Divide & Conquer 2

Ken Yasuhara is here to collect your feedback for me.

TAs and I will be in the waiting room (unable to hear anything)

Ken is recording to his local computer (I won’t have access to the recording).

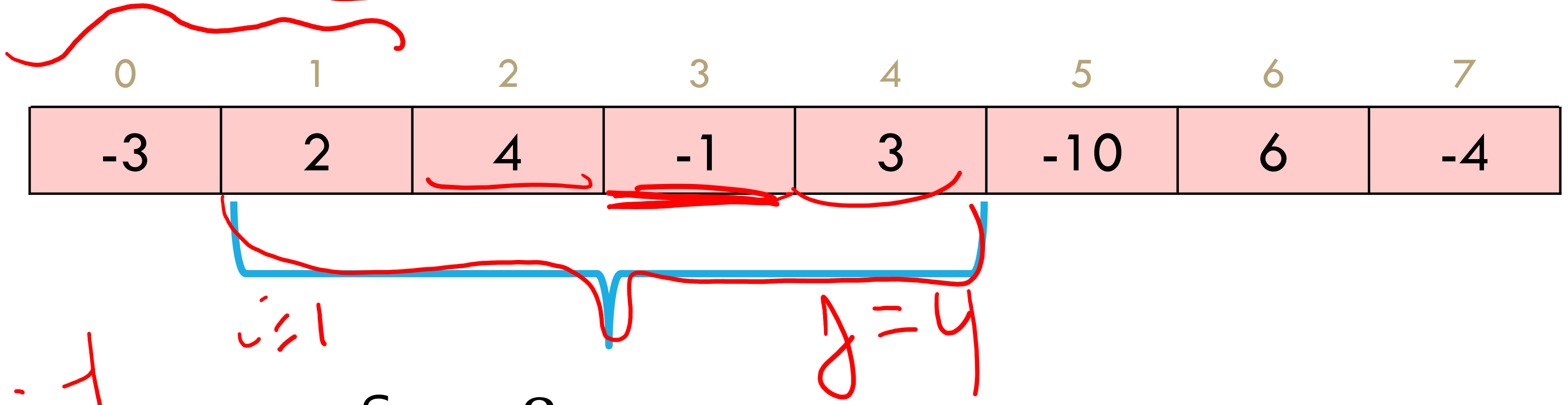
Asynchronous? You’ll get a way for your feedback to be incorporated later this week.

Another divide and conquer

Maximum subarray sum

Given: an array of integers (positive and negative), find the indices that give the maximum contiguous subarray sum.

i, j



i = 0, j=1

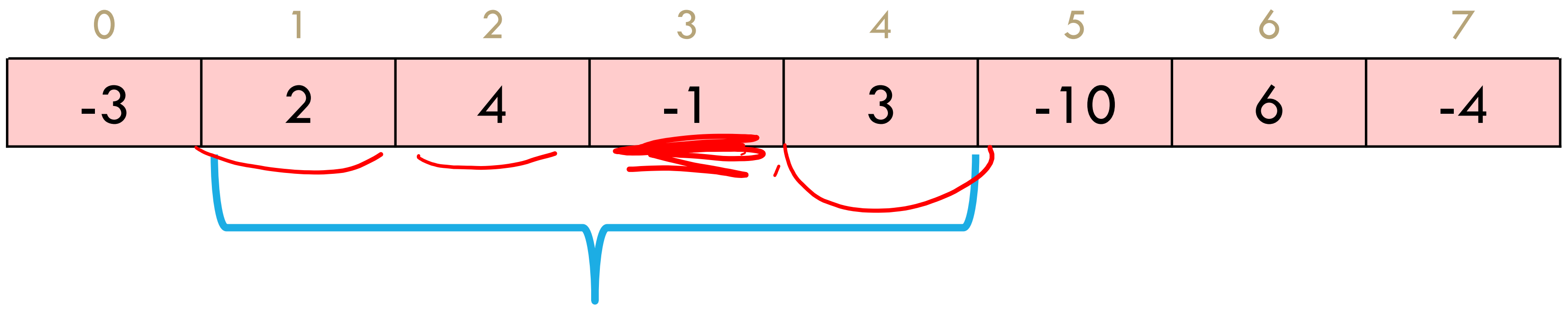
A[0]+A[1] = -1

Sum: 8

Another divide and conquer

Notice, greedy doesn't work! (at least not the first one you'd think of)

"add if it increases the sum" wouldn't let you add in 3 (because you'd decide not to include -1).



Sum: 8

Edge Cases

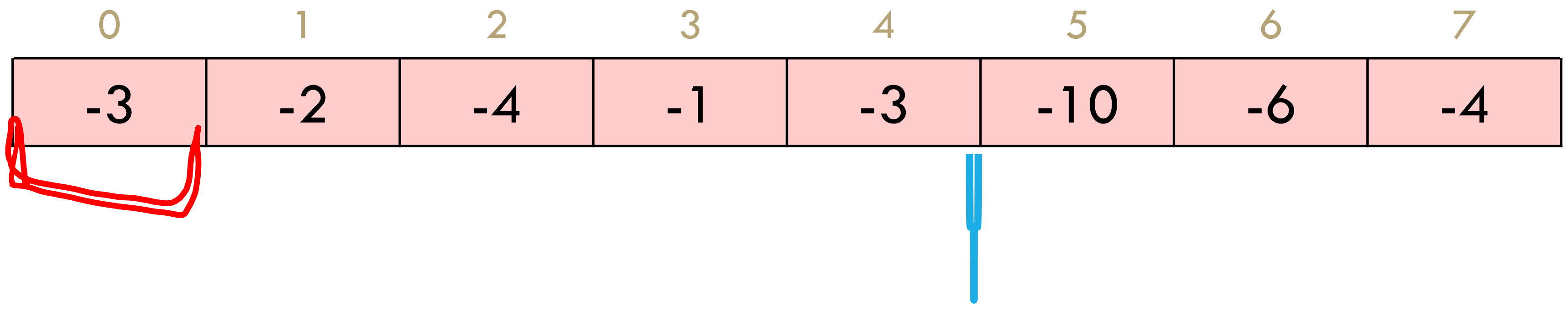
We'll allow for $i = j$. In that case, $A[i]$ is the sum.

We'll also allow for $j < i$. In that case the sum is 0.

This is the best option if and only if all the entries are negative.

$i=0, j=0$

Sum: -3



Sum: 0

Maximum Subarray Sum

i, j

sum

Brute force: How many subarrays to check? $\Theta(n^2)$

How long does it take to check a subarray?

If you keep track of partial sums, the overall algorithm can take $\Theta(n^2)$ time.

$(1, 5) \rightsquigarrow (1, 6)$

(If you calculated from scratch every time it would take $\Theta(n^3)$ time)

Can we do better?

Maximum Contiguous Subarray

1. Divide instance into subparts.
2. Solve the parts recursively.
3. Conquer by combining the answers

1. Split the array in half
2. Solve the parts recursively.
3. Just take the max?

Conquer

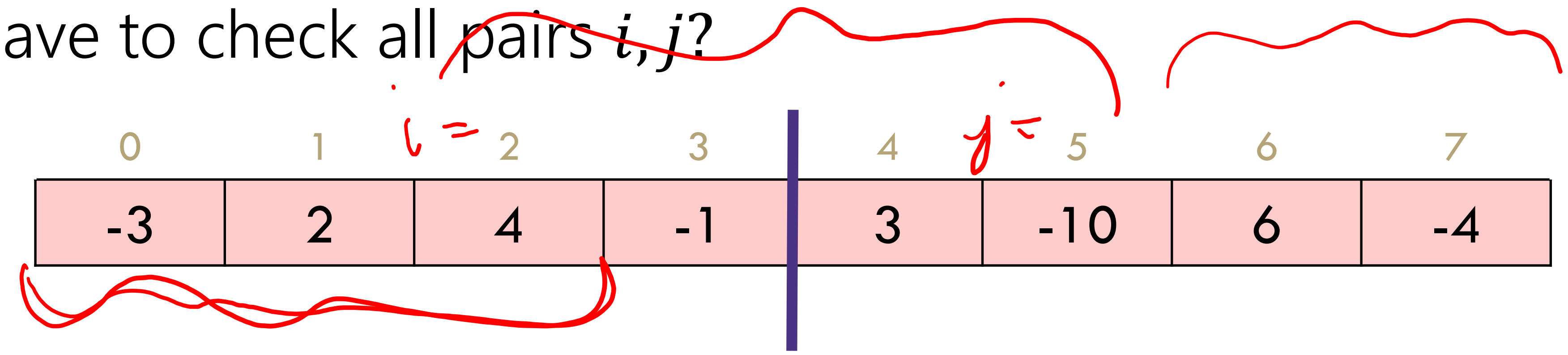
If the optimal subarray: i, j

is only on the left – handled by the first recursive call.

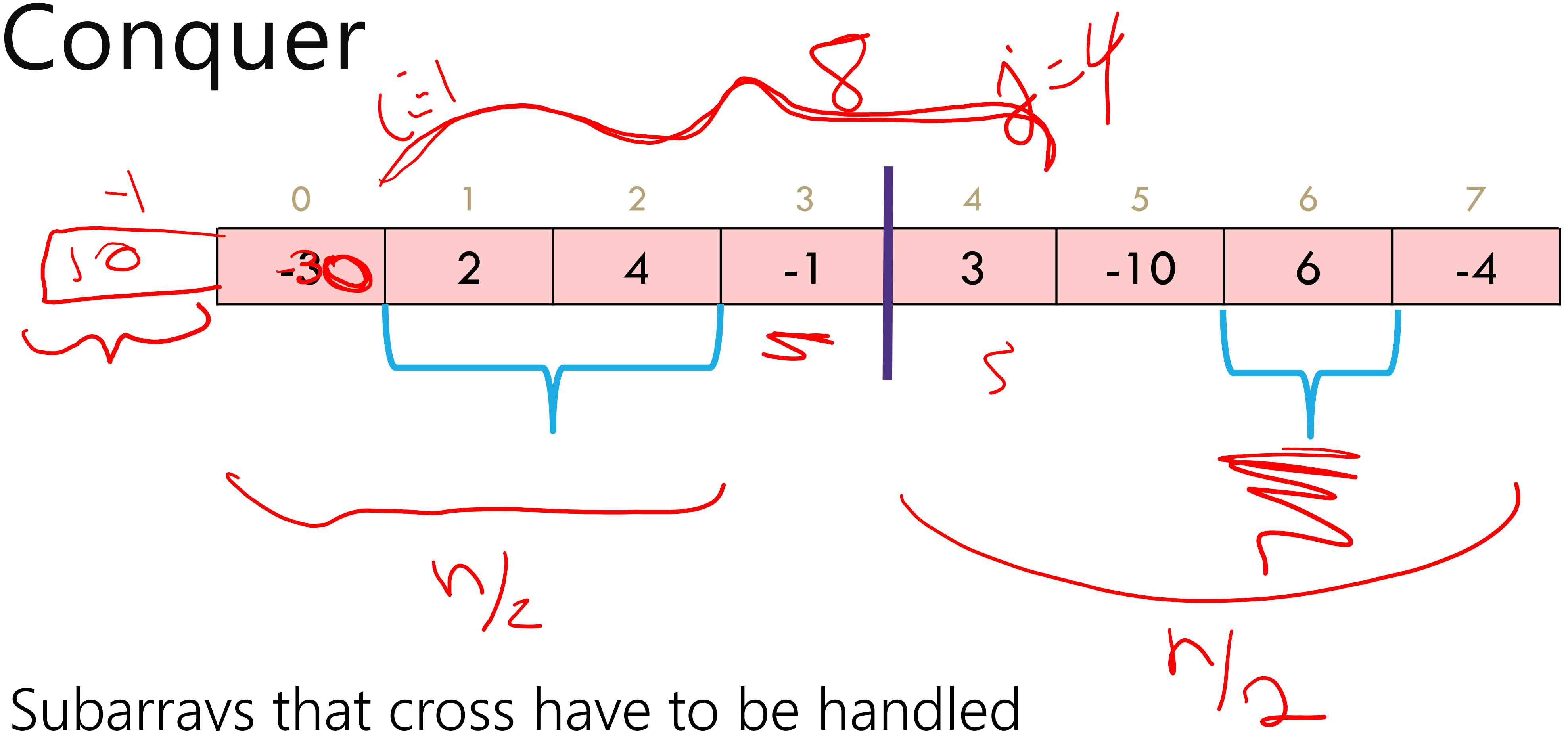
is only on the right – handled by the second recursive call.

crosses the middle – TODO

Do we have to check all pairs i, j?



Conquer



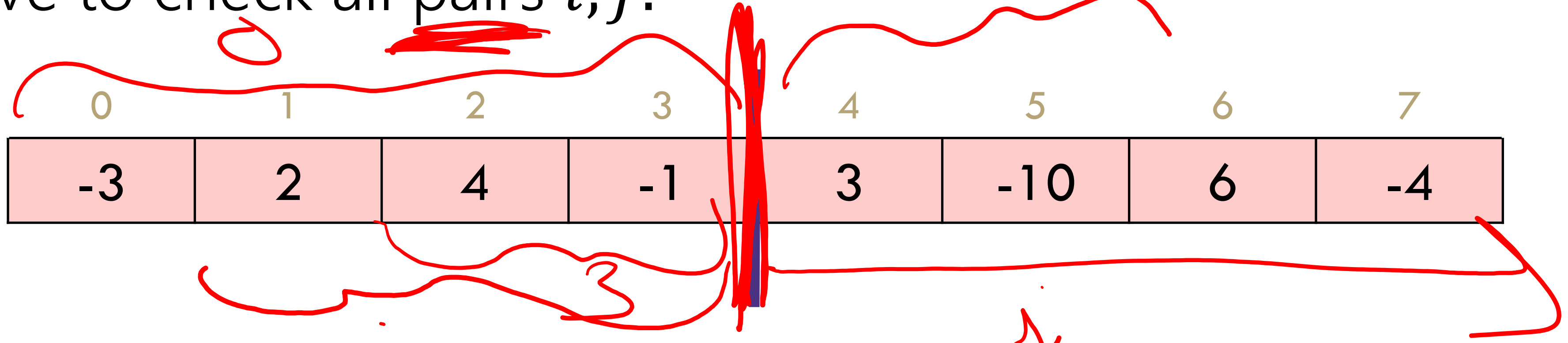
Subarrays that cross have to be handled

Do we have to check all pairs i, j?

$\frac{n^2}{4}$

Crossing Subarrays

Do we have to check all pairs i, j?



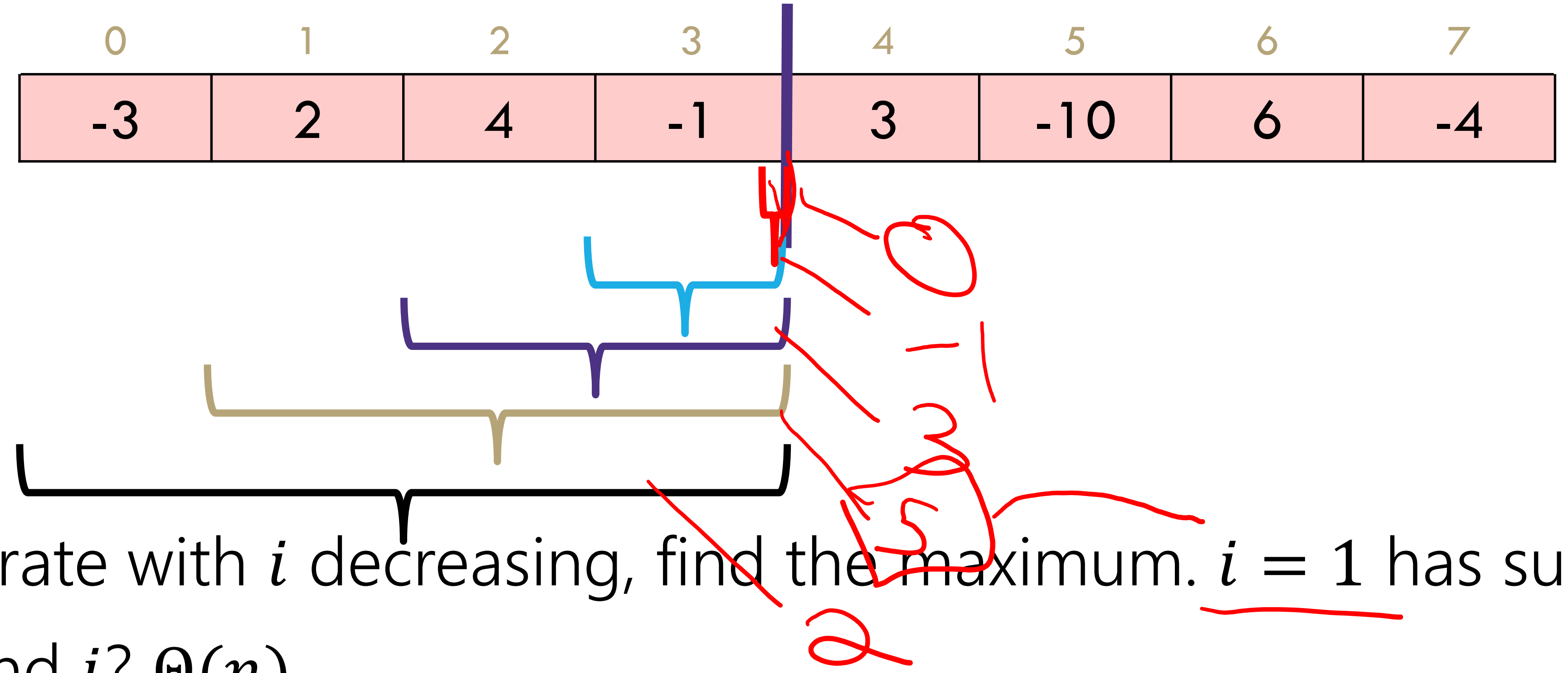
Sum is $A\left[\frac{n}{2}-1\right]+A\left[\frac{n}{2}-2\right]+\cdots+A[i]+A\left[\frac{n}{2}\right]+A\left[\frac{n}{2}+1\right]+\cdots A[j]$

i, j affect the sum. But they don't affect each other.

Calculate them **separately!**

Crossing Subarrays

Do we have to check all pairs i, j?

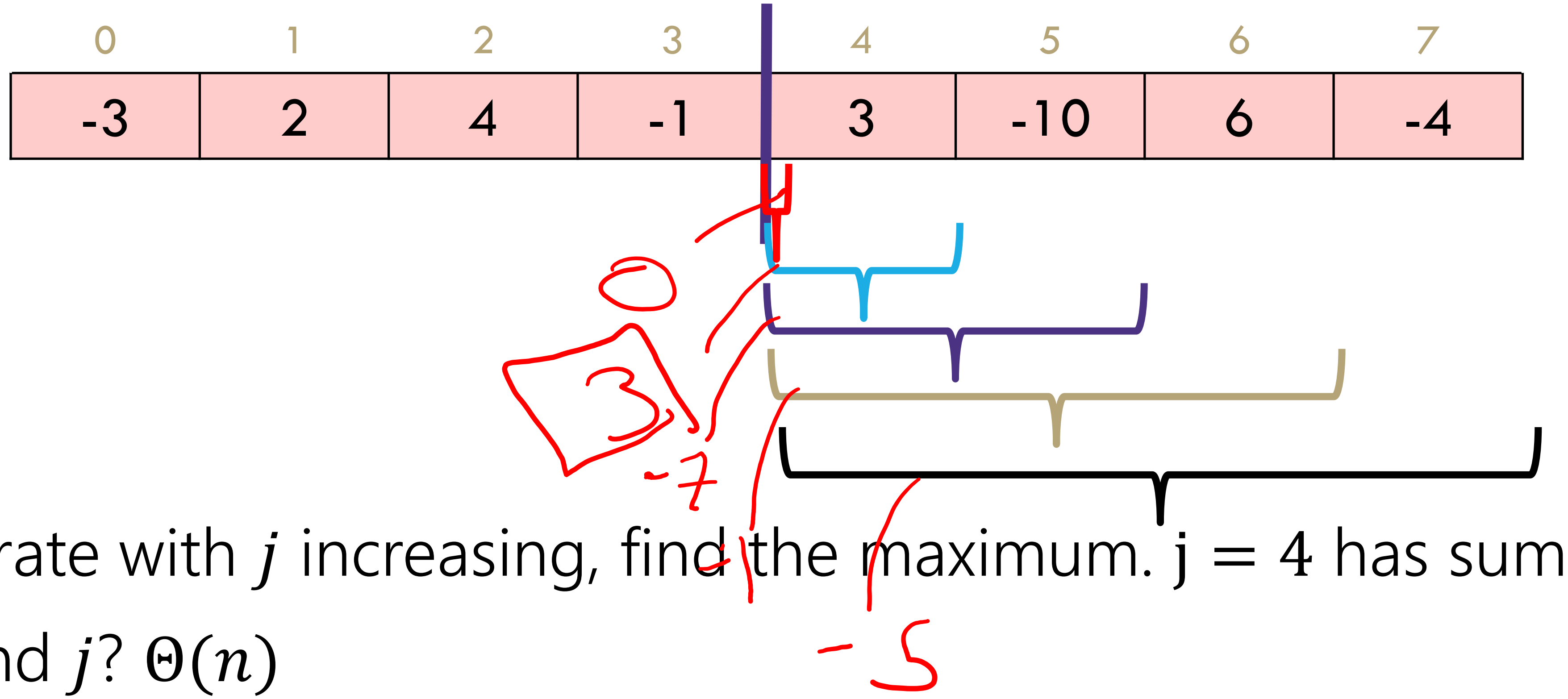


Best i? Iterate with i decreasing, find the maximum. $i = 1$ has sum 5.

Time to find i? $\Theta(n)$

Crossing Subarrays

Do we have to check all pairs i, j?

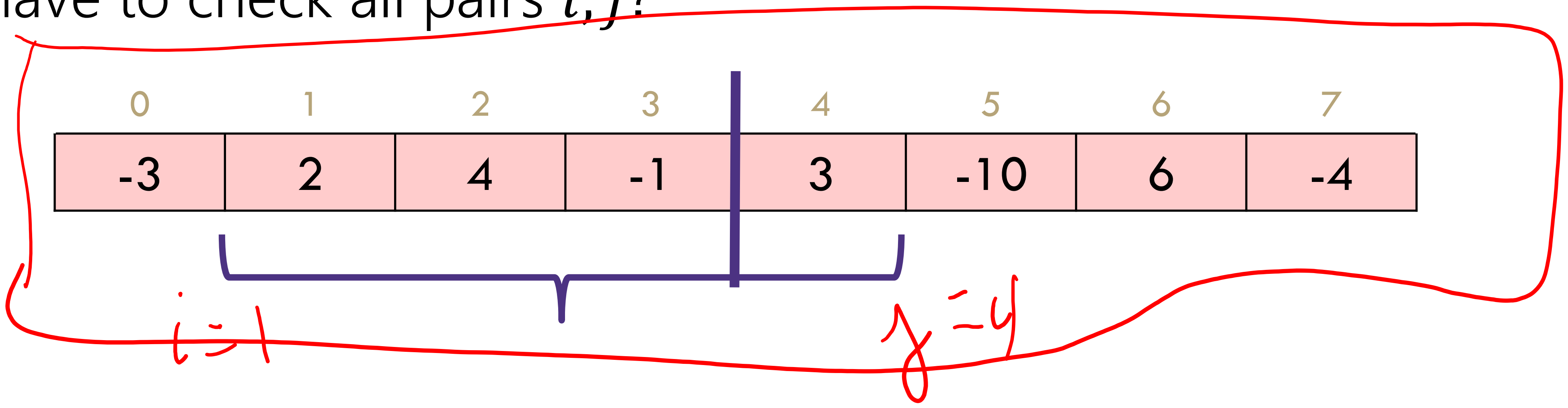


Best j? Iterate with j increasing, find the maximum. j = 4 has sum 3.

Time to find j? $\Theta(n)$

Crossing Subarrays

Do we have to check all pairs i, j?



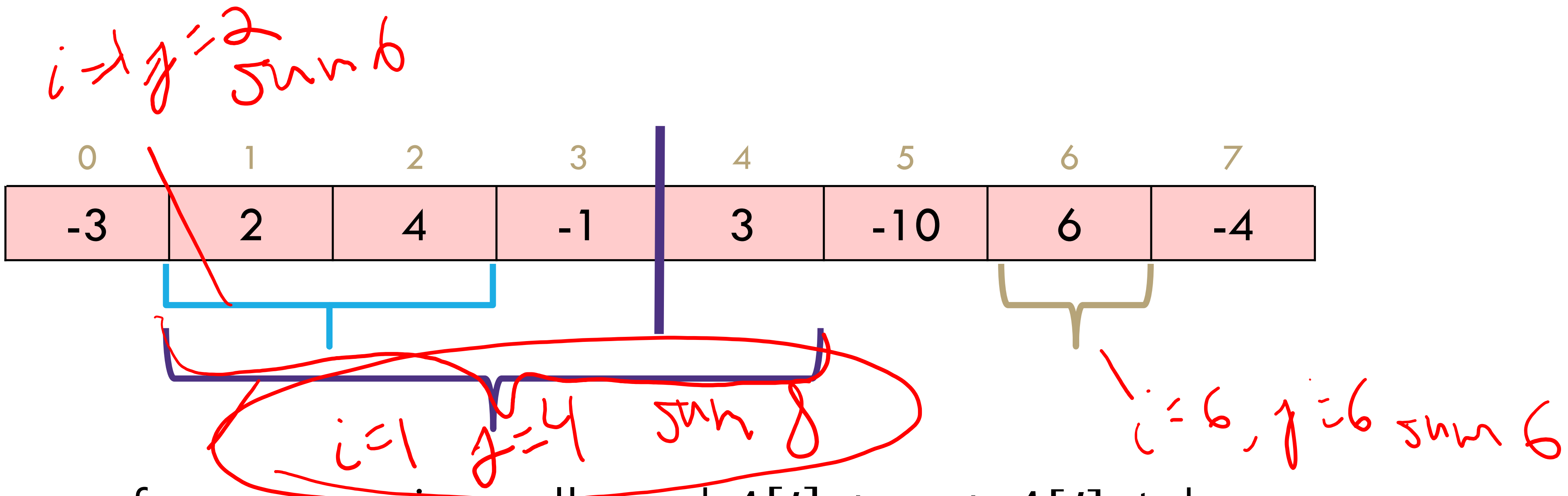
Sum: 5 + 3 = 8

Best crossing array From the i you found to the j you found.

Time to find? $\Theta(n)$ total.

Finishing the recursive call

Overall:



Compare sums from recursive calls and $A[i] + \cdots + A[j]$, take max.

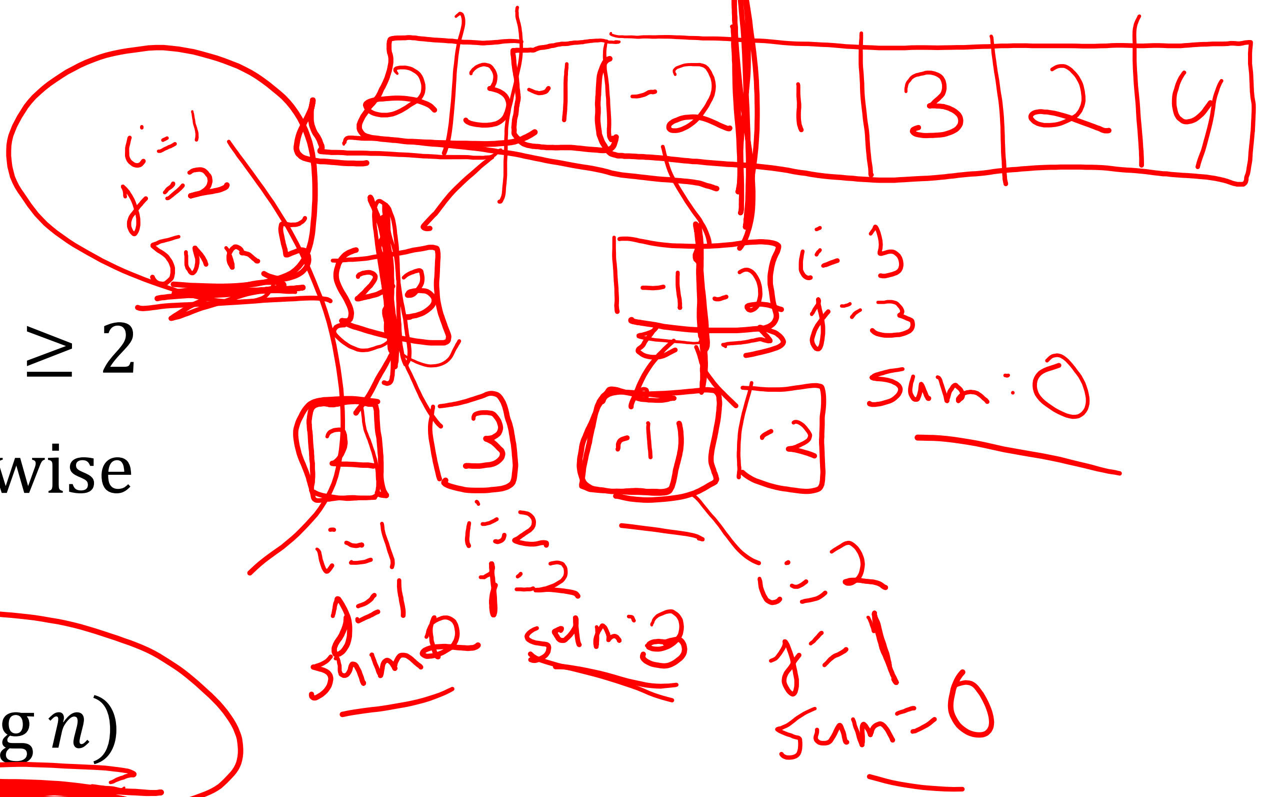
Running Time

What does the conquer step do?

Two separate $\underline{\Theta(n)}$ loops, then a constant time comparison.

So

$$T(n) = \begin{cases} \underline{2}T\left(\underline{\frac{n}{2}}\right) + \Theta(n) & \text{if } n \geq 2 \\ \Theta(1) & \text{otherwise} \end{cases}$$



Master Theorem says $\underline{\Theta(n \log n)}$

One More problem

We want to calculate $a^n \% b$ for a very big n. What do you do?

"Naïve" algorithm" – for-loop: multiply a running product by a and modding by b in each iteration.

> "Naïve" is computer-scientist-speak for "first algorithm you'd think of."
> Thinking of it **doesn't** mean you're naïve. It means you know your fundamentals. And no one told you the secret magic speedup trick.

Time? $O(n)$

A better plan

What's about half of the work?

$\underline{a^{n/2}}\%b$ is about half.

```
DivConqExponentiation(int a, int b, int n)
   if(n==0) return 1
   if(n==1) return a%b
   int k = divConqExponentiation(a,b,n/2) /*int div*/
   if(n%2==1) return (k*k*a)%b
   return (k*k)%b
```

$\left\lfloor \frac{n}{2} \right\rfloor$

Activity

Write a recurrence to describe the running time of this code. What's the big-O?

```
DivConqExponentiation(int a, int b, int n)
  if(n==0) return 1
  if(n==1) return a%b
  int k = divConqExponentiation(a,b,n/2) /*int div*/
  if(n%2==1) return (k*k*a)%b
  return (k*k)%b
```

Fill out the poll everywhere for Activity Credit!
Go to pollev.com/cse417 and login with your UW identity

Running Time?

$2T\left(\frac{n}{2}\right)$

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n \geq 2 \\ \Theta(1) & \text{otherwise} \end{cases}$$

Which is $\underline{\Theta(\log n)}$ time.

Much faster than $O(n)$.

Fun Fact: RSA encryption (one of the schemes most used for internet security) needs exactly this sort of "raise to a large power, mod b" operation.

When to use Divide & Conquer

Your problem has a structure that makes it easy to split up

Usually in half, but not always…

You can split "what you're looking for" (subarrays, inversions, etc.) into "just in one of the parts" and "split across parts"

The $\underline{\text{"split"}}$ ones can be studied faster than brute force.

Classic Divide & Conquer

Classic Applications

There are a few really famous divide & conquer algorithms where the way to recurse is **very clever**.

The point of the next few examples is **not** to teach you really useful tricks (they often don't generalize to new problems).

It's partially to show you that sometimes being really clever gives you a really big improvement

And partially because these are "standard" in algorithms classes, so you should at least have heard of these algorithms.

Classic Application

Suppose you need to multiply **really** big numbers.

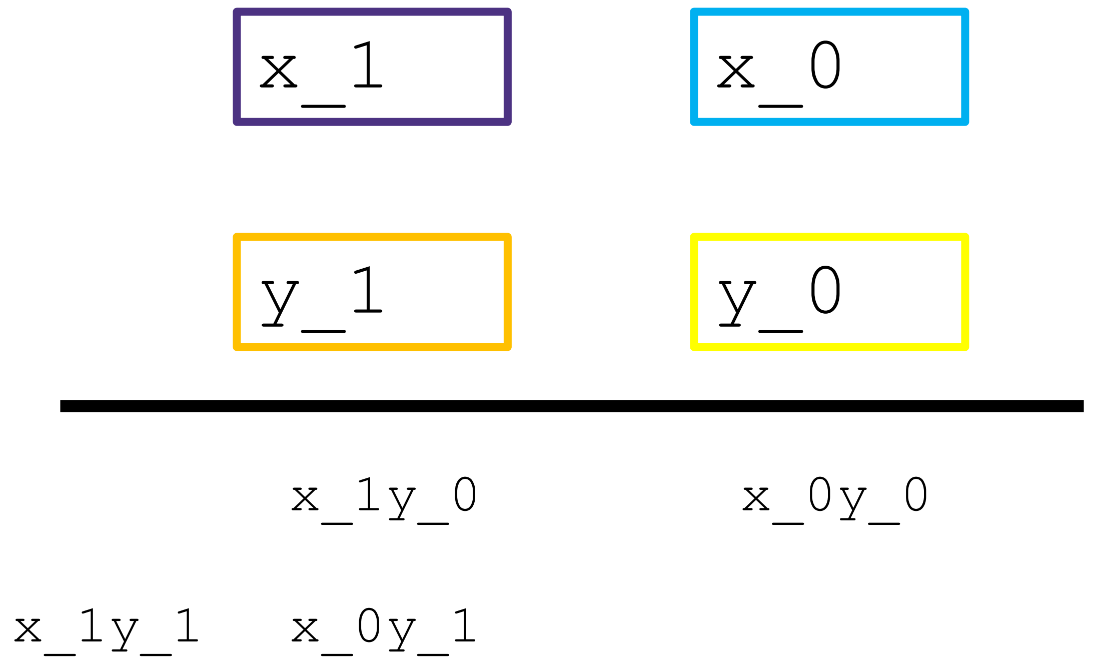
Much bigger than `ints`

Split the n bit numbers in half

Think of them as written in base $2^{n/2}$

What would the “normal” multiplication algorithm do?

4 multiplications, i.e. 4 recursive calls.

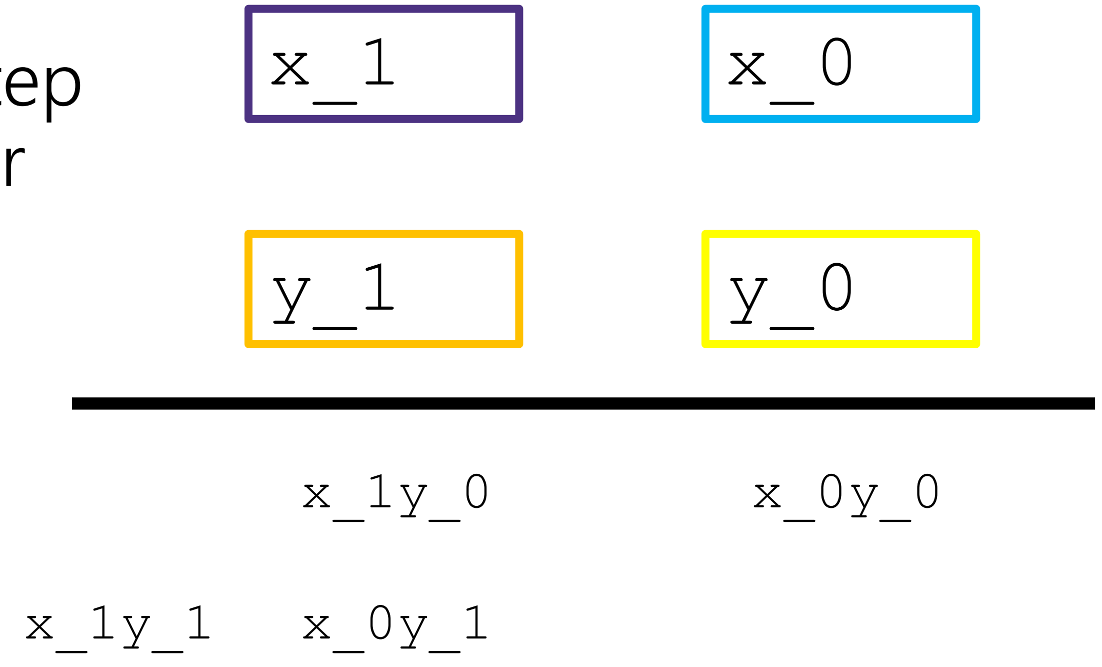


Classic Application

If n bits is too many to multiply in one step (e.g. it's more than one byte, or whatever your processor does in one cycle)

Recurse! Running time?

$$T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + O(n) \text{ if } n \text{ is large} \\ O(1) \quad \text{ if } n \text{ fits in one byte} \end{cases}$$



Why $O(n)$? It takes $O(n)$ time to add up $O(n)$ bit numbers – they have $O(n)$ bytes!
(Why have you never seen this before? We assumed our numbers were ints, where the number of bytes is a constant)

Overall running time is $\Theta(n^2)$

Clever Trick

We need to find x_1y_0 + x_0y_1.

Does that look familiar? It’s the middle to terms when you FOIL

Define $\alpha = (x_0 + x_1), \beta = (y_0 + y_1)$

$\alpha \cdot \beta = x_0 y_0 + x_1 y_0 + x_0 y_1 + x_1 y_1$

So $\alpha\beta - x_0 y_0 - x_1 y_1 = x_1 y_0 + x_0 y_1$

What do we need to find the overall multiplication?

$x_0 y_0 + (\alpha\beta - x_0 y_0 - x_1 y_1) \cdot 2^{\frac{n}{2}} + x_1 y_1 \cdot 2^n$

$x_0 y_0$, $x_1 y_1$ and $\alpha\beta$ are enough to calculate the overall answer! Only 3 multiplies of n/2 bits!

Running Time

$$T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + O(n) \text{ if } n \text{ is large} \\ O(1) \quad\quad \text{if } n \text{ fits in one byte} \end{cases}$$

$\log_2(3) > 1$,

So running time is $O\left(n^{\log_2(3)}\right)$

Or about $O\left(n^{1.585}\right)$

Strassen’s Algorithm

Apply that “save a multiplication” idea to multiplying matrices, and you can also get a speedup.

Called Strassen’s Algorithm

Instead of $O(n^3)$ time, can get down to $O\left(n^{\log_2 7}\right) \approx O\left(n^{2.81}\right)$.

Lots of more clever ideas have gotten matrix multiplication even faster **theoretically**. In practice, the constant factors are too high.