Divide & Conquer

CSE 417 Winter 21
Lecture 9

Announcements

Wednesday:

First ~15 minutes of lecture tomorrow will be a discussion with Ken Yasuhara from UW Engineering Teaching & Learning

Goal is to get me rapid feedback on some of the early changes in the course (what’s working, what isn’t, what would help).

If you’re participating asynchronously, we’ll get your feedback through a survey afterward.

Divide & Conquer

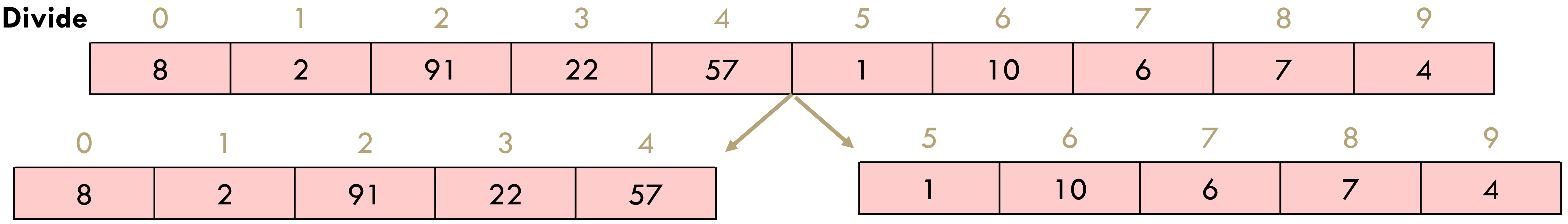
Algorithm Design Paradigm

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

Merge Sort

https://www.youtube.com/watch?v=XaqR3G_NVoo

Divide



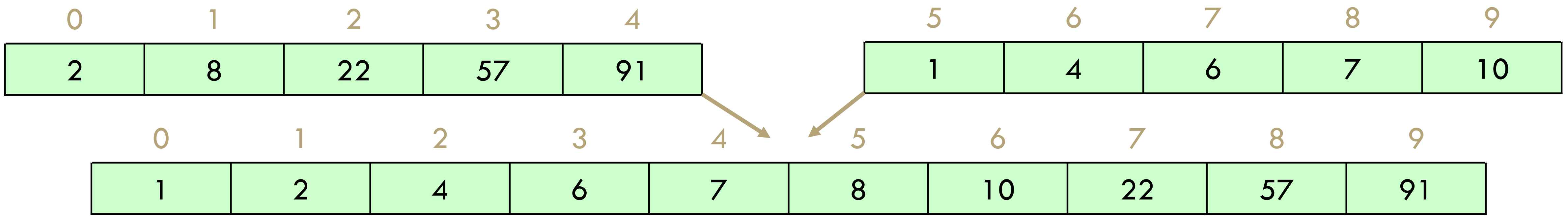
0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	10	6	7	4

Sort the pieces through the magic of recursion

Combine



0	1	2	3	4
2	8	22	57	91

5	6	7	8	9
1	4	6	7	10

0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

Merge Sort

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```

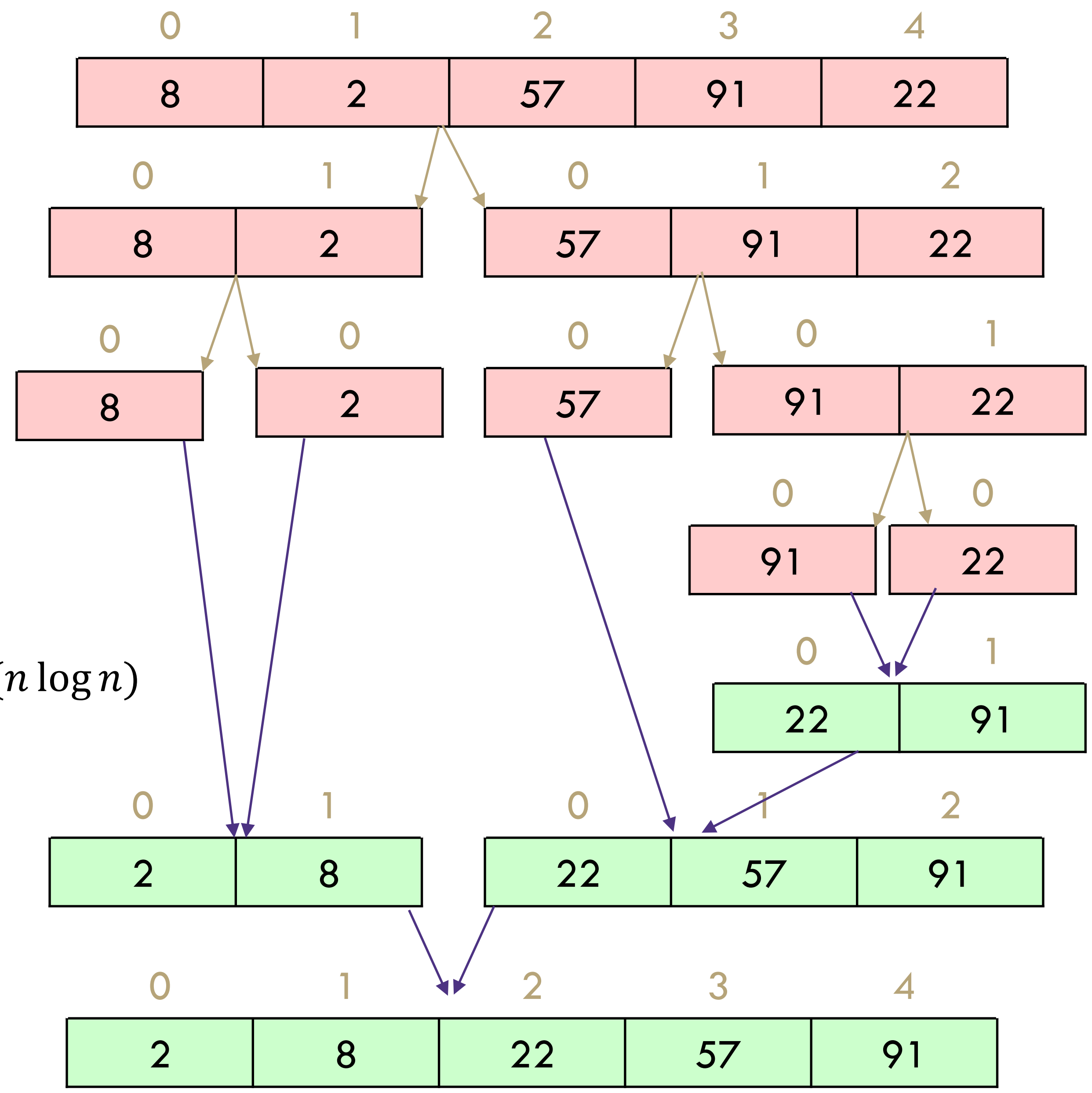
Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases} = O(n \log n)$

Best case runtime? Same

Average runtime? Same

Stable? Yes

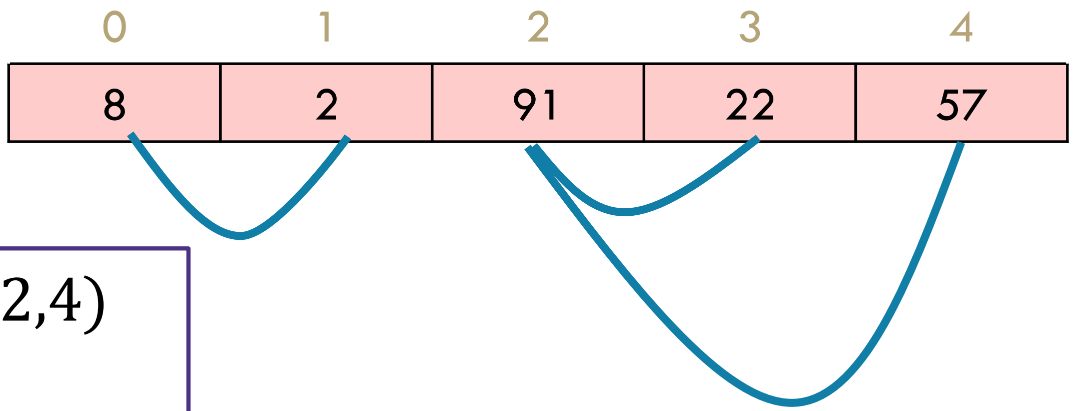
In-place? No



Counting Inversions

Given an array, A, determine how "unsorted" it is, by counting number of inversions:

Inversion: pair i, j such that $i < j$ but $A[i] > A[j]$



$(0,1), (2,3)$, and $(2,4)$ are inversions

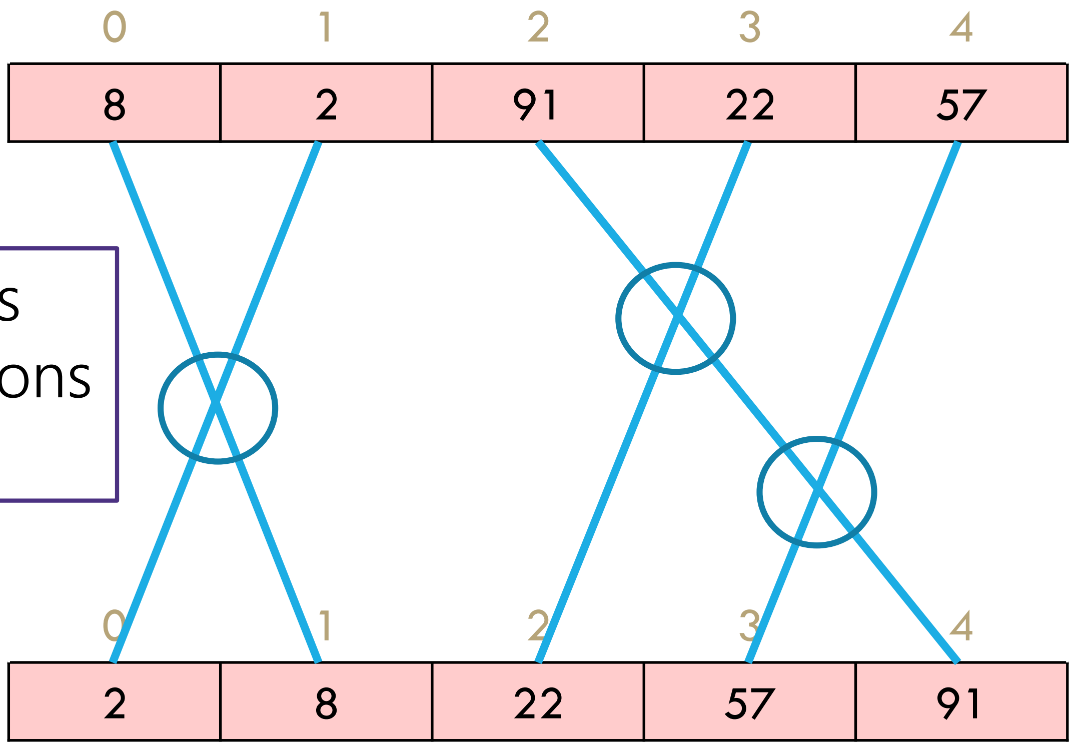
Intuitively, how many adjacent swaps to fully sort.

Why? Tell "how different" two lists are (e.g. tell if someone's opinion is an outlier, or if two people have similar preferences)

Counting Inversions

Given an array, A, determine how "unsorted" it is:

Find the number of pairs i, j such that $i < j$ but $A[i] > A[j]$



Visualization: overlaps correspond to inversions (needed swaps)

Counting Inversions

What's the first idea that comes to mind (don't try to divide and conquer yet).

Check every pair i, j

$\Theta(n^2)$ time.

Goal: do better than $\Theta(n^2)$

Divide & Conquer Inversions

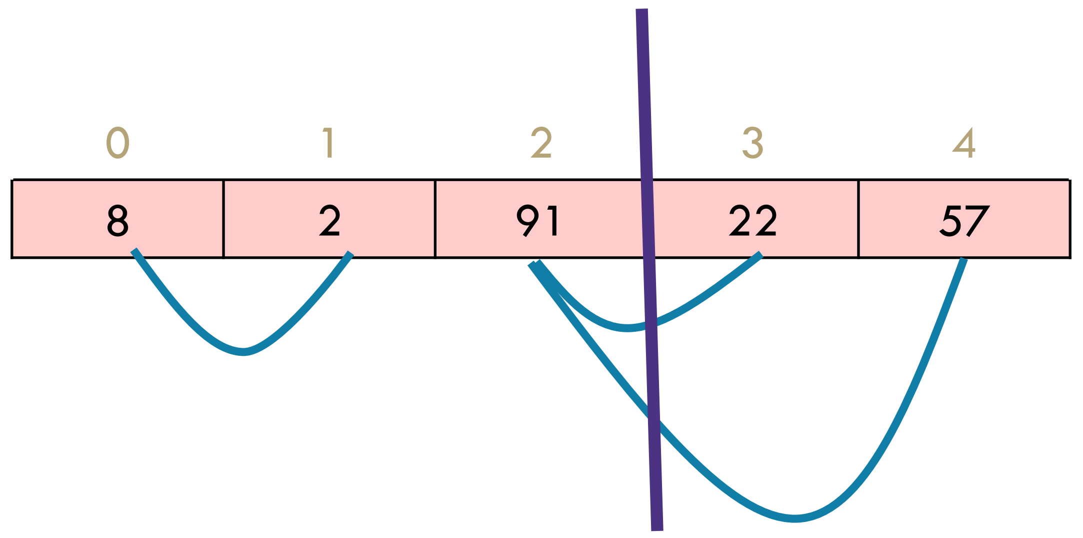
1. Divide instance into subparts.
2. Solve the parts recursively.
3. Conquer by combining the answers

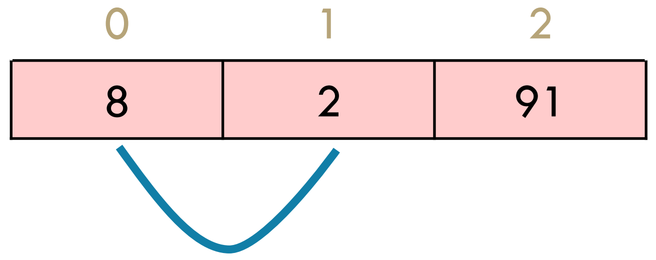
1. Split array in half (indices $0, \frac{n}{2} - 1$ and $\frac{n}{2}, n - 1$)
2. Solve the parts recursively (gives all inversions in each half)
3. Combine the answers

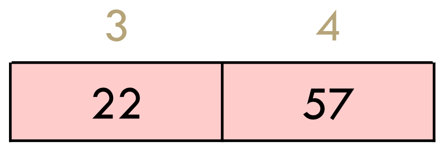
So…do we just add?

Conquer

Can't just add!







Conquer

Kinds of i, j

Both, i, j in left side – counted by recursive call

Both i, j in right side – counted by other recursive call

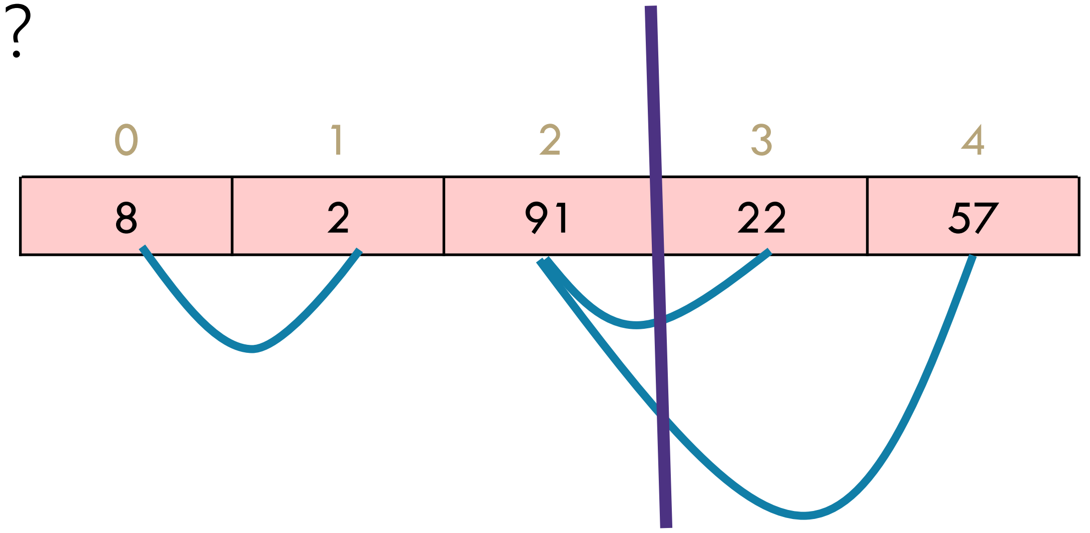
i in left side, j in right side – TODO

i in right side, j in left side – Can't have $i < j$ no inversions here.

Need to handle TODO. Then add together.

Inversions across the middle

Fix some i on the left side. How many j on the right side form inversions?



So how do we find all the "crossing inversions"

$\frac{n}{2}$ elements, each checking $\frac{n}{2}$ others, so that's time…$O(n^2)$ to merge

Running Time

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

Master Theorem says:

Master Theorem

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

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

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

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

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O\left(n^2\right) \text{ if } n \geq 2 \\ O(1) \qquad\qquad \text{othwerwise} \end{cases}$$

Master Theorem says:

$\log_2(2) = 1 < 2$

So $O(n^2)$

Not actually better than brute force.

Divide & Conquer Smarter

In fact, all that divide & conquer did was **rearrange** the work we were doing anyway.

We're still explicitly checking for every i, j "is i, j an inversion?"

The trick to making divide & conquer efficient is to make it so that conquering is **easier** than just solving the whole problem.

Counting Across the Middle

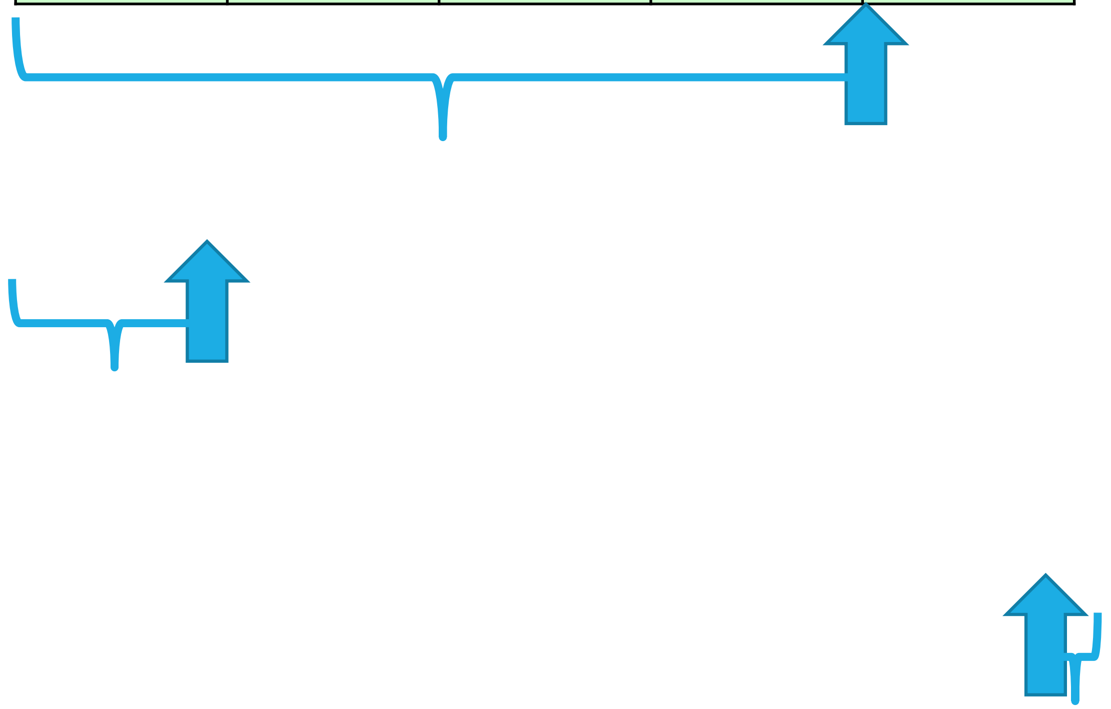
Fix some i on the left side. How many j on the right side form inversions?

What would we do if the right hand side were sorted?

Counting the inversions

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	4	6	7	10



Inversions involving index 0: $(0{,}5)$ … $(0{,}8)$

Inversions involving index 1: $(1{,}5)$

Inversions involving index 4: [none]

Counting Across the Middle

Fix some i on the left side. How many j on the right side form inversions?

What would we do if the right hand side were sorted?

Binary search!

Time? $\mathrm{O}(\log n)$ per element on the left side…so $O(n \log n)$ to combine

Analyze, round 2.

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n\log n) \text{ if } n \geq 2 \\ \quad O(1) \qquad\qquad \text{othwerwise} \end{cases}$$

Master Theorem says:

$\log_2(2) = 1\,?\,ummm$

Master Theorem

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

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

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

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 $T(n) \in \Theta\left(n^{\log_b a}\right)$

Pause

Lets get some intuition

$O(n \log n)$ is closest to which of these:

$O(n)$

$O(n^{1.1})$

$O(n\sqrt{n})$

$O(n^2)$

Pause

Lets get some intuition

$O(n \log n)$ is closest to which of these:

$\boxed{O(n)}$

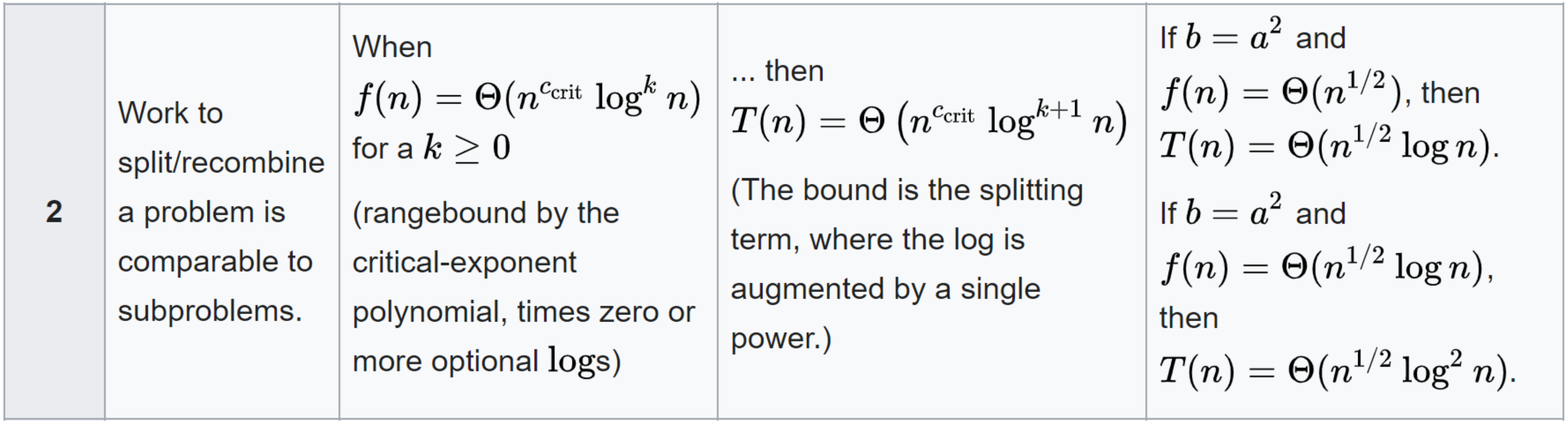
$O(n^{1.1})$

$O(n\sqrt{n})$

$O(n^2)$

So we'd expect to get an answer between $\Theta(n \log n)$ and $\Theta(n^{1.1} \log n)$, but closer to $\Theta(n \log n)$

Master Theorem, v2



2	Work to split/recombine a problem is comparable to subproblems.	When $f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for a $k \geq 0$ (rangebound by the critical-exponent polynomial, times zero or more optional $\log$s)	... then $T(n) = \Theta\left(n^{c_{\text{crit}}} \log^{k+1} n\right)$ (The bound is the splitting term, where the log is augmented by a single power.)	If $b = a^2$ and $f(n) = \Theta(n^{1/2})$, then $T(n) = \Theta(n^{1/2} \log n)$. If $b = a^2$ and $f(n) = \Theta(n^{1/2} \log n)$, then $T(n) = \Theta(n^{1/2} \log^2 n)$.

$\Theta(n \log^2 n)$

i.e. $\Theta(n \cdot \log(n) \cdot \log(n))$

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Counting Across the Middle

So sort the array first! As a preprocessing step

Then count the inversions.

What's the problem?

When you sort, the inversions disappear

Ok, sort as part of the process.

Almost there...

```
int CountInversions(A, int start, int stop)
  inversions = 0
  if(start >= stop)
    return 0
  int midpoint = (stop-start)/2 + start
  inversions += CountInversions(A, start, midpoint)
  inversions += CountInversions(A, midpoint+1, end)
  sort(A, midpoint+1, end)
  for(int i=start; i <= midpoint; i++)
    int k = binarySearch(A, midpoint+1, end, i)
    inversions += k-(midpoint+1)+1
  return inversions
```

Just a liiiiiiittle better

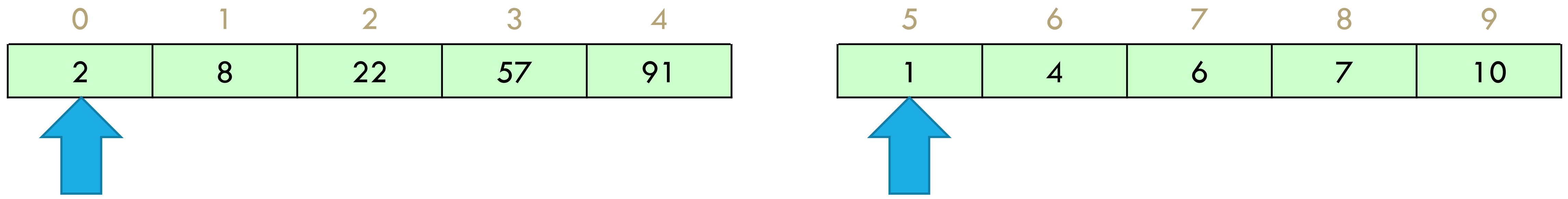
Sort the left subarray too!

Can that help us?

If i, j is an inversion then $i + 1, j$ and $i + 2, j$, and, … $\frac{n}{2} - 1, j$ are also inversions.

Don't have to binary search every time, can just "march down" lists.

Sorted example



$(0,5)$ is an inversion. $(1,5), (2,5), (3,5), (4,5)$ are too!

Know everything we need to about index 5.

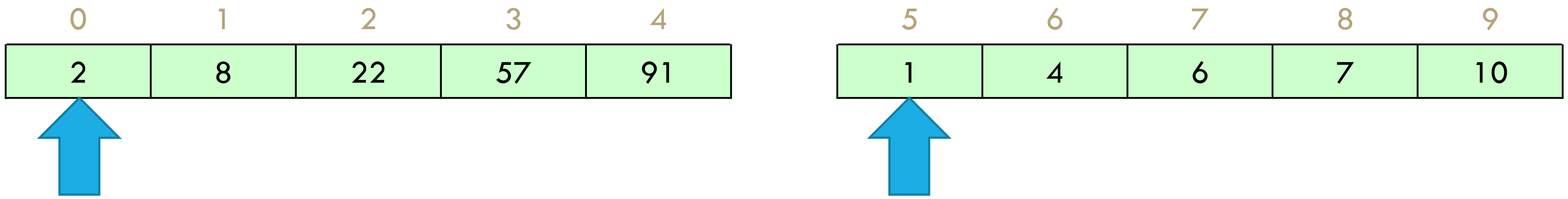
$(0,6)$ is not an inversion. $(0,7) \ldots, (0,9)$ aren't either.

Know everything we need to about index 0.

$(1,6)$ is an inversion $(2,6), (2,7), (2,8)$ are too!

Know everything we need to about index 6. …

In general



In general:

If (i, j) is an inversion, have $(\frac{n}{2} - i)$ inversions. Increase j.

If (i, j) is not an inversion, increase i.

Time to iterate?…$\Theta(n)$

Does this…look familiar

Having an arrow to a spot in two arrays, moving whichever is on the smaller value forward…

That’s how merge from mergesort works!

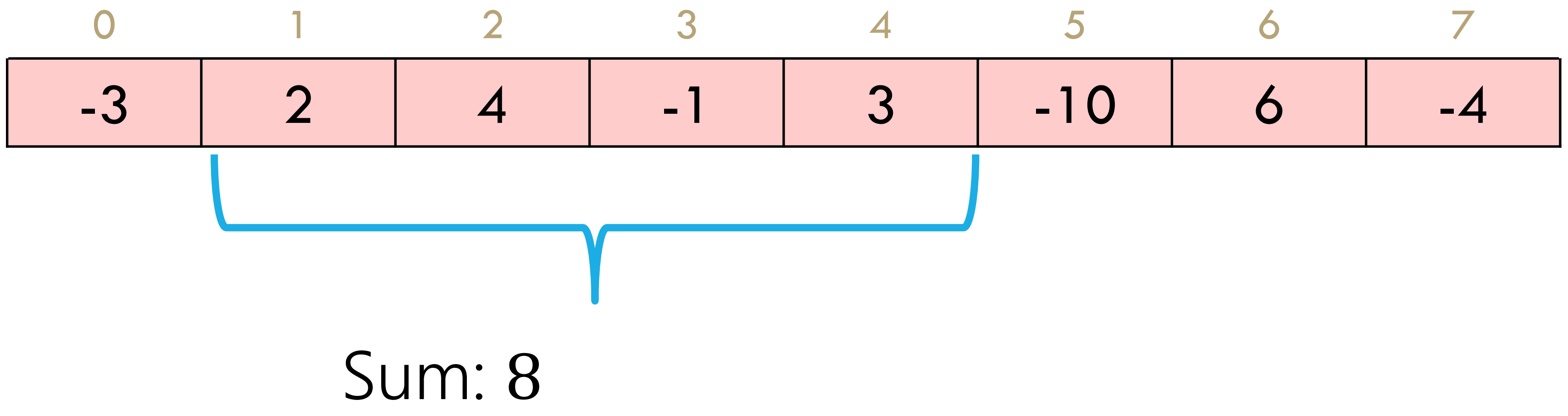
If we sort (by mergesort) and count inversions **as we’re merging** we save that log factor.

Running time $\Theta(n \log n)$

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.



Maximum 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.

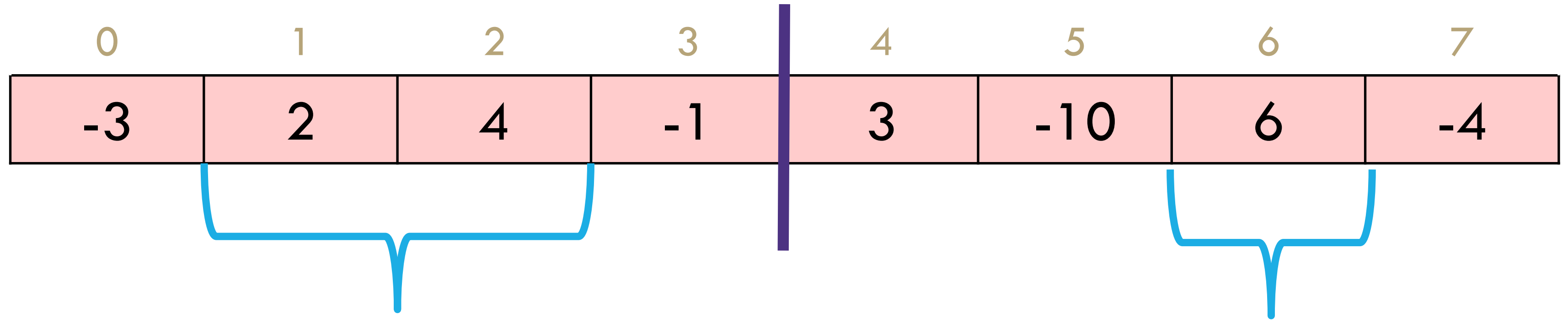
Can we do better?

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



Subarrays that cross have to be handled

Do we have to check all pairs i, j?

Conquer

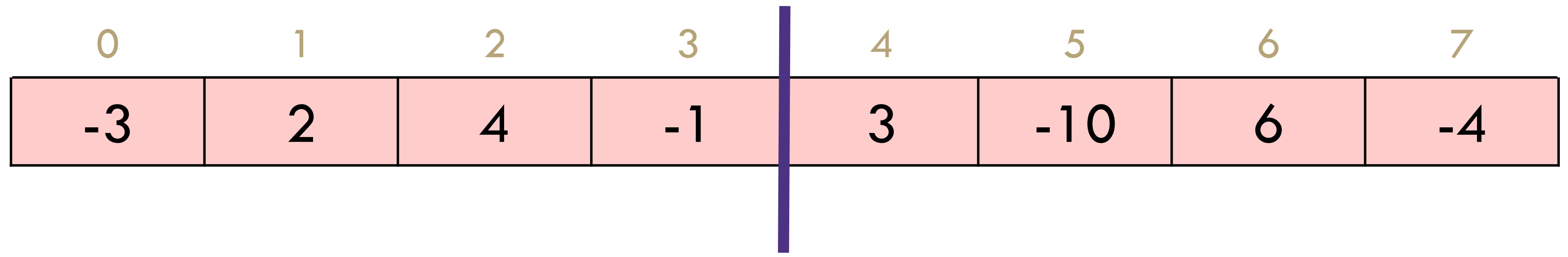
If the optimal subarray:

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

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

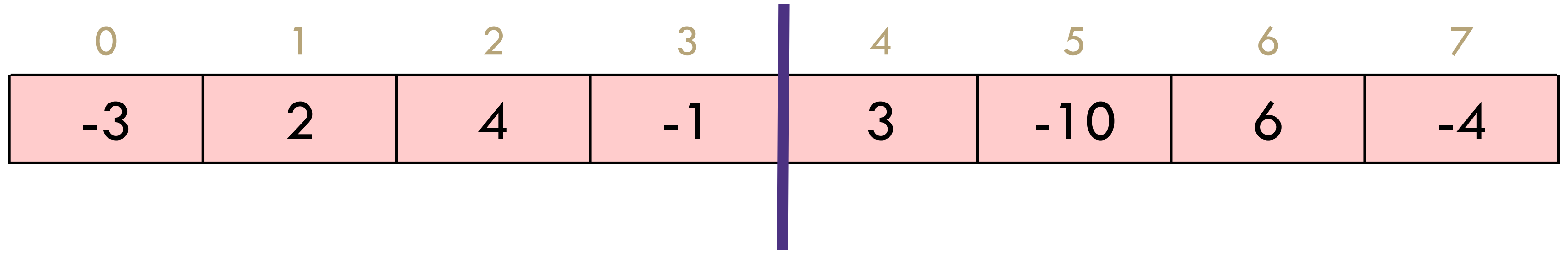
crosses the middle – TODO

Do we have to check all pairs i, j?



Crossing Subarrays

Do we have to check all pairs i, j?



0	1	2	3	4	5	6	7
-3	2	4	-1	3	-10	6	-4

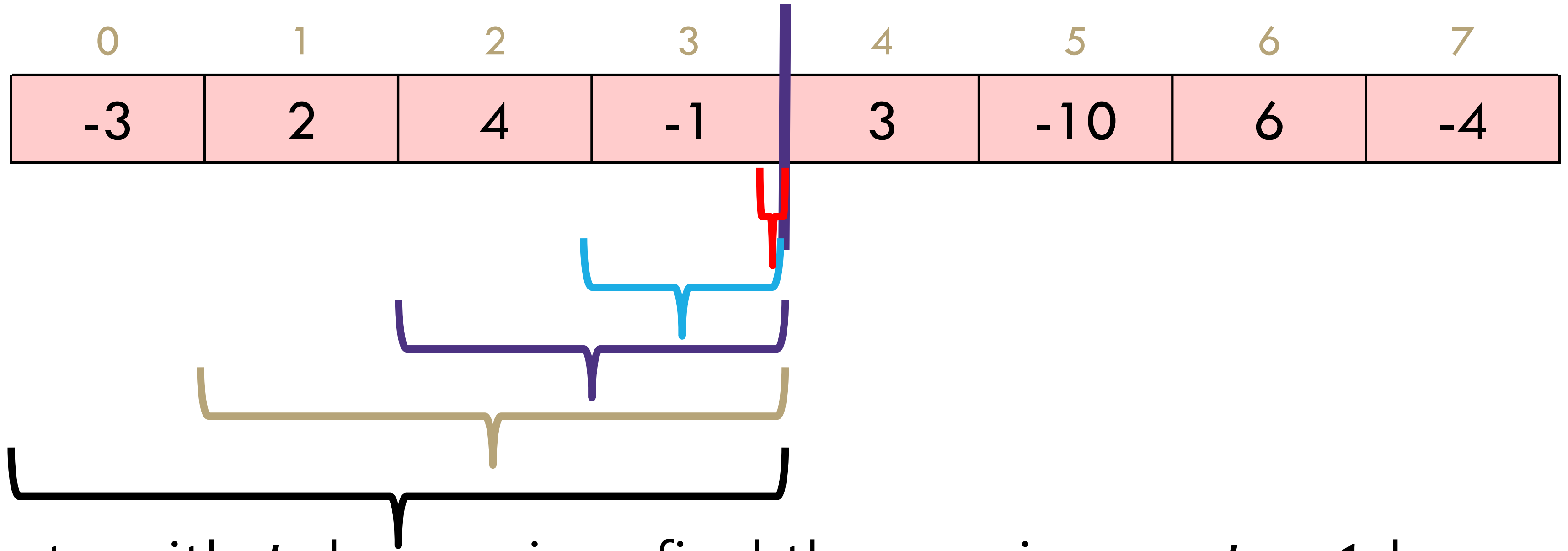
Sum is $A\left[\frac{n}{2}-1\right] + A\left[\frac{n}{2}-2\right] + \cdots + A[i] + A\left[\frac{n}{2}\right] + A\left[\frac{n}{2}+1\right] + \cdots A[j]$

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?

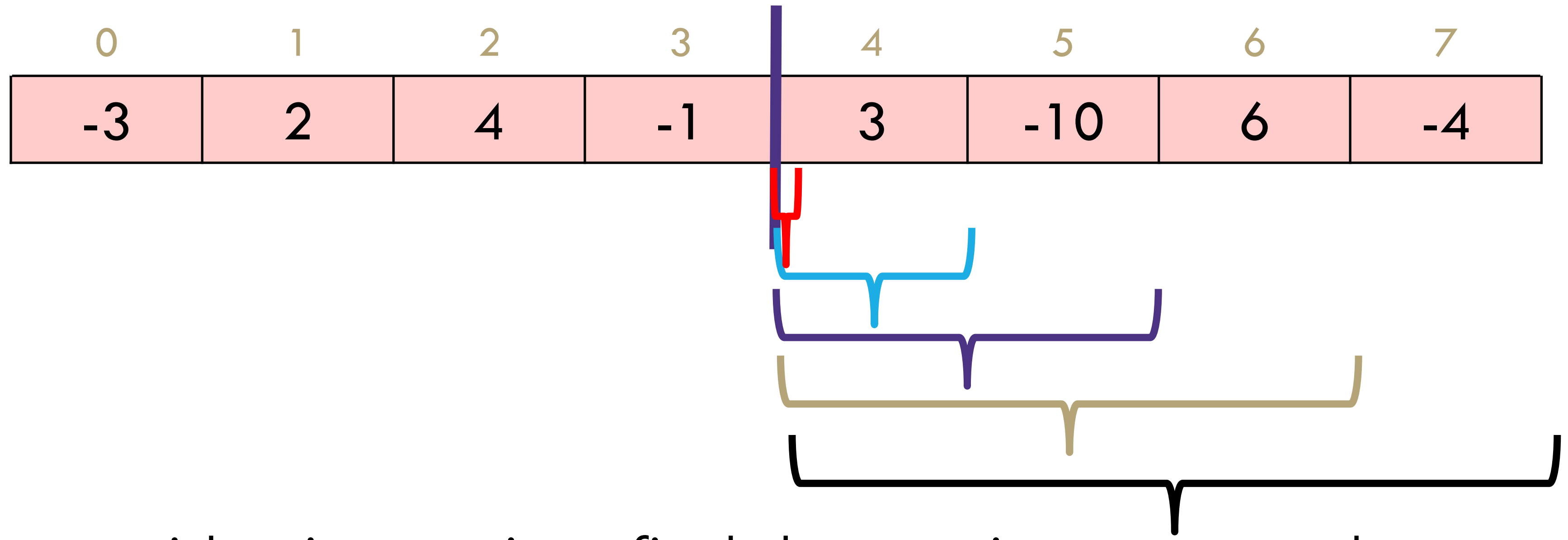


Best i? Iterate with i decreasing, find the maximum. $i = 1$ has sum 5.

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

Crossing Subarrays

Do we have to check all pairs i, j?

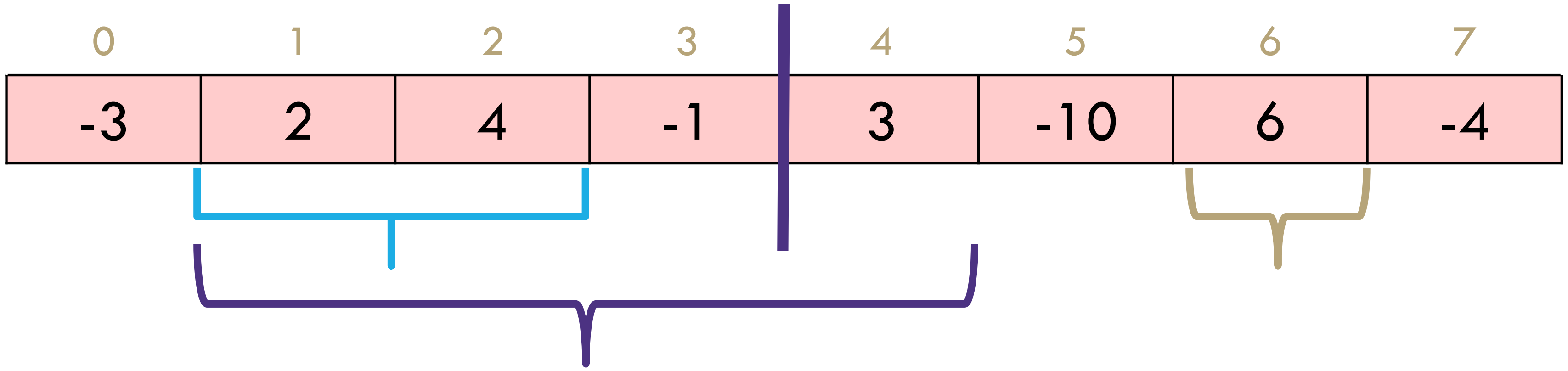


Best j? Iterate with j increasing, find the maximum. $\mathrm{j} = 4$ has sum 3.

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

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 $\Theta(n \log n)$