Searching and Sorting

Part Two

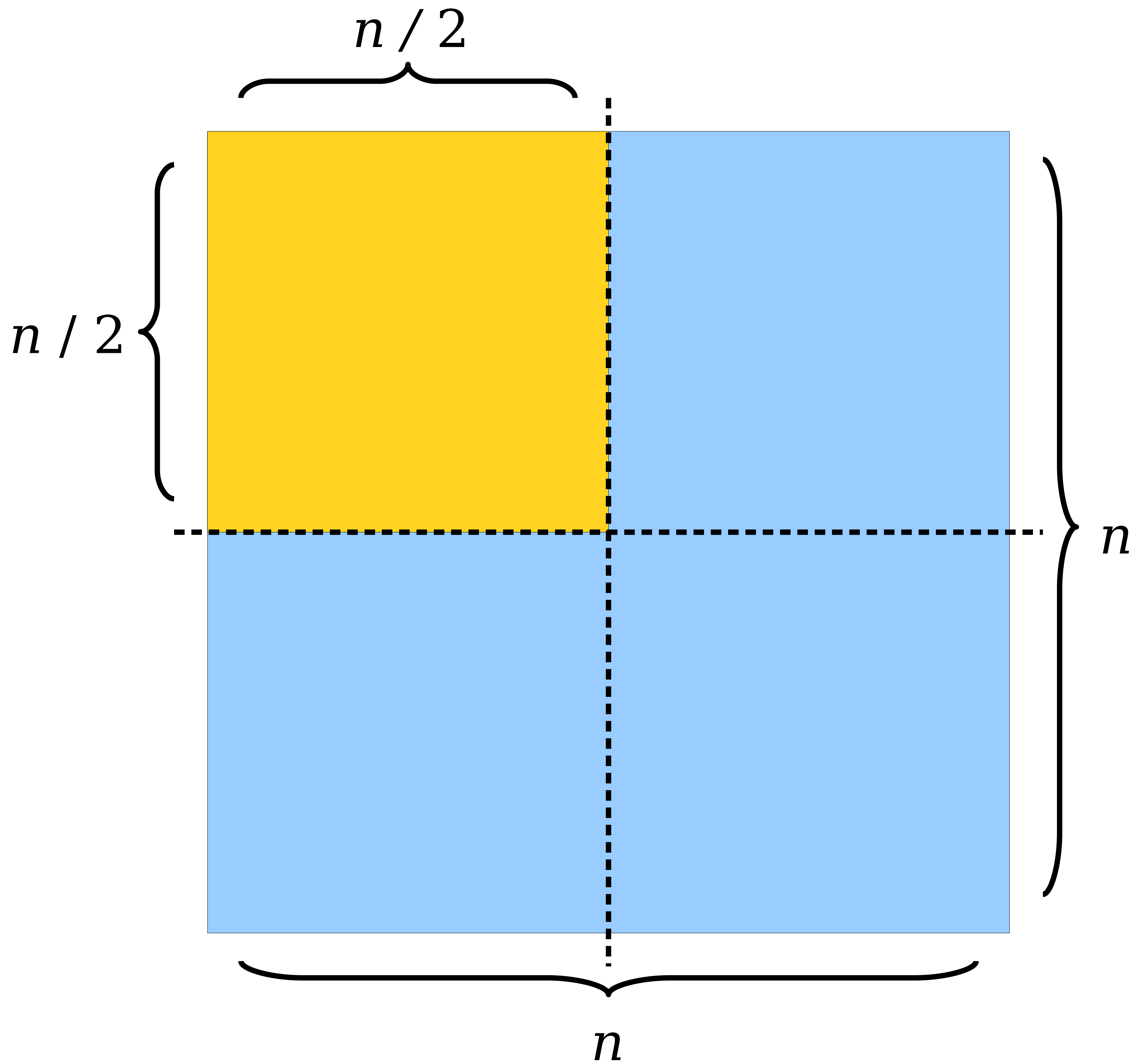
Building a Better Sorting Algorithm

A Thought Experiment

- Suppose it takes 100ms to selection sort an array of 20,000 random elements.
- Approximately how long will it take to selection sort two smaller arrays, each of which has 10,000 random elements?

Answer at

https://cs106b.stanford.edu/pollev


$n / 2$
$n / 2$
n
n

Thinking About $O(n^2)$

14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

$T(n)$

2	3	6	7	9	14	15	16

$\frac{1}{4}T(n)$

1	4	5	8	10	11	12	13

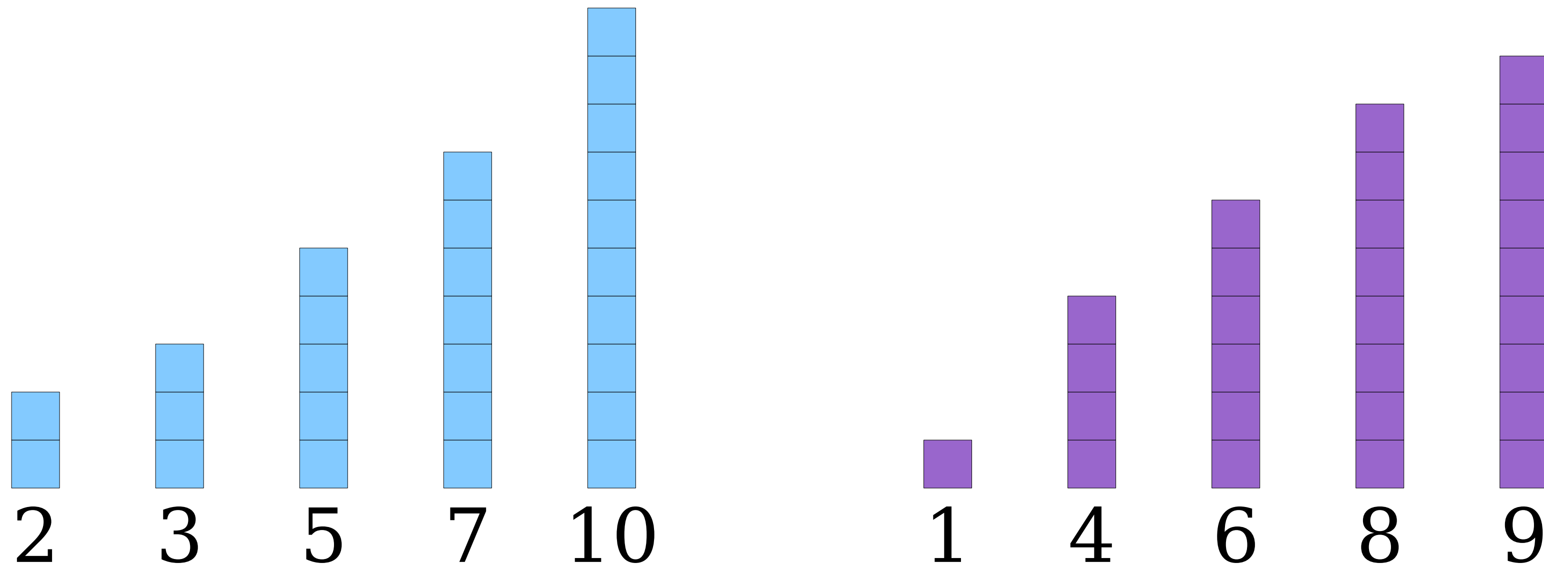
$\frac{1}{4}T(n)$

$$2 \cdot \frac{1}{4}T(n) = \frac{1}{2}T(n)$$

With an O(n^2)-time sorting algorithm, it takes twice as long to sort the whole array as it does to split the array in half and sort each half.

Can we exploit this?

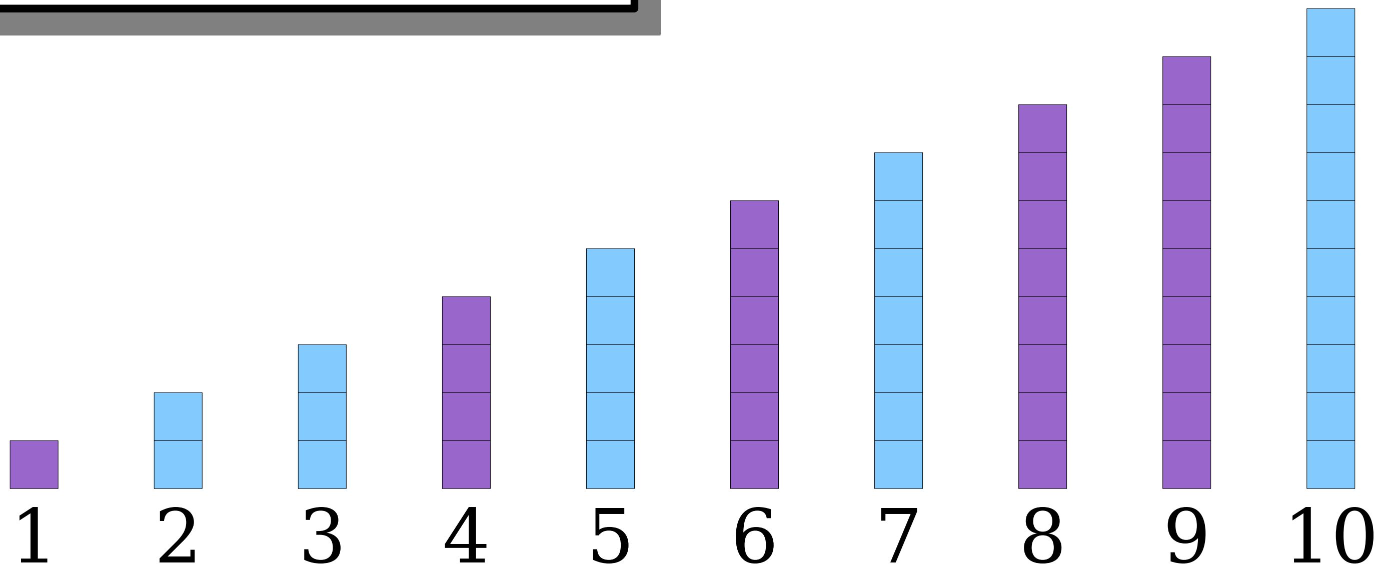
The Key Insight: ***Merge***



The Key Insight: ***Merge***

Each step makes a single comparison and reduces the number of elements by one.

If there are n total elements, this algorithm runs in time **O(n)**.

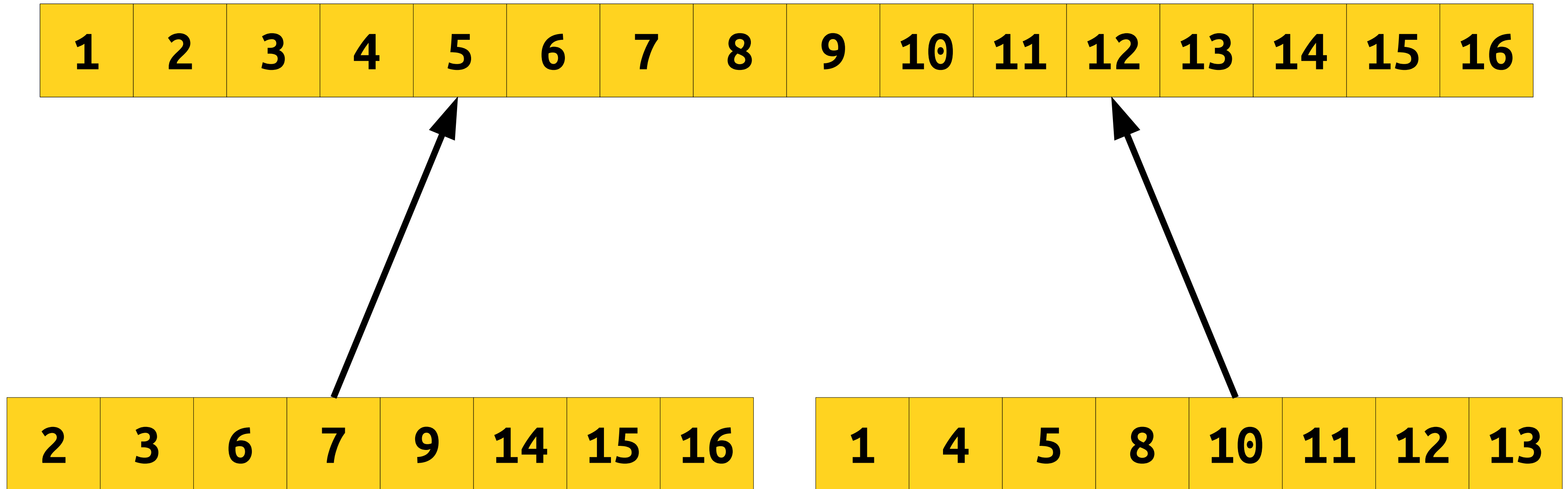


The Key Insight: ***Merge***

- The ***merge*** algorithm takes in two sorted lists and combines them into a single sorted list.
  - While both lists are nonempty, compare their first elements. Remove the smaller element and append it to the output.
  - Once one list is empty, add all elements from the other list to the output.
- It runs in time O(n), where n is the total number of elements being merged.

```
Vector<int> merge(const Vector<int>& one, const Vector<int>& two) {
    Vector<int> result;

    /* Track indices of the next unmerged elements of the vectors. */
    int oneIndex = 0, twoIndex = 0;

    /* Keep comparing elements until one vector runs out. */
    while (oneIndex < one.size() && twoIndex < two.size()) {
        /* Add the smaller element to the output. */
        if (one[oneIndex] < two[twoIndex]) {
            result += one[oneIndex];
            oneIndex++;
        } else {
            result += two[twoIndex];
            twoIndex++;
        }
    }

    /* We've exhausted a vector; add all elements of the other. */
    while (one < one.size()) {
        result += one[oneIndex];
        oneIndex++;
    }

    while (two < two.size()) {
        result += two[twoIndex];
        twoIndex++;
    }

    return result;
}
```

“Split Sort”



1. Split the input in half.
2. Selection sort each half.
3. Merge the halves back together.

“Split Sort”

```
void splitSort(Vector<int>& v) {
    /* Split the vector in half */
    int half = v.size() / 2;
    Vector<int> left  = v.subList(0, half);
    Vector<int> right = v.subList(half);

    /* Sort each half. */
    selectionSort(left);
    selectionSort(right);

    /* Merge them back together. */
    v = merge(left, right);
}
```

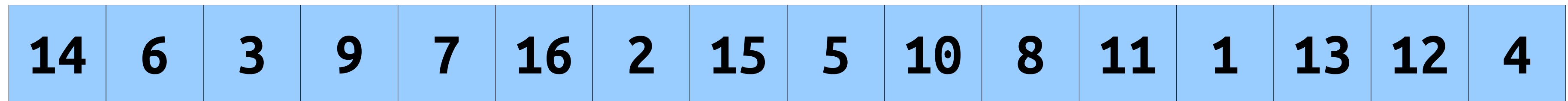
Takes O(n) time, since we copy all n elements into new `Vectors`.

Takes O(n^2) time, but about half as much as what we did before.

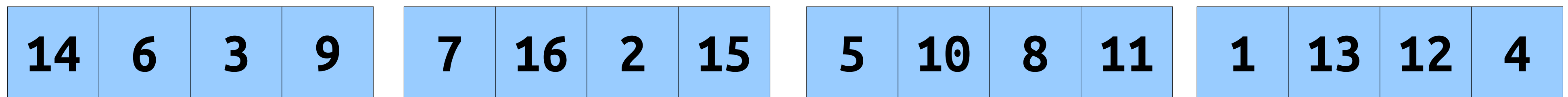
Takes O(n) time.

Prediction: This should still take time O(n^2), but be about twice as fast as selection sort.

“Double Split Sort”



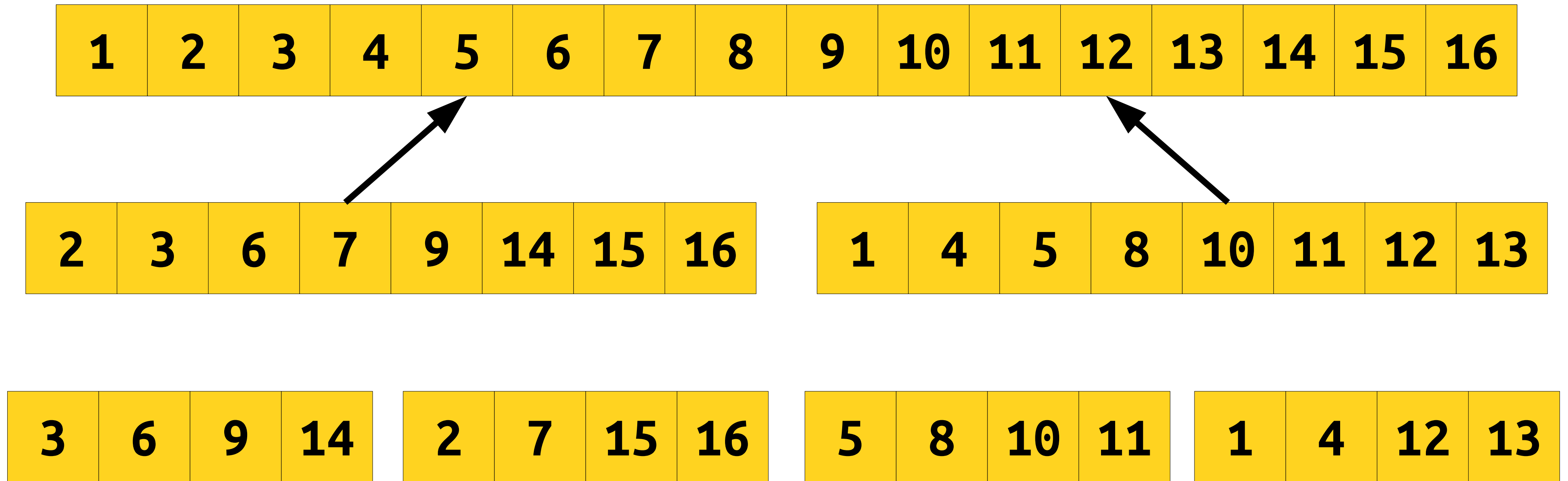
$T(n)$



$^1/_{16}\ T(n)$ $^1/_{16}\ T(n)$ $^1/_{16}\ T(n)$ $^1/_{16}\ T(n)$

$$4 \cdot {}^1/_{16}\ T(n) = \mathbf{{}^1/_4 T(\mathit{n})}$$

"Double Split Sort"

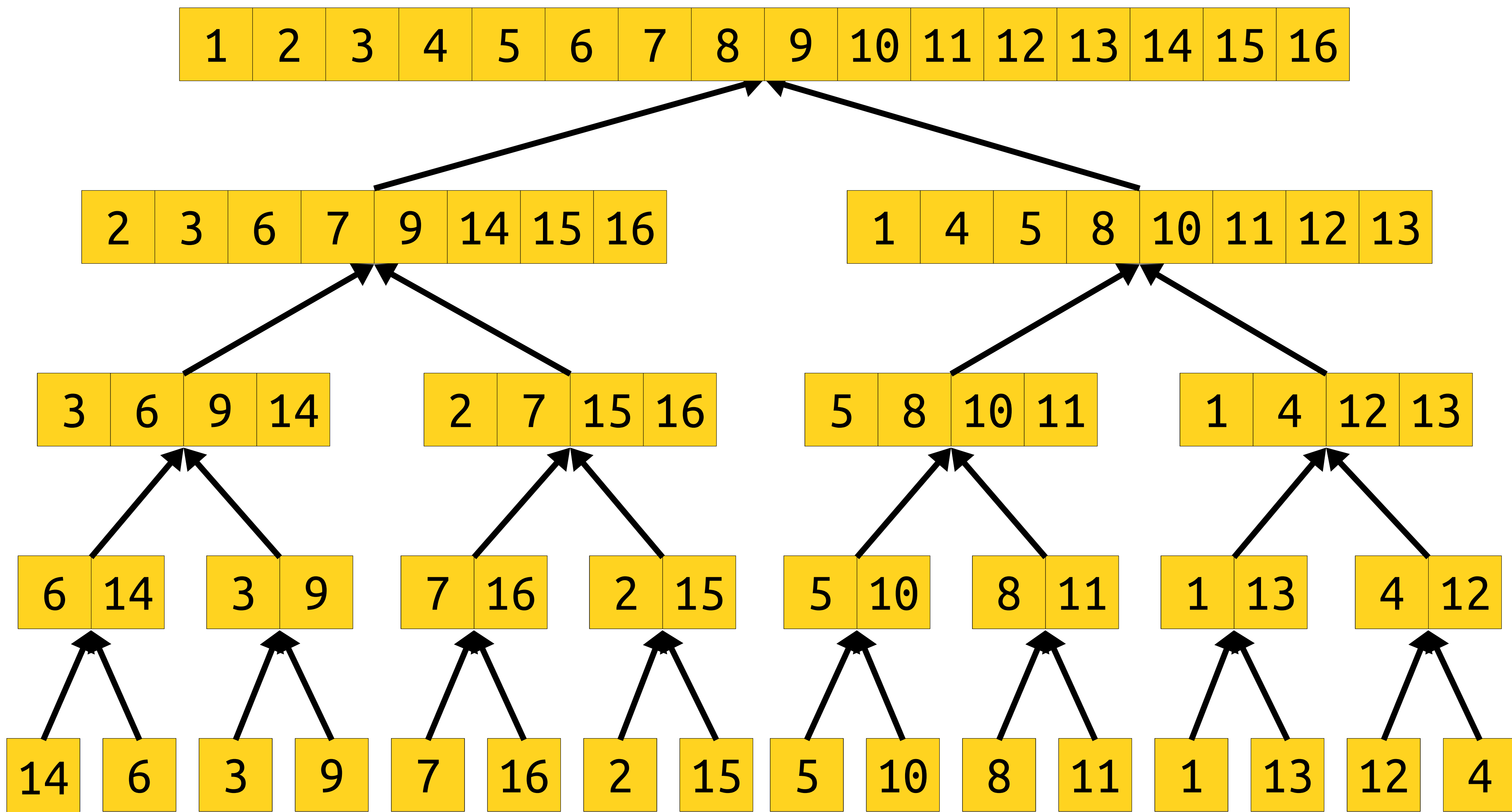


1. Split the input into quarters.
2. Selection sort each quarter.
3. Merge two pairs of quarters into halves.
4. Merge the two halves back together.

Prediction: This should be four times as fast as selection sort.

Splitting to the Extreme

- Splitting our array in half, sorting each half, and merging the halves was twice as fast as selection sort.
- Splitting our array in quarters, sorting each quarter, and merging the quarters was four times as fast as selection sort.
- ***Question:*** What happens if we *never stop splitting?*

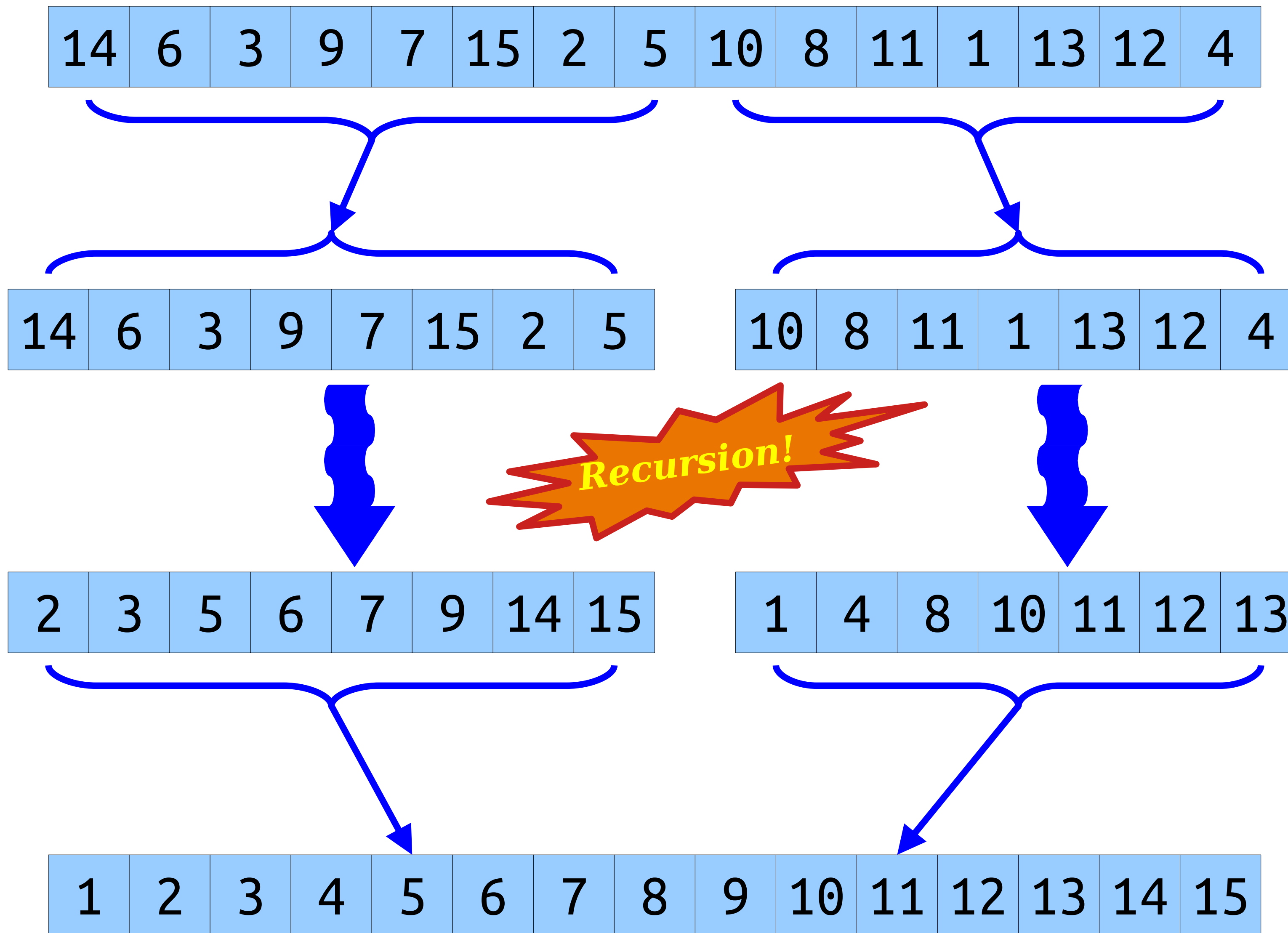


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

Mergesort

- A recursive sorting algorithm!
- ***Base Case:***
  - An empty or single-element list is already sorted.
- ***Recursive step:***
  - Break the list in half and recursively sort each part.
  - Use `merge` to combine them back into a single sorted list.

Mergesort, Intuitively



Split array into roughly equal halves

Recursively mergesort each half

Merge sorted subarrays

```
void mergesort(Vector<int>& v) {
   /* Base case: 0- or 1-element lists are
    * already sorted.
    */
   if (v.size() <= 1) {
      return;
   }

   /* Split v into two subvectors. */
   int half = v.size() / 2;
   Vector<int> left  = v.subList(0, half);
   Vector<int> right = v.subList(half);

   /* Recursively sort these arrays. */
   mergesort(left);
   mergesort(right);

   /* Combine them together. */
   merge(left, right, v);
}
```

Time-Out for Announcements!

Midterm Exam Logistics

- Our midterm exam will be on ***Monday, February 10th*** from ***7:00PM – 10:00PM***. Locations are assigned by last (family) name:
  - `A – H`: Go to Bishop Auditorium.
  - `I – S`: Go to Hewlett 200.
  - `T – Z`: Go to Hewlett 201.
- Exam format:
  - The exam covers L00 – L09 (basic C++ up through but not including recursive backtracking) and A0 – A3 (debugging through recursion).
  - It's a traditional sit-down, pencil-and-paper exam.
  - It's closed-book, closed-computer, and limited-note. You can bring an 8.5" × 11" sheet of notes with you to the exam. We will prove a syntax reference sheet for container types; it's up on the course website.
- We've posted a searchable bank of practice problems to the course website, along with three practice exams made of questions from that bank.
- We have reached out to everyone who we believe is taking an alternate exam with location / time info. If you haven't heard from us and you think you're taking an alternate exam, contact us ASAP.

Midterm Review Session

- The amazing SL team will be holding a midterm review session later today:

  5:00PM – 7:00PM

  Room 380-380C

- Come with questions, leave with answers!
- Slides and video will be posted, but you will get more out of this if you attend in person.

The Importance of Practice

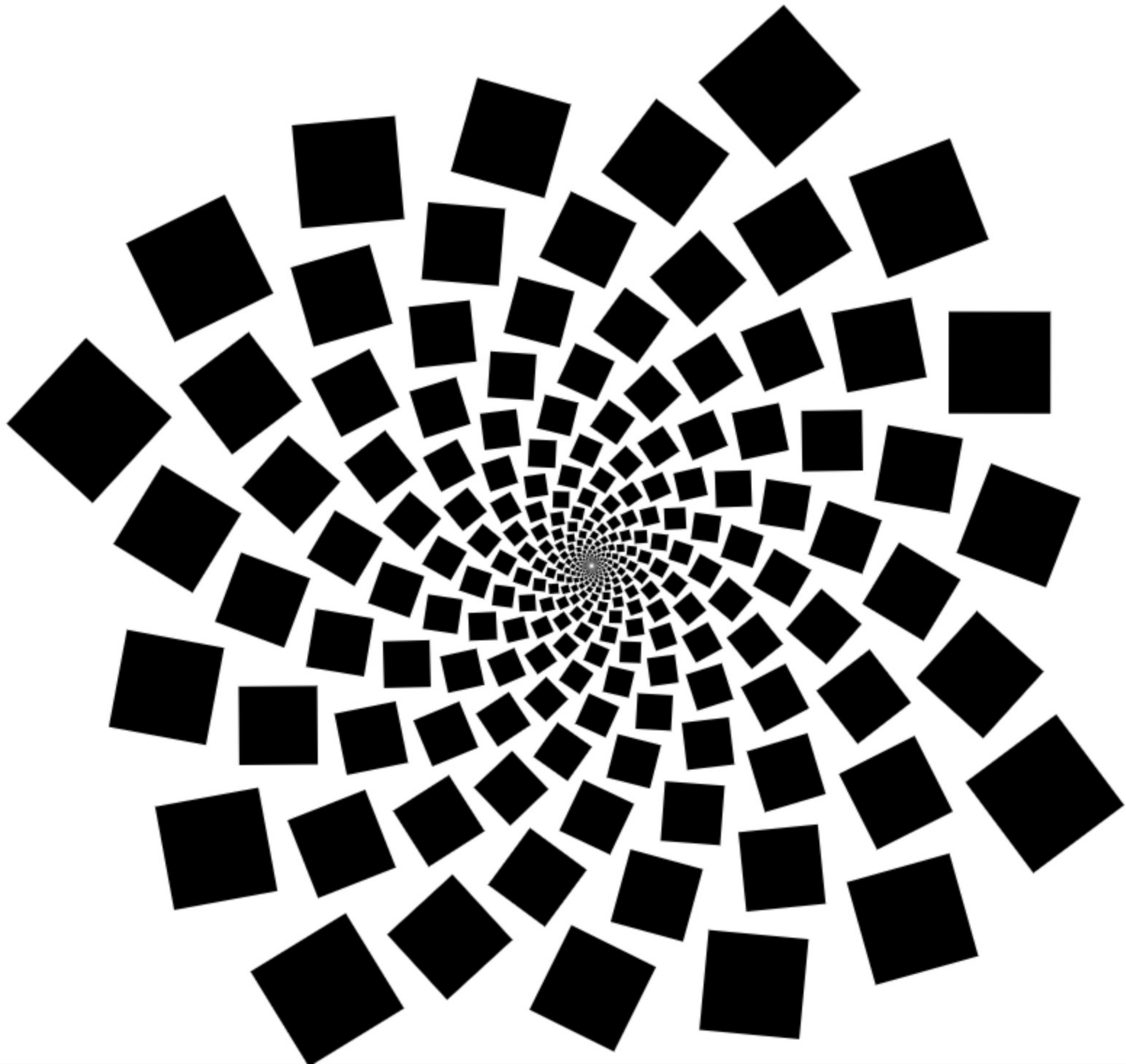
- The best way to prepare for the CS106B midterm is to work through practice problems.
  - Reading the textbook and slides help, but that's not sufficient on its own.
- ***Our Advice:*** If you get stuck on a problem, don't just look at the answer and say "oh, that's how you do it." Instead, go to the LaIR, post on EdStem, or ask your SL for help.
- If you do look at our solutions, and you see that we did something differently than you, don't move on until you understand what's different and whether it matters. Ask us if you aren't sure!

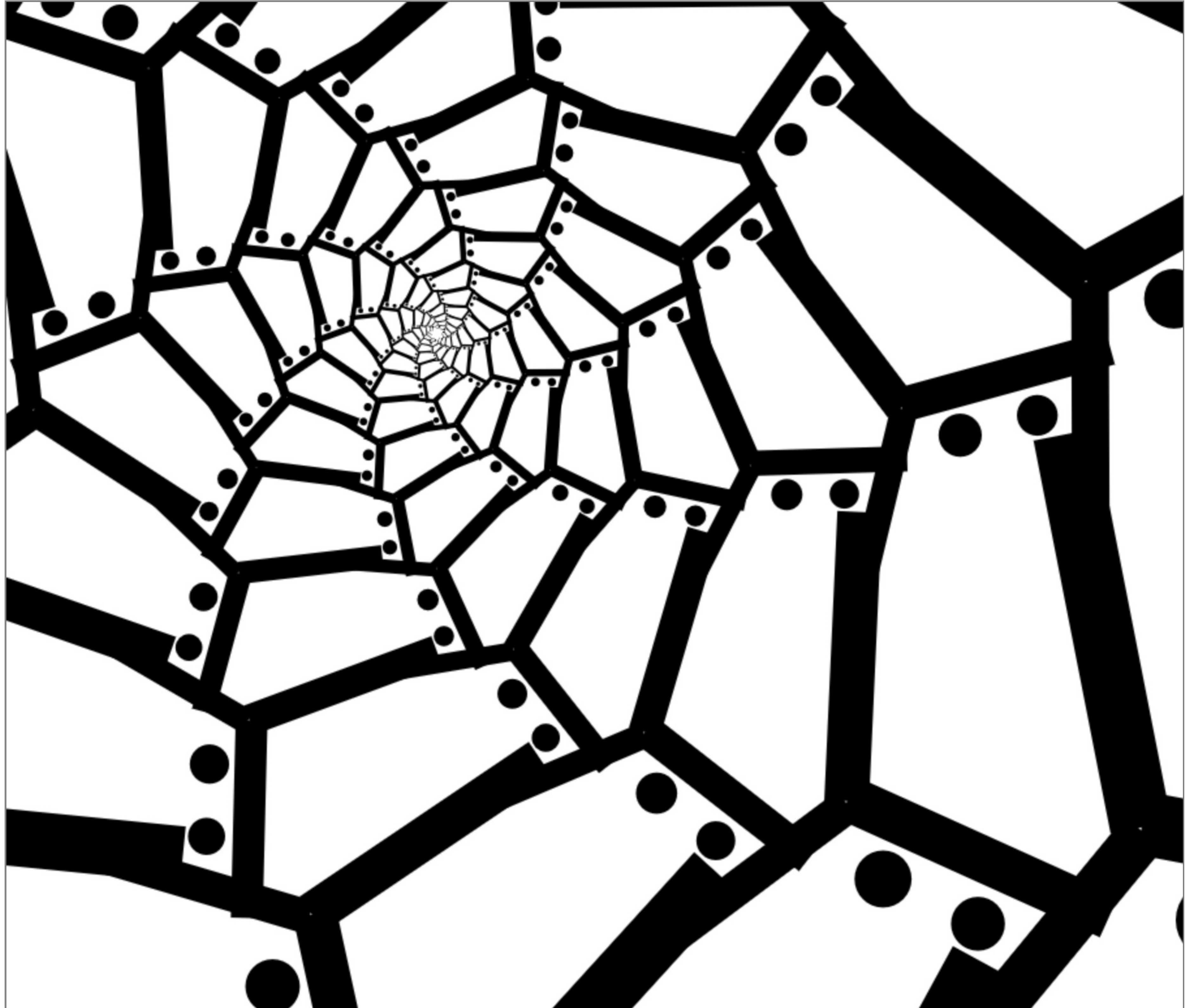
Recursive Drawing Prizes

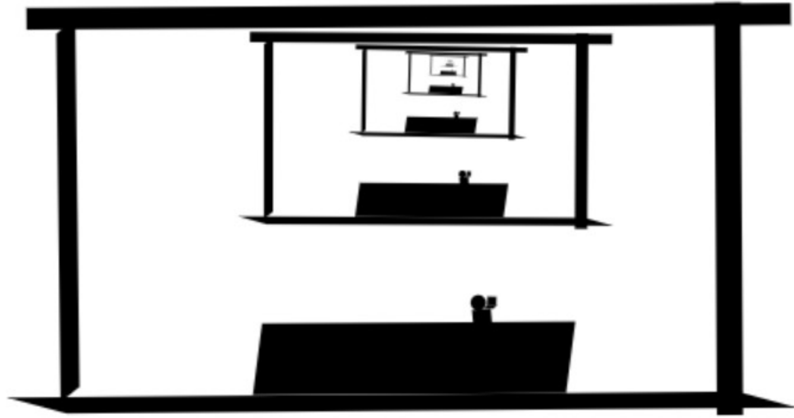
Recursive Cocoa!

- We have five boxes of Droste Cacao that we'll be awarding as prizes.
- We figured it's a nice recursive art prize for our recursive art contest.

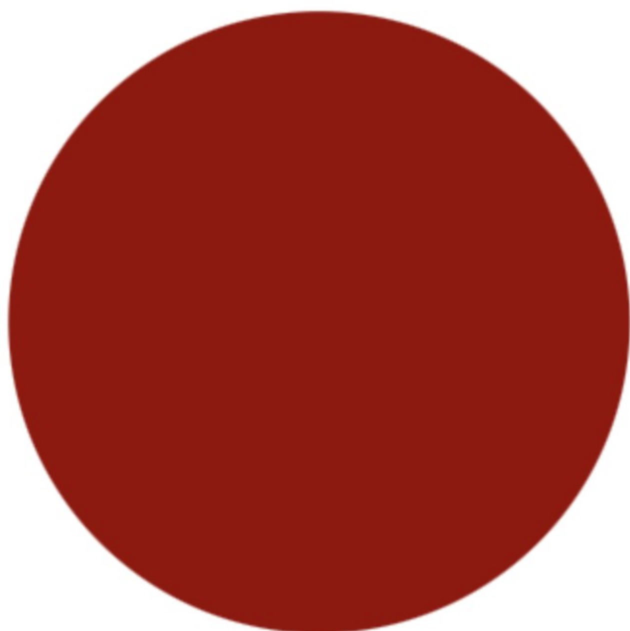
The Awardees







Honorable Mention



```
lecture.notify_all();
```

(A C++ command to wake up parts of the program that are sleeping and waiting for a signal to continue.)

How fast is mergesort?

This next section is the mathiest math
we’re going to math all quarter.

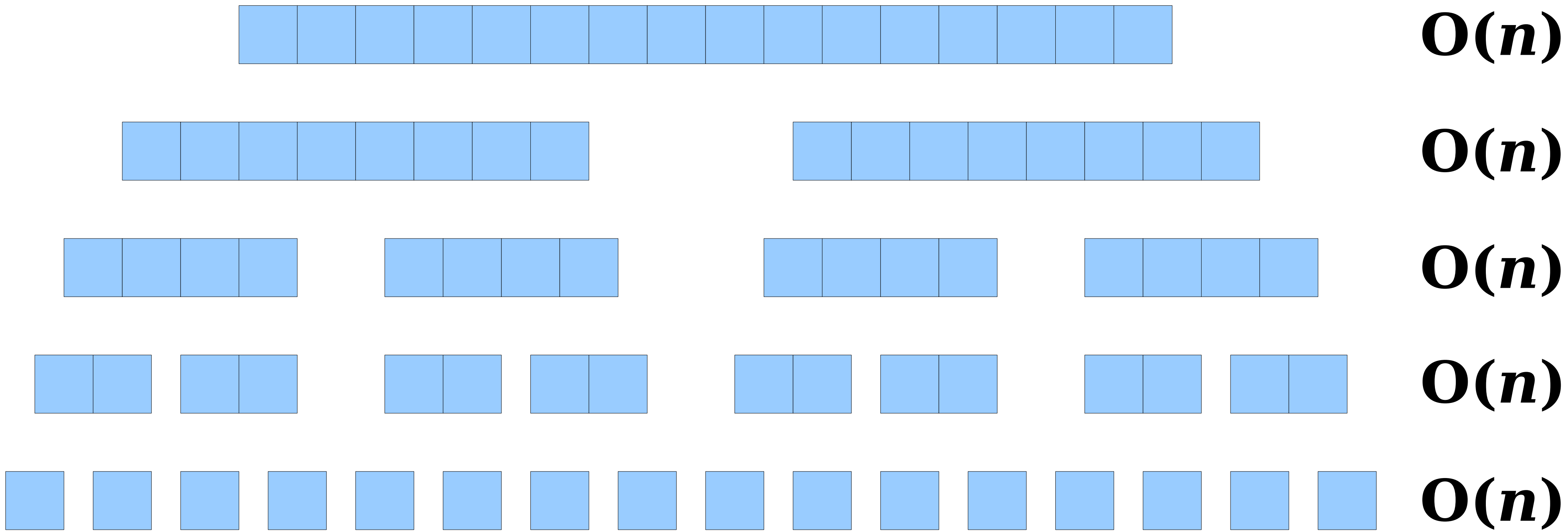
It’s great if you can follow along with it.

You aren’t expected to come up with
this on your own.

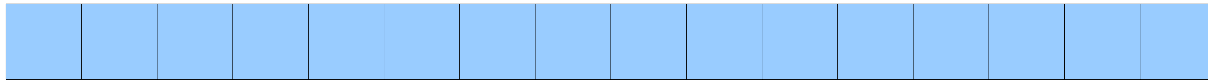
If you like this analysis, take CS161!

```
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) {
        return;
    }

    /* Split v into two subvectors. */
    int half = v.size() / 2;
    Vector<int> left  = v.subList(0, half);
    Vector<int> right = v.subList(half);

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
```

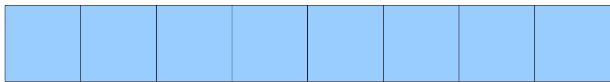
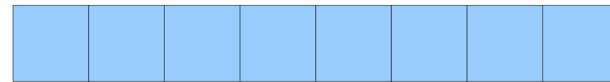
```
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) {
        return;
    }

    /* Split v into two subvectors. */
    int half = v.size() / 2;
    Vector<int> left  = v.subList(0, half);
    Vector<int> right = v.subList(half);

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
```



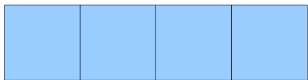
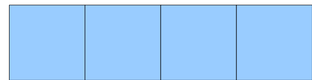
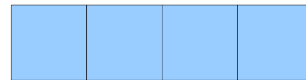
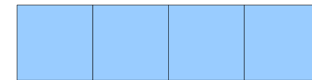
How much work does mergesort do at each level of recursion?



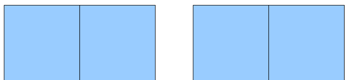
O(*n*)

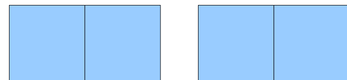
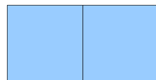
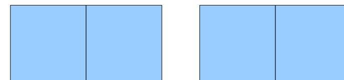
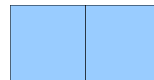
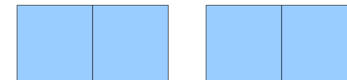



O(*n*)

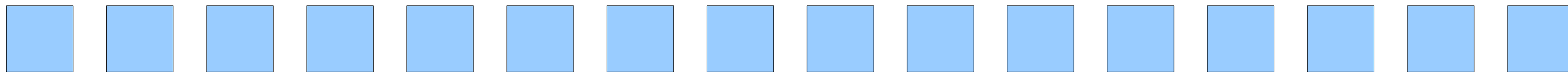





O(*n*)



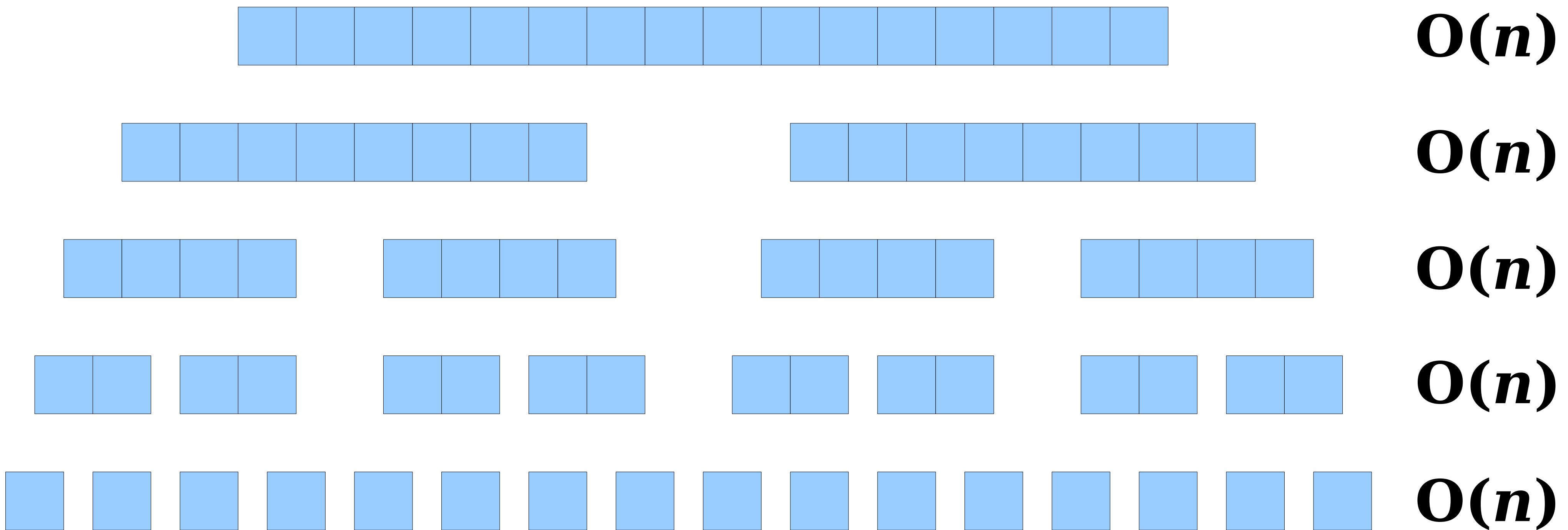
O(*n*)



O(*n*)

Suppose our array has *n* elements. How many levels will there be in the mergesort recursion?

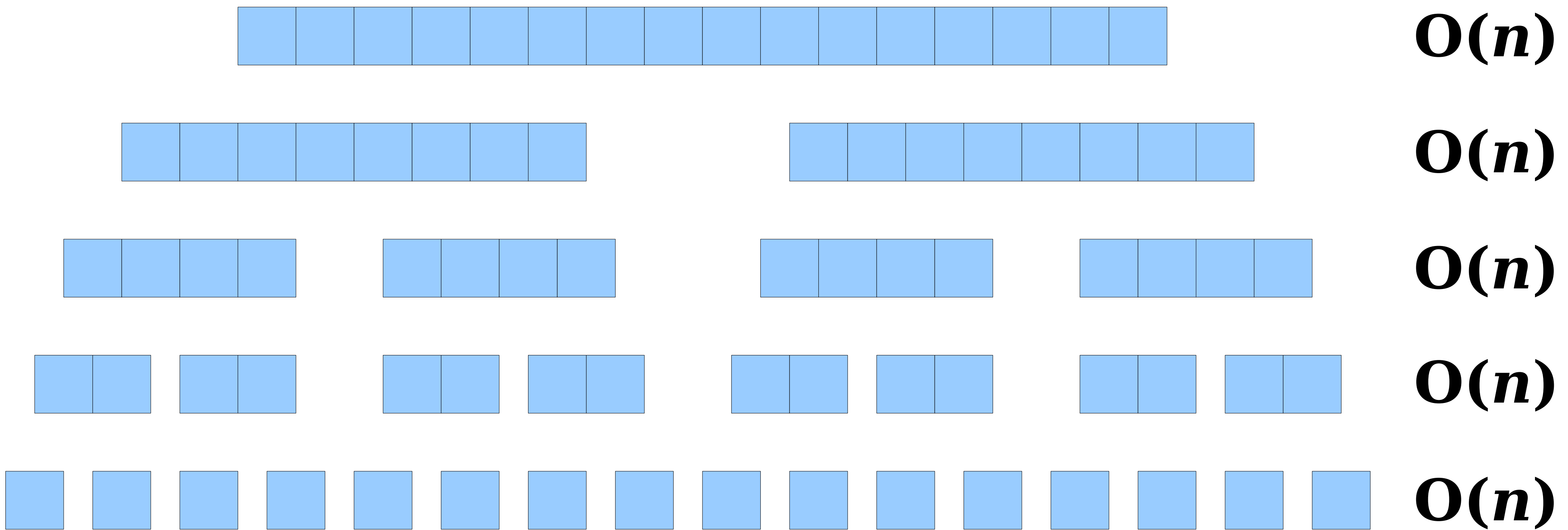
Answer at
https://cs106b.stanford.edu/pollev



Each recursive call cuts the array size in half.

We can only do that O(log *n*) times before we run out of elements in our arrays.

Number of layers: **O(log *n*)**.



There are O(log n) levels in the recursion.

Each level does O(n) work.

Total work done: **O(n log n)**.

What's Up With O(n log n)?

- Recall: log n grows really, really slowly.
  - $\log_2 1{,}000{,}000{,}000 \approx 30$.
- So a runtime of O(n log n)
  - … grows at a *slightly* faster rate than O(n), but
  - … grows at a *much* slower rate than O(n^2).
- That's one of the reasons mergesort runs so quickly on large inputs – it scales much better than selection sort.
- It can be hard to visually tease out a difference between O(n) and O(n log n) in a runtime plot because the O(log n) term grows so slowly.

Can we do Better?

- Mergesort runs in time O(n log n), which is faster than selection sort's O(n^2).
- Can we do better than this?
- A ***comparison sort*** is a sorting algorithm that only learns the relative ordering of its elements by making comparisons between elements.
  - All of the sorting algorithms we've seen so far are comparison sorts.
- ***Theorem:*** There are no comparison sorts whose average-case runtime can be better than O(n log n).
- If we stick with making comparisons, we can only hope to improve on mergesort by a constant factor!

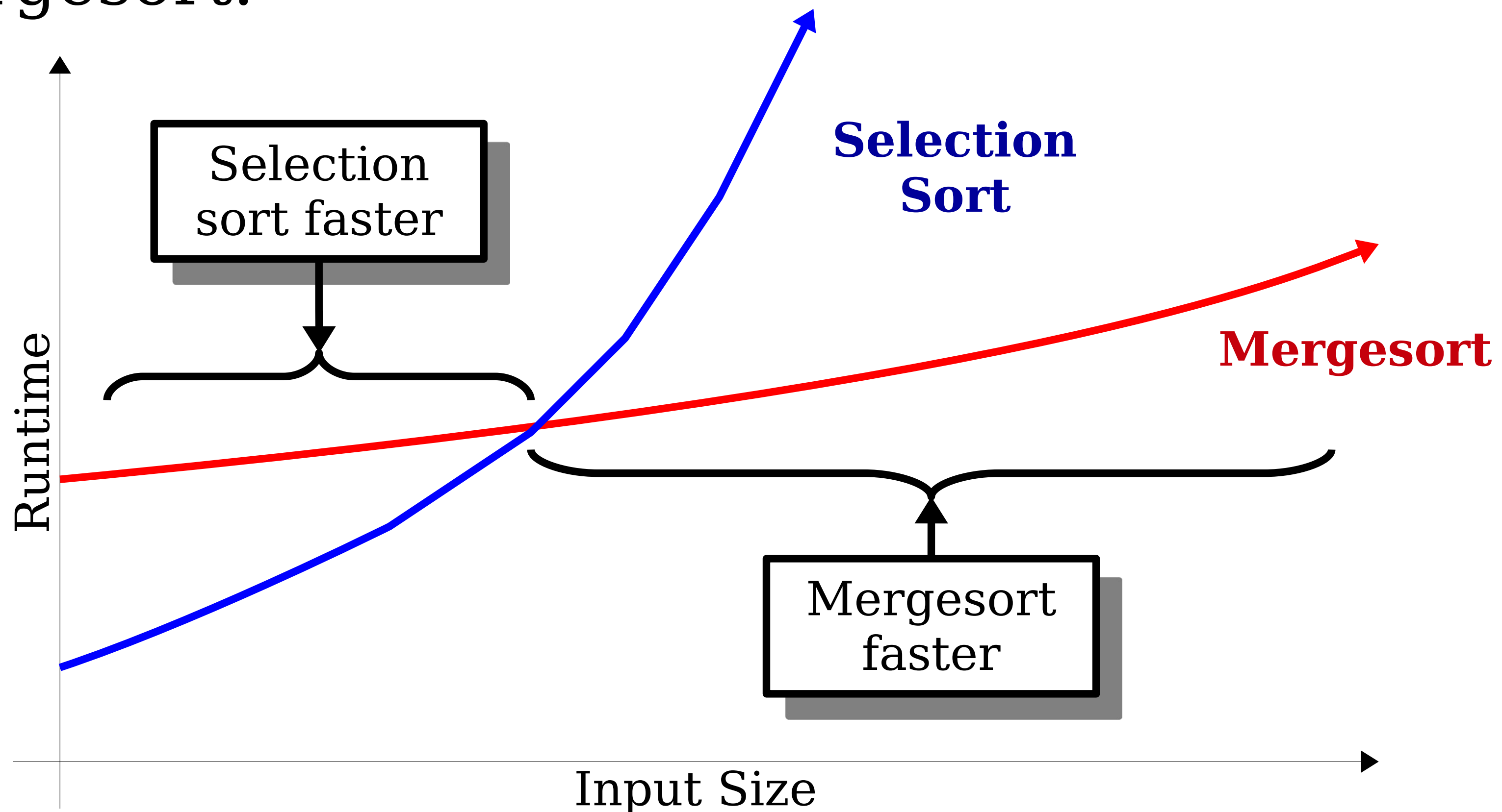
A Quick Historical Aside

- Mergesort was one of the first algorithms developed for computers as we know them today.
- It was invented by John von Neumann in 1945 (!) as a way of validating the design of the first "modern" (stored-program) computer.
- Want to learn more about what he did? Check out ***this article*** by Stanford's very own Donald Knuth.

Improving Mergesort

An Interesting Observation

- Big-O notation talks about long-term growth, but says nothing about small inputs.
- For small inputs, selection sort can be faster than mergesort.



Hybrid Mergesort

```
void hybridMergesort(Vector<int>& v) {
    if (v.size() <= kCutoffSize) {
        selectionSort(v);
    } else {
        int half = v.size() / 2;
        Vector<int> left  = v.subList(0, half);
        Vector<int> right = v.subList(half);

        hybridMergesort(left);
        hybridMergesort(right);

        merge(left, right, v);
     }
}
```

Why All This Matters

- Big-O notation gives us a ***quantitive way*** to predict runtimes.
- Those predictions provide a ***quantitive intuition*** for how to improve our algorithms.
- Understanding the nuances of big-O notation then leads us to design algorithms that are better than the sum of their parts.
- We can use ***binary search*** to look inside sorted sequences really, really quickly.

Your Action Items

- ***Read Chapter 10 of the textbook.***
  - It’s all about big-O and sorting.
- ***Finish Assignment 4.***
  - We’re here for you if you need help!
- ***Study for the Midterm***
  - Review old assignments, do practice exams, etc.

Next Time

- ***Designing Abstractions***
  - How do you build new container classes?
- ***Class Design***
  - What do classes look like in C++?