Algorithms Lab

Dynamic Programming

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 * Y, 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 * Y, 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) * (9 + 4 + 2) = 120

Total cost: 120

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 * Y, 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: 120

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 * Y, 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) * (1) = 15

Total cost: 135

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 * Y, 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: 135

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 * Y, 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) * (7) = 14

Total cost: 149

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
```

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
```

$$A[1] \cdot (B[1] + \ldots + B[j])$$

```
    if (j == 1) return
```

$$(A[1] + \ldots + A[i]) \cdot B[1]$$

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]·(B[1]+...+B[j])

    if (j == 1) return   (A[1]+...+A[i])·B[1]

    best = (A[1]+...+A[i])·(B[1]+...+B[j])  // take all
```

$$A[1] \cdot (B[1] + \ldots + B[j])$$

$$(A[1] + \ldots + A[i]) \cdot B[1]$$

$$(A[1] + \ldots + A[i]) \cdot (B[1] + \ldots + B[j])$$

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]·(B[1]+...+B[j])

   if (j == 1) return   (A[1]+...+A[i])·B[1]

   best = (A[1]+...+A[i])·(B[1]+...+B[j])  // take all

   for a = 1 to i - 1
     for b = 1 to j - 1
```

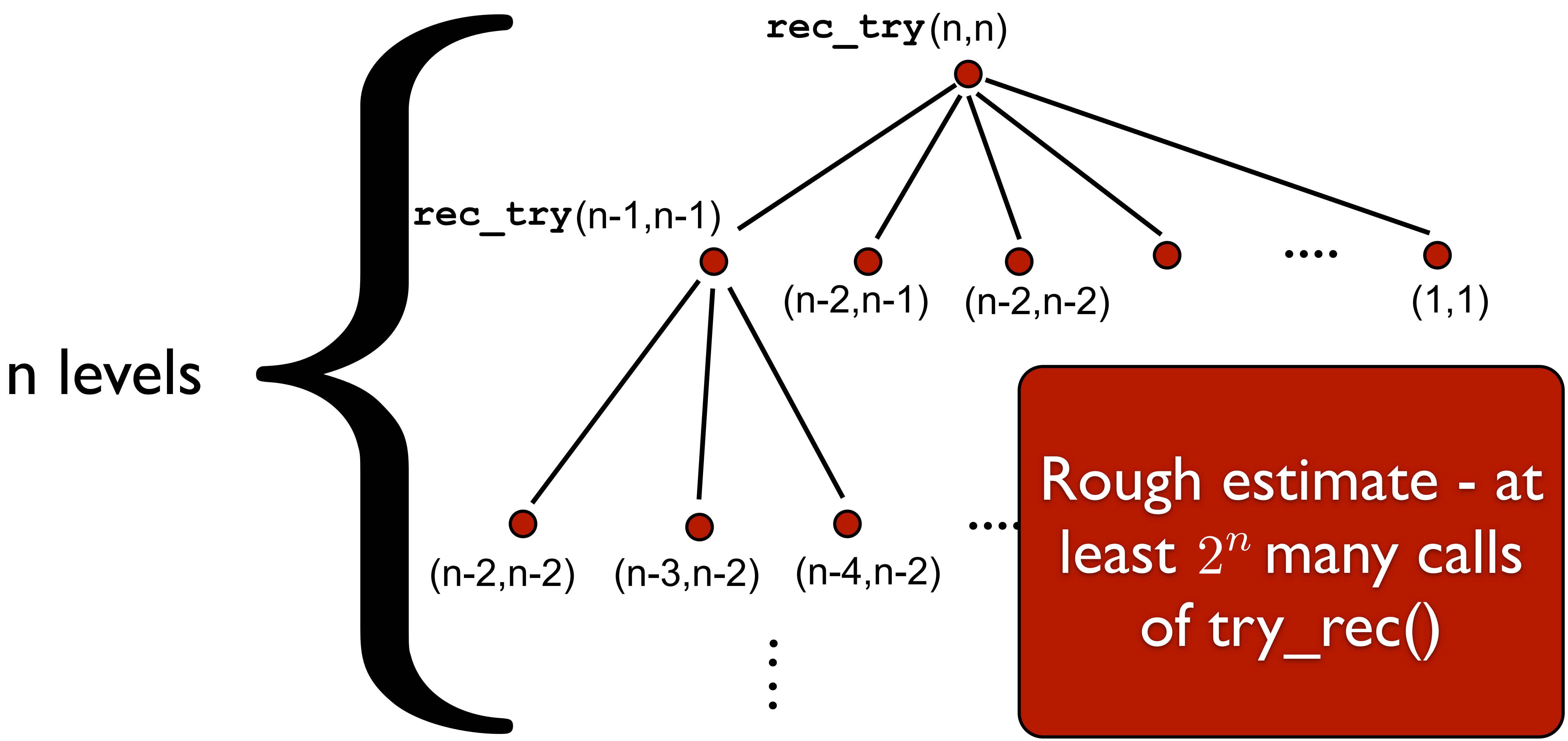
$$\text{cost} = \left(\sum_{t=i-a+1}^{i} A[t]\right) \cdot \left(\sum_{t=j-b+1}^{j} B[t]\right)$$

```
       if (cost + rec_try(i - a, j - b) < best)
           best = cost + rec_try(i - a, j - b)

   return best
```

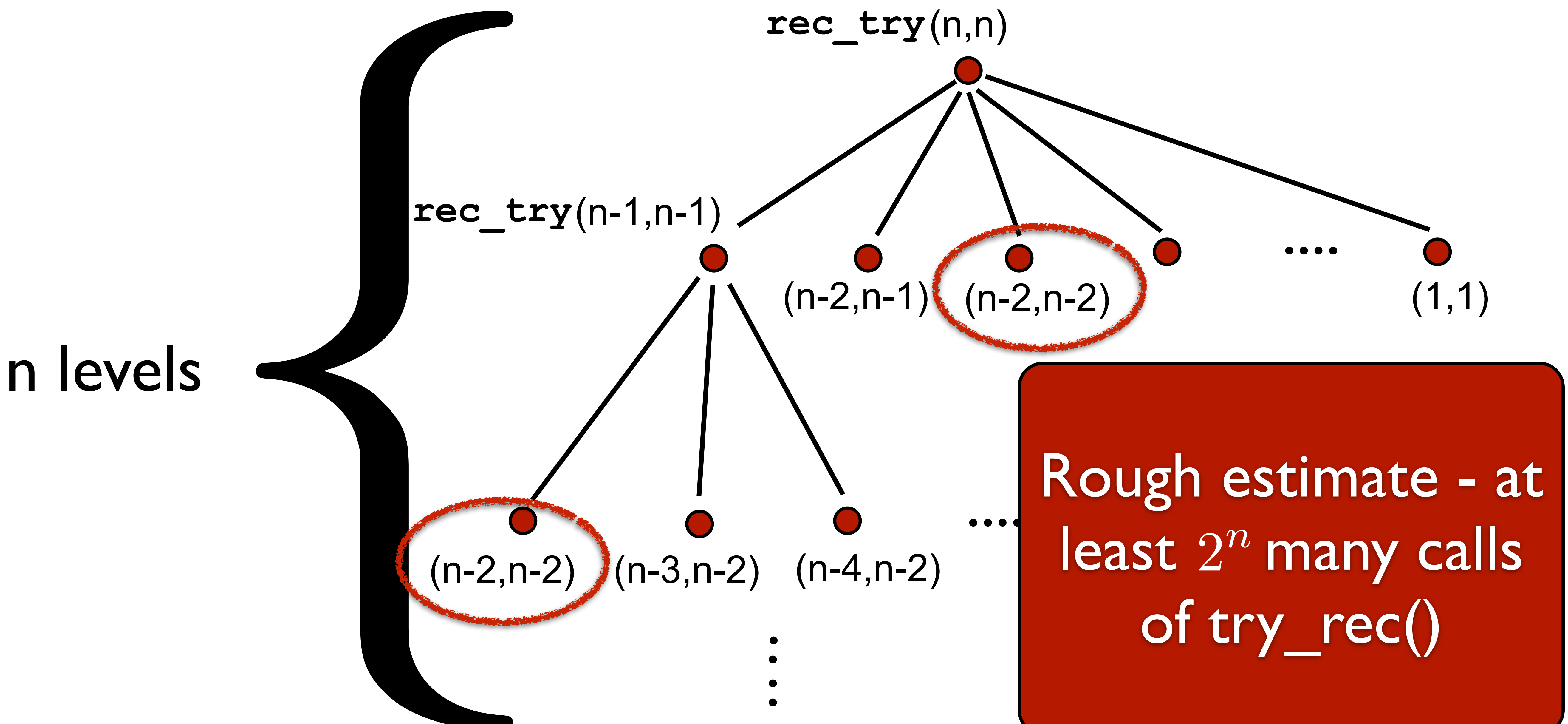
First approach - brute force

Recursively try all possible removals



First approach - brute force

Recursively try all possible removals



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!

What is the running time of try_rec(i,j) ?

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

  if (i == 1) return   A[1]·(B[1] + ... + B[j])

  if (j == 1) return   (A[1] + ... + A[i])·B[1]

  best = (A[1] + ... + A[i])·(B[1] + ... + B[j])  // take all

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

Where the math in the listing is:

$$A[1]\cdot(B[1]+\ldots+B[j])$$

$$(A[1]+\ldots+A[i])\cdot B[1]$$

$$(A[1]+\ldots+A[i])\cdot(B[1]+\ldots+B[j])$$

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

What is the running time of try_rec(i,j) ?

$O(n)$

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

    if (i == 1) return    A[1]·(B[1] + ... + B[j])

    if (j == 1) return    (A[1] + ... + A[i])·B[1]

    best = (A[1] + ... + A[i])·(B[1] + ... + B[j])  // take all

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

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

What is the running time of try_rec(i,j) ?

$O(n)$

$O(n)$

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

    if (i == 1) return   A[1]·(B[1] + ... + B[j])

    if (j == 1) return   (A[1] + ... + A[i])·B[1]

    best = (A[1] + ... + A[i])·(B[1] + ... + B[j])  // take all

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

$$A[1] \cdot (B[1] + \ldots + B[j])$$

$$(A[1] + \ldots + A[i]) \cdot B[1]$$

$$\text{best} = (A[1] + \ldots + A[i]) \cdot (B[1] + \ldots + B[j])$$

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

What is the running time of try_rec(i,j) ?

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

$O(n)$

```
    if (i == 1) return
```
$A[1] \cdot (B[1] + \ldots + B[j])$

```
    if (j == 1) return
```
$(A[1] + \ldots + A[i]) \cdot B[1]$

best = $(A[1] + \ldots + A[i]) \cdot (B[1] + \ldots + B[j])$ `// take all`

$O(n^2)$

```
    for a = 1 to i - 1
      for b = 1 to j - 1
```

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

```
         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
```

What is the running time of try_rec(i,j) ?

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

$O(n)$

if (i == 1) return $A[1]\cdot(B[1]+\ldots+B[j])$

if (j == 1) return $(A[1]+\ldots+A[i])\cdot B[1]$

best = $(A[1]+\ldots+A[i])\cdot(B[1]+\ldots+B[j])$ // take all

$O(n^3)$

for a = 1 to i - 1
  for b = 1 to j - 1

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

```
        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
```

What is the running time of try_rec(i,j) ?

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

$O(n^3)$

```
if (i == 1) return
```
$A[1] \cdot (B[1] + \ldots + B[j])$

```
if (j == 1) return
```
$(A[1] + \ldots + A[i]) \cdot B[1]$

best = $(A[1] + \ldots + A[i]) \cdot (B[1] + \ldots + B[j])$ `// take all`

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

```
    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 - running time

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

Second approach - running time

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

$O(n^3)$ per **try_rec(i,j)**

Second approach - running time

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

$O(n^3)$ per **try_rec(i,j)**

Total running time: $O(n^5)$

This was an example of
Dynamic Programming!

This was an example of
Dynamic Programming!

Let’s improve it!

Cheap improvement.

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

    if (i == 1) return   A[1]·(B[1]+...+B[j])

    if (j == 1) return   (A[1]+...+A[i])·B[1]

    best = (A[1]+...+A[i])·(B[1]+...+B[j])  // take all

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

$$A[1] \cdot (B[1] + \ldots + B[j])$$

$$(A[1] + \ldots + A[i]) \cdot B[1]$$

$$(A[1] + \ldots + A[i]) \cdot (B[1] + \ldots + B[j])$$

$$O(n^3)$$

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

Cheap improvement.

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

...urn $A[1] \cdot (B[1] + \ldots + B[j])$

$(A[1] + \ldots + A[i]) \cdot B[1]$

Can you improve me to $O(n^2)$?

be... $+ A[i]) \cdot (B[1] + \ldots + B[j])$ `// take all`

```
for a = 1 to i - 1
  for b = 1 to j - 1
```

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

```
    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
```

Cheap improve

```
rec_try(i, j)   // cons
                // A an
              urn    A[
```

$(A[1] + \ldots + A[i]) \cdot B[1]$

$+ A[i]) \cdot (B[1] + \ldots + B[j])$ `// take all`

```
    for a = 1 to i - 1
      for b = 1 to j - 1
```

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

```
        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
```

Yes!

Calculating `cost` can be done in constant time.

(Hint: Precomputing partial sums)

Can you improve me to $O(n^2)$?

From exponential to $O(n^5)$ and then to $O(n^4)$.
Not bad!

From exponential to $O(n^5)$ and then to $O(n^4)$.
Not bad!

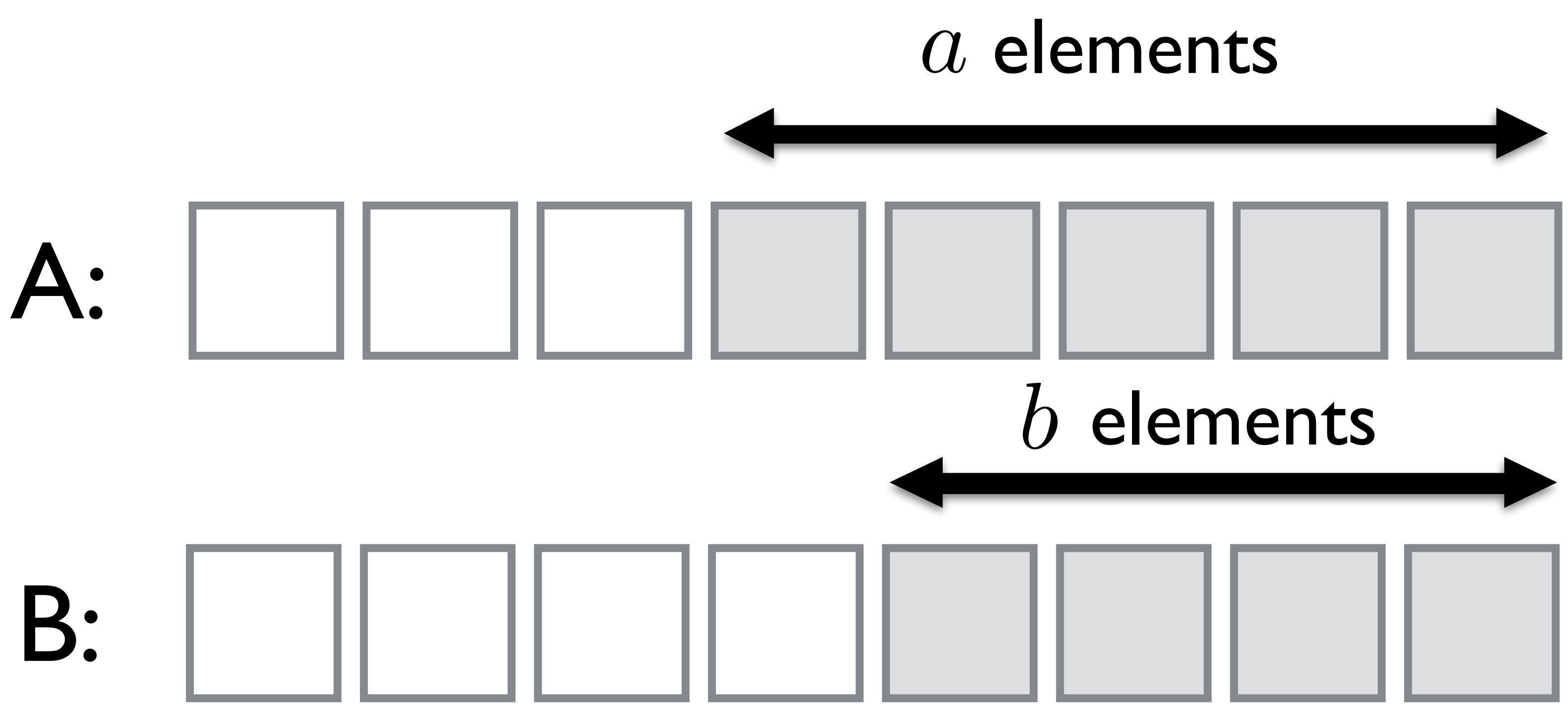
But we can do even better!

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

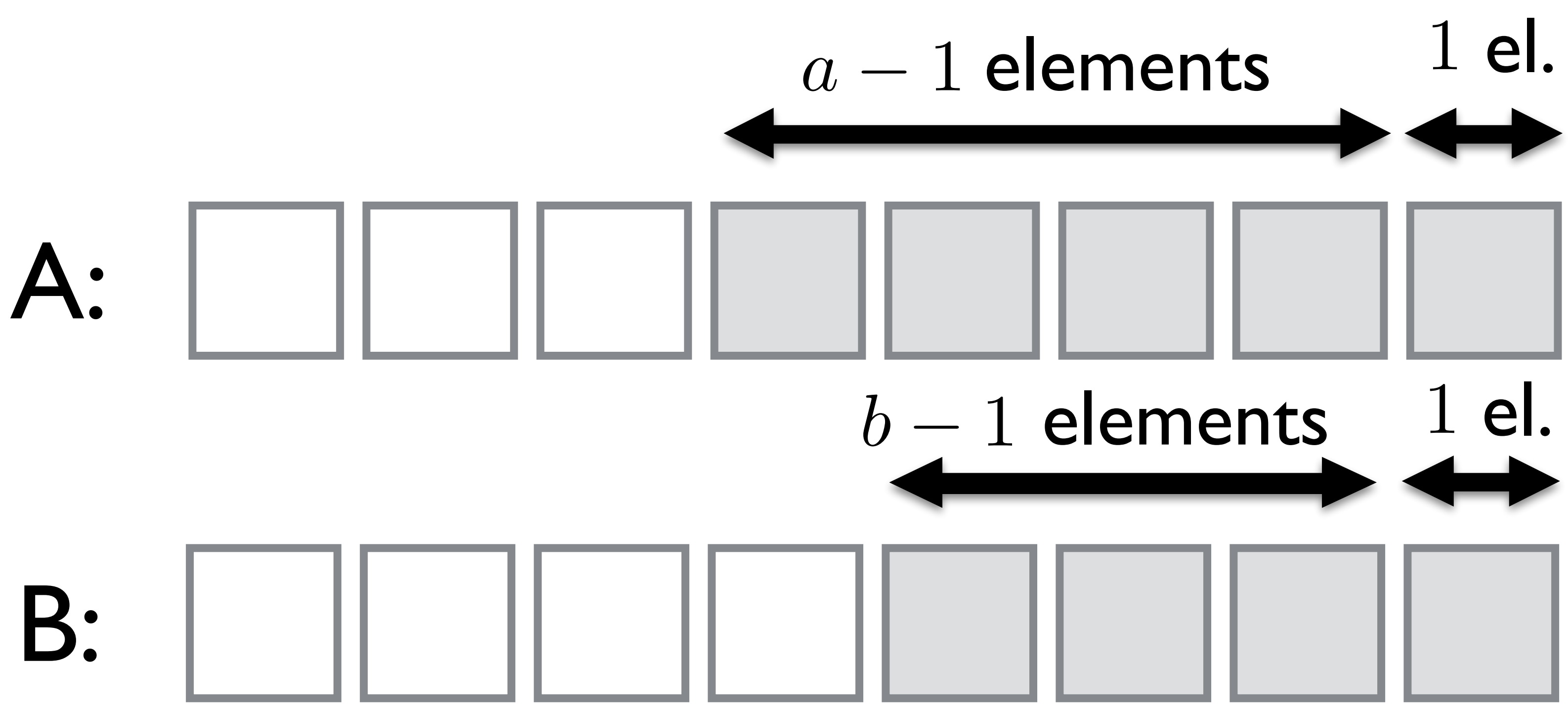


Cost 1: $(A[1] + A[2] + \cdots + A[a]) \cdot (B[1] + B[2] + \cdots + B[b])$

* The order of elements in A/B is reversed here, for the convenience of presentation.

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely



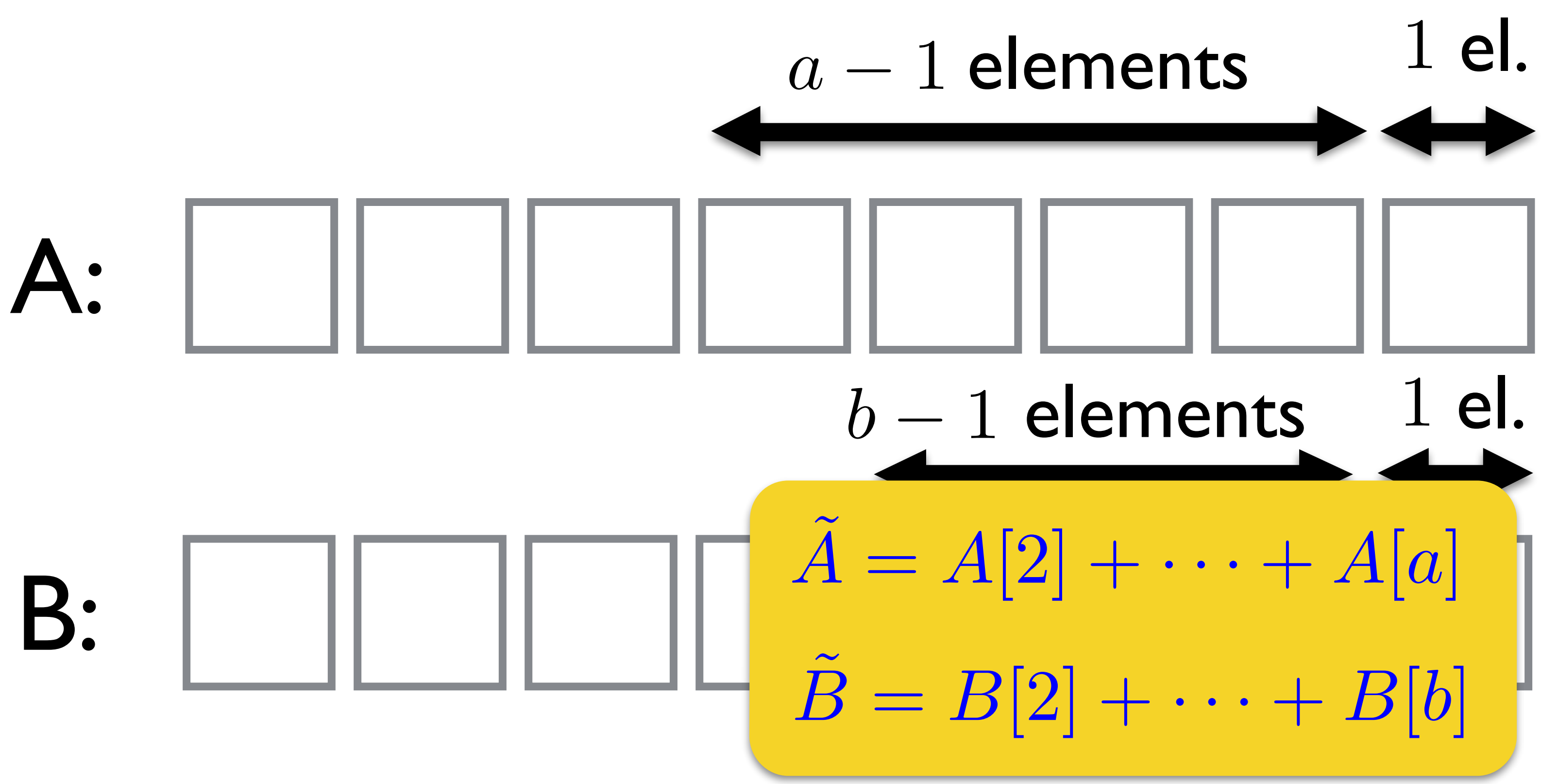
Cost 1: $(A[1]+A[2]+\cdots+A[a])\cdot(B[1]+B[2]+\cdots+B[b])$

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

* The order of elements in A/B is reversed here, for the convenience of presentation.

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely



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

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

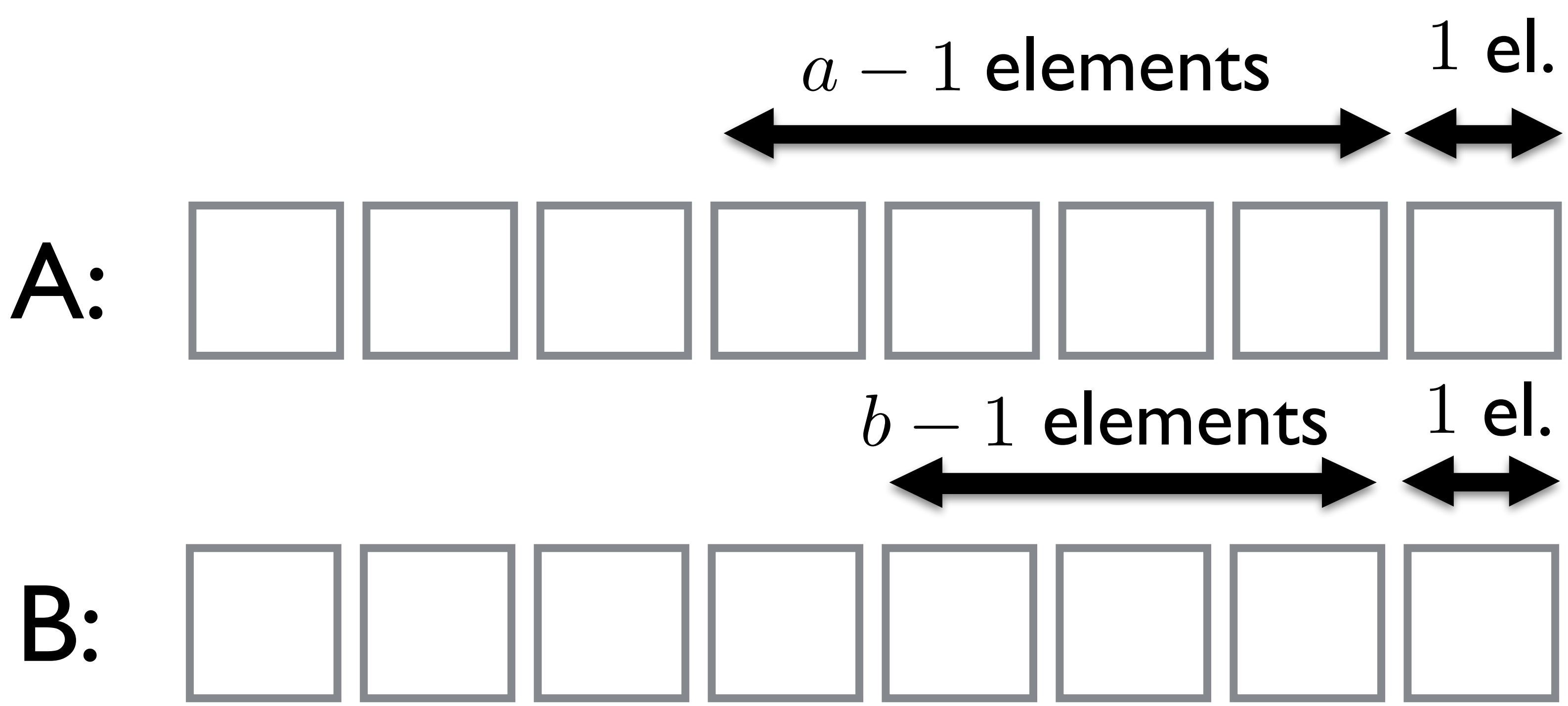
Cost 1: $(A[1] + A[2] + \cdots + A[a]) \cdot (B[1] + B[2] + \cdots + B[b])$

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

* The order of elements in A/B is reversed here, for the convenience of presentation.

Can we improve the running time of the previous algorithm?

Let us observe the cost function more closely



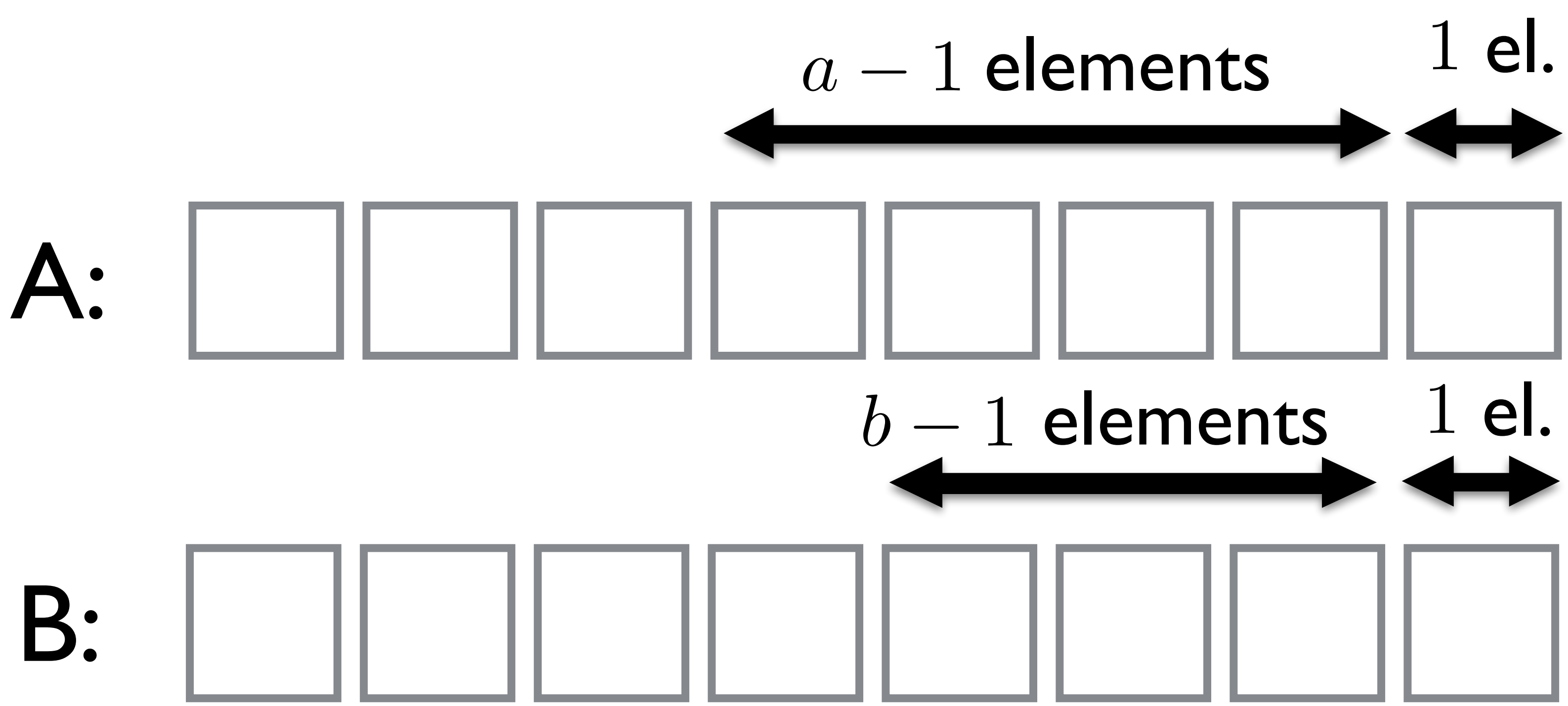
Cost 1: $(A[1]+\tilde{A})(B[1]+\tilde{B})$

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

* The order of elements in A/B is reversed here, for the convenience of presentation.

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

$$A[1]B[1] + \tilde{A}\tilde{B} + A[1]\tilde{B} + \tilde{A}B[1] \geq A[1]B[1] + \tilde{A}\tilde{B}$$

Cost 1 Cost 2

B:

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?

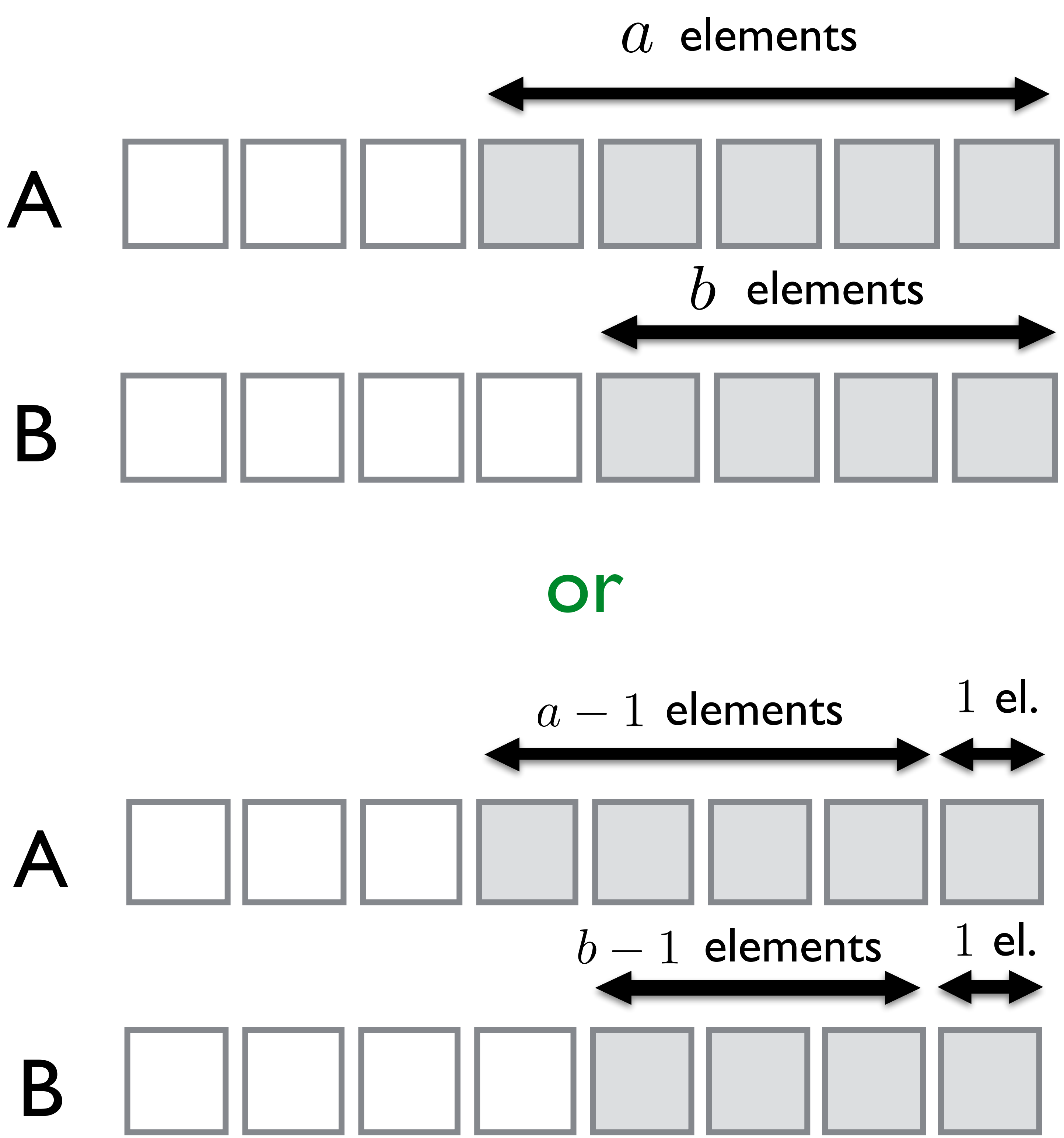
Let us observe the cost function more closely

Conclusion?

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

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

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



This only makes sense when $a > 1$ and $b > 1$

The optimal strategy will either:

1. Take one element from both sides (our analysis), or

The optimal strategy will either:

1. Take one element from both sides (our analysis), or
2. Take one element from A ($a = 1$) and more elements from B (or vice versa)

The optimal strategy will either:

1. Take one element from both sides (our analysis), or
2. Take one element from A ($a = 1$) and more elements from B (or vice versa)

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]·(B[1]+...+B[j])

   if (j == 1) return  (A[1]+...+A[i])·B[1]

   best = (A[1]+...+A[i])·(B[1]+...+B[j]) // take all

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

Where:

$A[1] \cdot (B[1] + \ldots + B[j])$

$(A[1] + \ldots + A[i]) \cdot B[1]$

$(A[1] + \ldots + A[i]) \cdot (B[1] + \ldots + B[j])$

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

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