cs473: Algorithms

Lecture 3: Dynamic Programming

Michael A. Forbes

University of Illinois at Urbana-Champaign

September 2, 2019

Today

Today

logistics:

Today

logistics:

- pset0 due R5,

Today

logistics:

- pset0 due R5, (aka, tomorrow)

Today

logistics:

- pset0 due R5, (aka, tomorrow) — submit *individually*!

Today

logistics:

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

Today

logistics:

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

Today

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:

Today

logistics:

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

last lecture:

- divide and conqueror

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

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

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

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
- dynamic programming

Recursion

Recursion

Definition

Recursion

Definition

A **reduction** transforms a given problem into a yet another problem,

Recursion

Definition

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

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.

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

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,

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)

Recursion (II)

recursive paradigms:

Recursion (II)

recursive paradigms:

- **tail recursion**:

Recursion (II)

recursive paradigms:

- **tail recursion**: expend effort

Recursion (II)

recursive paradigms:

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

Recursion (II)

recursive paradigms:

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

Recursion (II)

recursive paradigms:

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

Recursion (II)

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:**

Recursion (II)

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

Recursion (II)

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)

Recursion (II)

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,

Recursion (II)

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,

Recursion (II)

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.

Recursion (II)

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

Recursion (II)

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).

Recursion (II)

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:

Recursion (II)

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,

Recursion (II)

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,

Recursion (II)

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, ...

Recursion (II)

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:**

Recursion (II)

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.

Recursion (II)

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.

Recursion (II)

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,

Recursion (II)

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)

Recursion (II)

$\texttt{foo}(X)$

Recursion (II)

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

Recursion (II)

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

Recursion (II)

```
foo(X)
    if X is a base case
        solve it
        return solution
```

Recursion (II)

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

Recursion (II)

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

Recursion (II)

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

Recursion (II)

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

Recursion (II)

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

Recursion (II)

```
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
```

Recursion (II)

```
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)

```
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)

```
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)

```
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:*

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

Fibonacci Numbers

Definition (Fibonacci 1200,)

Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

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

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

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

- $F_0 = 0$

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

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

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

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

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 $n \geq 2$

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:

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

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

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

remarks:

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

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

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*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, φ 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*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, φ is the *golden ratio* $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618\cdots$
- $\implies 1 - \varphi \approx -.618\cdots \implies |(1-\varphi)^n| \leq 1$,

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*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, φ is the *golden ratio* $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618\cdots$
- $\implies 1 - \varphi \approx -.618\cdots \implies |(1-\varphi)^n| \leq 1$, and further $(1-\varphi)^n \to_{n\to\infty} 0$

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*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, φ is the *golden ratio* $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618\cdots$
- $\implies 1 - \varphi \approx -.618\cdots \implies |(1-\varphi)^n| \leq 1$, and further $(1-\varphi)^n \to_{n\to\infty} 0$ $\implies F_n = \Theta(\varphi^n)$.

Fibonacci Numbers (II)

Fibonacci Numbers (II)

question:

Fibonacci Numbers (II)

question: given n, compute F_n.

Fibonacci Numbers (II)

question: given n, compute F_n.
answer:

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
fib(n):
```

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
fib(n):
    if n = 0
        return 0
```

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
fib(n):
    if n = 0
        return 0
    elif n = 1
        return 1
    else
        return fib(n − 1)
```

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

correctness:

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
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:

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

correctness: clear

complexity:

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
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*.

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
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:

```
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

- $T(0) = 0$,

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
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:

```
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:

```
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

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

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
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:

```
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:

```
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

- $T(0) = 0,\ T(1) = 0$
- $T(2) = 1,$
- $T(n) = T(n-1) + T(n-2)$
- $\implies T(n) = F_{n-1} = \Theta(\varphi^n)$

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

```
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

- $T(0) = 0,\ T(1) = 0$
- $T(2) = 1,$
- $T(n) = T(n-1) + T(n-2)$
- $\implies T(n) = F_{n-1} = \Theta(\varphi^n) \implies$ exponential time!

Fibonacci Numbers (III)

Fibonacci Numbers (III)

recursion tree:

Fibonacci Numbers (III)

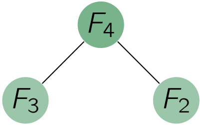
recursion tree: for F_4

Fibonacci Numbers (III)

recursion tree: for F_4

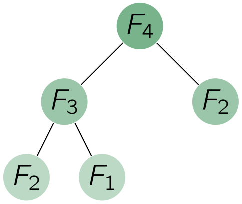
Fibonacci Numbers (III)

recursion tree: for F_4



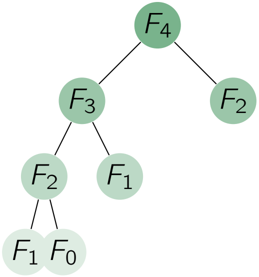
Fibonacci Numbers (III)

recursion tree: for F_4



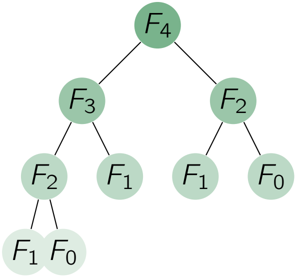
Fibonacci Numbers (III)

recursion tree: for F_4



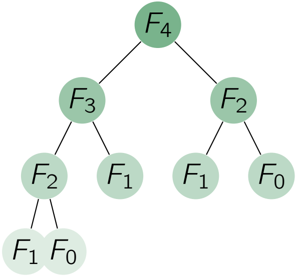
Fibonacci Numbers (III)

recursion tree: for F_4



Fibonacci Numbers (III)

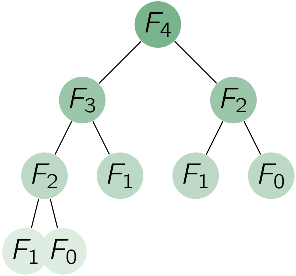
recursion tree: for F_4



dependency graph:

Fibonacci Numbers (III)

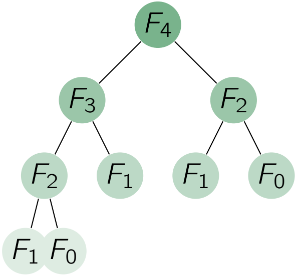
recursion tree: for F_4



dependency graph: for F_4

Fibonacci Numbers (III)

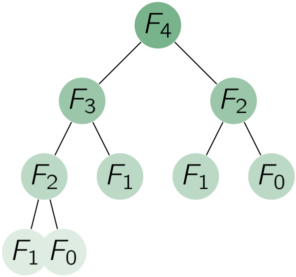
recursion tree: for F_4



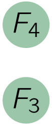
dependency graph: for F_4

Fibonacci Numbers (III)

recursion tree: for F_4

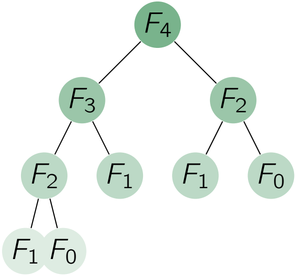


dependency graph: for F_4



Fibonacci Numbers (III)

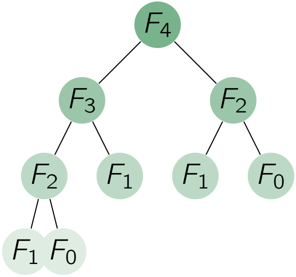
recursion tree: for F_4



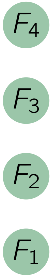
dependency graph: for F_4

Fibonacci Numbers (III)

recursion tree: for F_4

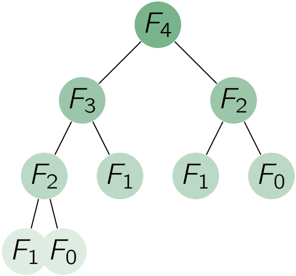


dependency graph: for F_4



Fibonacci Numbers (III)

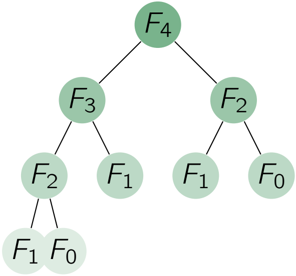
recursion tree: for F_4



dependency graph: for F_4

Fibonacci Numbers (III)

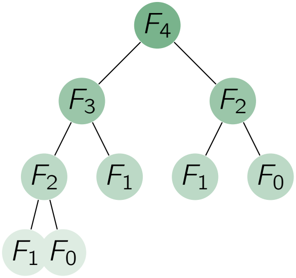
recursion tree: for F_4



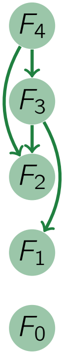
dependency graph: for F_4

Fibonacci Numbers (III)

recursion tree: for F_4

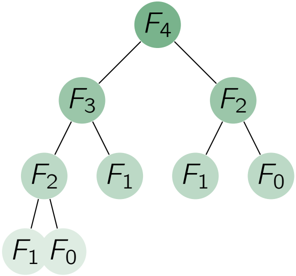


dependency graph: for F_4

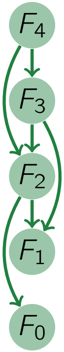


Fibonacci Numbers (III)

recursion tree: for F_4



dependency graph: for F_4



Fibonacci Numbers (IV)

Fibonacci Numbers (IV)

iterative algorithm:

Fibonacci Numbers (IV)

iterative algorithm:

```
fib-iter(n):
```

Fibonacci Numbers (IV)

iterative algorithm:

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

Fibonacci Numbers (IV)

iterative algorithm:

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

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
```

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]
```

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]
```

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 $F[n]$

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:

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:

```
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:

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

complexity: $O(n)$ additions

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

complexity: $O(n)$ additions

remarks:

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

complexity: $O(n)$ additions

remarks:

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

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: clear

complexity: $O(n)$ additions

remarks:

- $F_n = \Theta(\varphi^n) \implies F_n$ takes $\Theta(n)$ bits

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

complexity: $O(n)$ additions

remarks:

- $F_n = \Theta(\varphi^n) \implies F_n$ takes $\Theta(n)$ bits $\implies$ each addition takes $\Theta(n)$ steps

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

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:

Memoization

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

Memoization

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.

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:

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,

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)

Memoization (II)

question:

Memoization (II)

question: how to memoize exactly?

Memoization (II)

question: how to memoize exactly?

```
fib(n):
```

Memoization (II)

question: how to memoize exactly?

```
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
```

Memoization (II)

question: how to memoize exactly?

fib(n):
  if $n = 0$
    return 0
  if $n = 1$
    return 1
  if fib(n) was previously computed

Memoization (II)

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)
```

Memoization (II)

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
```

Memoization (II)

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($n-2$)

Memoization (II)

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)
```

Memoization (II)

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:

Memoization (II)

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?

Memoization (II)

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:*

Memoization (II)

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!

Memoization (II)

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!
- *implicitly:*

Memoization (II)

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!
- *implicitly:* allow clever data structures to do this automatically

Memoization (III)

Memoization (III)

global $\texttt{F}[\cdot]$

Memoization (III)

```
global F[·]
fib(n):
```

Memoization (III)

```
global F[·]
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
```

Memoization (III)

```
global F[·]
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
```

Memoization (III)

```
global F[·]
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
```

Memoization (III)

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

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)
```

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]
```

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]
```

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:

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

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

Memoization (III)

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

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

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:

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

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 an dictionary

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 an dictionary
  - requires *less* deliberation on problem structure,

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 an dictionary
  - requires *less* deliberation on problem structure, and can be less efficient

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 an dictionary
  - requires *less* deliberation on problem structure, and can be less efficient
  - sometimes can be done automatically by functional programming languages (LISP, etc.)

Fibonacci Numbers (V)

Fibonacci Numbers (V)

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

Fibonacci Numbers (V)

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

```
fib-iter(n):
```

Fibonacci Numbers (V)

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

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

Fibonacci Numbers (V)

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

`fib-iter`(n)`:`
  if $n = 0$
    return 0
  $F_{\texttt{prevprev}} = 0$

Fibonacci Numbers (V)

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

fib-iter(n):
  if $n = 0$
    return 0
  $F_{\texttt{prevprev}} = 0$
  if $n = 1$
    return 1

Fibonacci Numbers (V)

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
```

Fibonacci Numbers (V)

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

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

Fibonacci Numbers (V)

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
```

Fibonacci Numbers (V)

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

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

Fibonacci Numbers (V)

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

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

Fibonacci Numbers (V)

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
```

Fibonacci Numbers (V)

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
```

Fibonacci Numbers (V)

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:

Fibonacci Numbers (V)

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

Fibonacci Numbers (V)

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:

Fibonacci Numbers (V)

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,

Fibonacci Numbers (V)

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, $O(1)$ numbers stored

Fibonacci Numbers (V)

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, $O(1)$ numbers stored $\implies$ $O(n)$ *bits* stored

Memoization (IV)

Memoization (IV)

Definition

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

Memoization (IV)

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,

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.

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.

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.

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.

Edit Distance

Edit Distance

Definition

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ.

Edit Distance

Definition

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

Edit Distance

Definition

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

Edit Distance

Definition

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

Example

Edit Distance

Definition

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

Example

money

Edit Distance

Definition

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

Example

money $\rightarrow$

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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 $\rightarrow$

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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

Edit Distance

Definition

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

Example

$\underline{\text{m}}\text{oney} \rightarrow \text{bone}\underline{\text{y}} \rightarrow \text{bon}\underline{\text{e}} \rightarrow$

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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

Edit Distance

Definition

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

Example

m̲oney $\rightarrow$ boney̲ $\rightarrow$ bone̲ $\rightarrow$ bon̲a $\rightarrow$

Edit Distance

Definition

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

Example

$\underline{\text{m}}\text{oney} \rightarrow \text{bone}\underline{\text{y}} \rightarrow \text{bon}\underline{\text{e}} \rightarrow \text{bo}\underline{\text{n}}\text{a} \rightarrow \text{boa}$

Edit Distance

Definition

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

Example

money $\to$ boney $\to$ bone $\to$ bona $\to$ bo_a $\to$

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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

Edit Distance

Definition

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

Example

$\underline{m}$oney $\rightarrow$ bone$\underline{y}$ $\rightarrow$ bon$\underline{e}$ $\rightarrow$ bo$\underline{n}$a $\rightarrow$ bo$\underline{\ }$a $\rightarrow$ boba $\implies$ edit distance ≤ 5

Edit Distance

Definition

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

Example

$\underline{\text{m}}\text{oney} \to \text{bone}\underline{\text{y}} \to \text{bon}\underline{\text{e}} \to \text{bo}\underline{\text{n}}\text{a} \to \text{bo}\underline{\ \_\ }\text{a} \to \text{boba} \implies \text{edit distance} \leq 5$

remarks:

Edit Distance

Definition

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

Example

$\underline{\text{m}}\text{oney} \rightarrow \text{bone}\underline{\text{y}} \rightarrow \text{bon}\underline{\text{e}} \rightarrow \text{bo}\underline{\text{n}}\text{a} \rightarrow \text{bo}\underline{\ \ }\text{a} \rightarrow \text{boba} \implies \text{edit distance} \leq 5$

remarks:

- edit distance ≤ 4

Edit Distance

Definition

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

Example

$\underline{m}$oney $\to$ bone$\underline{y}$ $\to$ bon$\underline{e}$ $\to$ bo$\underline{n}$a $\to$ bo$\underline{\ }$a $\to$ boba $\Longrightarrow$ edit distance ≤ 5

remarks:

- edit distance ≤ 4
- intermediate strings can be arbitrary in $\Sigma^\star$

Edit Distance (II)

Edit Distance (II)

Definition

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ.

Edit Distance (II)

Definition

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

Edit Distance (II)

Definition

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

- an index could be empty,

Edit Distance (II)

Definition

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

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

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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:

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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 $j < j'$,

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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 $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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 $j > j'$

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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 $x_i \neq y_j$.

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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
```

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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
```

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

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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
```

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

Edit Distance (III)

Edit Distance (III)

question:

Edit Distance (III)

question: given two strings $x, y \in \Sigma^\star$,

Edit Distance (III)

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

Edit Distance (III)

question: given two strings $x, y \in \Sigma^{\star}$, compute their edit distance

Lemma

Edit Distance (III)

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

Lemma

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

Edit Distance (III)

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

Lemma

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

Proof.

Edit Distance (III)

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

Lemma

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

Proof.

Exercise. □

Edit Distance (III)

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

Lemma

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

Proof.

Exercise. □

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

Edit Distance (III)

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

Lemma

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

Proof.

Exercise. □

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

Edit Distance (III)

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

Lemma

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

Proof.

Exercise. □

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

remarks:

- can *also* ask to compute the alignment itself

Edit Distance (III)

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

Lemma

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

Proof.

Exercise. □

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

remarks:

- can *also* ask to compute the alignment itself
- widely solved in practice,

Edit Distance (III)

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

Lemma

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

Proof.

Exercise. □

question: given two strings $x, y \in \Sigma^\star$, 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

Edit Distance (IV)

Edit Distance (IV)

Lemma

Edit Distance (IV)

Lemma

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

Edit Distance (IV)

Lemma

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

Edit Distance (IV)

Lemma

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

Edit Distance (IV)

Lemma

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

$$\mathrm{dist}(x \circ a,$$

Edit Distance (IV)

Lemma

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

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

Edit Distance (IV)

Lemma

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

$$\text{dist}(x \circ a, y \circ b) = \min \left\{ \right.$$

Edit Distance (IV)

Lemma

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

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

Edit Distance (IV)

Lemma

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

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

Edit Distance (IV)

Lemma

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

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

Edit Distance (IV)

Lemma

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

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

Proof.

Edit Distance (IV)

Lemma

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

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

Proof.

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

Edit Distance (IV)

Lemma

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

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

Proof.

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

- a aligns to b,

Edit Distance (IV)

Lemma

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

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

Proof.

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

- a aligns to b, with cost $\mathbb{1}[\![a \neq b]\!]$

Edit Distance (IV)

Lemma

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

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

Proof.

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

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

Edit Distance (IV)

Lemma

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

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

Proof.

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

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

Edit Distance (IV)

Lemma

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

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

Proof.

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

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

Edit Distance (IV)

Lemma

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

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

Proof.

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

- a aligns to b, with cost $\mathbb{1}[\![a \neq b]\!]$
- a is deleted, with cost 1
- b is deleted, with cost 1 □

Edit Distance (V)

Edit Distance (V)

recursive algorithm:

Edit Distance (V)

recursive algorithm:

`dist(`x

Edit Distance (V)

recursive algorithm:

$$\texttt{dist}(x = x_1 x_2 \cdots x_n,$$

Edit Distance (V)

recursive algorithm:

$$\texttt{dist}(x = x_1 x_2 \cdots x_n, y$$

Edit Distance (V)

recursive algorithm:

$$\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)$$

Edit Distance (V)

recursive algorithm:

$$\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)$$

$$\textbf{if } \; n = 0, \; \textbf{return } \; m$$

Edit Distance (V)

recursive algorithm:

$\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)$

if $n = 0$, **return** m

if $m = 0$, **return** n

Edit Distance (V)

recursive algorithm:

$$
\begin{aligned}
&\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m) \\
&\quad \textbf{if } \ n = 0\,, \ \textbf{return } \ m \\
&\quad \textbf{if } \ m = 0\,, \ \textbf{return } \ n \\
&\quad d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!]
\end{aligned}
$$

Edit Distance (V)

recursive algorithm:

$$\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)$$
$$\quad \textbf{if } n = 0, \ \textbf{return } m$$
$$\quad \textbf{if } m = 0, \ \textbf{return } n$$
$$\quad d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!]$$
$$\quad d_2 = \textbf{dist}(x_{<n}, y) + 1$$

Edit Distance (V)

recursive algorithm:

$$
\begin{aligned}
&\texttt{dist}(x = x_1x_2\cdots x_n, y = y_1y_2\cdots y_m)\\
&\quad \textbf{if } n = 0,\ \textbf{return } m\\
&\quad \textbf{if } m = 0,\ \textbf{return } n\\
&\quad d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!]\\
&\quad d_2 = \textbf{dist}(x_{<n}, y) + 1\\
&\quad d_3 = \textbf{dist}(x, y_{<m}) + 1
\end{aligned}
$$

Edit Distance (V)

recursive algorithm:

$$\begin{aligned}
&\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m) \\
&\quad \textbf{if } \; n = 0, \; \textbf{return } \; m \\
&\quad \textbf{if } \; m = 0, \; \textbf{return } \; n \\
&\quad d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!] \\
&\quad d_2 = \textbf{dist}(x_{<n}, y) + 1 \\
&\quad d_3 = \textbf{dist}(x, y_{<m}) + 1 \\
&\quad \textbf{return } \; \min(d_1, d_2, d_3)
\end{aligned}$$

Edit Distance (V)

recursive algorithm:

```
dist(x = x_1 x_2 ··· x_n, y = y_1 y_2 ··· y_m)
    if n = 0, return m
    if m = 0, return n
    d_1 = dist(x_{<n}, y_{<m}) + 1[[x_n ≠ y_m]]
    d_2 = dist(x_{<n}, y) + 1
    d_3 = dist(x, y_{<m}) + 1
    return min(d_1, d_2, d_3)
```

$\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)$

- **if** $n = 0$, **return** m
- **if** $m = 0$, **return** n
- $d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!]$
- $d_2 = \textbf{dist}(x_{<n}, y) + 1$
- $d_3 = \textbf{dist}(x, y_{<m}) + 1$
- **return** $\min(d_1, d_2, d_3)$

Edit Distance (V)

recursive algorithm:

$\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)$

- **if** $n = 0$, **return** m
- **if** $m = 0$, **return** n
- $d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!]$
- $d_2 = \textbf{dist}(x_{<n}, y) + 1$
- $d_3 = \textbf{dist}(x, y_{<m}) + 1$
- **return** $\min(d_1, d_2, d_3)$

correctness:

Edit Distance (V)

recursive algorithm:

$\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m)$
    if $n = 0$, **return** m
    if $m = 0$, **return** n
    $d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!]$
    $d_2 = \textbf{dist}(x_{<n}, y) + 1$
    $d_3 = \textbf{dist}(x, y_{<m}) + 1$
    return $\min(d_1, d_2, d_3)$

correctness: clear

Edit Distance (V)

recursive algorithm:

$$
\begin{array}{l}
\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m) \\
\quad \textbf{if } n = 0, \ \textbf{return } m \\
\quad \textbf{if } m = 0, \ \textbf{return } n \\
\quad d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!] \\
\quad d_2 = \textbf{dist}(x_{<n}, y) + 1 \\
\quad d_3 = \textbf{dist}(x, y_{<m}) + 1 \\
\quad \textbf{return } \min(d_1, d_2, d_3)
\end{array}
$$

correctness: clear

complexity:

Edit Distance (V)

recursive algorithm:

$$\begin{aligned}
&\texttt{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_m) \\
&\qquad \textbf{if } n = 0, \ \textbf{return } m \\
&\qquad \textbf{if } m = 0, \ \textbf{return } n \\
&\qquad d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!] \\
&\qquad d_2 = \textbf{dist}(x_{<n}, y) + 1 \\
&\qquad d_3 = \textbf{dist}(x, y_{<m}) + 1 \\
&\qquad \textbf{return } \min(d_1, d_2, d_3)
\end{aligned}$$

correctness: clear

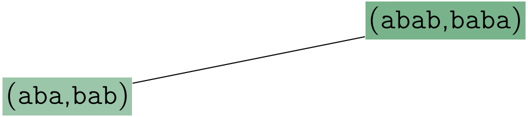
complexity: ???

Edit Distance (VI)

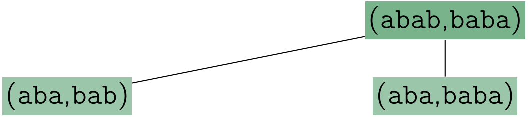
Edit Distance (VI)

(abab,baba)

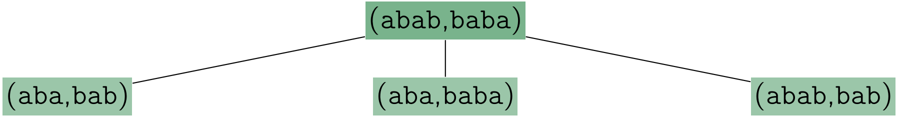
Edit Distance (VI)



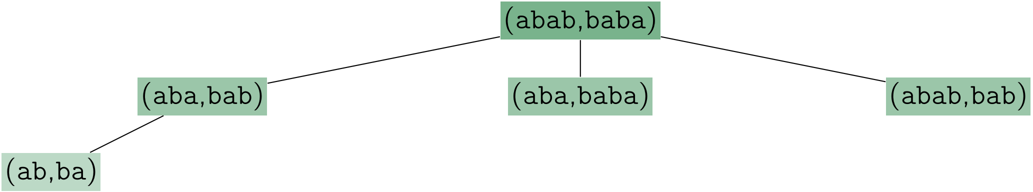
Edit Distance (VI)



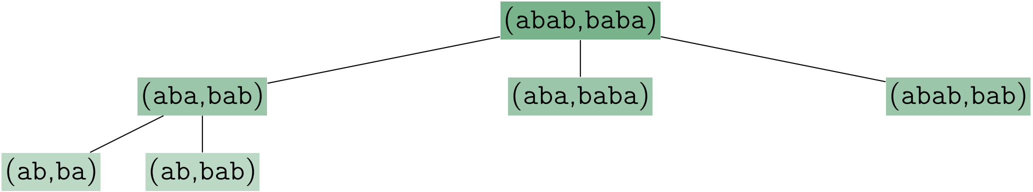
Edit Distance (VI)



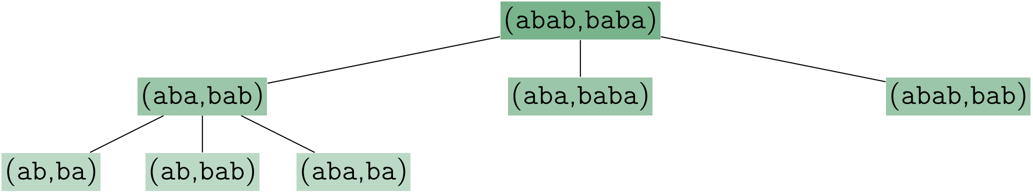
Edit Distance (VI)



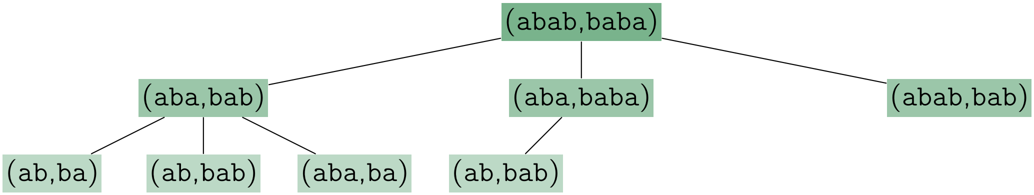
Edit Distance (VI)



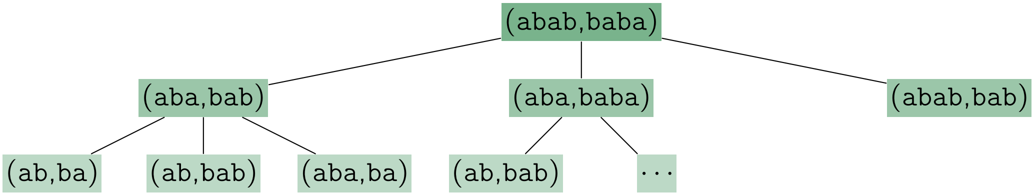
Edit Distance (VI)



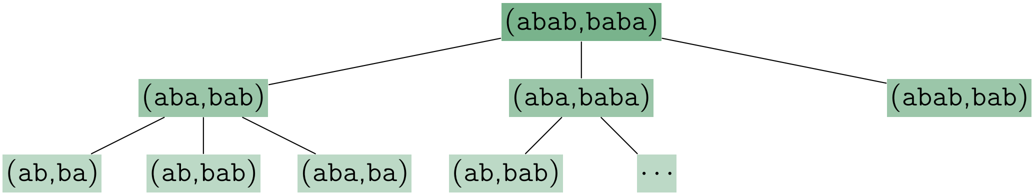
Edit Distance (VI)



Edit Distance (VI)

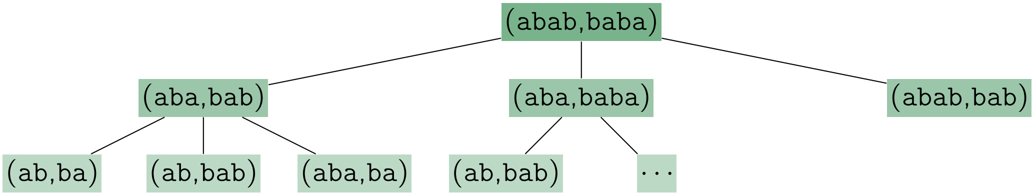


Edit Distance (VI)



(ab,bab) is *repeated*!

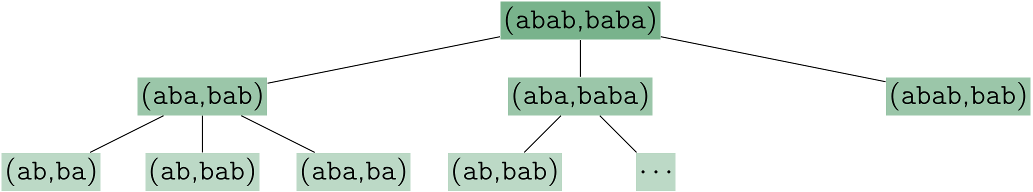
Edit Distance (VI)



(ab,bab) is *repeated*!

memoization:

Edit Distance (VI)



(ab,bab) is *repeated*!

memoization: define subproblem (i, j) as computing $\text{dist}(x_{\leq i}, y_{\leq j})$

Edit Distance (VII)

Edit Distance (VII)

memoized algorithm:

Edit Distance (VII)

memoized algorithm:

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

Edit Distance (VII)

memoized algorithm:

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

`dist(`$x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m,$

Edit Distance (VII)

memoized algorithm:

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

`dist`$(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i,j))$

Edit Distance (VII)

memoized algorithm:

global $d[\cdot][\cdot]$
`dist`$(x_1x_2\cdots x_n, y_1y_2\cdots y_m, (i,j))$
  if $d[i][j]$ initialized

Edit Distance (VII)

memoized algorithm:

```
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]
```

Edit Distance (VII)

memoized algorithm:

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

`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 $i = 0$

Edit Distance (VII)

memoized algorithm:

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

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m, (i, j))$

  if $d[i][j]$ initialized

    return $d[i][j]$

  if $i = 0$

    $d[i][j] = j$

Edit Distance (VII)

memoized algorithm:

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 $j = 0$

Edit Distance (VII)

memoized algorithm:

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

Edit Distance (VII)

memoized algorithm:

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

Edit Distance (VII)

memoized algorithm:

> **global** $d[\cdot][\cdot]$
> `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**
> $d_1 = \textbf{dist}(x, y, (i-1, j-1)) + \mathbb{1}[\![x_i \neq y_j]\!]$

Edit Distance (VII)

memoized algorithm:

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 $j = 0$
        $d[i][j] = i$
    else
        $d_1 = \textbf{dist}(x, y, (i-1, j-1)) + \mathbb{1}[\![x_i \neq y_j]\!]$
        $d_2 = \textbf{dist}(x, y, (i-1, j)) + 1$

Edit Distance (VII)

memoized algorithm:

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

`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

    $d_1 = \textbf{dist}(x, y, (i-1, j-1)) + \mathbb{1}[\![x_i \neq y_j]\!]$

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

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

Edit Distance (VII)

memoized algorithm:

global $d[\cdot][\cdot]$
`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
        $d_1 = \textbf{dist}(x, y, (i-1, j-1)) + \mathbb{1}[\![x_i \neq y_j]\!]$
        $d_2 = \textbf{dist}(x, y, (i-1, j)) + 1$
        $d_3 = \textbf{dist}(x, y, (i, j-1)) + 1$
        $d[i][j] = \min(d_1, d_2, d_3)$

Edit Distance (VII)

memoized algorithm:

```
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
        d3 = dist(x, y, (i, j − 1)) + 1
        d[i][j] = min(d1, d2, d3)
    return d[i][j]
```

memoized algorithm:

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 $j = 0$

$d[i][j] = i$

else

$d_1 = \textbf{dist}(x, y, (i-1, j-1)) + \mathbb{1}[\![x_i \neq y_j]\!]$

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

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

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

return $d[i][j]$

Edit Distance (VIII)

Edit Distance (VIII)

dependency graph:

Edit Distance (VIII)

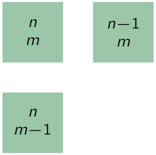
dependency graph:

Edit Distance (VIII)

dependency graph:

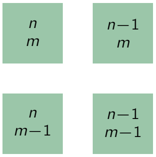
Edit Distance (VIII)

dependency graph:



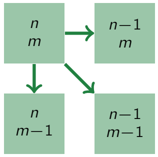
Edit Distance (VIII)

dependency graph:



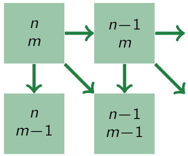
Edit Distance (VIII)

dependency graph:



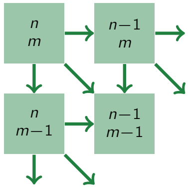
Edit Distance (VIII)

dependency graph:



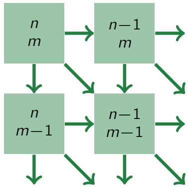
Edit Distance (VIII)

dependency graph:



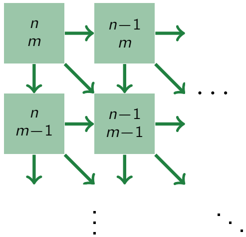
Edit Distance (VIII)

dependency graph:



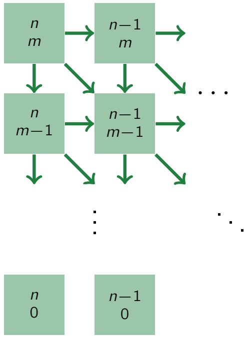
Edit Distance (VIII)

dependency graph:



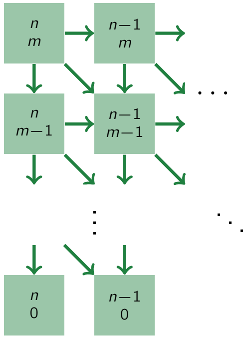
Edit Distance (VIII)

dependency graph:



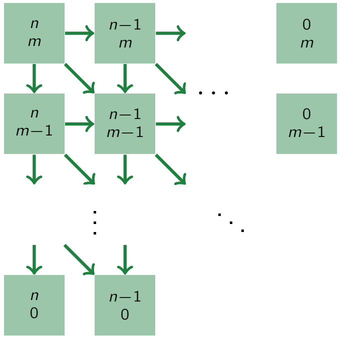
Edit Distance (VIII)

dependency graph:



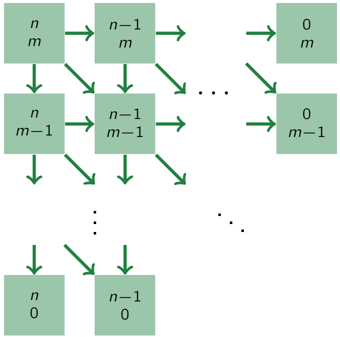
Edit Distance (VIII)

dependency graph:



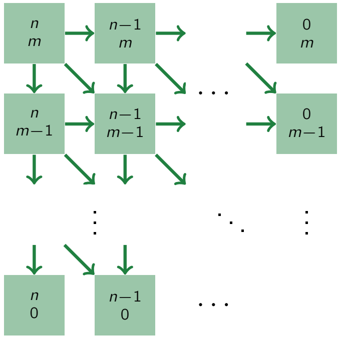
Edit Distance (VIII)

dependency graph:



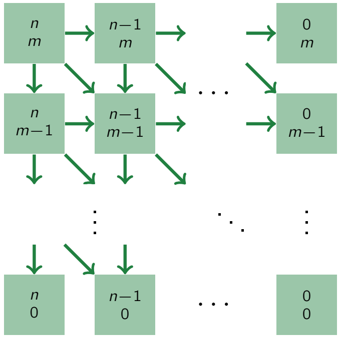
Edit Distance (VIII)

dependency graph:



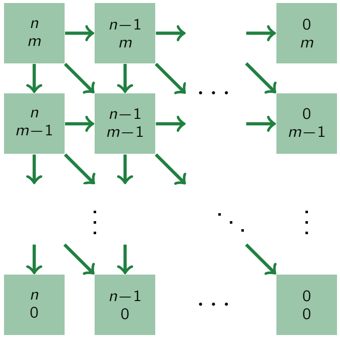
Edit Distance (VIII)

dependency graph:



Edit Distance (VIII)

dependency graph:



Edit Distance (IX)

Edit Distance (IX)

iterative algorithm:

Edit Distance (IX)

iterative algorithm:

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

Edit Distance (IX)

iterative algorithm:

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

  for $0 \le i \le n$

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
```

iterative algorithm:

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

for $0 \leq i \leq n$

$d[i][0] = i$

for $0 \leq j \leq m$

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
    for 0 ≤ j ≤ m
        d[0][j] = j
```

Edit Distance (IX)

iterative algorithm:

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

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 $0 \leq j \leq m$

Edit Distance (IX)

iterative algorithm:

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

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 $0 \leq j \leq m$

$d[i][j] =$

Edit Distance (IX)

iterative algorithm:

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

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 $0 \leq j \leq m$

$$d[i][j] = \min \left\{ \vphantom{\begin{array}{c} a \\ b \\ c \end{array}} \right.$$

Edit Distance (IX)

iterative algorithm:

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

for $0 \le i \le n$

$d[i][0] = i$

for $0 \le j \le m$

$d[0][j] = j$

for $0 \le i \le n$

for $0 \le j \le m$

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

Edit Distance (IX)

iterative algorithm:

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

for $0 \le i \le n$

$d[i][0] = i$

for $0 \le j \le m$

$d[0][j] = j$

for $0 \le i \le n$

for $0 \le j \le m$

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

Edit Distance (IX)

iterative algorithm:

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

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 $0 \leq j \leq m$

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

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
```

$\texttt{dist}(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

for $0 \le i \le n$

$\quad d[i][0] = i$

for $0 \le j \le m$

$\quad d[0][j] = j$

for $0 \le i \le n$

$\quad$ **for** $0 \le j \le m$

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

return $d[n][m]$

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
    for 0 ≤ j ≤ m
        d[0][j] = j
    for 0 ≤ i ≤ n
        for 0 ≤ j ≤ m
```

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

return $d[n][m]$

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
    for 0 ≤ j ≤ m
        d[0][j] = j
    for 0 ≤ i ≤ n
        for 0 ≤ j ≤ m
```

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

return $d[n][m]$

correctness:

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
    for 0 ≤ j ≤ m
        d[0][j] = j
    for 0 ≤ i ≤ n
        for 0 ≤ j ≤ m
            d[i][j] = min { d[i−1][j−1] + 𝟙⟦x_i ≠ y_j⟧
                            d[i−1][j] + 1
                            d[i][j−1] + 1
    return d[n][m]
```

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

correctness: clear

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
    for 0 ≤ j ≤ m
        d[0][j] = j
    for 0 ≤ i ≤ n
        for 0 ≤ j ≤ m
```

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

```
    return d[n][m]
```

correctness: clear

complexity:

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
    for 0 ≤ j ≤ m
        d[0][j] = j
    for 0 ≤ i ≤ n
        for 0 ≤ j ≤ m
            d[i][j] = min { d[i−1][j−1] + 𝟙[[x_i ≠ y_j]]
                            d[i−1][j] + 1
                            d[i][j−1] + 1
    return d[n][m]
```

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

correctness: clear

complexity: $O(nm)$ time,

Edit Distance (IX)

iterative algorithm:

```
dist(x_1 x_2 ··· x_n, y_1 y_2 ··· y_m)
    for 0 ≤ i ≤ n
        d[i][0] = i
    for 0 ≤ j ≤ m
        d[0][j] = j
    for 0 ≤ i ≤ n
        for 0 ≤ j ≤ m
            d[i][j] = min { d[i−1][j−1] + 1[[x_i ≠ y_j]]
                            d[i−1][j] + 1
                            d[i][j−1] + 1
    return d[n][m]
```

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

correctness: clear

complexity: $O(nm)$ time, $O(nm)$ space

Edit Distance (X)

Edit Distance (X)

Corollary

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ can compute the minimum cost alignment

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ *can compute the minimum cost alignment in* $O(nm)$*-time and* $O(nm)$*-space.*

Proof.

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Proof.

Exercise.

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Proof.

Exercise. *Hint:*

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ *can compute the minimum cost alignment in* $O(nm)$*-time and* $O(nm)$*-space.*

Proof.

Exercise. *Hint:* follow *how* each subproblem was solved.

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ can compute the minimum cost alignment in $O(nm)$-time and $O(nm)$-space.

Proof.

Exercise. *Hint:* follow *how* each subproblem was solved. $\square$

Dynamic Programming

template:

Dynamic Programming

template:

- develop recursive algorithm

Dynamic Programming

template:

- develop recursive algorithm
- understand structure of subproblems

Dynamic Programming

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

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

Knapsack

the knapsack problem:

Knapsack

the knapsack problem:

input: knapsack capacity $W \in \mathbb{N}$

Knapsack

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 $v_1, \ldots, v_n \in \mathbb{N}$.

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

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,

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

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

$$\max_{S \subseteq [n]} \quad \sum_{i \in S} v_i$$

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

$$\max_{\substack{S \subseteq [n] \\ \sum_{i \in S} w_i \leq W}} \sum_{i \in S} v_i$$

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

$$\max_{\substack{S \subseteq [n] \\ \sum_{i \in S} w_i \leq W}} \sum_{i \in S} v_i$$

remarks:

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

$$\max_{\substack{S \subseteq [n] \\ \sum_{i \in S} w_i \leq W}} \sum_{i \in S} v_i$$

remarks:

- prototypical problem in *combinatorial optimization*,

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

$$\max_{\substack{S \subseteq [n] \\ \sum_{i \in S} w_i \leq W}} \sum_{i \in S} v_i$$

remarks:

- prototypical problem in *combinatorial optimization*, can be generalized in numerous ways

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

$$\max_{\substack{S \subseteq [n] \\ \sum_{i \in S} w_i \leq W}} \sum_{i \in S} v_i$$

remarks:

- prototypical problem in *combinatorial optimization*, can be generalized in numerous ways
- needs to be solved in practice

Knapsack (II)

Knapsack (II)

Example

Knapsack (II)

Example

item	1	2	3	4	5
weight	1	2	5	6	7
value	1	6	18	22	28

Knapsack (II)

Example

item	1	2	3	4	5
weight	1	2	5	6	7
value	1	6	18	22	28

For $W = 11$,

Knapsack (II)

Example

item	1	2	3	4	5
weight	1	2	5	6	7
value	1	6	18	22	28

For $W = 11$, the best is $\{3, 4\}$ giving value 40.

Knapsack (II)

Example

item	1	2	3	4	5
weight	1	2	5	6	7
value	1	6	18	22	28

For $W = 11$, the best is $\{3, 4\}$ giving value 40.

Definition

Knapsack (II)

Example

item	1	2	3	4	5
weight	1	2	5	6	7
value	1	6	18	22	28

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,

Knapsack (II)

Example

item	1	2	3	4	5
weight	1	2	5	6	7
value	1	6	18	22	28

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)

Knapsack (III)

item	1	2	3	4	5
value	1	6	16	22	28
weight	1	2	5	6	7

Knapsack (III)

item	1	2	3	4	5
value	1	6	16	22	28
weight	1	2	5	6	7

and weight limit $W = 15$.

Knapsack (III)

item	1	2	3	4	5
value	1	6	16	22	28
weight	1	2	5	6	7

and weight limit $W = 15$. What is the best solution value?

Knapsack (III)

item	1	2	3	4	5
value	1	6	16	22	28
weight	1	2	5	6	7

and weight limit $W = 15$. What is the best solution value?

(a) 22

(b) 28

(c) 38

(d) 50

(e) 56

Knapsack (IV)

Knapsack (IV)

greedy approaches:

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

For $W = 2$,

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

For $W = 2$, greedy-value will pick $\{3\}$,

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$,

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$, greedy-weight will pick $\{1\}$,

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  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:

item	1	2	3
value	2	2	3
weight	1	1	2

For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

  For $W = 4$,

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

  For $W = 4$, greedy-value will pick $\{3\}$,

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

For $W = 4$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

  For $W = 4$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

remark:

Knapsack (IV)

greedy approaches:

- greedily select by maximum value:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

  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:

item	1	2	3
value	2	2	3
weight	1	1	2

  For $W = 2$, greedy-value will pick $\{3\}$, but optimal is $\{1, 2\}$.

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

  For $W = 2$, greedy-weight will pick $\{1\}$, but optimal is $\{2\}$.

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

  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 (V)

Knapsack (V)

Lemma

Knapsack (V)

Lemma

Consider the instance W, $(v_i)_{i=1}^n$, *and* $(w_i)_{i=1}^n$,

Knapsack (V)

Lemma

Consider the instance W, $(v_i)_{i=1}^{n}$, and $(w_i)_{i=1}^{n}$, with optimal solution $S \subseteq [n]$.

Knapsack (V)

Lemma

Consider the instance W, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

Knapsack (V)

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 $n \notin S$,*

Knapsack (V)

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* $S \subseteq [n-1]$

Knapsack (V)

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 $(W, (v_i)_{i<n}, (w_i)_{i<n})$.*

Knapsack (V)

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* $n \in S$,

Knapsack (V)

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 $S \setminus \{n\} \subseteq [n-1]$*

Knapsack (V)

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* $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

Knapsack (V)

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.

Knapsack (V)

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 $(W, (v_i)_{i<n}, (w_i)_{i<n})$,

Knapsack (V)

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

Knapsack (V)

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 $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$,

Knapsack (V)

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

Knapsack (V)

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 □

Knapsack (VI)

Knapsack (VI)

Corollary

Knapsack (VI)

Corollary

Fix an instance W*,* $v_1, \ldots, v_n$*, and* $w_1, \ldots, w_n$*.*

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$.

Knapsack (VI)

Corollary

Fix an instance W*,* $v_1, \ldots, v_n$*, and* $w_1, \ldots, w_n$*. Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w*,* $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$*. Then,*

$$\mathrm{OPT}(i, w) =$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 \\ \\ \\ \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ & \\ & \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W*,* $v_1, \ldots, v_n$*, and* $w_1, \ldots, w_n$*. Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w*,* $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$*. Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & \\ & \\ & \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\mathrm{OPT}(i, w)$ to be the maximum value of the knapsack instance w, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & w_i > w \\ & \\ & \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & w_i > w \\ \max \left\{ \right. & \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & w_i > w \\ \max \begin{cases} \mathrm{OPT}(i-1, w) \\ \end{cases} & \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i,w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1,w) & w_i > w \\ \max \begin{cases} \mathrm{OPT}(i-1,w) \\ \mathrm{OPT}(i-1,w-w_i) + v_i \end{cases} & \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & w_i > w \\ \max \begin{cases} \mathrm{OPT}(i-1, w) \\ \mathrm{OPT}(i-1, w - w_i) + v_i \end{cases} & \textit{else} \end{cases}$$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & w_i > w \\ \max \begin{cases} \mathrm{OPT}(i-1, w) \\ \mathrm{OPT}(i-1, w - w_i) + v_i \end{cases} & \textit{else} \end{cases}$$

$\Longrightarrow$ from instance W, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\mathrm{OPT}(i, w)$ to be the maximum value of the knapsack instance w, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & w_i > w \\ \max \begin{cases} \mathrm{OPT}(i-1, w) \\ \mathrm{OPT}(i-1, w - w_i) + v_i \end{cases} & \textit{else} \end{cases}$$

$\Longrightarrow$ from instance W, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$ we generate $O(n \cdot W)$-many subproblems

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, *and* $w_1, \ldots, w_n$. *Define* $\mathrm{OPT}(i, w)$ *to be the maximum value of the knapsack instance* w, $v_1, \ldots, v_i$ *and* $w_1, \ldots, w_i$. *Then,*

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \mathrm{OPT}(i-1, w) & w_i > w \\ \max \begin{cases} \mathrm{OPT}(i-1, w) \\ \mathrm{OPT}(i-1, w - w_i) + v_i \end{cases} & \textit{else} \end{cases}$$

$\Longrightarrow$ from instance W, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$ we generate $O(n \cdot W)$-many subproblems $(i, w)_{i \in [n], w \leq W}$.

Knapsack (VII)

Knapsack (VII)

an iterative algorithm:

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\text{OPT}(i, w)$

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

for $0 \leq w \leq W$
  $M[0, w] = 0$

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 $1 \leq i \leq n$

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

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 $\text{OPT}(i, w)$

for $0 \leq w \leq W$
    $M[0, w] = 0$
for $1 \leq i \leq n$
    for $1 \leq w \leq W$
        if $w_i > w$

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 $\mathrm{OPT}(i, w)$

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 $\text{OPT}(i, w)$

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
            $M[i, w] = \max($

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
            $M[i, w] = \max(M[i-1, w],$

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

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 $\mathrm{OPT}(i, w)$

```
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)
```

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

for $0 \le w \le W$
    $M[0, w] = 0$
for $1 \le i \le n$
    for $1 \le w \le W$
        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:

Knapsack (VII)

an iterative algorithm: $M[i,w]$ will compute $\text{OPT}(i,w)$

for $0 \le w \le W$
  $M[0,w] = 0$
for $1 \le i \le n$
  for $1 \le w \le W$
    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

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:

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

```
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,

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

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 $O(n$

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 $O(n + \log W$

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

$$
\begin{array}{l}
\textbf{for } 0 \le w \le W \\
\quad M[0, w] = 0 \\
\textbf{for } 1 \le i \le n \\
\quad \textbf{for } 1 \le w \le W \\
\quad\quad \textbf{if } w_i > w \\
\quad\quad\quad M[i, w] = M[i-1, w] \\
\quad\quad \textbf{else} \\
\quad\quad\quad M[i, w] = \max(M[i-1, w], \\
\quad\quad\quad\quad\quad\quad M[i-1, w-w_i] + v_i)
\end{array}
$$

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 $\text{OPT}(i, w)$

$$
\begin{array}{l}
\textbf{for } 0 \le w \le W \\
\quad M[0, w] = 0 \\
\textbf{for } 1 \le i \le n \\
\quad \textbf{for } 1 \le w \le W \\
\quad\quad \textbf{if } w_i > w \\
\quad\quad\quad M[i, w] = M[i-1, w] \\
\quad\quad \textbf{else} \\
\quad\quad\quad M[i, w] = \max(M[i-1, w], \\
\quad\quad\quad\quad\quad\quad M[i-1, w-w_i] + v_i)
\end{array}
$$

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 $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$
- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime

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

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**:

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

$$
\begin{array}{l}
\textbf{for } 0 \le w \le W \\
\quad M[0, w] = 0 \\
\textbf{for } 1 \le i \le n \\
\quad \textbf{for } 1 \le w \le W \\
\quad\quad \textbf{if } w_i > w \\
\quad\quad\quad M[i, w] = M[i-1, w] \\
\quad\quad \textbf{else} \\
\quad\quad\quad M[i, w] = \max(M[i-1, w], \\
\quad\quad\quad\quad\quad\quad M[i-1, w-w_i] + v_i)
\end{array}
$$

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 (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

```
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

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:

Knapsack (VII)

an iterative algorithm: $M[i,w]$ will compute $\mathrm{OPT}(i,w)$

> **for** $0 \le w \le W$
> $M[0,w] = 0$
> **for** $1 \le i \le n$
>   **for** $1 \le w \le W$
>   **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: had to correctly *parameterize* knapsack sub-problems $(v_j)_{j \le i}$,$(w_j)_{j \le i}$ by *also* considering arbitrary w.

Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\mathrm{OPT}(i, w)$

```
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 $\Longrightarrow$ 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 $\Longrightarrow$ no efficient algorithm is expected to compute the exact optimum

punchline: had to correctly *parameterize* knapsack sub-problems $(v_j)_{j \le i}, (w_j)_{j \le i}$ by *also* considering arbitrary w. This is a common theme in dynamic programming problems.

Today

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

TOC