Algorithmic Analysis and Sorting

Part Three

Friday Four Square!

4:15PM at Gates Computer Science

Midterm Logistics

- Midterm is next **Tuesday, May 7** from **7PM – 10PM**.
- We are in five different rooms, organized by last name:
  - A – G: Go to **Nvidia Auditorium** (right here!)
  - H – L: Go to **Skilling Auditorium**
  - M – U: Go to **Gates B01** or **Gates B03**
  - V – W: Go to **Huang 018** (next door!)
  - X – Z: Go to **Hewlett 101**
- Review session **Sunday 7PM–9PM** in **320-105**.
- Exam is open-book, open-note, but closed-computer.
- Covers material up through and including Wednesday's lecture.

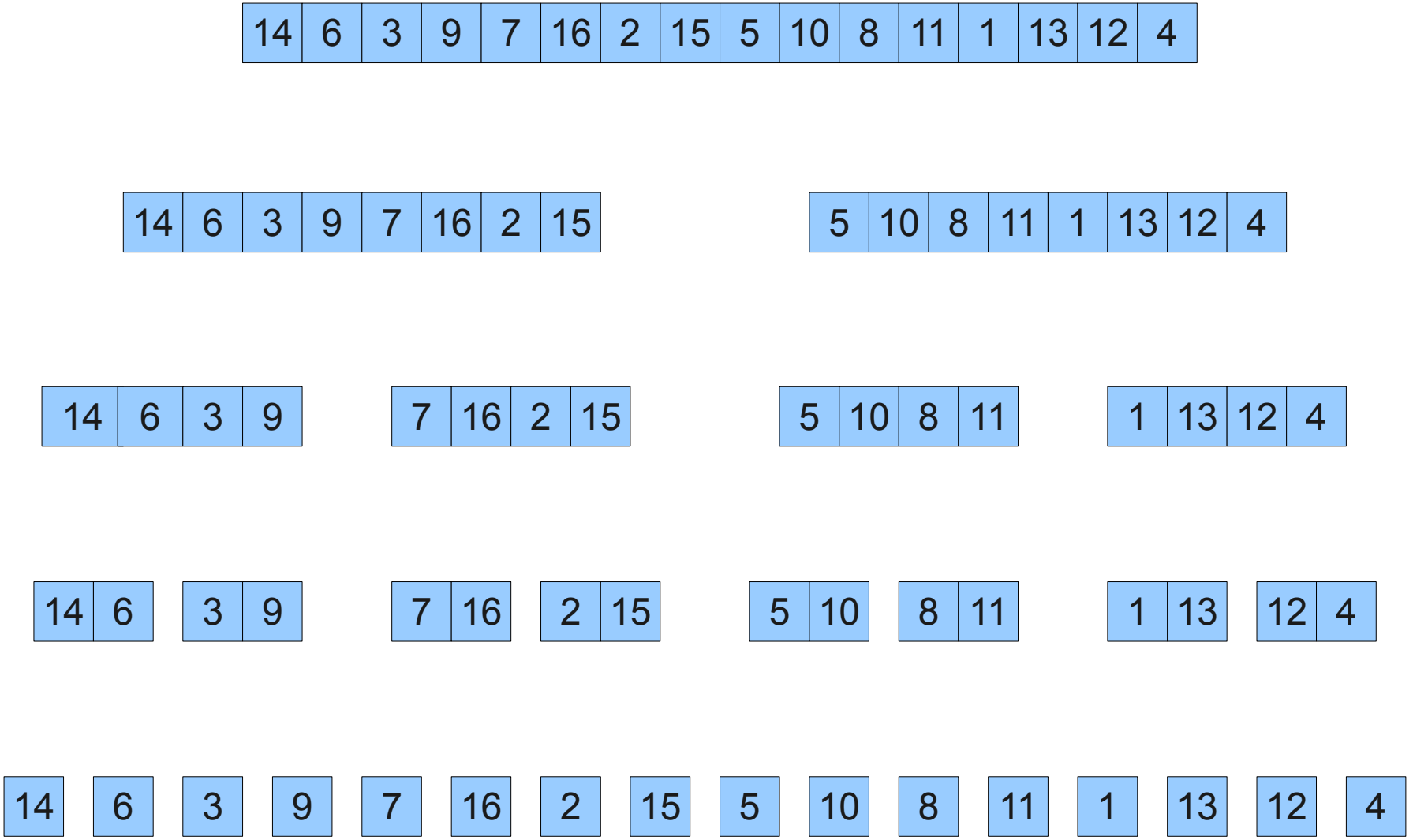
Announcements

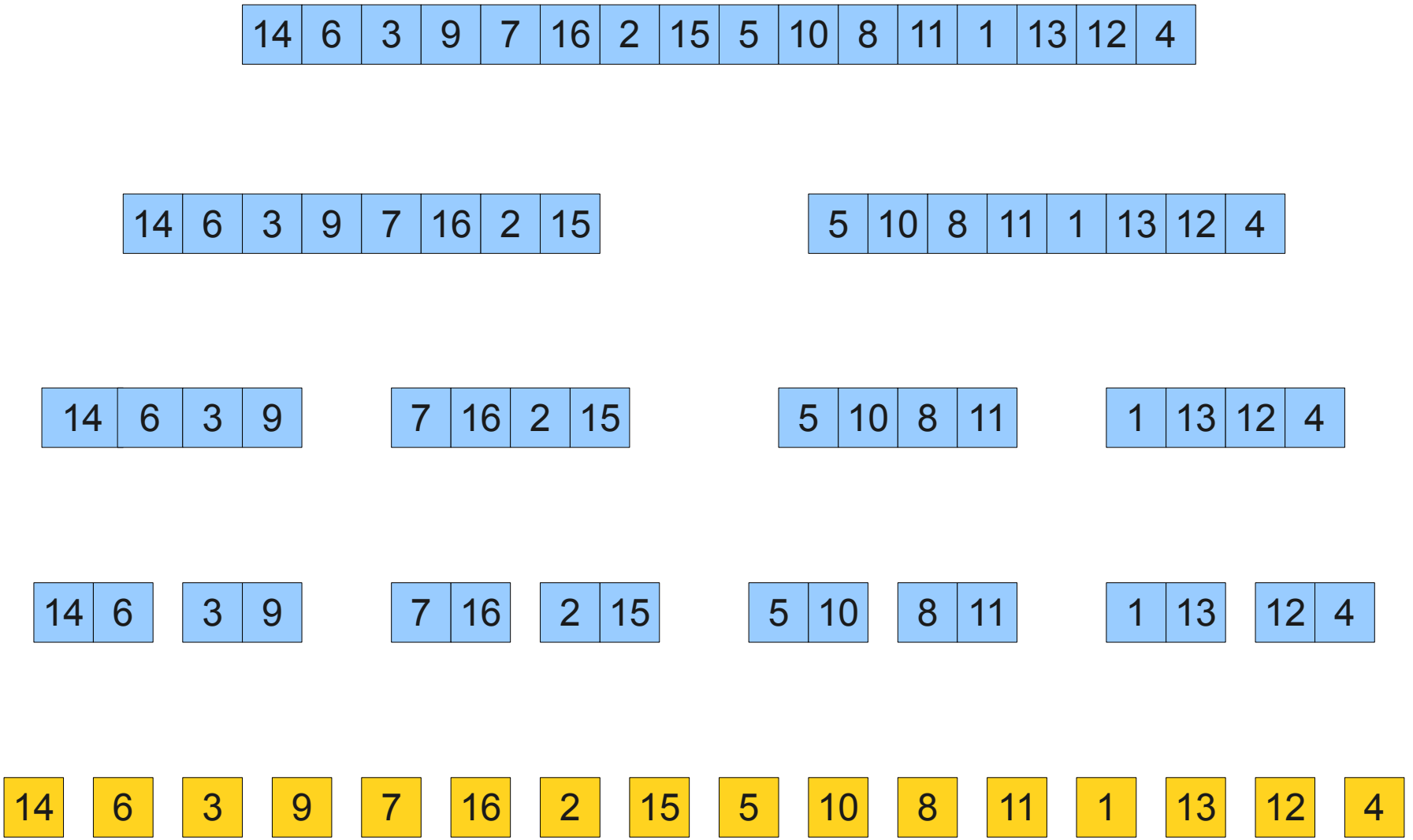
- Assignment 3 due right now.
- Assignment 4 (**Boggle**) out, due Monday, May 13 at 2:15PM.
  - Play around with exhaustive recursion and recursive backtracking!
  - Write a computer program that can beat you at your own game. ☺

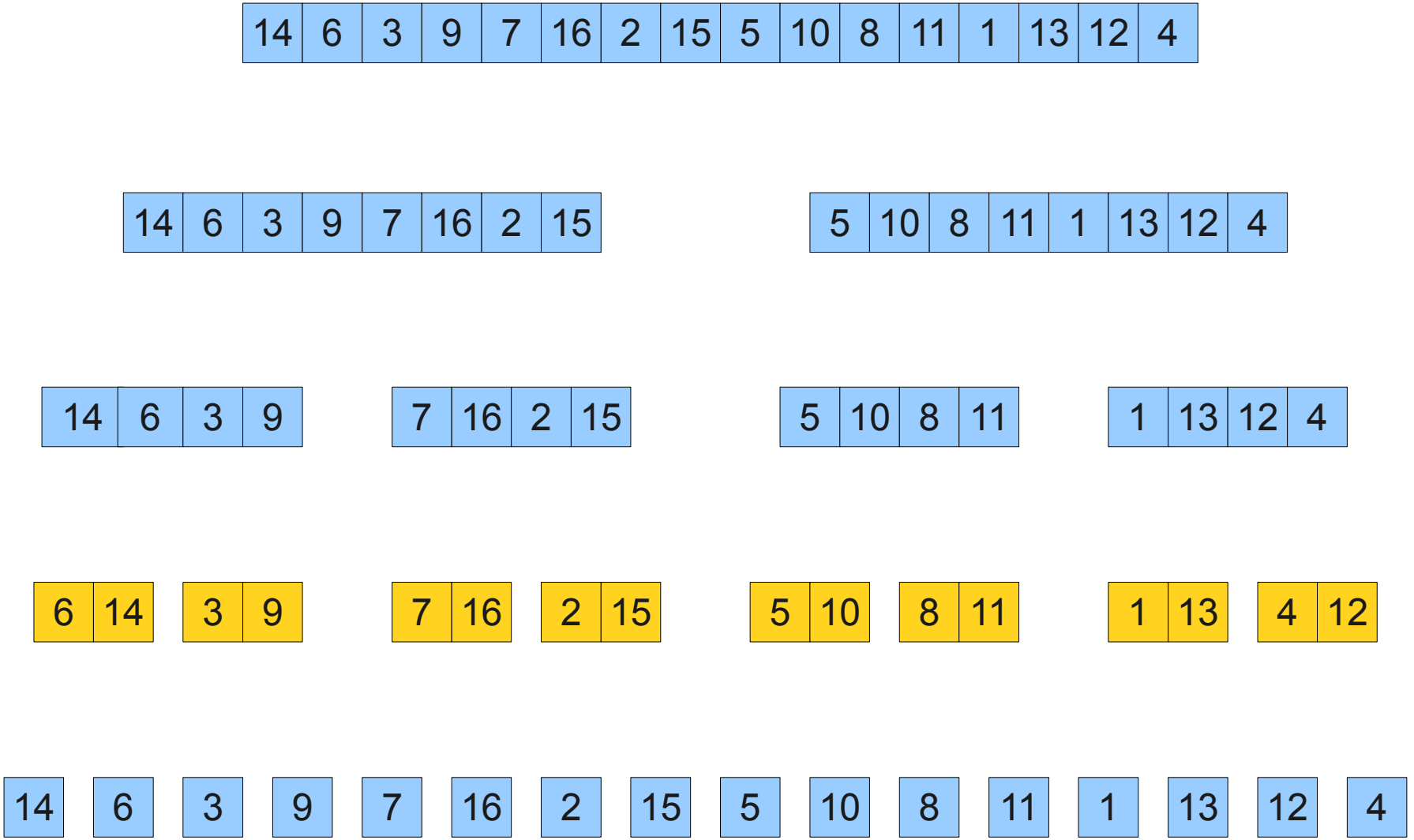
Previously on CS106B...

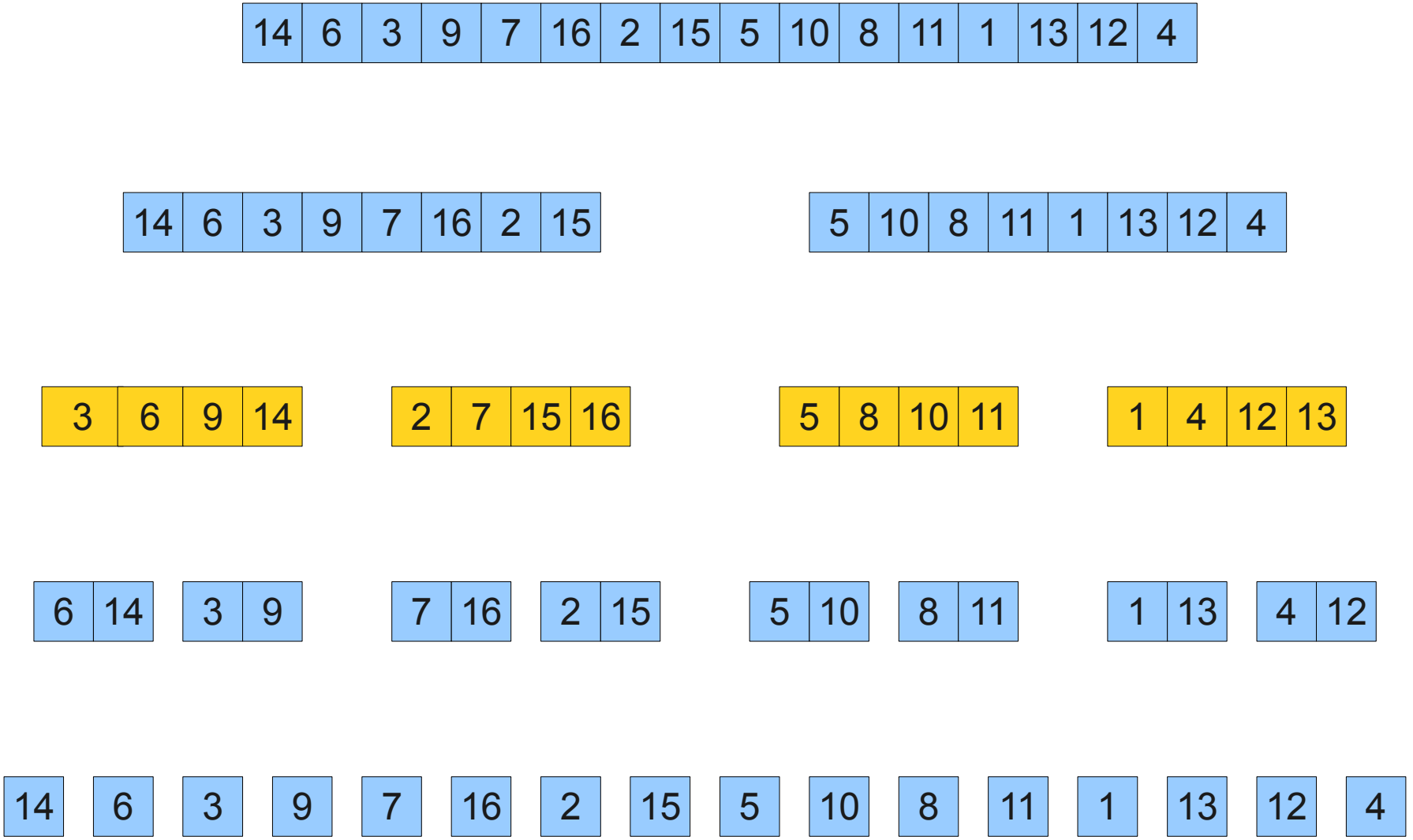
Big-O Notation

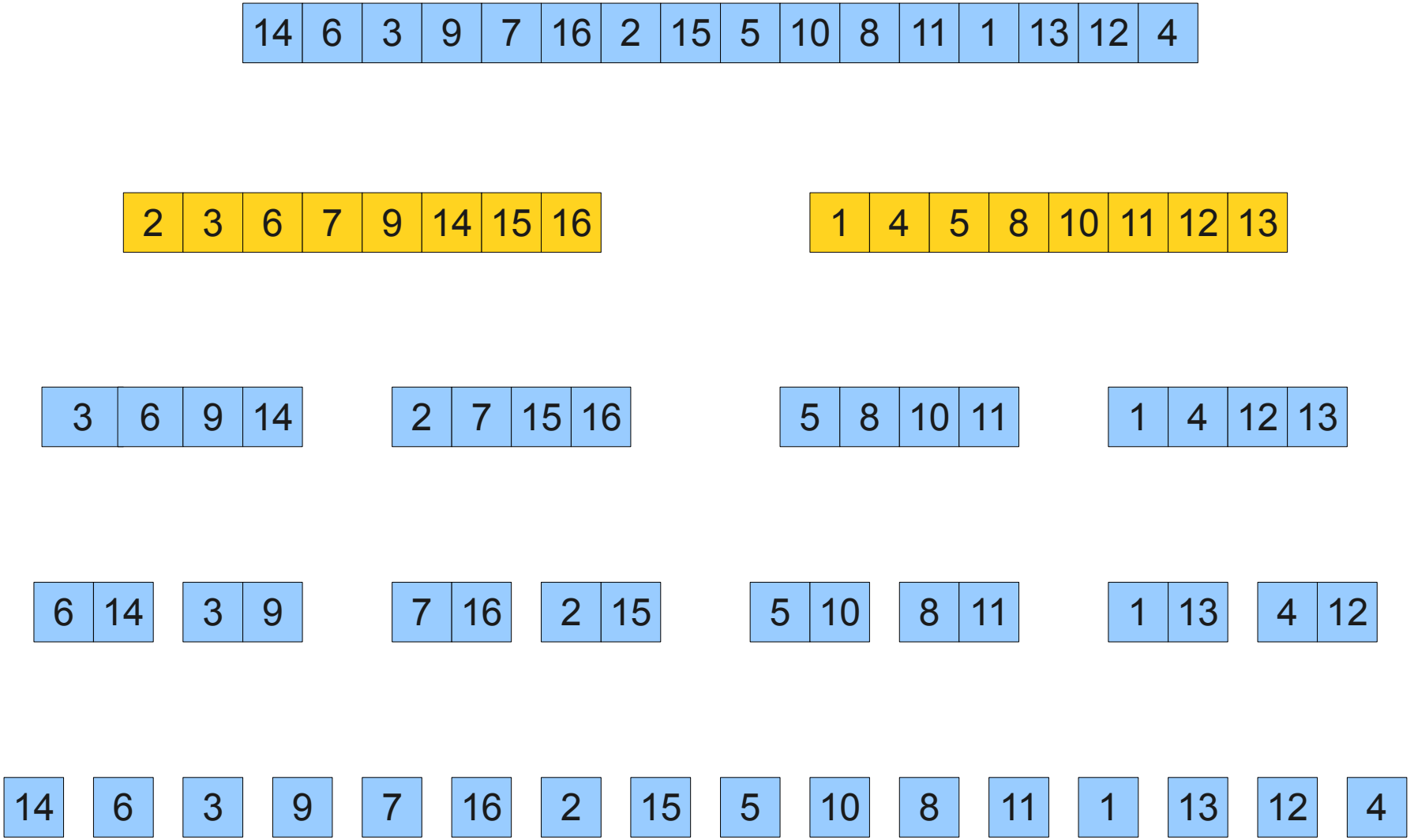
- Notation for summarizing the **long-term growth rate** of some function.
- Useful for analyzing runtime:
  - O(n): The runtime grows linearly.
  - O(n^2): The runtime grows quadratically.
  - O(2^n): The runtime grows exponentially.

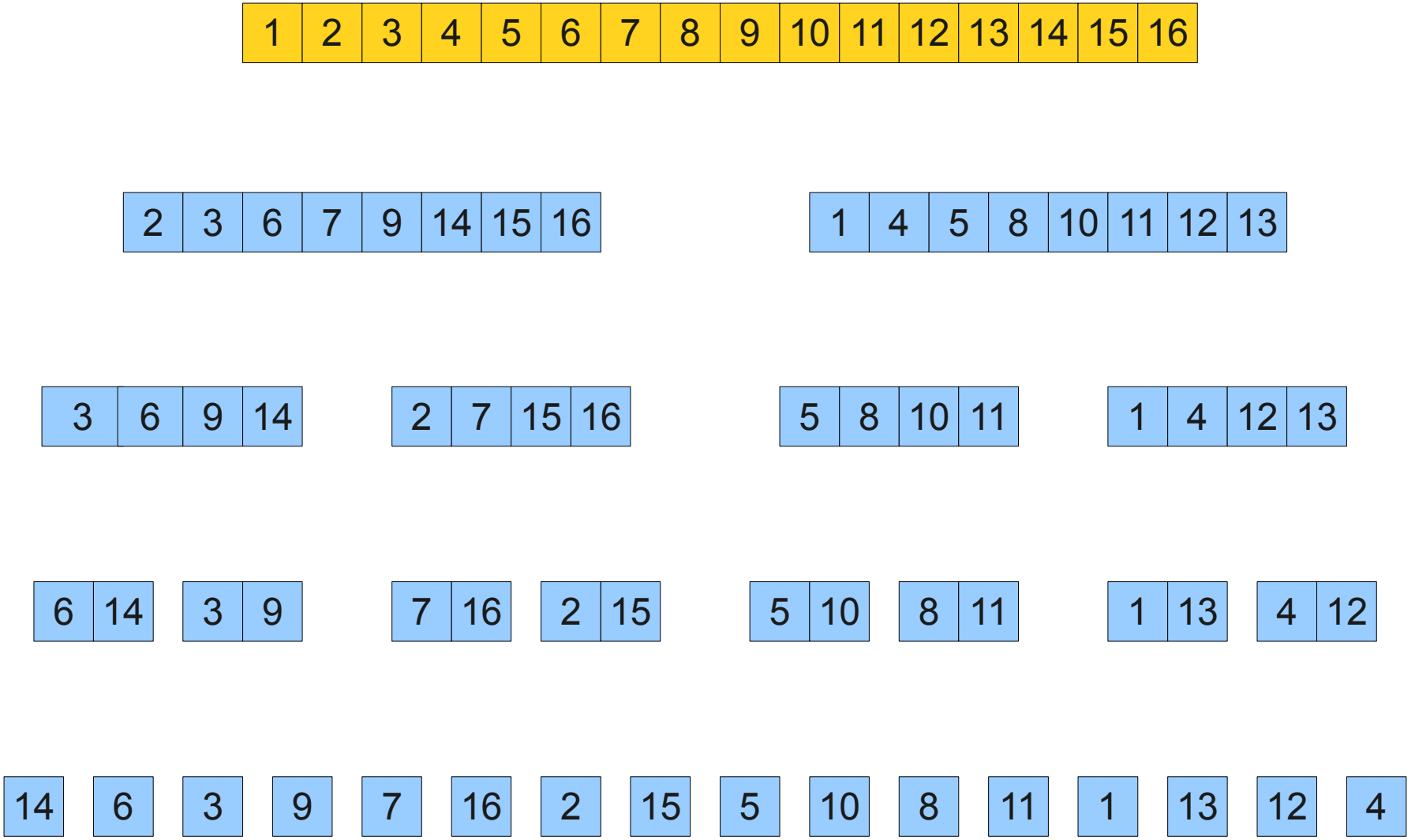

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


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


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


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

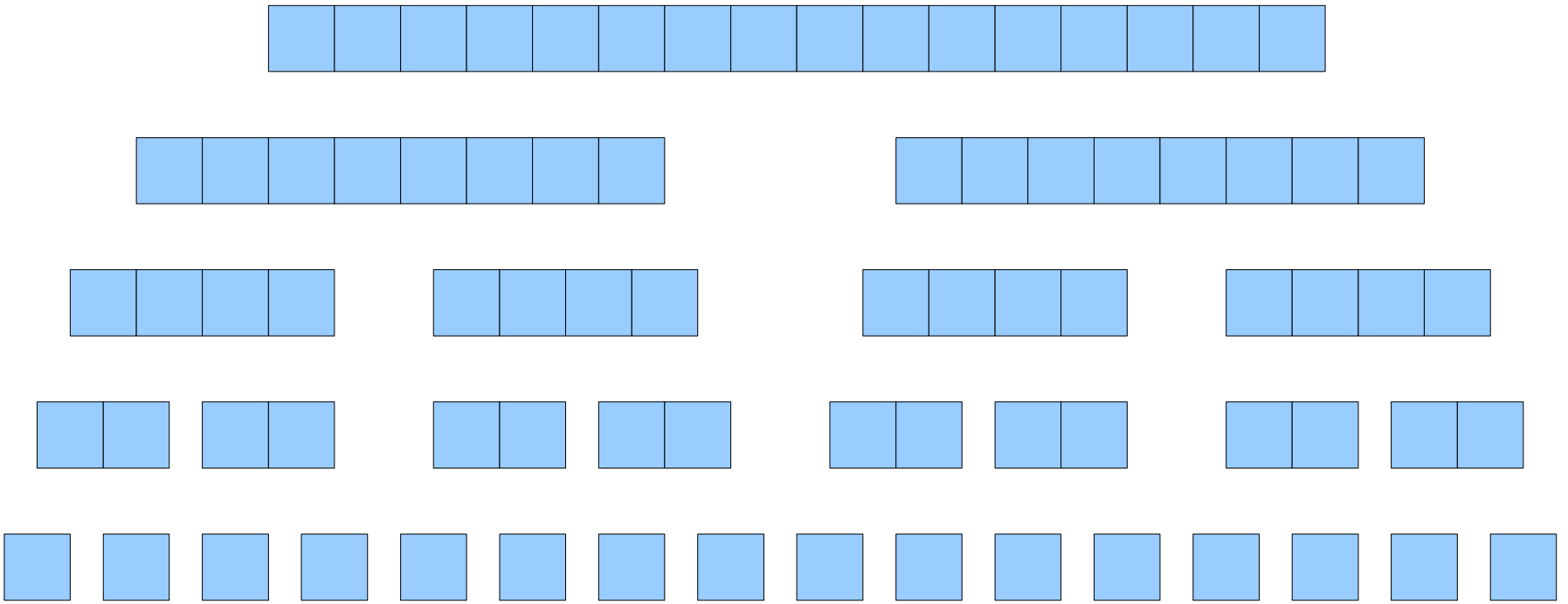

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4


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

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*.

A Graphical Intuition



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

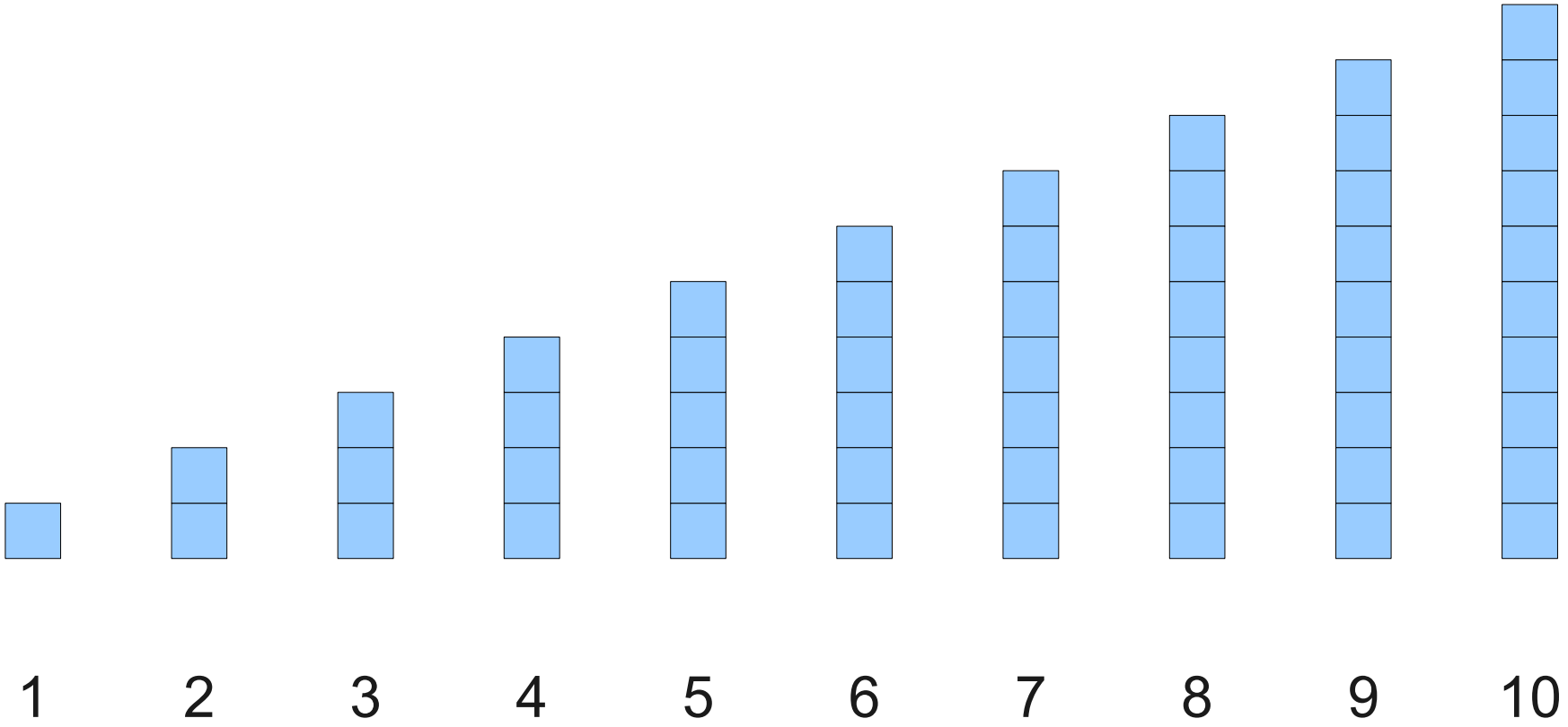
Can we do Better?

- Mergesort is O(n log n).
- This is asymptotically better than O(n^2)
- Can we do better?
  - 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!**

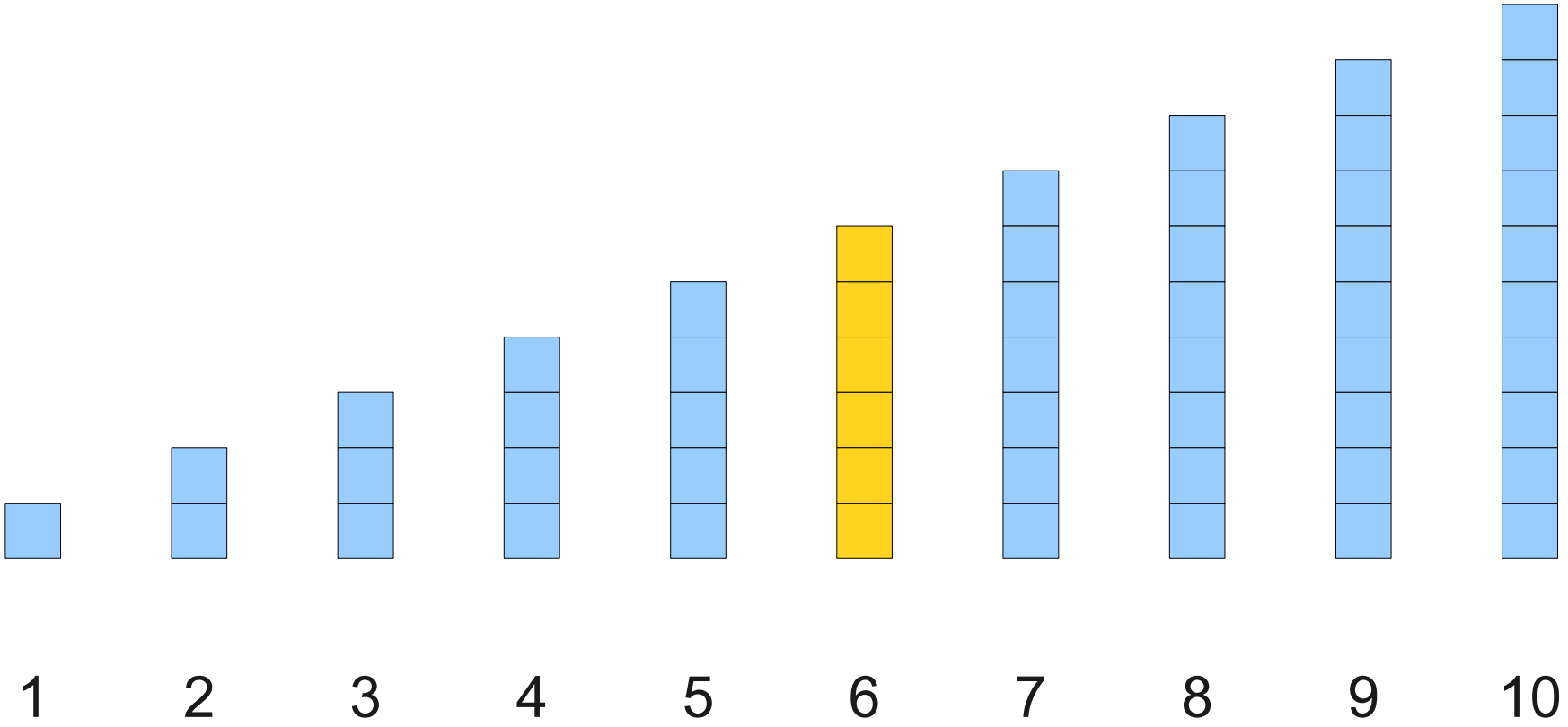
And now… new stuff!

A Trivial Observation

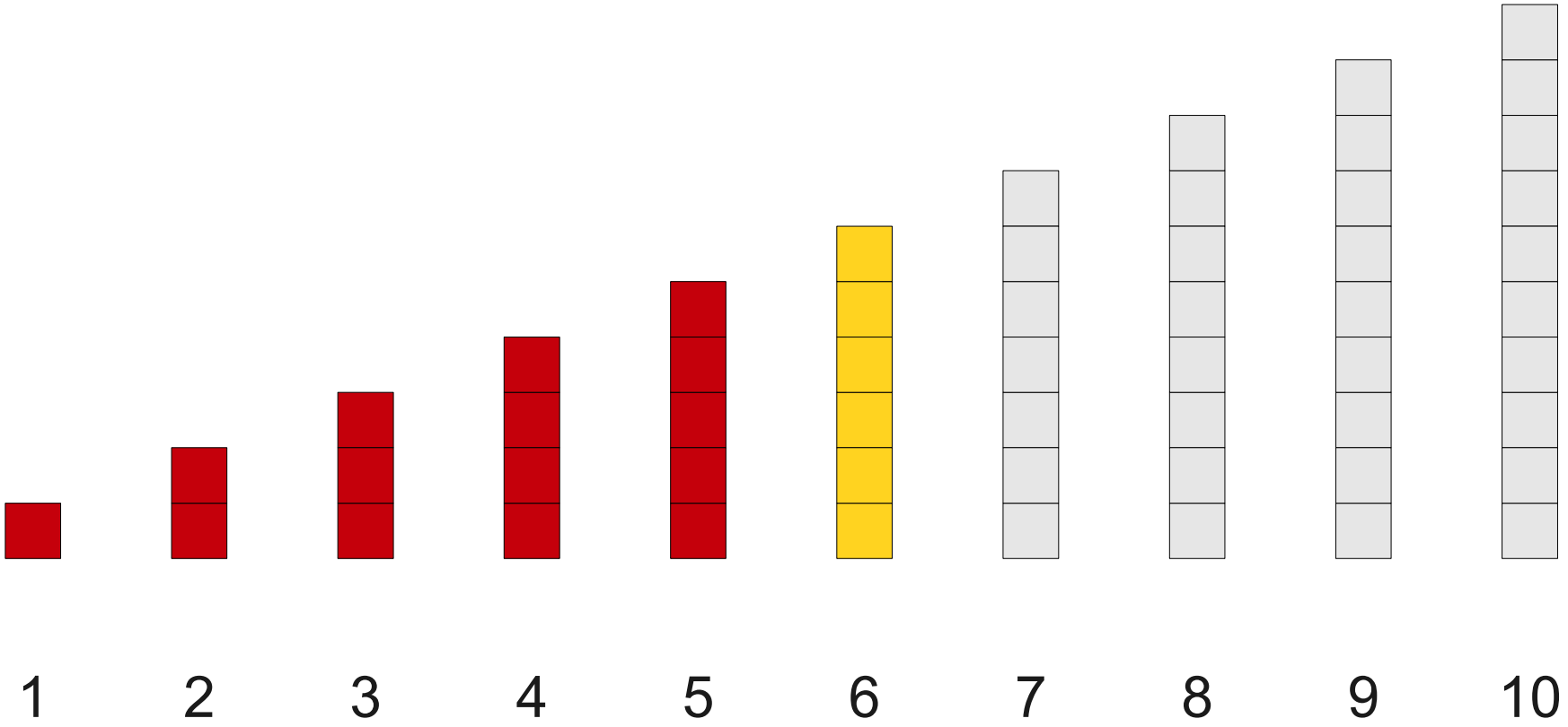
A Trivial Observation



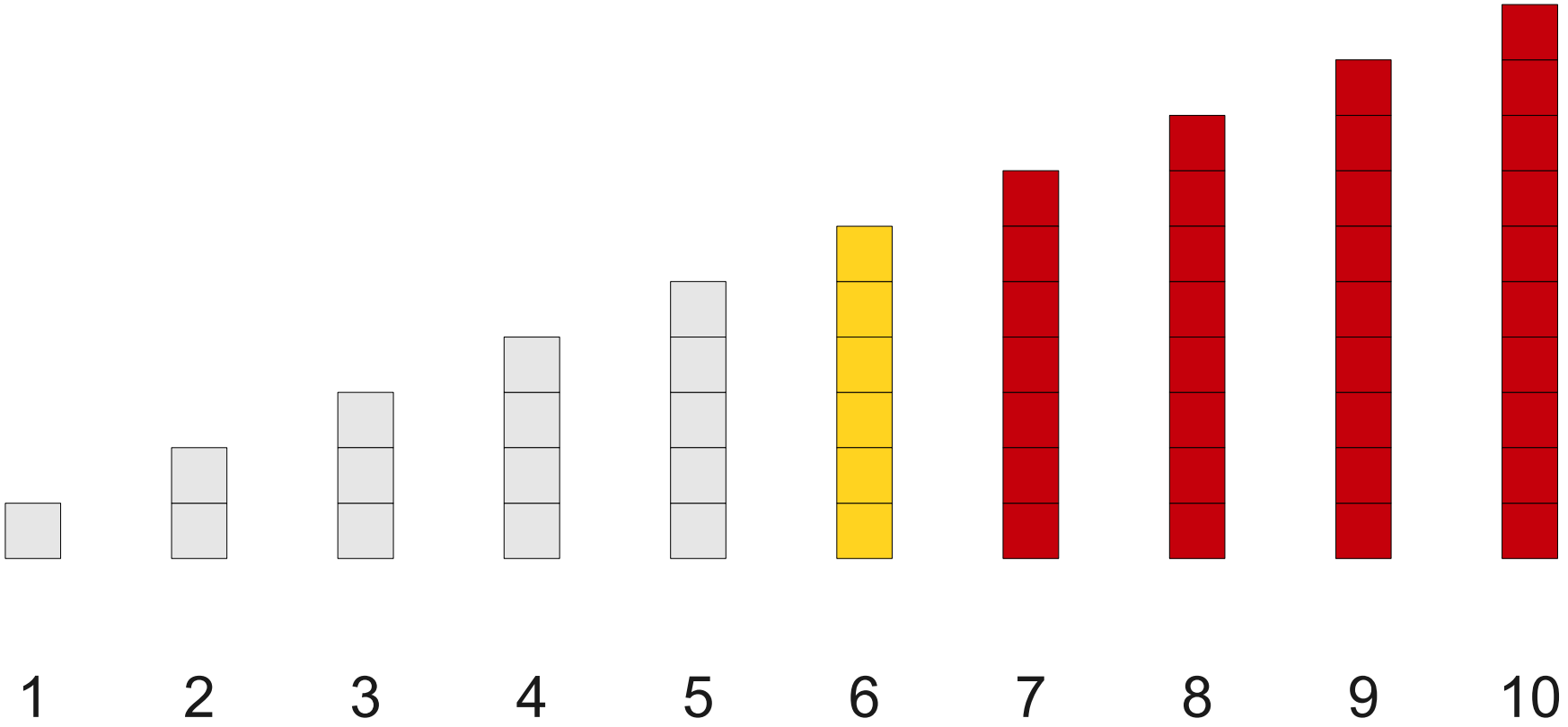
A Trivial Observation



A Trivial Observation



A Trivial Observation



So What?

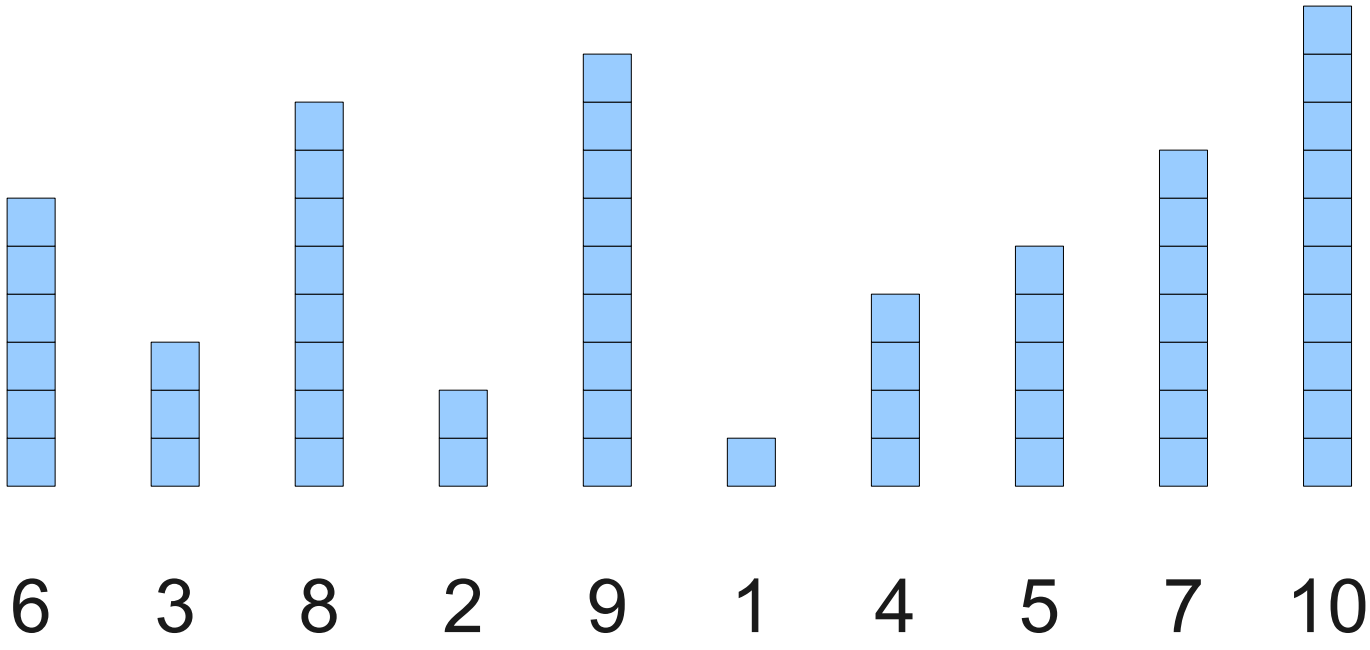
- This idea leads to a particularly clever sorting algorithm.
- Idea:
  - Pick an element from the array.
  - Put the smaller elements on one side.
  - Put the bigger elements on the other side.
  - Recursively sort each half.
- But how do we do the middle two steps?

Partitioning

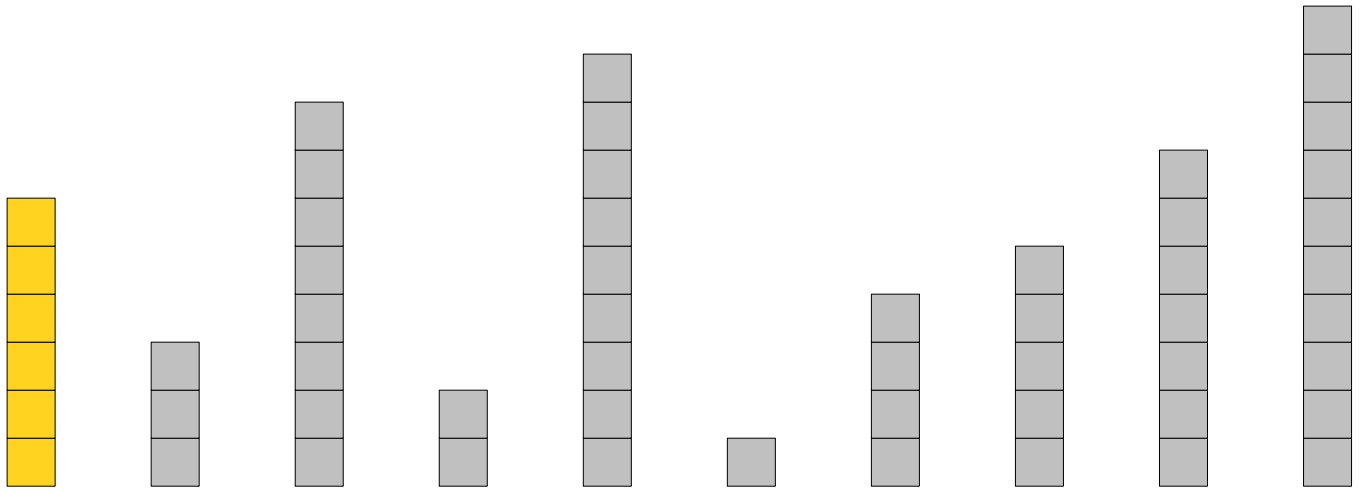
- Pick a **pivot element**.
- Move everything less than the pivot to the left of the pivot.
- Move everything greater than the pivot to the right of the pivot.
- Good news: O(n) algorithm exists!
- Bad news: it's a bit tricky...

The Partition Algorithm

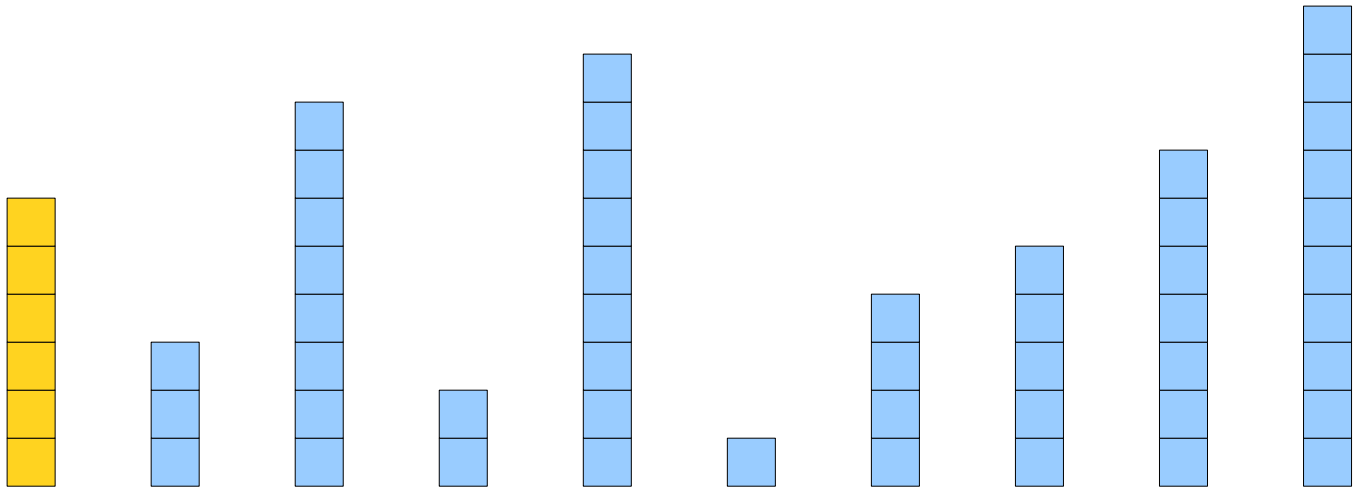
The Partition Algorithm



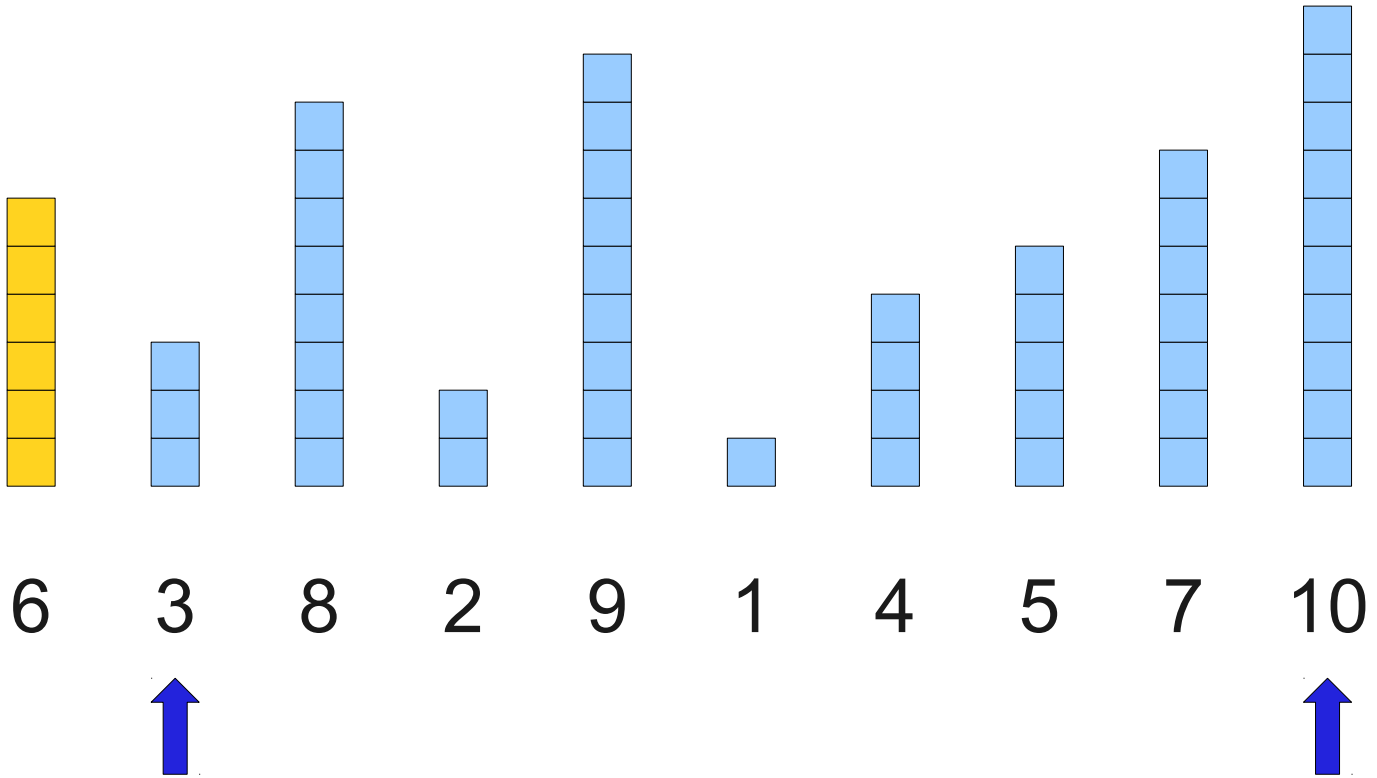
The Partition Algorithm



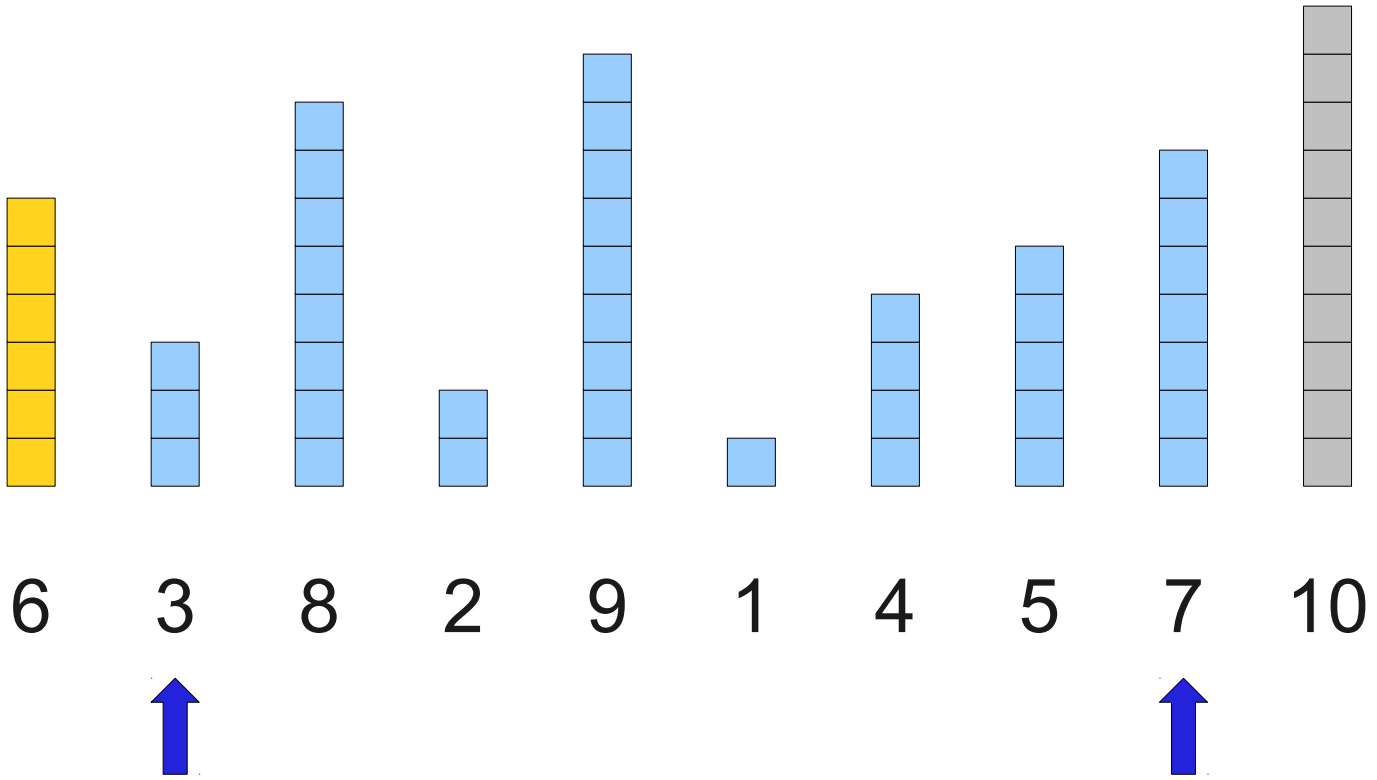
The Partition Algorithm



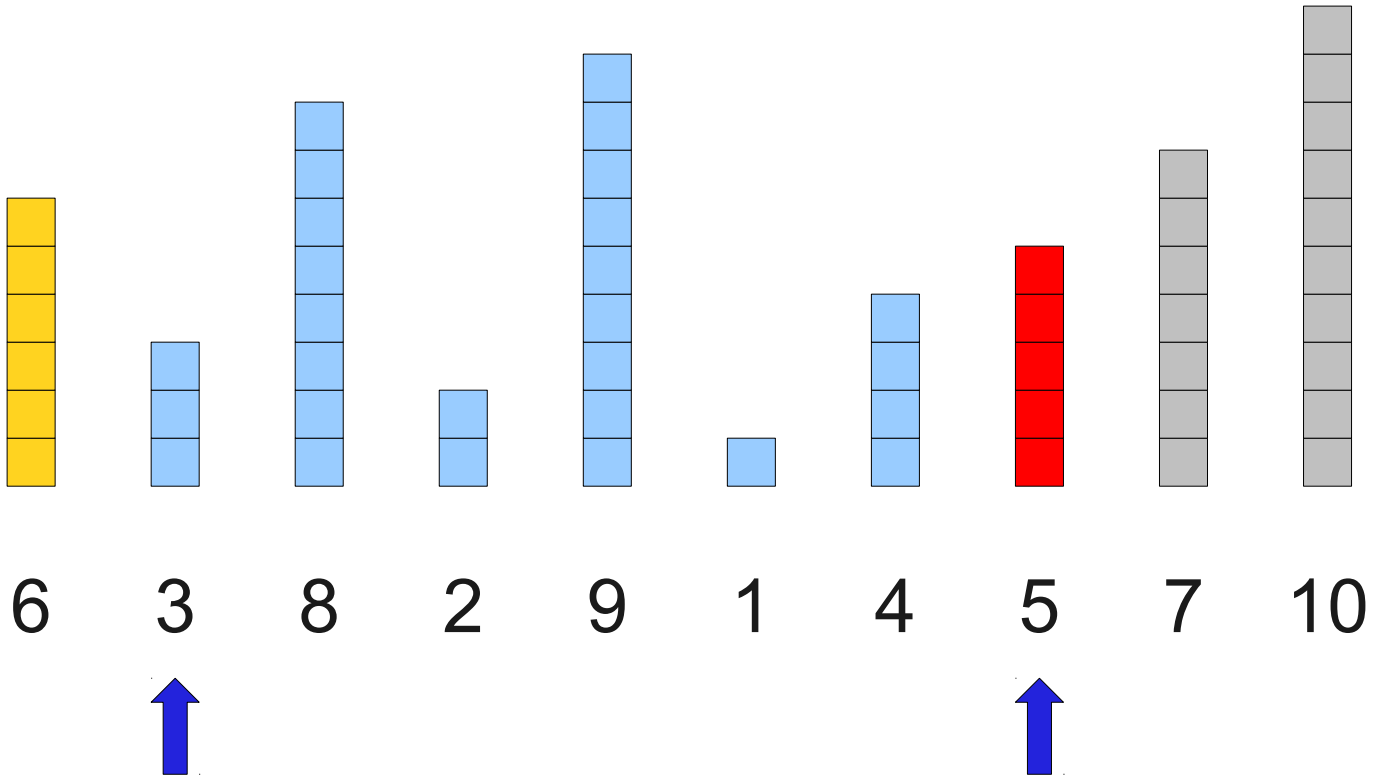
The Partition Algorithm



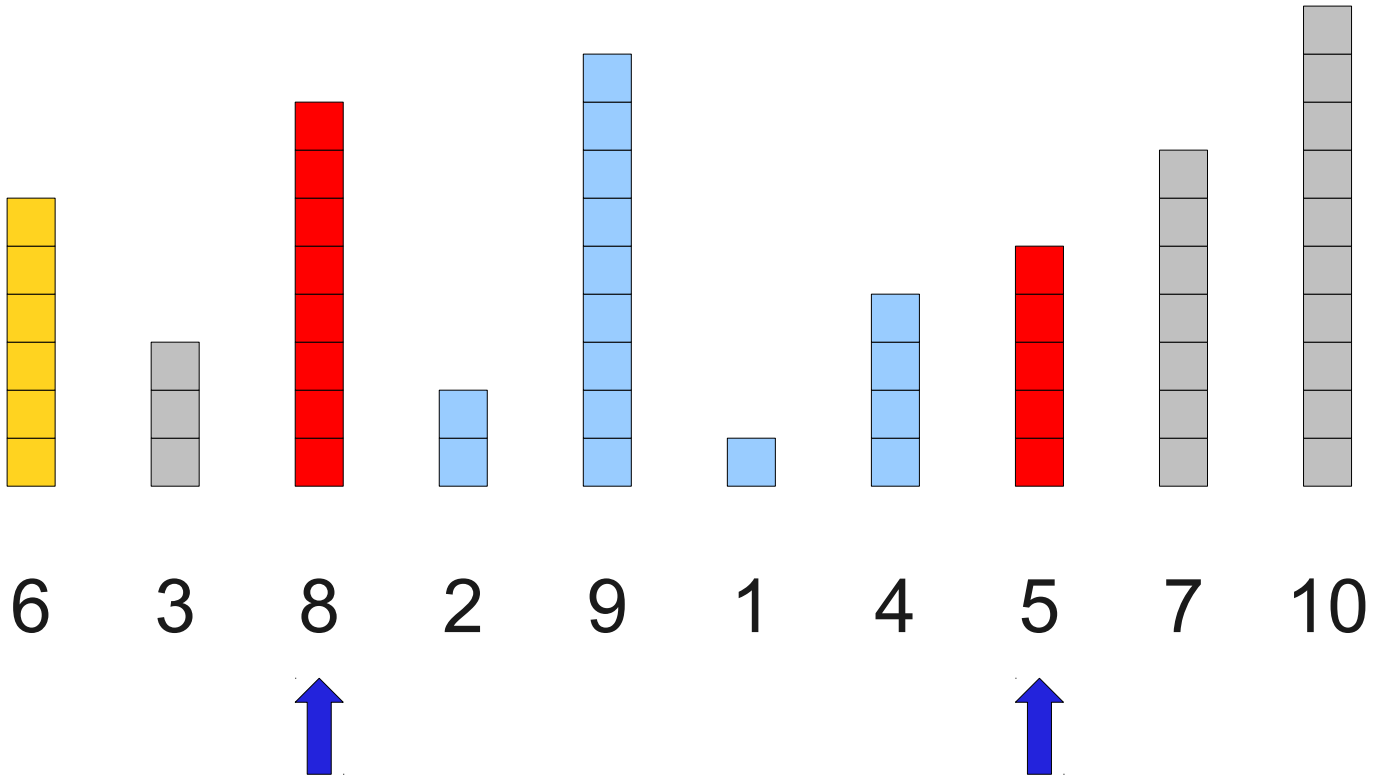
The Partition Algorithm



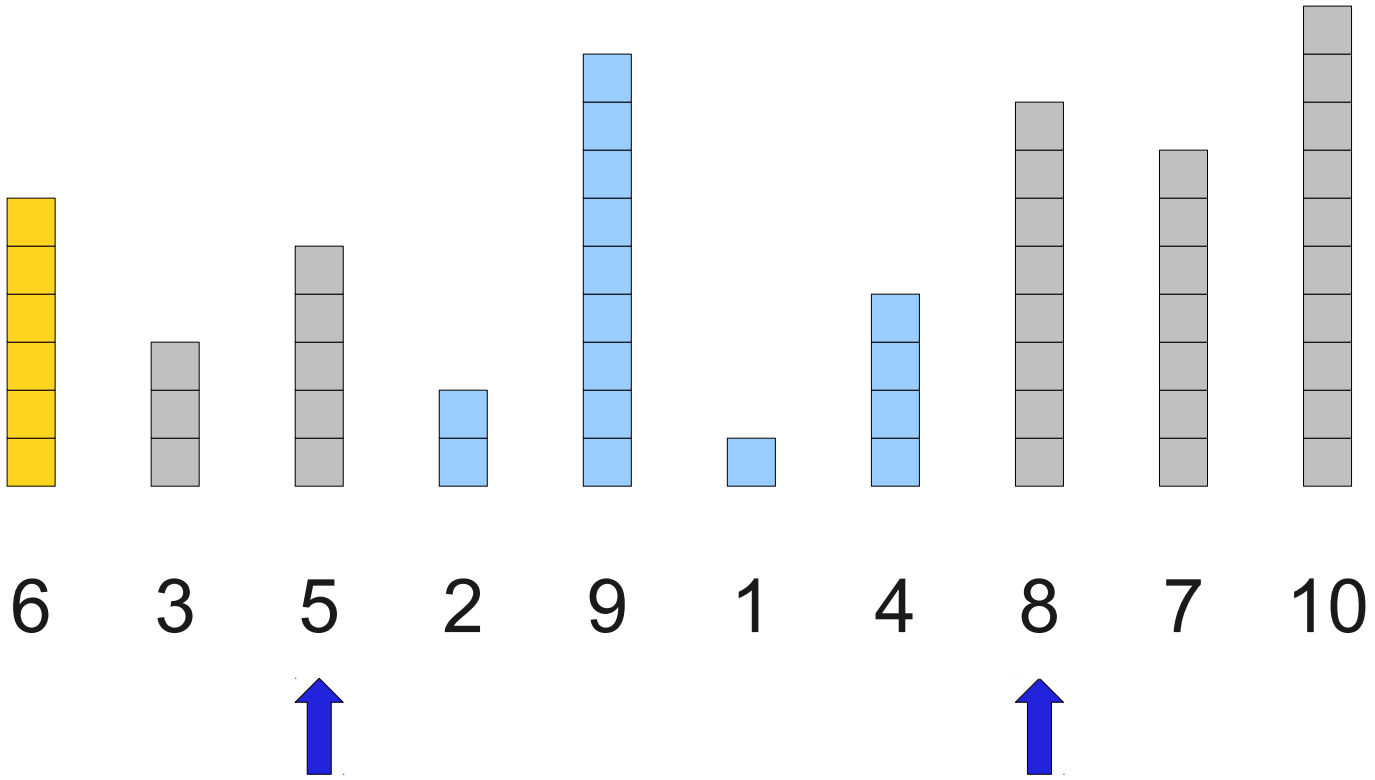
The Partition Algorithm



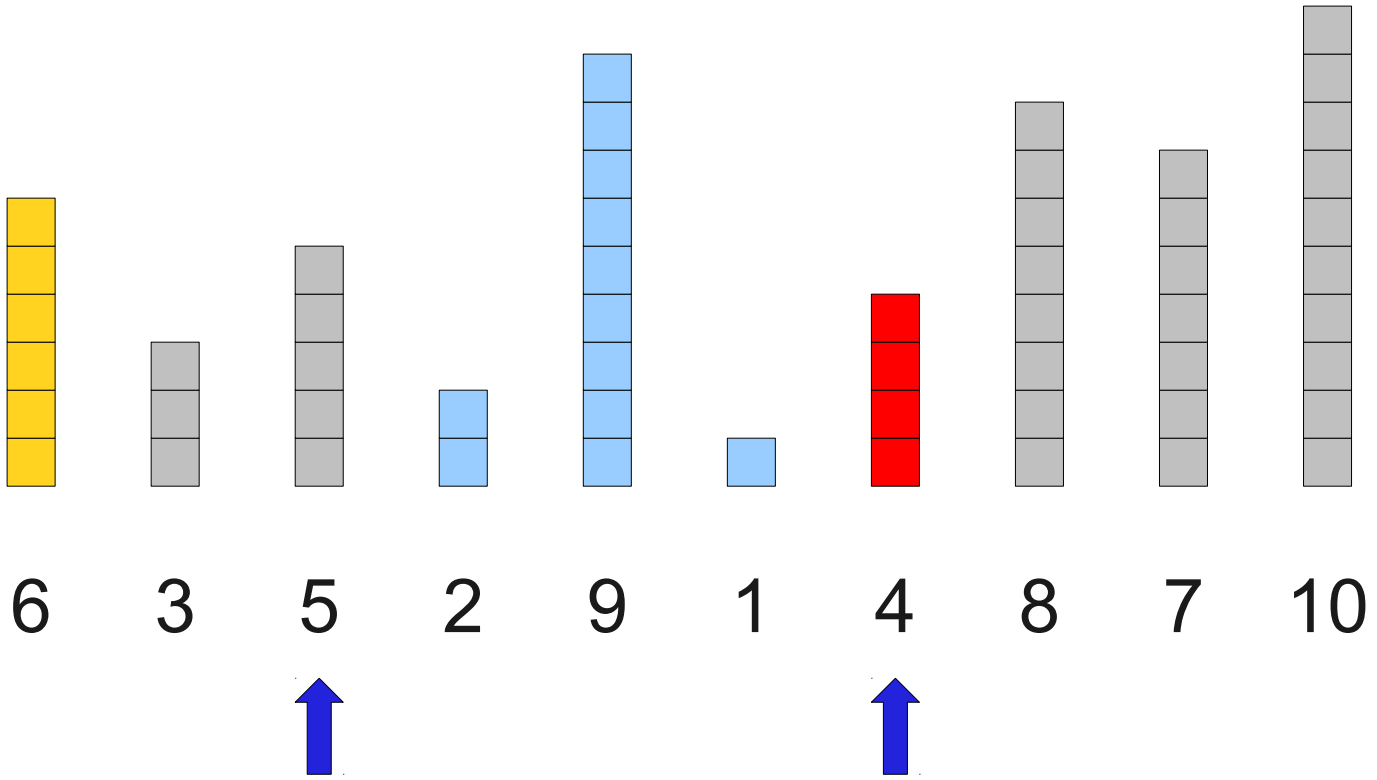
The Partition Algorithm



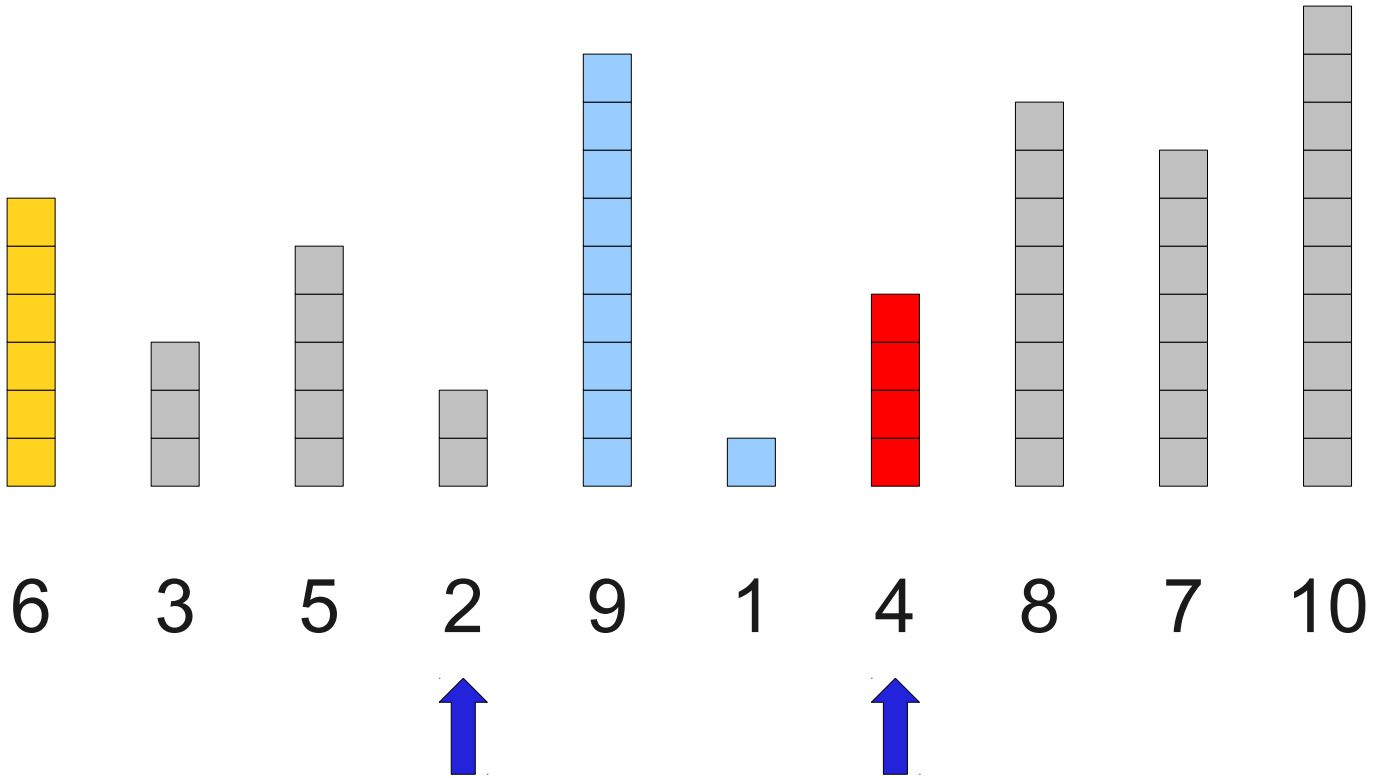
The Partition Algorithm



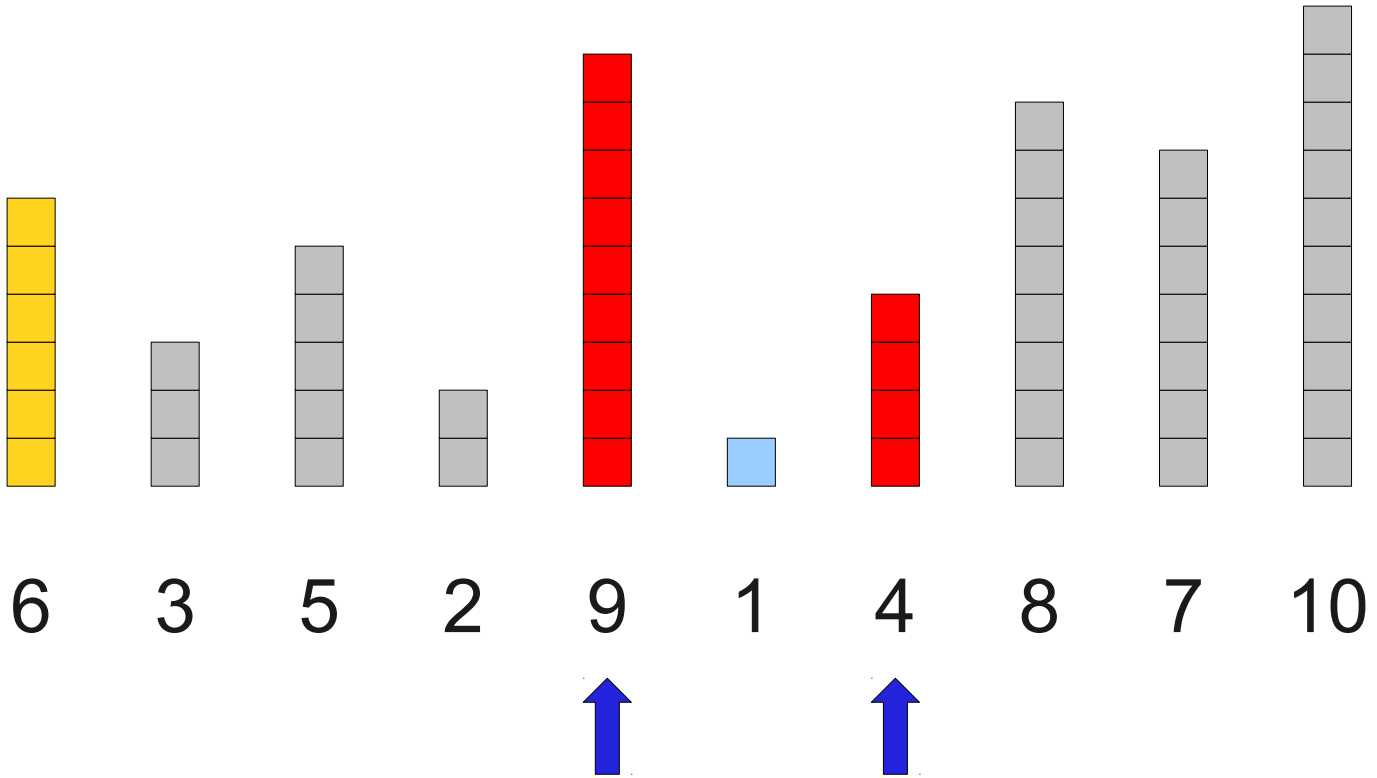
The Partition Algorithm



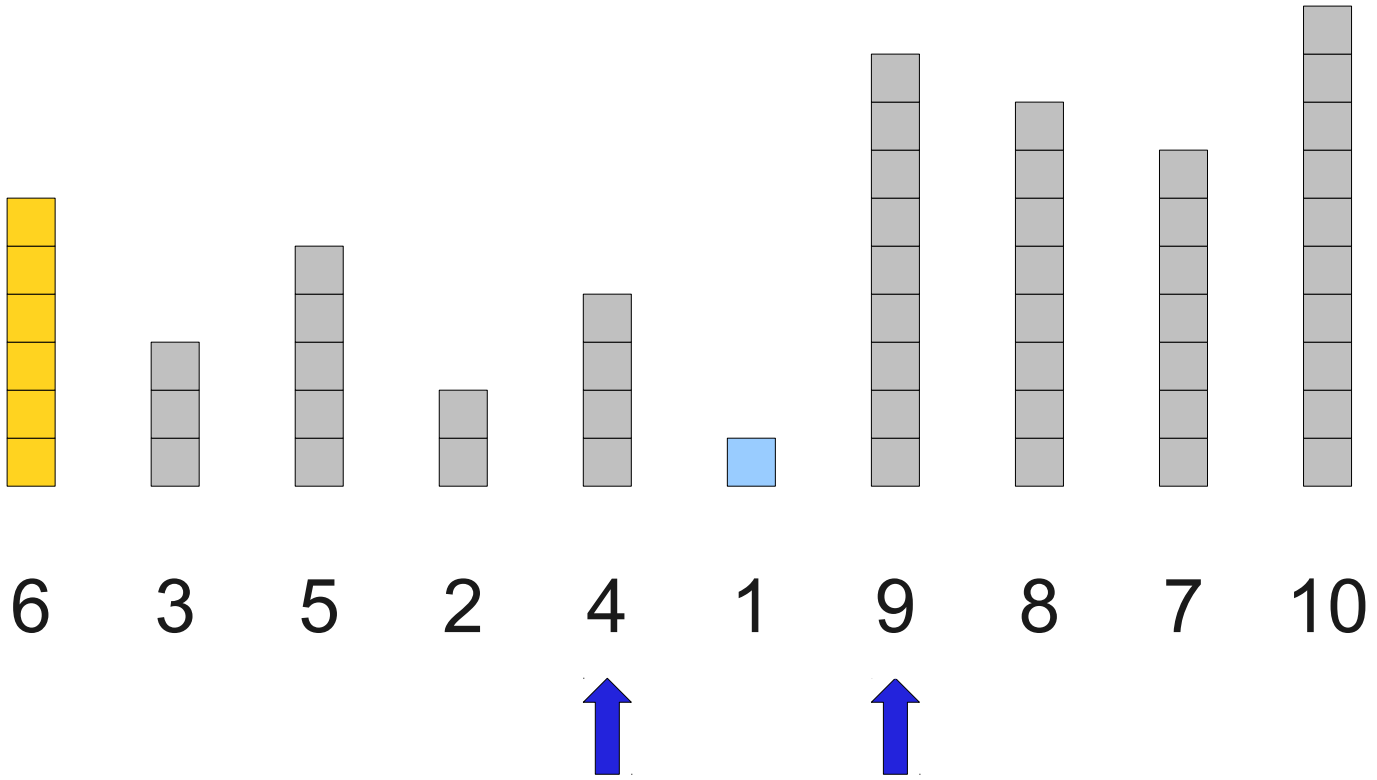
The Partition Algorithm



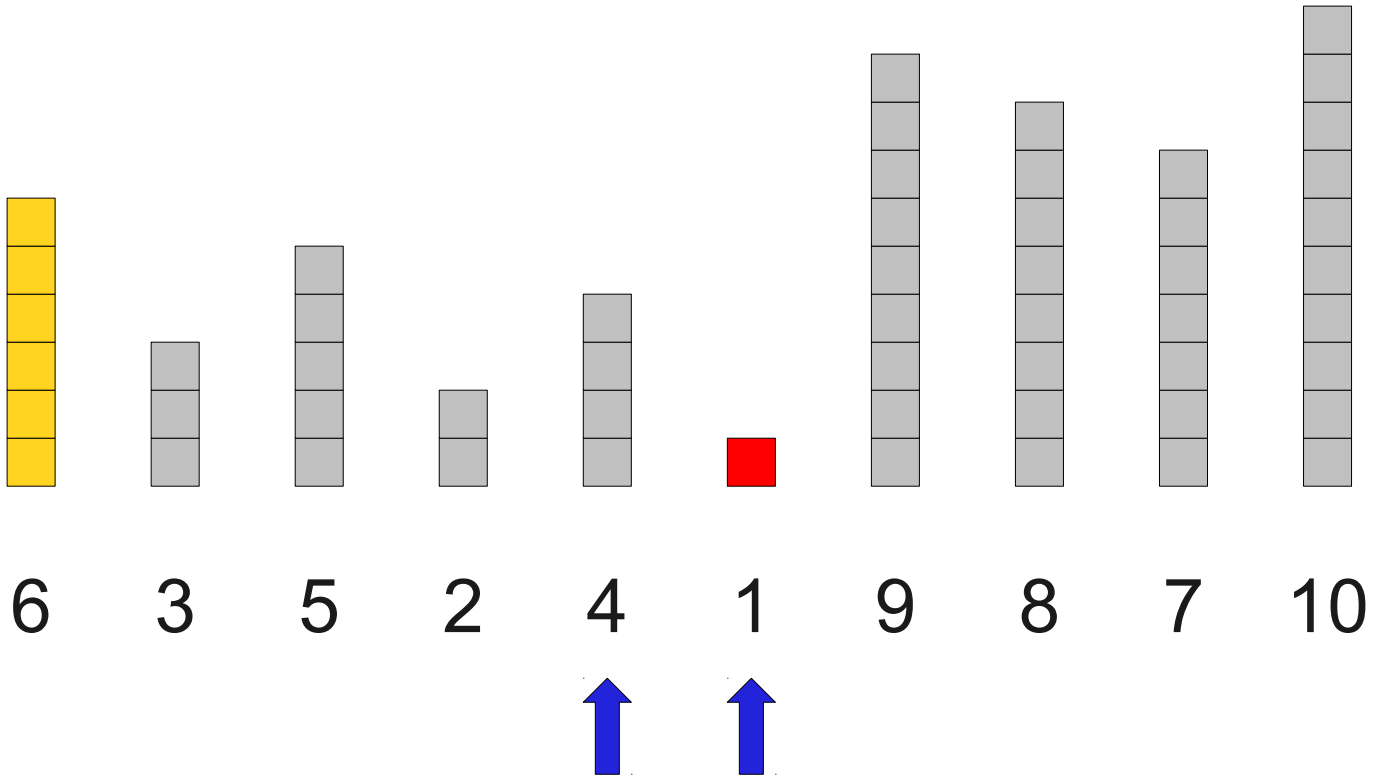
The Partition Algorithm



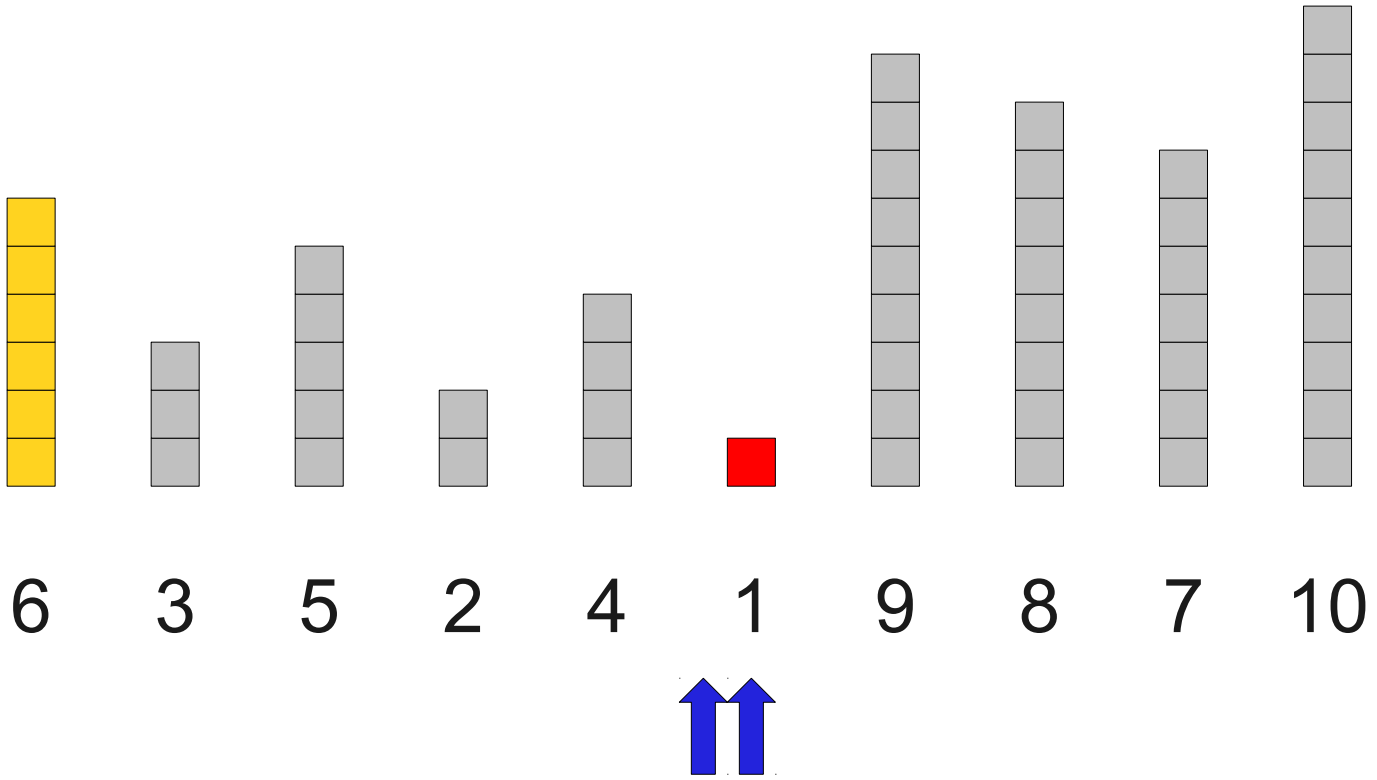
The Partition Algorithm



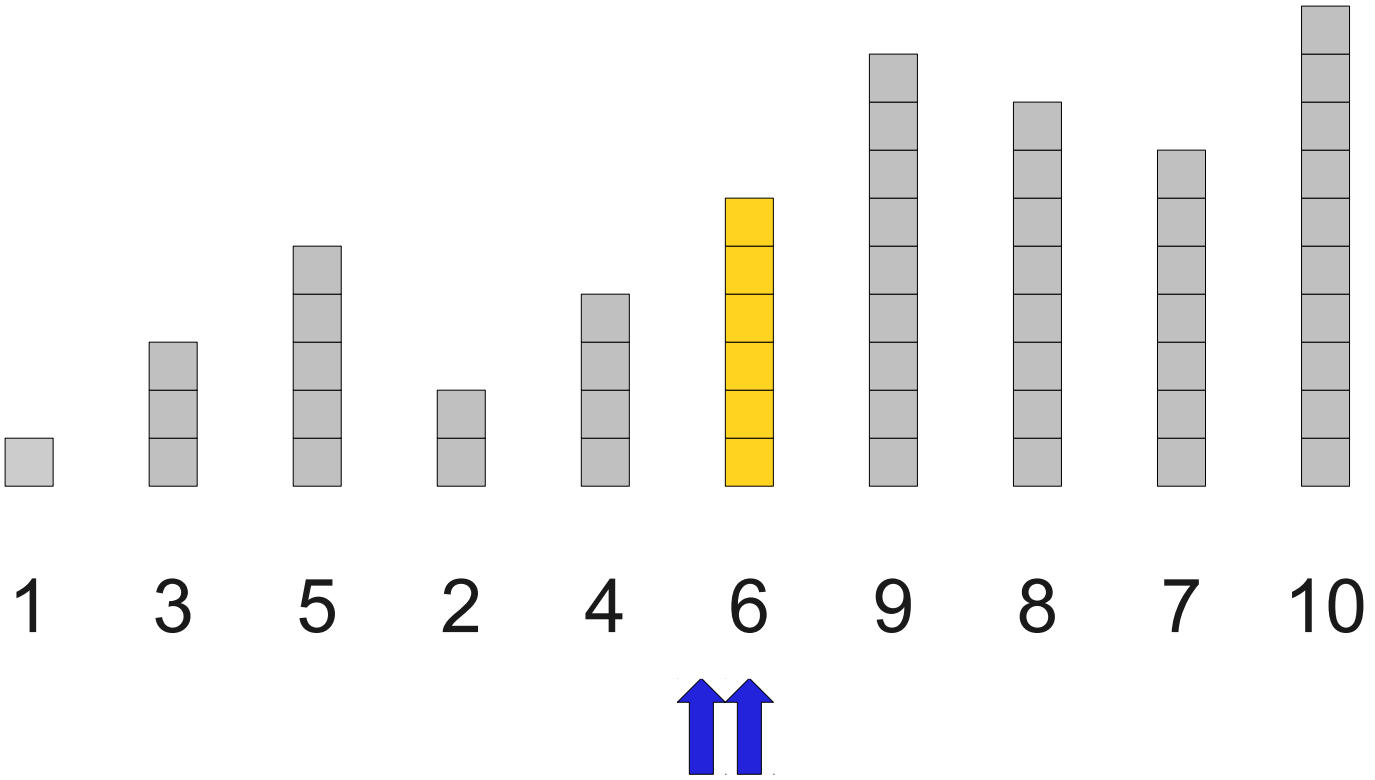
The Partition Algorithm



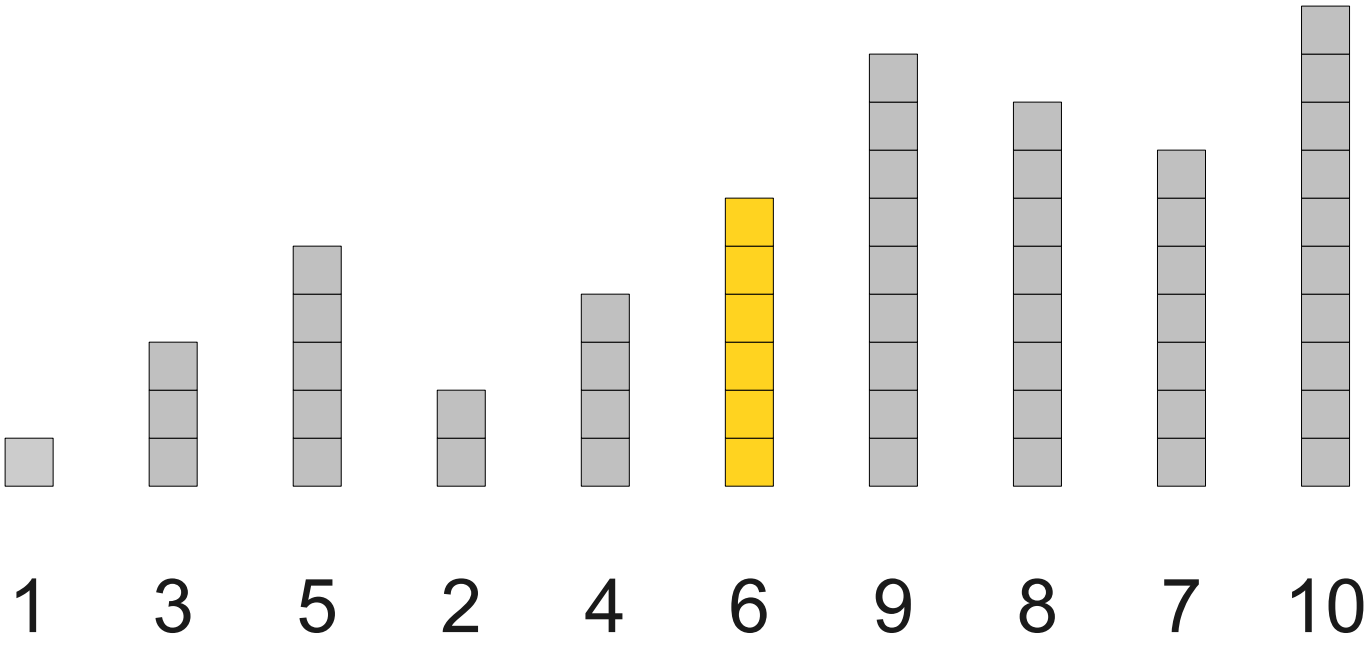
The Partition Algorithm



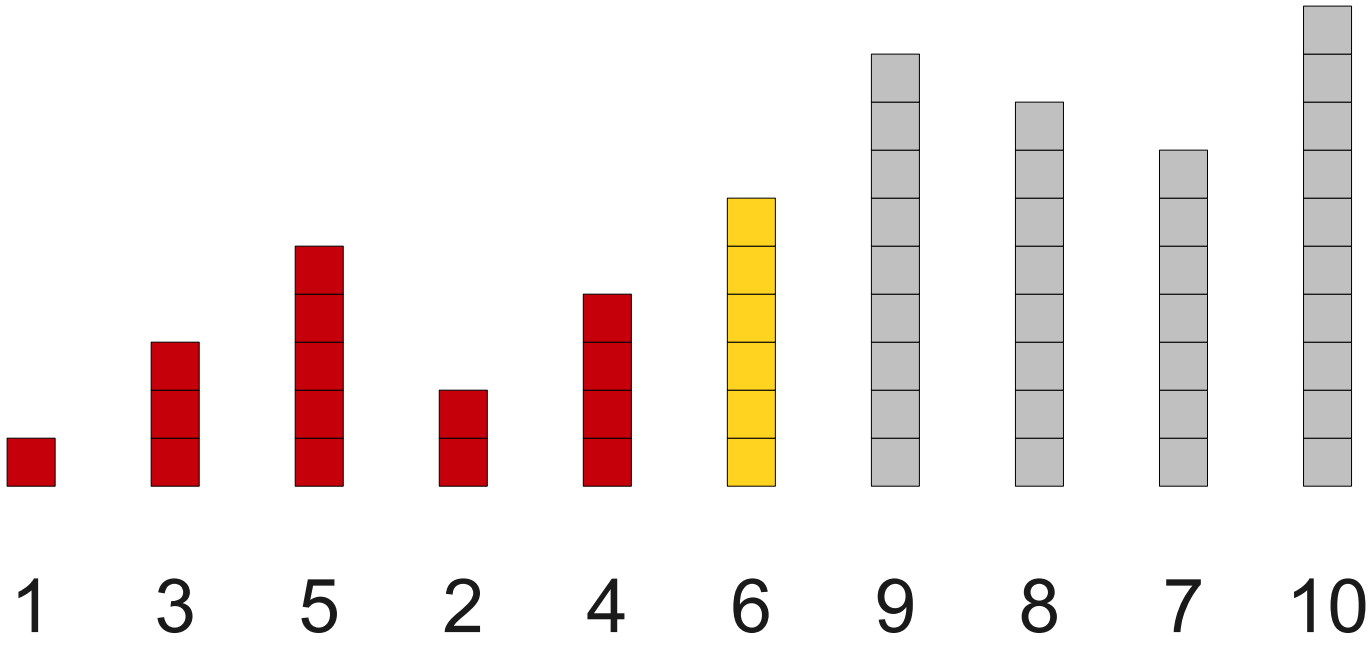
The Partition Algorithm



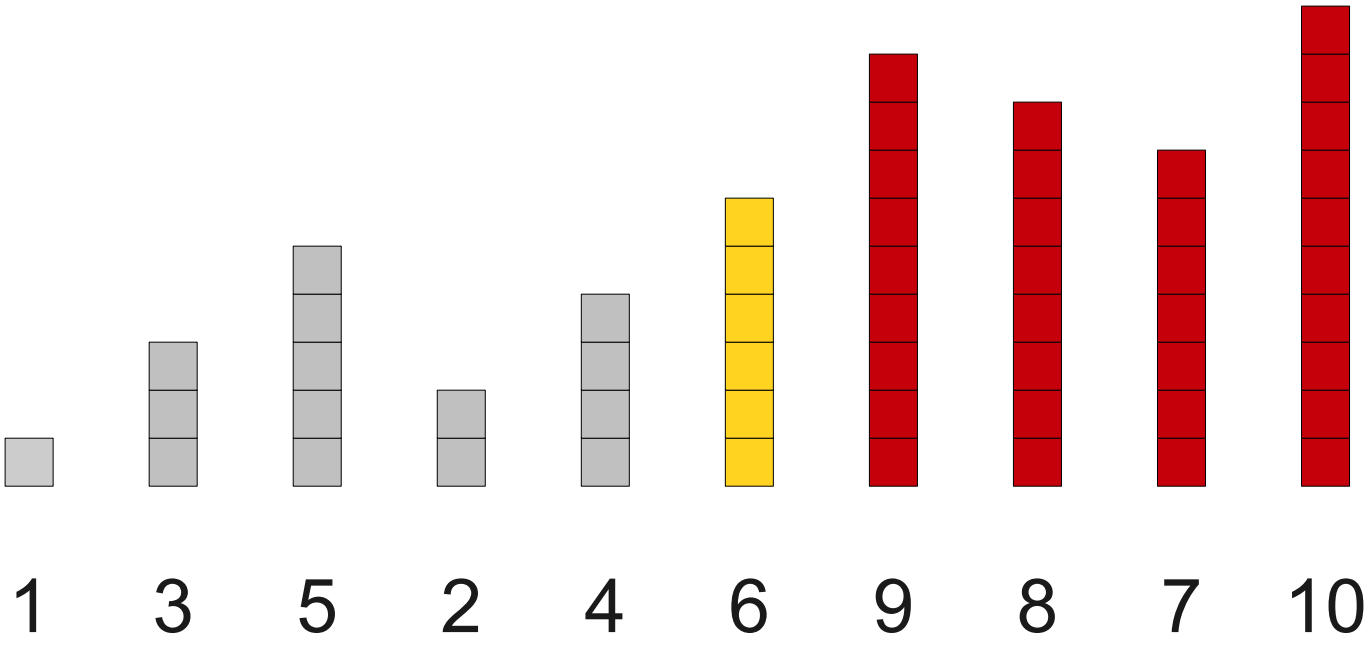
The Partition Algorithm



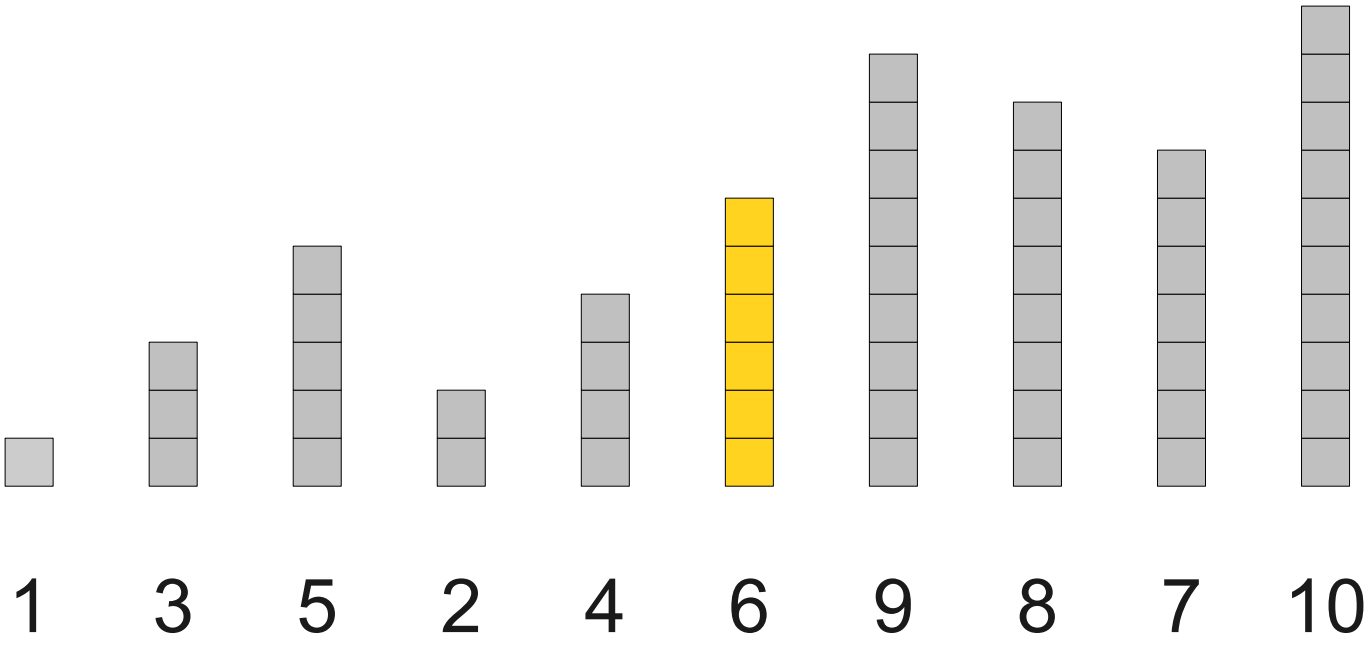
The Partition Algorithm



The Partition Algorithm



The Partition Algorithm



Code for Partition

Code for Partition

```
int partition(Vector<int>&
  int pivot = v[low];
  int left  = low + 1, rig

  while (left < right) {
    while (left < right &&                    ht;
    while (left < right &&                    t;

    if (left < right) swap
  }

  if (pivot < v[right]) re
  swap(v[low], v[right]);
  return right;
}
```

A Partition-Based Sort

- Idea:
  - Partition the array around some element.
  - Recursively sort the left and right halves.
- This works extremely quickly.
- In fact... the algorithm is called ***quicksort***.

Quicksort

Quicksort

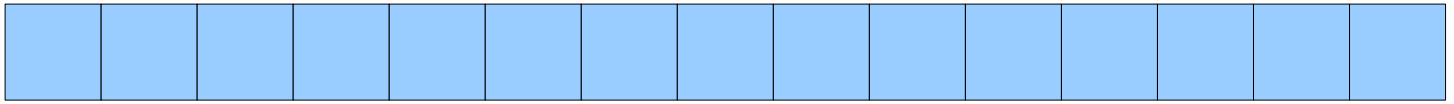
```
void quicksort(Vector<int>& v, int low, int high) {
    if (low >= high) return;

    int partitionPoint = partition(v, low, high);
    quicksort(v, low, partitionPoint – 1);
    quicksort(v, partitionPoint + 1, high);
}
```

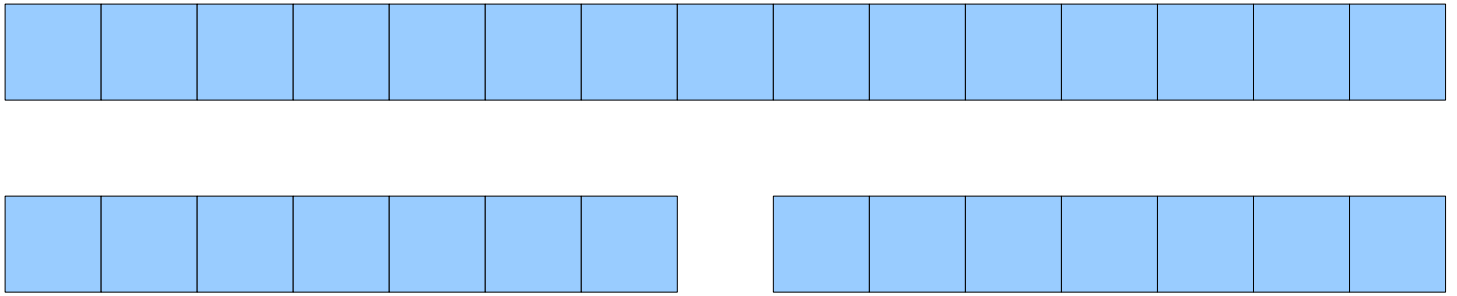
How fast is quicksort?

It depends on our choice of pivot.

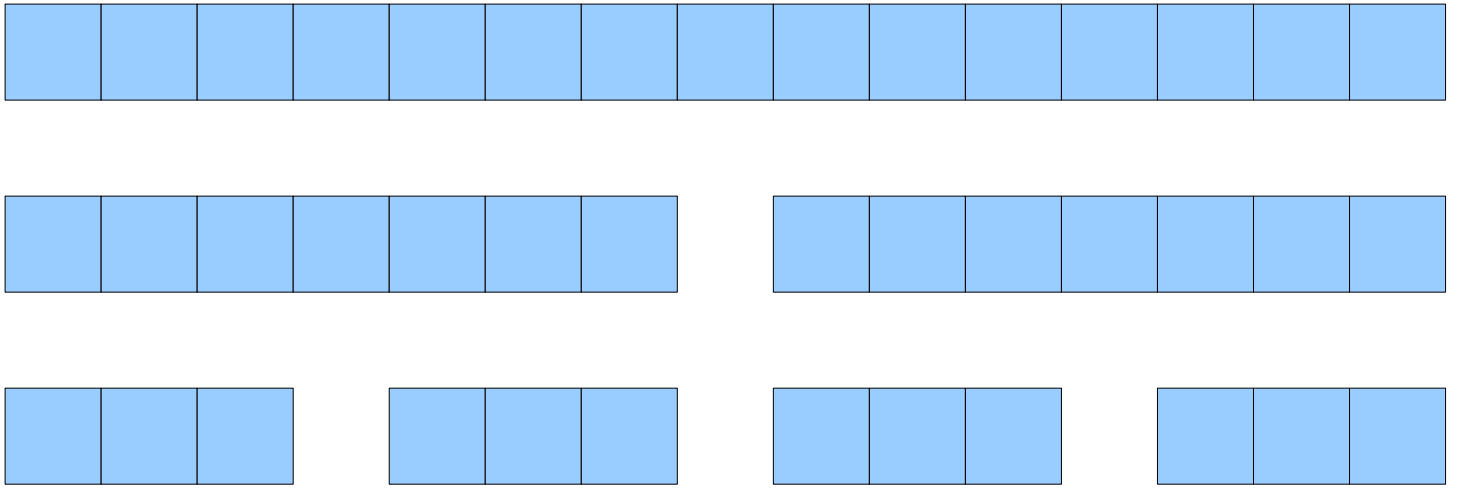
Suppose we get lucky...



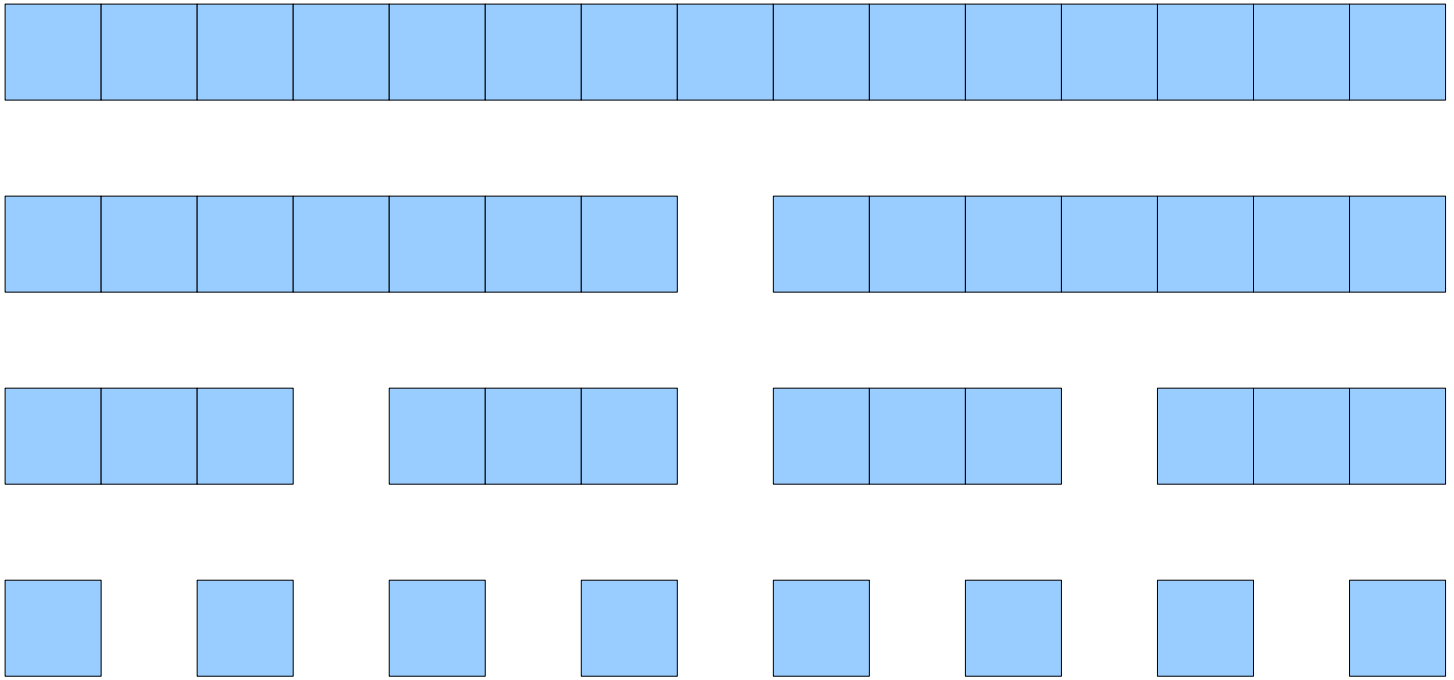
Suppose we get lucky...



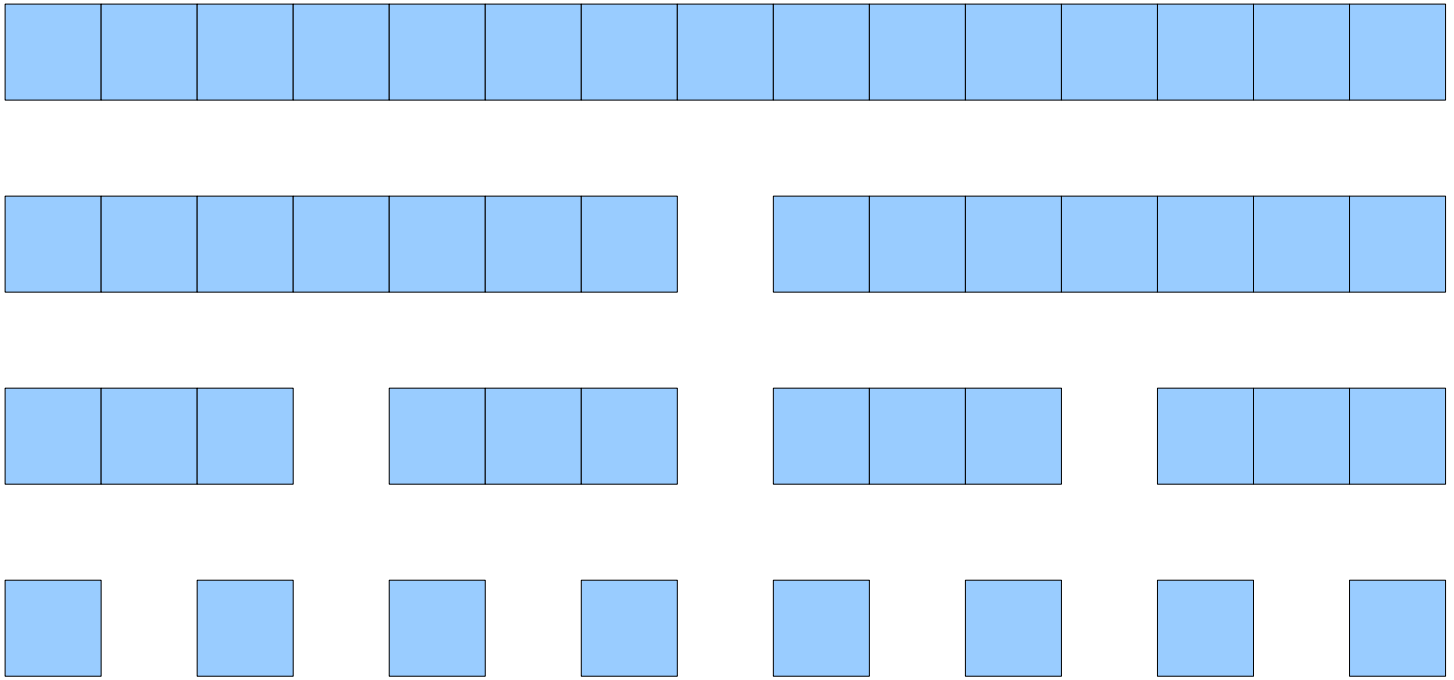
Suppose we get lucky...



Suppose we get lucky...

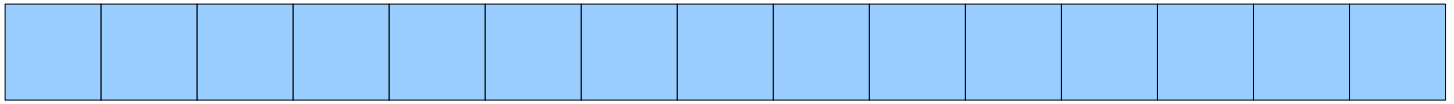


Suppose we get lucky...

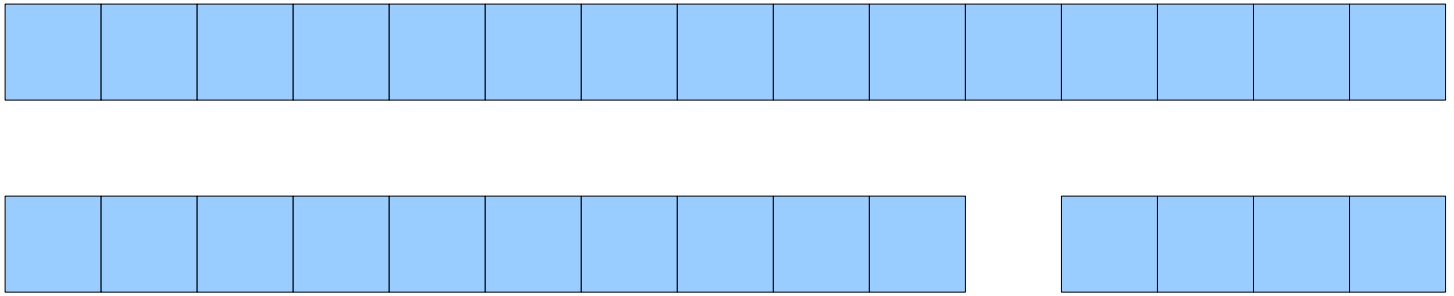


O(*n* log *n*)

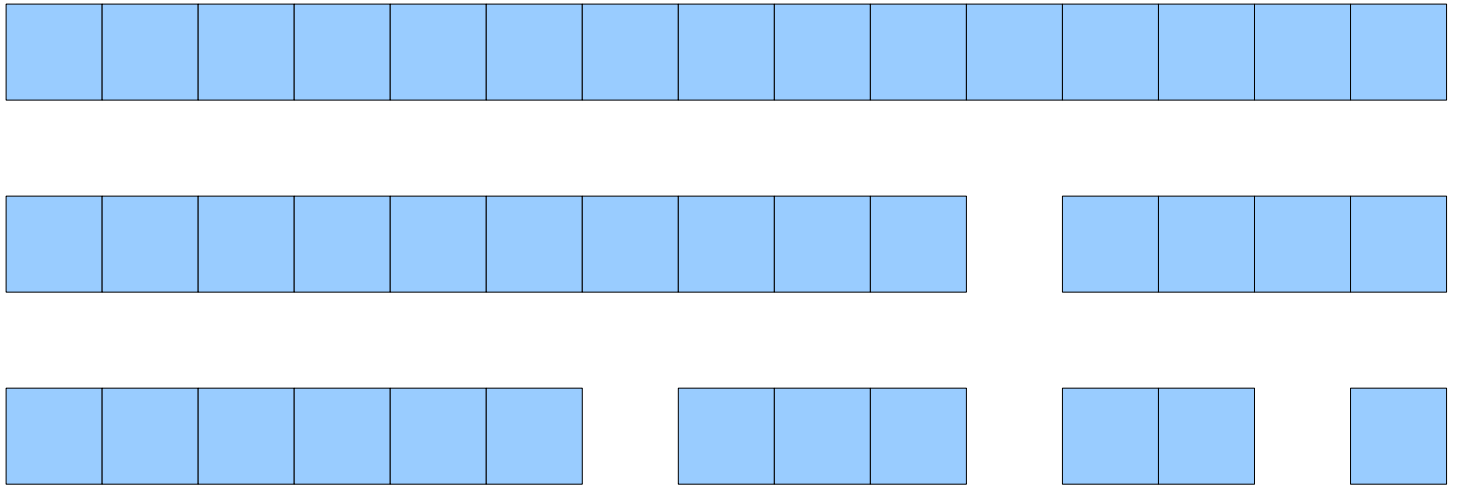
Suppose we get *sorta* lucky...



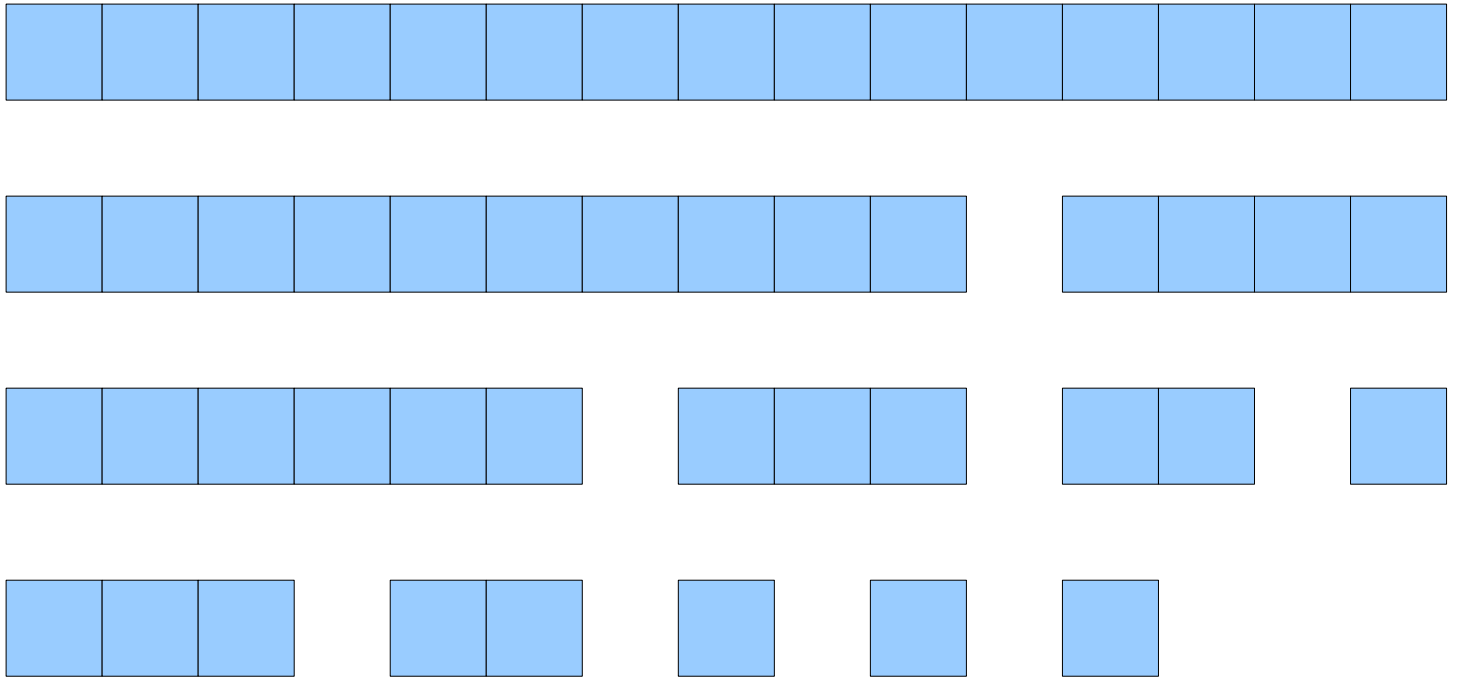
Suppose we get *sorta* lucky...



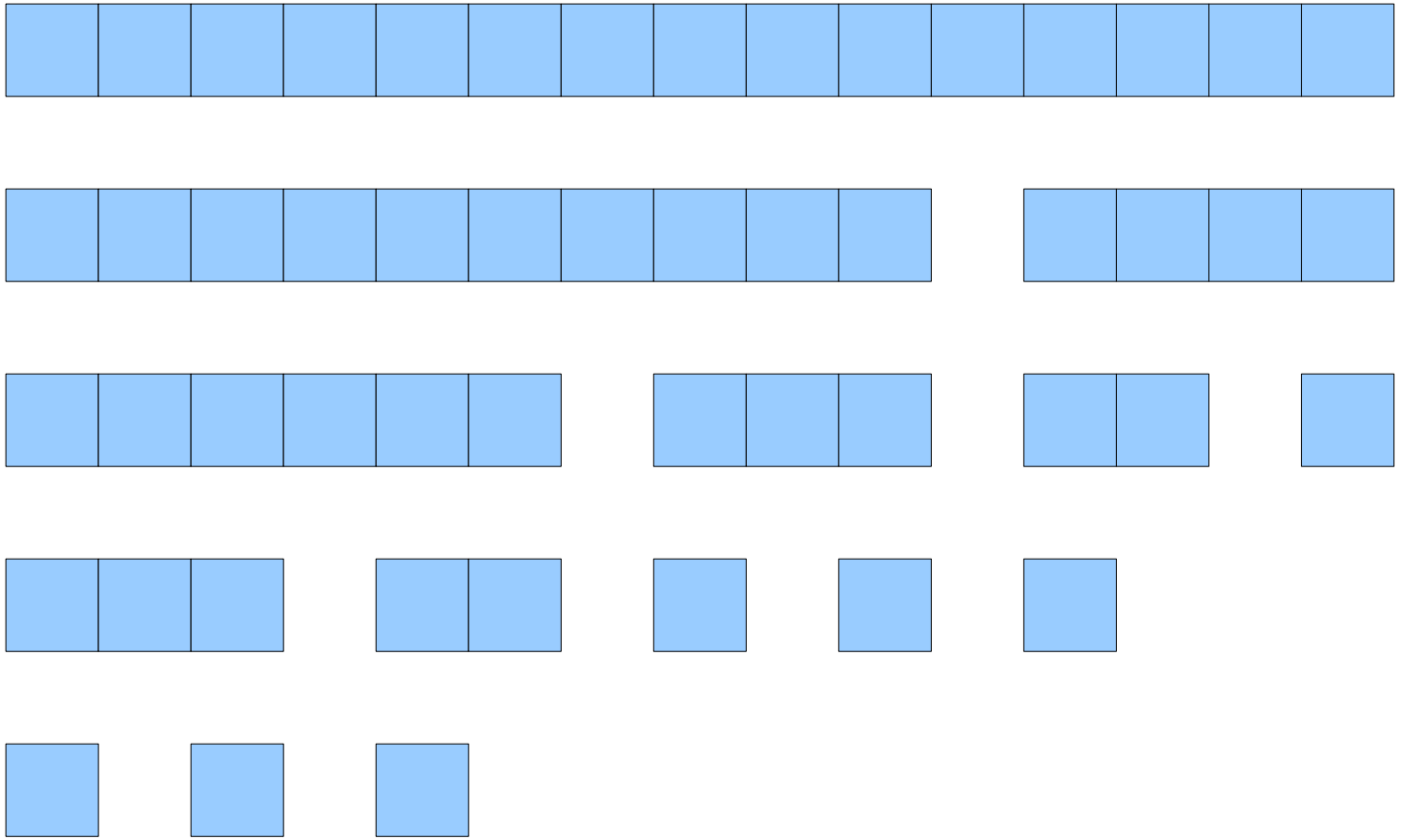
Suppose we get *sorta* lucky...



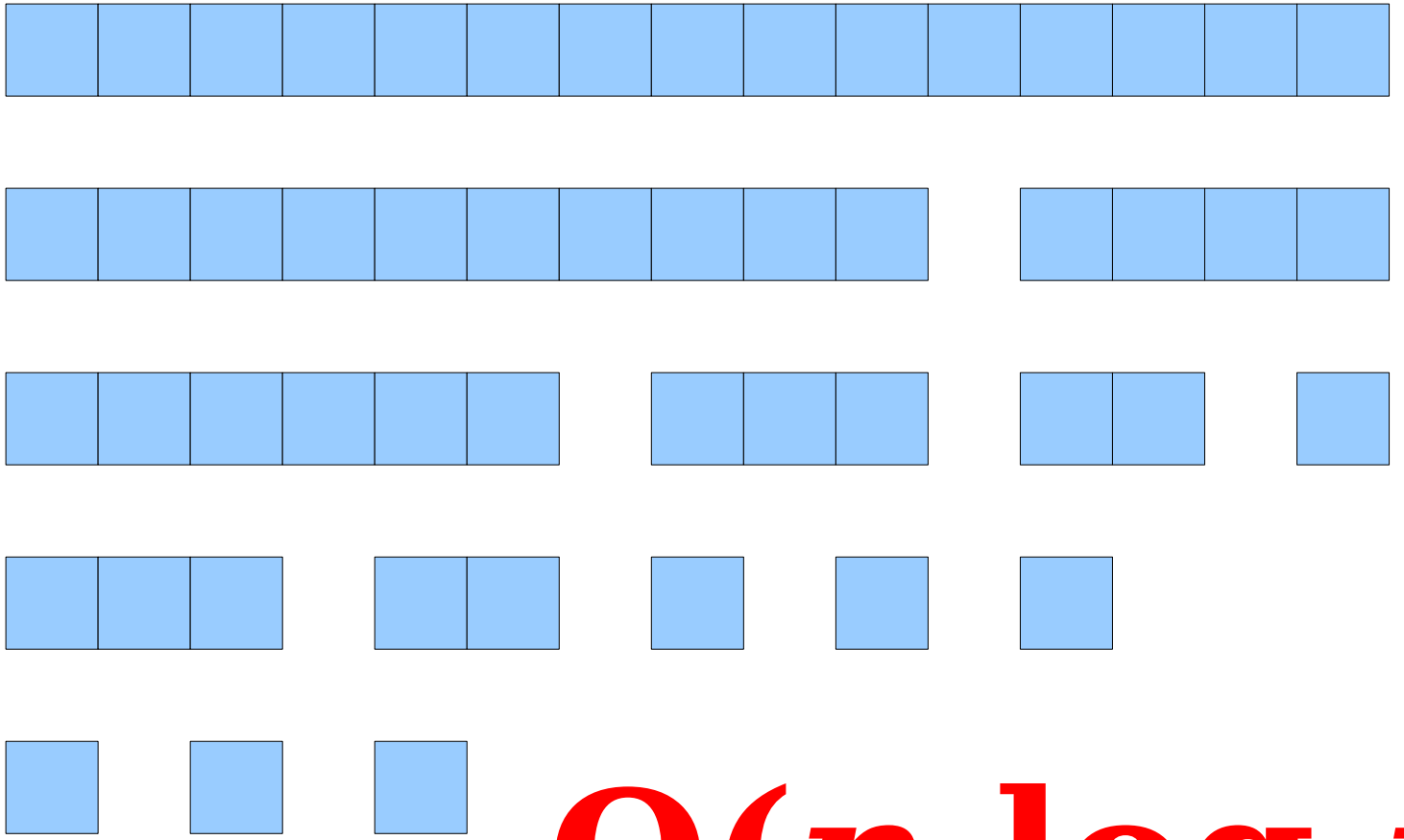
Suppose we get *sorta* lucky...



Suppose we get *sorta* lucky...

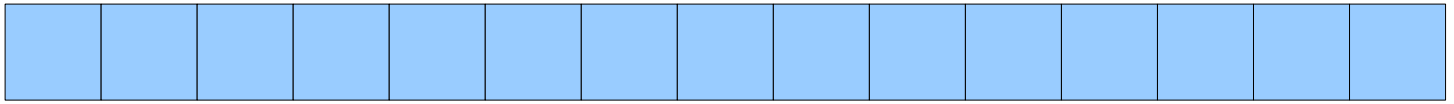


Suppose we get *sorta* lucky...

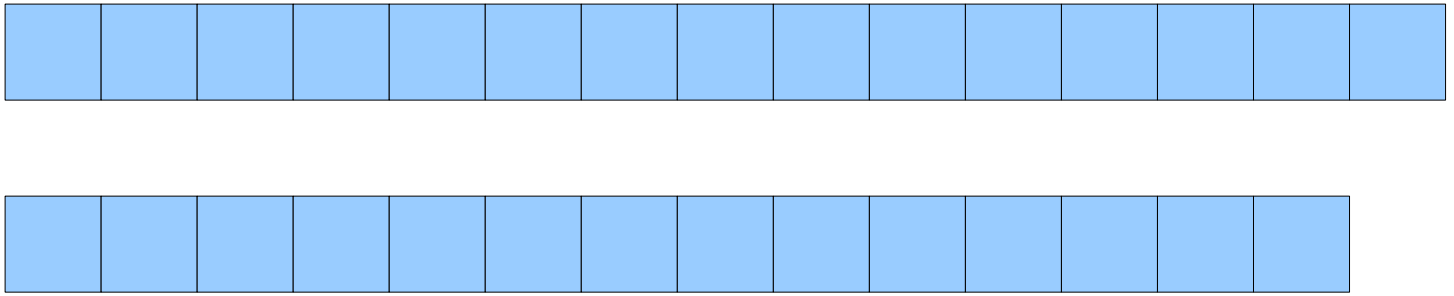


O(*n* log *n*)

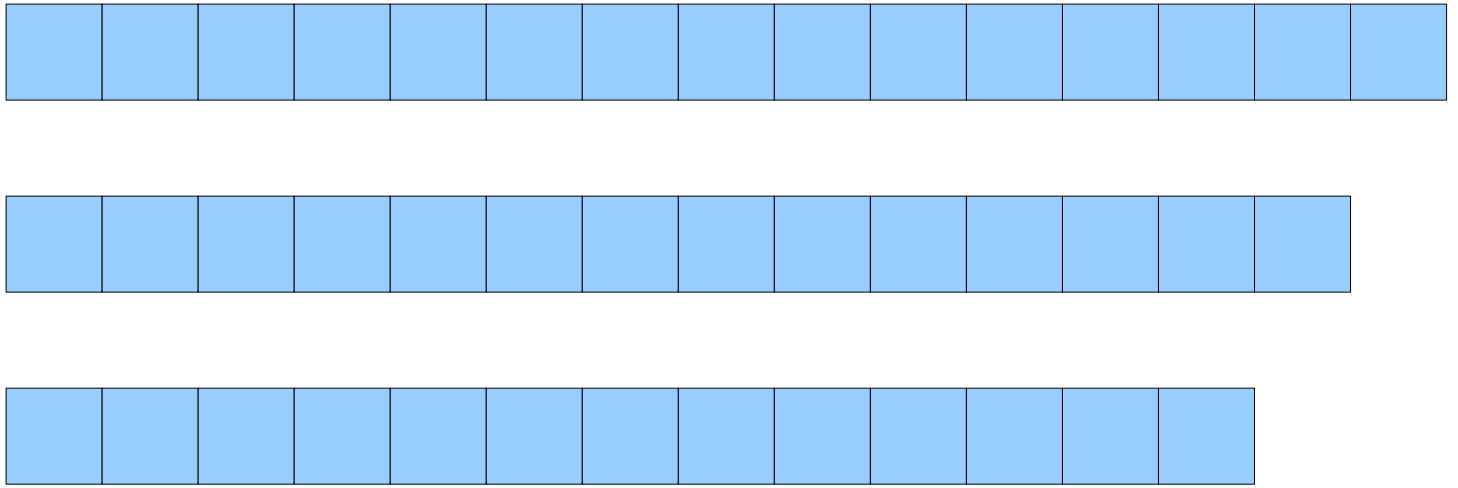
Suppose we get **unlucky**



Suppose we get **unlucky**



Suppose we get **unlucky**

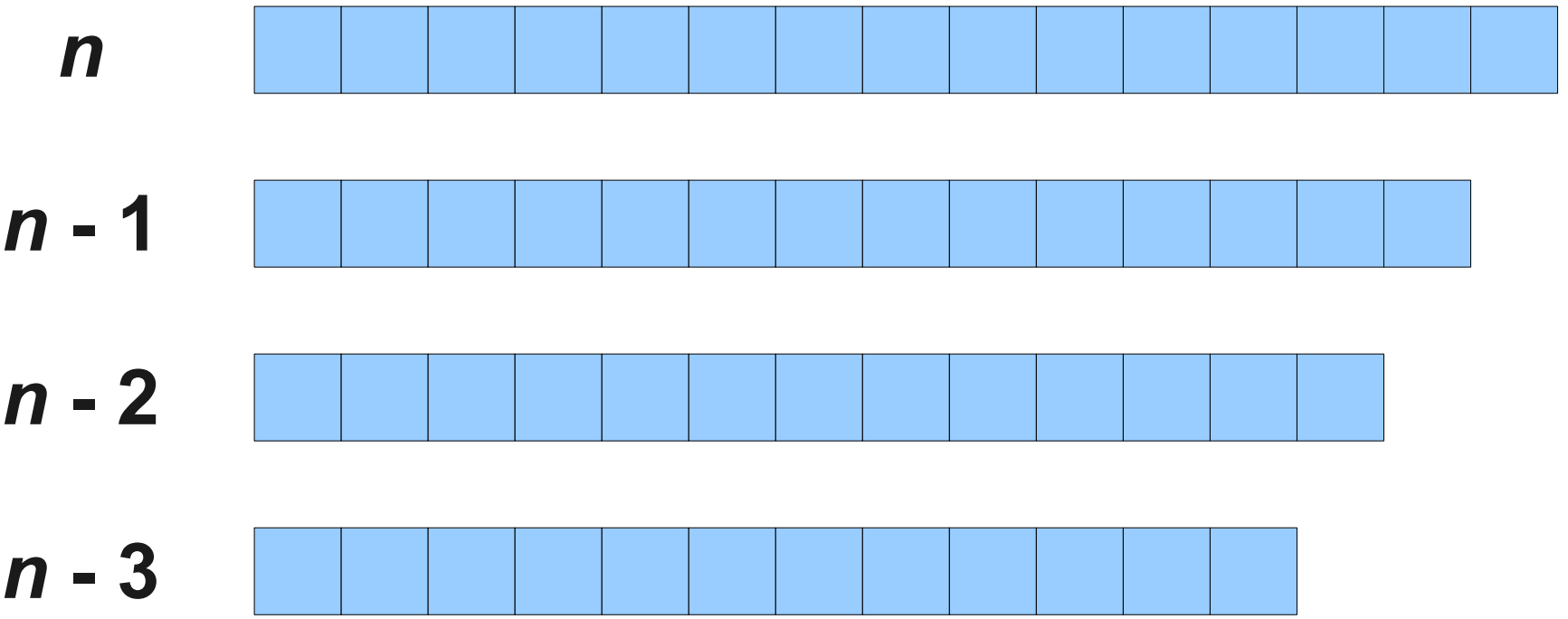


Suppose we get **unlucky**

Suppose we get **unlucky**

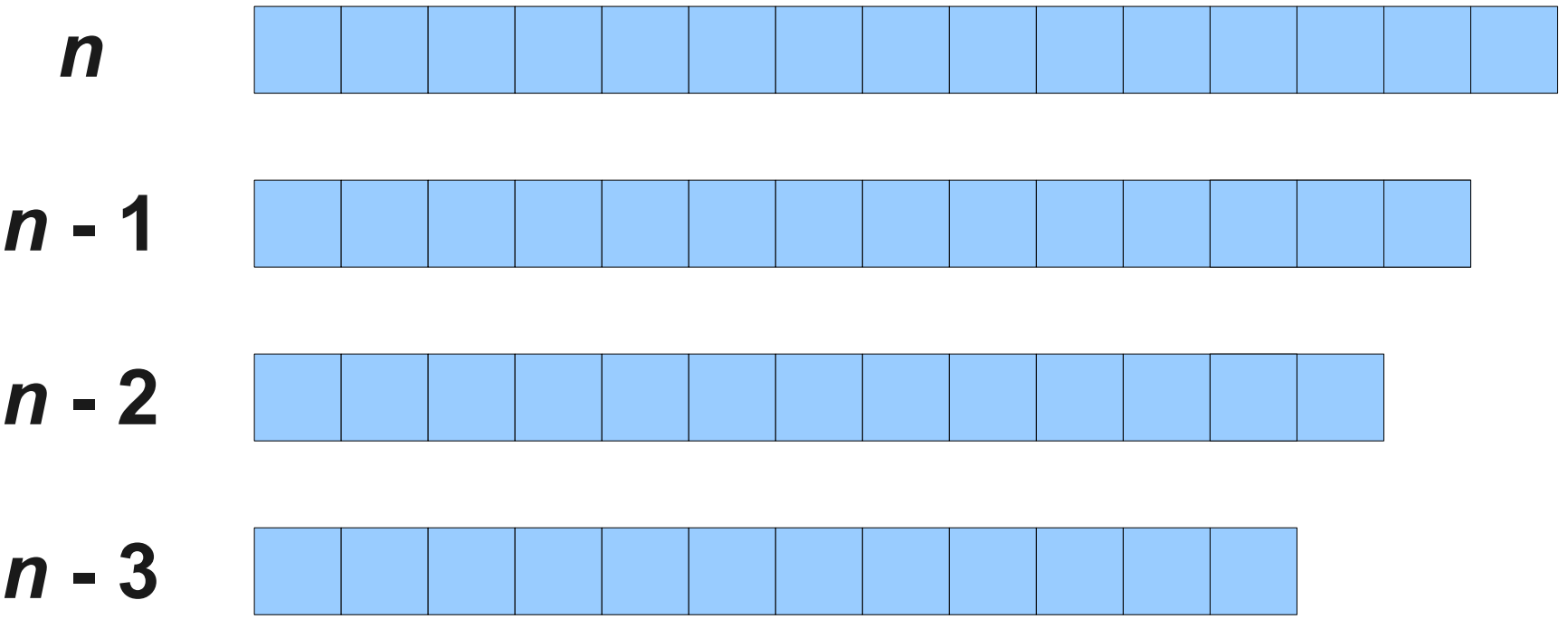
...

Suppose we get **unlucky**



...

Suppose we get **unlucky**



… $\mathbf{O(n^2)}$

Quicksort is Strange

- In most cases, quicksort has runtime $O(n \log n)$.
- In the worst case, quicksort has runtime $O(n^2)$.
- How can you avoid this?
- **Pick better pivots!**
  - Pick the median.
    - Can be done in $O(n)$, but *expensive* $O(n)$.
  - Pick the "median-of-three."
    - Better than nothing, but still can hit worst case.
  - Pick randomly.
    - Extremely low probability of $O(n^2)$.

Quicksort is Fast

- Although quicksort is $O(n^2)$ in the worst case, it is one of the fastest known sorting algorithms.
- $O(n^2)$ behavior is extremely unlikely with random pivots; runtime is usually a very good $O(n \log n)$.
- It's hard to argue with the numbers...

Timing Quicksort

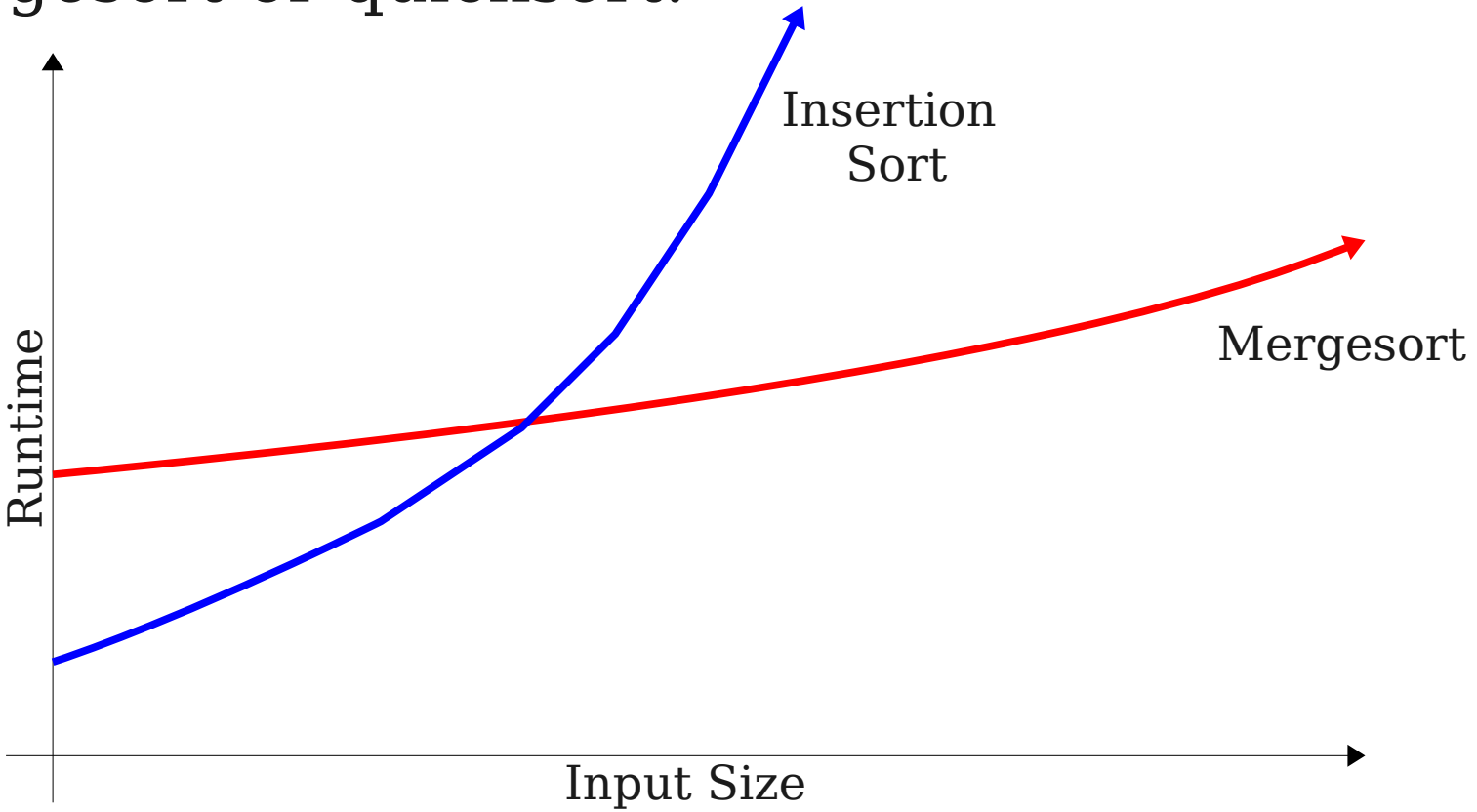
Size	Selection Sort	Insertion Sort	“Split Sort”	Mergesort
10000	0.304	0.160	0.161	0.006
20000	1.218	0.630	0.387	0.010
30000	2.790	1.427	0.726	0.017
40000	4.646	2.520	1.285	0.021
50000	7.395	4.181	2.719	0.028
60000	10.584	5.635	2.897	0.035
70000	14.149	8.143	3.939	0.041
80000	18.674	10.333	5.079	0.042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

Timing Quicksort

Size	Selection Sort	Insertion Sort	“Split Sort”	Mergesort	Quicksort
10000	0.304	0.160	0.161	0.006	0.001
20000	1.218	0.630	0.387	0.010	0.002
30000	2.790	1.427	0.726	0.017	0.004
40000	4.646	2.520	1.285	0.021	0.005
50000	7.395	4.181	2.719	0.028	0.006
60000	10.584	5.635	2.897	0.035	0.008
70000	14.149	8.143	3.939	0.041	0.009
80000	18.674	10.333	5.079	0.042	0.009
90000	23.165	12.832	6.375	0.048	0.012

An Interesting Observation

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



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);
    }
}
```

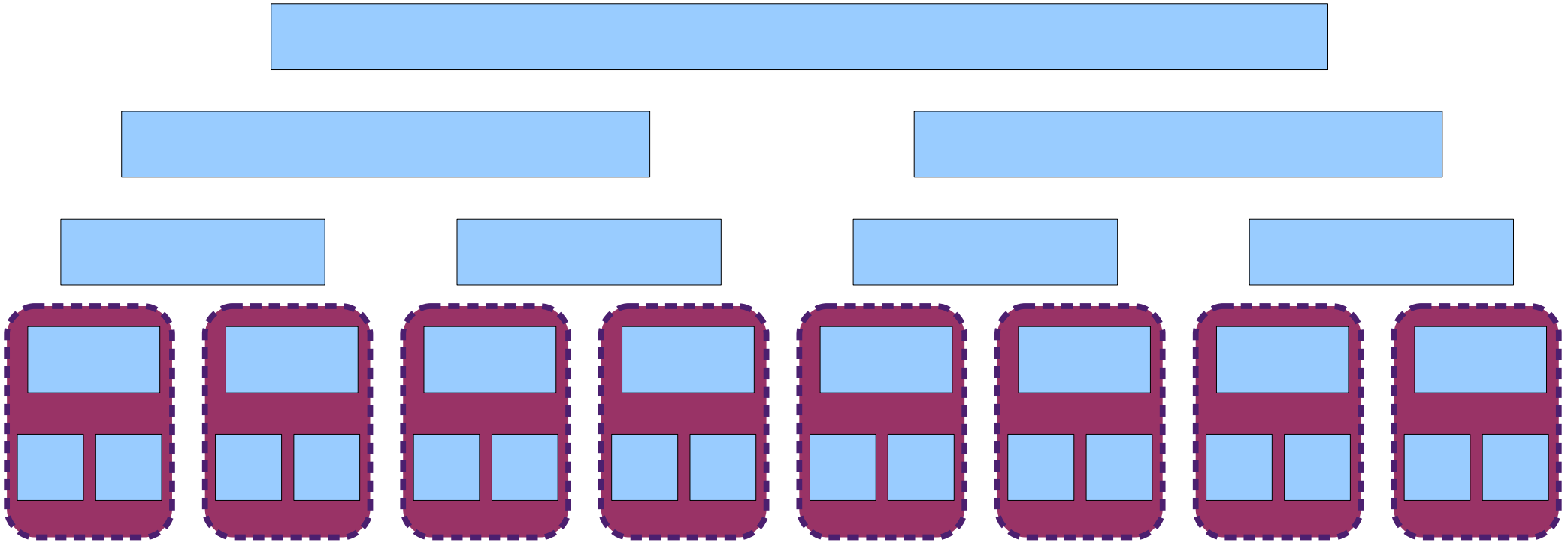
Hybrid Mergesort

```
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);
    }
}
```

Hybrid Sorting Algorithms



Runtime for Hybrid Mergesort

Size	Mergesort	Hybrid Mergesort	Quicksort
100000	0.063	0.019	0.012
300000	0.176	0.061	0.060
500000	0.283	0.091	0.063
700000	0.396	0.130	0.089
900000	0.510	0.165	0.118
1100000	0.608	0.223	0.151
1300000	0.703	0.246	0.179
1500000	0.844	0.28	0.215
1700000	0.995	0.326	0.243
1900000	1.070	0.355	0.274

Hybrid Sorts in Practice

- Introspective Sort (*Introsort)*
  - Based on quicksort, insertion sort, and ***heapsort***.
  - Heapsort is $O(n \log n)$ and a bit faster than mergesort.
  - Uses quicksort, then switches to heapsort if it looks like the algorithm is degenerating to $O(n^2)$.
  - Uses insertion sort for small inputs.
  - Gains the raw speed of quicksort without any of the drawbacks.

Runtime for Introsort

Size	Mergesort	Hybrid Mergesort	Quicksort
100000	0.063	0.019	0.012
300000	0.176	0.061	0.060
500000	0.283	0.091	0.063
700000	0.396	0.130	0.089
900000	0.510	0.165	0.118
1100000	0.608	0.223	0.151
1300000	0.703	0.246	0.179
1500000	0.844	0.28	0.215
1700000	0.995	0.326	0.243
1900000	1.070	0.355	0.274

Runtime for Introsort

Size	Mergesort	Hybrid Mergesort	Quicksort	Introsort
100000	0.063	0.019	0.012	0.009
300000	0.176	0.061	0.060	0.028
500000	0.283	0.091	0.063	0.043
700000	0.396	0.130	0.089	0.060
900000	0.510	0.165	0.118	0.078
1100000	0.608	0.223	0.151	0.092
1300000	0.703	0.246	0.179	0.107
1500000	0.844	0.28	0.215	0.123
1700000	0.995	0.326	0.243	0.139
1900000	1.070	0.355	0.274	0.158

We've spent all of our time talking about **fast** and **efficient** sorting algorithms.

However, we have neglected to find **slow** and **inefficient** sorting algorithms.

Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms

Hermann Gruber[1] and Markus Holzer[2] and Oliver Ruepp[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München,
Oettingenstraße 67, D-80538 München, Germany
email: gruberh@tcs.ifi.lmu.de
[2] Institut für Informatik, Technische Universität München,
Boltzmannstraße 3, D-85748 Garching bei München, Germany
email: {holzer,ruepp}@in.tum.de

Introducing **Bogosort**

Next Time

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