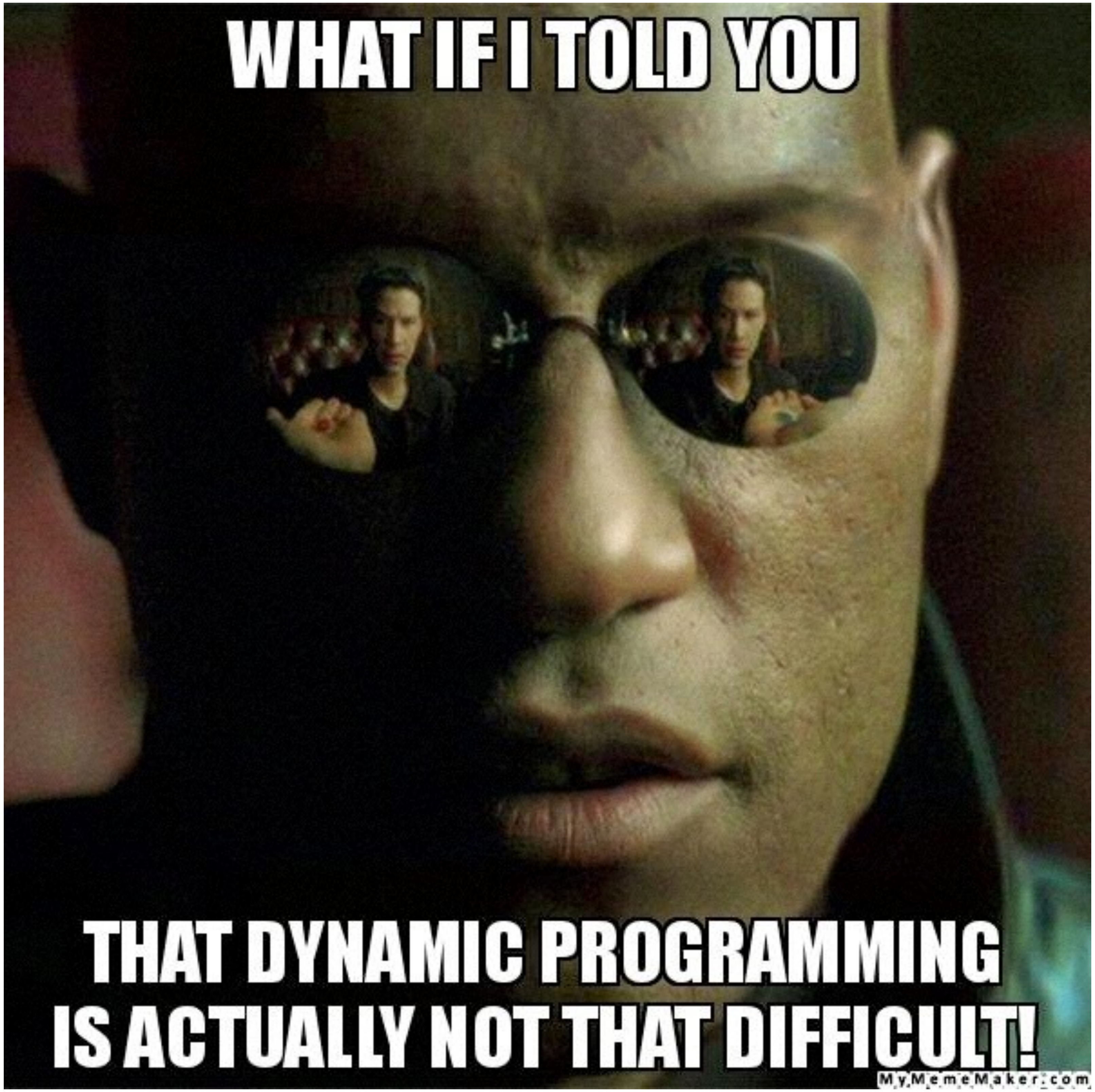
Algorithms Lab

Dynamic Programming


WHAT IF I TOLD YOU
THAT DYNAMIC PROGRAMMING
IS ACTUALLY NOT THAT DIFFICULT!
MyMemeMaker.com

Example problem: A, B two arrays of positive numbers. You can remove *a* last elements from A and *b* last elements from B (both *a* and *b* have to be at least 1), and the cost of such operation is (X - *a*) * (Y - *b*), where X (Y) is the sum of the removed elements from A (B).

Your **goal** is to **remove all elements from A and B** by repeatedly applying such operation, with the **minimal total cost**!

2 5 10 2 6

7 1 9 4 2

Cost:

Total cost:

Example problem: A, B two arrays of positive numbers. You can remove a last elements from A and b last elements from B (both a and b have to be at least 1), and the cost of such operation is $(X - a) * (Y - b)$, where X (Y) is the sum of the removed elements from A (B).

Your goal is to **remove all elements from A and B** by repeatedly applying such operation, with the **minimal total cost**!

2 5 10 ~~2 6~~

7 1 ~~9 4 2~~

Cost: $(2 + 6 - 2) * (9 + 4 + 2 - 3) = 72$

Total cost: 72

Example problem: A, B two arrays of positive numbers. You can remove *a* last elements from A and *b* last elements from B (both *a* and *b* have to be at least 1), and the cost of such operation is (X - *a*) * (Y - *b*), where X (Y) is the sum of the removed elements from A (B).

Your **goal** is to **remove all elements from A and B** by repeatedly applying such operation, with the **minimal total cost**!

2 ~~5 10~~ ~~2 6~~

7 ~~1~~ ~~9 4 2~~

Cost:

Total cost: 72

Example problem: A, B two arrays of positive numbers. You can remove *a* last elements from A and *b* last elements from B (both *a* and *b* have to be at least 1), and the cost of such operation is (X - *a*) * (Y - *b*), where X (Y) is the sum of the removed elements from A (B).

Your **goal** is to **remove all elements from A and B** by repeatedly applying such operation, with the **minimal total cost**!

2 ~~5 10~~ ~~2 6~~

7 ~~1~~ ~~9 4 2~~

Cost: (5 + 10 - 2) * (1 - 1) = 0

Total cost: 72

Example problem: A, B two arrays of positive numbers. You can remove *a* last elements from A and *b* last elements from B (both *a* and *b* have to be at least 1), and the cost of such operation is (X - *a*) * (Y - *b*), where X (Y) is the sum of the removed elements from A (B).

Your **goal** is to **remove all elements from A and B** by repeatedly applying such operation, with the **minimal total cost**!

~~2~~ ~~5 10~~ ~~2 6~~

~~7~~ ~~1~~ ~~9 4 2~~

Cost:

Total cost: 72

Example problem: A, B two arrays of positive numbers. You can remove *a* last elements from A and *b* last elements from B (both *a* and *b* have to be at least 1), and the cost of such operation is (X - *a*) * (Y - *b*), where X (Y) is the sum of the removed elements from A (B).

Your **goal** is to **remove all elements from A and B** by repeatedly applying such operation, with the **minimal total cost**!

~~2~~ ~~5 10~~ ~~2 6~~

~~7~~ ~~1~~ ~~9 4 2~~

Cost: (2 - 1) * (7 - 1) = 6

Total cost: 78

First approach - brute force

Recursively try all possible removals

rec_try(i, j) *// consider only first i elements of*
// A and j elements of B

if (i == 1) return (A[1] - 1) * ($\sum_{t=1}^{j} B[t]$ - j)

if (j == 1) return (B[1] - 1) * ($\sum_{t=1}^{i} A[t]$ - i)

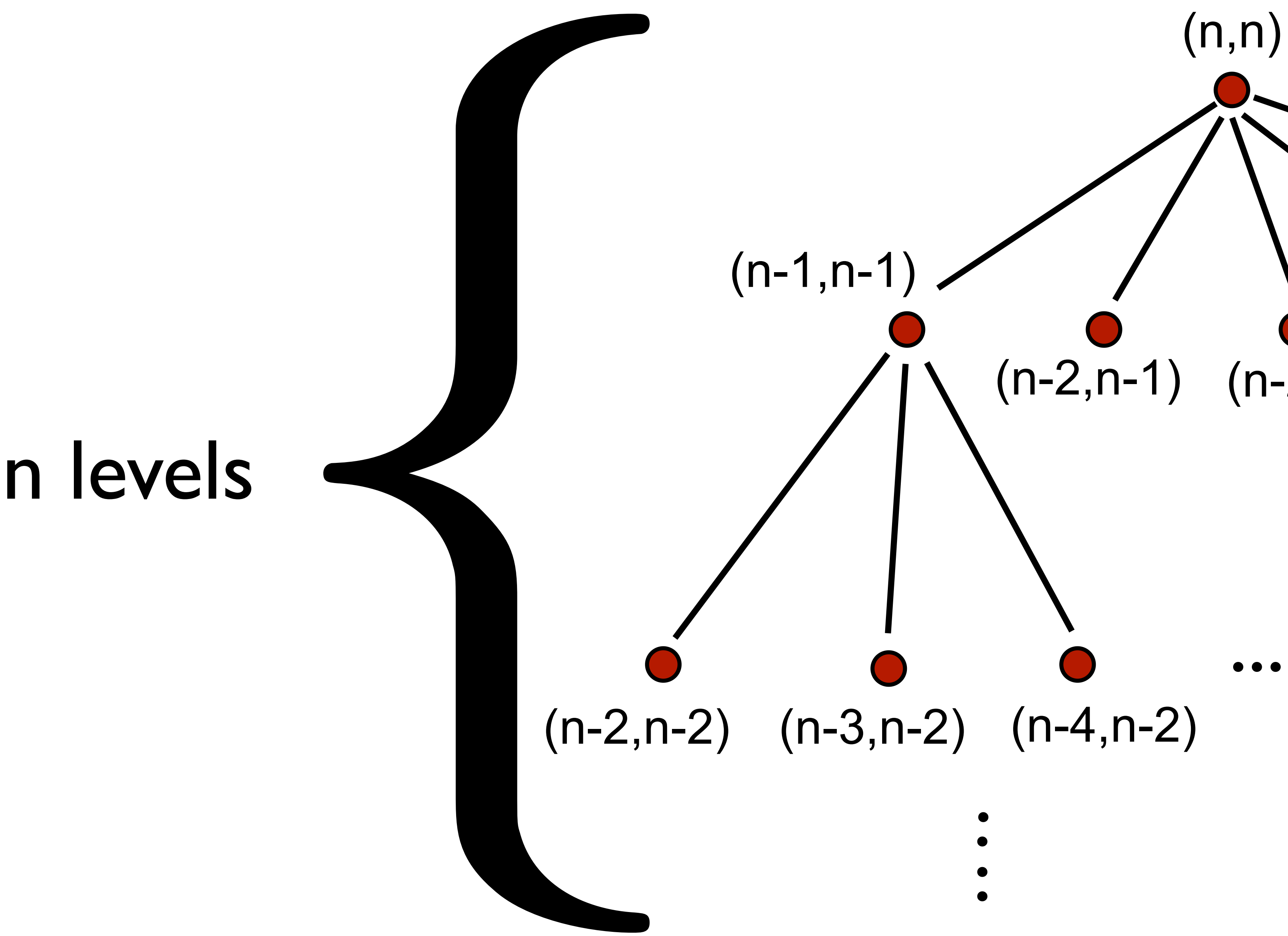
best =($\sum_{t=1}^{i} A[t]$ - i) * ($\sum_{t=1}^{j} B[t]$ - j) *// take all*

for a = 1 to i - 1
  for b = 1 to j - 1
    cost = ($\sum_{t=i-a+1}^{i} A[i]$ - a) * ($\sum_{t=j-b+1}^{i} B[i]$ - b)
    if (cost + **rec_try**(i - a, j - b) < best)
      best = cost + **rec_try**(i - a, j - b)

return best

First approach - brute force

Recursively try all possible removals



Rough estimate - at least 2^n many calls of try_rec()

First approach - brute force

Recursively try all possible removals

Problem: for some (i,j), we call **try_rec(i,j)** many times

Observation: for a fixed (i,j), **try_rec(i,j)** always returns the same value!

Solution: for each (i,j), store the value which **try_rec(i,j)** returns! Only call **try_rec(i,j)** if this value is not stored yet!

Second approach - brute force

with storing

rec_try(i, j) // consider only first i elements of
// A and j elements of B

if (i == 1) return (A[1] - 1) * ($\sum_{t=1}^{j} B[t]$ - j)

if (j == 1) return (B[1] - 1) * ($\sum_{t=1}^{i} A[t]$ - i)

best =($\sum_{t=1}^{i} A[t]$ - i) * ($\sum_{t=1}^{j} B[t]$ - j) // take all

for a = 1 to i - 1
for b = 1 to j - 1
cost = ($\sum_{t=i-a+1}^{i} A[i]$ - a) * ($\sum_{t=j-b+1}^{i} B[i]$ - b)

if (not stored(i - a, j - b)) rec_try(i-a, j-b)
if (cost **+ stored(i - a, j - b)** < best)
best = cost **+ stored(i - a, j - b)**

store(i,j) <- best

Second approach - brute force with storing

try_rec(i,j) will be executed at most once for each pair (*i*,*j*), $1 \leq i, j \leq n$

two nested loops: one up to *i*, one up to *j*

Total running time: $\mathcal{O}(n^4)$

This was an example of
Dynamic Programming!

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

Cost of taking first 10 elements from array A and 16 from B

$$((\sum_{i=1}^{10} A[i]) - 10) \cdot ((\sum_{i=1}^{16} B[i]) - 16)$$

*In the analysis we remove elements from the beginning and in the code from the end

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

Cost of taking first 10 elements from array A and 16 from B

$$((\sum_{i=1}^{10} A[i]) - 10) \cdot ((\sum_{i=1}^{16} B[i]) - 16)$$

$$= ((\sum_{i=1}^{10} (A[i] - 1)) \cdot ((\sum_{i=1}^{16} (B[i] - 1))$$

*In the analysis we remove elements from the beginning and in the code from the end

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

To make analysis simpler

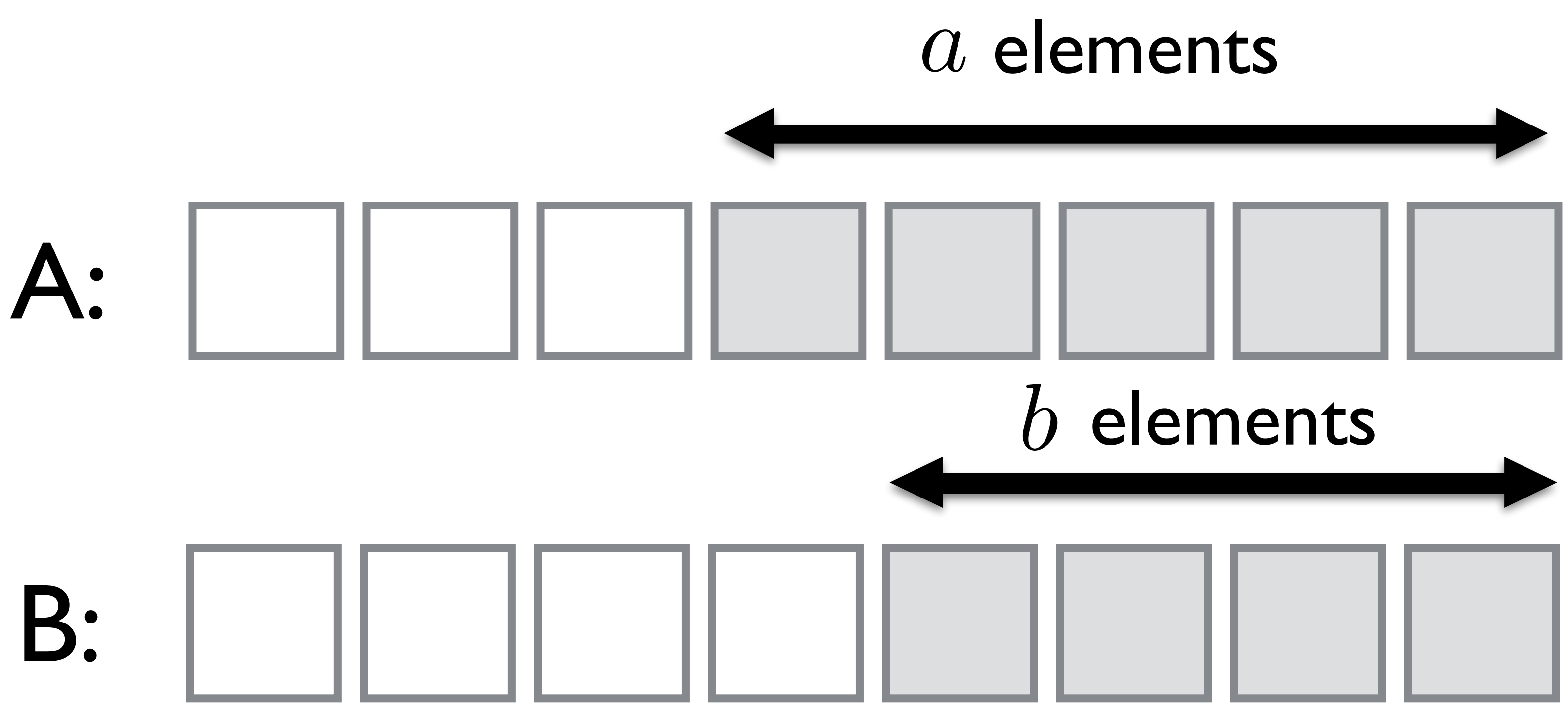
- Decrease every element of both A and B by 1
- Cost of taking a elements from A and b from B is

$$(\sum_{i=1}^{a} A[i]) \cdot (\sum_{i=1}^{b} B[i])$$

*In the analysis we remove elements from the beginning and in the code from the end

Can we improve the running time of the previous algorithm?

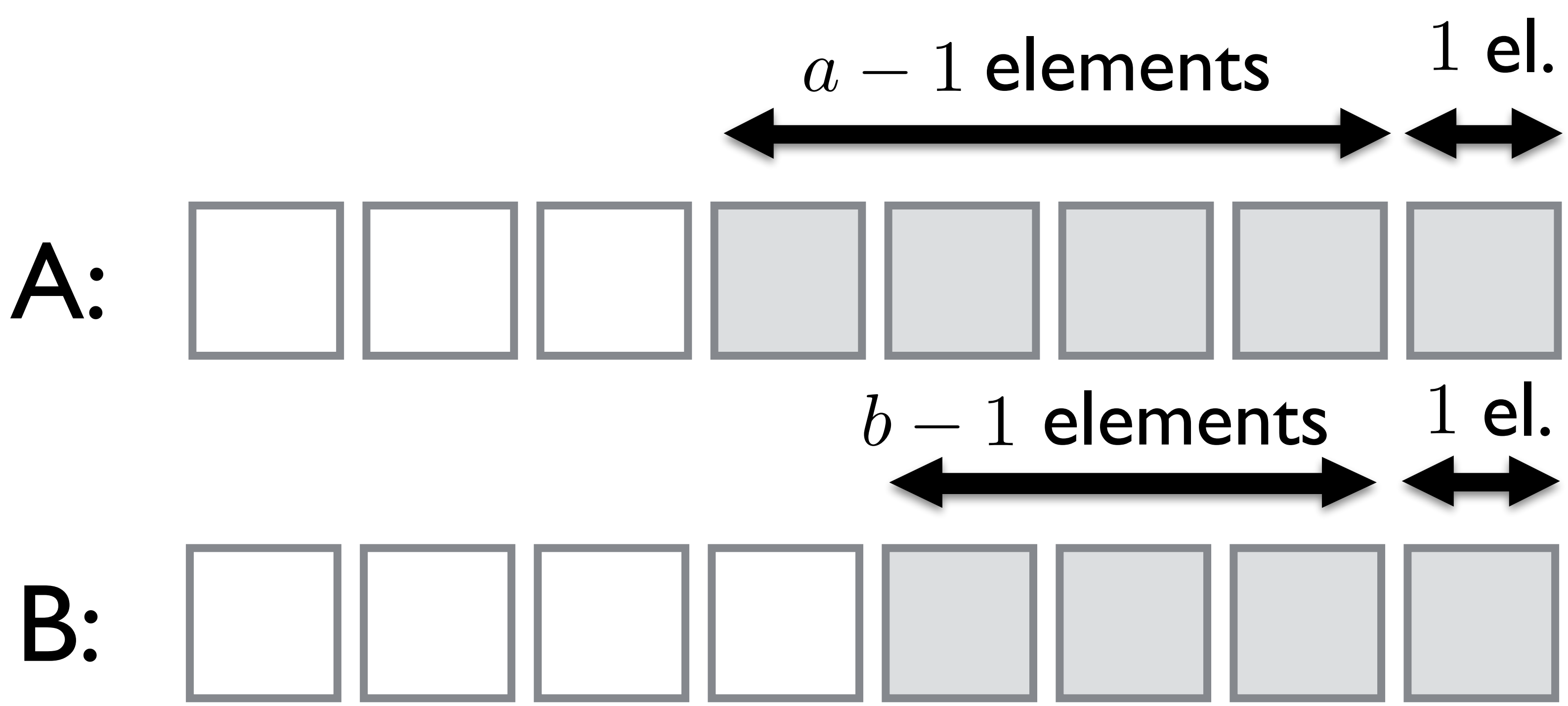
Let us observe the cost function more closely



Cost 1: $(A[1] + A[2] + \cdots + A[a]) \cdot (B[1] + B[2] + \cdots + B[b])$

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

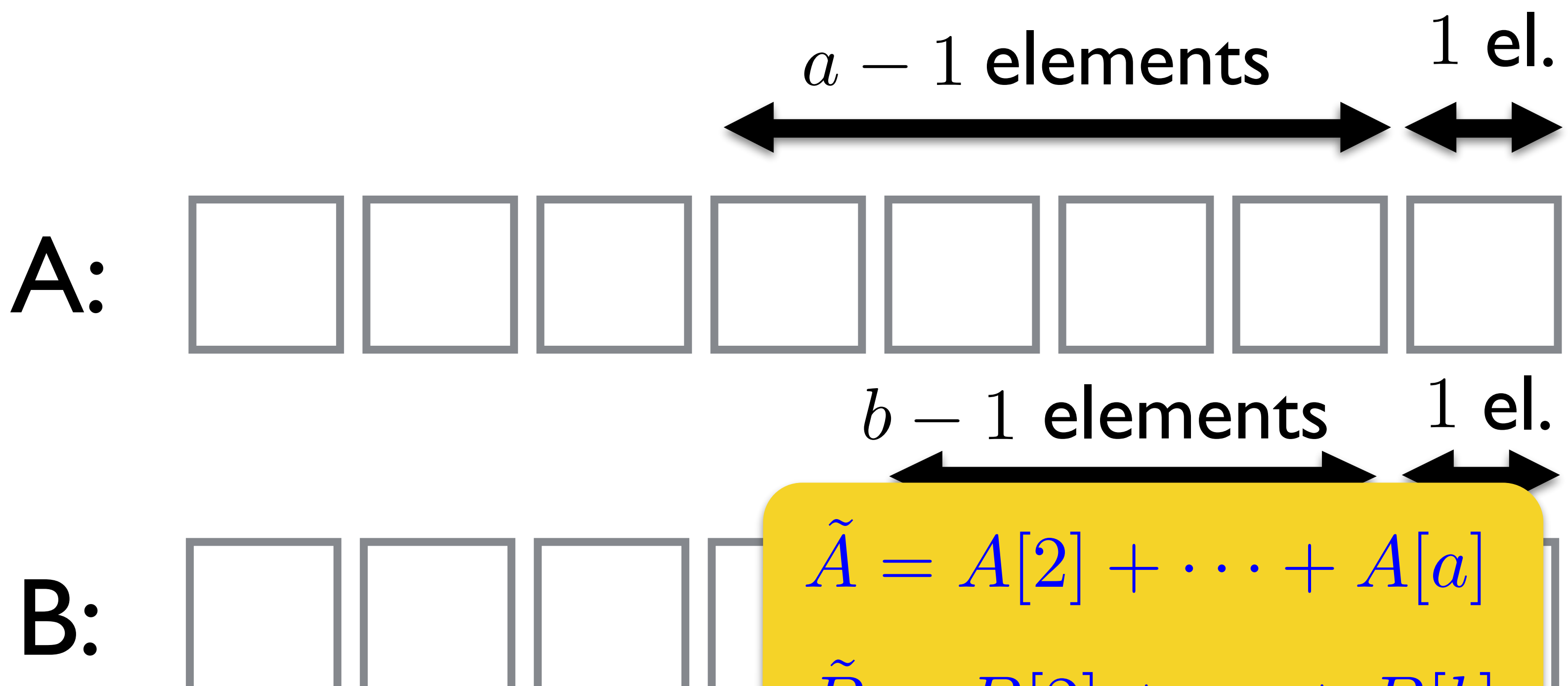


Cost 1: $(A[1]+A[2]+\cdots+A[a])\cdot(B[1]+B[2]+\cdots+B[b])$

Cost 2: $A[1]B[1]+(A[2]+\cdots+A[a])(B[2]+\cdots+B[b])$

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely



$\tilde{A} = A[2] + \cdots + A[a]$

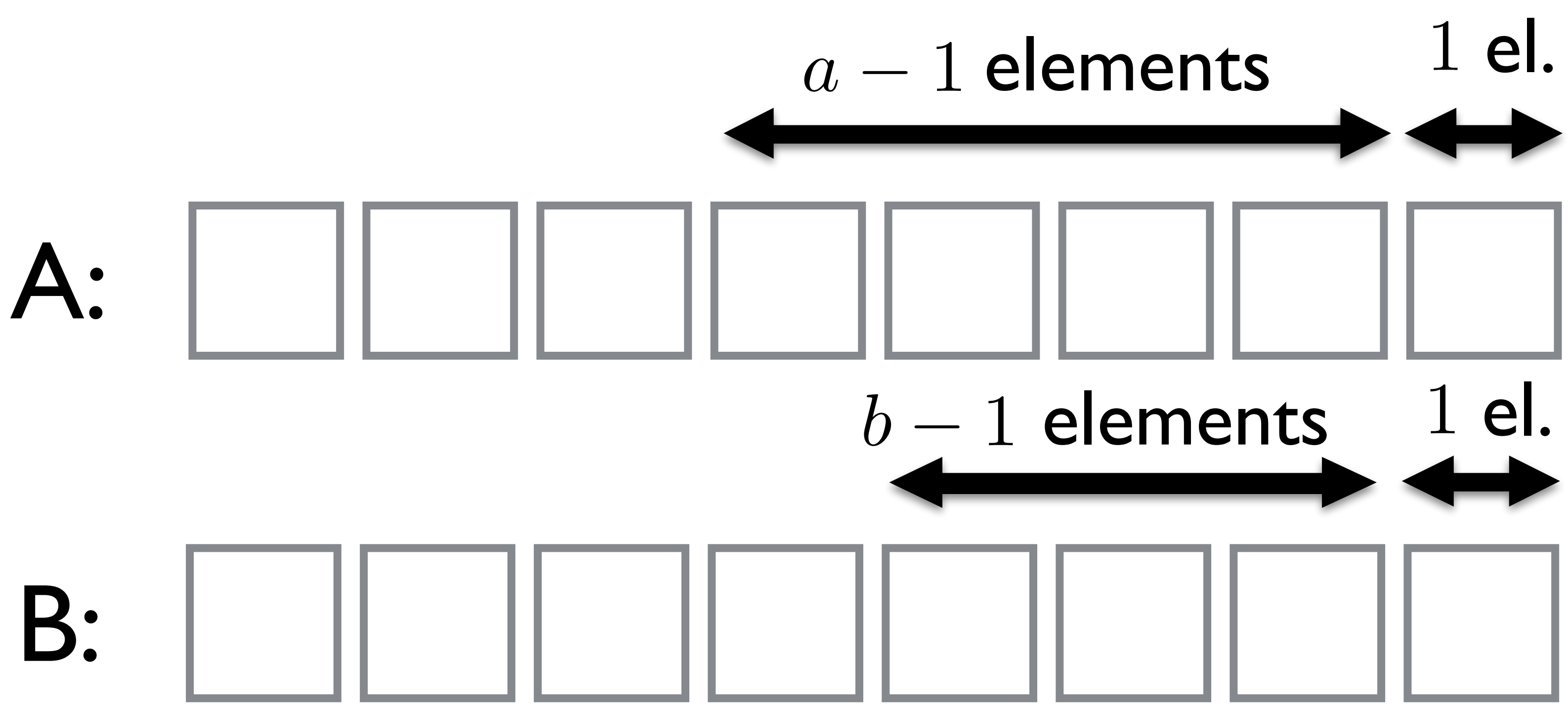
$\tilde{B} = B[2] + \cdots + B[b]$

Cost 1: $(A[1] + A[2] + \cdots + A[a]) \cdot (B[1] + B[2] + \cdots + B[b])$

Cost 2: $A[1]B[1] + (A[2] + \cdots + A[a])(B[2] + \cdots + B[b])$

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

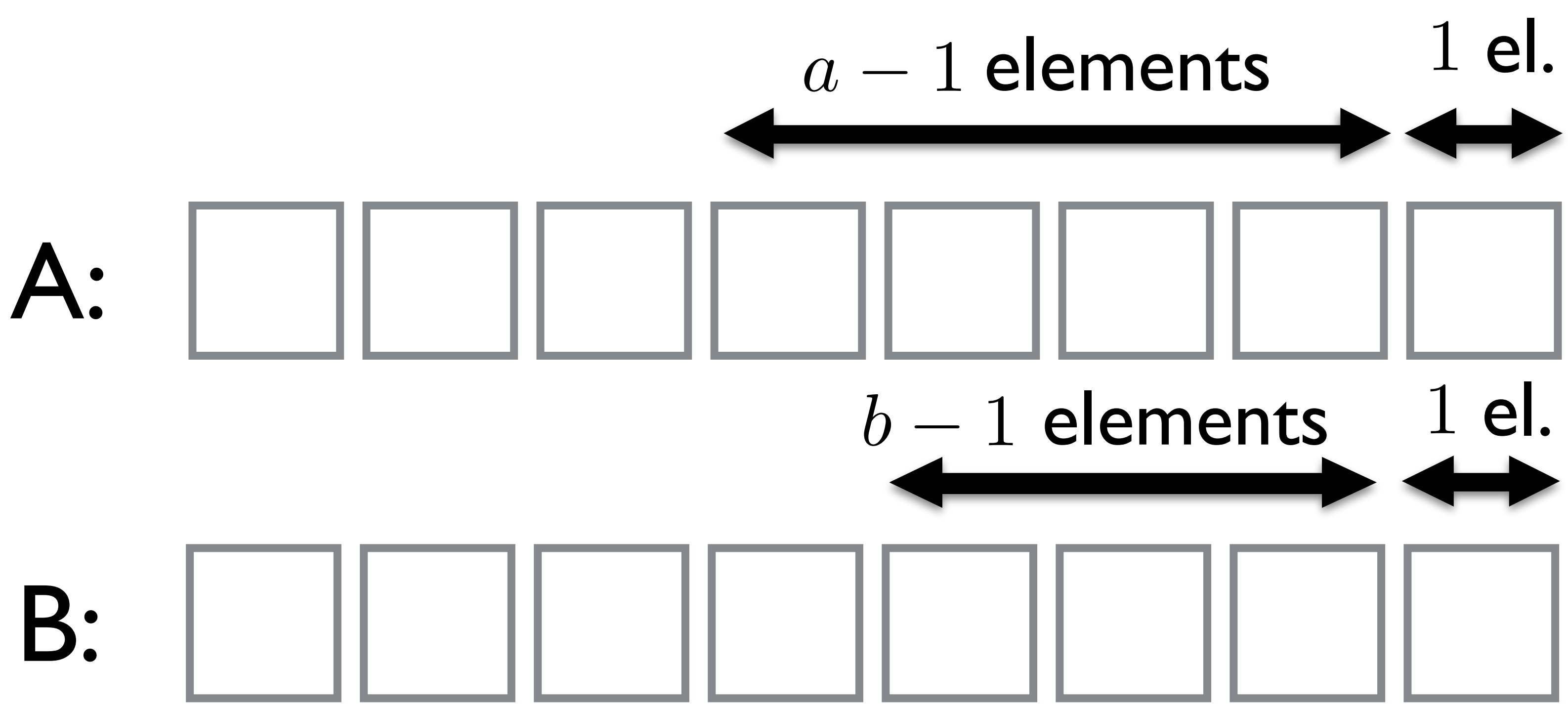


Cost 1: $(A[1]+\tilde{A})(B[1]+\tilde{B})$

Cost 2: $A[1]B[1]+\tilde{A}\tilde{B}$

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely



Cost 1: $(A[1]+\tilde{A})(B[1]+\tilde{B}) = A[1]B[1] + \tilde{A}\tilde{B} + A[1]\tilde{B} + \tilde{A}B[1]$

Cost 2: $A[1]B[1] + \tilde{A}\tilde{B}$

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

Conclusion?

Always take one element from each array. Right?

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely

Conclusion?

Always take one element from each array. Right? **No!**

Previous analysis only works when $a > 1$ and $b > 1$

Therefore: there exists an optimal strategy such that each removal takes either **exactly one element from A or exactly one element from B**!!!

Third approach

```
rec_try(i, j)  // consider only first i elements of
               // A and j elements of B

   if (i == 1) return (A[1] - 1) * ( Σ_{t=1}^{j} B[t]  - j)

   if (j == 1) return (B[1] - 1) * ( Σ_{t=1}^{i} A[t] - i)
   best =( Σ_{t=1}^{i} A[t] - i) * ( Σ_{t=1}^{j} B[t] - j) // take all

   for a = 1 to i - 1 // take one element from B
       cost = ( Σ_{t=i-a+1}^{i} A[i] - a) * (B[j] - 1)

       if (not stored(i - a, j - 1)) rec_try(i-a, j - 1)
       if (cost + stored(i - a, j - 1) < best)
          best = cost + stored(i - a, j - 1)

   for b = 1 to j - 1
       similar ...

   store(i,j) <- best
```

$$\text{cost} = \left(\sum_{t=i-a+1}^{i} A[i] - a\right) * (B[j]-1)$$

Third approach

try_rec(i,j) will be executed at most once
for each pair (*i*,*j*), $1 \leq i, j \leq n$

two **non-nested** loops: one up to *i*, one up to *j*

Total running time: $\mathcal{O}(n^3)$