

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

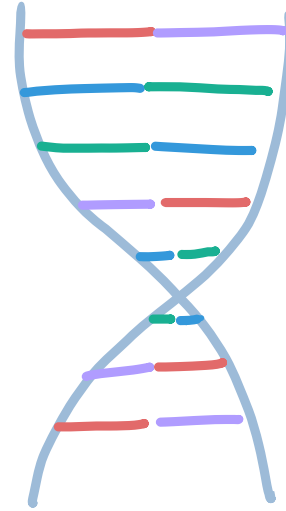
Ignore changes whose lines are all blank.

Genetics

I. An algorithmic metric on strings

A, C, G, T

A
C
G
T
G
C
T
A



how to measure similarity?

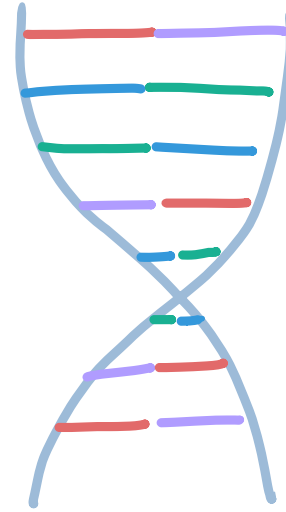
important for:

(a) analyzing hereditary traits

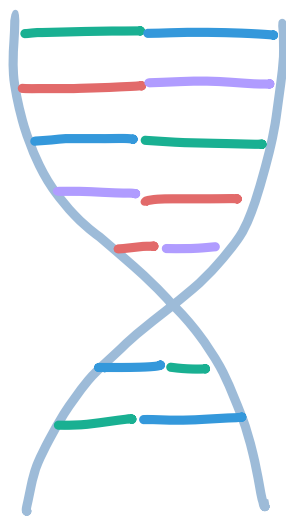
(b) distinguishing species

&

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,

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

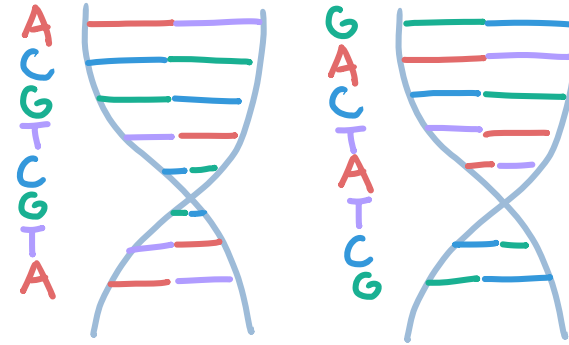
$x = \text{ABCDEFGG}$

$\times \times \times \times$

$y = \text{ABCEFGD}$

$\Rightarrow$ 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

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

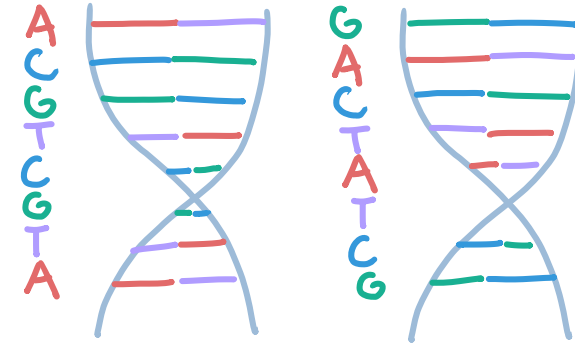
for strings x, y w/ same length

$x = \text{ABCDEFGG}$

$y = \text{ABCEFGD}$

$\Rightarrow$ 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. Edit distance

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

$x = \text{ABCDEFGG}$

$y = \text{ABCEFGD}$

$\Rightarrow$ edit distance 2

Hamming distance and edit distance

are both metrics:

$$\text{dist}: X \times X \Rightarrow \mathbb{R}_{\geq 0}$$

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

"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

$x[1..m], y[1..n]$

① Recursive specification

$\text{edit}(x[1..m], y[1..n])$: given two strings
 x, y , return min # 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 .

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

Designing a recursive algorithm

① Recursive specification

- declares input/output of algo
- "correct" 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

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

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

if $m = 0$: return n

if $n = 0$: return m

if $x[1] \neq y[1]$:

return $\min \begin{cases} 1 + \text{edit}(x[2..m], y[2..n]) \\ 1 + \text{edit}(x[2..m], y[1..n]) \\ 1 + \text{edit}(x[1..m], y[2..n]) \end{cases}$

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

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

ABCD —

BCDE —

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 $x[1]$

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
- "correct" algo must satisfy
- 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: $T(n) = 3T(n-1) + O(1)$

$$= O(1) + 3(3T(n-2) + O(1)) = 1 + 3 + 9 + \dots + 3^n$$

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 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

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

$$\sum_{i=1}^k 2^i$$

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

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

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 $x[1] \neq y[1]$ 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 $A[i, j]$.

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

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

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

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)

X

$O(mn)$ time total

caching reduced
running time
exponentially!

Caching, a.k.a. dynamic programming:

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

```
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)

See notes for full explanation & sample solutions

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)

```
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

$m \cdot n$ subproblems

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

$O(mn)$ time total

We don't want:

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

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"

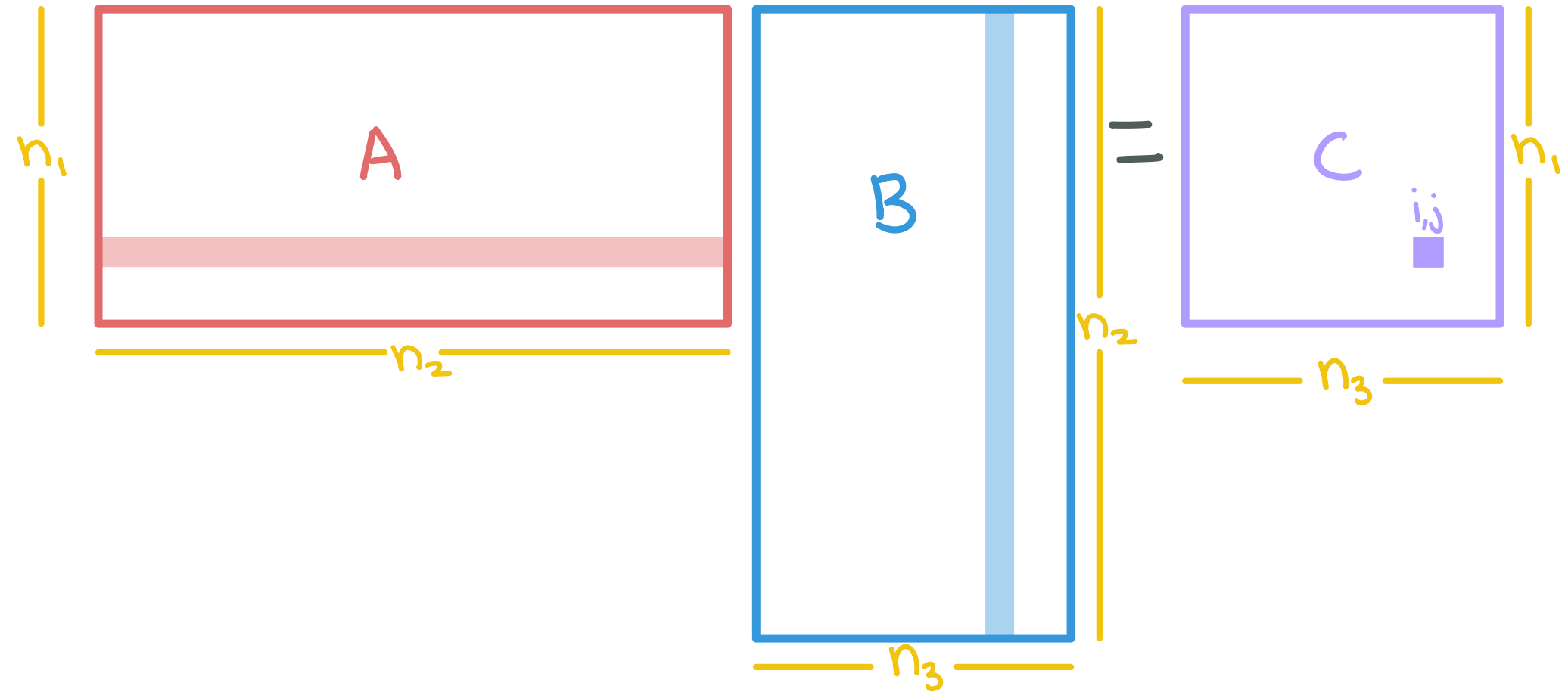
4. analyze run time

5. how to use recursive algo to solve problem

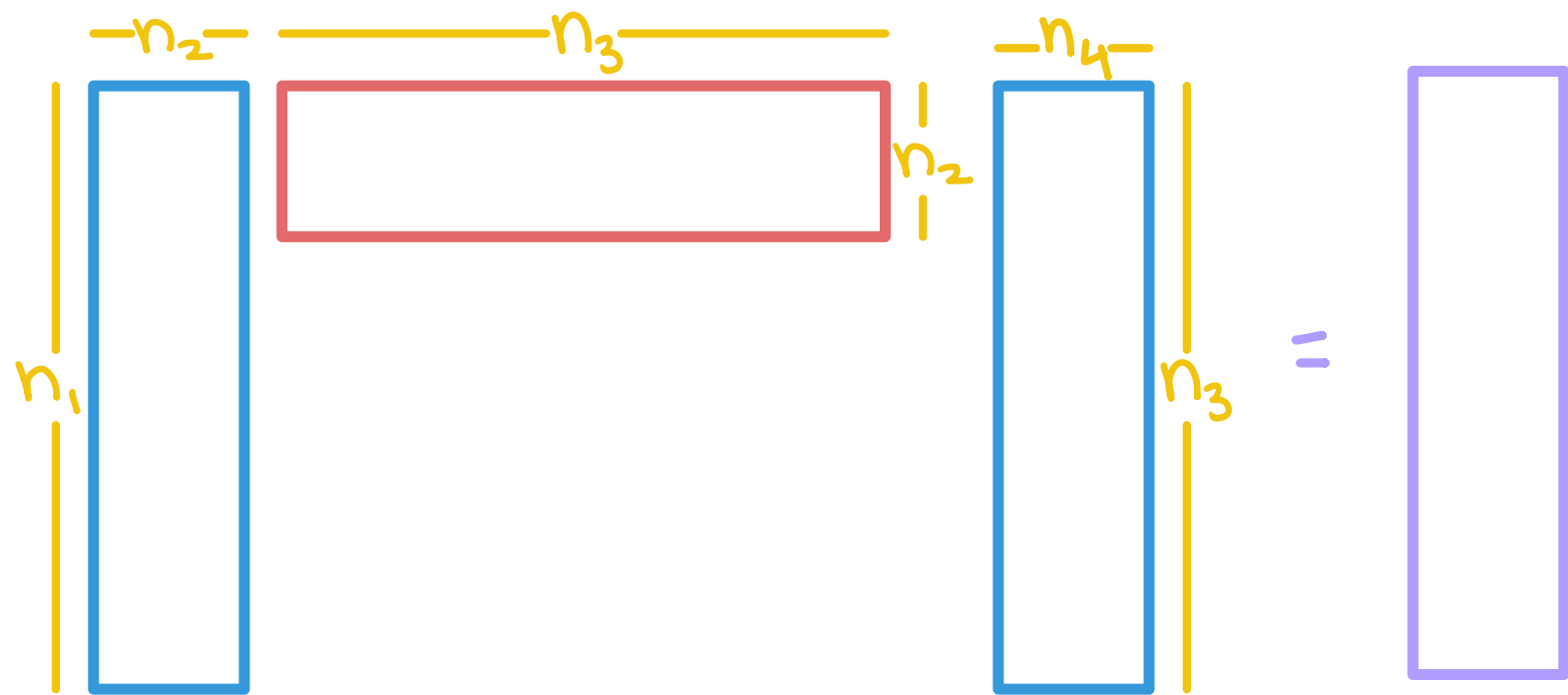
6. proof (by induction)

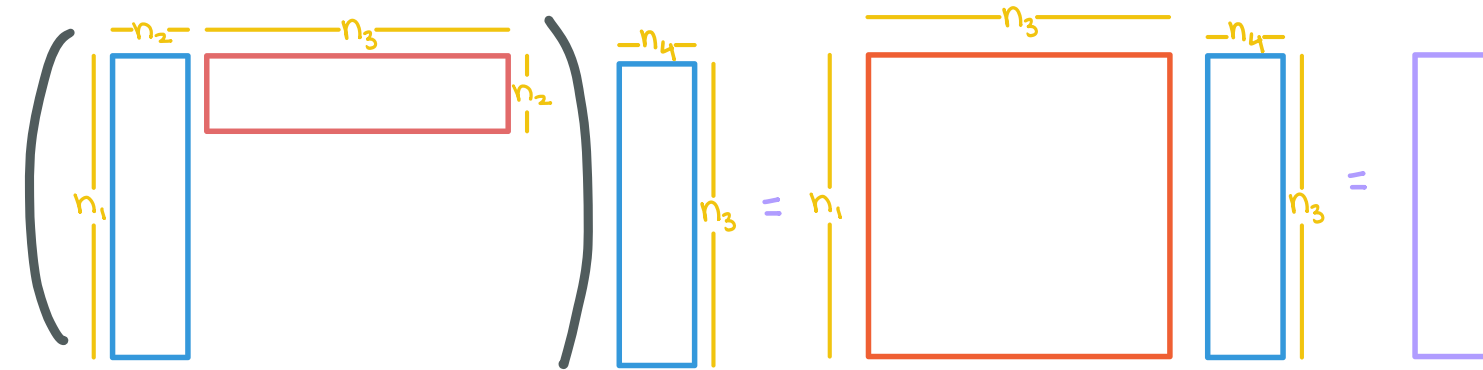
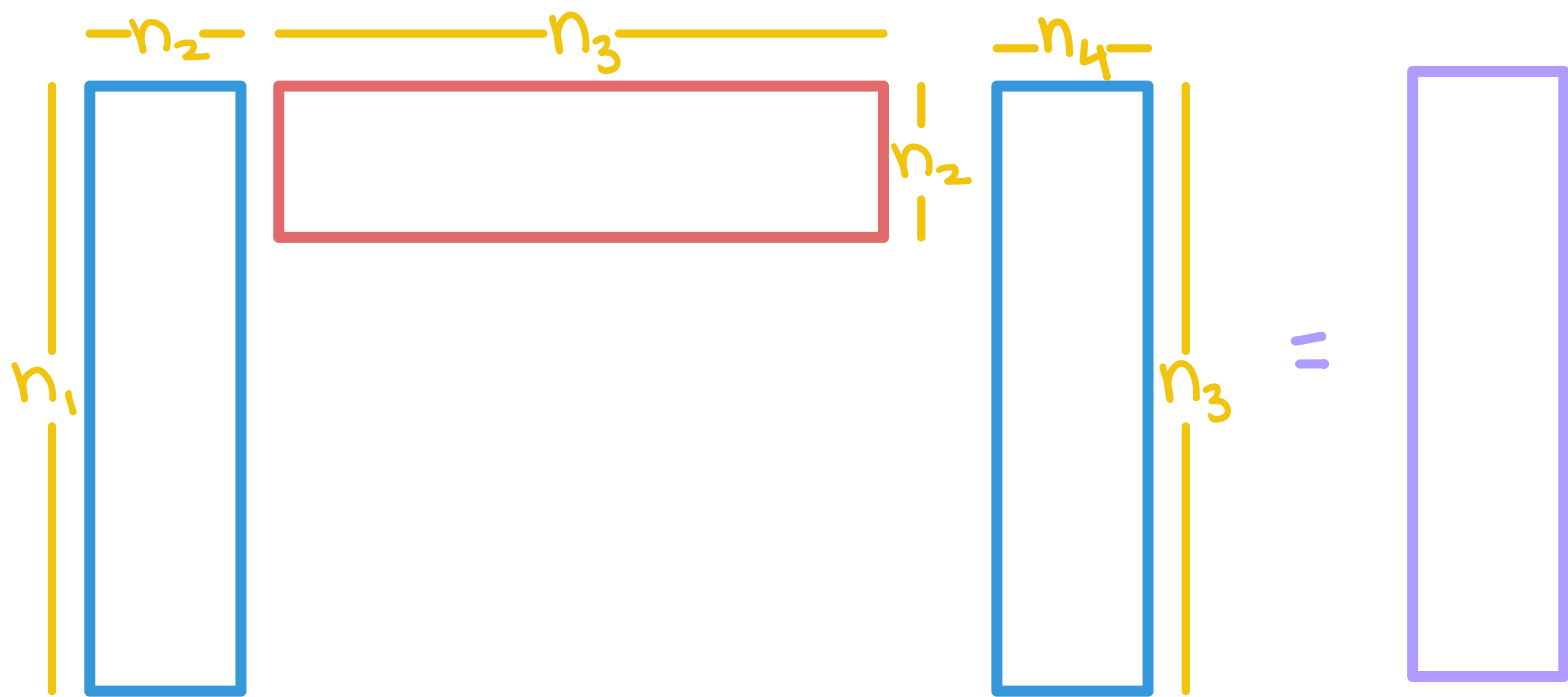
II Matrix chain multiplication

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

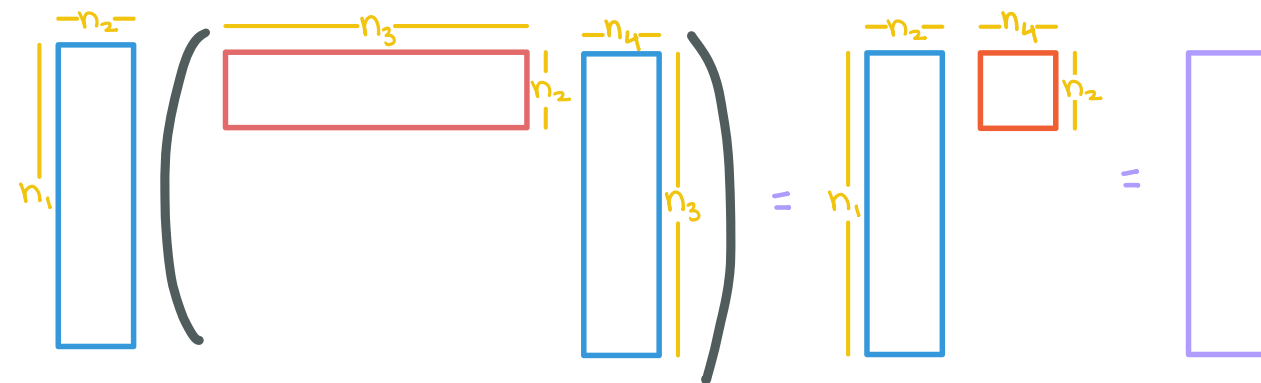


$O(n_1 n_2 n_3)$ time to compute C



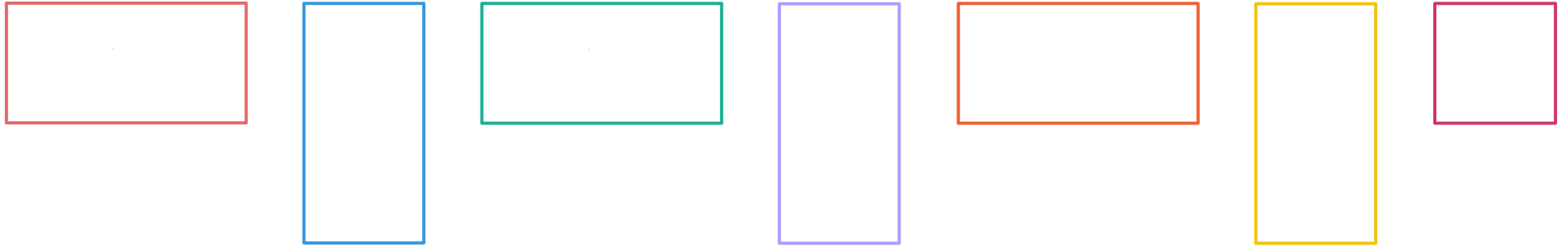


$$O(n_1 n_2 n_3 + n_1 n_3 n_4) \\ = O(n_1 n_3 (n_2 + n_4))$$



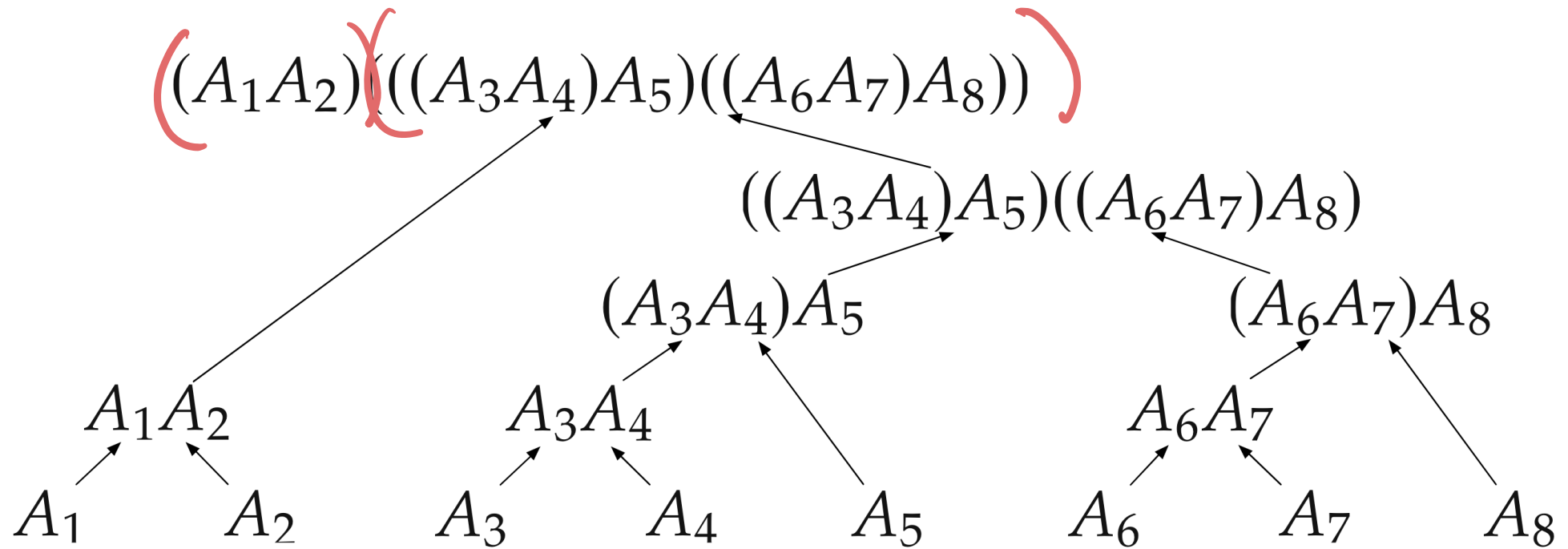
$$O(n_2 n_3 n_4 + n_1 n_2 n_4) \\ = O(n_2 n_4 (n_1 + n_3))$$

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

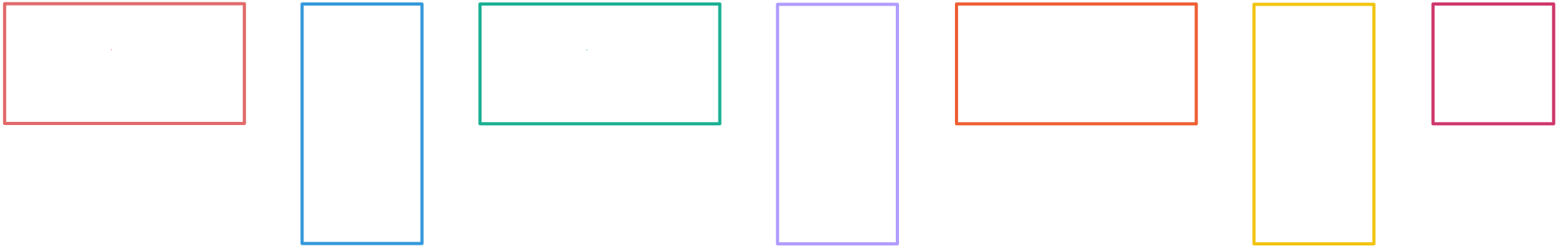


Output: min # (ordinary) multiplications needed to compute
 $A_1 A_2 A_3 \dots A_k$, where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$

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 $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$



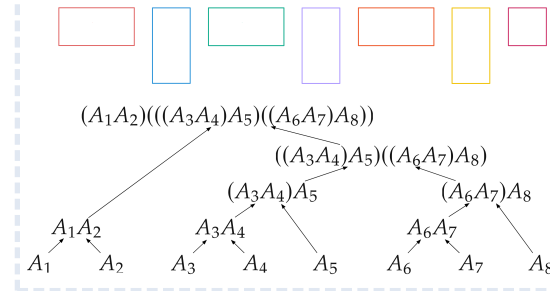
Ideas?

1. recursive spec

$$\text{mult}(A_1, A_2, \dots, A_k) = \min$$

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

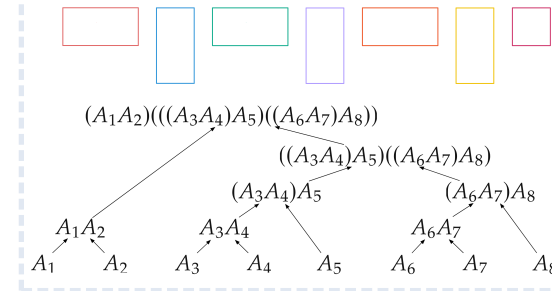
5. how to use recursive algo to solve problem

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

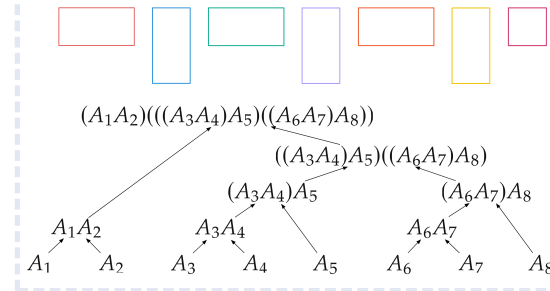
5. how to use recursive algo to solve problem

6. proof (by induction)

1. recursive spec

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

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 $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$



2. recursive implementation

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

if $i=j$ return 0

if $j=i+1$ return $\sim$

else

return $\min \{ \text{Fastest}(i, j) + n_i n_{j+1} n_{j+2}, \text{Fastest}(i, j-1) + n_{j-1} n_j n_{j+1} \}$

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

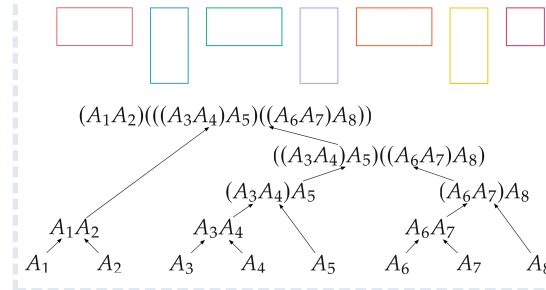
6. proof (by induction)

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 $A_i \dots A_j$.



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

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

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

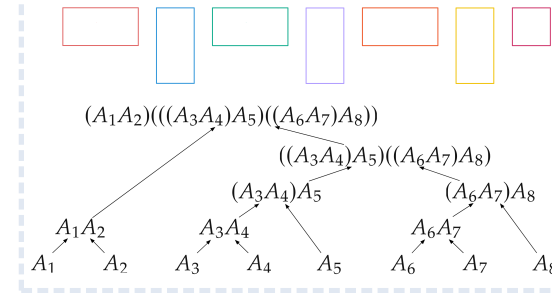
$\text{Fastest}(i, j)$ = the minimum number of operations to compute the matrix product $A_i \dots A_j$.

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

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

template for this class:

1. recursive spec

2. recursive implementation

3. how to use recursive algo to solve problem

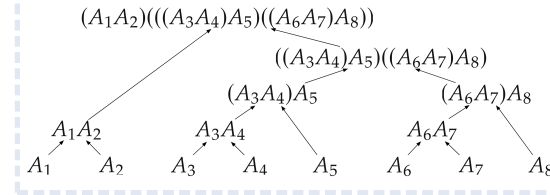
4. mention "caching" or "dynamic programming"

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.

n^2 subproblems

n time per subproblem

$O(n^3)$ 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 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 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.

1, 5, 3, 2, 11, 6, 7

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)

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 .

★ $LIS(i)$: length of LIS in $A[i..n]$
that includes i .

$LIS(i)$:

if $i=n$: return 1

if $i>n$: return 0

return max of

$1 + LIS(j)$

over $j=i+1, \dots, n$ s.t.

$A[j] > A[i]$

~~or $LIS(i+1)$~~

"driver" returns $\max_{i=1, \dots, n} LIS(i)$

5, 10, 7, 1, 2, 3, 4, 9, 5

"caching"

n subproblems

$\times n$

$O(n^2)$

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)

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 .

prove by induction that LIS implements
recursive spec

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)

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 .

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)

Problem. (Exercise 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 6.1. We emphasize that the recursive spec essentially solved the entire problem.

Remark 6.2. It is well known that there are faster algorithms but that is besides the point.

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 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 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 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 gohangasalamiimalasagnahog

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

⁸The exercises in [Eri19, Chapter 3] under the heading of “Trees and Subtrees” are better suited for ?? 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 $1 \leq i_1 < i_2 < \dots < i_k \leq n$.

¹⁰In 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.

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 $x_1, \dots, x_k$.

Sample solution.

Problem. (Exercise 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 6.1. We emphasize that the recursive spec essentially solved the entire problem.

Remark 6.2. It is well known that there are faster algorithms but that is besides the point.

References

- [DD11] Erik Demaine and Srinivas Devadas. *6.006: Introduction to Algorithms*. Fall 2011. Course website at <http://courses.csail.mit.edu/6.006/fall11/staff.shtml>. Video lectures posted at https://youtube.com/playlist?list=PLU14u3cNGP610q3tWYp6V_F-5jb5L2iHb.
- [DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008. Draft version available at <https://book.huihoo.com/pdf/algorithms/>.