

Edit distance and
matrix chain multiplication

I. An algorithmic metric on strings

Version control

I. An algorithmic metric on strings

▼ 2  homework/homework-6.md 

⌕ @@ -7,6 +7,6 @@

7	2. Design and analyze an algorithm ^[^2] to compute the even distance between every pair of vertices when the edge weights are real-valued, and there are no negative cycles.	7	2. Design and analyze an algorithm ^[^2] to compute the even distance between every pair of vertices when the edge weights are real-valued, and there are no negative cycles.
8	2. Design and analyze an algorithm ^[^2] that, given a directed graph with real-valued edges, detects if there is a negative cycle.	8	2. Design and analyze an algorithm ^[^2] that, given a directed graph with real-valued edges, detects if there is a negative cycle.
9		9	
10	- ^[^1] : We pushed back the deadline to Friday because the homework was also prepared late.	10	+ ^[^1] : We pushed back the deadline to Friday because we were late in getting the homework up.
11	^[^2] : As always, the faster the better.	11	^[^2] : As always, the faster the better.
12		12	

Github highlights changes

I. An algorithmic metric on strings

```
2 homework/homework-6.md
@@ -7,6 +7,6 @@
7      2. Design and analyze an algorithm[^2] to compute the
   even distance between every pair of vertices when the edge
   weights are real-valued, and there are no negative cycles.
8      2. Design and analyze an algorithm[^2] that, given a
   directed graph with real-valued edges, detects if there is
   a negative cycle.
9
10 - [^1]: We pushed back the deadline to Friday because the
   homework was also prepared late.
11      [^2]: As always, the faster the better.
12
```

DIFF(1)

User Commands

DIFF(1)

NAME

diff - compare files line by line

SYNOPSIS

diff [OPTION]... FILES

DESCRIPTION

Compare files line by line.

-i --ignore-case

Ignore case differences in file contents.

--ignore-file-name-case

Ignore case when comparing file names.

--no-ignore-file-name-case

Consider case when comparing file names.

-E --ignore-tab-expansion

Ignore changes due to tab expansion.

-b --ignore-space-change

Ignore changes in the amount of white space.

-w --ignore-all-space

Ignore all white space.

-B --ignore-blank-lines

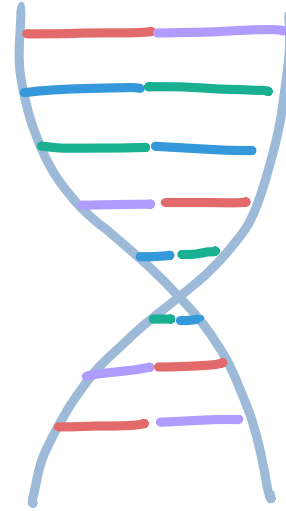
Ignore changes whose lines are all blank.

Genetics

I. An algorithmic metric on strings

A, C, G, T

A
C
G
T
G
C
T
A



how to measure similarity?

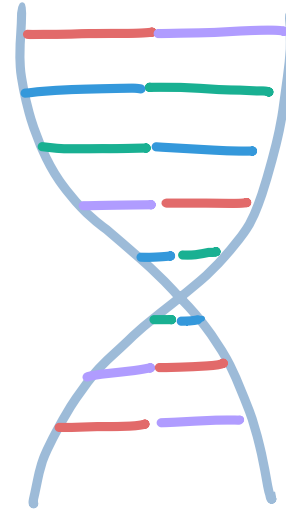
important for:

(a) analyzing hereditary traits

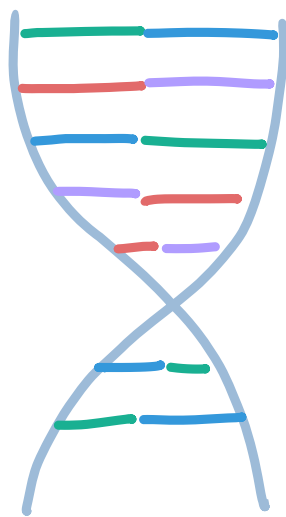
(b) distinguishing species

&

A
C
G
T
G
C
T
A



G
A
C
T
A
T
C
G



2 metrics for strings:

1. Hamming distance

for strings x, y w/ same length,

$$\text{Hamming}(x, y) = \left(\begin{array}{c} \# \text{ characters} \\ \text{differing in } x \text{ and } y \end{array} \right)$$

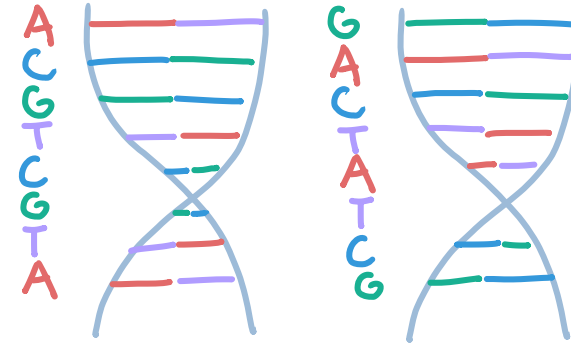
$x =$ ABCDEFG

x x x x

$y =$ ABCEFGD

Hamming dist. 4

```
2 homework/homework-6.md
18 @@ -7,6 +7,6 @@
7 2. Design and analyze an algorithm[*2] to compute the
  even distance between every pair of vertices when the edge
  weights are real-valued, and there are no negative cycles.
8 2. Design and analyze an algorithm[*2] that, given a
  directed graph with real-valued edges, detects if there is
  a negative cycle.
9
10 - [*1]: We pushed back the deadline to Friday because the
  homework was also prepared late.
11 [*2]: As always, the faster the better.
12
7 2. Design and analyze an algorithm[*2] to compute the
  even distance between every pair of vertices when the edge
  weights are real-valued, and there are no negative cycles.
8 2. Design and analyze an algorithm[*2] that, given a
  directed graph with real-valued edges, detects if there is
  a negative cycle.
9
10 + [*1]: We pushed back the deadline to Friday because we
  were late in getting the homework up.
11 [*2]: As always, the faster the better.
12
```



2 metrics for strings:

1. Hamming distance

$$\text{Hamming}(x, y) = \left(\begin{array}{c} \text{\# characters} \\ \text{differing in } x \text{ and } y \end{array} \right)$$

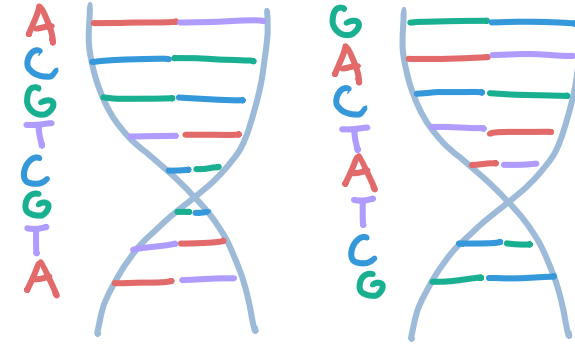
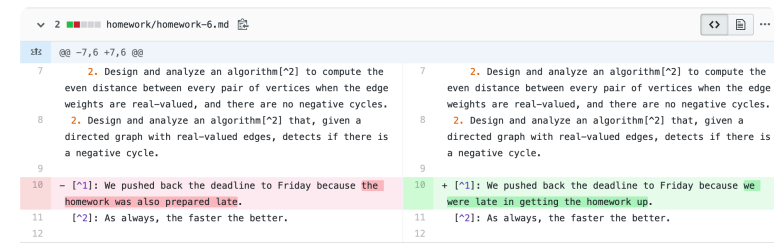
for strings x, y w/ same length

$x = \text{ABCDEFGG}$

$\times \times \times \times$

$y = \text{ABCEFGD}$

$\Rightarrow$ Hamming dist. 4



2. Edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

$x = \text{ABCDEF}^{\text{del}}\text{G}$

$y = \text{ABCEFGD}^{\text{ins}}$

$\Rightarrow$ edit distance 2

Hamming distance and edit distance

are both metrics: $d(x, y) \rightarrow 10$

- $\text{dist}(x, y) \geq 0$
- $\text{dist}(x, y) = 0 \iff x = y$
- $\text{dist}(x, y) = \text{dist}(y, x)$
- $\text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y)$ "triangle inequality"

Hamming distance

characters
differing in x and y

Edit distance

min # single character insertions,
deletions, and substitutions
to convert x into y .

1. Hamming distance

characters
differing in x and y

con: small misalignment
 $\Rightarrow$ huge Hamming dist

pro: easy, fast to compute

2. Edit distance

min # single character insertions,
deletions, and substitutions
to convert x into y .

pro: small misalignment
 $\Rightarrow$ small edit distance
(better model for many applications)

cons: more complicated,
how to compute?

Computing edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive
for all n by induction on n spec

① Recursive specification

$\text{diff}(x[1, \dots, m], y[1, \dots, n]):$ return minimum # of edits
to convert x to y

Computing edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive spec for all n by induction on n

① Recursive specification

$\text{edit}(x[1..m], y[1..n])$: Given two strings x and y , returns the minimum number of edit operations needed to convert x into y .

Computing edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive spec for all n by induction on n

① Recursive specification

$\text{edit}(x[1..m], y[1..n])$: Given two strings x and y , returns the minimum number of edit operations needed to convert x into y .

② Implement spec

$\text{edit}(x[1..m], y[1..n])$:

if $m=0$ or $n=0$

return $m+n$

if $x=y$ return 0 // $O(m+n)$

for $i=1, \dots, m$

let x' delete i th char from x

$A = 1 + \text{edit}(x', y)$

for $i=1, \dots, m$, $j=1, \dots, n$

let x' insert $y[j]$ into i th slot

$B = 1 + \text{edit}(x', y)$

similar for subs
 $i=1, \dots, m$, $j=1, \dots, n$,
substitute $x[i]$ w/ $y[j]$
recurse

Computing edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

① Recursive specification

$\text{edit}(x[1..m], y[1..n])$: Given two strings x and y , returns the minimum number of edit operations needed to convert x into y .

② Implement spec

$\text{edit}(x[1..m], y[1..n])$:

if $m=0$ or $n=0$

return $m+n$

~~if $x=y$ return 0~~ $O(m+n)$ ~~X~~

if $x[1]=y[1]$

return $\text{edit}(x[2..m], y[2..n])$

else $x[1] \neq y[1]$

$\min \left\{ \begin{array}{l} 1 + \text{edit}(x[2..m], y[2..n]) \\ 1 + \text{edit}(x[2..m], y) \\ 1 + \text{edit}(x, y[2..n]) \end{array} \right\}$

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive spec for all n by induction on n

Computing edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

① Recursive specification

$\text{edit}(x[1..m], y[1..n])$: Given two strings x and y , returns the minimum number of edit operations needed to convert x into y .

② Implement spec

$\text{edit}(x[1..k], y[1..\ell])$

1. If $k = 0$ or $\ell = 0$ then return $k + \ell$.
2. If $x[1] \neq y[1]$ then return the minimum of
 - A. $1 + \text{edit}(x[2..k], y[1..\ell])$
 - B. $1 + \text{edit}(x[1..k], y[2..\ell])$
 - C. $1 + \text{edit}(x[2..k], y[2..\ell])$
3. If $x[1] = y[1]$ then return the minimum of
 - A. $\text{edit}(x[2..k], y[2..\ell])$
 - B. $1 + \text{edit}(x[1..k], y[2..\ell])$
 - C. $1 + \text{edit}(x[2..k], y[1..\ell])$

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive
for all n by induction on n spec

// delete $x[1]$

// insert $y[1]$

// replace $x[1]$ with $y[1]$

// do nothing

// insert $y[1]$

// delete $x[1]$

Computing edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

① Recursive specification

$\text{edit}(x[1..m], y[1..n])$: Given two strings x and y , returns the minimum number of edit operations needed to convert x into y .

② Implement spec

$\text{edit}(x[1..k], y[1..l])$

1. If $k = 0$ or $l = 0$ then return $k + l$.

2. If $x[1] \neq y[1]$ then return the minimum of

A. $1 + \text{edit}(x[2..k], y[1..l])$

B. $1 + \text{edit}(x[1..k], y[2..l])$

C. $1 + \text{edit}(x[2..k], y[2..l])$

3. If $x[1] = y[1]$ then return the minimum of

A. $\text{edit}(x[2..k], y[2..l])$

B. $1 + \text{edit}(x[1..k], y[2..l])$

C. $1 + \text{edit}(x[2..k], y[1..l])$

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive spec for all n by induction on n

// delete $x[1]$

// insert $y[1]$

// replace $x[1]$ with $y[1]$

// do nothing

// insert $y[1]$

// delete $x[1]$

Running time: $T(n) \leq 3T(n-1) + 1 \leq 3^n$
 $\geq 3^{n/2}$ $O(3^n)$

Computing edit distance

min # single character insertions, deletions, and substitutions to convert x into y .

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive
for all n by induction on n spec

① Recursive specification

$\text{edit}(x[1..m], y[1..n])$: Given two strings x and y , returns the minimum number of edit operations needed to convert x into y .

② Implement spec

$\text{edit}(x[1..k], y[1..l])$

1. If $k = 0$ or $l = 0$ then return $k + l$.
2. If $x[1] \neq y[1]$ then return the minimum of
 - A. $1 + \text{edit}(x[2..k], y[1..l])$
 - B. $1 + \text{edit}(x[1..k], y[2..l])$
 - C. $1 + \text{edit}(x[2..k], y[2..l])$
3. If $x[1] = y[1]$ then return the minimum of
 - A. $\text{edit}(x[2..k], y[2..l])$
 - B. $1 + \text{edit}(x[1..k], y[2..l])$
 - C. $1 + \text{edit}(x[2..k], y[1..l])$

$$T(n) = 3T(n-1) + 1 = O(3^n)$$

Summary

edit distance is good model
can be computed recursively
recursive algo is exponential time

Q: How to make recursive
algo fast?

caching!

Caching edit distance

Idea:

- save answers to each subproblem
- never solve same subproblem twice.

min # single character insertions, deletions, and substitutions to convert x into y .

Designing a recursive algorithm

- ① Recursive specification
 - declares input/output of algo
 - "contract" algo must fulfill
 - induction hypothesis for correctness
- ② Implement spec
- ③ Prove correctness
 - argue algo satisfies recursive for all n by induction on n spec

1. index parameters to be more cache-friendly

① Recursive specification let $x[1..m]$, $y[1..n]$ be fixed.

$\text{edit}(i, j)$ = the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$.

2 Implement spec

$\text{edit}(i, j)$

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j$.
3. If $j > n$ then return $m - i$.
4. If $x[i] \neq y[j]$ then return $\min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}$.
5. Otherwise return $\min\{\text{edit}(i + 1, j + 1), 1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1)\}$.

Caching edit distance

Idea:

- save answers to each subproblem
- never solve same subproblem twice.

2. augment recursive code w/ caching

cached-edit(i, j)

/ Implements the same recursive spec as edit(i, j), but assumes there is an $m \times n$ array $A[1..m, 1..n]$ ("A" for "Answer") allocated in global memory, with all entries initially set to null. */*

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j$.
3. If $j > n$ then return $m - i$.
4. If $A[i, j]$ is null:
 - A. If $x[i] \neq y[j]$ then set

$$A[i, j] = \min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}.$$

B. Otherwise set

$$A[i, j] = \text{edit}(i + 1, j + 1), 1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1)\}.$$

5. Return $A[i, j]$.

min # single character insertions, deletions, and substitutions to convert x into y .

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive spec for all n by induction on n

1. index parameters to be more cache-friendly

① Recursive specification let $x[1..m]$, $y[1..n]$ be fixed
 $\text{edit}(i, j)$ = the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$.

② Implement spec

edit(i, j)

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j$.
3. If $j > n$ then return $m - i$.
4. If $x[i] \neq y[j]$ then return $\min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}$.
5. Otherwise return $\min\{\text{edit}(i + 1, j + 1), 1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1)\}$.

$A[1..m, 1..n]$ is a 2-dim array in global memory

Caching edit distance

- save answers to each subproblem
- never solve same subproblem twice.

2. augment recursive code w/ caching

cached-edit(i, j)

/ Implements the same recursive spec as edit(i, j), but assumes there is an $m \times n$ array $A[1..m, 1..n]$ ("A" for "Answer") allocated in global memory, with all entries initially set to null.*

**/*

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j$.
3. If $j > n$ then return $m - i$.
4. If $A[i, j]$ is null:

A. If $x[i] \neq y[j]$ then set

$$A[i, j] = \min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}.$$

B. Otherwise set

$$A[i, j] = \text{edit}(i + 1, j + 1).$$

5. Return $A[i, j]$.

min # single character insertions, deletions, and substitutions to convert x into y .

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness

argue algo satisfies recursive spec for all n by induction on n

1. index parameters to be more cache-friendly

① Recursive specification let $x[1..m], y[1..n]$ be fixed

$\text{edit}(i, j)$ = the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$.

② Implement spec

edit(i, j)

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j$.
3. If $j > n$ then return $m - i$.
4. If $x[i] \neq y[j]$ then return $\min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}$.
5. Otherwise return $\text{edit}(i + 1, j + 1)$.

$A[1..m, 1..n]$ is a 2-dim array in global memory

for orig problem, return ~~$x[1]$~~ $\text{edit}(1, 1)$

cached-edit(i, j)

/ Implements the same recursive spec as edit(i, j), but assumes there is an $m \times n$ array $A[1..m, 1..n]$ ("A" for "Answer") allocated in global memory, with all entries initially set to null. */*

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j$.
3. If $j > n$ then return $m - i$.
4. If $A[i, j]$ is null:
 - A. If $x[1] \neq y[1]$ then set

$$A[i, j] = \min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}.$$

- B. Otherwise set

$$A[i, j] = \min\{\text{edit}(i + 1, j + 1), 1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1)\}.$$

5. Return $A[i, j]$.

Running time

mn subproblems

$O(1)$ time per subproblem
(excl. recursive calls)

$O(mn)$ time total

Caching, a.k.a. dynamic programming:

If recursive algo has polynomial # of subproblems, then
caching $\Rightarrow$ polynomial time algo

template for this class:

1. recursive spec
2. recursive implementation
3. mention "caching" or "dynamic programming"
4. analyze run time
5. how to use recursive algo to solve problem
6. proof (by induction)

See notes for full explanation & sample solutions

```
cached-edit(i,j)
/* Implements the same recursive spec as edit(i,j), but assumes there is an  $m \times n$  array
   A[1..m, 1..n] ("A" for "Answer") allocated in global memory, with all entries initially set to
   null. */
1. If  $i > m$  and  $j > n$  then return 0.
2. If  $i > m$  then return  $n - j$ .
3. If  $j > n$  then return  $m - i$ .
4. If A[i,j] is null:
   A. If  $x[i] \neq y[j]$  then set
       $A[i,j] = \min\{1 + \text{edit}(i+1,j), 1 + \text{edit}(i,j+1), 1 + \text{edit}(i+1,j+1)\}$ .
   B. Otherwise set
       $A[i,j] = \min\{\text{edit}(i+1,j+1), 1 + \text{edit}(i+1,j), 1 + \text{edit}(i,j+1)\}$ .
5. Return A[i,j].
```

Running time

mn subproblems

$\times$ $O(1)$ time per subproblem
(excl. recursive calls)

$O(mn)$ time total

template for this class:

1. recursive spec
2. recursive implementation
3. mention "caching" or "dynamic programming"
4. analyze run time
5. how to use recursive algo to solve problem
6. proof (by induction)

We don't want:

- code implementing caching
- iterative code (w/ loops)
- space usage

(which is mechanical)

```
cached-edit(i,j)
/* Implements the same recursive spec as edit(i,j), but assumes there is an m x n array
   A[1..m, 1..n] ("A" for "Answer") allocated in global memory, with all entries initially set to
   null. */
1. If i > m and j > n then return 0.
2. If i > m then return n - j.
3. If j > n then return m - i.
4. If A[i,j] is null:
   A. If x[i] != y[j] then set
      A[i,j] = min{1 + edit(i+1,j), 1 + edit(i,j+1), 1 + edit(i+1,j+1)}.
   B. Otherwise set
      A[i,j] = min{edit(i+1,j+1), 1 + edit(i+1,j), 1 + edit(i,j+1)}.
5. Return A[i,j].
```

Running time

mn subproblems

$\times$ $O(1)$ time per subproblem
(excl. recursive calls)

$O(mn)$ time total

Sample solution.

Problem. Design and analyze an algorithm that, given two strings $x[1..m]$ and $y[1..n]$, computes the edit distance from x to y .

Solution.

1. *Recursive spec / induction hypothesis.* Let $x[1..m]$ and $y[1..n]$ be two strings. We define

$\text{edit}(i, j)$ = the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$.

2. *Recursive implementation.*

$\text{edit}(i, j)$

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j$.
3. If $j > n$ then return $m - i$.
4. If $x[1] \neq y[1]$ then return $\min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}$.
5. Otherwise return $\min\{\text{edit}(i + 1, j + 1), 1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1)\}$.

3. *Solving the original problem.* $\text{edit}(1, 1)$ gives the edit distance between $x[1..m]$ and $y[1..n]$.

4. *Running time with caching / dynamic programming.* We apply dynamic programming to the recursive algorithm above. We have $O(mn)$ subproblems, each of which take $O(1)$ time excluding recursive subcalls. Thus the total running time is bounded by

$$(\# \text{ subproblems}) \left(\begin{array}{c} \text{time per} \\ \text{subproblem} \\ \text{excl. recursive calls} \end{array} \right) \leq O(mn).$$

5. *Proof.* We prove that $\text{edit}(i, j)$ satisfies the recursive spec for all $i \in \{1, \dots, m + 1\}$ and $j \in \{1, \dots, n + 1\}$, by induction on $\min\{m - i, n - j\}$.

It is easy to see that $\text{edit}(i, j)$ returns the length of *some* edit sequence from $x[1..m]$ to

Template for this class:

1. recursive spec

2. recursive implementation

3. mention "caching" or "dynamic programming"

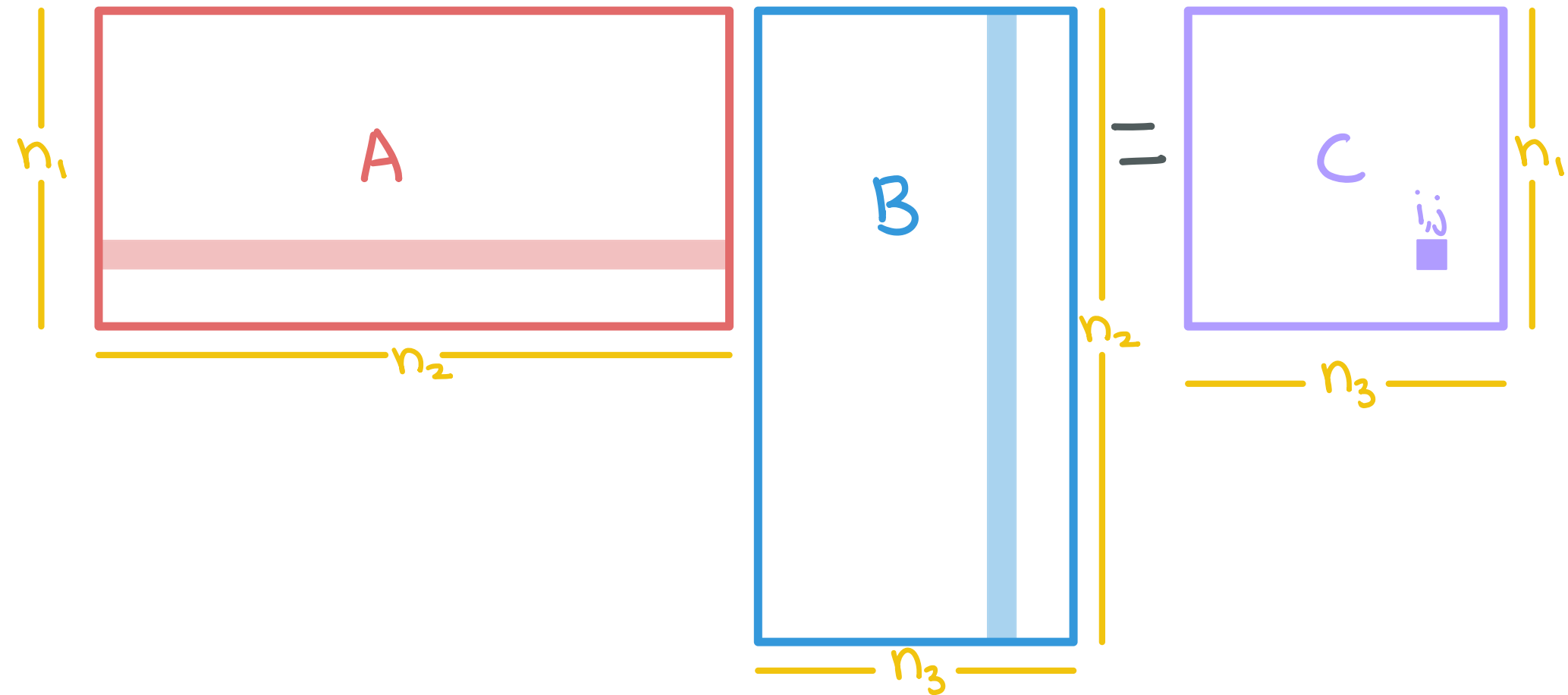
4. analyze run time

5. how to use recursive algo to solve problem

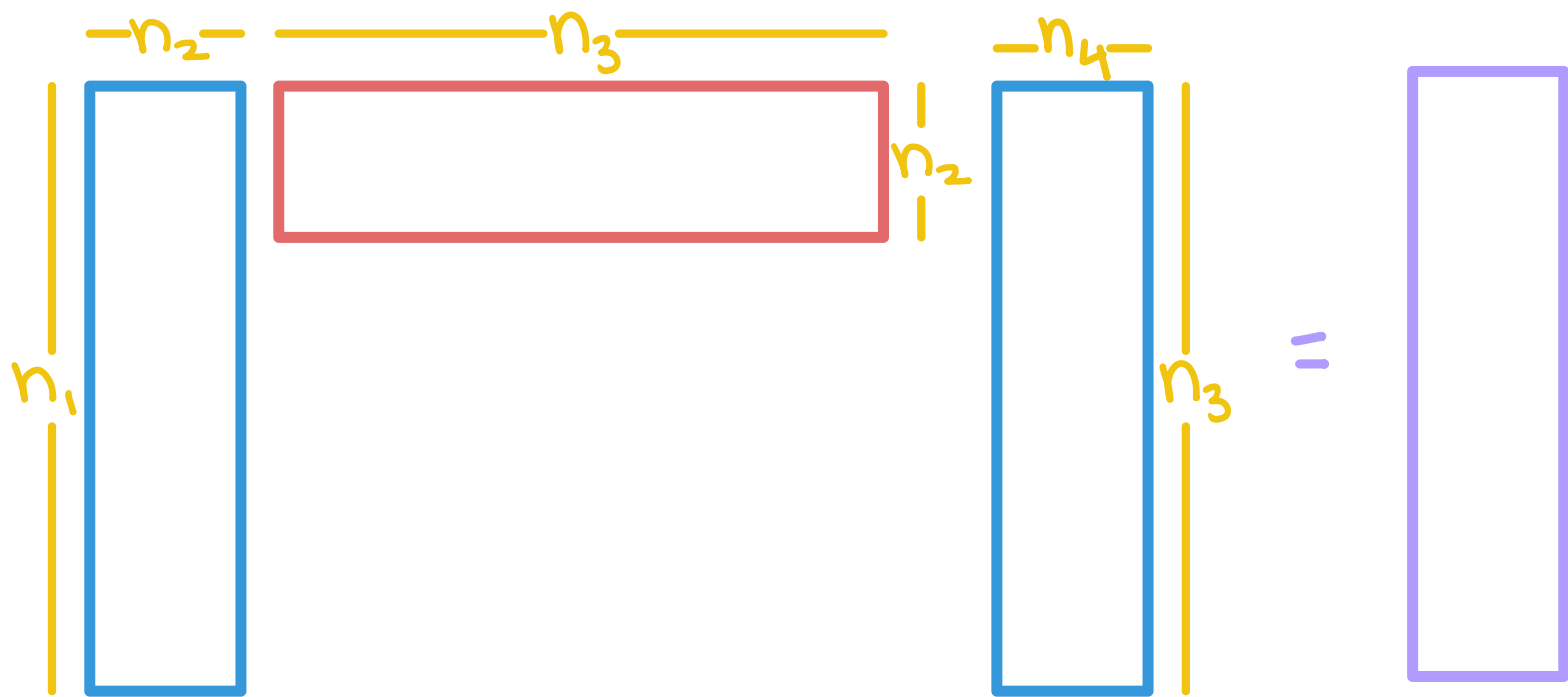
6. proof (by induction)

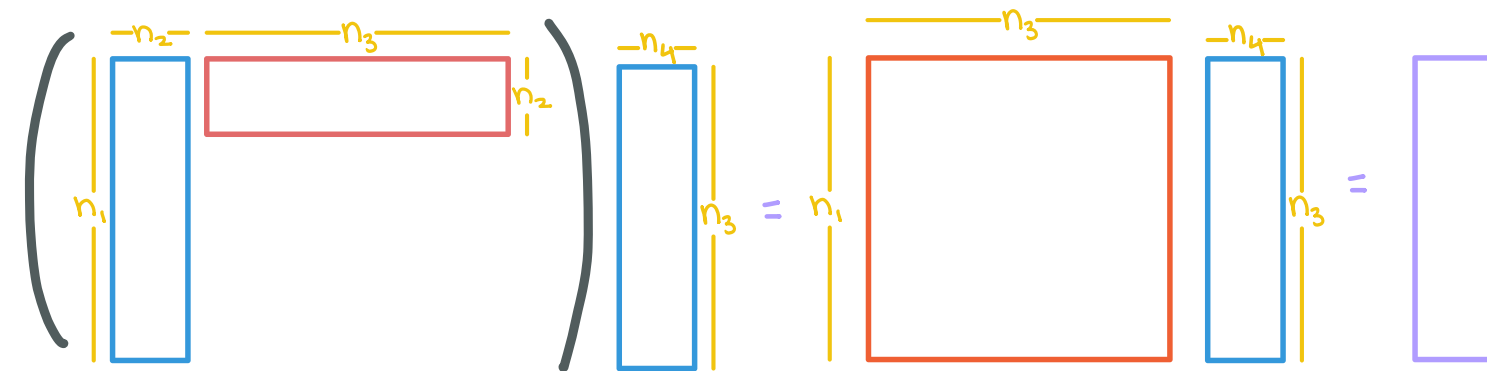
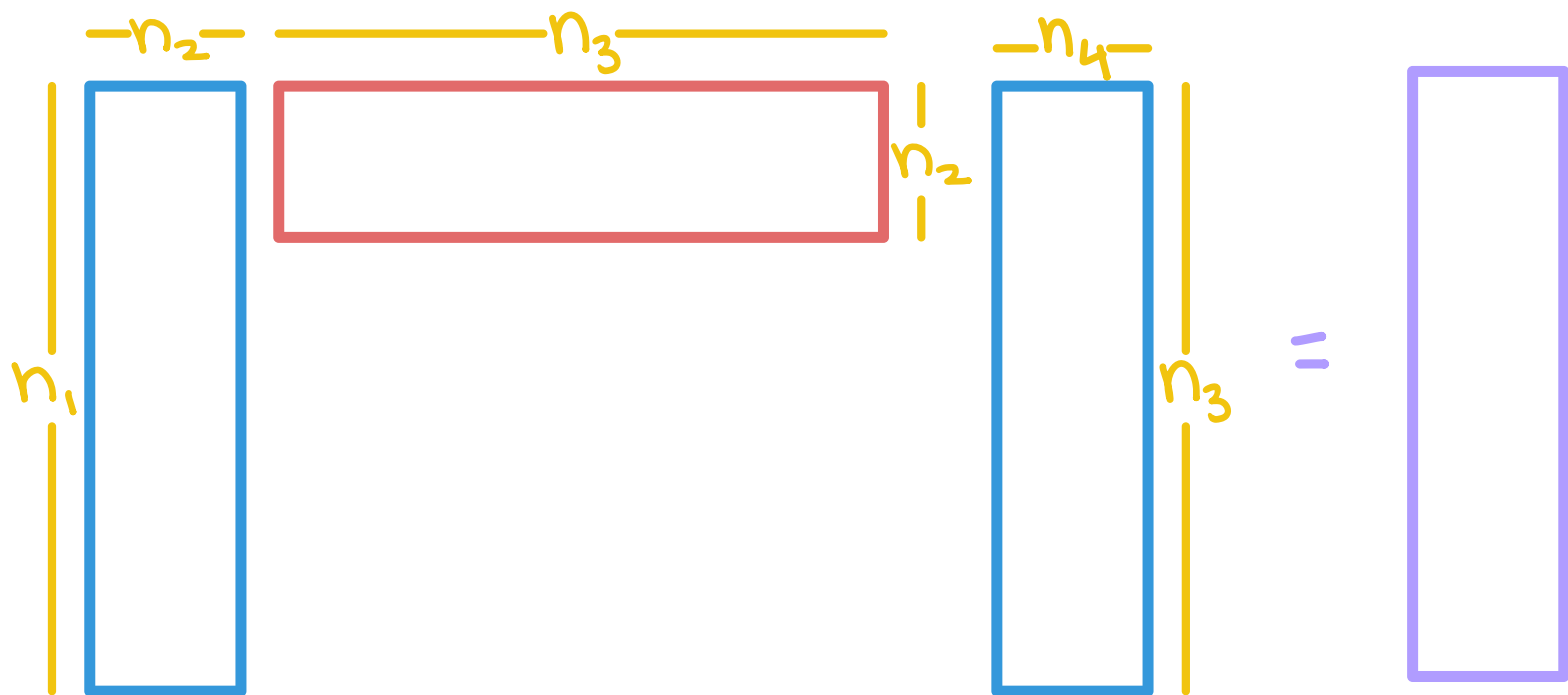
II Matrix chain multiplication

$$C_{ij} = \sum_{k=1}^{n_2} A_{ik} B_{kj}$$

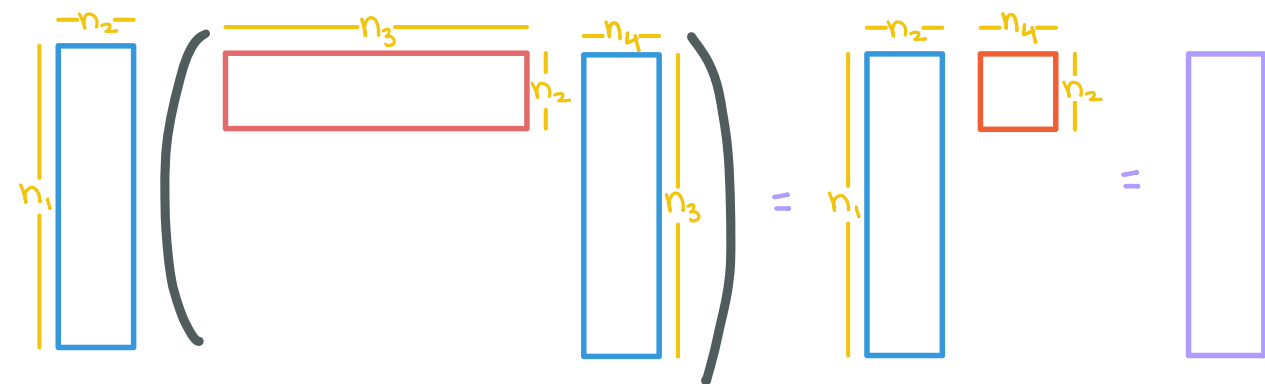


$O(n_1 n_2 n_3)$ time to compute C



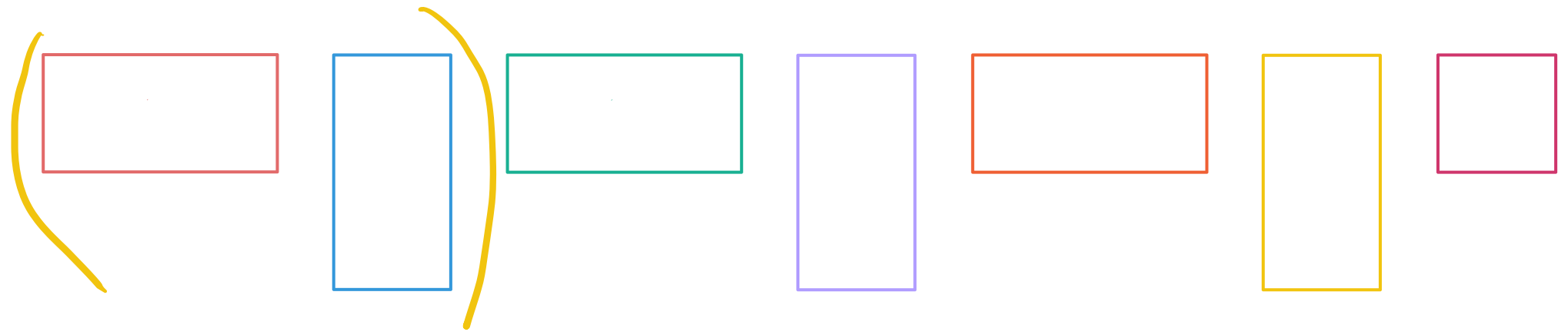


$$O(n_1 n_2 n_3 + n_1 n_3 n_4) \\ = O(n_1 n_3 (n_2 + n_4))$$



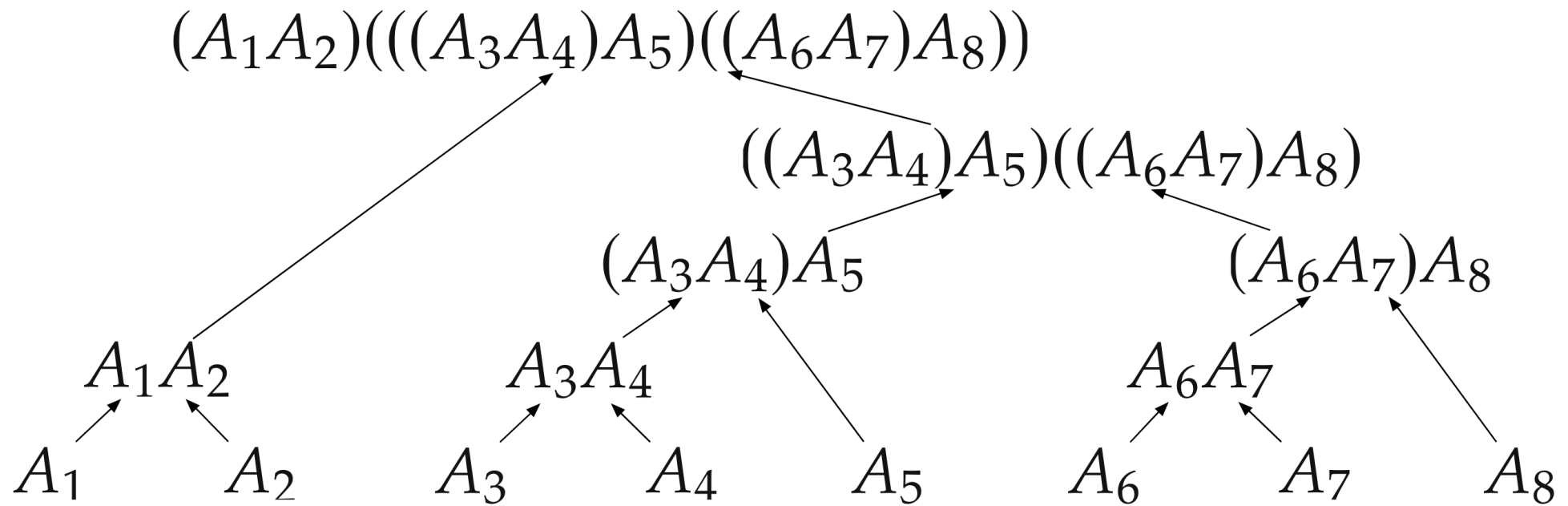
$$O(n_2 n_3 n_4 + n_1 n_2 n_4) \\ = O(n_2 n_4 (n_1 + n_3))$$

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$



Output: min # (ordinary) multiplications needed to compute
 $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

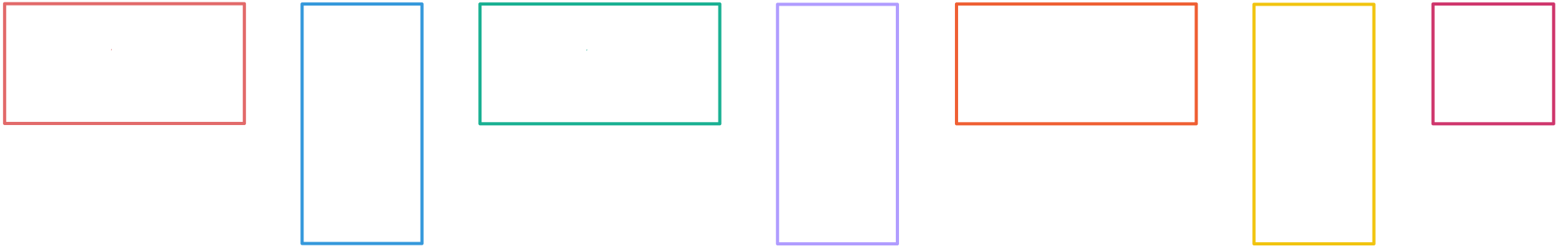


Output: min # (ordinary) multiplications needed to compute
 $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute

$A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$

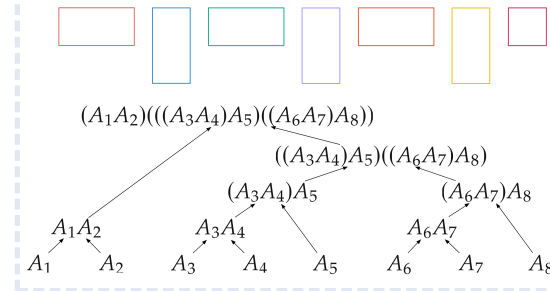


Ideas?

1. recursive spec

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$



template for this class:

1. recursive spec

2. recursive implementation

3. mention "caching" or "dynamic programming"

4. analyze run time

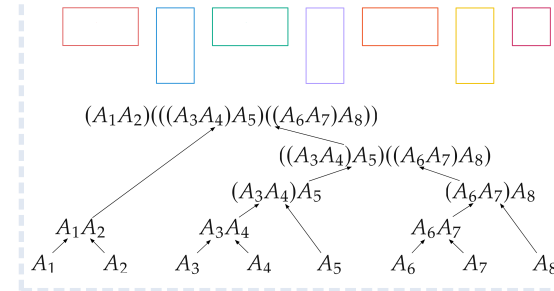
5. how to use recursive algo to solve problem

6. proof (by induction)

1. recursive spec

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$



$\text{Fastest}(i, j)$ = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

template for this class:

1. recursive spec

2. recursive implementation

3. mention "caching" or "dynamic programming"

4. analyze run time

5. how to use recursive algo to solve problem

6. proof (by induction)

1. recursive spec

$\text{Fastest}(i, j)$ = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

2. recursive implementation

$\text{Fastest}(i, j)$:

if $i > j$ ~~base~~ $+\infty$

if $i = j$: return 0

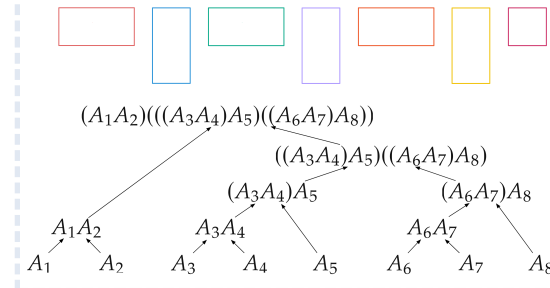
if $j = i+1$: return $n_i n_{i+1} n_{i+2}$

if $i < j$:

$$\min_{h=i+1, \dots, j} \text{Fastest}(i, h-1) + \text{Fastest}(h, j) + n_i n_h n_{j+1}$$

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$



template for this class:

1. recursive spec

2. recursive implementation

3. mention "caching" or "dynamic programming"

4. analyze run time

5. how to use recursive algo to solve problem

6. proof (by induction)

1. recursive spec

$\text{Fastest}(i, j)$ = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

2. recursive implementation

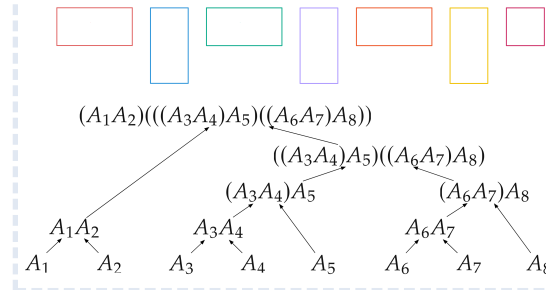
$\text{Fastest}(i, j)$

1. If $i \geq j$ then return 0.
2. Return the minimum, over all $k = i, \dots, j - 1$, of

$$\text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j.$$

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$



template for this class:

1. recursive spec

2. recursive implementation

3. mention "caching" or "dynamic programming"

4. analyze run time

5. how to use recursive algo to solve problem

6. proof (by induction)

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$

1. recursive spec

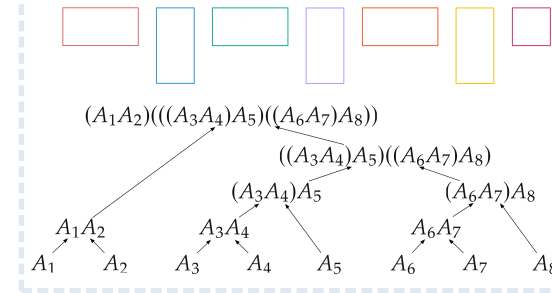
$\text{Fastest}(i, j)$ = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

2. recursive implementation

$\text{Fastest}(i, j)$

1. If $i \geq j$ then return 0.
2. Return the minimum, over all $k = i, \dots, j - 1$, of

$$\text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j.$$



3. how to use recursive algo to solve problem

return $\text{Fastest}(1, k)$

template for this class:

1. recursive spec

2. recursive implementation

3. how to use recursive algo to solve problem

4. mention "caching" or "dynamic programming"

5. analyze run time

6. proof (by induction)

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$

1. recursive spec

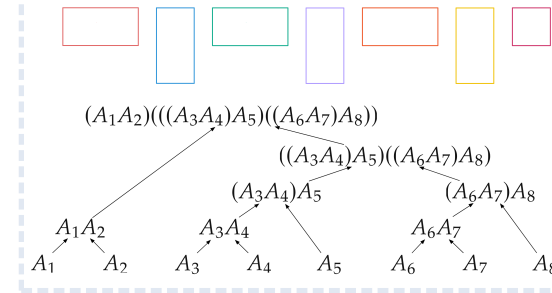
$\text{Fastest}(i, j)$ = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

2. recursive implementation

$\text{Fastest}(i, j)$

1. If $i \geq j$ then return 0.
2. Return the minimum, over all $k = i, \dots, j - 1$, of

$$\text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j.$$



3. how to use recursive algo to solve problem

$\text{Fastest}(1, n)$

template for this class:

1. recursive spec

2. recursive implementation

3. how to use recursive algo to solve problem

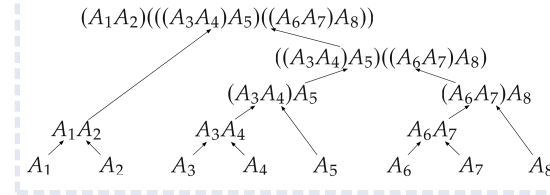
4. mention "caching" or "dynamic programming"

5. analyze run time

6. proof (by induction)

Input: dimensions $n_1, n_2, \dots, n_k, n_{k+1} \in \mathbb{N}$

Output: min # (ordinary) multiplications needed to compute $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$



1. recursive spec

$\text{Fastest}(i, j)$ = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

2. recursive implementation

$\text{Fastest}(i, j)$

1. If $i \geq j$ then return 0.
2. Return the minimum, over all $k = i, \dots, j - 1$, of

$$\text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j.$$

3. how to use recursive algo to solve problem

$\text{Fastest}(1, n)$

4/5: Running time w/ dyn prog.

k^2 subproblems

$O(k)$ time per subproblem

$O(k^3)$ time total

template for this class:

1. recursive spec
2. recursive implementation
3. how to use recursive algo to solve problem
4. mention "caching" or "dynamic programming"
5. analyze run time
6. proof (by induction)

Exercise 5.1. Below is a series of optimization problems that takes as input an array $A[1..n]$ of integers, and asks for optimal subsequences of A satisfying certain properties.⁹ Design and analyze an algorithm for each of these problems, addressing items 1–5 from section 5.2.¹⁰

1. A sequence of numbers $x_1, \dots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \dots, k-1$. Compute the length of the longest increasing subsequence of A .

LIS(i): return length of LIS of $A[i..n]$ starting
w/ $A[i]$

LIS(i):

if $i=n$: return 1

if $i>n$: return 0

for $j=i..n$:

return LIS(j)+1 if $A[i] < A[j]$

return $\begin{cases} \max_{j=i+1, \dots, n} \{ \text{LIS}(j)+1 & \text{if } A[j] > A[i] \\ 1 \end{cases}$

time

subpro
time

$$\begin{array}{r} n \\ \times n \\ \hline n^2 \end{array}$$

Exercise 5.2. Let $A[1..m]$ and $B[1..n]$ be two arrays. Each of the following problems asks for some aspect (e.g., the length) of a common subsequence of $A[1..n]$ and $B[1..n]$ satisfying or optimizing certain properties. (A common subsequence of A and B is a sequence that is a subsequence of both A and B .) Design and analyze an algorithm for each of these problems, addressing items 1–5 from section 5.2.¹⁰

1. Compute the length of the longest common subsequence of A and B .
2. Compute the length of the shortest common supersequence of A and B .
3. Compute the length of the longest common subsequence of A and B that is a palindrome.
4. Suppose A and B consists of integers. Compute the length of the longest common increasing subsequence of A and B .

Exercise 5.9. Suppose you are given a set of n points $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$ in the plane.

1. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in S is comparable. (Cf. exercise [1.8](#) on page [31](#).)
2. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in S is incomparable.

Chapter 5

Sequential optimization with caching

5.1 Edit distance: An algorithmic metric for strings

We are constantly discussing bit strings in this course, and in programming at large, we are constantly parsing, manipulating, and outputting strings. (Here, besides bit strings, we can speak more generally of strings that are sequences of characters from some fixed alphabet $\mathcal{A}$, like the ASCII characters). Naturally, then, many applications required a method to compare two strings, and some way to quantify the extent of their similarity. That is, we seek some notion of “distance” between strings.

One possible notion is *Hamming distance*¹, which counts the number of differing characters in two strings of the same length. That is, for a fixed alphabet $\mathcal{A}$, and two strings $x, y \in \mathcal{A}^k$ of the same length, the Hamming distance is defined as the number of differing bits:

$$\text{Hamming}(x, y) = |\{i \in [k] : x_i \neq y_i\}|.$$

We note that in some applications, the Hamming distance is normalized to be in $[0, 1]$, by dividing the quantity above by the length k . The Hamming distance is symmetric² and two strings have Hamming distance 0 iff they are equal. One can also verify that the Hamming distance satisfies the triangle inequality:

$$\text{Hamming}(x, y) \leq \text{Hamming}(x, z) + \text{Hamming}(z, y)$$

for any three strings $x, y, z \in \{0, 1\}^k$. (Why?) Thus Hamming distance satisfies the three technical requirements of a *metric*, just like in Euclidean geometry.

¹Richard Hamming (1915 – 1998) was an American mathematician and computer scientist who, among many other contributions, invented error correcting codes, and was an early pioneer of computing at Bell Labs. His well-known talk “You and Your Research” ([pdf](#), [video](#)) is highly recommended.

²That is, $\text{Hamming}(x, y) = \text{Hamming}(y, x)$ for all x and y .

The image shows two strings, X and Y, written in purple and red respectively. String X is "ABCDEFGG" and string Y is "ABCEFGD". Above the 'E' in Y, there are four 'x' marks, indicating that the characters from 'E' to 'D' in Y do not match the corresponding characters in X. To the right of the strings, it says "=> Hamming dist. 4".

Figure 5.1: An example of Hamming distance.

The Hamming distance is natural and suitable for many settings (e.g., in coding theory). But it has some weaknesses as well. First, it is only defined when two strings have the same length. Sure, given two strings of different length, one can truncate the longer one to match the shorter one, or pad the shorter one to match the longer one, but it is not clear that the Hamming distance is as meaningful then. (Also, it would no longer be a metric — why?)

Besides the issue of length, the Hamming distance does not match our intuitive notion of similarity for some basic cases. For example, suppose we had a string x . Suppose we remove the first character from the beginning, and appended this character to the end, to form a new string y . On one hand, we made a very small change to take x into y . On the other hand, the Hamming distance and x and y will be tremendous, because all the characters shift by an index. Let us give two real life examples where something similar frequently occurs.

When we write software, we frequently insert or delete a line in the middle of a program. The rest of the program is the same, and overall we think of this as a relatively small change. But in terms of Hamming distance, the misalignment that we introduce gives the impression that the entire program has turned over. This is an important consideration in version control systems (such as `git`), which tracks the changes to source code over time, and stores compressed logs that only describe the δ 's between versions of the same code. We don't want to confuse a minute change for a full rewriting.

A second example is from biology. We now know that we are described by a genetic code. For the sake of discussion, these are long strings from an alphabet of four letters, $\{A, T, G, C\}$, each standing for a different molecule. These genetic codes frequently undergo mutations, where one letter is swapped for another, or letters are dropped or inserted along the code. This too introduces some misalignment, but functionally the altered mutated code is not so different. The Hamming distance would suggest an entirely different species, but biologically this is just not so.

A more robust measure than Hamming distance, that better models the number of changes made to a string, is called *edit distance*. The edit distance between two strings $s, t \in \mathcal{A}^*$ is the minimum number of *insertions*, *deletions*, and *substitutions*

required to convert one string to another.³ These operations are defined as follows.

1. An insertion inserts a single letter at some point in the string.
2. A deletion deletes a single letter from the string.
3. A substitution replaces one letter in the string with another.

Note that edit distance is now completely well-defined. The edit distance between two strings x and y is bounded below by 0, and bounded above by the length of x plus the length of y (by deleting every character in x and then inserting every character from y). It is also computable, by brute force — e.g., by enumerating all sequences of edits with length at most the sum of lengths of x and y .

$x =$ ABCDEF^{del}G
 $y =$ ABC^{ins}DEFGD
 $\Rightarrow$ Edit. distance = 2

Figure 5.2: An example of edit distance

Let $\text{edit}(x, y)$ denote the edit distance between x and y . $\text{edit}(x, y)$ is symmetric, in the sense that the edit distance from x to y is equal to the edit distance from y to x . (This is because any sequence of edits converting x to y can be reversed to give a sequence of edits converting y to x). It also satisfies the triangle inequality. Indeed, let x, y, z , and suppose we want to show that

$$\text{edit}(x, y) \leq \text{edit}(x, z) + \text{edit}(z, y).$$

To this end, observe that any sequence of edits from x to z , followed by a sequence of edits from z to y , gives a sequence of edits from x to y . Thus $\text{edit}(x, z) + \text{edit}(z, y)$, being the length of some edit sequence from x to y , gives an upper bound on the minimum length sequence of edits from x to y .

Thus $\text{edit}(x, y)$ is also a metric between strings, like Hamming distance. Unlike Hamming distance, it gracefully handles strings of different, arbitrary lengths. For many applications, such as code versioning and genetic comparisons, it is a much more useful measure of distance between strings.

Now that the edit distance is well-defined, we seek an efficient algorithmic implementation. Edit distance is frequently applied to large data — genetic codes and

³Here $\mathcal{A}^*$ denotes the family of all finite strings with characters from $\mathcal{A}$. The $*$ indicates that the string may be of any length.

source code repositories can both be very large — so having efficient algorithms is important.

Before, we lazily pointed out that edit distance can always be computed (very slowly) by enumeration. Let us instead try to develop a recursive algorithm, starting with a recursive spec:

edit($x[1..m], y[1..n]$) : Given two strings x and y , returns the minimum number of edit operations needed to convert x into y .

To implement **edit**, we start from the easy base cases. If both strings are empty, then the edit distance is 0. Let's write that down.

$$\text{edit}(\emptyset, \emptyset) = 0,$$

where $\emptyset$ denotes the empty string. If only one of the strings is empty, then the edit distance is the length of the nonempty string:

$$\text{edit}(x[1..k], \emptyset) = k, \quad \text{edit}(\emptyset, y[1..\ell]) = \ell.$$

In the interesting case, we have two strings x and y that are nonempty. Suppose first that the first characters are unequal: $x[1] \neq y[1]$. Any edit sequence from x to y , simply to make the first character match, must either

- (a) insert $y[1]$ before $x[1]$,
- (b) delete $x[1]$, or
- (c) substitute $y[1]$ for $x[1]$.

If not, then the first character would end up being $x[1]$, a contradiction.

Out of the three choices above, it is not obvious what is the best option. One might think that we should always substitute, since it seems to address two letters ($x[1]$ and $y[1]$) at the cost of a single edit. But if (for example) $x[1] = y[2]$ or $y[1] = x[2]$, we might be making a mistake, and disrupting a “free match” later on. In general, the downstream effects of the current decision are not clear. Instead of analyzing further, let us be lazy, and try all three options. Thus, if $x[1] \neq y[1]$, we have

$$\text{edit}(x[1..k], y[1..\ell]) = 1 + \min \begin{cases} \text{edit}(x[2..k], y[1..\ell]), \\ \text{edit}(x[1..k], y[2..\ell]), \\ \text{edit}(x[2..k], y[2..\ell]). \end{cases}$$

The first line in the “min” models deleting $x[1]$, the second models inserting $y[1]$, and the third models substituting $y[1]$ for $x[1]$. We don't know which one of the choices is right, but we have deduced that it must be one of them.

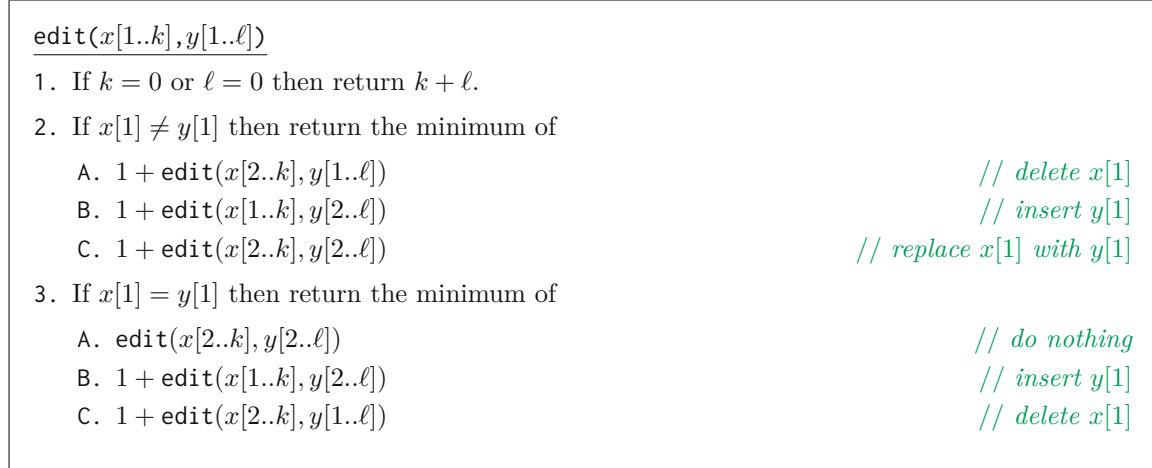


Figure 5.3: A slow, recursive algorithm for computing the edit distance.

The other case is where $x[1] = y[1]$. In this case, we can simply leave $x[1]$ and $y[1]$ alone, and focus on editing $x[2..k]$ and $y[2..l]$. Thus, if $x[1] = y[1]$, then

$$\text{edit}(x[1..k], y[1..l]) \leq \text{edit}(x[2..k], y[2..l]).$$

Note that we wrote an inequality, rather than an equality, because technically speaking, we have only upper bounded $\text{edit}(x[1..k], y[1..l])$. It seems obvious that one *should* match $x[1]$ to $y[1]$, and indeed one can prove this with some care. Being lazy, we instead do the following. If (for whatever reason) we don't want to match $x[1]$ to $y[1]$, then we must either insert $y[1]$ or delete $x[1]$. (Clearly there's no point in substituting when $x[1] = y[1]$ already.) Thus, if $x[1] = y[1]$, we have

$$\text{edit}(x[1..k], y[1..l]) = \min \begin{cases} \text{edit}(x[2..k], y[2..l]), \\ 1 + \text{edit}(x[2..k], y[1..l]), \\ 1 + \text{edit}(x[1..k], y[2..l]), \end{cases}$$

This last formula finishes covering all cases, and so we have a recursive definition for **edit-distance**. We gather the preceding discussion in a single pseudocode above.

One can show by induction on $k + \ell$ that the pseudocode is correct. This is not surprising since the code is essentially a brute force recursive algorithm, searching all possible edit sequences from x to y . For the same reason, the code will be incredibly inefficient: the code takes $O(3^{m+n})$ time for strings of length m and n .

Can we make this algorithm run in polynomial time? More broadly: can we retain the correctness of recursive enumeration, without paying for the running time?

Caching. We can speed up the algorithm by applying an old trick: caching. To make the code more cache-friendly, we index the parameters more compactly as follows. Fix the input strings $x[1..m]$ and $y[1..n]$ (globally). For $i, j \in \mathbb{N}$, we define

$\text{edit}(i, j)$ = the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$.

Then $\text{edit}(1, 1)$ gives the edit distance between $x[1..m]$ and $y[1..n]$. To implement this version of a recursive spec, we have:

$\text{edit}(i, j)$

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j + 1$.
3. If $j > n$ then return $m - i + 1$.
4. If $x[i] \neq y[j]$ then return $\min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}$.
5. Otherwise return $\min\{\text{edit}(i + 1, j + 1), 1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1)\}$.

Now here's the big idea: we cache (i.e., save) the answers to all our recursive calls so we never solve the same problem twice. Let us allocate a table $A[1..m, 1..n]$ in global memory, with all entries initially set to `null`. (“A” for “Answer”). In every recursive call, we first check A to see if we have already solved the subproblem. If so, then we can return the saved answer. If not, then we execute the pseudocode above, and store the answer before returning it. (We do not need to save the base cases in A , since they take constant time anyway.) See the code for `cached-edit` in fig. 5.4 on the following page.

Consider now the running time of `cached-edit`. Although the algorithm is recursive, the introduction of caching requires a different style of analysis from recursion trees.

We organize our runtime analysis by *subproblem*. Observe that the parameters i and j range from 1 to $m + 1$ and from 1 to $n + 1$, respectively. Thus, the *number of subproblems* — i.e., the number of choices of i and j — is

$$(\# \text{ subproblems}) \leq O(mn).$$

Each subproblem takes $O(1)$ time excluding recursive subcalls. That is,

$$\left(\begin{array}{c} \text{time per subproblem} \\ \text{excl. recursive calls} \end{array} \right) = O(1).$$

```

cached-edit(i,j)
/* Implements the same recursive spec as edit(i,j), but assumes there is an m × n array
   A[1..m, 1..n] (“A” for “Answer”) allocated in global memory, with all entries initially set to
   null. */
1. If i > m and j > n then return 0.
2. If i > m then return n − j + 1.
3. If j > n then return m − i + 1.
4. If A[i,j] is null:
   A. If x[i] ≠ y[j] then set
      A[i,j] = min{1 + edit(i + 1, j), 1 + edit(i, j + 1), 1 + edit(i + 1, j + 1)}.
   B. Otherwise set
      A[i,j] = min{edit(i + 1, j + 1), 1 + edit(i + 1, j), 1 + edit(i, j + 1)}.
5. Return A[i,j].

```

Figure 5.4: Computing edit distance recursively and with caching.

All put together, we have

$$(\text{total runtime}) \leq (\# \text{ subproblems}) \left(\begin{array}{c} \text{time per subproblem} \\ \text{excl. recursive calls} \end{array} \right) \leq O(mn).$$

With a little extra memory, we have reduced our running time from exponential to quadratic.

5.2 Caching, a.k.a. dynamic programming: a meta-algorithm

The difference in running time comes from the caching trick. It is a general technique that can be extended to all sorts of recursions, and is more commonly known as *dynamic programming*. The recipe is simple. Recursion, plus a little bit of memory to avoid redundancy, makes for more efficient algorithms.

The steps taken to convert the recursive algorithm `edit(i,j)` into the more efficient dynamic programming algorithm `cached-edit(i,j)` were mechanical and problem independent. When applying caching to a generic problem, rather than going through these routine details again and again, the following points suffice to understand what is going on. (In this class, they are all required.) The following steps are also useful

in helping you structure your thoughts and apply dynamic programming effectively, and with some practice, effortlessly.

1. A concise specification of the recursive function you are trying to implement, such as our one sentence description of `edit(i,j)`. It should be clear what is the input, and what is the desired output. The parameters may be indices to objects defined globally; which objects these refer to should be made clear.

The specification provides the induction hypothesis that justifies the correctness of your algorithm. It is the most important step.

2. Some kind of pseudocode or formula that implements the specification, such as the five line code for `edit(i,k)` on page 92.
3. Some explicit *mention* of the use of caching or dynamic programming to avoid recomputing subproblems.
4. An analysis of the total running time, which often breaks down as the number of subproblems times the amount of time per subproblem, excluding recursive calls.
5. A brief description of how to use the recursive function to produce the answer we are actually looking for. For edit distance, `edit(1,1)` returns the edit distance of $x[1..m]$ and $y[1..n]$.

(There will be problems where there is a bigger difference between the recursive spec and what the problem is asking for, which makes this step nontrivial.)

6. When needed, some justification or proof for the correctness of the algorithm.
(In general the overall proof should follow by induction, using the recursive spec for the induction hypothesis. This high-level setup is boilerplate. Sometimes some justification is needed to explain the choices the algorithm is making, such as how we showed the algorithm must make one of the three edit operations when $x[i] \neq y[j]$.)

A sample solution for edit distance, fulfilling the steps above, is given on page 92.

The critical thinking largely occurs in the first step, where we specify the function we want to implement. With a clear specification, the implementation in the second step can be straightforward. This is because recursive algorithms leverage induction, and a clearer specification gives a more useful induction hypothesis. Conversely, if it is difficult to implement the code in step 2, this may indicate that the specification

is not quite right. Sometimes, introducing another parameter, or strengthening the guarantees of the recursive spec, can make it much easier to implement.

Once steps 1 and 2 are done correctly and clearly, then the remaining steps can be straightforward. In particular, the correctness of the algorithm is often self-evident and hardly requires a proof.⁴ If one of the later steps feels awkward or difficult, then this might be best addressed by clarifying the specification or the pseudocode.

Steps that are not required include:

1. Literally implementing the caching (which is mechanical and clutters the code). That is, we are not asking for code that looks like “`cached-edit`”.
2. Providing code that iterates through the subproblems manually. (Recursion handles this automatically.)
3. The space usage (which is automatically bounded above by the running time, and rarely more interesting than that).

At some level we even discourage the above steps as they distract from the perspectives of top-down design and problem solving by induction.

Our first example of dynamic programming, `edit-distance`, was relatively simple. But in upcoming discussions we will see how versatile this tool is, and the large design space it encumbers.

⁴On tests or other situations where we favor expedience, we may not ask for a proof, and instead we will interpret your recursive spec as the induction hypothesis of a proof by induction and fill in the rest of the details ourselves.

Sample solution.

Problem. Design and analyze an algorithm that, given two strings $x[1..m]$ and $y[1..n]$, computes the edit distance from x to y .

Solution.

1. *Recursive spec / induction hypothesis.* Let $x[1..m]$ and $y[1..n]$ be two strings. We define

$\text{edit}(i, j)$ = the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$.

2. *Recursive implementation.*

$\text{edit}(i, j)$

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j + 1$.
3. If $j > n$ then return $m - i + 1$.
4. If $x[i] \neq y[j]$ then return
 $\min\{1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1), 1 + \text{edit}(i + 1, j + 1)\}$.
5. Otherwise return $\min\{\text{edit}(i + 1, j + 1), 1 + \text{edit}(i + 1, j), 1 + \text{edit}(i, j + 1)\}$.

3. *Solving the original problem.* $\text{edit}(1, 1)$ gives the edit distance between $x[1..m]$ and $y[1..n]$.

4. *Running time with caching / dynamic programming.* We apply dynamic programming to the recursive algorithm above. We have $O(mn)$ subproblems, each of which take $O(1)$ time excluding recursive subcalls. Thus the total running time is bounded by

$$(\# \text{ subproblems}) \left(\begin{array}{c} \text{time per} \\ \text{subproblem} \\ \text{excl. recursive calls} \end{array} \right) \leq O(mn).$$

5. *Proof.* We prove that $\text{edit}(i, j)$ satisfies the recursive spec for all $i \in \{1, \dots, m + 1\}$ and $j \in \{1, \dots, n + 1\}$, by induction on $\min\{m - i, n - j\}$.

It is easy to see that $\text{edit}(i, j)$ returns the length of *some* edit sequence from $x[1..m]$ to $y[1..n]$ (which can be shown by induction); we focus on showing it gives the length of the minimum edit sequence.

In the base of -1 , either $m < i$ or $n < j$ and so we have one of the base cases, the correctness of which is clear.

Otherwise $1 \leq i \leq m$ and $1 \leq j \leq n$. Fix an optimal edit sequence of $x[i..m]$ and $y[j..n]$, which we call OPT. If $x[i] \neq y[j]$, then to edit $x[i..m]$ to $y[j..n]$, then OPT must either delete $x[i]$, insert $y[j]$, or substitute $y[j]$ for $x[i]$.

- If OPT deletes $x[i]$, then the remaining edits if OPT transform $x[i+1, \dots, m]$ to $y[j..n]$ with the minimum number of steps. The optimum number of edits to do so is given $\text{edit}(i+1, j)$ by induction, so OPT has length $1 + \text{edit}(i+1, j)$.
- If OPT inserts $y[j]$, then the remaining edits transform $x[i..m]$ to $y[j+1..n]$ with the minimum number of steps. The minimum number of edits to do so is given by $\text{edit}(i, j+1)$ by induction. So OPT has $1 + \text{edit}(i, j+1)$.
- If OPT substitutes $y[j]$ for $x[i]$, then the remaining edits transform $x[i+1..m]$ to $y[j+1..n]$, and the minimum number of edits to do so is given by $\text{edit}(i+1, j+1)$. Thus OPT has length $1 + \text{edit}(i+1, j+1)$.

If $x[i] = y[j]$, then an optimal edit sequence can either delete $x[i]$, insert $y[j]$, or do neither and proceed to edit $x[i+1..m]$ to $y[j+1..n]$.

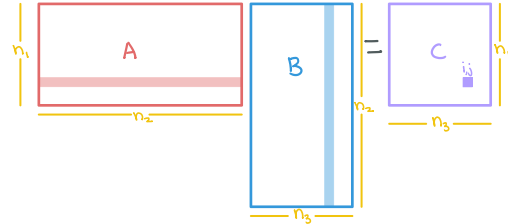
- If OPT deletes $x[i]$, then the remaining edits if OPT transform $x[i+1, \dots, m]$ to $y[j..n]$ with the minimum number of steps. The optimum number of edits to do so is given $\text{edit}(i+1, j)$ by induction, so OPT has length $1 + \text{edit}(i+1, j)$ edits.
- If OPT inserts $y[j]$, then the remaining edits transform $x[i..m]$ to $y[j+1..n]$ with the minimum number of steps. The minimum number of edits to do so is given by $\text{edit}(i, j+1)$ by induction. So OPT has $1 + \text{edit}(i, j+1)$ edits.
- Otherwise OPT's edits transform $x[i+1..m]$ to $y[j+1..n]$ with the minimum number of steps. The minimum number of edits to do so is given by $\text{edit}(i+1, j+1)$ by induction. So OPT has length $\text{edit}(i+1, j+1)$.

For each case ($x[i] \neq x[j]$ or $x[i] \neq y[j]$), the recursive spec enumerates the three possible lengths for OPT and outputs the minimum of the three, which is no greater than, hence equal to, the length of OPT.

5.3 Sequential matrix multiplication

Recall that given two matrices $A \in \mathbb{R}^{n_1 \times n_2}$ and $B \in \mathbb{R}^{n_2 \times n_3}$, the *product* $C = AB$ is defined coordinatewise by

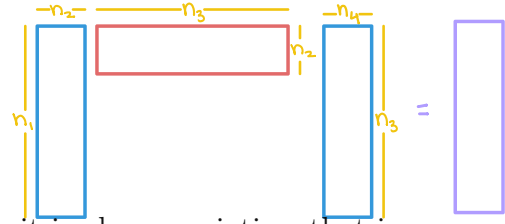
$$C_{i,j} = \sum_{k=1}^{n_2} A_{i,k} B_{k,j}.$$



Note that this is a sum of n_2 terms. (I think of this as “zipping up” along the inner dimension.) It takes $O(n_1 n_2 n_3)$ to compute all $n_1 n_3$ coordinates of AB .

Let $A_1 \in \mathbb{R}^{n_1 \times n_2}$, $A_2 \in \mathbb{R}^{n_2 \times n_3}$ and $A_3 \in \mathbb{R}^{n_3 \times n_4}$ be three matrices, and consider the matrix product

$$A_1 A_2 A_3.$$

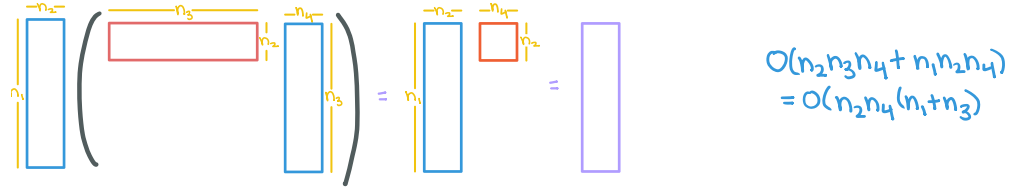


While matrix multiplication is defined pairwise, it is also associative; that is,

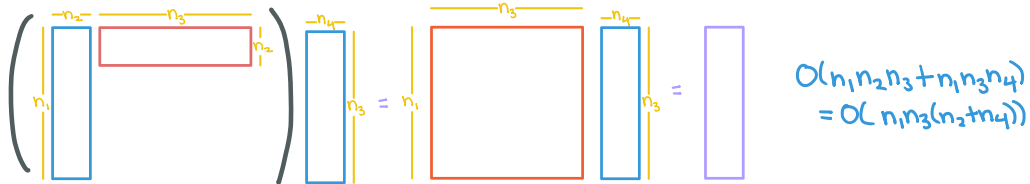
$$A_1(A_2 A_3) = (A_1 A_2) A_3.$$

⁵Consequently, we can write $A_1 A_2 A_3$ unambiguously.

Computationally, however, there is a distinction between computing $A_1 A_2 A_3$ as the product $A_1(A_2 A_3)$ or as the product $(A_1 A_2) A_3$.



Consider first computing the product as $A_1(A_2 A_3)$. Computing $(A_2 A_3)$ takes $O(n_2 n_3 n_4)$ time and produces a $n_2 \times n_4$ -dimension matrix. Multiplying $A_1(A_2 A_3)$ then takes $O(n_1 n_2 n_4)$ time. Overall we spend $O(n_2 n_3 n_4 + n_1 n_2 n_4) = O((n_1 + n_3) n_2 n_4)$ time.



Now consider $(A_1 A_2) A_3$. Computing $(A_1 A_2)$ takes $O(n_1 n_2 n_3)$ time and produces a $n_1 \times n_3$ -dimension matrix. Multiplying $(A_1 A_2) A_3$ takes $O(n_1 n_3 n_4)$ time. Together we spend $O(n_1 n_2 n_3 + n_1 n_3 n_4) = O(n_1 n_3 (n_2 + n_4))$ total time.

⁵We can verify this directly: for $i_1 \in [n_1]$ and $i_4 \in [n_4]$, we have

$$(A_1(A_2 A_3))_{i_1, i_4} = \sum_{i_2=1}^{n_2} A_{1, i_1, i_2} (A_2 A_3)_{i_2, i_4} = \sum_{i_2=1}^{n_2} A_{1, i_1, i_2} \sum_{i_3=1}^{n_3} A_{i_2, i_3} A_{i_3, i_4} = \sum_{i_2=1}^{n_2} \sum_{i_3=1}^{n_3} A_{1, i_1, i_2} A_{i_2, i_3} A_{i_3, i_4};$$

similarly one can show that $((A_1 A_2) A_3)_{i_1, i_4} = \sum_{i_2=1}^{n_2} \sum_{i_3=1}^{n_3} A_{1, i_1, i_2} A_{i_2, i_3} A_{i_3, i_4}$ by expanding out the appropriate sums.

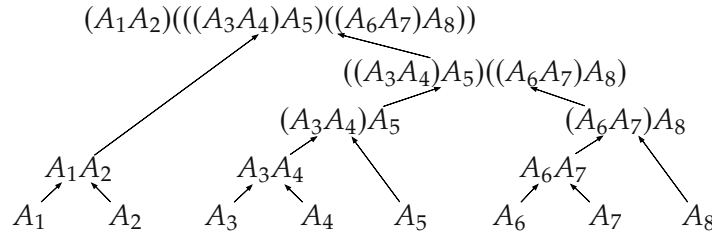
So there is a difference in running time. For example, if $n_1 = n_3 = m$ and $n_2 = n_4 = 1$ for m very large, the first approach, $A_1(A_2A_3)$, would take $O(m)$ time, while the second approach, $(A_1A_2)A_3$, would take $O(m^2)$ time. The first approach is much faster than the second.

More generally, suppose we had a sequence of k matrices $A_1, \dots, A_k$ where each A_i is an $n_i \times n_{i+1}$ -dimensional matrix. We want to compute the product $A_1A_2 \cdots A_k$ and we can only multiply two matrices at a time. Consider the optimization problem of identifying the sequence of pairwise matrix multiplications to produce the product $A_1A_2 \cdots A_k$ with as few numerical multiplications as possible. (Once we find the best order of operations, *then* we can follow them sequentially to actually compute the matrix product.)

One way to describe a multiplication strategy is by grouping the pairwise products with parentheses. For example, the following lays out an approach for 8 matrices.

$$(A_1A_2)((A_3A_4)A_5)((A_6A_7)A_8))$$

We can also envision the multiplications as a tree:



With these preliminary observations we pause to let the reader to try to solve the problem for themselves.

Let us first mention two greedy rules that seem natural and are commonly suggested, but one can devise counterexamples that show they are not optimal.

1. We can take a greedy bottom up approach, where in the first matrices, we take the product A_iA_{i+1} minimizing $n_in_{i+1}n_{i+2}$; replace A_i, A_{i+1} with their product (A_iA_{i+1}) , and continue in the same fashion.
2. We can take a greedy top down approach, where in the first iteration, we identify the split $(A_1 \cdots A_i)(A_{i+1} \cdots A_k)$ minimizing $n_1n_in_{k+1}$, and then recursively compute the two halves $(A_1 \cdots A_i)$ and $(A_{i+1} \cdots A_k)$ in the same fashion.

Let us instead take a more conservative, recursive approach. The first step, as usual, is to develop a recursive specification that also gives an algorithmic induction hypothesis. This is essentially the same task as identifying the right subproblems to break down the original problem. Let $A_1, \dots, A_n$ be n matrices.

We define:

Fastest(i, j) = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

At the moment, we are only interested in the minimum number of operations over all multiplication strategies, and are not worried about actually outputting a multiplication strategy. (We will show how to extract the strategy at the end).

To implement the recursive spec, we have:

Fastest(i, j)

1. If $i \geq j$ then return 0.
2. Return the minimum, over all $k = i, \dots, j - 1$, of

$$\text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j.$$

To justify the pseudocode, fix an instance **Fastest**(i, j). Inductively (on the number of matrices) we assume that the subcalls to **Fastest** with fewer matrices are correct. In this case, for each k ,

$$\text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j$$

gives the minimum number of multiplications among all those that have the split $(A_i \dots A_k)(A_{k+1} \dots A_j)$ at the top of the tree. Which k gives the best initial split? Who knows?! So instead we exhaustively test all choices of k and return the best one.

Of course **Fastest**($1, k$) would be exponentially slow as is. But if we apply dynamic programming then the running time is greatly reduced, to

$$(\# \text{ subproblems}) \times (\text{time per subproblem}) = k^2 \times O(k) = O(k^3).$$

Note that **Fastest**($1, k$) only reports the minimum number of numerical multiplications required to produce the product $A_1 \dots A_k$. Of course we also want the actually sequence of operations that produces it. Here we can make a second table that records, for each **Fastest**(i, j), the choice of index ℓ the lead to the value obtained in **Fastest**(i, j). Starting from the root problem **Fastest**($1, k$), we can follow the pointer of choices to reconstruct the optimal sequence of multiplications.

5.4 $\text{\TeX}$

These notes (and hopefully yours too) are typeset using a system called $\text{\LaTeX}$, which is an extension (by Leslie Lamport) of an earlier system called $\text{\TeX}$, designed and largely written by Donald Knuth. This is why it looks so nice⁶. In the 1970's, Donald Knuth was writing his famous (and still unfinished) series on *The Art Of Computer Programming*. At the same time, the printing industry was shifting from old-fashioned hot metal typesetting to computer driven approaches. Knuth was so unsatisfied with the galley proofs of his book under the new system that he set out to invent his own digital typesetting system. Thus $\text{\TeX}$ was born, and he began using it to typeset his books. $\text{\TeX}$ was soon widely used in academia, and then a set of user-friendly macros by Lamport was bundled with $\text{\TeX}$ to form $\text{\LaTeX}$.

$\text{\TeX}$ takes unstructured text files as input and compiles it into PDF's and other print formats, similar to how programming code is compiled into binary executables. This is in contrast to WYSIWYG editors, such as Microsoft Word, that immediately update a visual document as you type, and make their typesetting decisions immediately. The compilation step of $\text{\TeX}$ gives it a chance to optimize various global criteria when making its many decisions, and ultimately produce professional quality documents.

Here we will discuss how $\text{\TeX}$ does line-breaking (simplified for the sake of discussion), via an algorithm first proposed by Knuth in a 1977 memo⁷ and later published by Knuth and Plass [KP81]. When typesetting text, a basic decision one has to make is where to wrap the line. These decisions make a big difference on the overall visual product. Having too many words in a line squeezes the white space and makes it harder to delineate words. Too few words leads to too much white space between them, which is distracting and also a waste of paper. By English convention, we restrict ourselves to breaking at white spaces between words or particular points where we can introduce a hyphen, rather than at all points within words. This chunking makes line wrapping that much clunkier.

One way to approach line-breaking is by the following greedy heuristic. Choose a minimum width for white space between words (e.g., half a centimeter). Each time we consider adding the next word to a line, calculate the amount of white space that would be between each pair of words if the new word were included. If this calculated value is above the minimum, then add the word to the line. If the value is below the minimum, then start a new line.

The greedy approach observes very local information in making its decisions. The fact that the choice of words in one line affects the possible options in all the remaining

⁶Sadly, beautiful typesetting cannot otherwise improve this author's prose.

⁷<https://www.saildart.org/TEXDR.AFT%5B1,DEK%5D>

lines is overlooked.

Towards a more comprehensive approach, let's try to formulate the overall problem clearly. I think we can all agree that we want the “best overall set of line breaks, over all possible ways of breaking up the text.” But what does it mean to be “best”? Knuth proposed we use a *scoring function* to *quantify* it. He proposed a loss function **loss** with the following format. **loss** takes as input a sequence of words $w_1, \dots, w_k$, imagines these words as a line, and outputs a numerical score that measures how “bad” the line would look. Here we treat **loss**($w_1, \dots, w_k$) as a black box, and refer to [KP81] for implementation details. Our goal, then, is to take a long sequence of words $w_1, \dots, w_n$, and divide them up into (consecutive) sub-sequences $(w_1, \dots, w_{i_1-1}), (w_{i_1}, \dots, w_{i_2-1}), \dots, (w_{i_k}, \dots, w_n)$ to minimize the total loss,

$$\mathbf{loss}(w_1, \dots, w_{i_1-1}) + \mathbf{loss}(w_{i_1}, \dots, w_{i_2-1}) + \dots + \mathbf{loss}(w_{i_k}, \dots, w_n).$$

This establishes a formal definition of the optimal line breaking. Given text in the form of a sequence of words, it is the partition of the sequence of word into intervals that minimizes the sum, over all such partitions, of the loss of that interval. This is well-defined because one can enumerate the finite number of ways to partition the text into intervals and for each compute the corresponding loss; the minimum loss gives the optimum line-breaking (as defined). The only issue is whether it can be computed efficiently.

Let's now define a recursive algorithm to compute the optimal line breaking. We start with a recursive spec. Fix a sequence of words $w_1, \dots, w_n$ that we want to partition. We want to compute the minimum loss of any partition of the words $(w_1, \dots, w_n)$ into intervals. For $i \in \mathbb{N}$, we define **BLB**(i) (which stands for “best line-breaking”) as

BLB(i) = minimum loss of any partition of the sequence of words $(w_i, \dots, w_n)$ into intervals.

We are precisely interested in **BLB**(1). Consider any subproblem of the form **BLB**(i). Note that if $i > n$ then there are no words to partition and the minimum loss is automatically 0. So we write:

$$\mathbf{BLB}(i) = 0 \qquad \text{if } i > n. \qquad (5.1)$$

Consider the more interesting case where $i \leq n$. Any partition of $(w_i, \dots, w_n)$ into intervals must, in particular, have a first interval $(w_i, \dots, w_j)$ that includes w_i , followed by a partition of all the words $(w_{j+1}, \dots, w_n)$ that come after that first interval. For the remaining partition, we can use induction (on $n - i$) to obtain the optimum

partition. We do not know, however, where to end the first interval. And after thinking about it, it is not clear how to strategically choose that first interval, at least without much more info about the inner mechanisms of `loss`. But we don't have to be clever. We can just try all possible choices; one of them will be optimal. Thus we have

$$\text{BLB}(i) = \min_{j=i, \dots, n} \{\text{loss}(w_i, \dots, w_j) + \text{BLB}(j+1)\} \quad \text{if } i \leq n \quad (5.2)$$

Between (5.1) and (5.2) we have a full recursive algorithm. Consider now the running time. Let $T(n)$ be the running time required to compute BLB over n words. We have

$$T(0) = 1, \\ T(n) = \sum_{i=0}^{n-1} T(i) + O(n).$$

This running time, if we worked it out, is enormous - much larger than 2^n . But let us cut off the calculations and intervene with the computer scientist's favorite (only?) trick: caching. Obviously, we are solving the same problem $\text{BLB}(i)$ many times even though the answer is always the same. So we apply dynamic programming: we store our answers in an n -dimensional array and never compute the same answer twice. Now the running time becomes

$$(\# \text{ subproblems}) \times (\text{time per subproblem}) = n \times O(n) = O(n^2).$$

The space usage is $O(n)$. And your notes look great.

5.5 Additional notes and references

The content in this chapter overlaps with [Eri19, chapter 3], [KT06, chapter 6], [DPV08, chapter 6], and [DD11, (video) lectures 19 – 21].

Spring 2024 lecture notes. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

Spring 2023 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

Spring 2022 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

5.6 Exercises

In addition to some of the exercises below, there are many freely available exercises with similar themes as this chapter. See for example the exercises in [Eri19, Chapter 3].⁸

Exercise 5.1. Below is a series of optimization problems that takes as input an array $A[1..n]$ of integers, and asks for optimal subsequences of A satisfying certain properties.⁹ Design and analyze an algorithm for each of these problems, addressing items 1–5 from section 5.2.¹⁰

1. A sequence of numbers $x_1, \dots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \dots, k - 1$. Compute the length of the longest increasing subsequence of A .
2. A sequence of numbers $x_1, \dots, x_k$ is *convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \dots, k - 1$. Compute the length of the longest convex subsequence of A .
3. Compute the maximum sum of any increasing subsequence of A .
4. Compute both:
 - the length of the longest increasing subsequences of A where the sum of integers is even,
 - the length of the longest increasing subsequence where the sum of integers is odd.
5. Suppose each entry in A is also colored red, white, or blue. We say that a sequence is *American* if the colors alternate red, white, blue, red, white, blue, ... The first number in the sequence can be any color. Compute the length of the

⁸The exercises in [Eri19, Chapter 3] under the heading of “Trees and Subtrees” are better suited for chapter 13 of these notes.

⁹A “subsequence” of $x_1, \dots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ where $1 \leq i_1 < i_2 < \dots < i_k \leq n$.

¹⁰ In particular, no proof is required. We will interpret your recursive spec as the inductive hypothesis of a proof by induction, and fill in the the rest of the details ourselves. That said, some additional commentary may help the grader, especially in the event of partial credit.

longest increasing American subsequence of A . (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

6. A sequence $x_1, \dots, x_k$ is a *palindrome* if the reversed sequence is the same; i.e., $(x_1, \dots, x_k) = (x_k, \dots, x_1)$. For example,

mom, dad, racecar, and gohangasalamimalasagnahog

are all palindromes. Compute the length of the longest palindrome subsequence of A .

7. Compute the length of the shortest supersequence of A that is a palindrome.¹¹
8. Suppose each entry of A is colored red, white or blue. We say that a sequence is *un-American* if the colors in the sequence never cycle through the American dream; that is, no three consecutive entries have colors that form any of the following three sequences: (red, white, blue), (white, blue, red), or (blue, red, white). Compute the length of the longest increasing un-American subsequence of A . (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)
9. (Just for fun) Invent your own fun and interesting version of a dynamic programming problem over A .

Exercise 5.2. Let $A[1..m]$ and $B[1..n]$ be two arrays. Each of the following problems asks for some aspect (e.g., the length) of a common subsequence of $A[1..n]$ and $B[1..n]$ satisfying or optimizing certain properties. (A common subsequence of A and B is a sequence that is a subsequence of both A and B .) Design and analyze an algorithm for each of these problems, addressing items 1–5 from section 5.2.¹⁰

1. Compute the length of the longest common subsequence of A and B .
2. Compute the length of the shortest common supersequence of A and B .
3. Compute the length of the longest common subsequence of A and B that is a palindrome.
4. Suppose A and B consists of integers. Compute the length of the longest common increasing subsequence of A and B .

¹¹ $x_1, \dots, x_k$ is a *supersequence* of $y_1, \dots, y_\ell$ if $y_1, \dots, y_\ell$ is a subsequence of $x_1, \dots, x_k$.

5. Compute the length of the shortest common palindrome supersequence of A and B .

Exercise 5.3. Problem 6 of exercise 5.1 considered the problem of computing the longest palindrome subsequence. Here we recall that a palindrome is a string spelled the same forwards and backwards, such as “racecar”. Here we consider the problem of computing the maximum weight palindrome subsequence where the characters of the input string are each given weights.

For example, consider the array of characters below where we annotate each character with its weight in the subscript.

x₈ r₅ y₁ a₃ c₇ z₄ e₂ c₄ x₋₅ a₅ r₋₁

Above, **racecar** is a palindrome subsequence, with total weight $22 = 5 + 3 + 7 + 2 + 4 + 5 - 1$.

The input consists of a string $A[1..n]$ and an array $W[1..n]$ of real-valued weights. Each $W[i]$ gives the weight of character $A[i]$. The goal is to compute the maximum total weight of any palindrome subsequence of $A[1..n]$. Design and analyze an algorithm for this problem.

Exercise 5.4. Let $A[1..n]$ be an array of n integers and let $B[1..n]$ be an array of n characters. We say that a subsequence $i_1, i_2, \dots, i_k$ of $1, \dots, n$ is *increasing in A* if the corresponding subsequence of A , $A[i_1], \dots, A[i_k]$, is (weakly) increasing (i.e., $A[i_1] \leq A[i_2] \leq \dots \leq A[i_k]$). We say that a subsequence $i_1, i_2, \dots, i_k$ of $1, \dots, n$ is a *palindrome in B* if the corresponding subsequence of B , $B[i_1], B[i_2], \dots, B[i_k]$, forms a palindrome.

Consider the problem of computing the length of the longest subsequence of $1, \dots, n$ that is both increasing in A and a palindrome in B . Design and analyze an algorithm for this problem.

Exercise 5.5. Recall that a sequence of numbers $y_1, y_2, \dots, y_k$ is *increasing* if $y_i < y_{i+1}$ for each i . Recall from MCQ that a common subsequence of two arrays A and B is another sequence that is a subsequence¹² of both A and B . Consider the problem of

¹²As always, a subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings C, DAMN, YAIIOAI, and DYNAMICPROGRAMMING are all subsequences of the string DYNAMICPROGRAMMING.

computing the length of the *longest common increasing subsequence* of two arrays of numbers, $A[1..m]$ and $B[1..n]$. For example,

$$(1, 4, 5, 6, 7, 9)$$

is the longest common increasing subsequence of the sequences

$$(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3) \text{ and } (1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5).$$

Design and analyze an algorithm for this problem.

Exercise 5.6. Consider the edit distance problem from section 5.1. In section 5.1, insertions, deletions, and substitutions were all treated equal. Suppose instead that we had different costs $\alpha, \beta, \gamma > 0$ for each insertion, deletion, and substitution, respectively. Show how to modify the algorithm from section 5.1 for these nonuniform costs. This model is useful in biological settings where α, β, γ reflect prior assumptions on the likelihood of each operation.

Exercise 5.7. Recall that different cost models are designed to reflect assumptions on how edits are naturally generated. For example, a reasonable hypothesis is that k consecutive deletions, or k consecutive insertions, are more likely than k independent single-character insertions or deletions. For example, in writing these notes, the author frequently finds himself deleting entire sentences¹³ at a time. We can factor this propensity into our edit distance model by a “insert-or-delete-at-bulk” cost. Let $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ be a fixed nonnegative function. We introduce the operations of “bulk insertion” and “bulk deletion”, where one deletes k characters or inserts a string of k characters, at a cost of $f(k)$, for any $k \in \mathbb{N}$. We assume that f is monotonically increasing in k ; that is, $f(k) < f(k+1)$ for all k . We can then define the *bulk edit distance* of two strings x and y to be the minimum cost of any sequence of edit, bulk insertion, or bulk deletion operations.

Assume that f is provided as a subroutine that takes $O(1)$ time to evaluate. Design and analyze an algorithm for this problem.

Exercise 5.8. Consider the line-breaking problem from section 5.4. Suppose that you are given an additional input parameter k , which defines the maximum number of words allowed in a line. (You may assume $k \leq n$, and morally, $k \ll n$). Show how to modify the algorithm from section 5.4 to give a faster running time (in terms of n and k).

¹³or paragraphs, or more... sigh...

Exercise 5.9. Suppose you are given a set of n points $P = \{(x_1, y_1), \dots, (x_n, y_n)\}$ in the plane.

1. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in S is comparable. (Cf. exercise 1.8 on page 31.)
2. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in S is incomparable.

Exercise 5.10. Who needs friends to play Set? The card game *Set* has 81 different cards varying in four features across three possibilities for each kind of feature. The four features are:

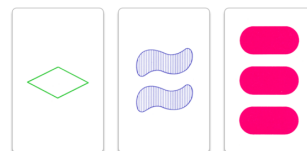
1. The shape, which is either a diamond, a squiggle, or an oval.
2. The number of shapes, which is either 1, 2, or 3.
3. The color of the shape, which is either red, green, purple.
4. The shading of the shape, which is either solid, striped, or open.

There are multiple variations of *Set* but they all are based on trying to form particular “sets” of three cards. A set consists of three cards satisfying *all* of these conditions:

1. They all have the same number or have three different numbers.
2. They all have the same shape or have three different shapes.
3. They all have the same shading or have three different shadings.
4. They all have the same color or have three different colors.

Loosely speaking, the rules of *Set* are summarized by: If you can sort a group of three cards into “two of _____ and one of _____”, then it is not a set.

For example, the three cards on the right form a set. We have one card of each color, one of each shape, one of each shading, and one of each number. (In case this is printed in black and white, we mention that the first card is green, the second is purple, and the third is red.)



We will study a solitaire version of *Set*. Solitaire *Set* starts with 3 decks of cards. Each round you choose one of the following options.

1. If the three cards on top of the deck form a set, you may remove all three from the top of their decks, and set them aside.
2. You may discard one card from the top of one of the decks, revealing a new card underneath (unless that deck is empty).

(Even if the top three cards form a set, you can choose to discard one of them instead.) The goal is to form as many sets as possible by the time you empty all the decks.

We will consider the game as an optimization problem where we also know the full sequence of cards in the deck. The three decks are represented by three arrays $A[1..m]$, $B[1..n]$, and $C[1..p]$.¹⁴ The cards are listed from top to bottom in each deck. For simplicity, we assume access to a constant-time subroutine $\text{IsItASet}(i, j, k)$ that takes as input three indices i, j, k and outputs **True** if the cards $(A[i], B[j], C[k])$ form a set, and **False** if they do not.

The problem is to compute the maximum number of sets that can be obtained in a game of Solitaire Set over the decks $A[1..m]$, $B[1..n]$, and $C[1..p]$. Design and analyze an algorithm for this problem.

Exercise 5.11. Recall that a sequence of numbers $x_1, \dots, x_k$ is *increasing* if $x_1 \leq x_2 \leq \dots \leq x_k$. Here we are interested in sequences that are increasing, but don't grow too fast, in the following sense.

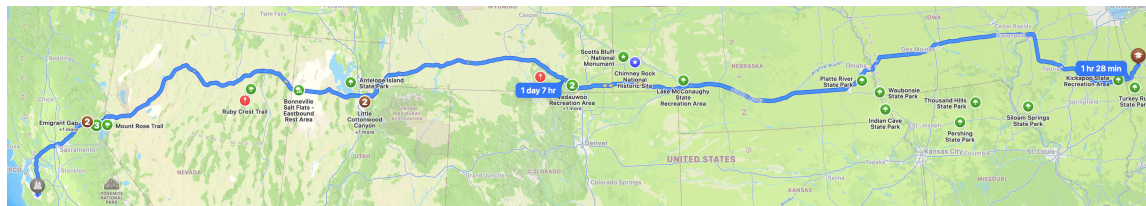
Recall that the Fibonacci sequence

$$1, 1, 2, 3, 5, 8, \dots$$

can be characterized as a sequence starting with 1, 1 and such that every successive number is the sum of the previous two numbers. ($1 + 1 = 2$, $1 + 2 = 3$, $2 + 3 = 5$, ...) Generalizing this notion, we say that a sequence $y_1, y_2, \dots, y_k$ is *sub-Fibonacci* if $y_i \leq y_{i-2} + y_{i-1}$ for $i = 3, 4, \dots, k$. That is, each successive number is at most the sum of the previous two.

Let $A[1..n]$ be an array of numbers. Design and analyze an algorithm that computes the length of the longest increasing and sub-Fibonacci subsequence of A .

Exercise 5.12. Imagine you are planning a long road trip to San Jose, CA and you've already fixed your route (through Illinois, Iowa, Nebraska, Wyoming, Utah, Nevada, and then California via Tahoe).



¹⁴The cards in these decks are not distinct; a particular card may have multiple copies. So, for example, there may be many more than 81 cards total.

You don't mind the drive, but you hate having to stop to get gas. What a pain! And some stations take longer than others. The car can drive at most 500 miles before needing to stop to get gas. The high-level goal is to plan your gas stops to minimize the amount of time spent pumping gas, while making sure you don't run out of gas between them.

We let n denote the total number of gas stations, and number the gas stations $1, \dots, n$ in order along the route. The input consists of two arrays $D[1..n]$ and $T[1..n]$, and an additional value $Z > 0$. For each i , $D[i] \in \mathbb{R}_{\geq 0}$ is a nonnegative real number representing the distance along the route at which the gas station appears. (Again, the gas stations are ordered to be nondecreasing in $D[i]$.) $T[i]$ is a positive real number representing the amount of time it takes to pump gas at station i . Z represents the distance to the destination. You start with a full tank of gas.

Design and analyze an algorithm to compute the minimum amount of time one has to spend at gas stations in going from location 0 to location Z , while ensuring that you never drive more than 500 miles without visiting a gas station. (You should return $+\infty$ if it is impossible to do so.)

Exercise 5.13. In fair redistricting, the goal is to divide up a population into geographic regions so that all the local elections are as close as possible. This is like the opposite of gerrymandering; having close elections maximizes the influence of each constituent's vote.

For simplicity we consider the following 1-dimensional formulation of the problem. Let $A[1..n]$ be a sequence of n (positive or negative) integers. Each index $i \in [n]$ represents a neighborhood block in a sequence of n blocks. We have two political parties, the Apples and the Oranges. For each index i , $A[i] \in \mathbb{Z}$ represents the number of people in block i that have registered for the Apple party, minus the number of people in block i that have registered for the Orange party; i.e.,

$$A[i] = \left(\begin{array}{c} \# \text{ Apples} \\ \text{in block } i \end{array} \right) - \left(\begin{array}{c} \# \text{ Oranges} \\ \text{in block } i \end{array} \right).$$

A *district* is defined as a consecutive interval $[i..j]$ of blocks, where $1 \leq i \leq j \leq n$. The *disparity* of a district $[i..j]$ is the absolute difference between the number of Apples and number of Oranges in the district:

$$(\text{disparity of } [i..j]) = \left| \left(\begin{array}{c} \# \text{ Apples} \\ \text{in blocks } i..j \end{array} \right) - \left(\begin{array}{c} \# \text{ Oranges} \\ \text{in blocks } i..j \end{array} \right) \right| = \left| \sum_{k=i}^j A[k] \right|.$$

Given $A[1..n]$ and $L \in [n]$, the goal is compute the minimum total disparity,

$$\sum_{k=1}^L (\text{disparity of } A[i_k..j_k])$$

over all partitions of the sequence blocks $[1..n]$ into L (nonempty) districts $[i_1..j_1], \dots, [i_L..j_L]$. (“Partition” means that the districts are disjoint and cover all of the n blocks.) Design and analyze an algorithm for this problem.

Exercise 5.14. Word segmentation is the problem of dividing a string of written language into its component words. In English and many other languages using some form of the Latin alphabet, the space is a good approximation of a word divider (word delimiter), although this concept has limits because of the variability with which languages emically regard collocations and compounds. Many English compound nouns are variably written (for example, ice box = ice-box = icebox; pig sty = pig-sty = pigsty) with a corresponding variation in whether speakers think of them as noun phrases or single nouns; there are trends in how norms are set, such as that open compounds often tend eventually to solidify by widespread convention, but variation remains systemic. In contrast, German compound nouns show less orthographic variation, with solidification being a stronger norm. However, the equivalent to the word space character is not found in all written scripts, and without it word segmentation is a difficult problem.

Languages which do not have a trivial word segmentation process include Chinese, Japanese, where sentences but not words are delimited, Thai and Lao, where phrases and sentences but not words are delimited, and Vietnamese, where syllables but not words are delimited. In some writing systems however, such as the Ge’ez script used for Amharic and Tigrinya among other languages, words are explicitly delimited (at least historically) with a non-whitespace character.

The Unicode Consortium has published a Standard Annex on Text Segmentation,[1] exploring the issues of segmentation in multiscript texts.¹⁵

We model the word segmentation problem as follows. Let $A[1..n]$ be a sequence of letters from a fixed alphabet. A *word segmentation* is a partition of A into contiguous intervals $A[i_0 + 1..i_1], A[i_1 + 1..i_2], \dots, A[i_\ell + 1..i_{\ell+1}]$; where $i_0 = 0$, $i_{\ell+1} = n$, and ℓ is any integer; such that each interval $A[i_j + 1..i_{j+1}]$ is a word. For example, the text

“youareawesome” has the word segmentation “you”, “are”, “awesome”.

¹⁵https://en.m.wikipedia.org/wiki/Text_segmentation#Word_segmentation

To determine which sequences form words, we assume access to a precomputed boolean array `words(1..n, 1..n)`, where for $i \leq j$, `words(i, j) = true` iff $A[i..j]$ forms a word.

Given $A[1..n]$ and `words(1..n, 1..n)`, consider the problem of deciding if there is a word segmentation of A . Design and analyze an algorithm for this problem.

Sample solution.

Problem. The following problem models the line breaking problem from section 5.4. The input consists of an array $X[1..n]$ where each $X[i]$ represents a word. We are also given a “loss function” `loss(i, j)` that takes as input two indices $i \leq j$, and outputs a numerical value that represents the “badness” of $X[i], \dots, X[j]$ as a line of text. For simplicity we assume that `loss(i, j)` takes constant time to evaluate.

The high-level goal is to partition X into contiguous subsequences

$$X[1..i_1], X[i_1 + 1..i_2], X[i_2 + 1..i_3], \dots, X[i_{k-1} + 1..n]$$

minimizing the total loss:

$$\text{loss}(1, i_1) + \text{loss}(i_1 + 1, i_2) + \text{loss}(i_2 + 1, i_3) + \dots + \text{loss}(i_{k-1} + 1, n).$$

Here the number of contiguous subsequences (above, k) is arbitrary. The problem is to compute the minimum loss over all such partitions of X .

Solution.

1. *Recursive spec / induction hypothesis.* For $i = 1, \dots, n + 1$, we define

`min-loss(i)` = the minimum loss over all partitions of $X[i..n]$. (For $i > n$, $X[i..n]$ indicates the empty sequence.)

2. *Recursive implementation.*

`min-loss(i)`

1. If $i > n$ then return 0.
2. Otherwise return the minimum, over all $j = i, i + 1, \dots, n$, of

$$\text{loss}(i, j) + \text{min-loss}(j + 1).$$

3. *Solving the original problem.* The solution to the problem is given by `min-loss(1)`.

4. *Mention “dynamic programming” or “caching”.* We apply dynamic programming to the recursive algorithm and cache the solutions to the subproblems.

5. *Running time.* For $i = 1, \dots, n$, `loss(i)` has a loop with at most n iterations, and each iteration takes constant time. So each subproblem takes $O(n)$ time. Over all n subproblems, the algorithm takes $O(n^2)$ time in total.

Sample solution.

Problem. (Exercise 5.1 problem 1) Let $A[1..n]$ be an array of integers. A sequence of numbers $x_1, \dots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \dots, k - 1$. Compute the length of the longest increasing subsequence of A .

Solution.

1. *Recursive spec / induction hypothesis.* For $i \in \{1, \dots, n\}$, we define

$\text{LIS}(i)$ = the length of the longest increasing subsequence of $A[i..n]$ where the subsequence starts with $A[i]$.

(Here LIS stands for longest increasing subsequence.)

2. *Recursive implementation.*

$\text{LIS}(i)$

1. If $A[i] \geq A[j]$ for all $j > i$ then return 1.
2. Otherwise return the maximum of

$$1 + \text{LIS}(j)$$

over all indices $j > i$ with $A[j] > A[i]$.

3. *Solving the original problem.* The maximum of $\text{LIS}(i)$ over all $i \in [n]$ gives the length of the longest increasing subsequence in A .

4. *Running time with caching / dynamic programming.* With dynamic programming, the running time is $O(n^2)$, where we have n subproblems each of which take $O(n)$ time (to iterate over $j > i$, in steps (1) and (2)).

Remark 5.1. We emphasize that the recursive spec essentially solved the entire problem.

Remark 5.2. It is well known that there are faster algorithms but that is besides the point.