Divide & Conquer 2

CSE 417 Winter 24
Lecture 9

Announcements

HW1 grades are back

Gradescope doesn't support E/S/N/U directly, so:
E will be represented as 3, S as 2, N as 1, U as 0.

Don't "add up" problems and get a percentage. That percentage doesn't mean anything! We're counting number of E's and S's and using the tables in the syllabus.

Be sure to read the individual feedback the TAs wrote for you.

Consider typesetting (word or LaTeX). If we can't read your submission easily it won't be graded.

Announcements

- Boxes also available for the resubmit of old written problems – when you submit, please also fill out the google form(s) on the assignments page.
  - To resubmit HW1, problem 3 (for example), you **don't** put it in the old HW1 P3 gradescope box; that one is closed anyway!
- For programming questions, you can upload a new file to test anytime (upload doesn't use a resubmission).
- When you're happy with your score, use a resubmission slot on it and we'll record the new score.
- If we made an error in grading, that's when to use gradescope's regrade requests. We don't release solutions (since you can resubmit those problems) but feel free to ask questions at office hours.
  - Regrade requests open 24 hours after grades are released; remain open for one week.

Announcements

The grading system is different…

You'll have to adjust "how you think about" feedback.

An 'S' isn't a ("2 out of 3" 67%) it's "counts for a bunch, but there's some edge case or analysis to fix to be completely right.

An 'N' doesn't count for much of anything in the gradebook, but…

- If you got an N, you've probably still made substantial progress, you've thought about the problem, and it should be easier now than if you saw the problem for the first time.
- We are usually "more helpful" with hints/explanations in office hours after the initial due date.

Today

A Divide & Conquer Algorithm that we'll design together

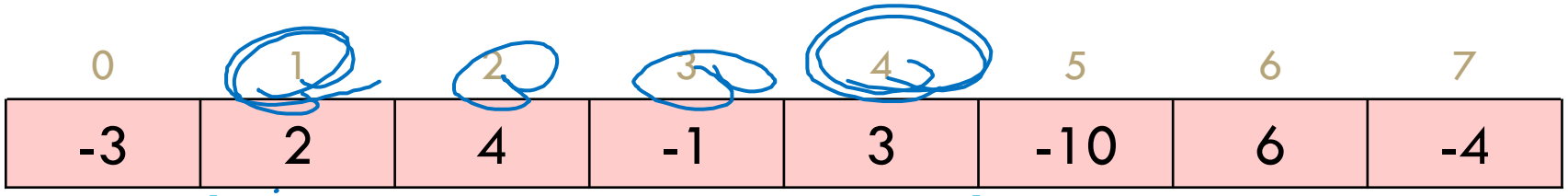
Two "classical" D&C results

Another divide and conquer

Maximum contiguous subarray sum

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

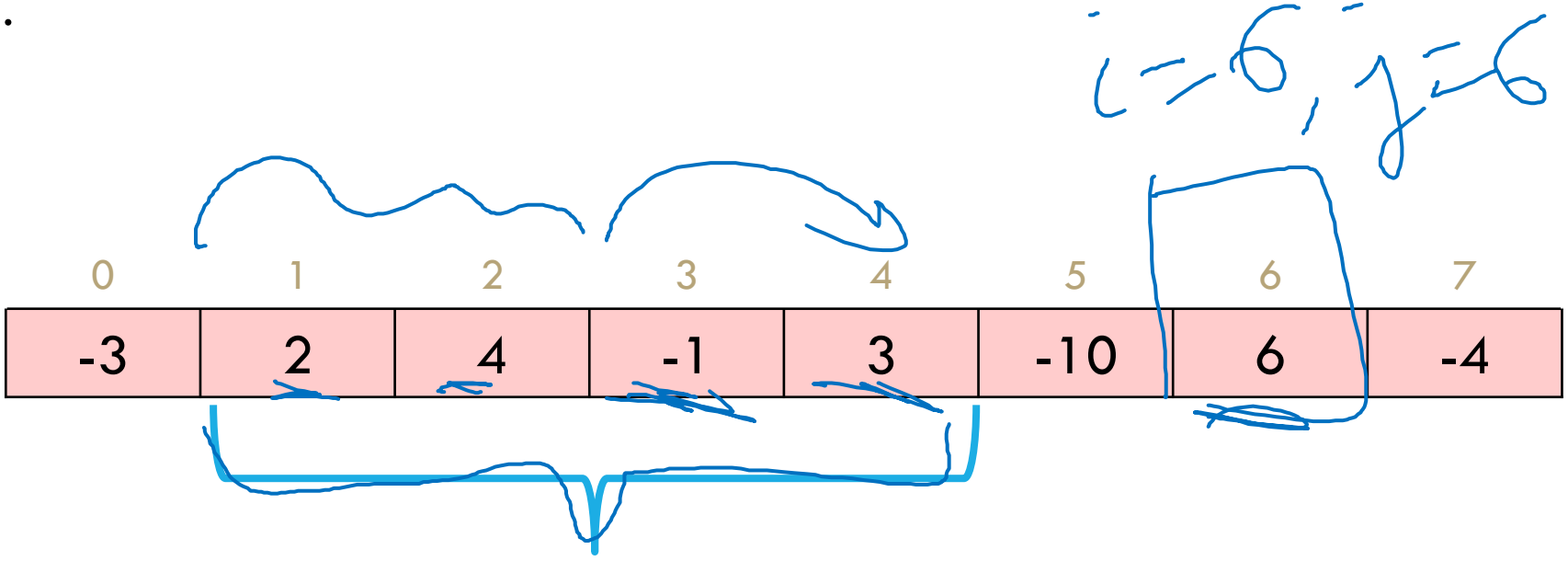
Find i, j that maximize $A[i] + A[i+1] + \cdots + A[j]$



Sum: 8

Another divide and conquer

Notice, it's sometimes a good idea to include negative numbers! They might let you access larger positive numbers and give you a net increase.



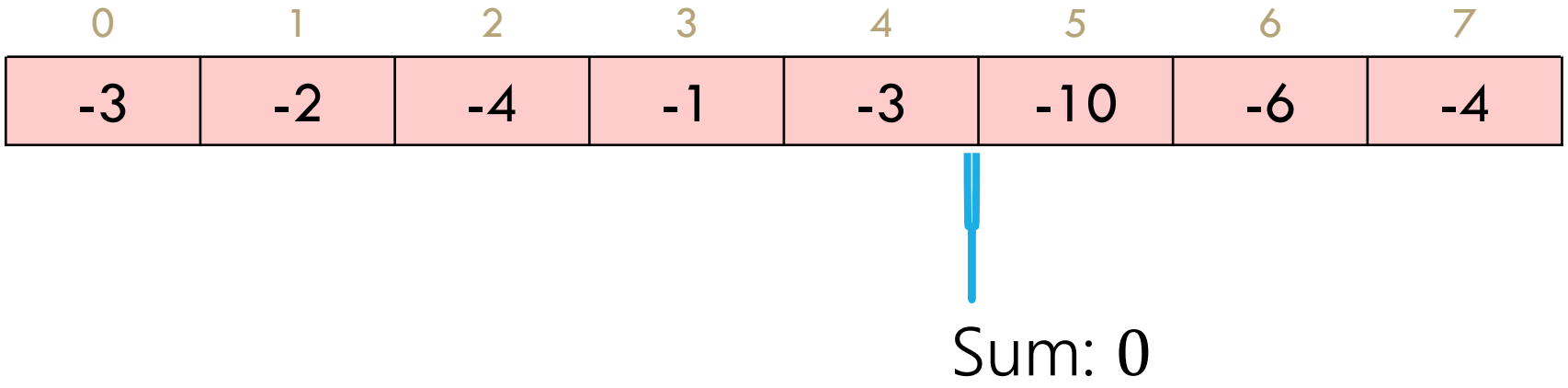
Sum: 8

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.



Maximum Contiguous Subarray 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.

(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

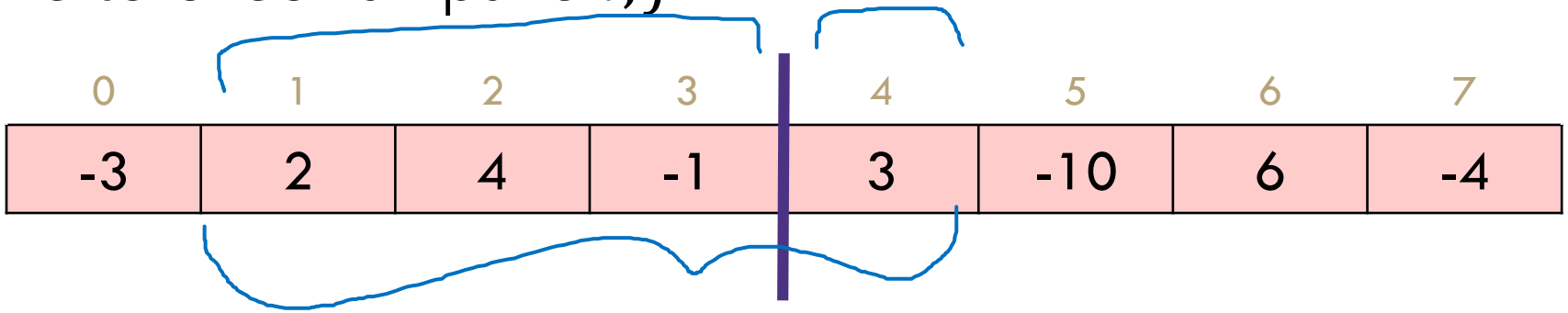
If the optimal subarray:

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

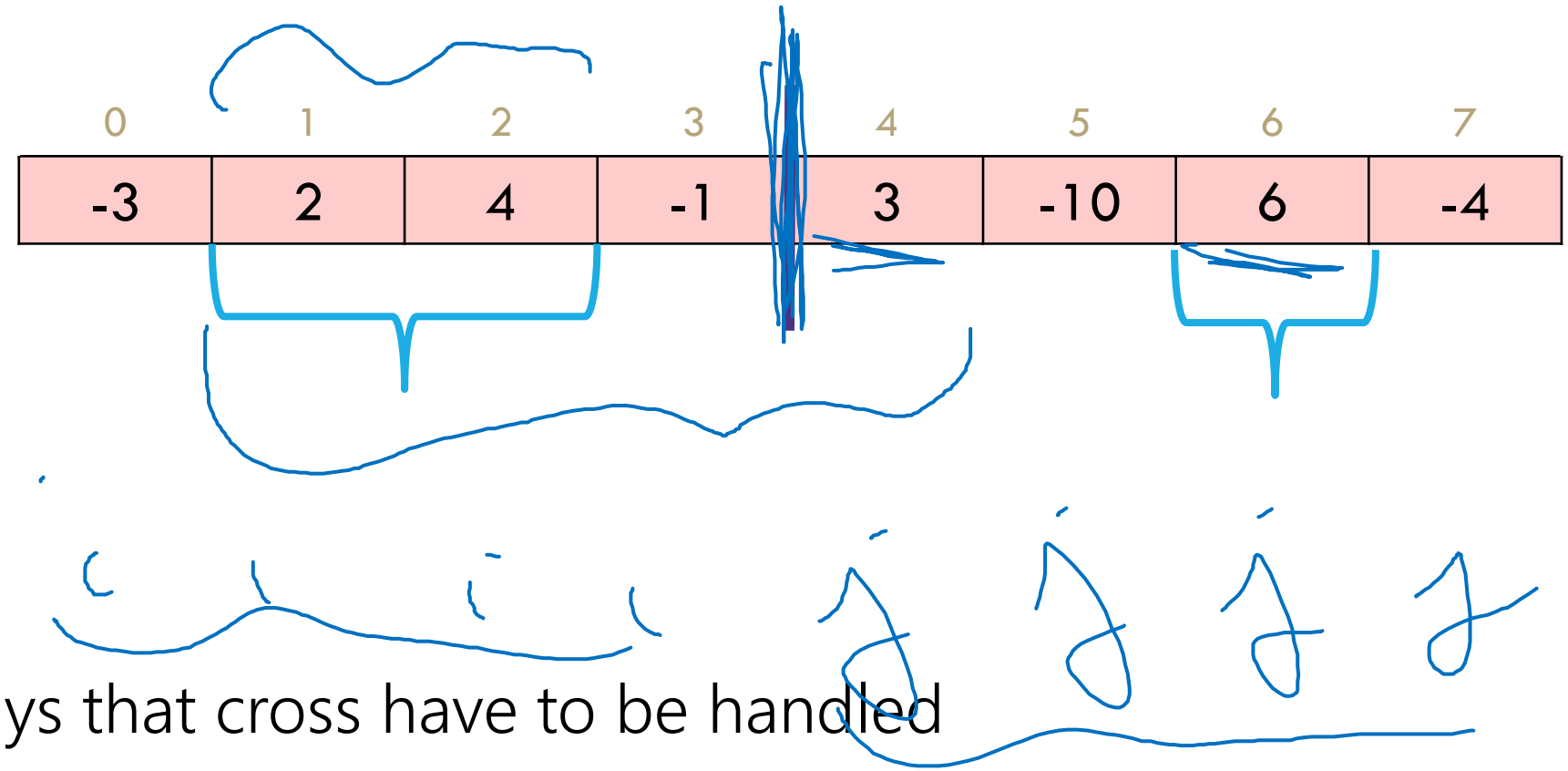
is only on the right – handled by the second recursive call.

crosses the middle – TODO

Do we have to check all pairs i, j?



Conquer

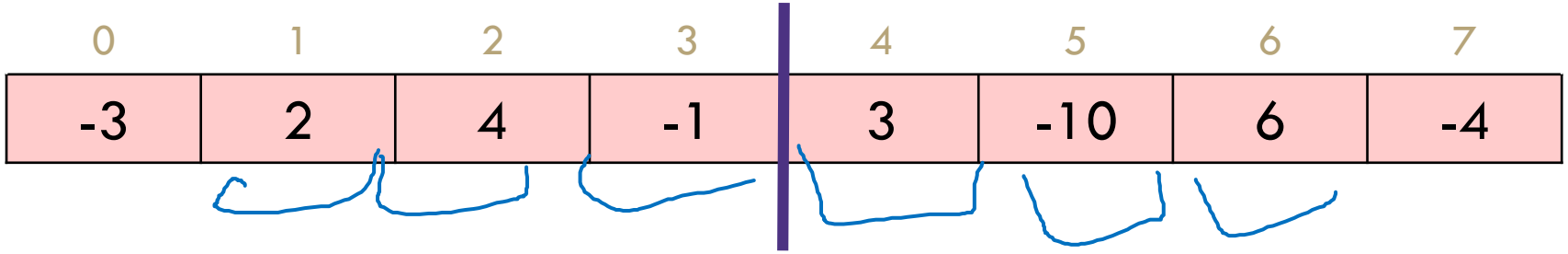


Subarrays that cross have to be handled

Do we have to check all pairs i, j?

Crossing Subarrays

Do we have to check all pairs i, j?



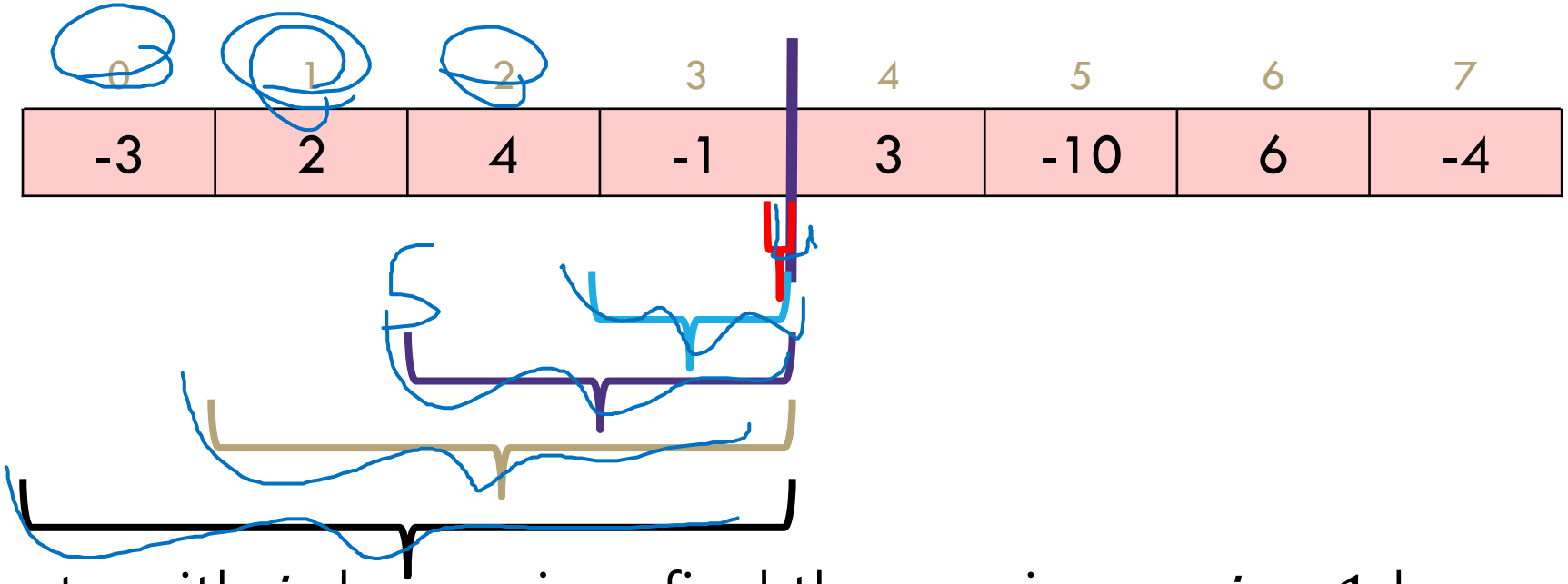
Sum is $A\left[\frac{n}{2}-1\right]+A\left[\frac{n}{2}-2\right]+\cdots+A[i]+A\left[\frac{n}{2}\right]+A\left[\frac{n}{2}+1\right]+\cdots A[j]$

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?

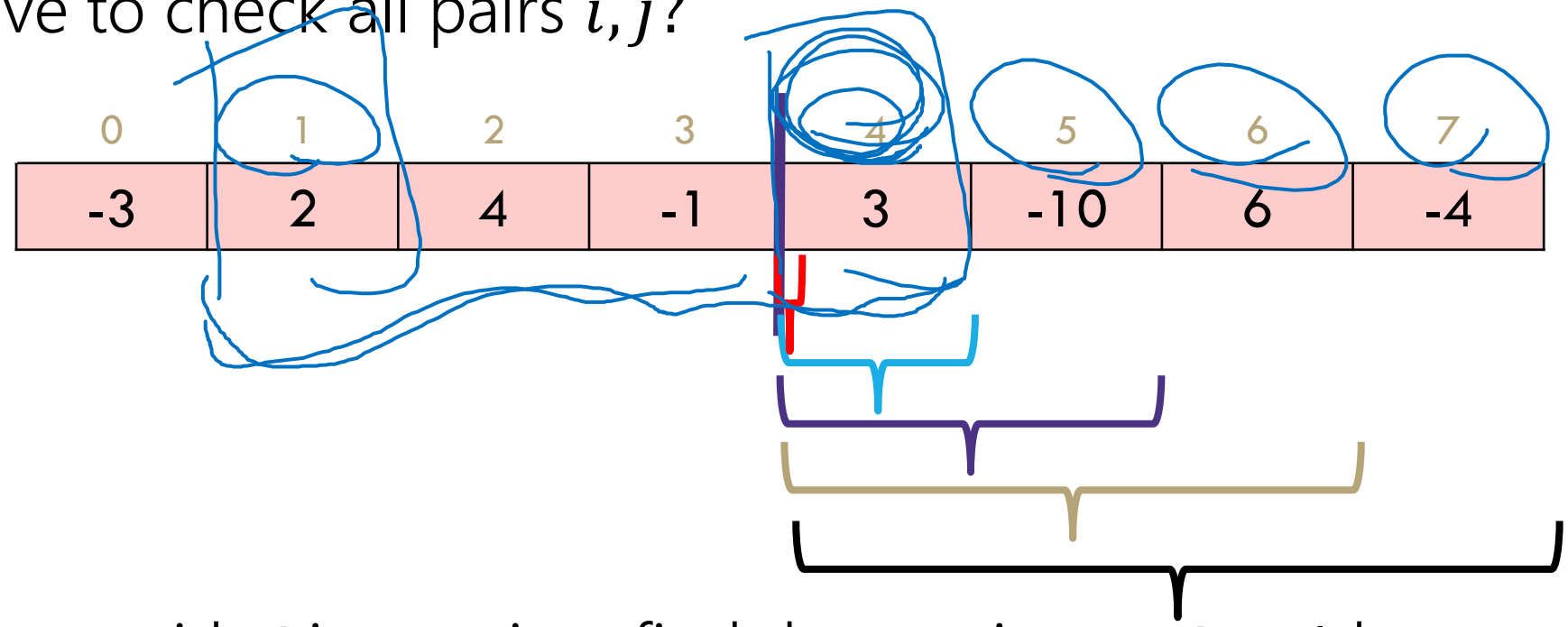


Best i? Iterate with i decreasing, find the maximum. $i = 1$ has sum 5.

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

Crossing Subarrays

Do we have to check all pairs i, j?

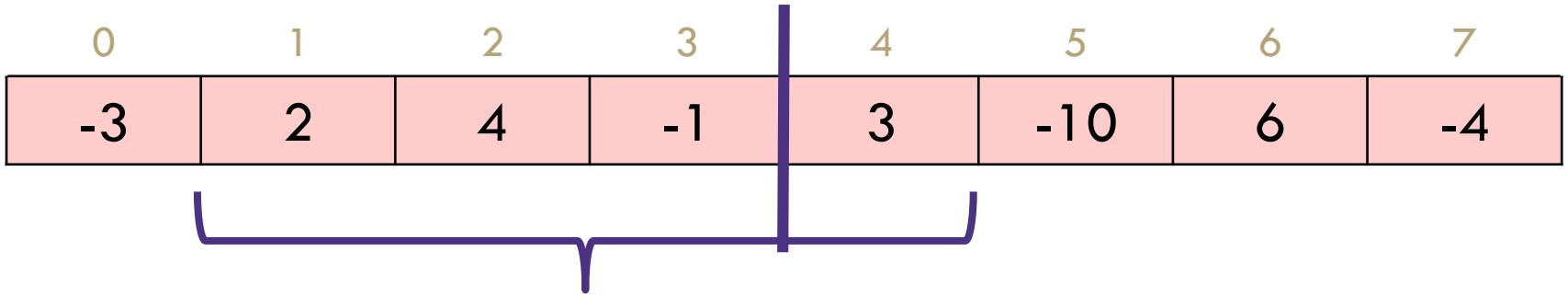


Best j? Iterate with j increasing, find the maximum. $\mathrm{j} = 4$ has sum 3.

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

Crossing Subarrays

Do we have to check all pairs i, j?

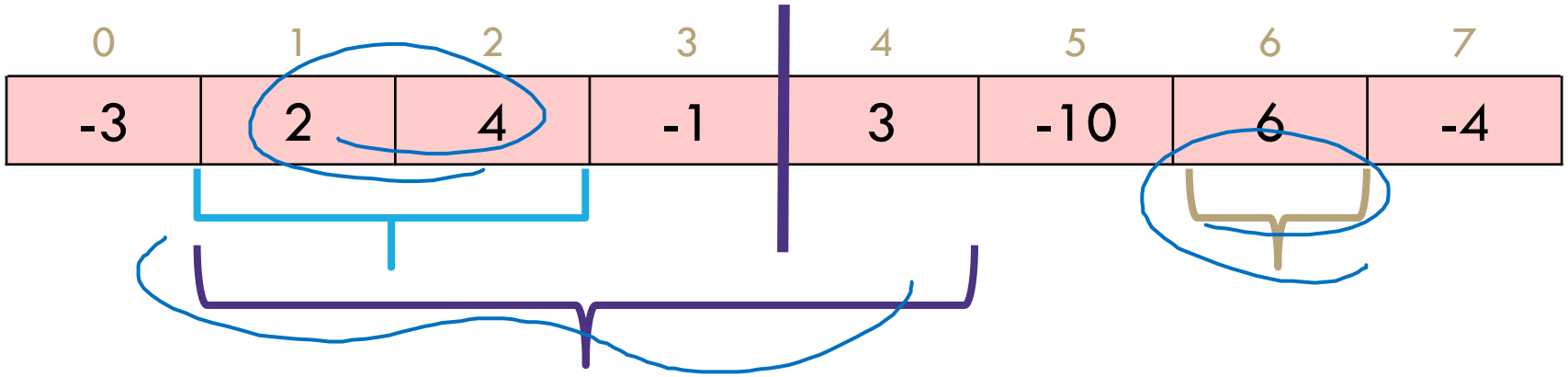


Best crossing array From the i you found to the j you found.

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 $\Theta(n)$ loops, then a constant time comparison.

So

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

Master Theorem says $\Theta(n \log n)$

```
MaxContigSubarraySum(int[] A, int start, int end)
  if(end - start < 0) //no elements
    return 0
  else if(end == start) //one element
    return max(A[start], 0) //allow for empty
    int mid = start + (start - end)/2
    int leftSum = MaxContigSubarraySum(A, start, mid)
    int rightSum = MaxContigSubarraySum(A, mid+1, end)
    //find best split sum
    int bestSumSoFar=0; int bestISoFar = mid+1; int runningSum=0;
    for(int i=mid; i >= start; i--)
      runningSum+=A[i];
      if(runningSum > bestSumSoFar)
        bestISoFar=i;
        bestSumSoFar=runningSum;
      endIf
    endFor
    bestSumSoFar=0; int bestJSoFar=mid; runningSum=0;
    for(int j=mid+1; j <= end; j++_
      runningSum+=A[j];
      if(runningSum > bestSumSoFar)
        bestJSoFar=j
        bestSumSoFar=runningSum
      EndIf
    EndFor
  EndIf
  splitSum = 0; for(int k = bestISoFar; k <= bestJSoFar; k++) splitSum+=A[k];
  return max(leftSum, rightSum, splitSum)
```

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?

$a^{n/2}\%b$ 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
```

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
```

Go to pollev.com/robbie so I know how long to explain

Master Theorem

Given a recurrence of the following form, where a, b, c, and d are constants:

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Running Time?

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

Which is $\Theta(\log n)$ 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.

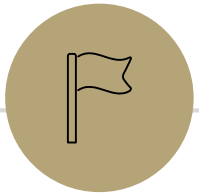
When to use Divide & Conquer

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 “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.

Multiplying Faster

Our end goal is to multiply really big numbers.

Really big.

Like around 1000 bits per number.

Why? Another step in the RSA encryption scheme!

Also a favorite problem of theoreticians.

Place Values

Remember the decimal number 12,345 is really:

$1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$

The "normal" representation for numbers is base-10. Each place value is worth 10 times more than the same number would be one spot over.

The choice of 10 is arbitrary. You can use 2 and get binary

101101_2 is $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Place Values

Imagine, we instead of using base-10, use base $\mathbf{100}$.

We need 100 symbols now. (we need 2 bits for binary, 10 digits for decimal) But other than that, it would make sense.

We could take 12345 and think of it as:

$$1 \cdot 100^2 + 23 \cdot 100^1 + 45 \cdot 100^0$$

If we thought in base-100, we'd have special symbols for 23 and 45…but it does show what's going on.

Notice that since 100 is a power of 10, we've "regrouped" the digits.

Place Values

If we have an n-digit number in base 10. We can think of it as a number in base 10^i by just "grouping together" the digits in groups of i.

We just saw $i = 2$

$i = 3$ 12345 in base 10 is:

$12 \cdot 1000^1 + 345 \cdot 10^0$ in base 1000.

Why does it matter?

Imagine I asked you to multiply

$$
\begin{array}{r}
12345 \\
\times\ 67210 \\
\hline
00000 \\
12345- \\
24690- - \\
\cdots
\end{array}
$$

Take the ones place, multiply digit-by-digit.
Take the 10's place, multiply digit-by-digit, and shift answer by 10.
Take the 100's place, multiply digit-by-digit, and shift the answer by 100.

Why does it matter?

Now imagine they're in base 1000

$$\begin{array}{r} 12345 \\ \times\ 67210 \\ \hline \end{array}$$

Take the ones place, multiply digit-by-digit.
Take the 1000's place, multiply digit-by-digit, and shift answer by 1000.

Alright…so…

We can multiply in a different way, why should you care?

It’s going to let us divide-and-conquer multiplication.

Classic Application

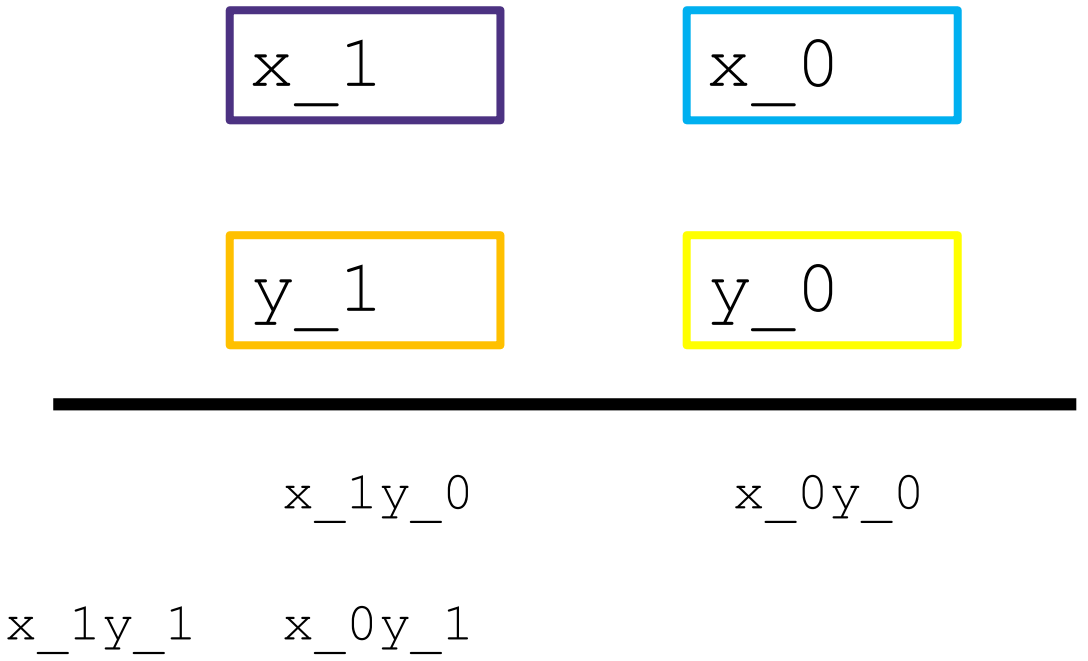
Suppose you need to multiply **really** big numbers.

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?



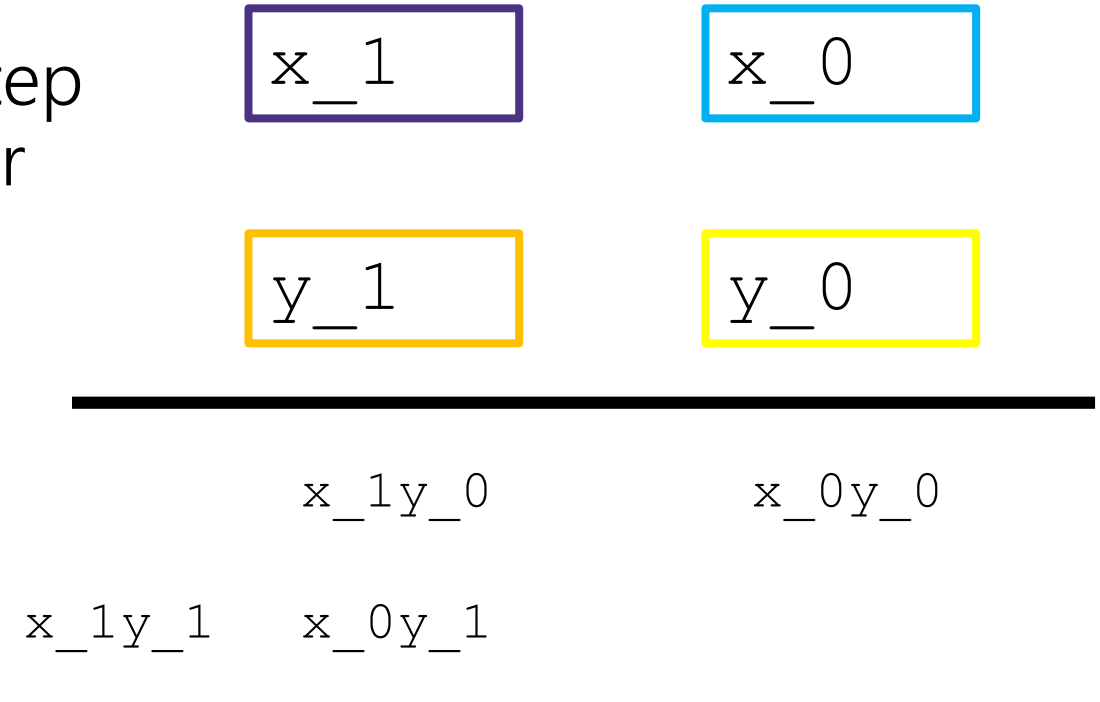
4 multiplications, i.e. 4 recursive calls.

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

$\alpha \cdot \beta = x_0y_0 + x_1y_0 + x_0y_1 + x_1y_1$

So $\alpha\beta - x_0y_0 - x_1y_1 = x_1y_0 + x_0y_1$

What do we need to find the overall multiplication?

$x_0y_0 + (\alpha\beta - x_0y_0 - x_1y_1) \cdot 2^{\frac{n}{2}} + x_1y_1 \cdot 2^n$

x_0y_0, x_1y_1 and $\alpha\beta$ are enough to calculate the overall answer! Only 3 multiplies of n/2 bits!

Running Time

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

$\log_2(3) > 1$,

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

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

You can go faster!

In 2019 (!!!) a paper was published describing an algorithm to multiply numbers in $O(n \log n)$ time

Suspected to be impossible to improve, but not proven yet!

Unfortunately, the algorithm isn't practical. It's defined only when the numbers to multiply have more than 2^{4096} digits.

But it really has been a favorite problem of theoreticians for decades through extremely recently.

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 $O(n^{\log_2 7}) \approx O(n^{2.81})$.

Lots of more clever ideas have gotten matrix multiplication even faster **theoretically**.

We even have special notation $O(n^\omega)$ is defined as "the fastest we know how to multiply matrices." It's another favorite problem.

In practice, the constant factors are too high to use the "fastest" algorithms.