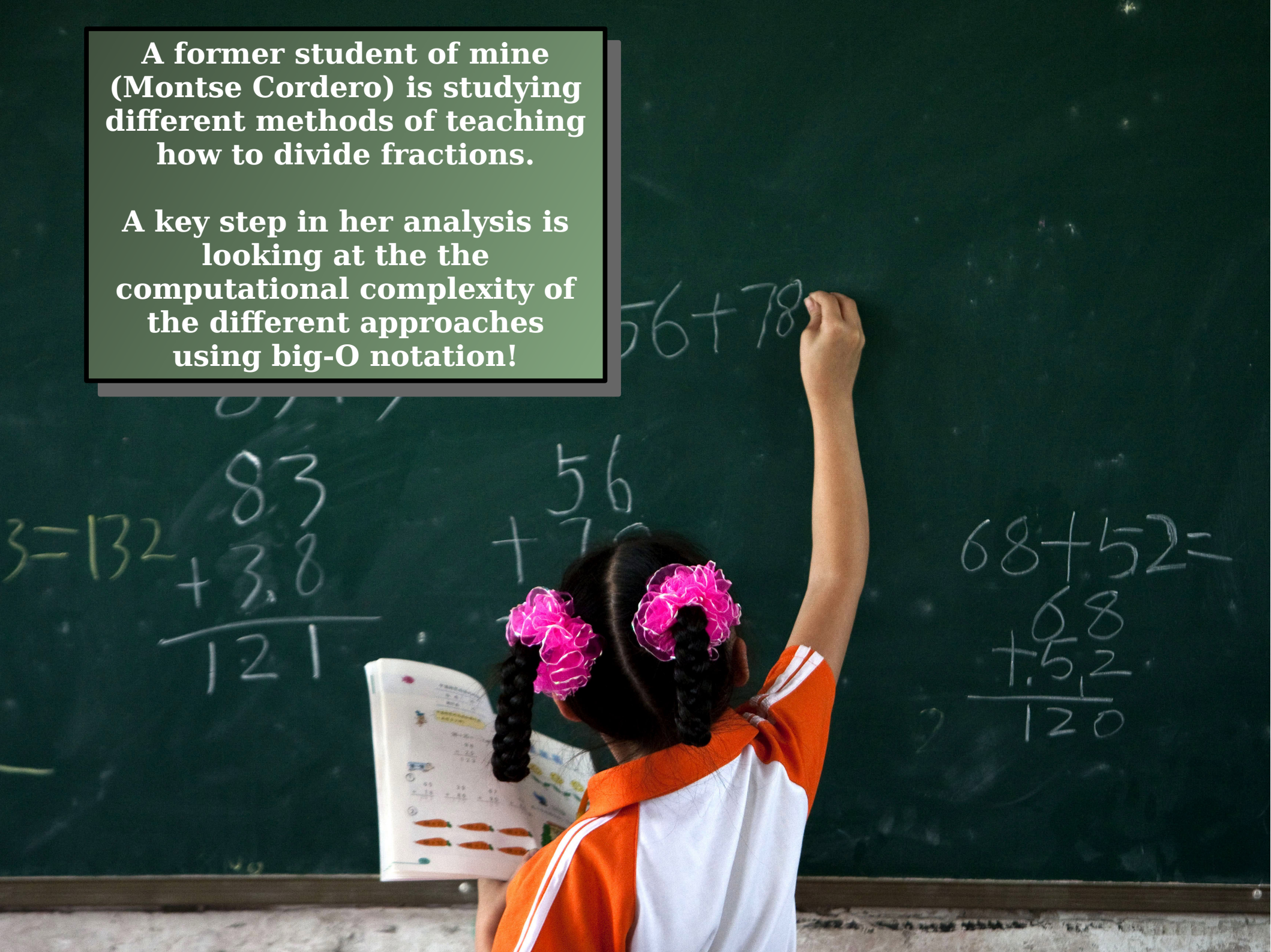


Algorithmic Analysis and Sorting

Part Two

A former student of mine (Montse Cordero) is studying different methods of teaching how to divide fractions.

A key step in her analysis is looking at the the computational complexity of the different approaches using big-O notation!



Recap from Last Time

Big-O Notation

- **Big-O notation** is a quantitative way to describe the runtime of a piece of code.
- Works by dropping constants and low-order growth terms.
- For example, this code runs in time **$O(n)$** :

```
for (int i = 0; i < vec.size(); i++) {  
    cout << vec[i] << endl;  
}
```

Big-O Notation

- **Big-O notation** is a quantitative way to describe the runtime of a piece of code.
- Works by dropping constants and low-order growth terms.
- For example, this code runs in time **$O(n^2)$** :

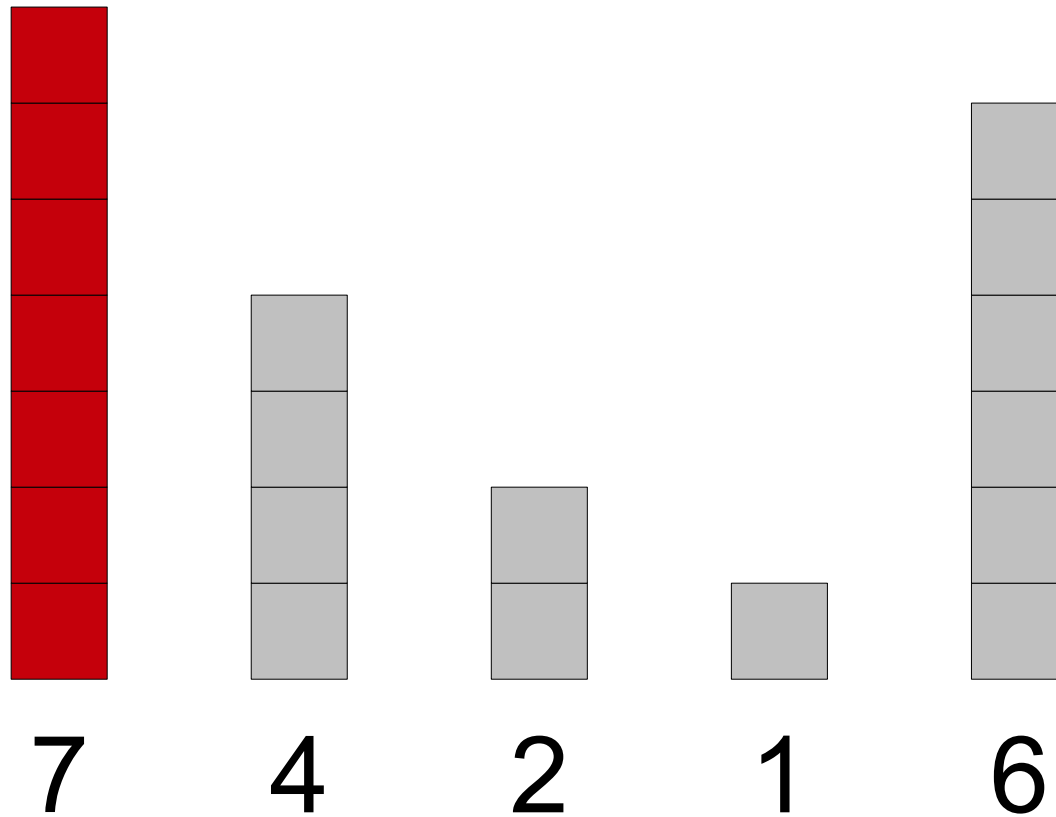
```
for (int i = 0; i < vec.size(); i++) {  
    for (int j = 0; j < vec.size(); j++) {  
        cout << (vec[i] + vec[j]) << endl;  
    }  
}
```

Sorting Algorithms

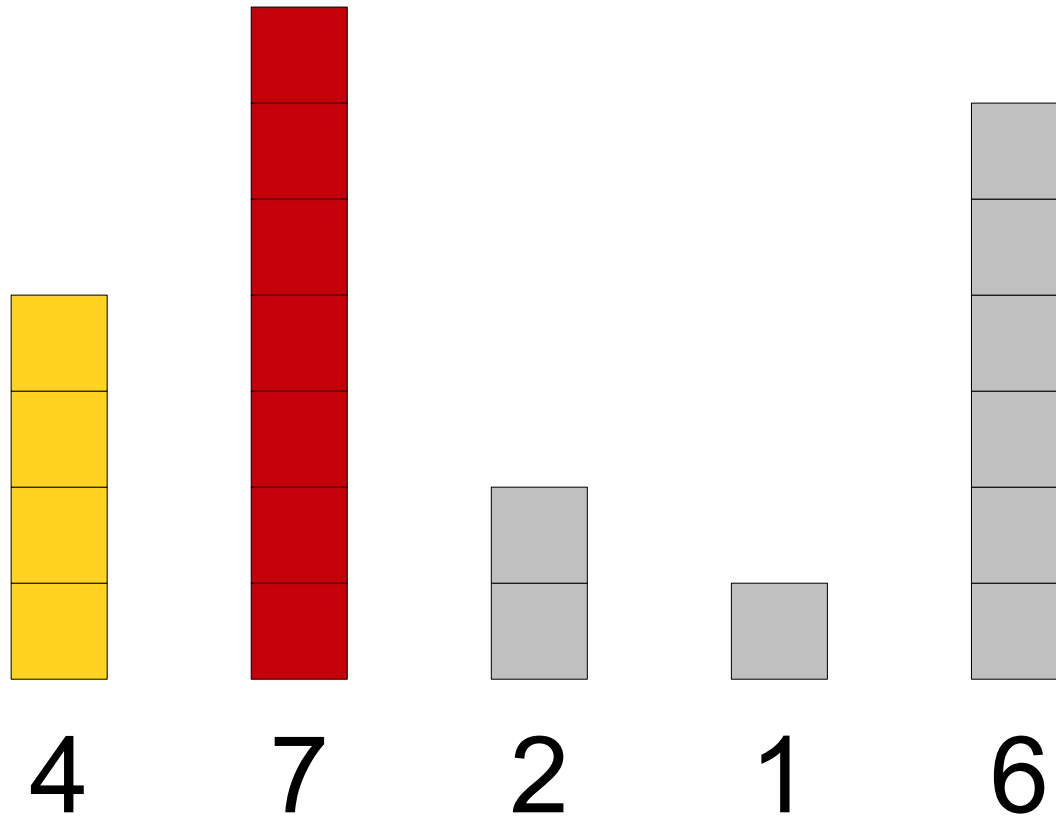
- The ***sorting problem*** is to take in a list of things (integers, strings, etc.) and rearrange them into sorted order.
- Last time, we saw ***selection sort***, an algorithm that runs in time $O(n^2)$.
- This means that doubling the objects of elements to sort will (roughly) quadruple the time required.
- Our question for today: can we sort numbers faster than this?

New Stuff!

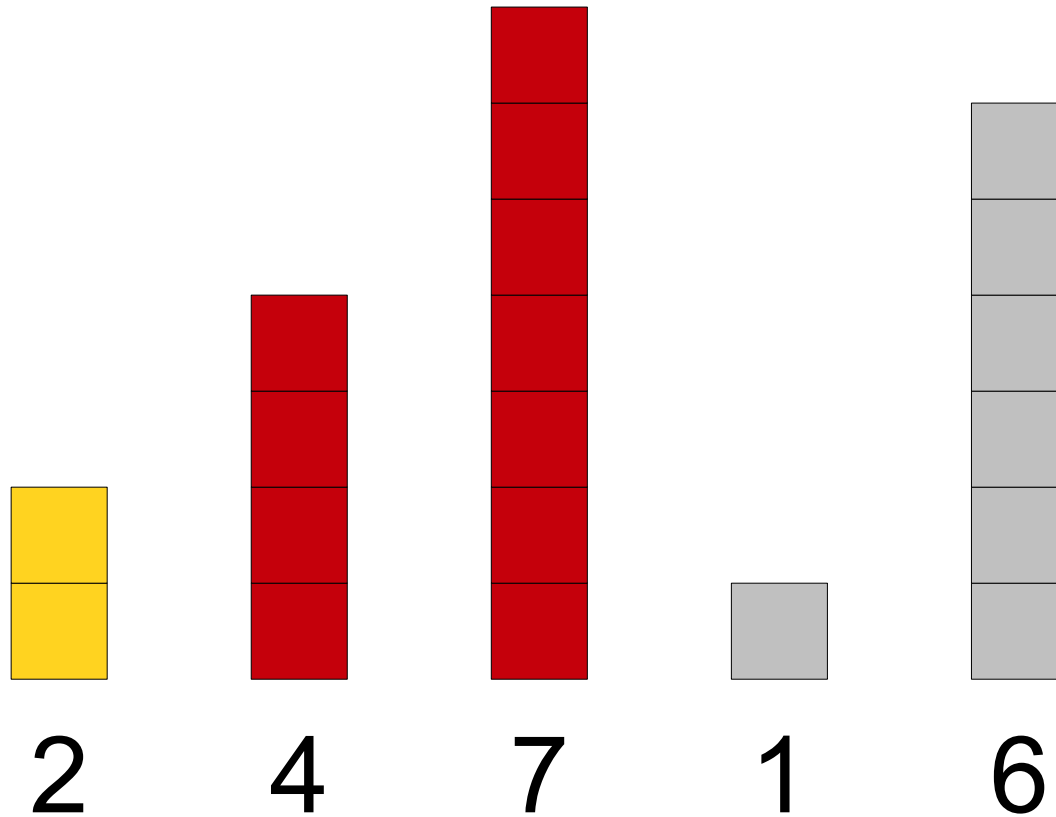
Another Idea: *Insertion Sort*



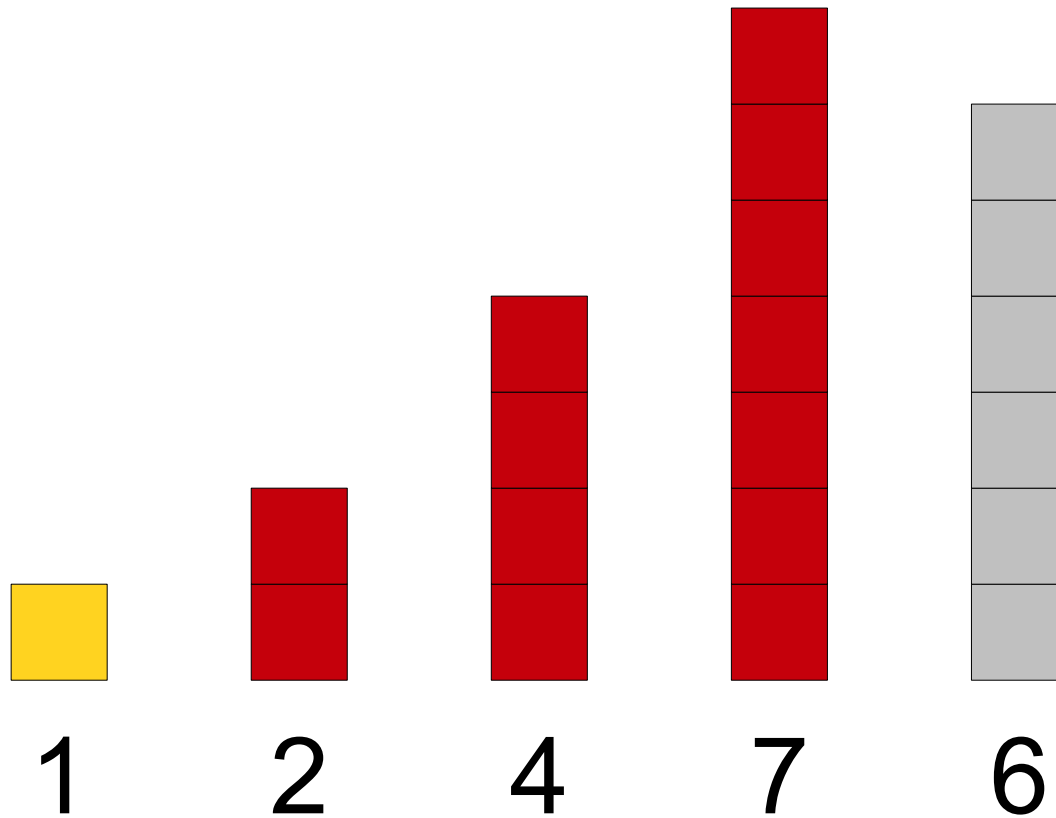
Another Idea: *Insertion Sort*



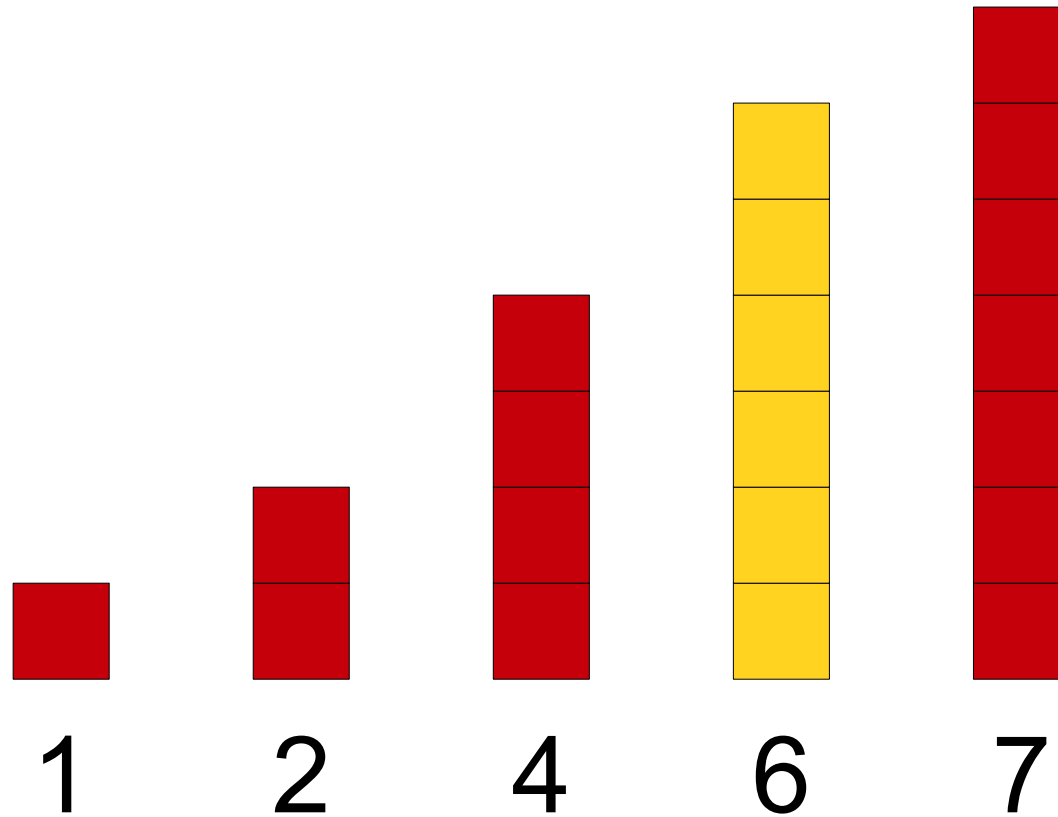
Another Idea: *Insertion Sort*



Another Idea: *Insertion Sort*



Another Idea: *Insertion Sort*



```

/**
 * Sorts the specified vector using insertion sort.
 *
 * @param v The vector to sort.
 */
void insertionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        /* Scan backwards until either (1) there is no
         * preceding element or the preceding element is
         * no bigger than us.
         */
        for (int j = i - 1; j >= 0; j--) {
            if (v[j] <= v[j + 1]) break;

            /* Swap this element back one step. */
            swap(v[j], v[j + 1]);
        }
    }
}

```

Selection Sort vs Insertion Sort

	Size	Selection Sort	Insertion Sort
	10000	0.304	0.160
	20000	1.218	0.630
	30000	2.790	1.427
	40000	4.646	2.520
	50000	7.395	4.181
	60000	10.584	5.635
	70000	14.149	8.143
	80000	18.674	10.333
	90000	23.165	12.832

Of insertion sort and selection sort:

1. Which algorithm runs does more work at the start?
2. Which algorithm runs does more work at the end?

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
---	---	---	---	---	----	----	----

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

1	4	5	8	10	11	12	13
---	---	---	---	----	----	----	----

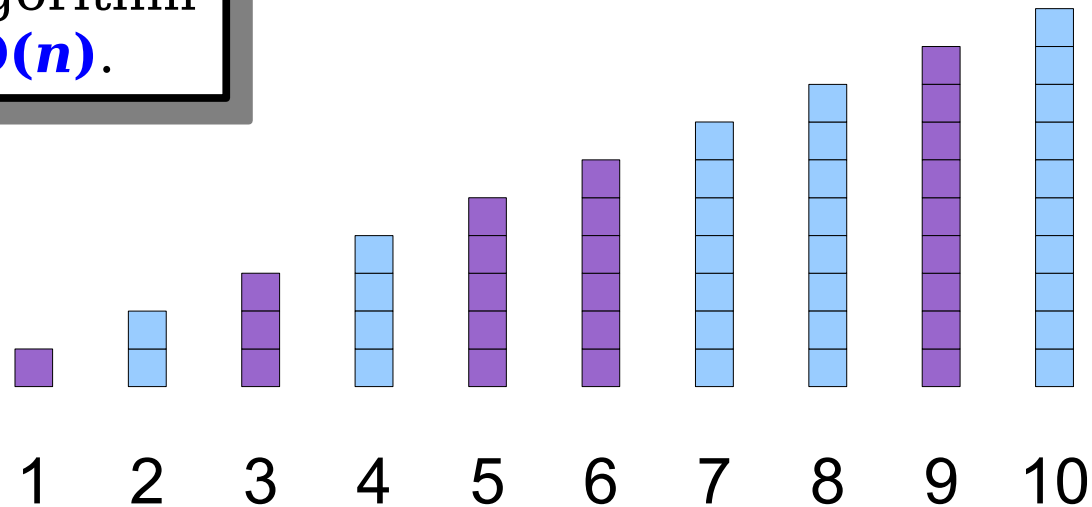
$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

$$(n / 2)^2 = n^2 / 4$$

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 $O(n)$.



```

/**
 * Given two queues of elements in sorted order, merges them together
 * into a single sorted sequence.
 *
 * This can easily be adapted to work with Vectors or other types of
 * sequences, but I thought it was easiest using Queues.
 *
 * @param one The first sorted queue.
 * @param two The second sorted queue.
 * @return A single sorted queue holding all elements of one and two.
 */
Queue<int> merge(Queue<int>& one, Queue<int>& two) {
    Queue<int> result;

    /* Keep comparing the first elements of each queue against one
     * another until one queue is exhausted.
     */
    while (!one.isEmpty() && !two.isEmpty()) {
        if (one.peek() < two.peek()) result.enqueue(one.dequeue());
        else result.enqueue(two.dequeue());
    }

    /* Add all remaining elements of the other queue to the result. */
    while (!one.isEmpty()) result.enqueue(one.dequeue());
    while (!two.isEmpty()) result.enqueue(two.dequeue());

    return result;
}

```

“Split Sort”

```
void splitSort(Vector<int>& v) {  
    /* Split the vector in half */  
    Vector<int> left, right;  
    for (int i = 0; i < v.size() / 2; i++) {  
        left += v[i];  
    }  
    for (int j = v.size() / 2; j < v.size(); j++) {  
        right += v[j];  
    }  
  
    /* Sort each half. */  
    insertionSort(left);  
    insertionSort(right);  
  
    /* Merge them back together. */  
    merge(left, right, v);  
}
```

Performance Comparison

Size	Selection Sort	Insertion Sort	“Split Sort”
10000	0.304	0.160	0.161
20000	1.218	0.630	0.387
30000	2.790	1.427	0.726
40000	4.646	2.520	1.285
50000	7.395	4.181	2.719
60000	10.584	5.635	2.897
70000	14.149	8.143	3.939
80000	18.674	10.333	5.079
90000	23.165	12.832	6.375

Time-Out for Announcements!

Assignment 4

- ***Reminder:*** Assignment 4 is due one week from Friday.
 - Following the assignment timetable we suggested, you should try to be done with the Doctors Without Orders problem by the end of the evening and start working on Disaster Planning.
 - Again, remember that the midterm is coming up right after the assignment due date, so this is probably not a good place to use late days.

Practice Midterm Exam

- We'll be holding a practice midterm exam on ***Monday, February 13th*** from ***7PM - 10PM***, in ***Hewlett 200***.
- It's held under realistic conditions and is a fantastic way to prepare for the exam. ***You should plan to attend this unless you have a compelling reason not to do so.***
- We'll have more details about the midterm exam on Friday.

Honor Code Reminder

- The good news is that the *overwhelming majority* of you are honest and hardworking.
- The bad news is that some of you turned in Assignment 2 submissions that are clearly not your own work.
- ***We take the Honor Code seriously in this course.*** We'll be writing up these cases and submitting them to the Office of Community Standards.
 - Standard sanction for a first offense is many hours of community service, an academic integrity seminar, and a possible suspension.
- Please ask for help if you need it. That's what we're here for!

More Assorted Sorts of Sorts!

A Better Idea

- We can speed up insertion sort by almost a factor of two by splitting the array in half, sorting each part independently, and merging the results together.
- So why not split into fourths? That would give a $4\times$ improvement.
- So why not split into eighths? That would give an $8\times$ improvement.
- **Question:** What happens if we *never stop splitting*?

High-Level Idea

- 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.
- This algorithm is called *mergesort*.

```
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. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++) {
        left += v[i];
    }
    for (int i = v.size() / 2; i < v.size(); i++) {
        right += v[i];
    }

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

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

What is the complexity of mergesort?

```

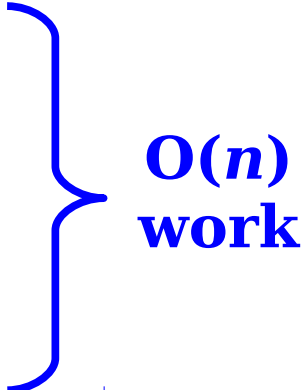
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) return;


    /* Split v into two subvectors. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++) {
        left += v[i];
    }
    for (int i = v.size() / 2; i < v.size(); i++) {
        right += v[i];
    }

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

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

```


**O(n)
work**


**O(n)
work**

```
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. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++) {
        left += v[i];
    }
    for (int i = v.size() / 2; i < v.size(); i++) {
        right += v[i];
    }

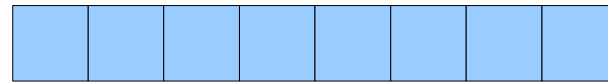
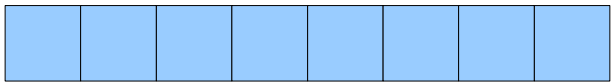
    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

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

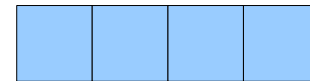
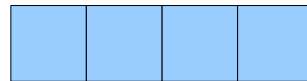
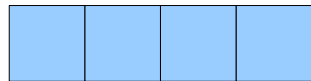
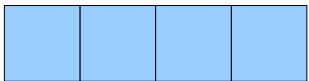
A Graphical Intuition



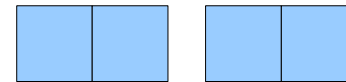
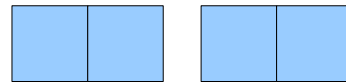
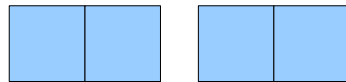
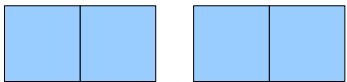
$O(n)$



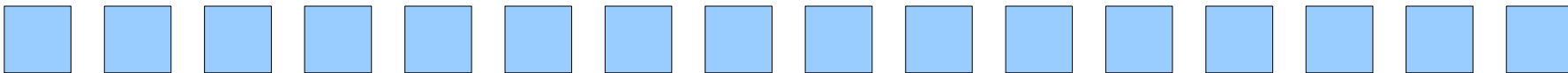
$O(n)$



$O(n)$



$O(n)$



$O(n)$

How many levels are there?

Slicing and Dicing

- After zero recursive calls: n
- After one recursive call: $n / 2$
- After two recursive calls: $n / 4$
- After three recursive calls: $n / 8$
- ...
- After k recursive calls: $n / 2^k$

Cutting in Half

- After k recursive calls, there are $n / 2^k$ elements left.
- Mergesort stops recursing when there are zero or one elements left.
- Solving for the number of levels:

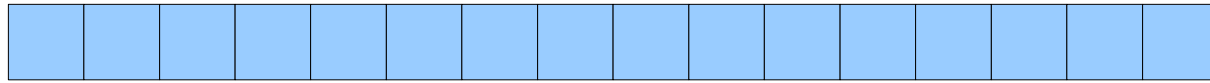
$$n / 2^k = 1$$

$$n = 2^k$$

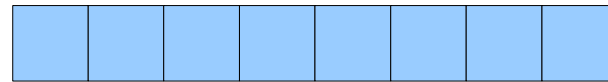
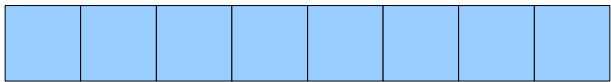
$$\log_2 n = k$$

- So mergesort recurses **$\log_2 n$** levels deep.

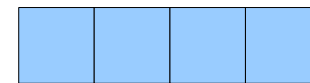
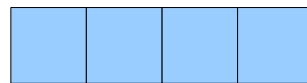
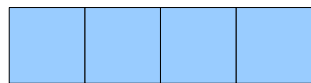
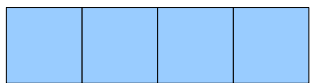
A Graphical Intuition



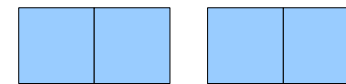
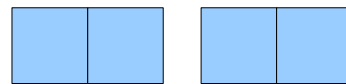
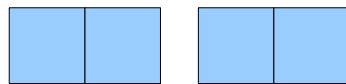
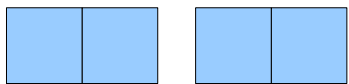
$O(n)$



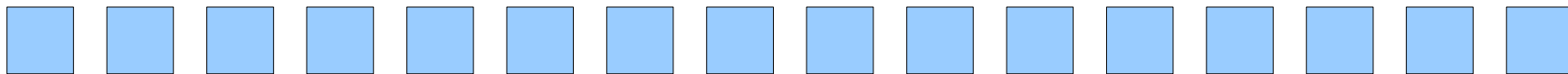
$O(n)$



$O(n)$



$O(n)$



$O(n)$

$O(n \log n)$

Mergesort Times

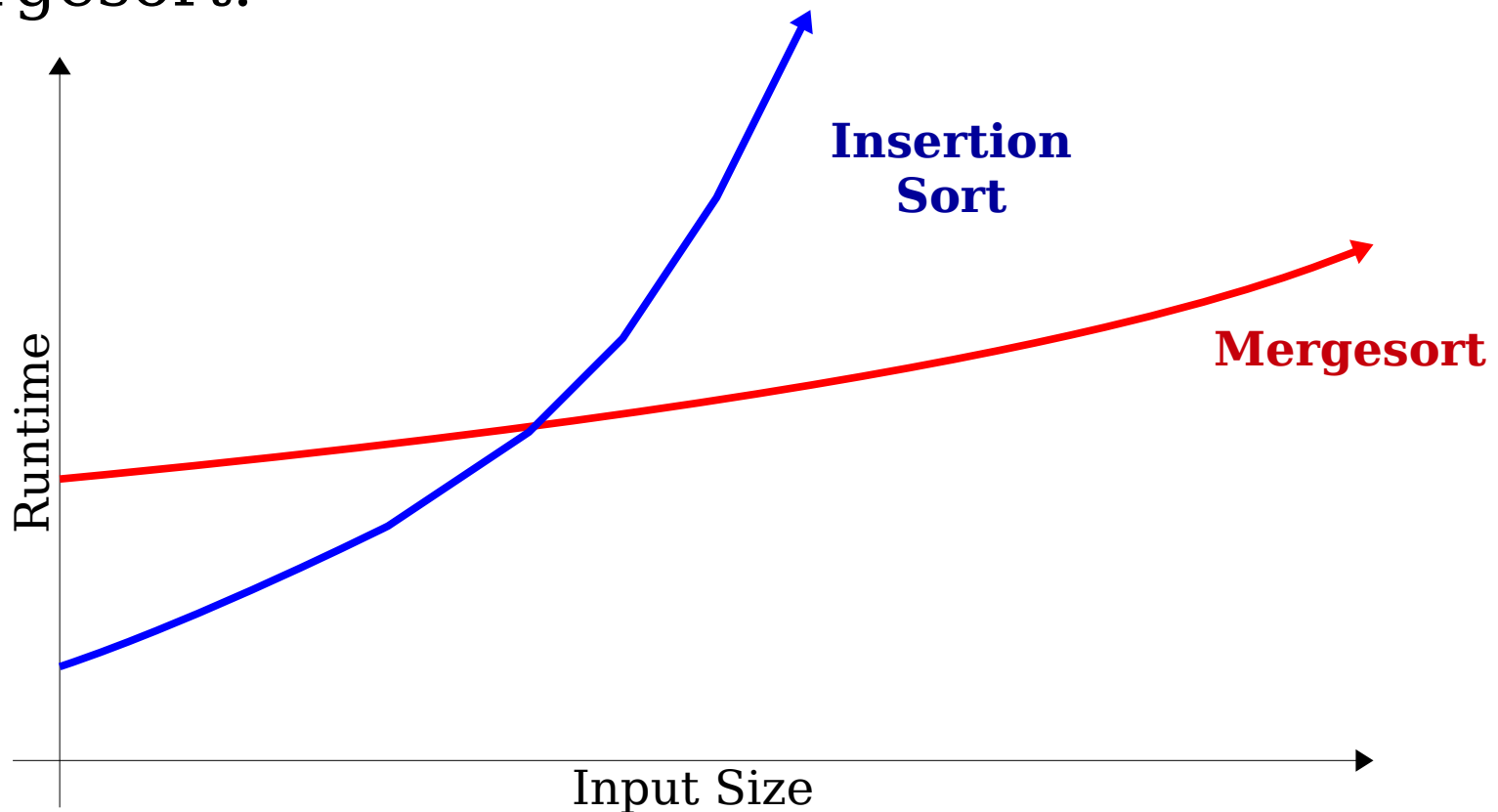
Size	Selection Sort	Insertion Sort	“Split Sort”	Mergesort
10000	0.304	0.160	0.161	0.006
20000	1.218	0.630	0.387	0.010
30000	2.790	1.427	0.726	0.017
40000	4.646	2.520	1.285	0.021
50000	7.395	4.181	2.719	0.028
60000	10.584	5.635	2.897	0.035
70000	14.149	8.143	3.939	0.041
80000	18.674	10.333	5.079	0.042
90000	23.165	12.832	6.375	0.048

Can we do Better?

- Mergesort is $O(n \log n)$, which is faster than insertion sort's $O(n^2)$ runtime.
- Can we do better than this?
 - In general, **no**: comparison-based sorts cannot have a worst-case runtime better than $O(n \log n)$.
- ***In the worst case, we can only get faster by a constant factor!***
- What might that look like?

An Interesting Observation

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



Hybrid Sorting Algorithms

- Modify the mergesort algorithm to switch to insertion sort when the input gets sufficiently small.
- This is called a ***hybrid sorting algorithm***.

Hybrid Mergesort

```
void hybridMergesort(Vector<int>& v) {
    if (v.size() <= kCutoffSize) {
        insertionSort(v);
    } else {
        Vector<int> left, right;
        for (int i = 0; i < v.size() / 2; i++) {
            left += v[i];
        }
        for (int i = v.size() / 2; i < v.size(); i++) {
            right += v[i];
        }

        hybridMergesort(left);
        hybridMergesort(right);

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


Runtime for Hybrid Mergesort

Size	Mergesort	Hybrid Mergesort
100000	0.063	0.019
300000	0.176	0.061
500000	0.283	0.091
700000	0.396	0.130
900000	0.510	0.165
1100000	0.608	0.223
1300000	0.703	0.246
1500000	0.844	0.28
1700000	0.995	0.326
1900000	1.070	0.355

Hybrid Sorts in Practice

- Introspective Sort (*Introsort*)
 - Based on three sorting algorithms: quicksort, heapsort, and insertion sort.
 - Quicksort runs in time $O(n \log n)$ on average, but in the worst case runs in time $O(n^2)$.
 - Heapsort runs in time $O(n \log n)$ and is very memory-efficient. You'll see it on Assignment 5!
 - Uses insertion sort for small inputs.

Runtime for Introsort

Size	Mergesort	Hybrid Mergesort	Introsort
100000	0.063	0.019	0.009
300000	0.176	0.061	0.028
500000	0.283	0.091	0.043
700000	0.396	0.130	0.060
900000	0.510	0.165	0.078
1100000	0.608	0.223	0.092
1300000	0.703	0.246	0.107
1500000	0.844	0.28	0.123
1700000	0.995	0.326	0.139
1900000	1.070	0.355	0.158

Why All This Matters

- Big-O notation gives us a ***quantitative way*** to predict runtimes.
- Those predictions provide a ***quantitative 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.

Next Time

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