsuba, Strassen, ...):**

- reduce problem instances of size $n$ to problem instances of size $n/2$
- terminate recursion at $O(1)$-size problem instances, solve straightforwardly as a base case
Recursion (II)

Recursive paradigms:

- Tail recursion: expend effort to reduce given problem to a single (smaller) problem. Often can be reformulated as a non-recursive iterative algorithm.

- Divide and conquer: expend effort to reduce the given problem to multiple, independent smaller problems, which are solved separately. Solutions to smaller problems are combined to solve the original problem.
  
  For example: Karatsuba, Strassen, . . .

- Dynamic programming: expend effort to reduce the given problem to multiple correlated smaller problems. Naive recursion often not efficient, use memoization to avoid wasteful recomputation.
recursive paradigms:

- **Tail recursion**: expend effort to reduce given problem to a single (smaller) problem. Often can be reformulated as a non-recursive iterative algorithm.

- **Divide and conquer**: expend effort to reduce the given problem to multiple, independent smaller problems, which are solved separately. Solutions to smaller problems are combined to solve the original problem (conquer). For example: Karatsuba, Strassen, ... .

- **Dynamic programming**: expend effort to reduce the given problem to multiple correlated smaller problems. Naive recursion often not efficient, use memoization to avoid wasteful recomputation.
recursive paradigms:

- **tail recursion**:
recursive paradigms:
  - **tail recursion**: expend effort
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* problem.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to single (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce given problem to multiple, independent smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, . . .

- **dynamic programming**: expend effort to reduce given problem to multiple correlated smaller problems. Naive recursion often not efficient, use memoization to avoid wasteful recomputation.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to single (smaller) problem. Often can be reformulated as a non-recursive iterative algorithm.

- **divide and conquer**: expend effort to reduce
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide)
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, smaller problems,
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems,
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to single (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to multiple, *independent* smaller problems, which are solved separately.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer).
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple, independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example:
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to single (smaller) problem. Often can be reformulated as a non-recursive iterative algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to multiple, independent smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba,
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen,
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, . . . .
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, . . . .

- **dynamic programming**: 
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, . . . .

- **dynamic programming**: expend effort to reduce given problem to multiple smaller problems.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, . . . .

- **dynamic programming**: expend effort to reduce given problem to multiple *correlated* smaller problems.
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, . . . .

- **dynamic programming**: expend effort to reduce given problem to multiple *correlated* smaller problems. Naive recursion often *not* efficient,
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive *iterative* algorithm.

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple*, *independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, . . . .

- **dynamic programming**: expend effort to reduce given problem to multiple *correlated* smaller problems. Naive recursion often *not* efficient, use **memoization** to avoid wasteful recomputation.
Recursion (II)

```python
foo(X)
if X is a base case
   solve it
   return solution
else
   do stuff
   foo(X1)
   do stuff
   foo(X2)
   foo(X3)
   more stuff
   return solution for X
```

**analysis:**
- **recursion tree:** each instance X spawns new children X1, X2, X3.
- **dependency graph:** each instance X links to sub-problems X1, X2, X3.
Recursion (II)

```ruby
def foo(X):
    if X is a base case:
        solve it
        return solution
    else:
        do stuff
        foo(X1)
        do stuff
        foo(X2)
        foo(X3)
        more stuff
        return solution for X
```

**Analysis:**
- **Recursion Tree:** Each instance of `X` spawns new children `X1`, `X2`, `X3`.
- **Dependency Graph:** Each instance of `X` links to sub-problems `X1`, `X2`, `X3`.

5 / 33
Recursion (II)

foo($X$)
    if $X$ is a base case

analysis:
recursion tree:
each instance $X$ spawns new children $X_1, X_2, X_3$
dependency graph:
each instance $X$ links to sub-problems $X_1, X_2, X_3$
Recursion (II)

```python
def foo(X):
    if X is a base case
        solve it
```

Analysis:
Recursion tree:
each instance \( X \) spawns new children \( X_1, X_2, X_3 \)

Dependency graph:
each instance \( X \) links to sub-problems \( X_1, X_2, X_3 \)
Recursion (II)

```python
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X1)
        do stuff
        foo(X2)
        foo(X3)
        more stuff
        return solution for X
```

analysis:
- recursion tree: each instance of $X$ spawns new children $X_1$, $X_2$, $X_3$
- dependency graph: each instance of $X$ links to sub-problems $X_1$, $X_2$, $X_3$
Recursion (II)

```python
def foo(X):
    if X is a base case
        solve it
        return solution
    else
```

*Recursion tree:* each instance of $X$ spawns new children $X_1$, $X_2$, $X_3$.

*Dependency graph:* each instance of $X$ links to sub-problems $X_1$, $X_2$, $X_3$.
Recursion (II)

```python
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff

analysis:
recursion tree:
each instance X spawns new children X1, X2, X3
dependency graph:
each instance X links to sub-problems X1, X2, X3
```
Recursion (II)

foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X₁)

Analysis:

- **Recursion tree:**
  Each instance X spawns new children X₁, X₂, X₃.

- **Dependency graph:**
  Each instance X links to sub-problems X₁, X₂, X₃.
Recursion (II)

foo(X)
  if X is a base case
    solve it
    return solution
  else
    do stuff
    foo(X_1)
    do stuff
  return solution for X
Recursion (II)

```python
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X_1)
        do stuff
        foo(X_2)
        foo(X_3)
```

analysis:

recursion tree:
each instance $X$ spawns new children $X_1$, $X_2$, $X_3$

dependency graph:
each instance $X$ links to sub-problems $X_1$, $X_2$, $X_3$
Recursion (II)

```python
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X_1)
        do stuff
        foo(X_2)
        foo(X_3)
        more stuff
```
Recursion (II)

```python
def foo(X):
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X1)
        do stuff
        foo(X2)
        foo(X3)
        more stuff
        return solution for X
```
Recursion (II)

```
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X_1)
        do stuff
        foo(X_2)
        foo(X_3)
        more stuff
        return solution for X
```
Recursion (II)

```python
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X_1)
        do stuff
        foo(X_2)
        foo(X_3)
        more stuff
        return solution for X
```

analysis:
Recursion (II)

```python
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X_1)
        do stuff
        foo(X_2)
        foo(X_3)
        more stuff
        return solution for X
```

**analysis:**

- *recursion tree:*
Recursion (II)

foo(X)
    if X is a base case
    solve it
    return solution
    else
    do stuff
    foo(X_1)
    do stuff
    foo(X_2)
    foo(X_3)
    more stuff
    return solution for X

analysis:
- recursion tree: each instance X spawns new children X_1, X_2, X_3
Recursion (II)

```python
foo(X)
    if X is a base case
        solve it
        return solution
    else
        do stuff
        foo(X_1)
        do stuff
        foo(X_2)
        foo(X_3)
        more stuff
        return solution for X
```

**analysis:**

- *recursion tree:* each instance X spawns *new* children X₁, X₂, X₃
- *dependency graph:*
Recursion (II)

```
foo(X)
  if X is a base case
    solve it
    return solution
  else
    do stuff
    foo(X_1)
    do stuff
    foo(X_2)
    foo(X_3)
    more stuff
    return solution for X
```

**analysis:**

- *recursion tree*: each instance $X$ spawns *new* children $X_1, X_2, X_3$
- *dependency graph*: each instance $X$ links to sub-problems $X_1, X_2, X_3$
Fibonacci Numbers

The Fibonacci sequence \( F_0, F_1, F_2, F_3, \ldots \) is the sequence of numbers defined by
\[
F_0 = 0
\]
\[
F_1 = 1
\]
\[
F_n = F_{n-1} + F_{n-2}, \quad \text{for } n \geq 2
\]

Remarks:
- Arises in surprisingly many places—the journal *The Fibonacci Quarterly*
- \( F_n = \phi^n - (1 - \phi)^n \sqrt{5} \), where \( \phi \) is the golden ratio \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \cdots \)
- \( 1 - \phi \approx -0.618 \cdots \)
- \( |(1 - \phi)^n| \leq 1 \)
- Further, \( (1 - \phi)^n \to 0 \) as \( n \to \infty \)

\( F_n = \Theta(\phi^n) \).
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence \( F_0, F_1, F_2, F_3, \ldots \) is the sequence of numbers defined by

\[
F_0 = 0 \\
F_1 = 1 \\
F_n = F_{n-1} + F_{n-2}, \quad \text{for } n \geq 2
\]

Remarks:

- Arises in surprisingly many places - the journal *The Fibonacci Quarterly*

\[
F_n = \varphi^n - (1 - \varphi)^n \sqrt{5},
\]

where \( \varphi \) is the golden ratio \( \varphi := \frac{1 + \sqrt{5}}{2} \approx 1.618 \ldots \)

\[
1 - \varphi \approx -0.618 \ldots
\]

\[
| (1 - \varphi)^n | \leq 1,
\]

and further \( (1 - \varphi)^n \rightarrow 0 \) as \( n \rightarrow \infty \).

\[
F_n = \Theta(\varphi^n).
\]
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

Remarks:

- Arises in surprisingly many places - the journal *The Fibonacci Quarterly*

- $F_n = \varphi^n - (1 - \varphi)^n \sqrt{5}$, where $\varphi$ is the golden ratio $\varphi := 1 + \sqrt{5}/2 \approx 1.618 \ldots$.

- $1 - \varphi \approx -0.618 \ldots$

- $| (1 - \varphi)^n | \leq 1$, and further $(1 - \varphi)^n \to 0$ as $n \to \infty$.

- $F_n = \Theta(\varphi^n)$. 


Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}, \quad \text{for } n \geq 2$$

Remarks:

- Arises in surprisingly many places — the journal *The Fibonacci Quarterly*

- $F_n = \phi^n - (1 - \phi)^n \sqrt{5}$, where $\phi$ is the golden ratio $\phi := \frac{1 + \sqrt{5}}{2} \approx 1.618\ldots$

- $1 - \phi \approx -0.618\ldots$ implies $| (1 - \phi)^n | \leq 1$, and further $(1 - \phi)^n \rightarrow n \rightarrow \infty 0 = \Rightarrow F_n = \Theta(\phi^n)$.\]
Fibonacci Numbers

<table>
<thead>
<tr>
<th>Definition (Fibonacci 1200, Pingala -200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Fibonacci sequence ( F_0, F_1, F_2, F_3, \ldots \in \mathbb{N} ) is the sequence of numbers defined by</td>
</tr>
</tbody>
</table>

\[
F_0 = 0, \\
F_1 = 1, \\
F_n = F_{n-1} + F_{n-2}, \quad n \geq 2
\]
Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
<table>
<thead>
<tr>
<th>Fibonacci Numbers</th>
</tr>
</thead>
</table>

**Definition (Fibonacci 1200, Pingala -200)**

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$

And further, $F_n = \phi^n - (1 - \phi)^n \sqrt{5}$, where $\phi$ is the golden ratio $\phi := \frac{1 + \sqrt{5}}{2}$, approximately $1.618 \ldots$. $|1 - \phi| \approx -0.618 \ldots$, and thus $|(1 - \phi)^n| \leq 1$, so $F_n = \Theta(\phi^n)$.
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$,
Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence \( F_0, F_1, F_2, F_3, \ldots \in \mathbb{N} \) is the sequence of numbers defined by

- \( F_0 = 0 \)
- \( F_1 = 1 \)
- \( F_n = F_{n-1} + F_{n-2} \), for \( n \geq 2 \)

**remarks:**

- \( F_n = \phi^n - (1 - \phi)^n \frac{1}{\sqrt{5}} \), where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \cdots \)
- \( |1 - \phi| \approx -0.618 \cdots \)
- \( |(1 - \phi)^n| \leq 1 \)
- \( (1 - \phi)^n \to 0 \) as \( n \to \infty \)

\( F_n = \Theta(\phi^n) \)
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

**remarks:**

- arises in surprisingly many places

ϕ is the golden ratio $\approx 1.618 \ldots = \Rightarrow 1 - \varphi \approx -0.618 \ldots \Rightarrow |(1 - \varphi)| \leq 1$, and further $(1 - \varphi) \rightarrow 0 \Rightarrow F_n = \Theta(\varphi^n)$. 6 / 33
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

**remarks:**

- arises in surprisingly many places — the journal *The Fibonacci Quarterly*
Fibonacci Numbers

**Definition (Fibonacci 1200, Pingala -200)**

The Fibonacci sequence \( F_0, F_1, F_2, F_3, \ldots \in \mathbb{N} \) is the sequence of numbers defined by

- \( F_0 = 0 \)
- \( F_1 = 1 \)
- \( F_n = F_{n-1} + F_{n-2}, \text{ for } n \geq 2 \)

**Remarks:**

- Arises in surprisingly many places — the journal *The Fibonacci Quarterly*
- \( F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \), where \( \varphi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \cdots \)
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

remarks:

- arises in surprisingly many places — the journal *The Fibonacci Quarterly*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, $\varphi$ is the *golden ratio* $\varphi := \frac{1+\sqrt{5}}{2}$
Fibonacci Numbers

**Definition (Fibonacci 1200, Pingala -200)**

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

**remarks:**

- arises in surprisingly many places — the journal *The Fibonacci Quarterly*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, $\varphi$ is the *golden ratio* $\varphi := \frac{1 + \sqrt{5}}{2} \approx 1.618 \ldots$
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence \( F_0, F_1, F_2, F_3, \ldots \in \mathbb{N} \) is the sequence of numbers defined by

- \( F_0 = 0 \)
- \( F_1 = 1 \)
- \( F_n = F_{n-1} + F_{n-2}, \text{ for } n \geq 2 \)

**remarks:**

- arises in surprisingly many places — the journal *The Fibonacci Quarterly*
- \( F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \), \( \varphi \) is the *golden ratio* \( \varphi := \frac{1+\sqrt{5}}{2} \approx 1.618 \cdots \)
- \( \implies 1 - \varphi \approx -.618 \cdots \)
## Fibonacci Numbers

**Definition (Fibonacci 1200, Pingala -200)**

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

### Remarks:

- arises in surprisingly many places — the journal *The Fibonacci Quarterly*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, $\varphi$ is the *golden ratio* $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618 \cdots$
- $\implies 1 - \varphi \approx -0.618 \cdots \implies |(1 - \varphi)^n| \leq 1,$
Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

**Remarks:**

- arises in surprisingly many places — the journal *The Fibonacci Quarterly*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, $\varphi$ is the golden ratio $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618 \cdots$
- $\Rightarrow 1 - \varphi \approx -0.618 \cdots \Rightarrow |(1 - \varphi)^n| \leq 1$, and further $(1 - \varphi)^n \rightarrow_{n \rightarrow \infty} 0$
The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by:

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

**Remarks:**

- Arises in surprisingly many places — the journal *The Fibonacci Quarterly*

- $F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$, $\phi$ is the golden ratio $\phi := \frac{1+\sqrt{5}}{2} \approx 1.618 \ldots$

- $1 - \phi \approx -0.618 \ldots \Rightarrow |(1 - \phi)^n| \leq 1$, and further $(1 - \phi)^n \rightarrow_{n \to \infty} 0$

- $F_n = \Theta(\phi^n)$.
Fibonacci Numbers (II)

question:
given $n$, compute $F_n$.

answer:
$\text{fib}(n) :$

if $n = 0$
return 0

elif $n = 1$
return 1

else
return $\text{fib}(n - 1) + \text{fib}(n - 2)$

correctness:
clear

complexity:
let $T(n)$ denote the number of additions.

Then $T(0) = 0$, $T(1) = 0$,
$T(2) = 1$, $T(n) = T(n - 1) + T(n - 2) = \Rightarrow T(n) = F_{n-1} = \Theta(\phi^n) = \Rightarrow$ exponential time!
question:

given $n$, compute $F_n$.

answer:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

correctness:

The correctness of the function can be verified by checking the base cases and the recursive case.

complexity:

Let $T(n)$ denote the number of additions. Then $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, and $T(n) = T(n-1) + T(n-2)$.

$\Rightarrow T(n) = F_n - 1 = \Theta(\varphi^n)$

$\Rightarrow$ exponential time!
question: given $n$, compute $F_n$. 
question: given $n$, compute $F_n$.

answer:
question: given $n$, compute $F_n$.
answer:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```
question: given $n$, compute $F_n$.

answer:

```python
fib(n):
    if $n = 0$
```

correctness:

complexity:

let $T(n)$ denote the number of additions.

Then $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n-1) + T(n-2) = \Rightarrow T(n) = F_n - 1 = \Theta(\varphi^n) = \Rightarrow$ exponential time!
question: given \( n \), compute \( F_n \).

answer:

\[
\text{fib}(n):
\]
\[
\quad \text{if} \ n = 0
\]
\[
\quad \text{return} \ 0
\]
question: given $n$, compute $F_n$.

answer:

```python
fib(n):
    if $n = 0$
        return 0
    elif $n = 1$
        return 1
    else
        return fib($n - 1$) + fib($n - 2$)
```

correctness:
clear

complexity:
let $T(n)$ denote the number of additions.
Then $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n - 1) + T(n - 2) = \Rightarrow T(n) = F_{n-1} = \Theta(\phi^n) = \Rightarrow$ exponential time!
question: given $n$, compute $F_n$.

answer:

```python
fib(n):
    if $n = 0$
        return 0
    elif $n = 1$
        return 1
```

correctness:

complexity:

let $T(n)$ denote the number of additions.

Then $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n-1) + T(n-2)$ $\Rightarrow T(n) = F_n - 1$ $\Rightarrow \Theta(\phi^n) = \text{exponential time!}$
question: given $n$, compute $F_n$.

answer:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

correctness:

complexity:

Let $T(n)$ denote the number of additions. Then

- $T(0) = 0$,
- $T(1) = 0$,
- $T(2) = 1$,
- $T(n) = T(n-1) + T(n-2)$.

$⇒ T(n) = F_{n-1}$

$= Θ(\phi^n)$

$⇒$ exponential time!
question: given $n$, compute $F_n$.

answer:

```python
fib(n):
    if $n = 0$
        return 0
    elif $n = 1$
        return 1
    else
        return fib($n - 1$)
```

correctness:

complexity:

let $T(n)$ denote the number of additions. Then $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n - 1) + T(n - 2)$, which implies $T(n) = F_n - 1 = \Theta(\phi^n) = \Rightarrow$ exponential time!
question: given $n$, compute $F_n$.
answer:

```python
def fib(n):
    if n == 0
        return 0
    elif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
```

correctness: clear
complexity: let $T(n)$ denote the number of additions. Then $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n-1) + T(n-2) = \Rightarrow T(n) = F_n - 1 = \Theta(\phi^n) = \Rightarrow$ exponential time!
**question:** given $n$, compute $F_n$.

**answer:**

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

**correctness:**

**complexity:**

Let $T(n)$ denote the number of additions. Then $T(0) = 0$, $T(1) = 0$, $T(2) = 1$, $T(n) = T(n - 1) + T(n - 2) = \Rightarrow T(n) = F_{n - 1} = \Theta(\phi^n)$ = $\Rightarrow$ exponential time!
question: given \( n \), compute \( F_n \).

answer:

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

correctness:
question: given \( n \), compute \( F_n \).

answer:

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

correctness: clear
Fibonacci Numbers (II)

**question:** given $n$, compute $F_n$.

**answer:**

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

**correctness:** clear

**complexity:**
question: given $n$, compute $F_n$.

answer:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

correctness: clear

complexity: let $T(n)$ denote the number of additions.
question: given $n$, compute $F_n$.

answer:

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

correctness: clear

complexity: let $T(n)$ denote the number of additions. Then
Fibonacci Numbers (II)

**question:** given $n$, compute $F_n$.

**answer:**

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

**correctness:** clear

**complexity:** let $T(n)$ denote the number of additions. Then
- $T(0) = 0$, 
- $T(n) = T(n-1) + T(n-2)$
Fibonacci Numbers (II)

**question:** given $n$, compute $F_n$.

**answer:**

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

**correctness:** clear

**complexity:** let $T(n)$ denote the number of *additions*. Then

- $T(0) = 0$, $T(1) = 0$
Fibonacci Numbers (II)

**question:** given $n$, compute $F_n$.

**answer:**

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

**correctness:** clear

**complexity:** let $T(n)$ denote the number of *additions*. Then
- $T(0) = 0$, $T(1) = 0$
- $T(2) =$
Fibonacci Numbers (II)

**question:** given $n$, compute $F_n$.

**answer:**

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

**correctness:** clear

**complexity:** let $T(n)$ denote the number of additions. Then

- $T(0) = 0$, $T(1) = 0$
- $T(2) = 1$, 

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

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

exponential time!
question: given $n$, compute $F_n$.

answer:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

correctness: clear

complexity: let $T(n)$ denote the number of additions. Then

- $T(0) = 0$, $T(1) = 0$
- $T(2) = 1$
- $T(n) = T(n-1) + T(n-2)$
Fibonacci Numbers (II)

**question:** given $n$, compute $F_n$.

**answer:**

```python
fib(n):
    if $n = 0$
        return 0
    elif $n = 1$
        return 1
    else
        return fib($n - 1$) + fib($n - 2$)
```

**correctness:** clear

**complexity:** let $T(n)$ denote the number of *additions*. Then

- $T(0) = 0$, $T(1) = 0$
- $T(2) = 1$,
- $T(n) = T(n - 1) + T(n - 2)$
- $\implies T(n) = F_{n-1}$
**Fibonacci Numbers (II)**

**question:** given $n$, compute $F_n$.

**answer:**

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

**correctness:** clear

**complexity:** let $T(n)$ denote the number of *additions*. Then

- $T(0) = 0$, $T(1) = 0$
- $T(2) = 1$,
- $T(n) = T(n-1) + T(n-2)$
- $\Rightarrow T(n) = F_{n-1} = \Theta(\varphi^n)$
question: given \( n \), compute \( F_n \).

answer:

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

correctness: clear

complexity: let \( T(n) \) denote the number of additions. Then

- \( T(0) = 0, \ T(1) = 0 \)
- \( T(2) = 1, \)
- \( T(n) = T(n - 1) + T(n - 2) \)
- \( \Rightarrow \ \ T(n) = F_{n-1} = \Theta(\varphi^n) \ \Rightarrow \ \text{exponential time!} \)
Fibonacci Numbers (III)

recursion tree:

dependency graph:
recursion tree:
**recursion tree:** for $F_4$
recursion tree: for $F_4$
recursion tree: for $F_4$
recursion tree: for $F_4$
Fibonacci Numbers (III)

**recursion tree:** for $F_4$

```
  F_4
 /   \\   
F_3  F_2
 /     \
F_2  F_1
 /     \
F_1  F_0
```
recursion tree: for $F_4$
recursion tree: for $F_4$

dependency graph:
Fibonacci Numbers (III)

**recursion tree:** for $F_4$

![Recursion Tree for $F_4$]

**dependency graph:** for $F_4$

![Dependency Graph for $F_4$]
Fibonacci Numbers (III)

**recursion tree:** for $F_4$

```
  F_4
 /|
/  |
F_3 F_2
 /|
/  |
F_1 F_1 F_0
 /|
/  |
F_1 F_0
```

**dependency graph:** for $F_4$

```
F_4
```
Fibonacci Numbers (III)

**recursion tree:** for $F_4$

```
        F_4
       /   \
   F_3     F_2
  /   \    /   \ \
F_2  F_1  F_1  F_0
 / \    /  \\  /   \\
F_1 F_0
```

**dependency graph:** for $F_4$

```
F_4
F_3
```

$8 / 33$
recursion tree: for $F_4$

dependency graph: for $F_4$
Fibonacci Numbers (III)

recursion tree: for $F_4$

```
  F_4
 /   \
F_3   F_2
 /     \
F_2   F_1
 /     \
F_1   F_0
```

dependency graph: for $F_4$

```
F_4
F_3
F_2
F_1
```
**Fibonacci Numbers (III)**

**recursion tree:** for $F_4$

```
F_4
   /   \
  /     \
F_3   F_2
  /   /   \
F_2 F_1 F_1
   /     /   \
F_1 F_0 F_0
```

**dependency graph:** for $F_4$

```
F_4
   /   \
  /     \
F_3   F_2
  /   /   \
F_1 F_1 F_0
   /     /   \
F_0 F_0 F_1
```
Fibonacci Numbers (III)

**recursion tree:** for $F_4$

```
F_4
  F_3
    F_2
      F_1
        F_0
```

**dependency graph:** for $F_4$

```
F_4
  F_3
    F_2
      F_1
        F_0
```
Fibonacci Numbers (III)

**recursion tree:** for $F_4$

```
F_4
  /   /
F_3  F_2
 /     /
F_2  F_1  F_1
 /     /   /
F_1  F_0  F_0
```

**dependency graph:** for $F_4$

```
F_4
  /   /
F_3  F_2
 /     /
F_1
```

8 / 33
Fibonacci Numbers (III)

**Recursion tree:** for $F_4$

```
   F_4
  /   |
F_3   F_2
  |   / |
F_2 F_1 F_1 F_0
 |   |   |
F_1 F_0
```

**Dependency graph:** for $F_4$

```
   F_4
  /   |
F_3   F_2
  |   / |
F_2 F_1
  |   / |
F_1 F_0
```
Fibonacci Numbers (IV)

iterative algorithm:

\[
\text{fib-iter}(n) =
\begin{align*}
&\text{if } n = 0 \text{ return } 0 \\
&\text{if } n = 1 \text{ return } 1 \\
&F_0 = 0 \\
&F_1 = 1 \\
&\text{for } 2 \leq i \leq n \\
&\quad F_i = F_{i-1} + F_{i-2} \\
&\text{return } F_n
\end{align*}
\]

correctness:

complexity:

\[O(n)\] additions

remarks:

\[F_n = \Theta(\phi^n)\]

\[\Rightarrow F_n \text{ takes } \Theta(n) \text{ bits} \]

\[\Rightarrow \text{each addition takes } \Theta(n) \text{ steps} \]

\[\Rightarrow O(n^2) \text{ is the actual runtime} \]
Fibonacci Numbers (IV)

iterative algorithm:

fib-iter(n):
    if n = 0 return 0
    if n = 1 return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i−1] + F[i−2]
    return F[n]

correctness:

complexity:
O(n) additions

remarks:
F_n = Θ(ϕ^n) ⇒ F_n takes Θ(n) bits ⇒ each addition takes Θ(n) steps ⇒ O(n^2) is the actual runtime
Fibonacci Numbers (IV)

iterative algorithm:

fib-iter(n):

- if \( n = 0 \) return 0
- if \( n = 1 \) return 1
- \( F[0] = 0 \)
- \( F[1] = 1 \)
- for \( 2 \leq i \leq n \) \( F[i] = F[i - 1] + F[i - 2] \)
- return \( F[n] \)

correctness:

complexity:

O(n) additions

remarks:

\( F_n = \Theta(\phi^n) \)

\( \Rightarrow F_n \) takes \( \Theta(n) \) bits

\( \Rightarrow \) each addition takes \( \Theta(n) \) steps

\( \Rightarrow O(n^2) \) is the actual runtime
Fibonacci Numbers (IV)

iterative algorithm:

fib-iter(n):
    if $n = 0$
        return 0
    if $n = 1$
        return 1

for $2 \leq i \leq n$
    $F[i] = F[i-1] + F[i-2]$

return $F[n]$
Fibonacci Numbers (IV)

iterative algorithm:

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

correctness:

complexity:

O(n) additions

remarks:

F_n = Θ(ϕ^n) ⇒ F_n takes Θ(n) bits ⇒ each addition takes Θ(n) steps ⇒ O(n^2) is the actual runtime
iterative algorithm:

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

correctness:

complexity:

O(n) additions

remarks:

F_n = \Theta(\phi^n) \Rightarrow F_n takes \Theta(n) bits \Rightarrow each addition takes \Theta(n) steps \Rightarrow O(n^2) is the actual runtime
Fibonacci Numbers (IV)

iterative algorithm:

\[
\begin{align*}
\text{fib-iter}(n): \\
\quad \text{if } n = 0 \\
\quad \quad \text{return } 0 \\
\quad \text{if } n = 1 \\
\quad \quad \text{return } 1 \\
F[0] &= 0 \\
F[1] &= 1 \\
\text{for } 2 \leq i \leq n \\
\quad \quad F[i] &= F[i - 1] + F[i - 2]
\end{align*}
\]

correctness:

complexity:

O(n) additions

remarks:

F_n = \Theta(\phi^n) \Rightarrow F_n \text{ takes } \Theta(n) \text{ bits} \Rightarrow \text{each addition takes } \Theta(n) \text{ steps} \Rightarrow O(n^2) \text{ is the actual runtime}
iterative algorithm:

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

**correctness:**

**complexity:** $O(n)$ additions

**remarks:** $F_n = \Theta(\phi^n) \Rightarrow F_n$ takes $\Theta(n)$ bits $\Rightarrow$ each addition takes $\Theta(n)$ steps $\Rightarrow O(n^2)$ is the actual runtime
Fibonacci Numbers (IV)

iterative algorithm:

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

correctness:

complexity:

O(n) additions

remarks:

F_n = Θ(ϕ^n) =⇒ F_n takes Θ(n) bits =⇒ each addition takes Θ(n) steps =⇒ O(n^2) is the actual runtime
**Fibonacci Numbers (IV)**

**iterative algorithm:**

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

**correctness:**
Fibonacci Numbers (IV)

iterative algorithm:

```
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

correctness: clear
Fibonacci Numbers (IV)

iterative algorithm:

```python
def fib_iter(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    F[0] = 0
    F[1] = 1
    for i in range(2, n + 1):
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

correctness: clear

complexity:
Fibonacci Numbers (IV)

iterative algorithm:

```python
def fib_iter(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    F[0] = 0
    F[1] = 1
    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

correctness: clear

complexity: $O(n)$ additions
Fibonacci Numbers (IV)

iterative algorithm:

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

correctness: clear

complexity: $O(n)$ additions

remarks:

$F_n = \Theta(\phi^n) \Rightarrow F_n$ takes $\Theta(n)$ bits $\Rightarrow$ each addition takes $\Theta(n)$ steps $\Rightarrow O(n^2)$ is the actual runtime
Fibonacci Numbers (IV)

iterative algorithm:

```plaintext
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

correctness: clear

complexity: \( O(n) \) additions

remarks:

- \( F_n = \Theta(\varphi^n) \)
Fibonacci Numbers (IV)

iterative algorithm:

\[
\text{fib-iter}(n):
\]

\[
\begin{align*}
\text{if } n &= 0 \\
& \quad \text{return } 0 \\
\text{if } n &= 1 \\
& \quad \text{return } 1 \\
F[0] &= 0 \\
F[1] &= 1 \\
\text{for } 2 \leq i \leq n \\
& \quad F[i] = F[i - 1] + F[i - 2] \\
& \quad \text{return } F[n]
\end{align*}
\]

correctness: clear

complexity: \(O(n)\) additions

remarks:
- \(F_n = \Theta(\varphi^n) \implies F_n\) takes \(\Theta(n)\) bits
iterative algorithm:

```python
fib-iter(n):
    if \( n = 0 \)
        return 0
    if \( n = 1 \)
        return 1
    \( F[0] = 0 \)
    \( F[1] = 1 \)
    for \( 2 \leq i \leq n \)
        \( F[i] = F[i - 1] + F[i - 2] \)
    return \( F[n] \)
```

correctness: clear

complexity: \( O(n) \) additions

remarks:
- \( F_n = \Theta(\varphi^n) \implies F_n \) takes \( \Theta(n) \) bits \implies each addition takes \( \Theta(n) \) steps
iterative algorithm:

```python
def fib_iter(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n:
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

correctness: clear

complexity: $O(n)$ additions

remarks:
- $F_n = \Theta(\varphi^n) \implies F_n$ takes $\Theta(n)$ bits $\implies$ each addition takes $\Theta(n)$ steps $\implies O(n^2)$ is the actual runtime
Memoization

recursive paradigms for $F_n$:
recursive paradigms for $F_n$:

- **naive recursion:**
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems,
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times
recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times

- **iterative algorithm**:
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times

- **iterative algorithm**: stores solutions to subproblems to avoid recomputation
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the same subproblem multiple times
- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — memoization
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times
- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — memoization

**Definition**
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times
- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — memoization

**Definition**

*Dynamic programming* is the method of speeding up naive recursion through memoization.
recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the same subproblem multiple times
- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — memoization

**Definition**

**Dynamic programming** is the method of speeding up naive recursion through memoization.

**remarks:**
recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times
- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — memoization

**Definition**

**Dynamic programming** is the method of speeding up naive recursion through memoization.

**remarks:**

- If number of subproblems is polynomially bounded,
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times
- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — memoization

**Definition**

*Dynamic programming* is the method of speeding up naive recursion through memoization.

**Remarks:**

- If number of subproblems is polynomially bounded, often implies a polynomial-time algorithm
Memoization

recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the same subproblem multiple times

- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — memoization

**Definition**

**Dynamic programming** is the method of speeding up naive recursion through memoization.

**Remarks**:

- If number of subproblems is polynomially bounded, often implies a polynomial-time algorithm

- Memoizing a recursive algorithm is done by tracing through the dependency graph
Memoization (II)

question: how to memoize exactly?

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

question: how to memoize exactly?

explicitly: just do it!

implicitly: allow clever data structures to do this automatically
question:

\[
\text{fib(n)}:\ 
\begin{align*}
\text{if } n &= 0 \quad \text{return } 0 \\
\text{if } n &= 1 \quad \text{return } 1 \\
\text{if fib(n) was previously computed } \quad \text{return } \text{stored value fib(n)} \\
\text{else } \quad \text{return } \text{fib(n-1)} + \text{fib(n-2)}
\end{align*}
\]

question: how to memoize exactly?

explicitly: just do it!

implicitly: allow clever data structures to do this automatically
question: how to memoize exactly?

fib(n):
if \( n = 0 \) return 0
if \( n = 1 \) return 1
if fib(n) was previously computed return stored value
else return fib(n - 1) + fib(n - 2)

question: how to memoize exactly?

explicitly: just do it!
implicitly: allow clever data structures to do this automatically
**question:** how to memoize exactly?

```python
fib(n):
    if n == 0 return 0
    if n == 1 return 1
    if fib(n) was previously computed return stored value
    else return fib(n-1) + fib(n-2)
```
question: how to memoize exactly?

```
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value
    else
        return fib(n-1) + fib(n-2)
```
question: how to memoize exactly?

```python
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
```
question: how to memoize exactly?

```python
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
```

explicitly: just do it!

implicitly: allow clever data structures to do this automatically
question: how to memoize exactly?

fib(n):
    if $n = 0$
        return 0
    if $n = 1$
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
question: how to memoize exactly?

```python
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
        return fib(n-1) + fib(n-2)
```
question: how to memoize exactly?

\[
\text{fib}(n):
\begin{align*}
&\quad \text{if } n = 0 \\
&\quad \quad \text{return } 0 \\
&\quad \text{if } n = 1 \\
&\quad \quad \text{return } 1 \\
&\quad \text{if } \text{fib}(n) \text{ was previously computed} \\
&\quad \quad \text{return stored value } \text{fib}(n) \\
&\quad \text{else} \\
&\quad \quad \text{return } \text{fib}(n-1) + \text{fib}(n-2)
\end{align*}
\]
**question:** how to memoize exactly?

```python
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
        return fib(n-1) + fib(n-2)
```

**question:**
**question:** how to memoize exactly?

```python
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
        return fib(n-1) + fib(n-2)
```

**question:** how to memoize exactly?
**question:** how to memoize exactly?

```
fib(n):
  if n = 0
    return 0
  if n = 1
    return 1
  if fib(n) was previously computed
    return stored value fib(n)
  else
    return fib(n - 1) + fib(n - 2)
```

**question:** how to memoize exactly?

- *explicitly:*
  - just do it!
  - allow clever data structures to do this automatically
question: how to memoize exactly?

```
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
        return fib(n−1) + fib(n−2)
```

question: how to memoize exactly?

- explicitly: just do it!
**question:** how to memoize exactly?

```python
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
        return fib(n−1) + fib(n−2)
```

**question:** how to memoize exactly?

- *explicitly:* just do it!
- *implicitly:*
question: how to memoize exactly?

```python
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
        return fib(n-1) + fib(n-2)
```

question: how to memoize exactly?

- **explicitly**: just do it!
- **implicitly**: allow clever data structures to do this automatically
Memoization (III)

Global Memoization:
\[
F[n] = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F[n] & \text{if } F[n] \text{ initialized} \\
F[n] = F[n-1] + F[n-2] & \text{else} \\
\end{cases}
\]

Explicit Memoization:
- We decide ahead of time what types of objects \( F \) stores.
- E.g., \( F \) is an array.
- Requires more deliberation on problem structure, but can be more efficient.

Implicit Memoization:
- We let the data structure for \( F \) handle whatever comes its way.
- E.g., \( F \) is a dictionary.
- Requires less deliberation on problem structure, and can be less efficient.
- Sometimes can be done automatically by functional programming languages (LISP, etc.).
Memoization (III)

```plaintext
global F[·]
```

explicit memoization: we decide ahead of time what types of objects F stores e.g., F is an array requires more deliberation on problem structure, but can be more efficient

implicit memoization: we let the data structure for F handle whatever comes its way e.g., F is an dictionary requires less deliberation on problem structure, and can be less efficient sometimes can be done automatically by functional programming languages (LISP, etc.)
Memoization (III)

```python
global F[:]
fib(n):
```

explicit memoization: we decide ahead of time what types of objects \( F \) stores e.g., \( F \) is an array requires more deliberation on problem structure, but can be more efficient

implicit memoization: we let the data structure for \( F \) handle whatever comes its way e.g., \( F \) is an dictionary requires less deliberation on problem structure, and can be less efficient sometimes can be done automatically by functional programming languages (LISP, etc.)
global $F[\cdot]$

```python
def fib(n):
    if $n = 0$
        return 0
    if $n = 1$
        return 1
```

explicit memoization: we decide ahead of time what types of objects $F$ stores e.g., $F$ is an array

requires more deliberation on problem structure, but can be more efficient

implicit memoization: we let the data structure for $F$ handle whatever comes its way e.g., $F$ is a dictionary

requires less deliberation on problem structure, and can be less efficient

sometimes can be done automatically by functional programming languages (LISP, etc.)
Memoization (III)

```python
global F[]

fib(n):
    if n == 0
        return 0
    if n == 1
        return 1
    if F[n] initialized
        return F[n]
    F[n] = fib(n-1) + fib(n-2)
    return F[n]
```

**Explicit memoization:**
We decide ahead of time what types of objects `F` stores. For example, `F` is an array, which requires more deliberation on problem structure, but can be more efficient.

**Implicit memoization:**
We let the data structure for `F` handle whatever comes its way. For example, `F` is a dictionary, which requires less deliberation on problem structure, and can be less efficient. Sometimes can be done automatically by functional programming languages (LISP, etc.).
**Memoization (III)**

```python
# global F[]

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif F[n] initialized:
        return F[n]
    else:
        F[n] = fib(n-1) + fib(n-2)
        return F[n]
```
global F[:]
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else

global F[

fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
global F[

fib(n):
    if n == 0
        return 0
    if n == 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
global F[·]
fib(n):
    if $n = 0$
        return 0
    if $n = 1$
        return 1
    if $F[n]$ initialized
        return $F[n]$
    else
        $F[n] = \text{fib}(n - 1) + \text{fib}(n - 2)$
        return $F[n]$

explicit memoization: we decide ahead of time what types of objects $F$ stores e.g., $F$ is an array requires more deliberation on problem structure, but can be more efficient

implicit memoization: we let the data structure for $F$ handle whatever comes its way e.g., $F$ is an dictionary requires less deliberation on problem structure, and can be less efficient

sometimes can be done automatically by functional programming languages (LISP, etc.)
Memoization (III)

```python
global F[

fib(n):
    if n == 0
        return 0
    if n == 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
    return F[n]
```

- *explicit* memoization:
Memoization (III)

```
global F[

fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```

- **explicit memoization**: we decide ahead of time what types of objects $F$ stores
global F[

fib(n):
    if \( n = 0 \)
        return 0
    if \( n = 1 \)
        return 1
    if \( F[n] \) initialized
        return \( F[n] \)
    else
        \( F[n] = \text{fib}(n - 1) + \text{fib}(n - 2) \)
        return \( F[n] \)

- \textit{explicit} memoization: we decide \textit{ahead} of time what types of objects \( F \) stores
  - e.g., \( F \) is an array

- \textit{implicit} memoization: we let the data structure for \( F \) handle whatever comes its way
  - e.g., \( F \) is a dictionary
  - requires less deliberation on problem structure, and can be less efficient
  - sometimes can be done automatically by functional programming languages (LISP, etc.)
Memoization (III)

```python
global F[:]
fib(n):
    if n == 0
        return 0
    if n == 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```

- **explicit** memoization: we decide *ahead* of time what types of objects $F$ stores
  - e.g., $F$ is an array
  - requires more deliberation on problem structure,

- **implicit** memoization: we let the data structure for $F$ handle whatever comes its way
  - e.g., $F$ is a dictionary
  - requires less deliberation on problem structure, and can be less efficient
  - sometimes can be done automatically by functional programming languages (LISP, etc.)
```
global F[

fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```
Memoization (III)

```python
global F[

fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```

- **explicit memoization**: we decide *ahead* of time what types of objects $F$ stores
  - e.g., $F$ is an array
  - requires more deliberation on problem structure, but can be more efficient

- **implicit memoization**:
Memoization (III)

```
global F[]
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```

- **explicit** memoization: we decide *ahead* of time what types of objects $F$ stores
  - e.g., $F$ is an array
  - requires more deliberation on problem structure, but can be more efficient

- **implicit** memoization: we let the data structure for $F$ handle whatever comes its way
  - requires less deliberation on problem structure, and can be less efficient
  - sometimes can be done automatically by functional programming languages (LISP, etc.)
Memoization (III)

```
global F[

fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```

- **explicit** memoization: we decide *ahead of time* what types of objects $F$ stores
  - e.g., $F$ is an array
  - requires more deliberation on problem structure, but can be more efficient
- **implicit** memoization: we let the data structure for $F$ handle whatever comes its way
  - e.g., $F$ is a dictionary
Memoization (III)

```python
global F[:]
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```

- **explicit memoization**: we decide ahead of time what types of objects $F$ stores
  - e.g., $F$ is an array
  - requires more deliberation on problem structure, but can be more efficient
- **implicit memoization**: we let the data structure for $F$ handle whatever comes its way
  - e.g., $F$ is an dictionary
  - requires less deliberation on problem structure,
```
global F[

fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]
```

- **explicit memoization**: we decide *ahead of time* what types of objects $F$ stores
  - e.g., $F$ is an array
  - requires more deliberation on problem structure, but can be more efficient

- **implicit memoization**: we let the data structure for $F$ handle whatever comes its way
  - e.g., $F$ is an dictionary
  - requires *less* deliberation on problem structure, and can be less efficient
global F[

fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n - 1) + fib(n - 2)
        return F[n]

] explicit memoization: we decide ahead of time what types of objects F stores
    ■ e.g., F is an array
    ■ requires more deliberation on problem structure, but can be more efficient

] implicit memoization: we let the data structure for F handle whatever comes its way
    ■ e.g., F is an dictionary
    ■ requires less deliberation on problem structure, and can be less efficient
    ■ sometimes can be done automatically by functional programming languages (LISP, etc.)
question: how much space do we need to memoize?

\[ \text{fib-iter}(n) : \]

\[
\begin{align*}
\text{if } n &= 0 \\
\quad &\text{return } 0 \\
\text{prevprev} &= 0 \\
\text{if } n &= 1 \\
\quad &\text{return } 1 \\
\text{prev} &= 1 \\
\text{for } 2 \leq i \leq n \\
\quad &\text{\textcolor{red}{\text{\textbf{F}cur} = \text{\textcolor{green}{\textbf{F}prev} + \text{\textcolor{blue}{\textbf{F}prevprev}}}}} \\
\text{prev} &= \text{cur} \\
\text{prevprev} &= \text{prev} \\
\text{return } \text{cur}
\end{align*}
\]

correctness:
clear

complexity:
= \Omega(1) additions, \Omega(1) numbers stored
\Rightarrow \Omega(n) \text{ bits stored}
question: how much space do we need to memoize?
question: how much \textit{space} do we need to memoize?

\begin{verbatim}
  \texttt{fib-iter}(n):
  \begin{verbatim}
  if \texttt{n} = 0 return 0
  \texttt{F}_{prevprev} = 0
  if \texttt{n} = 1 return 1
  \texttt{F}_{prev} = 1
  for 2 \leq i \leq \texttt{n} \texttt{F}_{cur} = \texttt{F}_{prev} + \texttt{F}_{prevprev}
  \texttt{F}_{prevprev} = \texttt{F}_{prev}
  \texttt{F}_{prev} = \texttt{F}_{cur}
  return \texttt{F}_{cur}
  \end{verbatim}
\end{verbatim}

\textbf{correctness:} clear

\textbf{complexity:} \textit{O}(\texttt{n}) additions, \textit{O}(1) numbers stored = \textit{O}(\texttt{n}) bits stored
question: how much space do we need to memoize?

fib-iter(n):
    if $n = 0$
        return 0
**question:** how much *space* do we need to memoize?

```
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    F_prev = 1
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```
**question:** how much space do we need to memoize?

```python
def fib_iter(n):
    if n == 0
        return 0
    F_prevprev = 0
    if n == 1
        return 1
```
question: how much space do we need to memoize?

fib-iter(n):
    if $n = 0$
        return 0
    $F_{prevprev} = 0$
    if $n = 1$
        return 1
    $F_{prev} = 1$
**question:** how much *space* do we need to memoize?

```
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    if n = 1
        return 1
    F_prev = 1
    for 2 ≤ i ≤ n
```
question: how much \textit{space} do we need to memoize?

\begin{verbatim}
fib-iter(n):
    if \( n = 0 \)
        \textbf{return} 0
    \( F_{\text{prevprev}} = 0 \)
    if \( n = 1 \)
        \textbf{return} 1
    \( F_{\text{prev}} = 1 \)
    \textbf{for} \( 2 \leq i \leq n \)
        \( F_{\text{cur}} = F_{\text{prev}} + F_{\text{prevprev}} \)
\end{verbatim}
**question:** how much *space* do we need to memoize?

```python
def fib_iter(n):
    if n == 0
        return 0
    F_prevprev = 0
    if n == 1
        return 1
    F_prev = 1
    for 2 <= i <= n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

correctness:
complexity: $O(n)$ additions, $O(1)$ numbers stored $\Rightarrow O(n)$ bits stored
question: how much space do we need to memoize?

fib-iter(n):
    if $n = 0$
        return 0
    $F_{prevprev} = 0$
    if $n = 1$
        return 1
    $F_{prev} = 1$
    for $2 \leq i \leq n$
        $F_{cur} = F_{prev} + F_{prevprev}$
        $F_{prevprev} = F_{prev}$
        $F_{prev} = F_{cur}$

correctness:
clear
complexity:
$O(n)$ additions, $O(1)$ numbers stored $\Rightarrow O(n)$ bits stored
**question:** how much *space* do we need to memoize?

```python
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    if n = 1
        return 1
    F_prev = 1
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

**correctness:**

**complexity:** $O(n)$ additions, $O(1)$ numbers stored $⇒ O(n)$ bits stored
**question:** how much space do we need to memoize?

```python
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    if n = 1
        return 1
    F_prev = 1
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

**correctness:** self-explanatory

**complexity:** $O(n)$ additions, $O(1)$ numbers stored $⇒ O(n)$ bits stored
question: how much *space* do we need to memoize?

```python
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    if n = 1
        return 1
    F_prev = 1
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

correctness:
**question:** how much *space* do we need to memoize?

```python
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    if n = 1
        return 1
    F_prev = 1
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

**correctness:** clear
question: how much space do we need to memoize?

fib-iter(n):
  if \( n = 0 \)
    return 0
  \( F_{\text{prevprev}} = 0 \)
  if \( n = 1 \)
    return 1
  \( F_{\text{prev}} = 1 \)
  for \( 2 \leq i \leq n \)
    \( F_{\text{cur}} = F_{\text{prev}} + F_{\text{prevprev}} \)
    \( F_{\text{prevprev}} = F_{\text{prev}} \)
    \( F_{\text{prev}} = F_{\text{cur}} \)
  return \( F_{\text{cur}} \)

correctness: clear
complexity:
question: how much space do we need to memoize?

fib-iter(n):
  if n = 0
    return 0
  F_prevprev = 0
  if n = 1
    return 1
  F_prev = 1
  for 2 ≤ i ≤ n
    F_cur = F_prev + F_prevprev
    F_prevprev = F_prev
    F_prev = F_cur
  return F_cur

correctness: clear
complexity: $O(n)$ additions,
question: how much space do we need to memoize?

```python
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    if n = 1
        return 1
    F_prev = 1
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

correctness: clear

complexity: $O(n)$ additions, $O(1)$ numbers stored
**question:** how much *space* do we need to memoize?

```python
fib-iter(n):
    if n = 0
        return 0
    F_prevprev = 0
    if n = 1
        return 1
    F_prev = 1
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

**correctness:** clear

**complexity:** $O(n)$ additions, $O(1)$ numbers stored $\implies O(n)$ bits stored
Memoization (IV)

Definition

Dynamic programming is the method of speeding up naive recursion through memoization.

Goals:
- Given a recursive algorithm, analyze the complexity of its memoized version.
- Find the right recursion that can be memoized.
- Recognize when dynamic programming will efficiently solve a problem.
- Further optimize time- and space-complexity of dynamic programming algorithms.
Memoization (IV)

Definition

**Dynamic programming** is the method of speeding up naive recursion through memoization.
Definition

**Dynamic programming** is the method of speeding up naive recursion through memoization.

goals:
**Memoization (IV)**

**Definition**

*Dynamic programming* is the method of speeding up naive recursion through memoization.

**goals:**

- Given a recursive algorithm,
**Definition**

**Dynamic programming** is the method of speeding up naive recursion through memoization.

**goals:**
- Given a recursive algorithm, analyze the complexity of its memoized version.
Memoization (IV)

**Definition**

**Dynamic programming** is the method of speeding up naive recursion through memoization.

**goals:**

- Given a recursive algorithm, analyze the complexity of its memoized version.
- Find the *right* recursion that can be memoized.
Definition

**Dynamic programming** is the method of speeding up naive recursion through memoization.

goals:

- Given a recursive algorithm, analyze the complexity of its memoized version.
- Find the *right* recursion that can be memoized.
- Recognize when dynamic programming will efficiently solve a problem.
Definition

**Dynamic programming** is the method of speeding up naive recursion through memoization.

**goals:**

- Given a recursive algorithm, analyze the complexity of its memoized version.
- Find the *right* recursion that can be memoized.
- Recognize when dynamic programming will efficiently solve a problem.
- Further optimize time- and space-complexity of dynamic programming algorithms.
Edit Distance

Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example\[=\Rightarrow\]edit distance $\leq 5$

Remarks: edit distance $\leq 4$ intermediate strings can be arbitrary in $\Sigma^*$
Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example $= \Rightarrow$ edit distance $\leq 5$

Remarks:
- edit distance $\leq 4$
- intermediate strings can be arbitrary in $\Sigma^\star$
**Definition**

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$. Remarks: Edit distance $\leq 4$; intermediate strings can be arbitrary in $\Sigma^*$. 

| 15 / 33 |
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$

Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$. 

*Example*: $\Rightarrow$ edit distance $\leq 5$

**Remarks**: edit distance $\leq 4$; intermediate strings can be arbitrary in $\Sigma^*$. 

15 / 33
**Definition**

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). The **edit distance** between \( x \) and \( y \) is the minimum number of insertions, deletions and substitutions required to transform \( x \) into \( y \).

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Edit Distance

Definition
Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example
money
**Definition**

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). The **edit distance** between \( x \) and \( y \) is the minimum number of insertions, deletions and substitutions required to transform \( x \) into \( y \).

**Example**

money →
Definition

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). The \textbf{edit distance} between \( x \) and \( y \) is the minimum number of insertions, deletions and substitutions required to transform \( x \) into \( y \).

Example

\( \text{money} \rightarrow \text{boney} \)
Definition
Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example

$\text{money} \rightarrow \text{boney} \rightarrow$
Definition

Let \(x, y \in \Sigma^*\) be two strings over the alphabet \(\Sigma\). The edit distance between \(x\) and \(y\) is the minimum number of insertions, deletions and substitutions required to transform \(x\) into \(y\).

Example

\[
\text{money} \rightarrow \text{boney} \rightarrow \text{bone}
\]
### Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

### Example

$\text{money} \rightarrow \text{boney} \rightarrow \text{bone} \rightarrow$
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example

$\underline{\text{money}} \rightarrow \underline{\text{boney}} \rightarrow \underline{\text{bone}} \rightarrow \text{bona}$
**Definition**

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

**Example**

```plaintext
money → boney → bone → bona →
```
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example

money $\rightarrow$ boney $\rightarrow$ bone $\rightarrow$ bona $\rightarrow$ boa
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example

```
  money  →  boney  →  bone  →  bona  →  bo_a  →
```
Definition

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). The **edit distance** between \( x \) and \( y \) is the minimum number of insertions, deletions and substitutions required to transform \( x \) into \( y \).

Example

\[
\begin{align*}
\text{money} & \rightarrow \text{boney} \rightarrow \text{bone} \rightarrow \text{bona} \rightarrow \text{bo_a} \rightarrow \text{boba}
\end{align*}
\]
## Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$ is the minimum number of insertions, deletions, and substitutions required to transform $x$ into $y$.

## Example

$\_\text{money} \rightarrow \_\text{boney} \rightarrow \_\text{bone} \rightarrow \_\text{bona} \rightarrow \_\text{bo_a} \rightarrow \_\text{boba} \implies \text{edit distance} \leq 5$
Edit Distance

Definition
Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example

$\text{money} \rightarrow \text{boney} \rightarrow \text{bone} \rightarrow \text{bona} \rightarrow \text{bo_a} \rightarrow \text{boba} \implies \text{edit distance } \leq 5$

remarks:
Edit Distance

**Definition**

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The **edit distance** between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

**Example**

$\underline{\text{money}} \rightarrow \underline{\text{boney}} \rightarrow \underline{\text{bone}} \rightarrow \underline{\text{bona}} \rightarrow \text{bo}_a \rightarrow \text{boba} \implies \text{edit distance} \leq 5$

**remarks:**

- edit distance $\leq 4$
Definition

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). The **edit distance** between \( x \) and \( y \) is the minimum number of insertions, deletions and substitutions required to transform \( x \) into \( y \).

Example

- money \( \rightarrow \) boney \( \rightarrow \) bone \( \rightarrow \) bona \( \rightarrow \) bo_a \( \rightarrow \) boba \( \implies \) edit distance \( \leq 5 \)

**remarks:**

- edit distance \( \leq 4 \)
- intermediate strings can be arbitrary in \( \Sigma^* \)
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An alignment is a sequence $M$ of pairs of indices $(i, j)$ such that an index could be empty, such as $(1, 4)$ or $(5, 2)$ each index appears exactly once per coordinate no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$, or $i > i'$ and $j > j'$.

The cost of an alignment is the number of pairs $(i, j)$ where $x_i \neq y_j$.

Example

```
mon ey
bo ba
```

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

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An alignment is a sequence $M$ of pairs of indices $(i, j)$ such that an index could be empty, such as $(, 4)$ or $(5, )$, each index appears exactly once per coordinate. The cost of an alignment is the number of pairs $(i, j)$ where $x_i \neq y_j$.

Example: $\text{money}$ $\text{bo ba}$

$M = \{(1, 1), (2, 2), (3, ), (, 3), (4, 4), (5, )\}$, cost $= 5$
Definition
Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. 

Example: 
$\text{mon ey}$ $\text{bo ba}$ 
$M = \{(1,1), (2,2), (3,), (4,4), (5,\)}$, cost $5$
Definition

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). An **alignment** is a sequence \( M \) of pairs of indices \((i, j)\) such that

\[
\text{cost} \text{ of an alignment is the number of pairs } (i, j) \text{ where } x_i \neq y_j.
\]
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An **alignment** is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty,
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An alignment is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5, )$

The cost of an alignment is the number of pairs $(i, j)$ where $x_i \neq y_j$. 

Example

```
mon ey
bo ba
```

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

cost $= 5$
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An alignment is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5, )$
- each index appears exactly once per coordinate

The cost of an alignment is the number of pairs $(i, j)$ where $x_i \neq y_j$. 

Example: 

```
monkey
boba
```

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

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). An alignment is a sequence \( M \) of pairs of indices \( (i, j) \) such that

- an index could be empty, such as \((, 4)\) or \((5, )\)
- each index appears exactly once per coordinate
- no crossings:

\[
M = \{(1, 1), (2, 2), (3, ), (4, 4), (5, )\},
\]

Cost 5
**Definition**

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An **alignment** is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5,)$
- each index appears exactly once per coordinate
- no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$,
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An **alignment** is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5, )$
- each index appears exactly once per coordinate
- no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$, or
Edit Distance (II)

Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An **alignment** is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5, )$
- each index appears exactly once per coordinate
- no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$, or $i > i'$ and $j > j'$
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An alignment is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5, )$
- each index appears exactly once per coordinate
- no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$, or $i > i'$ and $j > j'$

The cost of an alignment is the number of pairs $(i, j)$ where $x_i \neq y_j$. 

Example

\begin{align*}
\text{money} & & \text{bo ba} \\
\mathcal{M} & = & \{(1, 1), (2, 2), (3), (4, 4), (5, )\} \\
\text{cost} & = & \frac{5}{16}/33
\end{align*}
Edit Distance (II)

Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An alignment is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5, )$
- each index appears exactly once per coordinate
- no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$, or $i > i'$ and $j > j'$

The cost of an alignment is the number of pairs $(i, j)$ where $x_i \neq y_j$.

Example

mon ey
bo ba
**Definition**

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). An **alignment** is a sequence \( M \) of pairs of indices \((i, j)\) such that

- an index could be empty, such as \((, 4)\) or \((5, )\)
- each index appears exactly once per coordinate
- no crossings: for \((i, j), (i', j') \in M\) either \(i < i'\) and \(j < j'\), or \(i > i'\) and \(j > j'\)

The **cost** of an alignment is the number of pairs \((i, j)\) where \(x_i \neq y_j\).

**Example**

```
mon ey
bo ba
```

\[
M = \{(1, 1), (2, 2), (3, ), (, 3), (4, 4), (5, )\},
\]
### Definition

Let \( x, y \in \Sigma^* \) be two strings over the alphabet \( \Sigma \). An alignment is a sequence \( M \) of pairs of indices \((i,j)\) such that

- an index could be empty, such as \((,4)\) or \((5,)\)
- each index appears exactly once per coordinate
- no crossings: for \((i,j),(i',j')\) \(\in M\) either \(i < i'\) and \(j < j'\), or \(i > i'\) and \(j > j'\)

The **cost** of an alignment is the number of pairs \((i,j)\) where \(x_i \neq y_j\).

### Example

<table>
<thead>
<tr>
<th>money</th>
<th>bo ba</th>
</tr>
</thead>
</table>

\[ M = \{(1, 1), (2, 2), (3, ), (, 3), (4, 4), (5, )\}, \text{ cost 5} \]
question: given two strings $x, y \in \Sigma^*$, compute their edit distance.

Lemma: The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.

Proof. Exercise.

question: given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment.

remarks: can also ask to compute the alignment itself. Widely solved in practice, e.g., the BLAST heuristic for DNA edit distance.
question:

given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.

Proof.

Exercise.

question:

given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment

remarks:

can also ask to compute the alignment itself

widely solved in practice, e.g., the BLAST heuristic for DNA edit distance
question: given two strings $x, y \in \Sigma^*$,
**question:** given two strings $x, y \in \Sigma^*$, compute their edit distance
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.

Proof.
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

*The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.*

Proof.

Exercise.
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.

Proof.

Exercise.

question: given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.

Proof.

Exercise.

question: given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment

remarks:
**question:** given two strings $x, y \in \Sigma^*$, compute their edit distance

**Lemma**

*The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.*

**Proof.**

Exercise.

**question:** given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment

**remarks:**

- can also ask to compute the alignment itself
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

*The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.*

Proof.

Exercise.

question: given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment

remarks:

- can also ask to compute the alignment itself
- widely solved in practice,
**question:** given two strings $x, y \in \Sigma^*$, compute their edit distance

**Lemma**

*The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.*

**Proof.**

Exercise.

**question:** given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment

**remarks:**

- can *also* ask to compute the alignment itself
- widely solved in practice, e.g., the BLAST heuristic for DNA edit distance
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \cdot a, y \cdot b) = \min \begin{cases} \text{dist}(x, y) + 1 \\ J_{a \neq b} \\ \text{dist}(x, y \cdot b) + 1 \\ \text{dist}(x \cdot a, y) + 1 \end{cases}$$

Proof.

In an optimal alignment from $x \cdot a$ to $y \cdot b$, either:

- $a$ aligns to $b$, with cost $1$.
- $a$ is deleted, with cost $1$.
- $b$ is deleted, with cost $1$.
Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ symbols. Then
\[
\text{dist}(x \circ a, y \circ b) = \min \begin{cases}
\text{dist}(x, y) + 1 & \text{if } a \neq b \\
\text{dist}(x, y \circ b) + 1 & \\
\text{dist}(x \circ a, y) + 1 & 
\end{cases}
\]

Proof. In an optimal alignment from $x \circ a$ to $y \circ b$, either:
- $a$ aligns to $b$, with cost 1
- $a$ is deleted, with cost 1
- $b$ is deleted, with cost 1
Lemma

Let $x, y \in \Sigma^*$ be strings,
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols.
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \begin{cases} 
\text{dist}(x, y) + 1 \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1 
\end{cases}$$

Proof. In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1$
- $a$ is deleted, with cost $1$
- $b$ is deleted, with cost $1$
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \begin{cases} \text{dist}(x, y) + 1, & \text{if } a \neq b \\ \text{dist}(x, y \circ b) + 1, & \text{if } a \neq b \\ \text{dist}(x \circ a, y) + 1, & \text{if } a \neq b \end{cases}$$

Proof.
In an optimal alignment from $x \circ a$ to $y \circ b$, either:
- $a$ aligns to $b$, with cost 1
- $a$ is deleted, with cost 1
- $b$ is deleted, with cost 1
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) =$$
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{l}
\text{dist}(x, y) + 1 \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1
\end{array} \right\}$$

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1$ if $a \neq b$;
- $a$ is deleted, with cost $1$;
- $b$ is deleted, with cost $1$. 

Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{l}
\text{dist}(x, y) + 1[a \neq b] \\
\text{dist}(x \circ a, y) + 1 \\
\text{dist}(x, y \circ b) + 1 
\end{array} \right\}$$
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$
\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{ll}
\text{dist}(x, y) + 1 & [a \neq b] \\
\text{dist}(x, y \circ b) + 1 & 
\end{array} \right.
$$

Proof. In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1$ if $a \neq b$,
- $a$ is deleted, with cost $1$,
- $b$ is deleted, with cost $1$.
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$
\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{l}
\text{dist}(x, y) + 1[a \neq b] \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1 \\
\end{array} \right. 
$$
Lemma

Let \( x, y \in \Sigma^* \) be strings, and \( a, b \in \Sigma \) be symbols. Then

\[
\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{ll}
\text{dist}(x, y) + 1 & [a \neq b] \\
\text{dist}(x, y \circ b) + 1 & \\
\text{dist}(x \circ a, y) + 1 & 
\end{array} \right.
\]

Proof.
Lemma

Let \( x, y \in \Sigma^* \) be strings, and \( a, b \in \Sigma \) be symbols. Then

\[
\text{dist}(x \circ a, y \circ b) = \min \begin{cases} \text{dist}(x, y) + 1[a \neq b] \\ \text{dist}(x, y \circ b) + 1 \\ \text{dist}(x \circ a, y) + 1 \end{cases}
\]

Proof.

In an optimal alignment from \( x \circ a \) to \( y \circ b \), either:
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{ll}
\text{dist}(x, y) + 1 \mathbb{I}[a \neq b] \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1
\end{array} \right\}.$$ 

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, 

...
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{c} \text{dist}(x, y) + 1 [a \neq b] \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1 \end{array} \right. .$$

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1 [a \neq b]$
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \begin{cases} \text{dist}(x, y) + 1[a \neq b] \\ \text{dist}(x, y \circ b) + 1 \\ \text{dist}(x \circ a, y) + 1 \end{cases}.$$ 

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1[a \neq b]$
- $a$ is deleted,
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$
\text{dist}(x \circ a, y \circ b) = \min \begin{cases} 
\text{dist}(x, y) + 1 \mathbbm{1}[a \neq b] \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1
\end{cases}
$$

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1 \mathbbm{1}[a \neq b]$
- $a$ is deleted, with cost 1
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$
\text{dist}(x \circ a, y \circ b) = \min \begin{cases} 
\text{dist}(x, y) + 1[\text{a} \neq \text{b}] \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1
\end{cases}
$$

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1[\text{a} \neq \text{b}]$
- $a$ is deleted, with cost 1
- $b$ is deleted,
Lemma

Let \( x, y \in \Sigma^* \) be strings, and \( a, b \in \Sigma \) be symbols. Then

\[
\dist(x \circ a, y \circ b) = \min \begin{cases} 
\dist(x, y) + 1 \mathbb{1}[a \neq b] \\
\dist(x, y \circ b) + 1 \\
\dist(x \circ a, y) + 1
\end{cases}
\]

Proof.

In an optimal alignment from \( x \circ a \) to \( y \circ b \), either:

- \( a \) aligns to \( b \), with cost \( 1 \mathbb{1}[a \neq b] \)
- \( a \) is deleted, with cost 1
- \( b \) is deleted, with cost 1
Edit Distance (V)

The recursive algorithm for Edit Distance is:

\[
\text{dist}(x_1x_2\ldots x_n, y_1y_2\ldots y_m) =
\begin{cases}
  m & \text{if } n = 0, \\
  n & \text{if } m = 0, \\
  \text{dist}(x_1x_2\ldots x_{n-1}, y_1y_2\ldots y_m) + 1 & \text{if } x_n \neq y_m, \\
\end{cases}
\]

Correctness:

Complexity: ???
recursive algorithm:
recursive algorithm:

\[ \text{dist}(x_1, y_1, \ldots, x_n, y_m) \]

**correctness:**

**complexity:**
recursive algorithm:

\[
\text{dist}(x = x_1x_2\cdots x_n, y = y_1y_2\cdots y_m) = \\
\begin{cases}
0 & \text{if } n = 0, \\
0 & \text{if } m = 0, \\
\text{dist}(x_{<n}, y_{<m}) + 1 & \text{if } x_n \neq y_m, \\
\text{dist}(x_{<n}, y_{<m}) & \text{otherwise}.
\end{cases}
\]
recursive algorithm:

\[ \text{dist}(x = x_1x_2 \ldots x_n, y) \]
Edit Distance (V)

**recursive algorithm:**

\[
\text{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)
\]
recursion algorithm:

\[
\text{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)
\]

\[
\text{if } n = 0, \ \text{return } m
\]
**Edit Distance (V)**

**recursive algorithm:**

\[
\text{dist}(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_m) \begin{align*}
\text{if } n = 0, & \quad \text{return } m \\
\text{if } m = 0, & \quad \text{return } n
\end{align*}
\]
recursive algorithm:

\[
\text{dist}(x = x_1x_2\cdots x_n, y = y_1y_2\cdots y_m) \\
\quad \text{if } n = 0, \ \text{return } m \\
\quad \text{if } m = 0, \ \text{return } n \\
\quad d_1 = \text{dist}(x_{<n}, y_{<m}) + \mathbb{1}[x_n \neq y_m]
\]
recursive algorithm:

\[
\text{dist}(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_m)
\]
\[
\text{if } n = 0, \text{ return } m
\]
\[
\text{if } m = 0, \text{ return } n
\]
\[
d_1 = \text{dist}(x_{< n}, y_{< m}) + 1 [x_n \neq y_m]
\]
\[
d_2 = \text{dist}(x_{< n}, y) + 1
\]

correctness:
clear

complexity:
???
recursive algorithm:

\[
\text{dist}(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_m) \\
\quad \text{if } n = 0, \text{ return } m \\
\quad \text{if } m = 0, \text{ return } n \\
\quad d_1 = \text{dist}(x_{<n}, y_{<m}) + 1[ x_n \neq y_m ] \\
\quad d_2 = \text{dist}(x_{<n}, y) + 1 \\
\quad d_3 = \text{dist}(x, y_{<m}) + 1 \\
\]

return min(d_1, d_2, d_3)
recursive algorithm:

\[
\text{dist}(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_m) \\
\text{if } n = 0, \ \text{return } m \\
\text{if } m = 0, \ \text{return } n \\
d_1 = \text{dist}(x_{<n}, y_{<m}) + 1[x_n \neq y_m] \\
d_2 = \text{dist}(x_{<n}, y) + 1 \\
d_3 = \text{dist}(x, y_{<m}) + 1 \\
\text{return } \min(d_1, d_2, d_3)
\]
Edit Distance (V)

recursive algorithm:

\[
\text{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)
\]

\[
\begin{align*}
\text{if } n &= 0, \quad \text{return } m \\
\text{if } m &= 0, \quad \text{return } n \\
\text{if } x_n \neq y_m \quad &d_1 = \text{dist}(x_{<n}, y_{<m}) + 1 \\
\text{dist}(x_{<n}, y) + 1 &d_2 = \\
\text{dist}(x, y_{<m}) + 1 &d_3 = \\
\text{return } \min(d_1, d_2, d_3)
\end{align*}
\]
recursive algorithm:

\[
\text{dist}(x = x_1x_2\cdots x_n, y = y_1y_2\cdots y_m) \\
\quad \text{if } n = 0, \ \text{return} \ m \\
\quad \text{if } m = 0, \ \text{return} \ n \\
\quad d_1 = \text{dist}(x_{<n}, y_{<m}) + 1 [x_n \neq y_m] \\
\quad d_2 = \text{dist}(x_{<n}, y) + 1 \\
\quad d_3 = \text{dist}(x, y_{<m}) + 1 \\
\quad \text{return} \ \min(d_1, d_2, d_3)
\]

correctness:
Edit Distance (V)

recursive algorithm:

\[
\begin{align*}
\text{dist}(x = x_1x_2 \cdots x_n, y = y_1y_2 \cdots y_m) &= \quad \\
\text{if} \ n = 0, \ \text{return} \ m & \\
\text{if} \ m = 0, \ \text{return} \ n & \\
d_1 &= \text{dist}(x\lessdot n, y\lessdot m) + 1 \mathbb{1}[x_n \neq y_m] \\
d_2 &= \text{dist}(x\lessdot n, y) + 1 & \\
d_3 &= \text{dist}(x, y\lessdot m) + 1 & \\
\text{return} \ \min(d_1, d_2, d_3) & \\
\end{align*}
\]

correctness: clear
Edit Distance (V)

**recursive algorithm:**

\[
\text{dist}(x = x_1x_2\cdots x_n, y = y_1y_2\cdots y_m) \\
\quad \text{if } n = 0, \text{ return } m \\
\quad \text{if } m = 0, \text{ return } n \\
\quad d_1 = \text{dist}(x_{<n}, y_{<m}) + 1[ x_n \neq y_m ] \\
\quad d_2 = \text{dist}(x_{<n}, y) + 1 \\
\quad d_3 = \text{dist}(x, y_{<m}) + 1 \\
\quad \text{return } \min(d_1, d_2, d_3)
\]

**correctness:** clear

**complexity:**
recursive algorithm:

\[
\text{dist}(x = x_1x_2\cdots x_n, y = y_1y_2\cdots y_m)
\]
\[
\text{if } n = 0, \text{ return } m
\]
\[
\text{if } m = 0, \text{ return } n
\]
\[
d_1 = \text{dist}(x_{< n}, y_{< m}) + 1\mathbb{I}[x_n \neq y_m]
\]
\[
d_2 = \text{dist}(x_{< n}, y) + 1
\]
\[
d_3 = \text{dist}(x, y_{< m}) + 1
\]
\[
\text{return } \min(d_1, d_2, d_3)
\]

correctness: clear
complexity: ???
Edit Distance (VI)

... is repeated!

Memoization: define subproblem \((i, j)\) as computing dist\((x \leq i, y \leq j)\)
Edit Distance (VI)

(abab, baba)
Edit Distance (VI)

- (aba, bab)
- (abab, baba)

...
Edit Distance (VI)

(a, b) is repeated!

Memoization: define subproblem \((i, j)\) as computing \(d\) \((x \leq i, y \leq j)\)
Edit Distance (VI)

\[(abab, baba)\]

\[(aba, bab)\] (aba, baba) (bab, bab)
Edit Distance (VI)

- (ab, ba)

  - (aba, bab)
- (aba, baba)

  - (abab, baba)

  - (abab, bab)
- (aba, bab)

  - (abab, bab)
Edit Distance (VI)

$$\begin{align*}
\text{(abab, baba)} \\
\text{(aba, bab)} \\
\text{(ab, baba)} \\
\text{(ab, ba)}
\end{align*}$$

Memoization:

Define subproblem \((i, j)\) as computing \(\text{dist}(x \leq i, y \leq j)\)
Edit Distance (VI)

(aba,bab)

(ab,ba) (ab,bab) (aba,ba)

(abab,baba)

(aba,baba) (abab,bab)
Edit Distance (VI)

Edit Distance is a measure of similarity between two sequences, which is defined as the minimum number of operations required to transform one sequence into the other, where operations allowed are insertion, deletion, or substitution of a single element.

The tree diagram illustrates the recursive structure of the edit distance problem. Each node represents a pair of sequences, and the edges represent the possible operations that can be performed to transform one sequence into another.

For example, the pair (abab, baba) is repeated at the top level, indicating that these sequences are at the root of the edit distance calculation.

The process involves breaking down the sequences into smaller subproblems, which are solved recursively until the base case is reached, where the edit distance is zero for identical substrings.

Memoization is a technique used to optimize the computation by caching the results of expensive function calls and reusing them when the same inputs occur again, significantly reducing the computation time for sequences with repeated patterns.
Edit Distance (VI)

Memoization:

Define subproblem \((i, j)\) as computing dist\((x \leq i, y \leq j)\).
(ab, bab) is repeated!
Edit Distance (VI)

(abab, baba)

(aba, bab)  (aba, bab)  (aba, ba)  (ab, bab)  ...

(ab, ba)  (ab, bab)  (aba, ba)  (ab, bab)  ...

(ab, bab) is repeated!

memoization:
(ab,bab) is repeated!

**memoization**: define subproblem \((i, j)\) as computing \(\text{dist}(x_{\leq i}, y_{\leq j})\)
Edit Distance (VII)

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i, j))
\]

if \(d_{i,j}\) initialized

return \(d_{i,j}\)

if \(i = 0\)

\(d_{i,j} = j\)

elif \(j = 0\)

\(d_{i,j} = i\)

else

\(d_1 = \text{dist}(x, y, (i-1, j-1)) + 1\) if \(x_i \neq y_j\)

\(d_2 = \text{dist}(x, y, (i-1, j)) + 1\)

\(d_3 = \text{dist}(x, y, (i, j-1)) + 1\)

\(d_{i,j} = \min(d_1, d_2, d_3)\)

return \(d_{i,j}\)
memoized algorithm:

\[
d[i][j] = \begin{cases} 
\text{if } d[i][j] \text{ initialized} & \text{return } d[i][j] \\
\text{if } i = 0 & \text{return } d[i][j] = j \\
\text{elif } j = 0 & \text{return } d[i][j] = i \\
\text{else} & 
\begin{align*} 
& d_1 = \text{dist}(x, y, (i-1, j-1)) + 1 \\
& \text{if } x_i \neq y_j \\
& d_2 = \text{dist}(x, y, (i-1, j)) + 1 \\
& d_3 = \text{dist}(x, y, (i, j-1)) + 1 \\
& \text{return } d[i][j] = \min(d_1, d_2, d_3) 
\end{align*}
\end{cases}
\]
memoized algorithm:

```
global d[·][·]
```

```latex
\text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i, j))
```

if $d[i][j]$ initialized return $d[i][j]$

if $i = 0$

$d[i][j] = j$

elif $j = 0$

$d[i][j] = i$

else

$d_1 = \text{dist}(x_i, y_j, (i-1, j-1)) + 1$ if $x_i \neq y_j$

$d_2 = \text{dist}(x_i, y_j, (i-1, j)) + 1$

$d_3 = \text{dist}(x_i, y_j, (i, j-1)) + 1$

$d[i][j] = \min(d_1, d_2, d_3)$

return $d[i][j]$
```
memoized algorithm:

```plaintext
global  d[·][·]
dist(x_1x_2⋯x_n, y_1y_2⋯y_m,
```

\[
\begin{align*}
\text{if } d[i][j] \text{ initialized} & \quad \text{return } d[i][j] \\
\text{if } i = 0 & \quad d[i][j] = j \\
\text{elif } j = 0 & \quad d[i][j] = i \\
\text{else} & \quad \begin{align*}
& d_1 = \text{dist}(x_1x_2⋯x_n, y_1y_2⋯y_m, \\
& \quad (i-1, j-1)) + 1 \\
& d_2 = \text{dist}(x_1x_2⋯x_n, y_1y_2⋯y_m, \\
& \quad (i-1, j)) + 1 \\
& d_3 = \text{dist}(x_1x_2⋯x_n, y_1y_2⋯y_m, \\
& \quad (i, j-1)) + 1 \\
\end{align*}
\end{align*}
\]

\[
d[i][j] = \min(d_1, d_2, d_3)
\]

\[\text{return } d[i][j]\]
```
memoized algorithm:

\[ \text{global } d[i][j] \]
\[ \text{dist}(x_1x_2 \cdots x_n, y_1y_2 \cdots y_m, (i,j)) \]
memoized algorithm:

```
global $d[\cdot][\cdot]$

$\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i,j))$

if $d[i][j]$ initialized
```
memoized algorithm:

```python
memoized algorithm:

global d[

```

```python
dist(x_1 \cdots x_n, y_1 y_2 \cdots y_m, (i, j))

if d[i][j] initialized
    return d[i][j]
```
Edit Distance (VII)

memoized algorithm:

```python
global d[:,:,]

dist(x₁x₂⋯xₙ,y₁y₂⋯yₘ,(i,j))
    if d[i][j] initialized
        return d[i][j]
    if i = 0
```
memoized algorithm:

```python
global d[:][:]

dist(x_1, x_2, ..., x_n, y_1, y_2, ..., y_m, (i, j))
    if d[i][j] initialized
        return d[i][j]
    if i = 0
        d[i][j] = j
```

memoized algorithm:

```plaintext
global  \( d[\cdot][\cdot] \)
dist(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i,j))
  if  \( d[i][j] \) initialized
      return  \( d[i][j] \)
  if  \( i = 0 \)
      \( d[i][j] = j \)
  elif  \( j = 0 \)
```
memoized algorithm:

```
global d[:][:]
dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i, j))
    if d[i][j] initialized
        return d[i][j]
    if i = 0
        d[i][j] = j
    elif j = 0
        d[i][j] = i
    d_1 = dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i - 1, j - 1)) + 1
    d_2 = dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i - 1, j)) + 1
    d_3 = dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i, j - 1)) + 1
    d[i][j] = min(d_1, d_2, d_3)
```

memoized algorithm:

```python
global d[i][j]
dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i, j))
    if d[i][j] initialized
        return d[i][j]
    if i = 0
        d[i][j] = j
    elif j = 0
        d[i][j] = i
    else
        d_1 = dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i-1, j-1)) + 1
        d_2 = dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i-1, j)) + 1
        d_3 = dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i, j-1)) + 1
        d[i][j] = min(d_1, d_2, d_3)
return d[i][j]
```
memoized algorithm:

```plaintext
global  d[i][j]

dist(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i,j))
    if  d[i][j] initialized
        return  d[i][j]
    if  i = 0
        d[i][j] = j
    elif  j = 0
        d[i][j] = i
    else
        d_1 = dist(x, y, (i - 1, j - 1)) + 1[x_i \neq y_j]
```

```plaintext
d[i][j] = \min(d_1, d_2, d_3)
return  d[i][j]
```
memoized algorithm:

```plaintext
global d[·]·
dist(x₁x₂⋯xₙ, y₁y₂⋯yₘ, (i,j))
  if d[i][j] initialized
    return d[i][j]
  if i = 0
    d[i][j] = j
  elif j = 0
    d[i][j] = i
  else
    d₁ = dist(x, y, (i - 1, j - 1)) + 1[xᵢ ≠ yⱼ]
    d₂ = dist(x, y, (i - 1, j)) + 1
    d[i][j] = min(d₁, d₂)
```

return d[i][j]
memoized algorithm:

```
global d[:][i]

dist(x_1x_2...x_n, y_1y_2...y_m, (i, j))
  if d[i][j] initialized
    return d[i][j]
  if i = 0
    d[i][j] = j
  elif j = 0
    d[i][j] = i
  else
    d_1 = dist(x, y, (i - 1, j - 1)) + I[x_i \neq y_j]
    d_2 = dist(x, y, (i - 1, j)) + 1
    d_3 = dist(x, y, (i, j - 1)) + 1
    d[i][j] = min(d_1, d_2, d_3)

return d[i][j]
```
memoized algorithm:

```python
global d[:][

dist(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i, j))

if d[i][j] initialized
    return d[i][j]

if i = 0
    d[i][j] = j

elif j = 0
    d[i][j] = i

e else
    d_1 = dist(x, y, (i - 1, j - 1)) + 1_{x_i \neq y_j}
    d_2 = dist(x, y, (i - 1, j)) + 1
    d_3 = dist(x, y, (i, j - 1)) + 1
    d[i][j] = min(d_1, d_2, d_3)
```
memoized algorithm:

```python
global d[:][i]

dist(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i, j))
    if d[i][j] initialized
        return d[i][j]
    if i = 0
        d[i][j] = j
    elif j = 0
        d[i][j] = i
    else
        d_1 = dist(x, y, (i - 1, j - 1)) + \mathbb{1}_{x_i \neq y_j}
        d_2 = dist(x, y, (i - 1, j)) + 1
        d_3 = dist(x, y, (i, j - 1)) + 1
        d[i][j] = \min(d_1, d_2, d_3)
        return d[i][j]
```

memoized algorithm:

```plaintext
global  \( d[\cdot][\cdot] \)

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i,j))
\]

if \( d[i][j] \) initialized
    return \( d[i][j] \)

if \( i = 0 \)
    \( d[i][j] = j \)

elif \( j = 0 \)
    \( d[i][j] = i \)

else
    \( d_1 = \text{dist}(x, y, (i - 1, j - 1)) + 1 \cdot [x_i \neq y_j] \)
    \( d_2 = \text{dist}(x, y, (i - 1, j)) + 1 \)
    \( d_3 = \text{dist}(x, y, (i, j - 1)) + 1 \)
    \( d[i][j] = \min(d_1, d_2, d_3) \)

return \( d[i][j] \)
```

Edit Distance (VIII)
dependency graph:
dependency graph:
dependency graph:

\[
\begin{array}{cc}
\text{n} & \text{m} \\
n-1 & m \\
\end{array}
\]
Edit Distance (VIII)

dependency graph:

```
  n  m
  n-1 m
  n  m-1
```
dependency graph:

\[
\begin{array}{cccc}
 n & n-1 & m & m \\
 m & m-1 & n & n-1 \\
 m-1 & m-1 & n-1 & m \\
\end{array}
\]
dependency graph:
dependency graph:

\[
\begin{array}{ccc}
  n & m & \rightarrow & n-1 & m \\
  \downarrow & & \downarrow & \downarrow & \downarrow \\
  n & m-1 & \rightarrow & n-1 & m-1
\end{array}
\]
dependency graph:

```
  n  m
  ↓  ↓
  n  m−1
  ↓  ↓
  n−1 m−1
```

...
dependency graph:
dependency graph:
dependency graph:

\[ \begin{array}{ccc}
  n & \rightarrow & n-1 \\
  m & \rightarrow & m \\
  n-1 & \rightarrow & n-1 \\
  m-1 & \rightarrow & m-1 \\
  \vdots & \rightarrow & \vdots \\
  0 & \rightarrow & 0 \\
  \end{array} \]
dependency graph:

\[ \begin{array}{c}
\text{Edit Distance (VIII)} \\
\text{dependency graph:} \\
\end{array} \]
dependency graph:

```
 n  m  \rightarrow  n-1  m  \
 m-1  \downarrow  \downarrow  \downarrow  \
 n  m-1  \rightarrow  n-1  m-1  \
 m-1  \downarrow  \downarrow  \downarrow  \
 ...  \ldots  \
 n  0  \rightarrow  n-1  0  \
 0  m  \rightarrow  0  m-1  \
 ...  \ldots  
```
dependency graph:
dependency graph:
dependency graph:

- From $(n, m)$ to $(n-1, m)$
- From $(n, m)$ to $(n, m-1)$
- From $(n, m-1)$ to $(n-1, m-1)$
- From $(n-1, m)$ to $(n-1, m-1)$
- From $(n-1, m-1)$ to $(0, m-1)$
- From $(n, m-1)$ to $(0, m)$
- From $(n-1, m-1)$ to $(0, m)$
- From $(0, m)$ to $(0, m-1)$
- From $(0, m-1)$ to $(0, 0)$
- From $(0, m)$ to $(0, 0)$
dependency graph:
Edit Distance (IX)

iterative algorithm:

$$
\text{dist}(x_1 x_2 \ldots x_n, y_1 y_2 \ldots y_m)
$$

for $0 \leq i \leq n$

$$d[i][0] = i$$

for $0 \leq j \leq m$

$$d[0][j] = j$$

for $0 \leq i \leq n$

for $0 \leq j \leq m$

$$d[i][j] = \min \begin{cases} 
   d[i-1][j-1] + 1 & \text{if } x_i \neq y_j \\
   d[i-1][j] + 1 \\
   d[i][j-1] + 1 
\end{cases}$$

return $d[n][m]$
Iterative algorithm:

\[
\text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)
\]

for \(0 \leq i \leq n\)

\[d[i][0] = i\]

for \(0 \leq j \leq m\)

\[d[0][j] = j\]

for \(0 \leq i \leq n\)

for \(0 \leq j \leq m\)

\[d[i][j] = \min\left\{ \begin{array}{ll}
    d[i-1][j-1] + 1 & \text{if } x_i \neq y_j \\
    d[i-1][j] + 1 & \\
    d[i][j-1] + 1 & 
\end{array} \right.\]

return \(d[n][m]\)

Correctness:

Complexity:

\(O(nm)\) time, \(O(nm)\) space
iterative algorithm:

\[ \text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m) \]
iterative algorithm:

$$\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)$$

for $0 \leq i \leq n$
iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

\[
\text{for } 0 \leq i \leq n
\]

\[
d[i][0] = i
\]
iterative algorithm:

\[
\text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)
\]

for \(0 \leq i \leq n\)

\[d[i][0] = i\]

for \(0 \leq j \leq m\)

\[
\text{return } d[n][m]
\]

correctness:

complexity:

\(O(nm)\) time,

\(O(nm)\) space
Edit Distance (IX)

iterative algorithm:

\[
\text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)
\]

\[
\text{for } 0 \leq i \leq n \\
\hspace{1em} d[i][0] = i
\]

\[
\text{for } 0 \leq j \leq m \\
\hspace{1em} d[0][j] = j
\]

\[
\text{for } 1 \leq i \leq n \\
\hspace{1em} d[i][j] = \min \\
\hspace{2em} \begin{cases} \\
\text{d}[i-1][j-1] + 1 & \text{if } x_i \neq y_j \\
\text{d}[i-1][j] + 1 & \\
\text{d}[i][j-1] + 1 \\ 
\end{cases}
\]

\[
\text{return } d[n][m]
\]

correctness:

\[
\text{complexity: } O(nm) \text{ time, } O(nm) \text{ space}
\]
iterative algorithm:

\[
dist(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

for 0 ≤ i ≤ n
\[
d[i][0] = i
\]

for 0 ≤ j ≤ m
\[
d[0][j] = j
\]

for 0 ≤ i ≤ n
\[
\text{for } 0 \leq j \leq m
\]

\[
d[i][j] = \min\left\{\begin{array}{ll}
d[i-1][j-1] + 1 & \text{if } x_i \neq y_j \\
& \text{otherwise} \\
& \end{array}\right. \\
& \begin{array}{ll}
d[i-1][j] + 1 \\
d[i][j-1] + 1 \\
& \end{array} \\
\]

return \[d[n][m]\]
iterative algorithm:

\[ \text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m) \]

\[
\text{for } 0 \leq i \leq n
\]

\[ d[i][0] = i \]

\[
\text{for } 0 \leq j \leq m
\]

\[ d[0][j] = j \]

\[
\text{for } 0 \leq i \leq n
\]

\[
\text{for } 0 \leq j \leq m
\]

\[ d[i][j] = \]

\[
\begin{cases} 
\min 
\begin{pmatrix}
 d[i-1][j-1] + 1 
 d[i-1][j] + 1 
 d[i][j-1] + 1 
\end{pmatrix} & x_i \neq y_j \\
 d[i-1][j-1] & x_i = y_j 
\end{cases}
\]

\[ \text{return } d[n][m] \]

\[ \text{correctness:} \]

\[ \text{clear} \]

\[ \text{complexity:} \]

\[ O(nm) \text{ time}, O(nm) \text{ space} \]
 iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

\[
\text{for } 0 \leq i \leq n
\]

\[
d[i][0] = i
\]

\[
\text{for } 0 \leq j \leq m
\]

\[
d[0][j] = j
\]

\[
\text{for } 0 \leq i \leq n
\]

\[
\text{for } 0 \leq j \leq m
\]

\[
d[i][j] = \min \left\{ 
\begin{array}{c}
\text{dist}(x_1x_2\cdots x_i, y_1y_2\cdots y_{j-1}) + 1 \\
\text{dist}(x_1x_2\cdots x_{i-1}, y_1y_2\cdots y_j) + 1 \\
\text{dist}(x_1x_2\cdots x_{i-1}, y_1y_2\cdots y_{j-1}) + 1
\end{array} \right\}
\]

\]

correctness:
clear

complexity:
\(O(nm)\) time, \(O(nm)\) space
iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m) \\
\text{for } 0 \leq i \leq n \\
\quad d[i][0] = i \\
\text{for } 0 \leq j \leq m \\
\quad d[0][j] = j \\
\text{for } 0 \leq i \leq n \\
\quad \text{for } 0 \leq j \leq m \\
\quad \quad d[i][j] = \min \left\{ \begin{array}{l}
 d[i-1][j-1] + 1 \quad [x_i \neq y_j] \\
 1 \end{array} \right. 
\]

Iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

for \(0 \leq i \leq n\)

\[
d[i][0] = i
\]

for \(0 \leq j \leq m\)

\[
d[0][j] = j
\]

for \(0 \leq i \leq n\)

for \(0 \leq j \leq m\)

\[
d[i][j] = \min\left\{ \begin{array}{ll} 
d[i-1][j-1] + 1_{[x_i \neq y_j]} \\
d[i-1][j] + 1 
\end{array} \right. 
\]

return \(d[n][m]\)
iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

\[
\text{for } 0 \leq i \leq n \\
\quad d[i][0] = i
\]

\[
\text{for } 0 \leq j \leq m \\
\quad d[0][j] = j
\]

\[
\text{for } 0 \leq i \leq n \\
\quad \text{for } 0 \leq j \leq m
\]

\[
\quad d[i][j] = \min \left\{ d[i-1][j-1] + 1[ x_i \neq y_j ] , \quad d[i-1][j] + 1 , \quad d[i][j-1] + 1 \right\}
\]

\[
\text{return } d[n][m]
\]
iterative algorithm:

\[
\text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)
\]

\[
\text{for } 0 \leq i \leq n \\
\quad d[i][0] = i
\]

\[
\text{for } 0 \leq j \leq m \\
\quad d[0][j] = j
\]

\[
\text{for } 0 \leq i \leq n \\
\quad \text{for } 0 \leq j \leq m
\]

\[
d[i][j] = \min \left\{ \begin{array}{ll}
& d[i - 1][j - 1] + 1 \mathbb{1}[x_i \neq y_j] \\
& d[i - 1][j] + 1 \\
& d[i][j - 1] + 1 
\end{array} \right.
\]

\[
\text{return } d[n][m]
\]
iterative algorithm:

\[
\text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)
\]

\[
\begin{align*}
\text{for } & 0 \leq i \leq n \\
& d[i][0] = i \\
\text{for } & 0 \leq j \leq m \\
& d[0][j] = j \\
\text{for } & 0 \leq i \leq n \\
& \quad \text{for } 0 \leq j \leq m \\
& \quad d[i][j] = \min \left\{ \\
& \quad \quad d[i-1][j-1] + 1 \mathbb{I}[x_i \neq y_j] \\
& \quad \quad d[i-1][j] + 1 \\
& \quad \quad d[i][j-1] + 1 \\
& \right\} \\
\text{return } & d[n][m]
\end{align*}
\]
iterative algorithm:

\[
\text{dist}(x_1x_2 \cdots x_n, y_1y_2 \cdots y_m)
\]

\[
\begin{align*}
  &\text{for } 0 \leq i \leq n \\
  &\quad d[i][0] = i \\
  &\text{for } 0 \leq j \leq m \\
  &\quad d[0][j] = j \\
  &\text{for } 0 \leq i \leq n \\
  &\quad \text{for } 0 \leq j \leq m \\
  &\quad d[i][j] = \min \left\{ \\
  &\quad \quad d[i-1][j-1] + 1 \mathbb{1}[x_i \neq y_j] \\
  &\quad \quad d[i-1][j] + 1 \\
  &\quad \quad d[i][j-1] + 1 \right\} \\
  &\text{return } d[n][m]
\end{align*}
\]

correctness:
iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

\[
\begin{align*}
\text{for } 0 \leq i \leq n & \\
& d[i][0] = i \\
\text{for } 0 \leq j \leq m & \\
& d[0][j] = j \\
\text{for } 0 \leq i \leq n & \\
& \text{for } 0 \leq j \leq m \\
& d[i][j] = \min \left\{ d[i-1][j-1] + 1, d[i][j-1] + 1, d[i-1][j] + 1 \right\} \\
\text{return } d[n][m]
\end{align*}
\]

correctness: clear
iterative algorithm:

\[
dist(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

for \(0 \leq i \leq n\)
\[
d[i][0] = i
\]

for \(0 \leq j \leq m\)
\[
d[0][j] = j
\]

for \(0 \leq i \leq n\)
for \(0 \leq j \leq m\)
\[
d[i][j] = \min \left\{ d[i-1][j-1] + 1[x_i \neq y_j],
\begin{align*}
d[i - 1][j] + 1 \\
d[i][j - 1] + 1
\end{align*}
\right\}
\]

return \(d[n][m]\)

correctness: clear

complexity:
iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]

\[
\text{for } 0 \leq i \leq n \\
d[i][0] = i
\]

\[
\text{for } 0 \leq j \leq m \\
d[0][j] = j
\]

\[
\begin{align*}
\text{for } 0 \leq i \leq n \\
\quad \text{for } 0 \leq j \leq m \\
\quad \quad d[i][j] &= \min \left\{ d[i-1][j-1] + 1, d[i-1][j] + 1, d[i][j-1] + 1 \right\} \\
\end{align*}
\]

\[
\text{return } \ d[n][m]
\]

correctness: clear

complexity: \(O(nm)\) time,
**Iterative Algorithm:**

$$\text{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$$

for $0 \leq i \leq n$
$$d[i][0] = i$$

for $0 \leq j \leq m$
$$d[0][j] = j$$

for $0 \leq i \leq n$
for $0 \leq j \leq m$
$$d[i][j] = \min \left\{ d[i-1][j-1] + 1, d[i-1][j] + 1, d[i][j-1] + 1 \right\}$$

**Correctness:** clear

**Complexity:** $O(nm)$ time, $O(nm)$ space
Given two strings $x, y \in \Sigma^*$, we can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Proof. Exercise. Hint: follow how each subproblem was solved.
Corollary

Given two strings $x, y \in \Sigma^*$ can compute the minimum cost alignment in \( O(nm) \)-time and \( O(nm) \)-space.

Proof. Exercise. Hint: follow how each subproblem was solved.
Corollary

Given two strings \( x, y \in \Sigma^* \) can compute the minimum cost alignment
Corollary

Given two strings $x, y \in \Sigma^*$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.
Corollary

Given two strings $x, y \in \Sigma^*$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Proof.
<table>
<thead>
<tr>
<th>Corollary</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Given two strings }x, y \in \Sigma^* \textit{ can compute the minimum cost alignment in } O(nm)\text{-time and } O(nm)\text{-space.}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proof.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise.</td>
</tr>
</tbody>
</table>
Corollary

Given two strings $x, y \in \Sigma^*$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Proof.

Exercise. *Hint:*
Corollary

Given two strings $x, y \in \Sigma^*$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Proof.

Exercise. *Hint:* follow how each subproblem was solved.
Corollary

Given two strings \( x, y \in \Sigma^* \) can compute the minimum cost alignment in \( O(nm) \)-time and \( O(nm) \)-space.

Proof.

Exercise. *Hint*: follow how each subproblem was solved.
Dynamic Programming

- Develop recursive algorithm
- Understand structure of subproblems
- Memoize implicitly, via data structure
- Memoize explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
- Analysis (time, space)
- Further optimization
Dynamic Programming

template:
Dynamic Programming

**template:**
- develop recursive algorithm
Dynamic Programming

**template:**
- develop recursive algorithm
- understand structure of subproblems
template:
- develop recursive algorithm
- understand structure of subproblems
- memoize
Dynamic Programming

**template:**
- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly,
Dynamic Programming

**template:**

- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
**Dynamic Programming**

**template:**
- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly,
Dynamic Programming

template:
- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm
Dynamic Programming

**template:**

- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph

further optimization
Dynamic Programming

**template:**

- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
Dynamic Programming

template:

- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
- analysis
Dynamic Programming

template:

- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
- analysis (time,
Dynamic Programming

template:
- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
- analysis (time, space)
Dynamic Programming

**template:**
- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
- analysis (time, space)
- further optimization
Knapsack

Input: knapsack capacity \( W \in \mathbb{N} \) (in pounds), \( n \) items with weights \( w_1, \ldots, w_n \in \mathbb{N} \), and values \( v_1, \ldots, v_n \in \mathbb{N} \).

Goal: a subset \( S \subseteq [n] \) of items that fit in the knapsack, with maximum value

\[
\sum_{i \in S} w_i \leq W \\
\sum_{i \in S} v_i
\]

Remarks: prototypical problem in combinatorial optimization, can be generalized in numerous ways, needs to be solved in practice.
the knapsack problem:
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds).
Knapsack

the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$,
Knapsack

the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$. 

**goal:** a subset $S \subseteq \{1, \ldots, n\}$ of items that fit in the knapsack, with maximum value

$$\sum_{i \in S} v_i$$

**remarks:** prototypical problem in combinatorial optimization, can be generalized in numerous ways, needs to be solved in practice.
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items
Knapsack

the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items that fit in the knapsack,
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items that fit in the knapsack, with maximum value
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items that fit in the knapsack, with maximum value

$$\max_{S \subseteq [n]} \sum_{i \in S} v_i$$
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items that fit in the knapsack, with maximum value

$$\max_{S \subseteq [n]} \sum_{i \in S} v_i$$

subject to

$$\sum_{i \in S} w_i \leq W$$
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items that fit in the knapsack, with maximum value

$$\max_{S \subseteq [n]} \left( \sum_{i \in S} v_i \right)$$

subject to

$$\sum_{i \in S} w_i \leq W$$

**remarks:**
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items that fit in the knapsack, with maximum value

$$\max_{S \subseteq [n]} \left\{ \sum_{i \in S} v_i \mid \sum_{i \in S} w_i \leq W \right\}$$

**remarks:**

- prototypical problem in *combinatorial optimization*,

26 / 33
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subseteq [n]$ of items that fit in the knapsack, with maximum value

$$\max_{S \subseteq [n]} \sum_{i \in S} v_i \quad \text{subject to} \quad \sum_{i \in S} w_i \leq W$$

**remarks:**

- prototypical problem in *combinatorial optimization*, can be generalized in numerous ways
the knapsack problem:

**input:** knapsack capacity \( W \in \mathbb{N} \) (in pounds). \( n \) items with weights \( w_1, \ldots, w_n \in \mathbb{N} \), and values \( v_1, \ldots, v_n \in \mathbb{N} \).

**goal:** a subset \( S \subseteq [n] \) of items that fit in the knapsack, with maximum value

\[
\max_{S \subseteq [n]} \sum_{i \in S} v_i \quad \text{subject to} \quad \sum_{i \in S} w_i \leq W
\]

**remarks:**
- prototypical problem in *combinatorial optimization*, can be generalized in numerous ways
- needs to be solved in practice
Knapsack (II)

Example

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>28</td>
</tr>
</tbody>
</table>

For $W = 11$, the best is \{3, 4\} giving value 40.

Definition

In the special case of when $v_i = w_i$ for all $i$, the knapsack problem is called the subset sum problem.
### Example

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>28</td>
</tr>
</tbody>
</table>

For $W = 11$, the best is \{3, 4\} giving value 40.

**Definition**

In the special case of when $v_i = w_i$ for all $i$, the knapsack problem is called the **subset sum** problem.
Example

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

For $W = 11$, the best is \{3, 4\} giving value 40.

Definition

In the special case of when $v_i = w_i$ for all $i$, the knapsack problem is called the subset sum problem.
For $W = 11$, the best is \{3, 4\} giving value 40.

Definition

In the special case of when $v_i = w_i$ for all $i$, the knapsack problem is called the subset sum problem.
For $W = 11$, the best is $\{3, 4\}$ giving value 40.
Example

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

For $W = 11$, the best is $\{3, 4\}$ giving value 40.

Definition
Example

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

For $W = 11$, the best is $\{3, 4\}$ giving value 40.

Definition

In the special case of when $v_i = w_i$ for all $i$,
Example

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

For $W = 11$, the best is $\{3, 4\}$ giving value 40.

Definition

In the special case of when $v_i = w_i$ for all $i$, the knapsack problem is called the subset sum problem.
## Knapsack (III)

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

And weight limit \( W = 15 \).

What is the best solution value?

- (a) 22
- (b) 28
- (c) 38
- (d) 50
- (e) 56
What is the best solution value?

(a) 22  
(b) 28  
(c) 38  
(d) 50  
(e) 56

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>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>
Knapsack (III)

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>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 $W = 15$. 

What is the best solution value?

(a) 22
(b) 28
(c) 38
(d) 50
(e) 56
and weight limit $W = 15$. What is the best solution value?
Knapsack (III)

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>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 $W = 15$. What is the best solution value?
(a) 22
(b) 28
(c) 38
(d) 50
(e) 56
Knapsack (IV)

Greedy approaches:

1. Greedily select by maximum value:

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

2. Greedily select by minimum weight:

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

3. Greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

For $W = 4$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

Remark: While greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
Knapsack (IV)

**greedy** approaches:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-value will pick {3}, but optimal is {1, 2}.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-weight will pick {1}, but optimal is {2}.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

For $W = 4$, greedy-value will pick {3}, but optimal is {1, 2}.

Remark: While greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

For $W = 4$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

Remark: while greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
**greedy** approaches:

- greedily select by maximum value:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W=2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W=2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

For $W=4$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

Remark: While greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
Knapsack (IV)

**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For $W = 4$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

Remark: while greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
**Knapsack (IV)**

**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-value will pick \{3\},

**remark:** while greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
Knapsack (IV)

**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For $W = 4$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

Remark: While greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

remark: while greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Knapsack (IV)

**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$,
**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick \{1\},

**remark:** while greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
**Knapsack (IV)**

**greedy** approaches:

- greedily select by maximum value:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

**remark:** while greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
**Knapsack (IV)**

**greedy** approaches:

- greedily select by maximum value:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

- greedily select by maximum value/weight ratio:
Knapsack (IV)

**greedy** approaches:

- **greedily select by maximum value:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- **greedily select by minimum weight:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

- **greedily select by maximum value/weight ratio:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Knapsack (IV)

**greedy** approaches:

- greedily select by maximum value:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For $W = 4$,
Knapsack (IV)

**greedy** approaches:

- **greedily select by maximum value:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
  
  For \( W = 2 \), greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- **greedily select by minimum weight:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
  
  For \( W = 2 \), greedy-weight will pick \{1\}, but optimal is \{2\}.

- **greedily select by maximum value/weight ratio:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
  
  For \( W = 4 \), greedy-value will pick \{3\},
**Knapsack (IV)**

**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For $W = 4$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.
Knapsack (IV)

**greedy** approaches:

- greedily select by maximum value:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

- greedily select by maximum value/weight ratio:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For $W = 4$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

**remark:**

While greedy algorithms fail to get the best result, they can still be useful for getting solutions that are approximately the best.
Knapsack (IV)

**greedy** approaches:

- **greedily select by maximum value:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- **greedily select by minimum weight:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For $W = 2$, greedy-weight will pick \{1\}, but optimal is \{2\}.

- **greedily select by maximum value/weight ratio:**
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For $W = 4$, greedy-value will pick \{3\}, but optimal is \{1, 2\}.

**remark:** while greedy algorithms fail to get the best result,
Knapsack (IV)

**greedy** approaches:

- greedily select by maximum value:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For \( W = 2 \), greedy-value will pick \{3\}, but optimal is \{1, 2\}.

- greedily select by minimum weight:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For \( W = 2 \), greedy-weight will pick \{1\}, but optimal is \{2\}.

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For \( W = 4 \), greedy-value will pick \{3\}, but optimal is \{1, 2\}.

**remark:** while greedy algorithms fail to get the *best* result, they can still be useful for getting solutions that are *approximately* the best.
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq \{1, \ldots, n\}$. Then,

1. if $n \not\in S$, then $S \subseteq \{1, \ldots, n-1\}$ is an optimal solution for the knapsack instance $(W, (v_i)_{i=1}^n, (w_i)_{i=1}^n)$.

2. if $n \in S$, then $S \setminus \{n\} \subseteq \{1, \ldots, n-1\}$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i=1}^n, (w_i)_{i=1}^n)$.

Proof.

1. Any $S \subseteq \{1, \ldots, n-1\}$ feasible for $(W, (v_i)_{i=1}^n, (w_i)_{i=1}^n)$ will also satisfy the original weight constraint.

2. Any $S \subseteq \{1, \ldots, n-1\}$ feasible for $(W - w_n, (v_i)_{i=1}^n, (w_i)_{i=1}^n)$ will have that $S \cup \{n\}$ will also satisfy the original weight constraint.
**Knapsack (V)**

<table>
<thead>
<tr>
<th><strong>Lemma</strong></th>
</tr>
</thead>
</table>
| Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \not\in S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.

2. if $n \in S$, then $S \cup \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$. |

**Proof.**
1. Any $S \subseteq [n-1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$ will also satisfy the original weight constraint.

2. Any $S \subseteq [n-1]$ feasible for $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$ will have that $S \cup \{n\}$ will also satisfy the original weight constraint.
Lemma

Consider the instance $W$, $(v_i)^n_{i=1}$, and $(w_i)^n_{i=1}$,

1. if $n \not\in S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)^n_{i=1}, (w_i)^n_{i=1})$.

2. if $n \in S$, then $S \cup \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)^n_{i=1}, (w_i)^n_{i=1})$.

Proof.

1. Any $S \subseteq [n-1]$ feasible for $(W, (v_i)^n_{i=1}, (w_i)^n_{i=1})$ will also satisfy the original weight constraint.

2. Any $S \subseteq [n-1]$ feasible for $(W - w_n, (v_i)^n_{i=1}, (w_i)^n_{i=1})$ will have that $S \cup \{n\}$ will also satisfy the original weight constraint.
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. 
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \notin S$,
Lemma

Consider the instance \( W, (v_i)_{i=1}^n \) and \( (w_i)_{i=1}^n \), with optimal solution \( S \subseteq [n] \). Then,

1. \( n \notin S \), then \( S \subseteq [n - 1] \)
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \not\in S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$. 

2. if $n \in S$, then $S \cup \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W-w_n, (v_i)_{i<n}, (w_i)_{i<n})$. 

Proof.

1. Any $S \subseteq [n-1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$, will also satisfy the original weight constraint.

2. Any $S \subseteq [n-1]$ feasible for $(W-w_n, (v_i)_{i<n}, (w_i)_{i<n})$, will have that $S \cup \{n\}$ will also satisfy the original weight constraint.
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \notin S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.

2. if $n \in S$,
Lemma

Consider the instance $W, (v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \notin S$, then $S \subseteq [n - 1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.
2. if $n \in S$, then $S \setminus \{n\} \subseteq [n - 1]$.
Lemma

Consider the instance \( W, (v_i)_{i=1}^{n}, \) and \( (w_i)_{i=1}^{n}, \) with optimal solution \( S \subseteq [n]. \) Then,

1. if \( n \notin S, \) then \( S \subseteq [n - 1] \) is an optimal solution for the knapsack instance \( (W, (v_i)_{i<n}, (w_i)_{i<n}). \)

2. if \( n \in S, \) then \( S \setminus \{n\} \subseteq [n - 1] \) is an optimal solution for the knapsack instance \( (W - w_n, (v_i)_{i<n}, (w_i)_{i<n}). \)
Lemma

Consider the instance $W, (v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \notin S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.

2. if $n \in S$, then $S \setminus \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

Proof.
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. If $n \notin S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.
2. If $n \in S$, then $S \setminus \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

Proof.

1. Any $S \subseteq [n-1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$. 
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \not\in S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.

2. if $n \in S$, then $S \setminus \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

Proof.

1. Any $S \subseteq [n-1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$, will also satisfy the original weight constraint.
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. If $n \notin S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.
2. If $n \in S$, then $S \setminus \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

Proof.

1. Any $S \subseteq [n-1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$, will also satisfy the original weight constraint
2. Any $S \subseteq [n-1]$ feasible for $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$. 
Consider the instance $W, (v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \notin S$, then $S \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.

2. if $n \in S$, then $S \setminus \{n\} \subseteq [n-1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

**Proof.**

1. Any $S \subseteq [n-1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$, will also satisfy the original weight constraint

2. Any $S \subseteq [n-1]$ feasible for $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$, will have that $S \cup \{n\}$ will also satisfy the original weight constraint
**Lemma**

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. If $n \notin S$, then $S \subseteq [n - 1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.

2. If $n \in S$, then $S \setminus \{n\} \subseteq [n - 1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

**Proof.**

1. Any $S \subseteq [n - 1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$, will also satisfy the original weight constraint.

2. Any $S \subseteq [n - 1]$ feasible for $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$, will have that $S \cup \{n\}$ will also satisfy the original weight constraint.
Corollary

Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w$, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \text{OPT}(i-1, w) & w_i > w_{\text{max}} \\ \text{OPT}(i-1, w-w_i) + v_i & \text{else} \end{cases}$$

from instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$ we generate $O(n \cdot W)$-many subproblems $(i, w) \in [n]$, $w \leq W$. 
Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$.

Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w$, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$.

Then,

$$\text{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \text{OPT}(i-1, w) & w_i > w_{\text{max}} \\ \max\{\text{OPT}(i-1, w), \text{OPT}(i-1, w-w_i) + v_i\} & \text{else} \end{cases}$$

From instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$ we generate $O(n \cdot W)$-many subproblems $(i, w) \in [n], w \leq W$. 
Corollary

Fix an instance $W, v_1, \ldots, v_n, and w_1, \ldots, w_n$. 

Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i and w_1, \ldots, w_i$. 

Then, 

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

from instance $W, v_1, \ldots, v_n, and w_1, \ldots, w_n$ we generate $O(n \cdot W)$-many subproblems $i \in [n]$, $w \leq W$. 


Corollary

Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$
\text{OPT}(i, w) = \begin{cases}
0 & i = 0 \\
\text{OPT}(i-1, w) & w_i > w_{\text{max}} \\
\max \left\{ \text{OPT}(i-1, w), \text{OPT}(i-1, w-w_i) + v_i \right\} & \text{else}
\end{cases}
$$

from instance $W, v_1, \ldots, v_n, and w_1, \ldots, w_n$. We generate $O(n \cdot W)$-many subproblems $(i, w)$, $i \in [n], w \leq W$. 


Corollary

Fix an instance $W, v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \text{OPT}(i-1, w) & w_i > \text{w}_{\text{max}} \\ \text{OPT}(i-1, w) + v_i & \text{else} \end{cases}$$
Corollary

Fix an instance \( W, v_1, \ldots, v_n, \) and \( w_1, \ldots, w_n \). Define \( \text{OPT}(i, w) \) to be the maximum value of the knapsack instance \( w, v_1, \ldots, v_i \) and \( w_1, \ldots, w_i \). Then,

\[
\text{OPT}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
& \text{else} \\
\text{OPT}(i-1, w) & \text{if } w_i > w_{\max} \\
\text{OPT}(i-1, w - w_i) + v_i & \text{otherwise}
\end{cases}
\]
Corollary

Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 
0 & \quad i = 0 \\
\text{OPT}(i-1, w) & \quad w_i > W \\
\text{OPT}(i-1, w-w_i) + v_i & \text{else}
\end{cases}$$
Knapsack (VI)

Corollary

Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w$, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 
0 & i = 0 \\
\text{OPT}(i - 1, w) & \text{else}
\end{cases}$$
Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $OPT(i, w)$ to be the maximum value of the knapsack instance $w$, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$OPT(i, w) = \begin{cases} 0 & i = 0 \\ OPT(i - 1, w) & w_i > w \end{cases}$$
Corollary

Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 
0 & i = 0 \\
\text{OPT}(i - 1, w) & w_i > w \\
\max \left\{ \right. & 
\end{cases}$$
Corollary

Fix an instance \( W, v_1, \ldots, v_n, \) and \( w_1, \ldots, w_n \). Define \( \text{OPT}(i, w) \) to be the maximum value of the knapsack instance \( w, v_1, \ldots, v_i \) and \( w_1, \ldots, w_i \). Then,

\[
\text{OPT}(i, w) = \begin{cases} 
0 & i = 0 \\
\text{OPT}(i - 1, w) & w_i > w \\
\max \left\{ \text{OPT}(i - 1, w) \right\} & \text{otherwise}
\end{cases}
\]
Corollary

Fix an instance $W$, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w$, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

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

Fix an instance $W, v_1, \ldots, v_n, \text{ and } w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 
0 & i = 0 \\
\text{OPT}(i-1, w) & w_i > w \\
\max \left\{ \text{OPT}(i-1, w), \text{OPT}(i-1, w-w_i) + v_i \right\} & \text{else}
\end{cases}$$
Corollary

Fix an instance $W, v_1, \ldots, v_n, \text{ and } w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 
0 & i = 0 \\
\text{OPT}(i - 1, w) & w_i > w \\
\max \left\{ \text{OPT}(i - 1, w) \right. \\
\left. \text{OPT}(i - 1, w - w_i) + v_i \right\} & \text{else}
\end{cases}$$

$\implies$ from instance $W, v_1, \ldots, v_n, \text{ and } w_1, \ldots, w_n$
Fix an instance $W, v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

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

$\Rightarrow$ from instance $W, v_1, \ldots, v_n$, and $w_1, \ldots, w_n$ we generate $O(n \cdot W)$-many subproblems
Corollary

Fix an instance $W, v_1, \ldots, v_n, \text{ and } w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i \text{ and } w_1, \ldots, w_i$. Then,

$$\text{OPT}(i, w) = \begin{cases} 
0 & i = 0 \\
\text{OPT}(i - 1, w) & w_i > w \\
\max \left\{ \begin{array}{ll}
\text{OPT}(i - 1, w) \\
\text{OPT}(i - 1, w - w_i) + v_i 
\end{array} \right. & \text{else}
\end{cases}$$

$\Rightarrow$ from instance $W, v_1, \ldots, v_n, \text{ and } w_1, \ldots, w_n \text{ we generate } O(n \cdot W)$-many subproblems $(i, w)_{i \in [n], w \leq W}$. 
Knapsack (VII)

an iterative algorithm:

\[ M[i, w] \text{ will compute } \text{OPT}(i, w) \text{ for } 0 \leq w \leq W \]

\[ M[0, w] = 0 \]

\[ M[i, w] = \begin{cases} 
M[i-1, w] & \text{if } w_i > w \\
\max(M[i-1, w], M[i-1, w-w_i] + v_i) & \text{else}
\end{cases} \]

**Correctness:**

**Complexity:** $O(nW)$ time, but input size is $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$

*E.g.,* $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\Rightarrow$ running time is not polynomial in the input.

Algorithm is pseudo-polynomial: running time is polynomial in the magnitude of the input numbers.

Knapsack is NP-hard in general $\Rightarrow$ no efficient algorithm is expected to compute the exact optimum.

**Punchline:** had to correctly parameterize knapsack sub-problems $(v_j)_{j \leq i}, (w_j)_{j \leq i}$ by also considering arbitrary $w_j$.

This is a common theme in dynamic programming problems.
an iterative algorithm:

\[ M[i, w] \] will compute \( OPT(i, w) \) for \( 0 \leq w \leq W \)

\[ M[0, w] = 0 \]

for \( 1 \leq i \leq n \)

for \( 1 \leq w \leq W \)

if \( w_i > w \)

\[ M[i, w] = M[i-1, w] \]

else

\[ M[i, w] = \max(M[i-1, w], M[i-1, w-w_i] + v_i) \]

**Correctness:**

**Complexity:** \( O(nW) \) time, but input size is \( O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i)) \).

For example, \( W = 2^n \) has \( O(n) \) bits but requires \( \Omega(2^n) \) runtime, \( \Rightarrow \) running time is not polynomial in the input.

Algorithm is pseudo-polynomial: running time is polynomial in the magnitude of the input numbers.

Knapsack is \( NP \)-hard in general, \( \Rightarrow \) no efficient algorithm is expected to compute the exact optimum.

**Punchline:**

Had to correctly parameterize knapsack sub-problems \((v_j)_{j \leq i}, (w_j)_{j \leq i}\) by also considering arbitrary \( w_i \). This is a common theme in dynamic programming problems.
an iterative algorithm: $M[i, w]$ will compute $OPT(i, w)$
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

for $0 \leq w \leq W$

$M[0, w] = 0$
an iterative algorithm: \( M[i, w] \) will compute \( \text{OPT}(i, w) \)

\[
\begin{align*}
&\text{for } 0 \leq w \leq W \\
&\quad M[0, w] = 0 \\
&\text{for } 1 \leq i \leq n \\
\end{align*}
\]
an iterative algorithm: $M[i, w]$ will compute $OPT(i, w)$

for $0 \leq w \leq W$

$M[0, w] = 0$

for $1 \leq i \leq n$

for $1 \leq w \leq W$
**Knapsack (VII)**

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

```plaintext
for $0 \leq w \leq W$
    $M[0, w] = 0$

for $1 \leq i \leq n$
    for $1 \leq w \leq W$
        if $w_i > w$
```

**Correctness:**

**Complexity:** $O(nW)$ time, but input size is $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$.

*E.g.,* $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\Rightarrow$ running time is not polynomial in the input.

*Algorithm is pseudo-polynomial:* running time is polynomial in the magnitude of the input numbers.

*Knapsack is NP-hard in general* $\Rightarrow$ no efficient algorithm is expected to compute the exact optimum.

**Punchline:** had to correctly parameterize knapsack sub-problems ($v_j$) $j \leq i$, ($w_j$) $j \leq i$ by also considering arbitrary $w$.

*This is a common theme in dynamic programming problems.*
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

for $0 \leq w \leq W$

$M[0, w] = 0$

for $1 \leq i \leq n$

for $1 \leq w \leq W$

if $w_i > w$

$M[i, w] = M[i-1, w]$
Knapsack (VII)

**an iterative algorithm:** \( M[i, w] \) will compute \( \text{OPT}(i, w) \)

for \( 0 \leq w \leq W \)

\( M[0, w] = 0 \)

for \( 1 \leq i \leq n \)

for \( 1 \leq w \leq W \)

if \( w_i > w \)

\[ M[i, w] = M[i - 1, w] \]

else
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

```latex
\text{for } 0 \leq w \leq W \\
\quad M[0, w] = 0 \\
\text{for } 1 \leq i \leq n \\
\quad \text{for } 1 \leq w \leq W \\
\quad \quad \text{if } w_i > w \\
\quad \quad \quad M[i, w] = M[i - 1, w] \\
\quad \quad \text{else} \\
\quad \quad \quad M[i, w] = \max( \\
```

**correctness:**

**complexity:**

$O(nW)$ time, but input size is $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$.

- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime.
- ⇒ running time is not polynomial in the input.

Algorithm is pseudo-polynomial: running time is polynomial in the magnitude of the input numbers.

Knapsack is $\mathsf{NP}$-hard in general ⇒ no efficient algorithm is expected to compute the exact optimum.

**punchline:** had to correctly parameterize knapsack sub-problems ($v_j \leq i, w_j \leq i$) by also considering arbitrary $w_i$. This is a common theme in dynamic programming problems.
Knapsack (VII)

**an iterative algorithm:** \( M[i, w] \) will compute \( \text{OPT}(i, w) \)

```plaintext
for 0 ≤ w ≤ W
    \( M[0, w] = 0 \)

for 1 ≤ i ≤ n
    for 1 ≤ w ≤ W
        if \( w_i > w \)
            \( M[i, w] = M[i - 1, w] \)
        else
            \( M[i, w] = \max(M[i - 1, w], \)\)
```

Correctness:
clear

Complexity:
\( O(nW) \) time, but input size is \( O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)) \)

E.g., \( W = 2^n \) has \( O(n) \) bits but requires \( \Omega(2^n) \) runtime.
Algorithm is pseudo-polynomial:
running time is polynomial in magnitude of the input numbers.

Knapsack is \( \text{NP} \)-hard in general
⇒ no efficient algorithm is expected to compute the exact optimum.

Punchline:
had to correctly parameterize knapsack sub-problems \( (v_j)_{j \leq i}, (w_j)_{j \leq i} \) by also considering arbitrary \( w_i \).
This is a common theme in dynamic programming problems.
an iterative algorithm: $M[i, w]$ will compute $\text{OPT}(i, w)$

for $0 \leq w \leq W$
  $M[0, w] = 0$

for $1 \leq i \leq n$
  for $1 \leq w \leq W$
    if $w_i > w$
      $M[i, w] = M[i - 1, w]$
    else
      $M[i, w] = \max(M[i - 1, w],$
      $M[i - 1, w - w_i] + v_i)$
**Knapsack (VII)**

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

\[
\begin{array}{l}
\text{for } 0 \leq w \leq W \\
M[0, w] = 0 \\
\text{for } 1 \leq i \leq n \\
\quad \text{for } 1 \leq w \leq W \\
\quad \quad \text{if } w_i > w \\
\quad \quad \quad M[i, w] = M[i - 1, w] \\
\quad \quad \text{else} \\
\quad \quad \quad M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i) \\
\end{array}
\]
an iterative algorithm: $M[i, w]$ will compute $\text{OPT}(i, w)$

\begin{verbatim}
for 0 ≤ w ≤ W
    M[0, w] = 0
for 1 ≤ i ≤ n
    for 1 ≤ w ≤ W
        if $w_i > w$
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
\end{verbatim}

correctness:
an iterative algorithm: \( M[i, w] \) will compute \( \text{OPT}(i, w) \)

\[
\begin{array}{l}
\text{for } 0 \leq w \leq W \\
\quad M[0, w] = 0 \\
\text{for } 1 \leq i \leq n \\
\quad \text{for } 1 \leq w \leq W \\
\quad \quad \text{if } w_i > w \\
\quad \quad \quad M[i, w] = M[i - 1, w] \\
\quad \quad \text{else} \\
\quad \quad \quad M[i, w] = \max(M[i - 1, w], \quad \quad \quad M[i - 1, w - w_i] + v_i) \\
\end{array}
\]

correctness: clear

Knapsack is NP-hard in general \( \Rightarrow \) no efficient algorithm is expected to compute the exact optimum

This is a common theme in dynamic programming problems.
an iterative algorithm: $M[i, w]$ will compute $\text{OPT}(i, w)$

\begin{verbatim}
for 0 ≤ w ≤ W
    M[0, w] = 0
for 1 ≤ i ≤ n
    for 1 ≤ w ≤ W
        if $w_i > w$
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
\end{verbatim}

correctness: clear

complexity:
an iterative algorithm: $M[i, w]$ will compute $\text{OPT}(i, w)$

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

correctness: clear

complexity:
- $O(nW)$ time,
**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

```plaintext
for $0 \leq w \leq W$
    $M[0, w] = 0$

for $1 \leq i \leq n$
    for $1 \leq w \leq W$
        if $w_i > w$
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
```

**correctness:** clear

**complexity:**

- $O(nW)$ time, but *input size* is $O(n)$
an iterative algorithm: $M[i, w]$ will compute $\text{OPT}(i, w)$

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

correctness: clear

complexity:

- $O(nW)$ time, but input size is $O(n + \log W)$
Knapsack (VII)

**an iterative algorithm:** \(M[i, w]\) will compute \(\text{OPT}(i, w)\)

```plaintext
for 0 \leq w \leq W
  \[ M[0, w] = 0 \]
for 1 \leq i \leq n
  for 1 \leq w \leq W
    if \(w_i > w\)
      \[ M[i, w] = M[i - 1, w] \]
    else
      \[ M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i) \]
```

correctness: clear

**complexity:**

- \(O(nW)\) time, but *input size* is
  \(O(n + \log W + \sum_{i=1}^{n}(\log v_i))\)
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $OPT(i, w)$

```
for 0 ≤ w ≤ W
    M[0, w] = 0
for 1 ≤ i ≤ n
    for 1 ≤ w ≤ W
        if $w_i > w$
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
```

correctness: clear

**complexity:**

- $O(nW)$ time, but input size is $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

```
for 0 ≤ w ≤ W
    M[0, w] = 0
for 1 ≤ i ≤ n
    for 1 ≤ w ≤ W
        if $w_i > w$
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
```

correctness: clear

**complexity:**

- $O(nW)$ time, but input size is $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$
Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $OPT(i, w)$

for $0 \leq w \leq W$
    $M[0, w] = 0$

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

correctness: clear

complexity:

- $O(nW)$ time, but input size is $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$

- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\implies$ running time is not polynomial in the input
Knapsack (VII)

**an iterative algorithm:** \( M[i, w] \) will compute \( \text{OPT}(i, w) \)

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

**correctness:** clear

**complexity:**

- \( O(nW) \) time, but input size is \( O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)) \)

- e.g., \( W = 2^n \) has \( O(n) \) bits but requires \( \Omega(2^n) \) runtime \( \Rightarrow \) running time is not polynomial in the input

- Algorithm is **pseudo-polynomial**:
**Knapsack (VII)**

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

```
for 0 ≤ w ≤ W
    M[0, w] = 0
for 1 ≤ i ≤ n
    for 1 ≤ w ≤ W
        if $w_i > w$
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
```

**correctness:** clear

**complexity:**
- $O(nW)$ time, but input size is
  $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$
- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\implies$ running time is **not** polynomial in the input
- Algorithm is **pseudo-polynomial**: running time is polynomial in magnitude of the input numbers

Knapsack is $\text{NP}$-hard in general $\implies$ no efficient algorithm is expected to compute the exact optimum

This is a common theme in dynamic programming problems.
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

- **for** $0 \leq w \leq W$
  - $M[0, w] = 0$
- **for** $1 \leq i \leq n$
  - **for** $1 \leq w \leq W$
    - if $w_i > w$
      - $M[i, w] = M[i - 1, w]$
    - else
      - $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$

**correctness:** clear

**complexity:**
- $O(nW)$ time, but *input size* is $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$

- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\implies$ running time is **not** polynomial in the input

- Algorithm is **pseudo-polynomial**: running time is polynomial in *magnitude* of the input numbers

- Knapsack is NP-hard in general
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

```
for 0 ≤ w ≤ W
  M[0, w] = 0
for 1 ≤ i ≤ n
  for 1 ≤ w ≤ W
    if $w_i > w$
      $M[i, w] = M[i - 1, w]$
    else
      $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
```

correctness: clear

complexity:

- $O(nW)$ time, but input size is
  $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$

- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\implies$ running time is not polynomial in the input

- Algorithm is **pseudo-polynomial**: running time is polynomial in magnitude of the input numbers

- Knapsack is NP-hard in general $\implies$ no efficient algorithm is expected to compute the exact optimum
Knapsack (VII)

**an iterative algorithm:** $M[i, w]$ will compute $\text{OPT}(i, w)$

```
for 0 ≤ w ≤ W
    M[0, w] = 0
for 1 ≤ i ≤ n
    for 1 ≤ w ≤ W
        if $w_i > w$
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
```

correctness: clear

**complexity:**

- $O(nW)$ time, but *input size* is $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$

- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\implies$ running time is **not** polynomial in the input

- Algorithm is **pseudo-polynomial**: running time is polynomial in *magnitude* of the input numbers

- Knapsack is NP-hard in general $\implies$ no efficient algorithm is expected to compute the exact optimum

**punchline:**
an iterative algorithm: \( M[i, w] \) will compute \( \text{OPT}(i, w) \)

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

**correctness:** clear

**complexity:**
- \( O(nW) \) time, but input size is \( O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)) \)
- e.g., \( W = 2^n \) has \( O(n) \) bits but requires \( \Omega(2^n) \) runtime \( \implies \) running time is not polynomial in the input
- Algorithm is *pseudo-polynomial*: running time is polynomial in magnitude of the input numbers
- Knapsack is \( \text{NP}-\text{hard} \) in general \( \implies \) no efficient algorithm is expected to compute the exact optimum

**punchline:** had to correctly parameterize knapsack sub-problems \((v_j)_{j \leq i}, (w_j)_{j \leq i}\) by also considering arbitrary \( w \).
**Knapsack (VII)**

**an iterative algorithm:**  
\[ M[i, w] \] will compute \( \text{OPT}(i, w) \)

\[
\text{for } 0 \leq w \leq W \\
\quad M[0, w] = 0 \\
\text{for } 1 \leq i \leq n \\
\quad \text{for } 1 \leq w \leq W \\
\quad \quad \text{if } w_i > w \\
\quad \quad \quad M[i, w] = M[i - 1, w] \\
\quad \quad \text{else} \\
\quad \quad \quad M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)
\]

**correctness:** clear

**complexity:**
- \( O(nW) \) time, but input size is \( O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i)) \)
- e.g., \( W = 2^n \) has \( O(n) \) bits but requires \( \Omega(2^n) \) runtime \( \implies \) running time is not polynomial in the input
- Algorithm is **pseudo-polynomial**: running time is polynomial in magnitude of the input numbers
- Knapsack is NP-hard in general \( \implies \) no efficient algorithm is expected to compute the exact optimum

**punchline:** had to correctly parameterize knapsack sub-problems \( (v_j)_{j \leq i}, (w_j)_{j \leq i} \) by also considering arbitrary \( w \). This is a common theme in dynamic programming problems.
today:
- recursion
- dynamic programming
  - fibonacci numbers
  - edit distance
  - knapsack

next time: more dynamic programming logistics:
- pset0 due R5, (aka, tomorrow) — submit individually!
- pset1 out tomorrow, due R5 (next week)
- piazza signup