Algorithms Case Study: Sorting

Quote of the Week: “It would be reasonable to suppose that a routine time or an eventless time would seem interminable. It should be so, but it is not. It is the dull eventless times that have no duration whatsoever. A time splashed with interest, wounded with tragedy, crevassed with joy - that’s the time that seems long in the memory. … Eventlessness has no posts to drape duration on. From nothing to nothing is no time at all.”

Next week’s labs are optional labs

- Each worth 1 extra credit point
- Monday’s lab is special — regex puzzle hunt!

I know it’s soon, but…

- You have a final in 1.5 weeks
- It looks like it’ll be the same difficulty as midterm 2
- Expect it to be fully cumulative

Midterm 2

- Certain questions on midterm 2 didn't have as high averages as I hoped
- So I feel compelled to reteach these concepts

Big O Set

- When using big O notation, we like to write things like:
    - The runtime of our program is **in** $O(n)$
- What does this mean? Why are we using the word “in”?

Big O Set

- We use the word “in” because O(n) is actually a **set**. In fact, it is a set of functions.
- By claiming that the runtime of a program is in O(n), we are claiming that the runtime of our program can be expressed by a function that the set O(n) **contains**

Big O Set

- In general, we can make statements like
  - f(n) is in O(g(n))
- We claim that some function **f(n)** is in the set of functions **O(g(n))** — a set that looks like it has something to do with the function **g(n)**

Big O Set

- **O(g(n))** can be thought of as the set of functions that grow similarly to **g(n)** as **n** gets big
  - For example, **O(n)** is the set of functions that grow similarly to the function **g(n) = n**
- To decided whether a particular function f(n) was in this set, we use the following condition:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Big O Set

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

- This essentially just means that **f(n)** isn't a *lot* bigger than **g(n)**
- How do we represent this condition in Java? It's kinda difficult. Luckily, there is a shortcut condition for polynomials.

Big O Set

- For big O with polynomials, we decided that constant factors didn't matter, and only the highest term mattered
    - To decide if **$2N^2 + 3N + 4$** is in **$O(5N + 7)$** we ignore the constants multiplied to each term, and just consider the top terms
    - Because **N^2** is bigger than **N**, **$2N^2 + 3N + 4$** is NOT in **$O(5N + 7)$**

Big O Set

- But anything with a highest term of **N,** or lower, would be in **O(5N + 7)**
- For example, **N** is in **O(5N + 7)**. So is **2N + 3.** So is **10N + 1000**. So is **4**. And so on
- We decide there are **infinitely** many functions in **O(5N + 7)**

Big O Set

```
public class BigO {
  int myDegree;

  public BigO(Polynomial p) {
    myDegree = p.myCoefficients.length;
  }

  private double size() {
    return Double.POSITIVE_INFINITY;
  }

  public boolean contains(Polynomial p) {
    return myDegree >= p.myCoefficients.length;
  }
}
```

Quiz: Redo Bookstore

- This was the most important question on the midterm
- This question gets at the heart of what the class is about

Quiz part 2: Bookstore

- Design a data structure where you can…
  - Add a book with an author **O(1)**
  - Remove a book **O(1)**
  - Find the author of a book **O(1)**
  - Print all books by an author **O(b)**
  - Print all books in the order they were added **O(B)**

Writing efficient programs

- In 61A, you learned to program
- In 61BL, you are learning to program *well*

Choosing efficient data structures

- As we've seen, different data structures can have different runtimes for basic operations
    - For example, checking if an **ArrayList** contains a certain item is slow, but checking if a **HashSet** contains a certain item is fast
- When programming, you should be sure to choose the data structure that makes sense for your problem

Choosing efficient algorithms

- But choosing data structures isn't everything
- Sometimes choosing the problem-solving strategy, or the *algorithm*, makes a big difference

Example: sorting

- Problem: Given a list of numbers (or **Comparable** objects), arrange the list in order from smallest to largest (or vice versa)

My first sorting algorithm: bubble sort

- How could we sort an array of integers?
- **Idea 1**: Iterate through the array, and swap adjacent items if out of order

```
public static void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
      if (arr[i] > arr[i + 1]) {
        swap(arr, i, i + 1);
      }
    }
  }
```

My first sorting algorithm: bubble sort

4	3	1	7	2	8	5	0

My first sorting algorithm: bubble sort

Iterate and swap!

4	3	1	7	2	8	5	0

My first sorting algorithm: bubble sort

Iterate and swap!

3	4	1	7	2	8	5	0

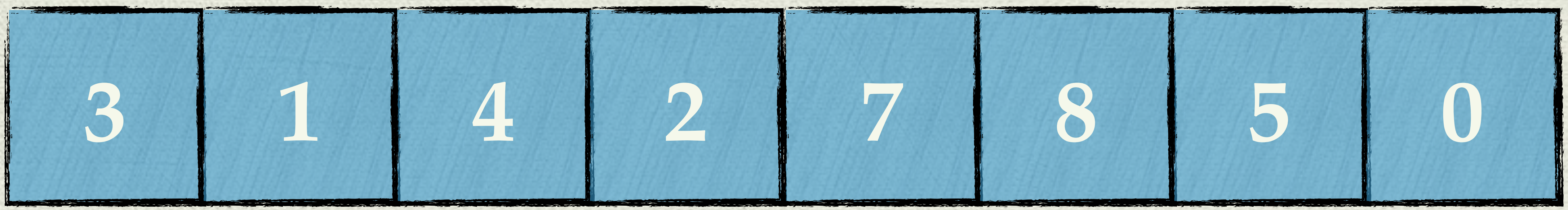
My first sorting algorithm: bubble sort

Iterate and swap!

3	1	4	7	2	8	5	0

My first sorting algorithm: bubble sort

Iterate and swap!



My first sorting algorithm: bubble sort

Iterate and swap!

3	1	4	2	7	5	8	0

My first sorting algorithm: bubble sort

Iterate and swap!

3	1	4	2	7	5	0	8

My first sorting algorithm: bubble sort

Still not sorted…

3	1	4	2	7	8	0	5

My first sorting algorithm: bubble sort

- How could we sort an array of integers?
- **Idea 1**: Iterate through the array, and swap adjacent items if out of order
- *This doesn't actually work*. Have to repeat the process multiple times, until no more need to swap

My first sorting algorithm: bubble sort

```
public static void bubbleSort(int[] arr) {
  boolean swappedSomething = true;

  while (swappedSomething) {
    swappedSomething = false;
    for (int i = 0; i < arr.length - 1; i++) {
      if (arr[i] > arr[i + 1]) {
        swap(i, i + 1);
        swappedSomething = true;
      }
    }
  }
}
```

My first sorting algorithm: bubble sort

- The code we just wrote implements a sorting algorithm called **bubble sort**
- Have we solved the problem of sorting?

Take it from the President

- **Eric Schmidt**: “What is the most efficient way to sort a million 32-bit integers?”
- **Barack Obama**: “I think the bubble sort would be the wrong way to go.”

Take it from the President

- Oh no.

What's wrong with bubble sort?

- In the worst case, the runtime of bubble sort will be **O(N²)**, where there are **N** items we are sorting
  - We may have to repeat the loop in the worst case **N** times
- Can we do better?

Runtime hierarchy

- Sorting N items…
  - **$O(N^2)$**: bad for a sorting algorithm
  - **$O(N \log N)$**: normal for a sorting algorithm
  - **$O(N)$**: the ideal
  - **$O(\log N)$**: probably not going to happen

So many sorting algorithms

- **Bubble sort**
- **Selection sort**
    - **Heapsort** (selection sort with a priority queue)
- **Insertion sort**
- **Merge sort**
- **Quicksort**

So many sorting algorithms

- Bubble sort
- Selection sort
  - Heapsort
- Insertion sort

You coded all these in lab a long time ago

- Merge sort
- Quicksort

These are new for this week, so I'll go over them

Merge sort

- The algorithm:
  - **Step 1**: Split your list of items in half
  - **Step 2**: Recursively merge sort each half
  - **Step 3**: Merge the two now sorted halves into a sorted whole

Merge sort walkthrough

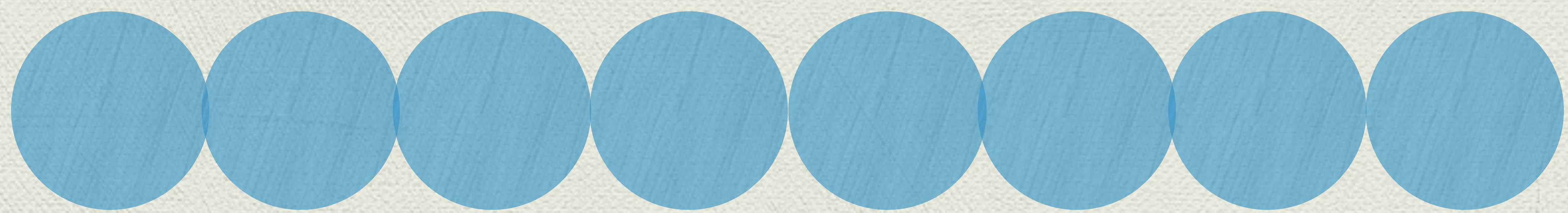
- The key is that taking two lists that are individually sorted, and then merging them into one bigger list that is sorted, is easy to do
- If between them the lists have N items, then the merge step takes O(N) time
- You already coded merge in the linked list labs

Merge sort runtime

- What is the runtime of merge sort?
- A picture will help illustrate it…

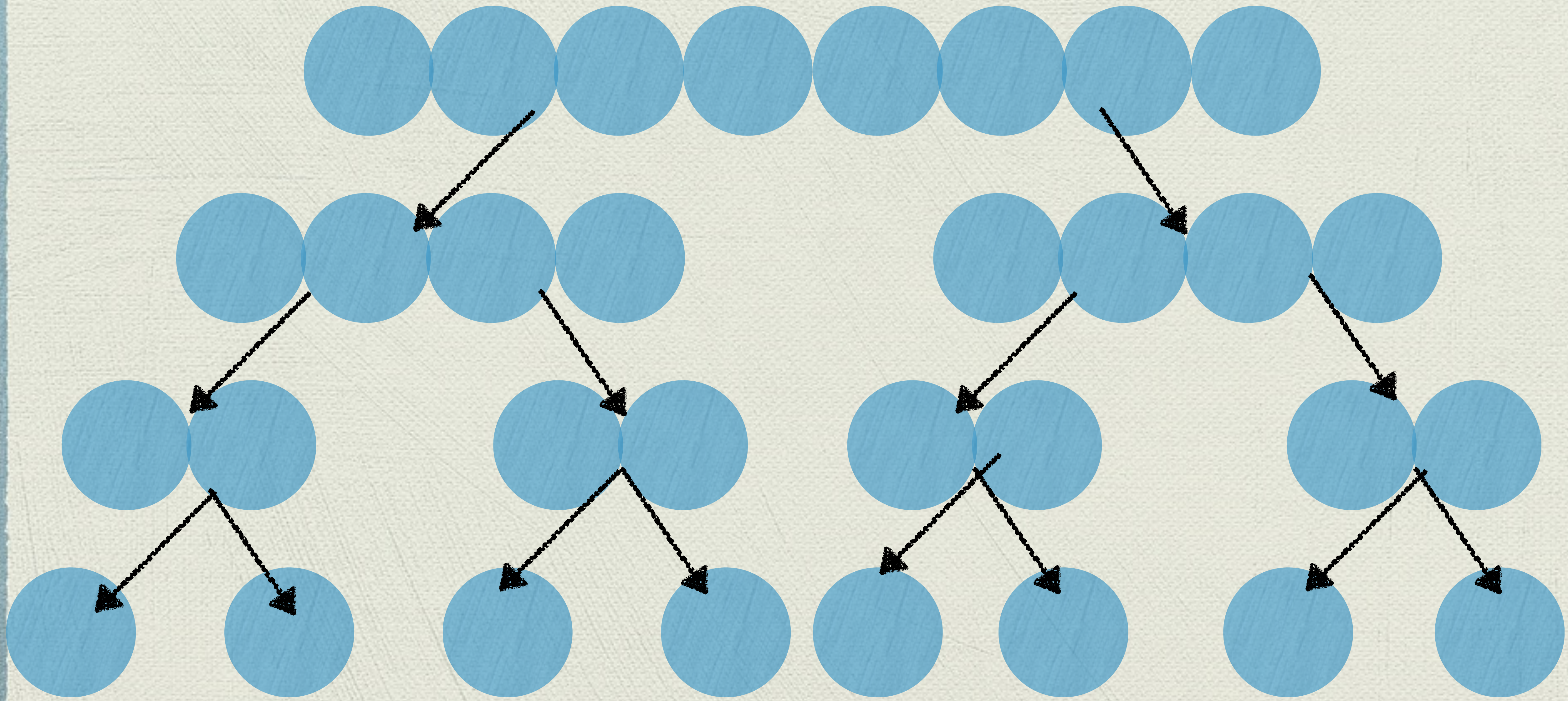
Merge sort runtime

- Say we start with N items



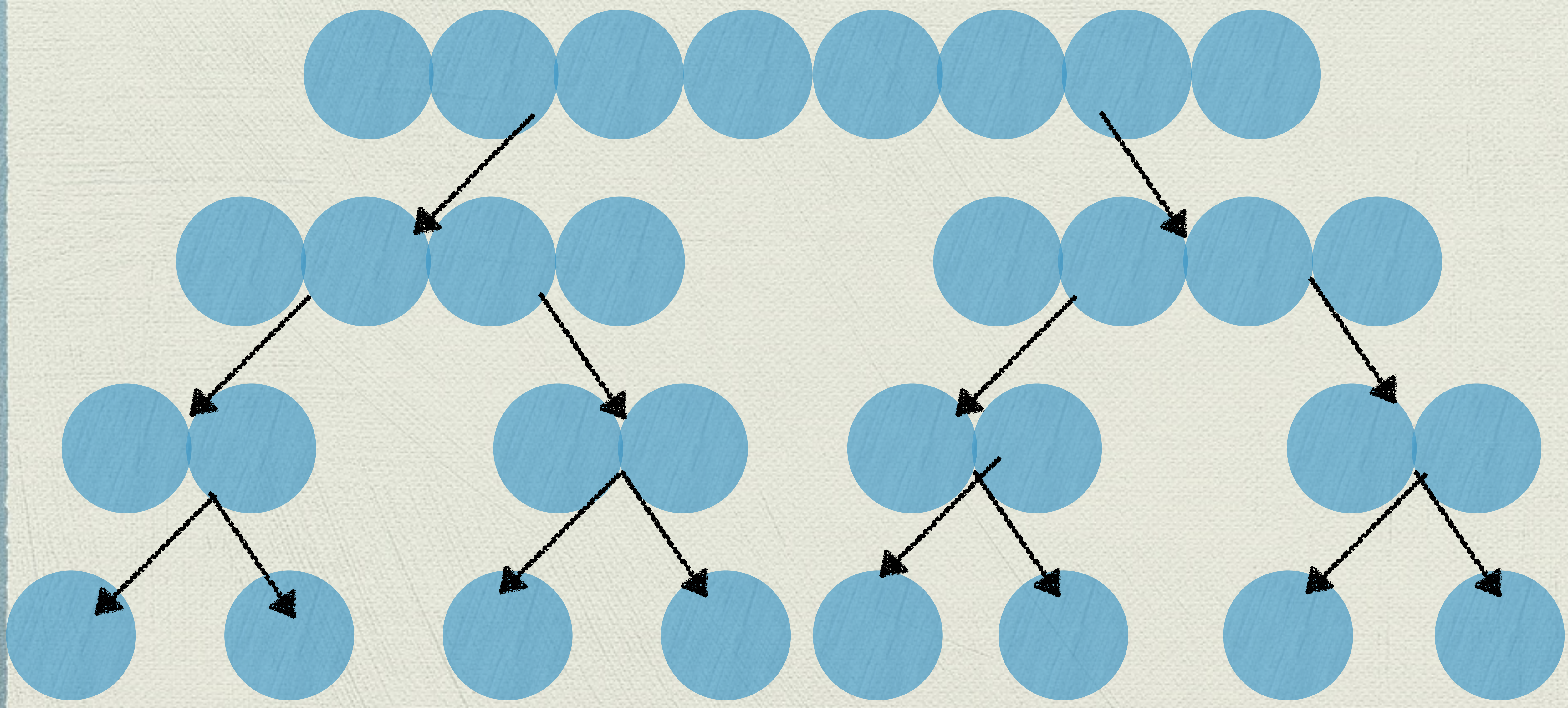
Merge sort runtime

- At each step, we divide in two…

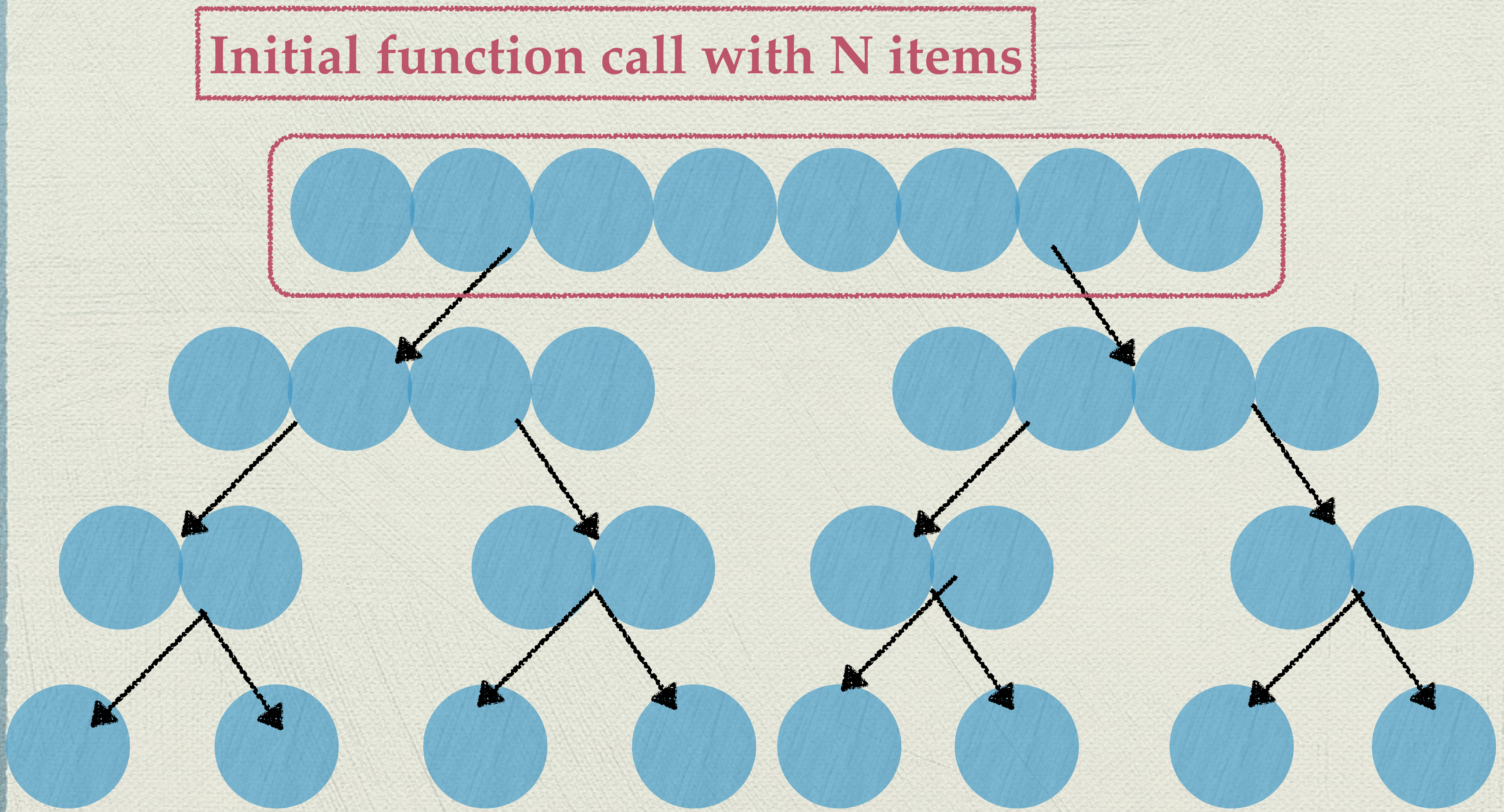


Merge sort runtime

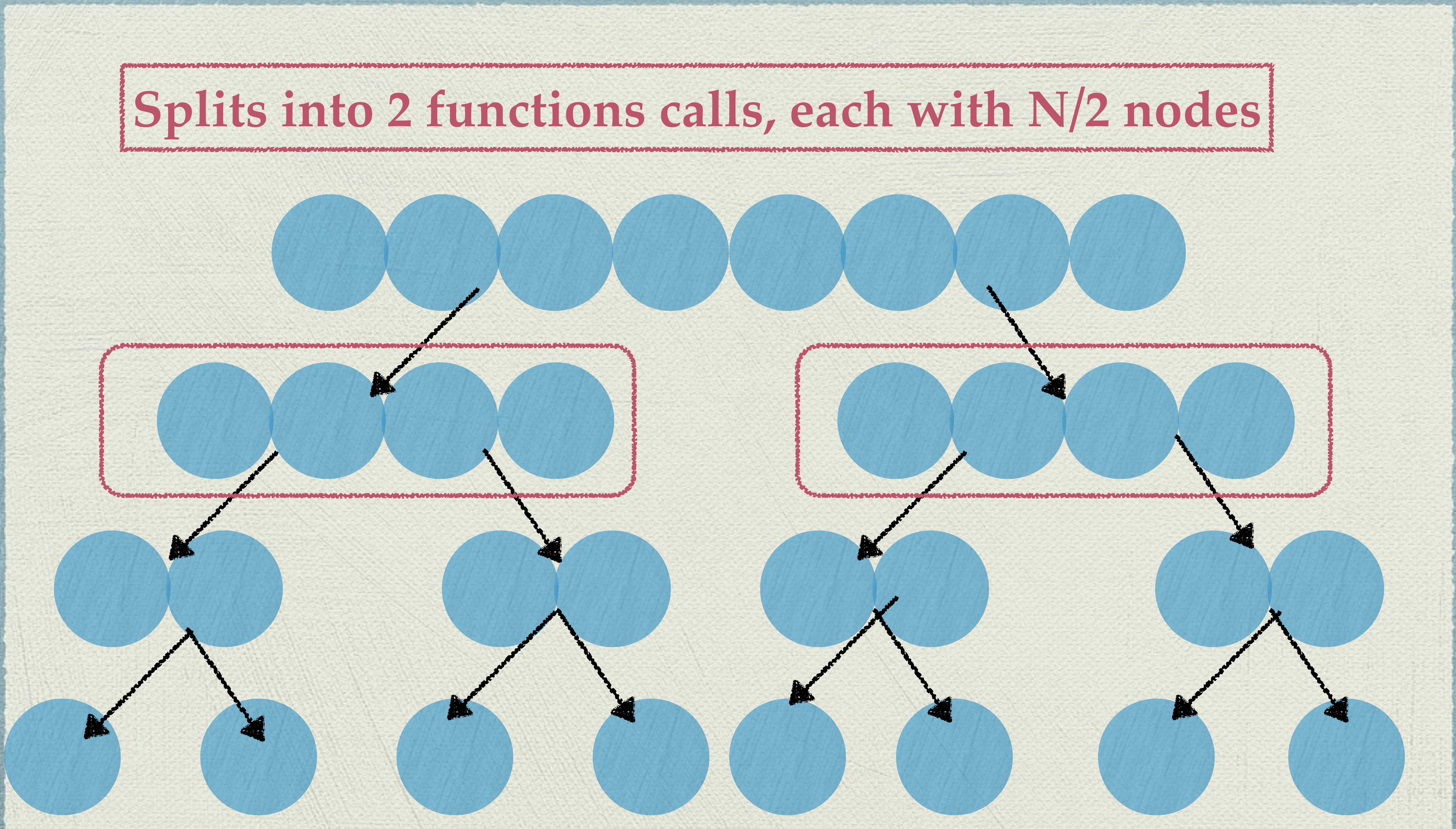
- Each group represents a recursive call



Merge sort runtime

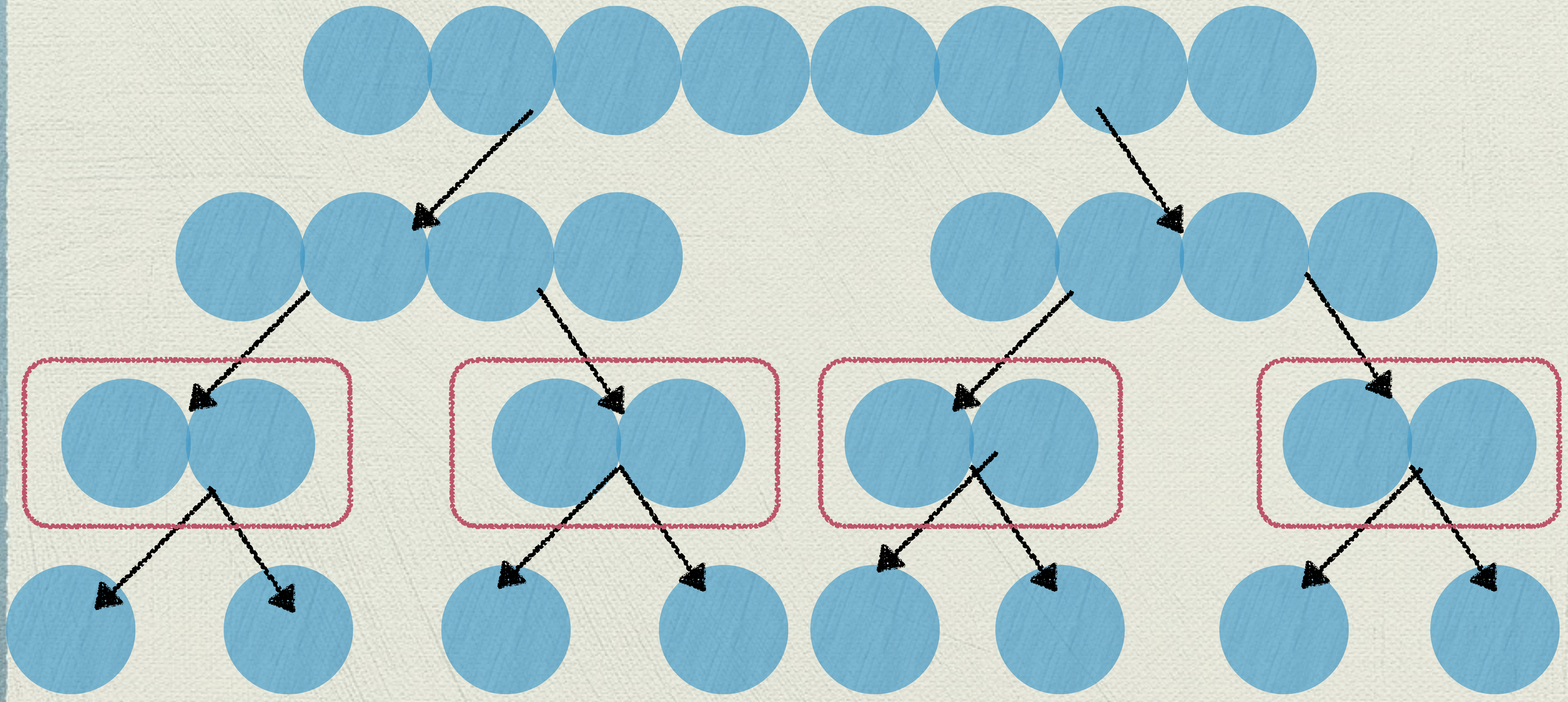


Merge sort runtime

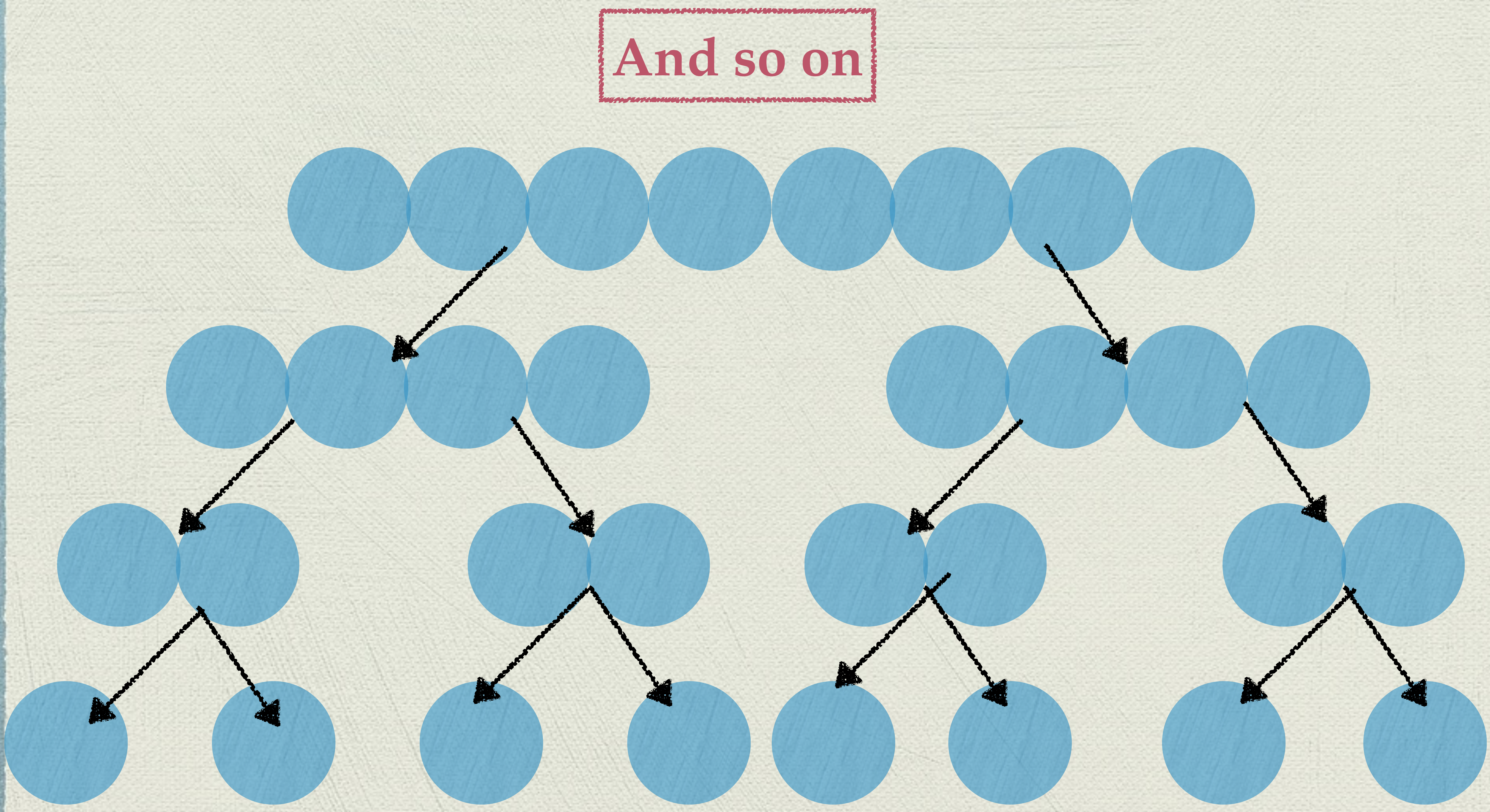


Merge sort runtime

Each of which splits into 2 more, each with N/4 nodes

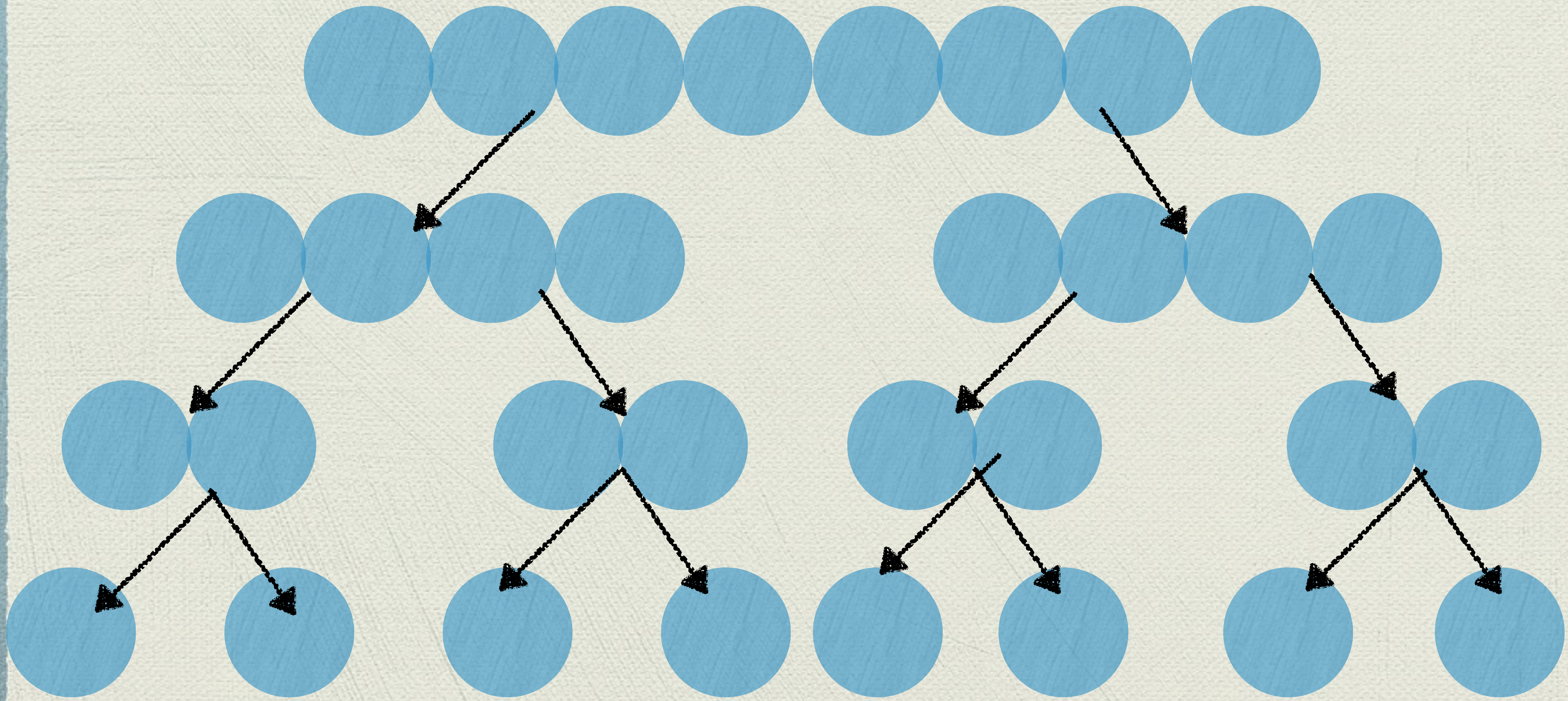


Merge sort runtime



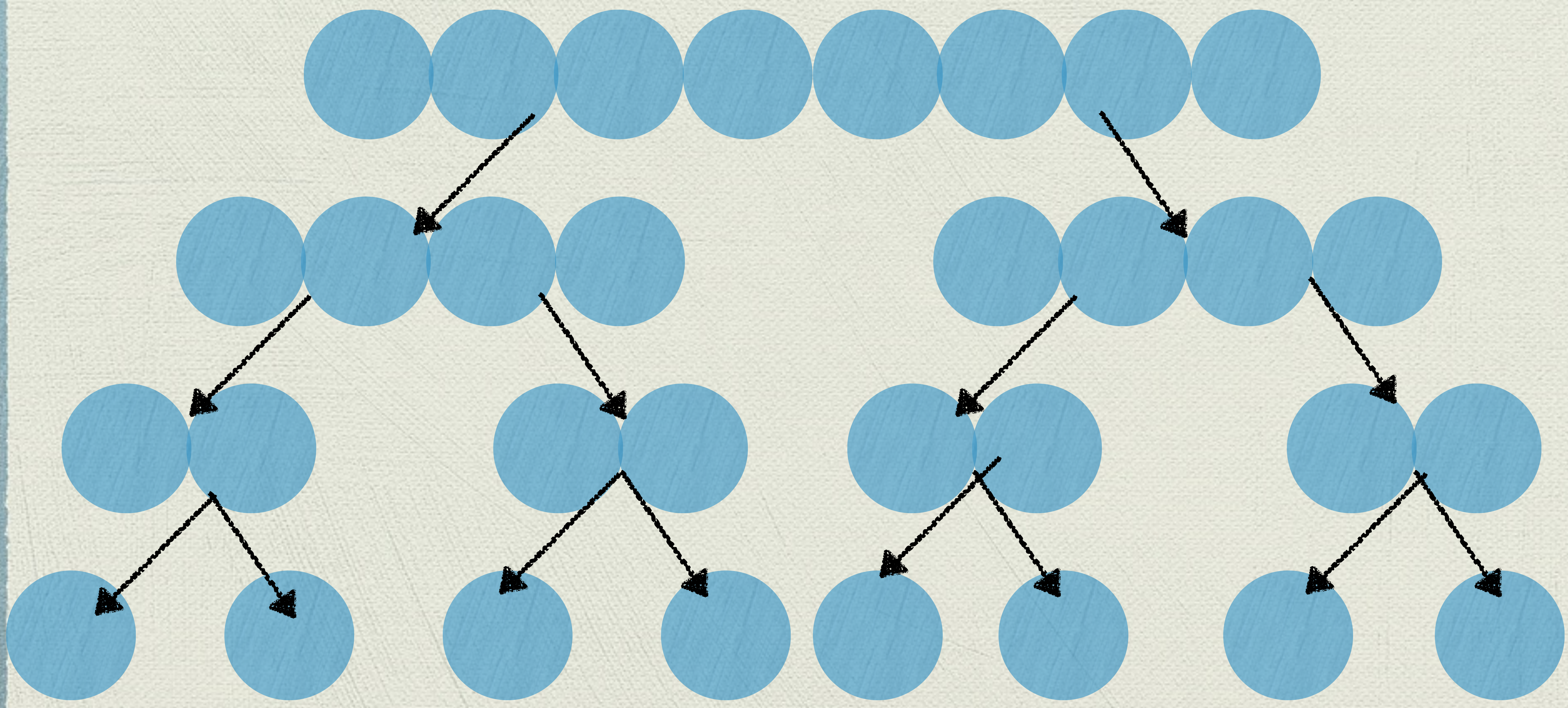
Merge sort runtime

- How much time does each function call take?

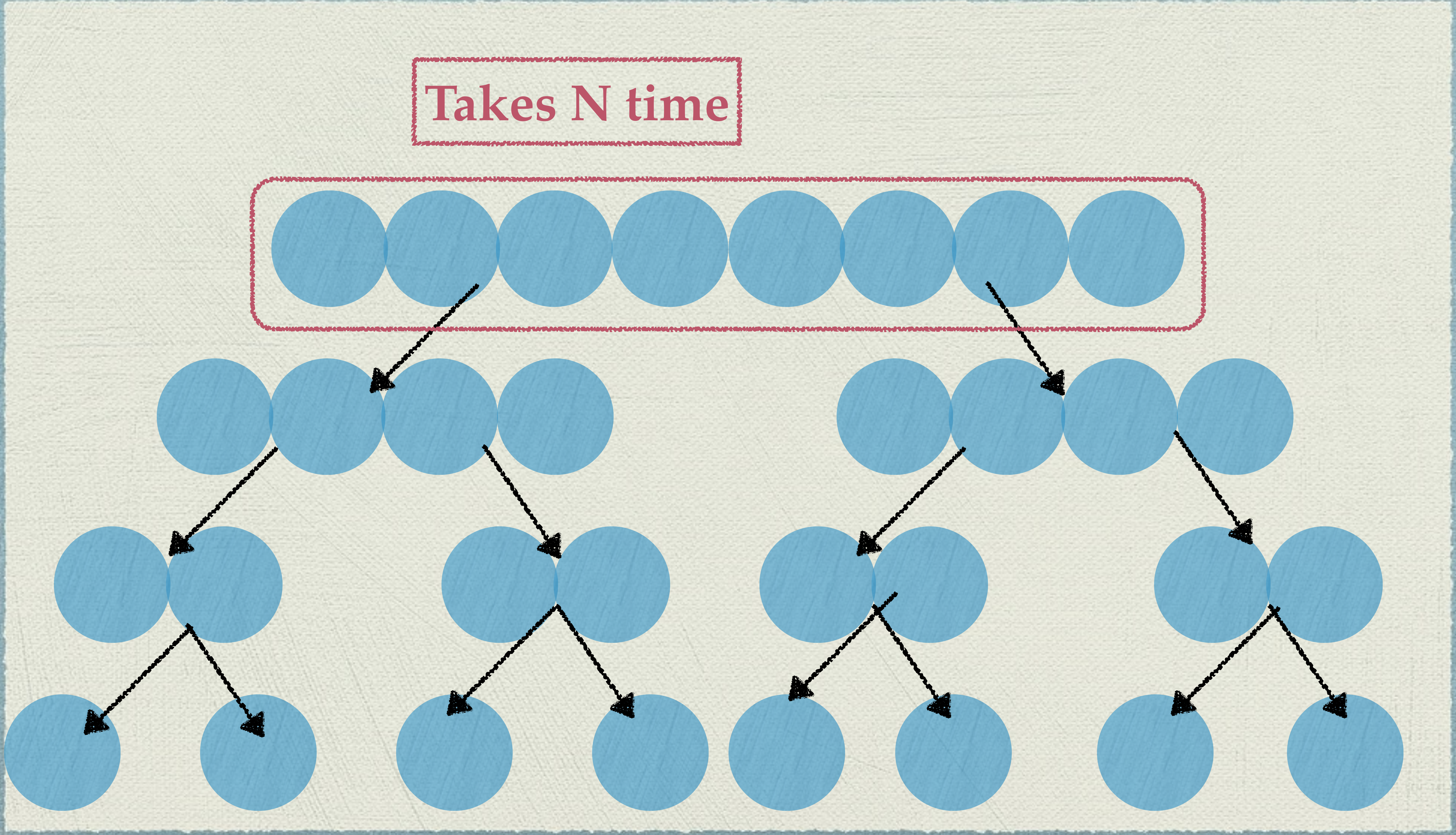


Merge sort runtime

- Each function call has to merge, which takes time linear with the number of nodes in the function

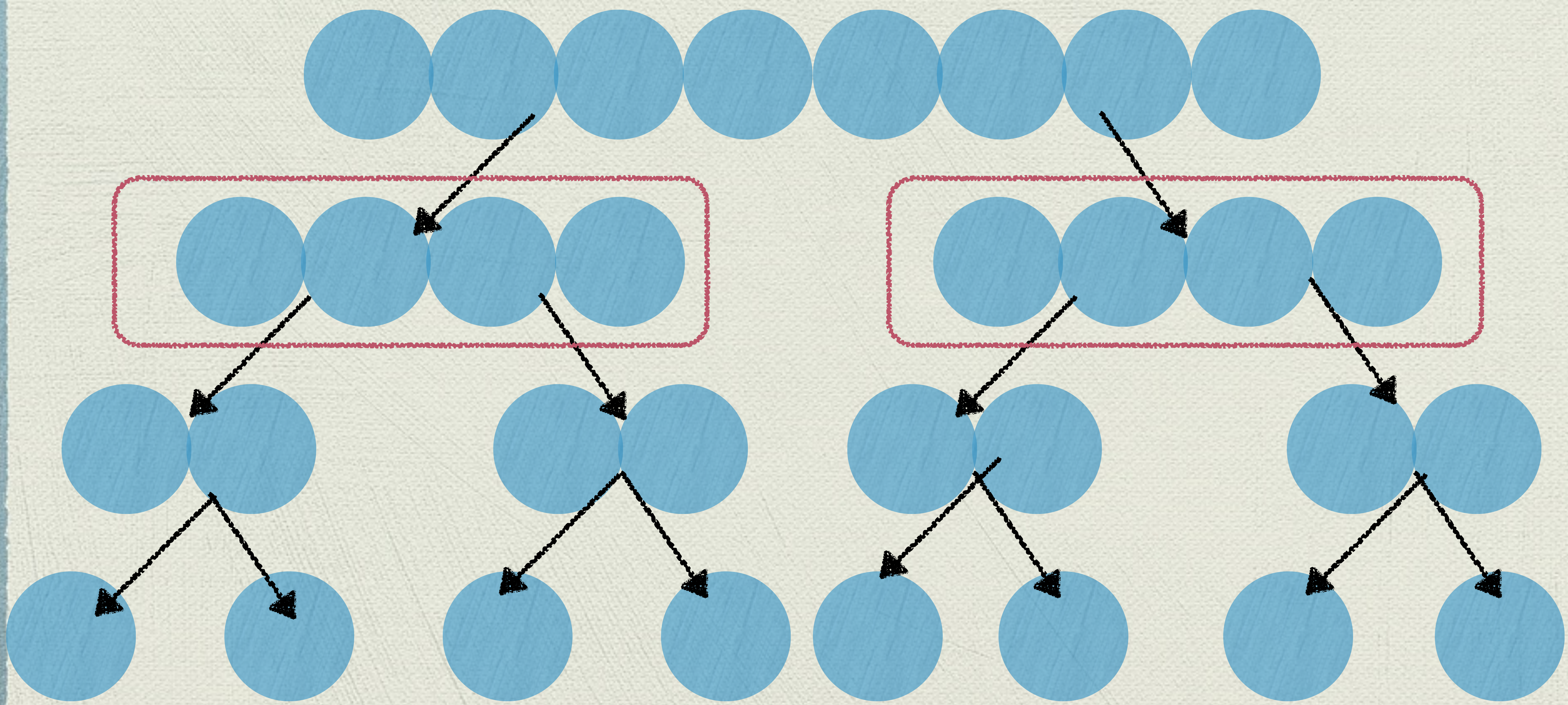


Merge sort runtime



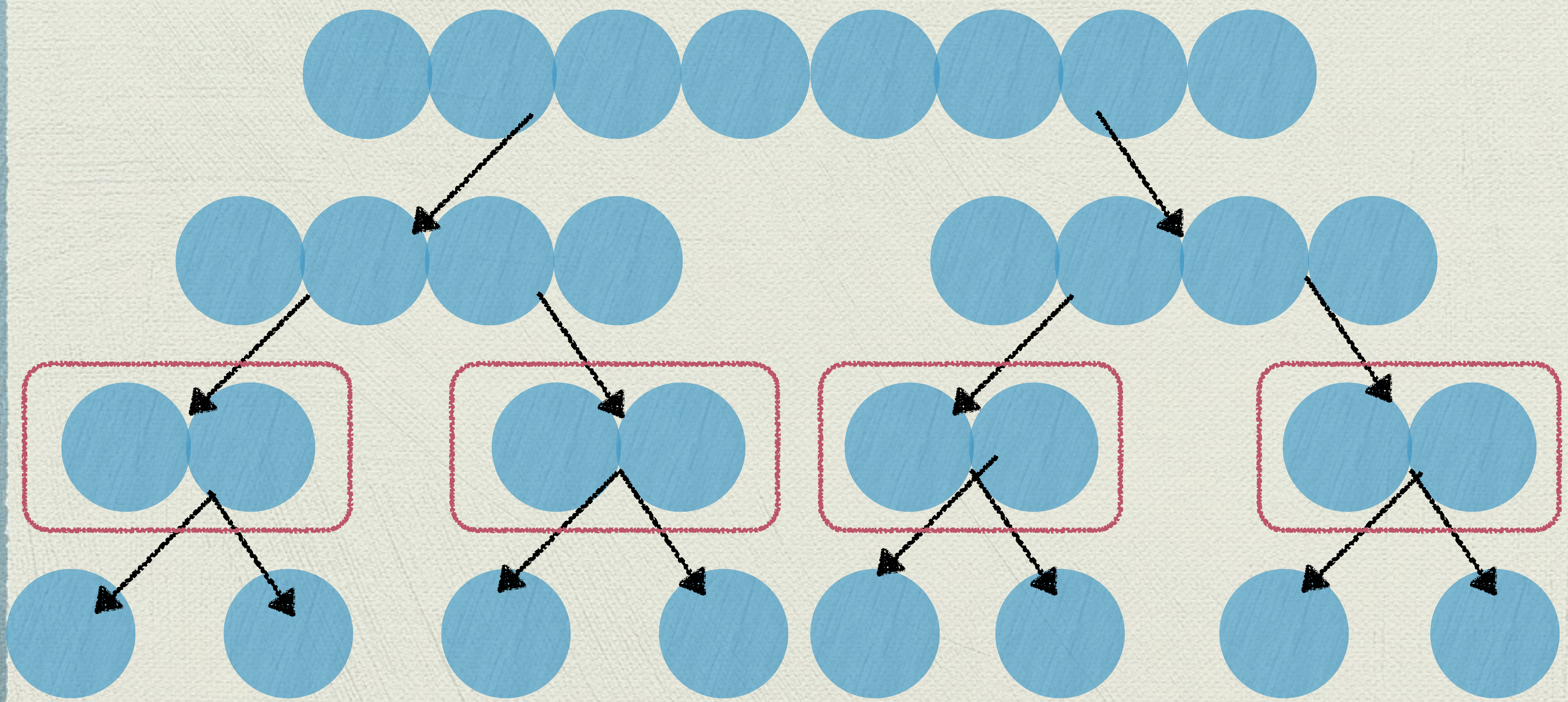
Merge sort runtime

Each one takes N/2 time. In total, N/2 + N/2 = N time

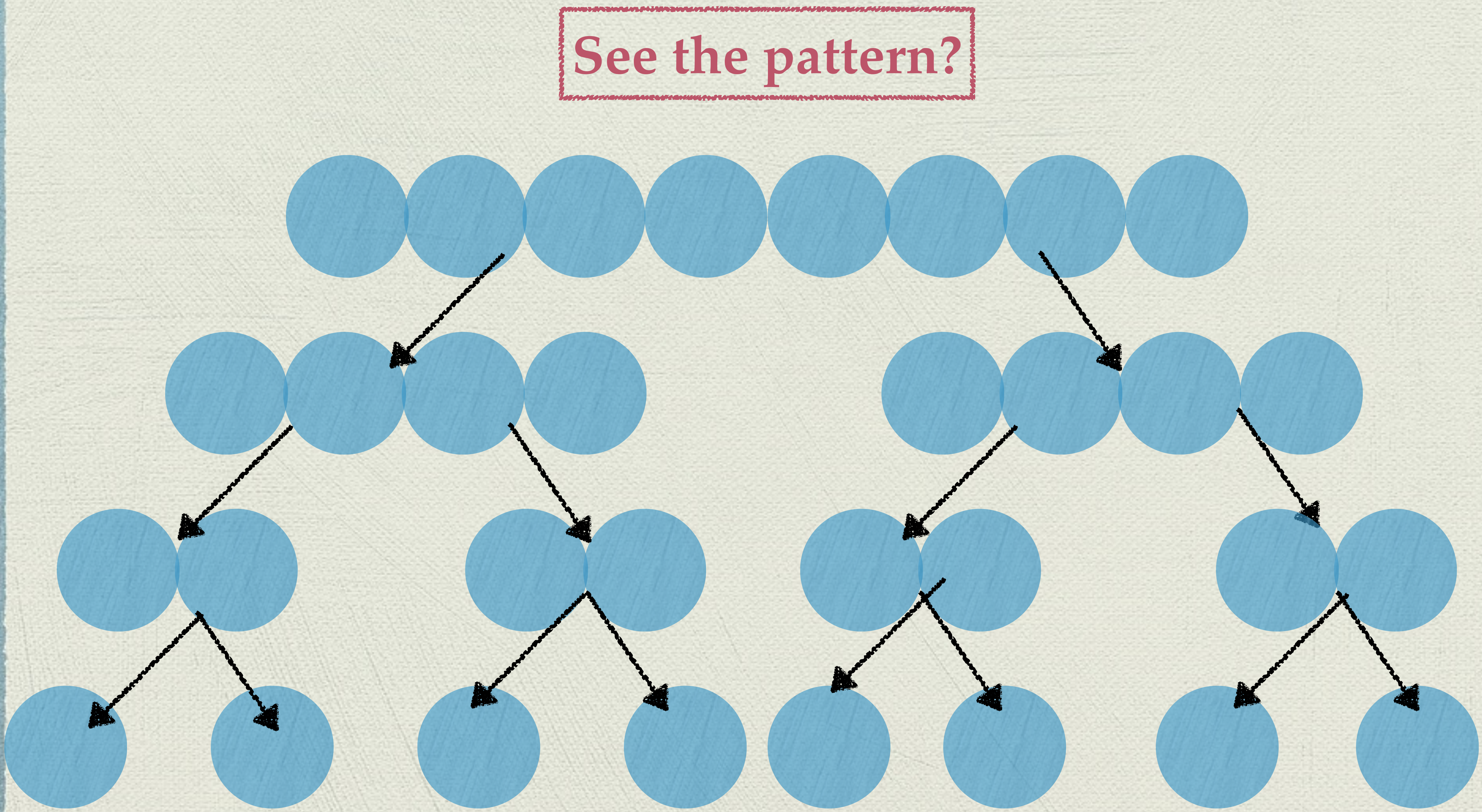


Merge sort runtime

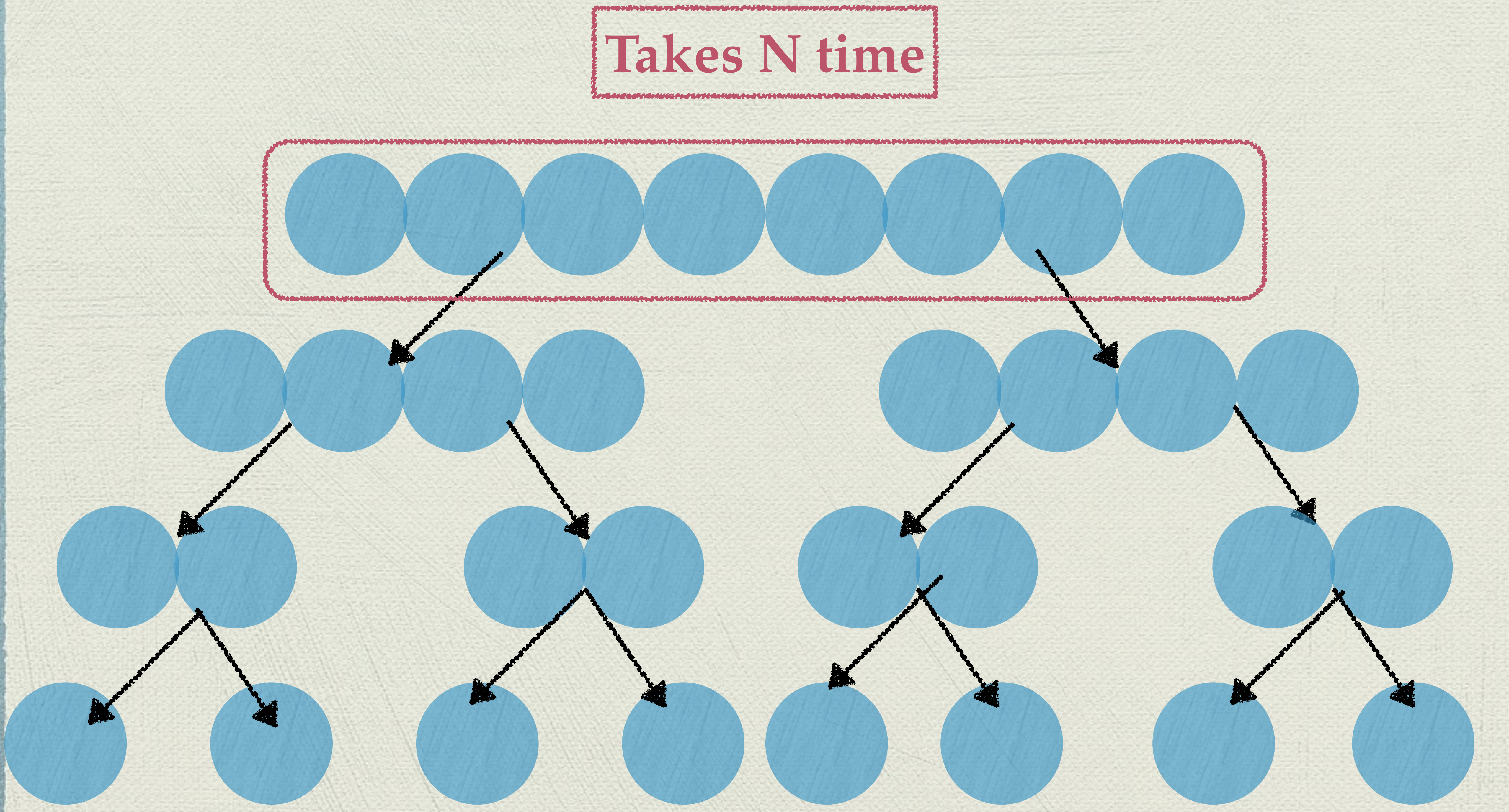
Each one takes N/4 time. In total, N/4 + N/4 + N/4 + N/4 = N



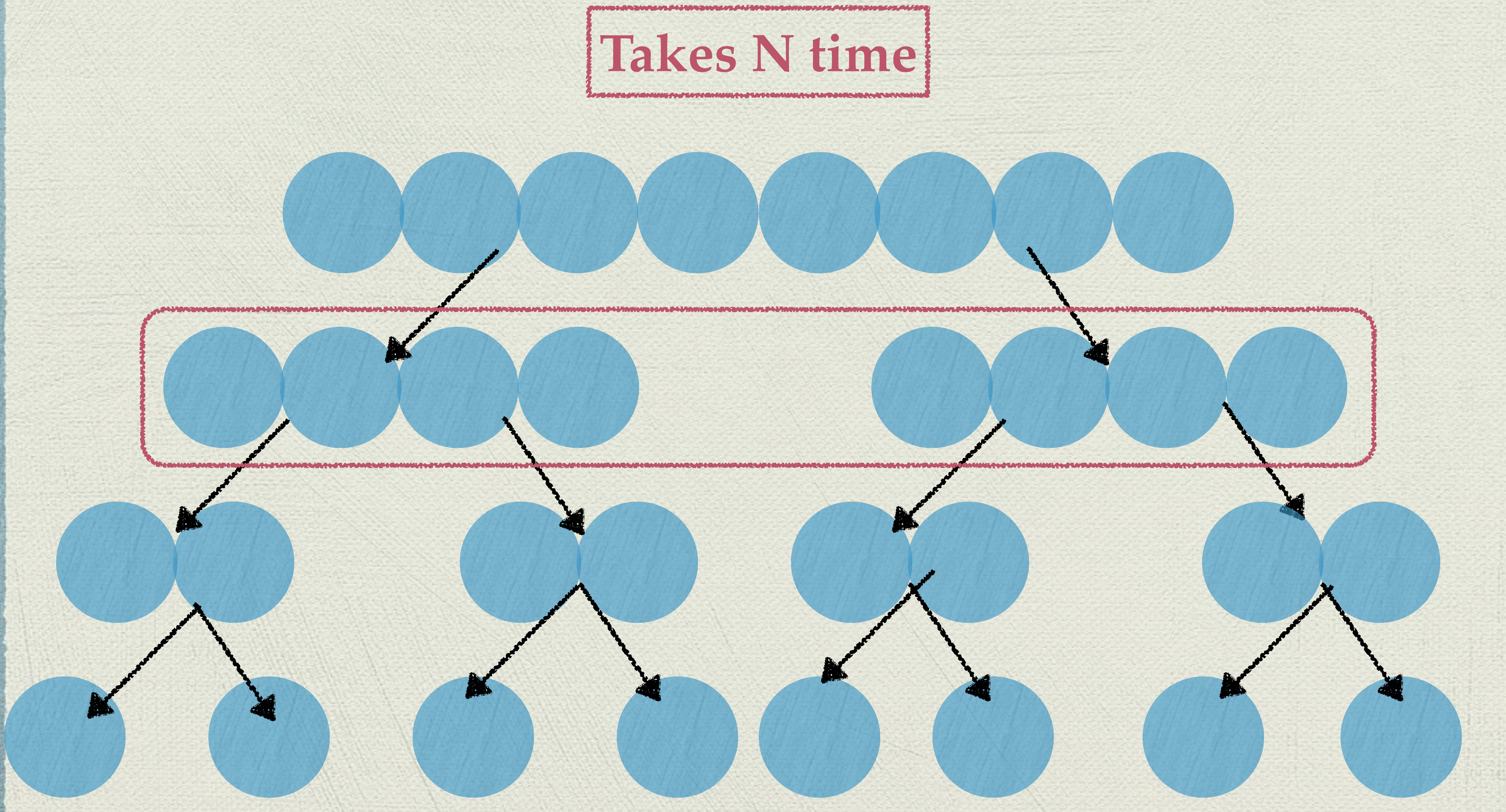
Merge sort runtime



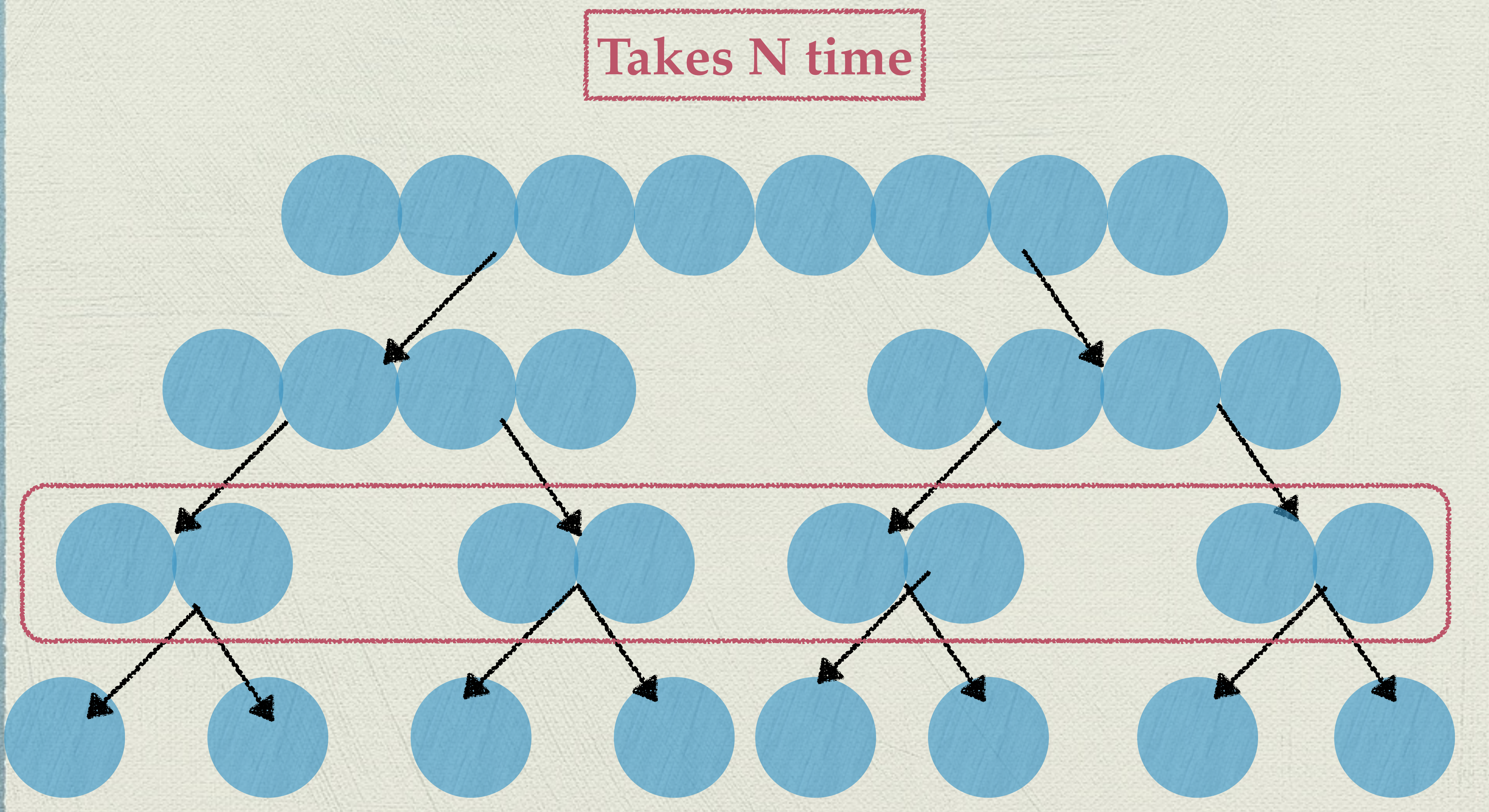
Merge sort runtime



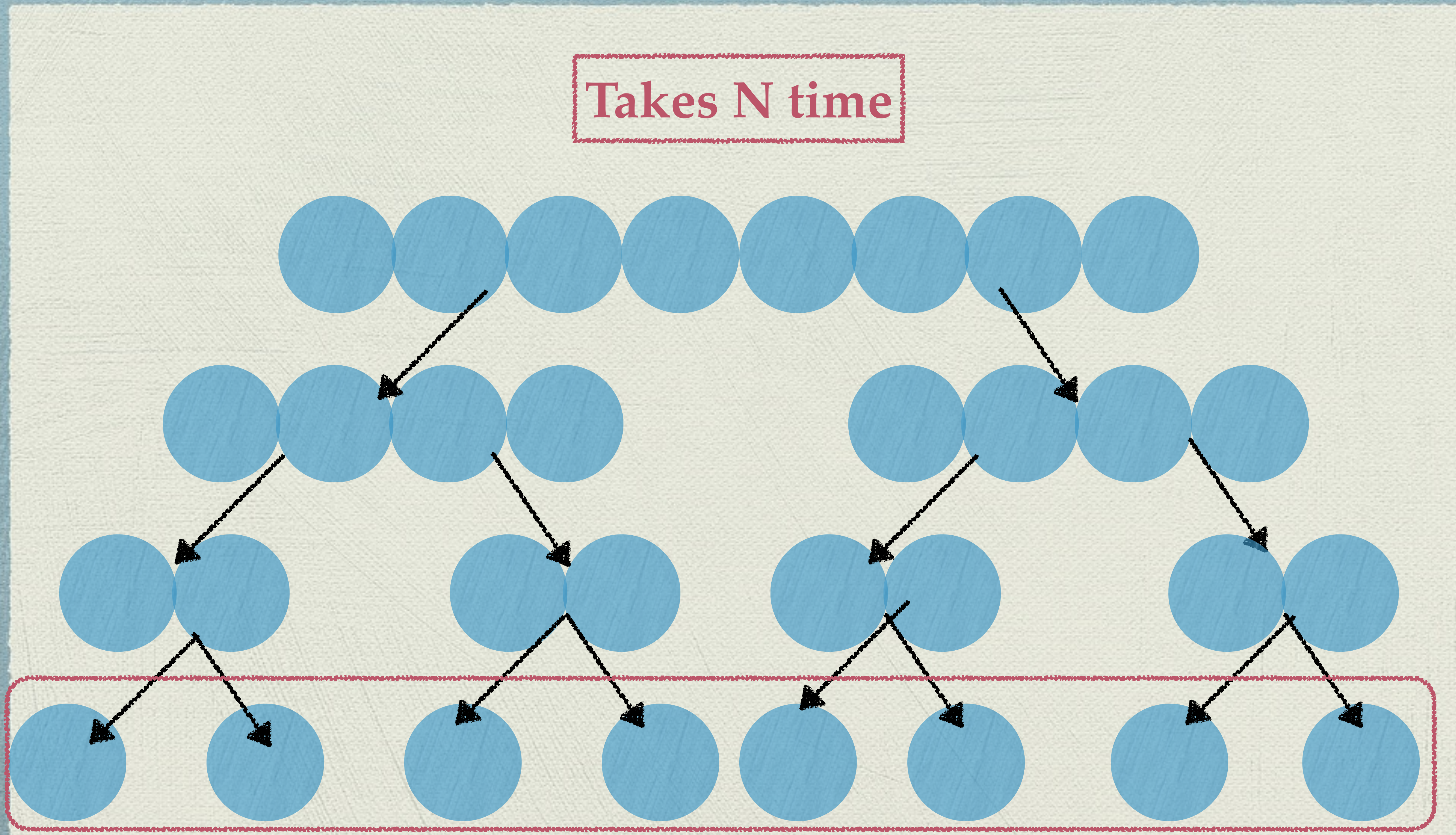
Merge sort runtime



Merge sort runtime

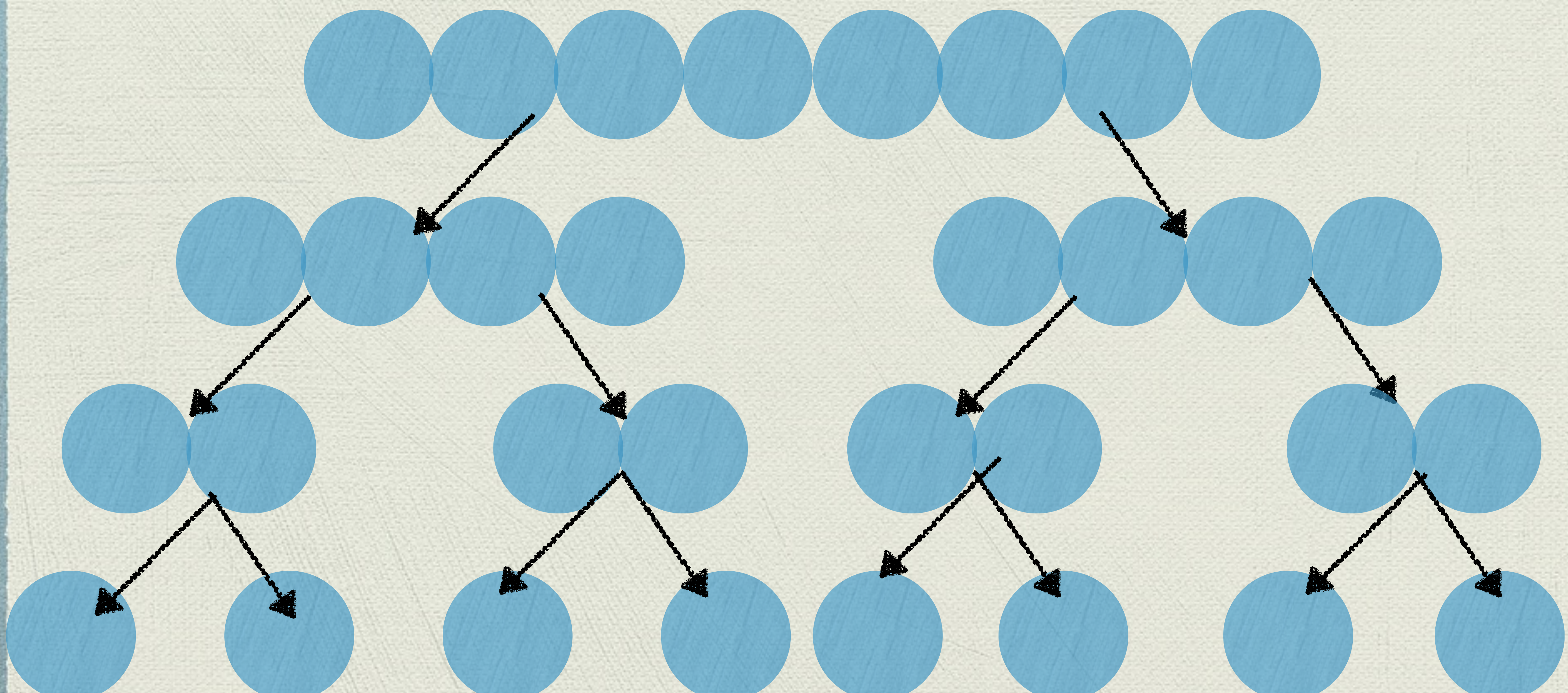


Merge sort runtime



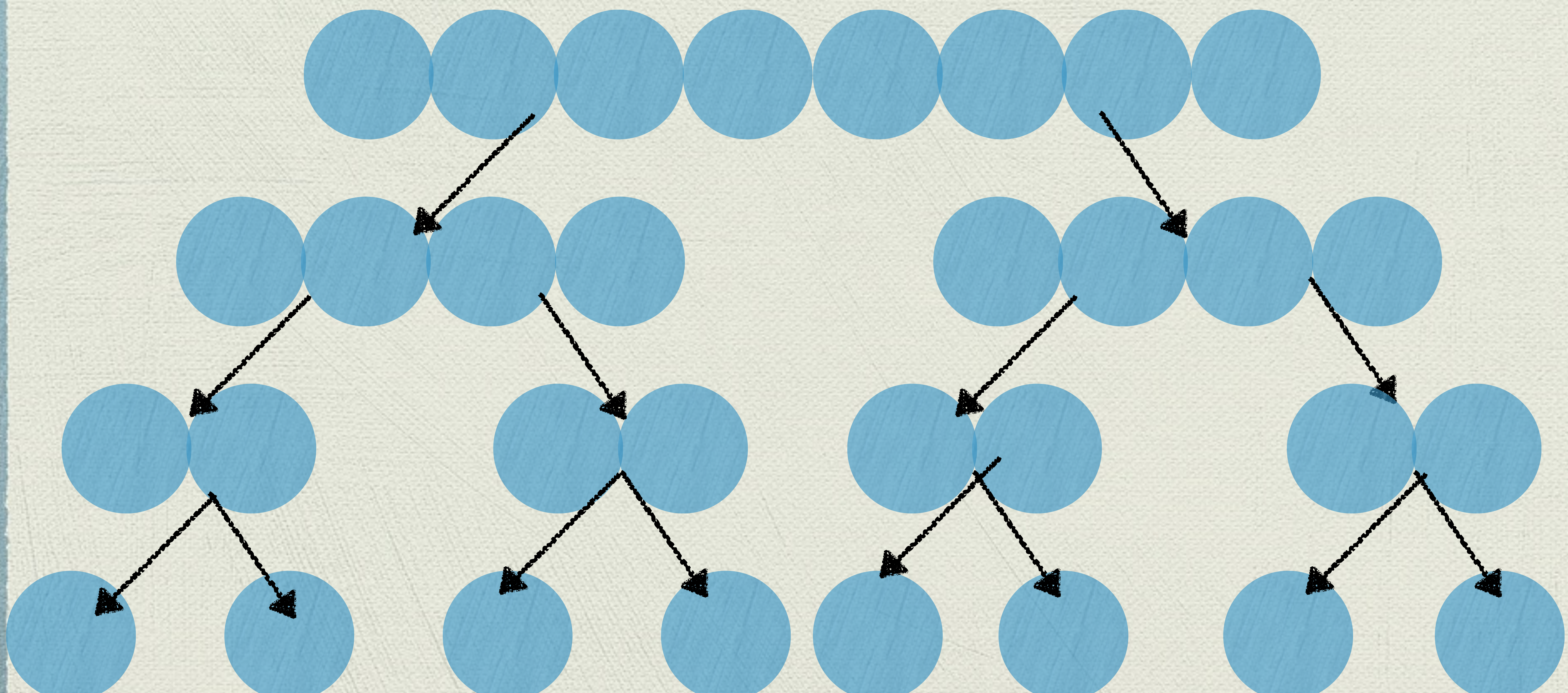
Merge sort runtime

- Each **set** of recursive calls at the same depth takes **N** time



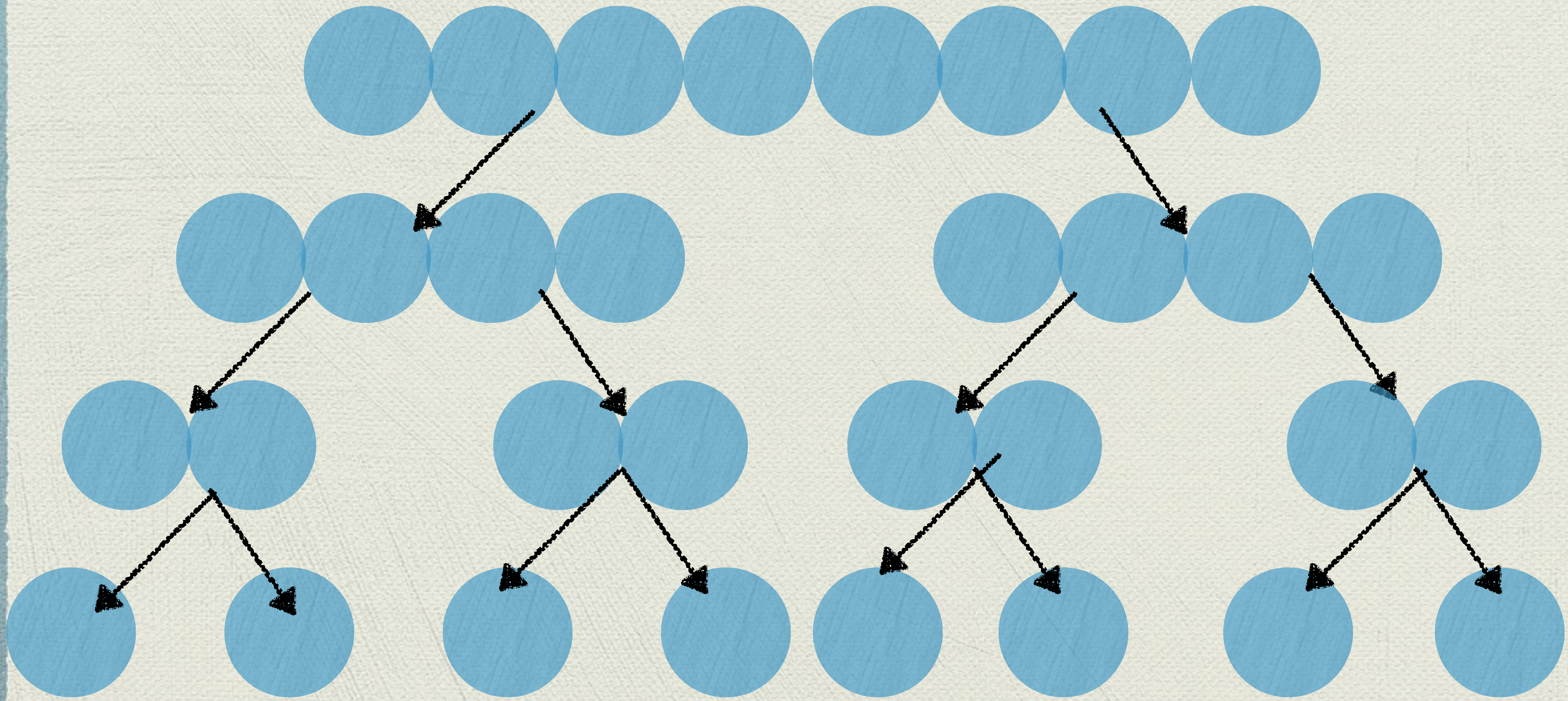
Merge sort runtime

- The total runtime must be N * the number of levels



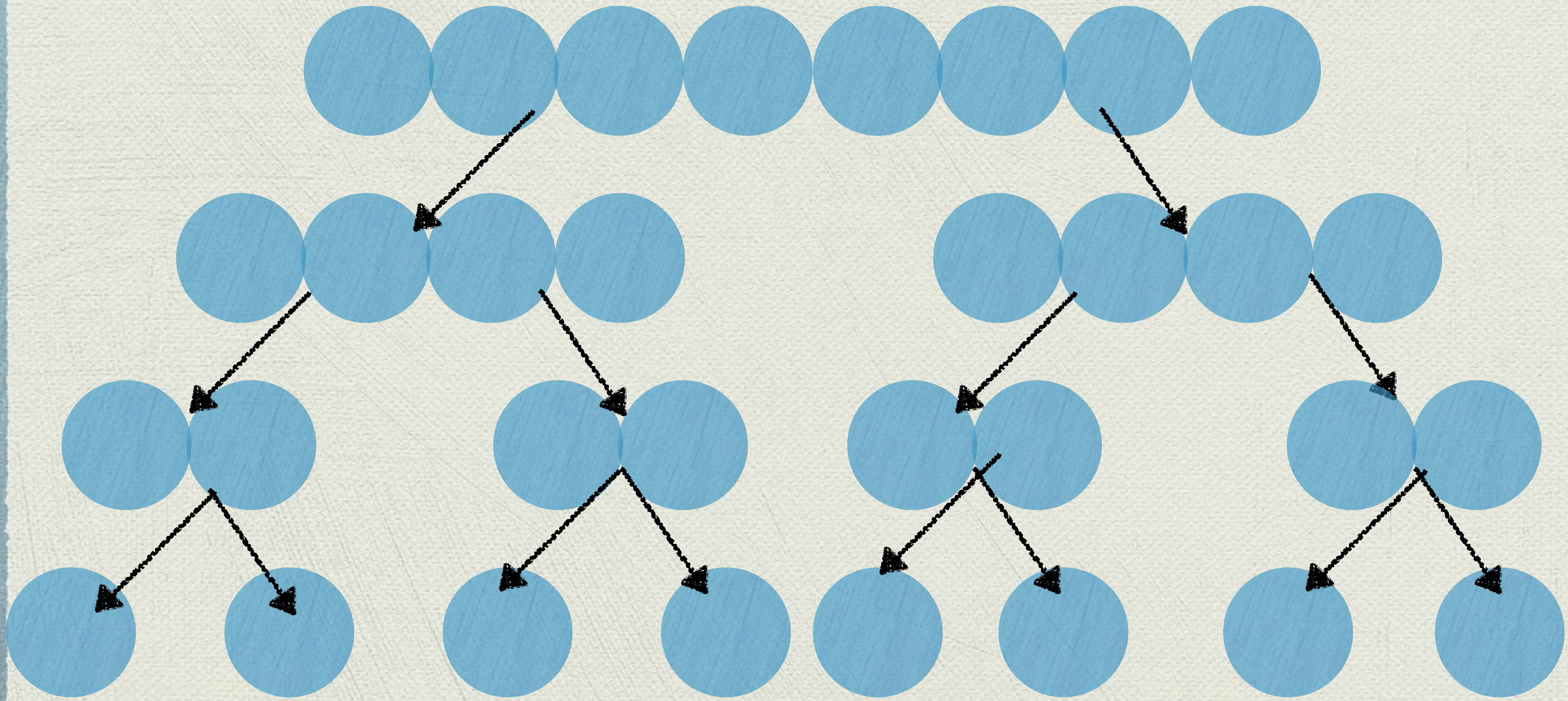
Merge sort runtime

- How many levels?



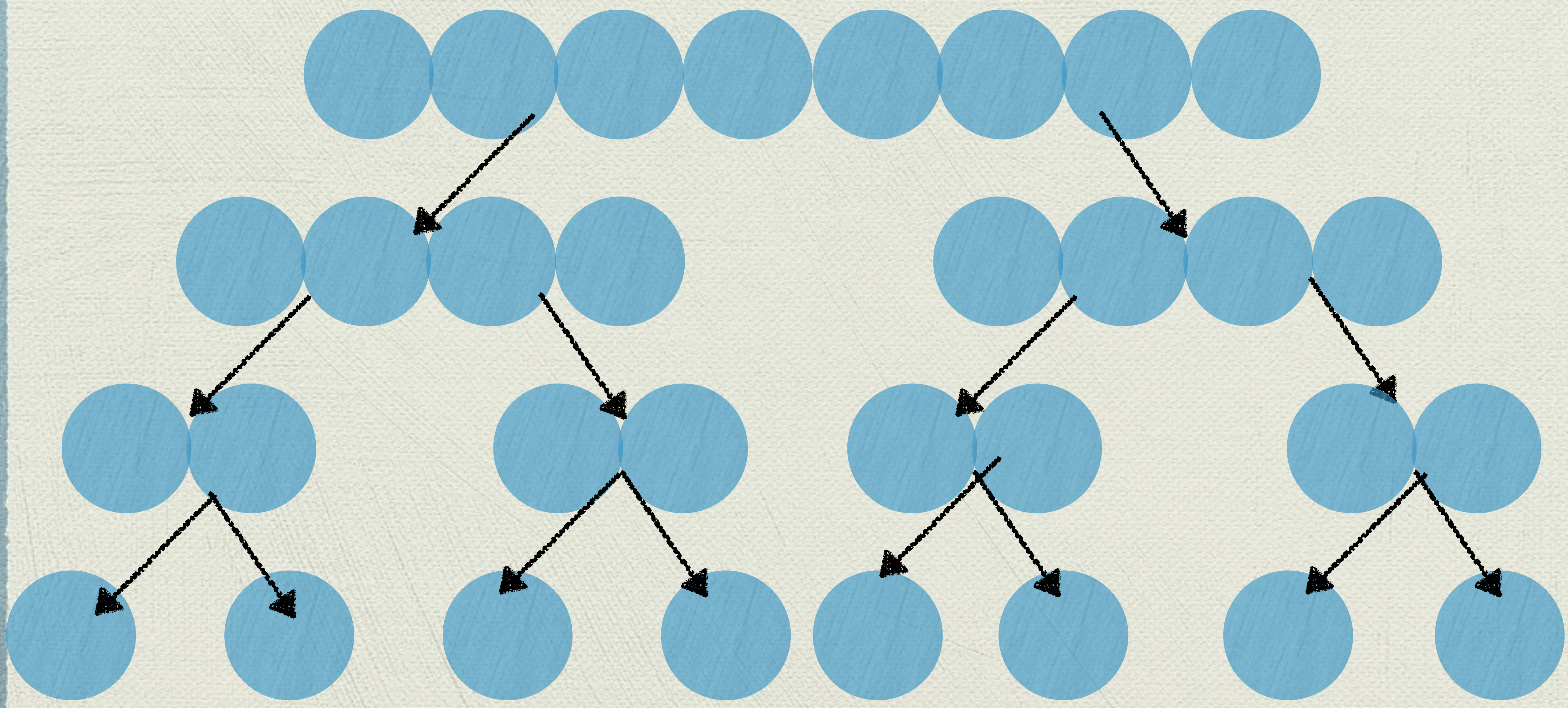
Merge sort runtime

- We keep dividing N by 2 until we hit 1…



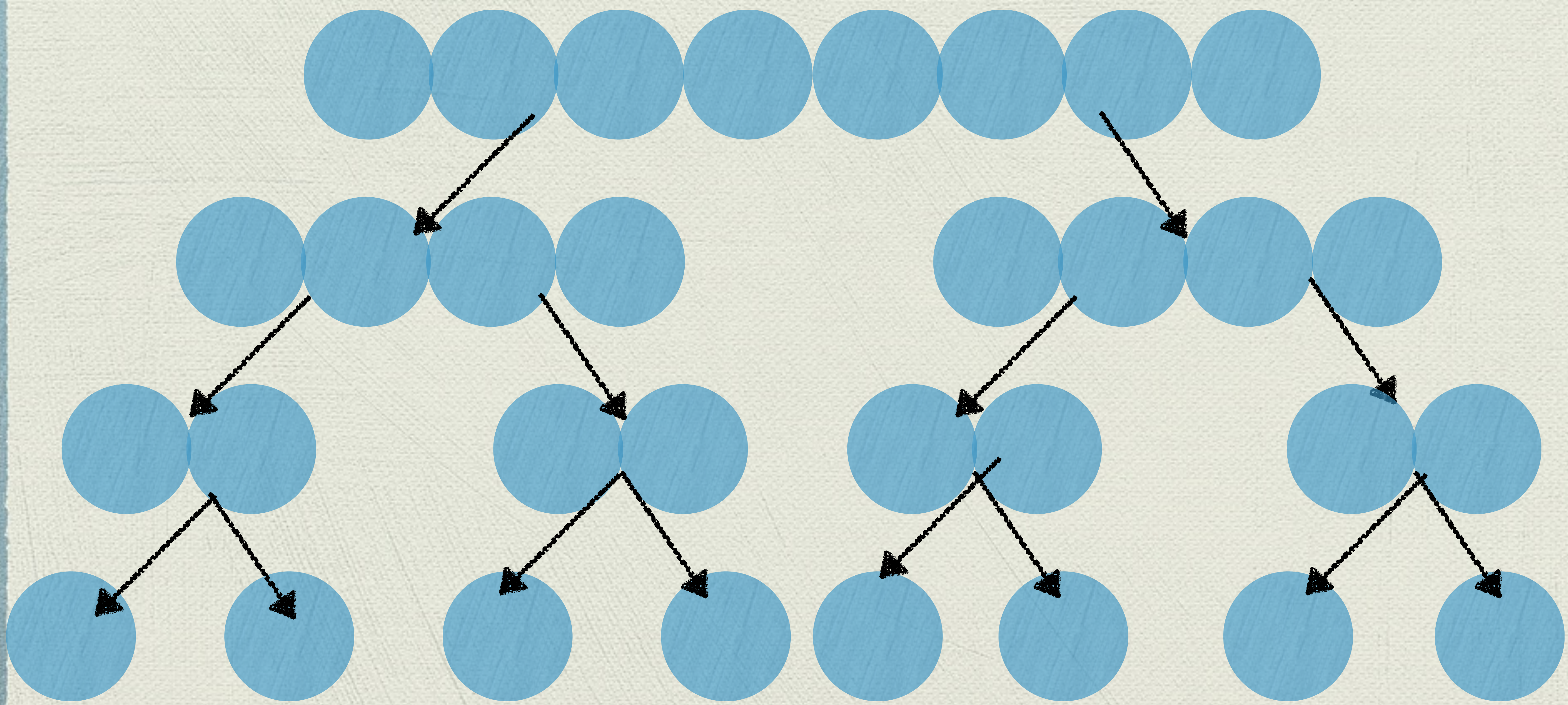
Merge sort runtime

- Oh, it's our old friend logN!



Merge sort runtime

- Runtime is O(N*logN)



Quicksort

- Merge sort is nice and all, but it's not the only cool kid on the block

Quicksort

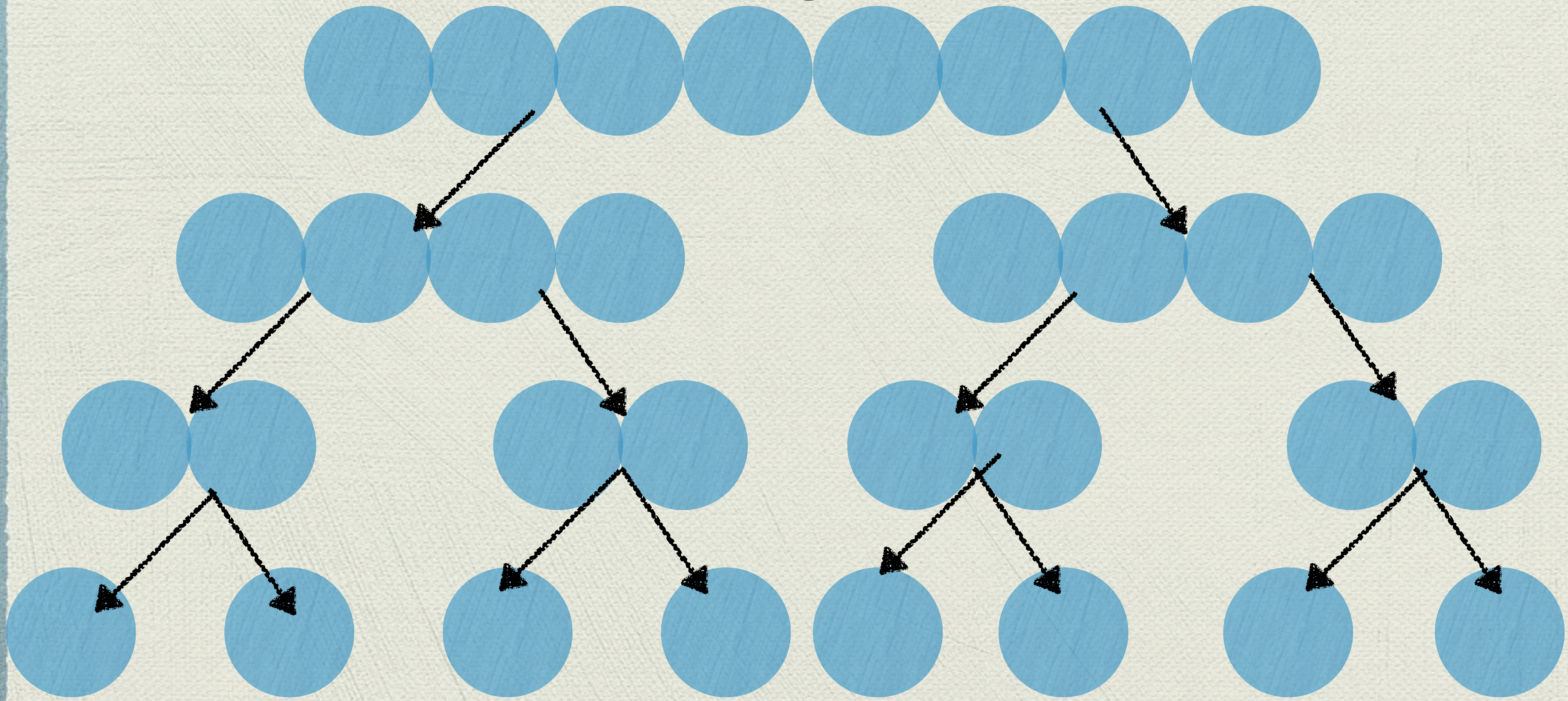
- The algorithm
  - Choose one item from the list (randomly?), call it the **pivot**
  - Divide your list in **two halves**: items smaller than the pivot, and items larger than it
  - Recursively quicksort each half
  - Concatenate (not merge) the two halves together

Quicksort runtime

- We can use the exact same argument we used with merge sort to show quicksort's runtime is also in O(NlogN)…

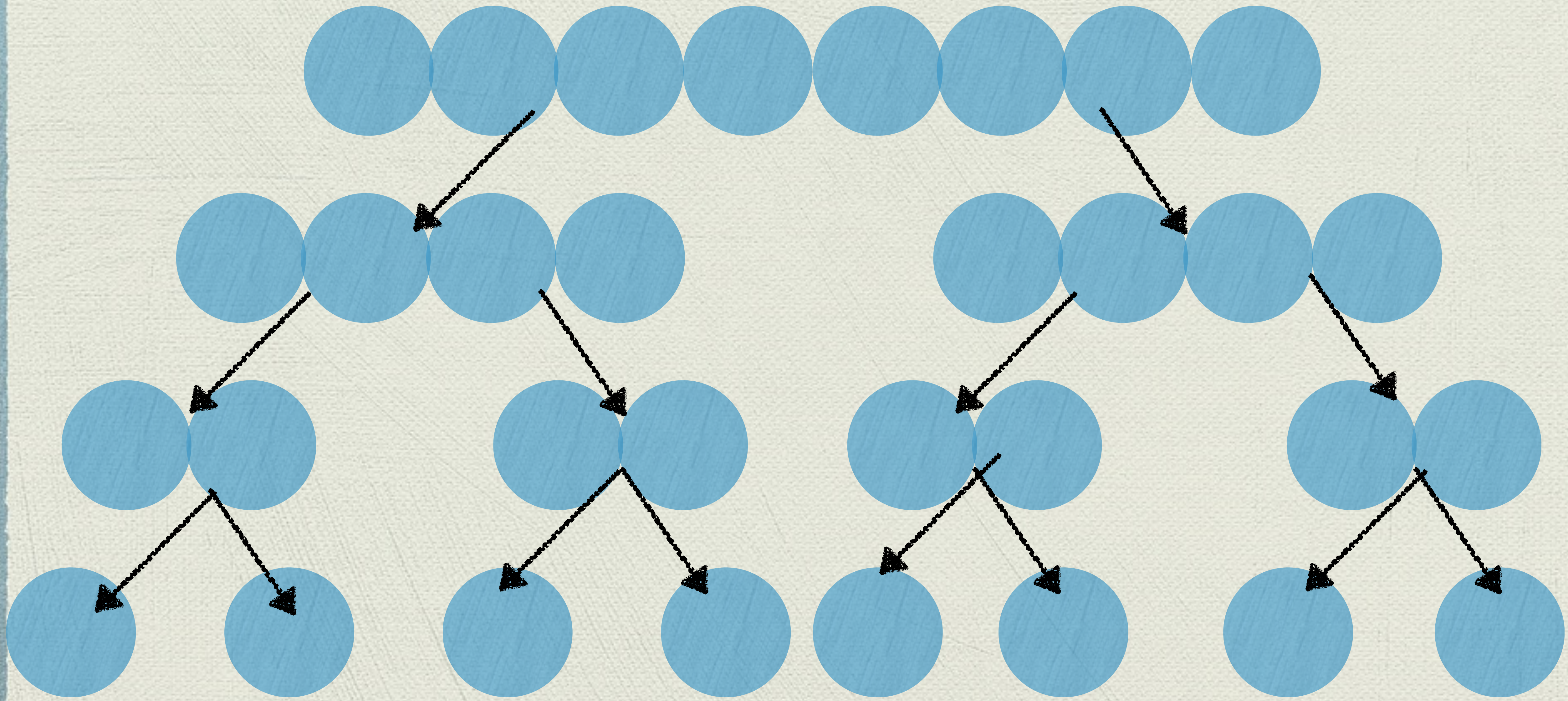
Quicksort runtime

- At each step, we put the smaller half of items in one recursive call, and the larger half of items in the other



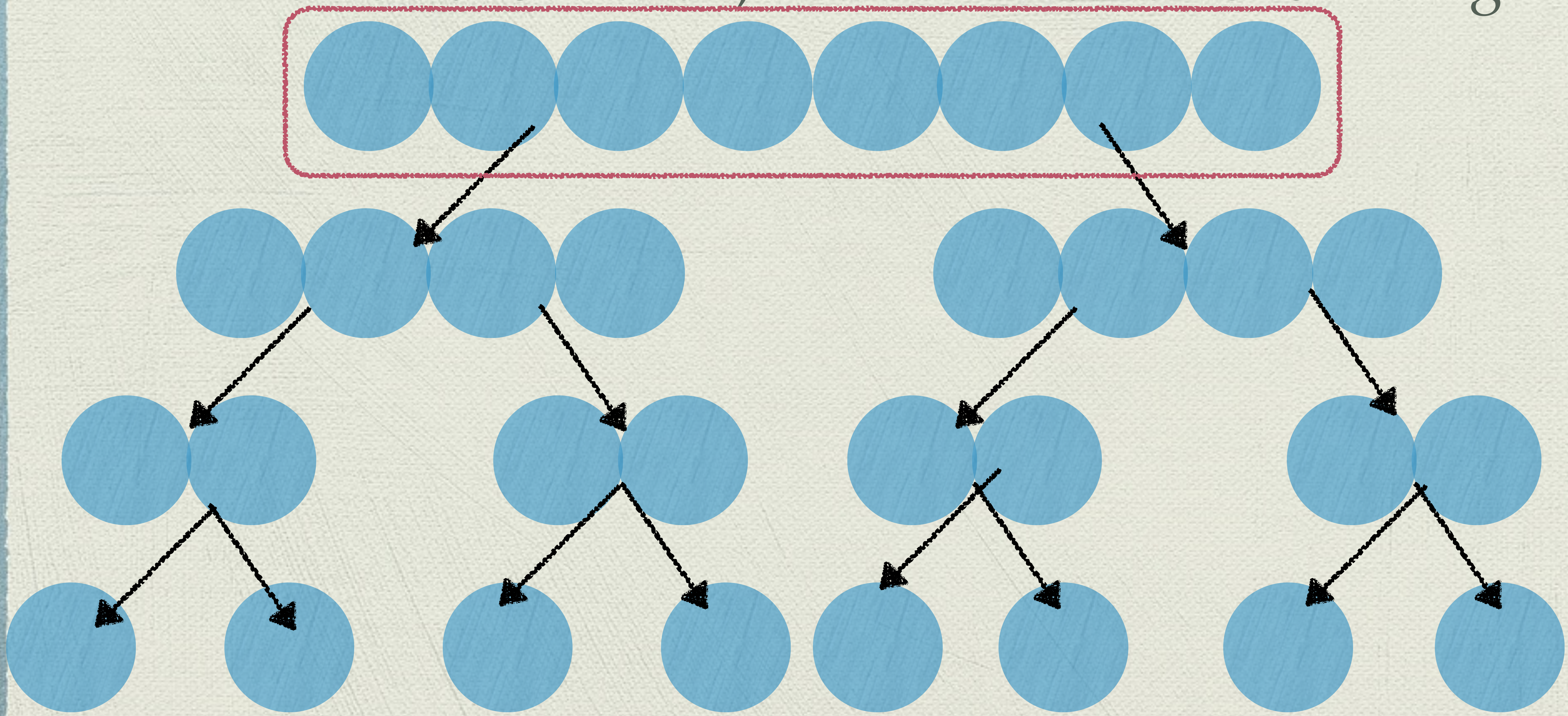
Quicksort runtime

- So we keep dividing by two until there is just one item



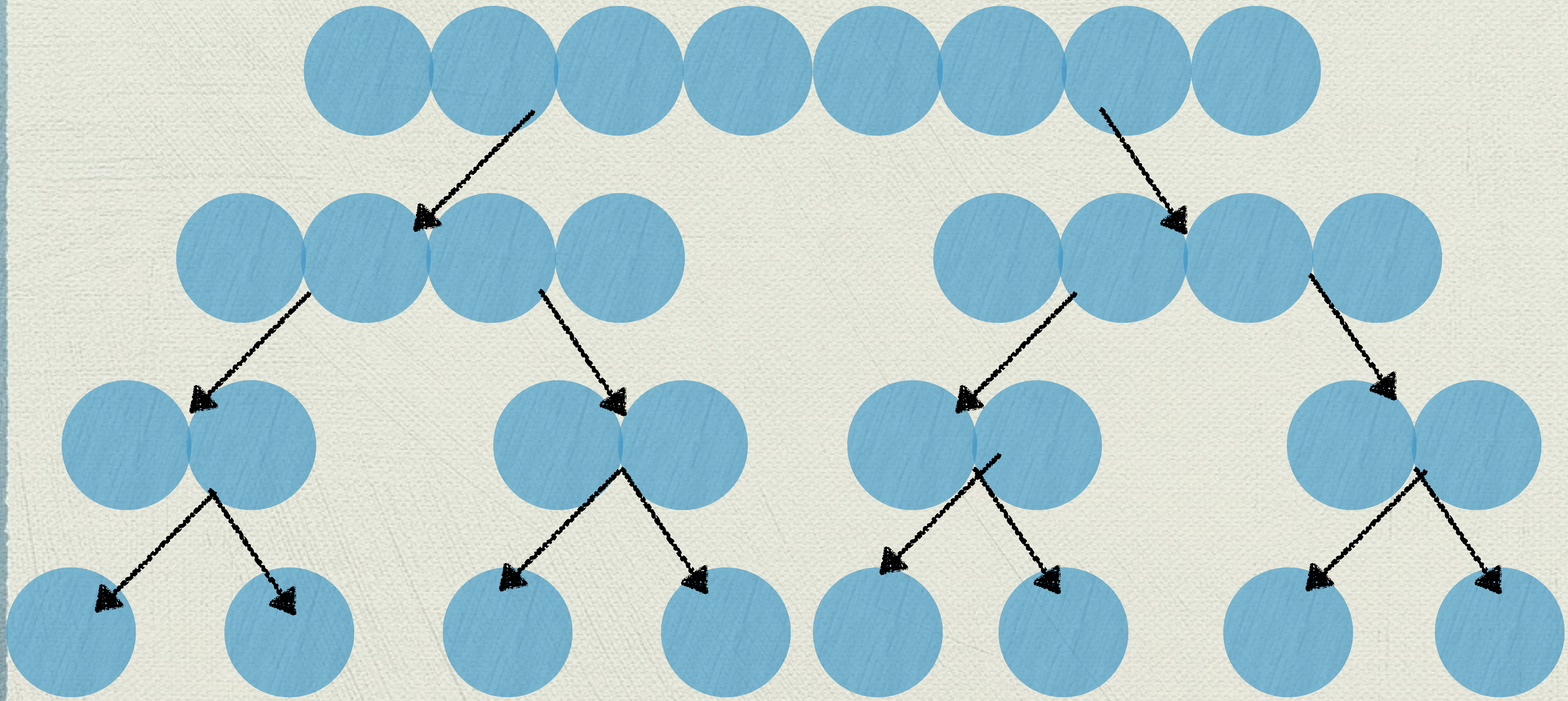
Quicksort runtime

- A function call with **N** nodes takes **N** time to move half the items to the left, and half the items to the right



Quicksort runtime

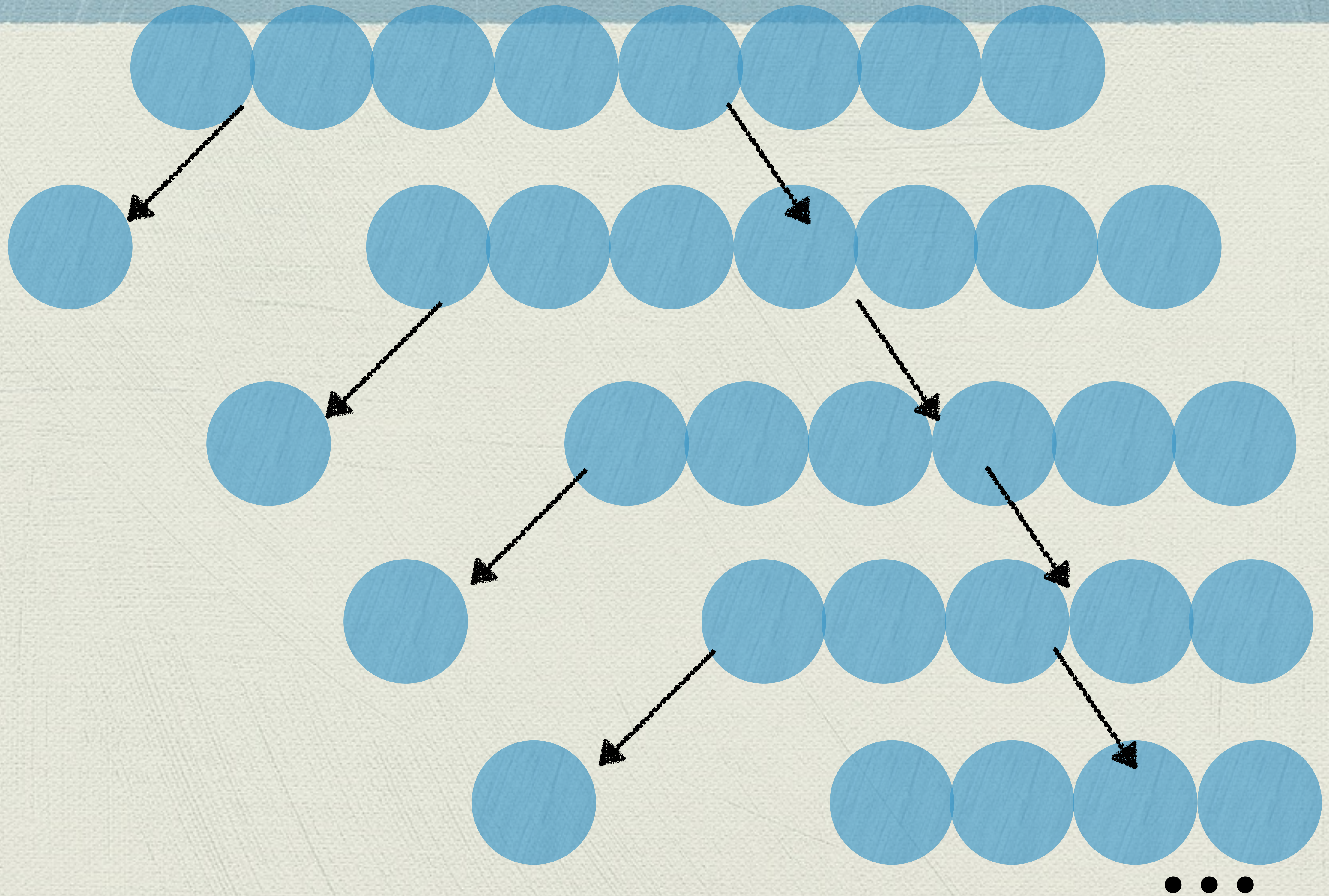
- So it's actually the exact same argument as merge sort



Quicksort runtime *problem*!

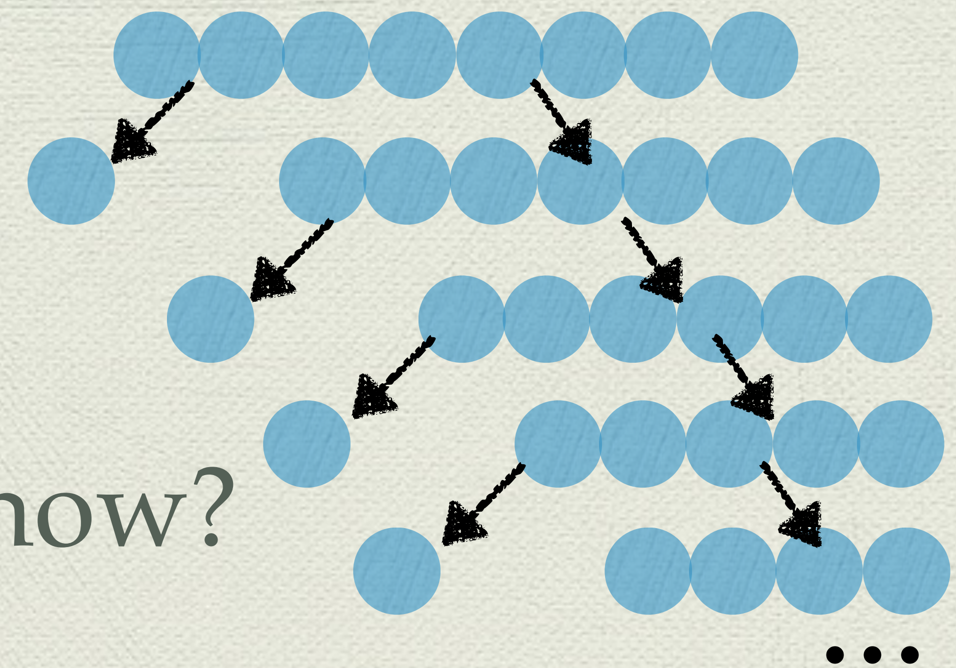
- In the previous argument, we assumed that half the items would end up on one side of the pivot, and half would end up on the other
- This relies on the assumption that the pivot is the **median** item
- What if it's not? What if we chose the smallest item as the pivot, for example?

Quicksort with smallest item pivot



Quicksort with smallest item pivot

- Remember, the runtime is
- O(N * number of levels)
- How many levels are here now?
- If we only split off one element each time, it will take us N levels to get to the bottom
- So the runtime is $O(N^2)$



Quicksort runtime problem!

- So if the pivot is the smallest item, runtime is $O(N^2)$ **(slow!!)**
- If the pivot is the median item, runtime is $O(N\log N)$ **(fast!!)**
- So, should we always make the median item the pivot?

Finding the median item

- Algorithm:
  - First **sort** the list, and then choose the item at the middle index
- Uh oh.

Finding the median item

- Actually, there's a better algorithm that you (should) learn in CS 170
- Even so, finding the median element takes enough time that it slows down quicksort significantly

Choosing the pivot

- **Another idea:**
  - The pivot isn't the median element, but is just a **random** item from the list
  - On average, this will *roughly* divide the list in half
  - The tradeoff is worth it, because it's a lot faster to pick randomly than to calculate the median

Choosing the pivot

- **An even better idea:** Randomly select three items, and then choose the median of them
  - Trying to balance tradeoffs between choosing an exact median, and choosing randomly

The results

- Bubble sort O(N) best, O(N^2) worst
- Selection sort O(N^2)
  - Heapsort O(NlogN)
- Insertion sort O(N) best, O(N^2) worst
- Merge sort O(NlogN)

Basically never happens

- Quicksort O(N^2) worst, O(NlogN) best

How much of a difference does it make, anyway?

- http://www.youtube.com/watch?v=SJwEwA5gOkM&t=24m15s

All right, Mr. President, we're convinced!

- The bubble sort is clearly **not** the way to go
- Quicksort appears to be the fastest (hence its name)
- Is this the end of the story?

Asymptotic runtime isn't everything

- How would you choose between quicksort, merge sort, and heapsort, anyway? Is insertion sort ever useful?
- Quicksort tends to be fastest in practice
- Okay, but… there are additional factors to consider.

Stability

- A sort is **stable** if…
- …items with the same value end up in the same relative positions before and after the sort
- What?

Stability

- Here's an list with two **4**s in it. I've colored one blue, and the other pink.

4, 5, 3, 2, **4**, 1, 9, 0

- There are two valid ways to sort this list of numbers

0, 1, 2, 3, **4**, **4**, 5, 9   0, 1, 2, 3, **4**, **4**, 5, 9

- If the algorithm is *guaranteed* to give us the left one, then the algorithm is **stable**

Stability

- *Why would this even matter*

Sorting with multiple keys

- Imagine you have an array of **Product** objects you're selling online:

```
public class Product {
  String myName;
  double myPrice;
  double myRating;
}
```

- You want to sort the products by price. But among products with the same price, you want to sort them by rating. How could you do this?

Sorting with multiple keys

- Algorithm:
  - First, sort the products by rating
  - Then, **stably** sort the products by price
- On the second sort, you’re guaranteed that products with the same price will end up in the order they started in (which was sorted by rating)

Okay, so I guess stability might be useful

- So what?
  - The fastest way to implement quicksort on arrays isn't stable
  - Heapsort isn't stable either
  - But merge sort is
- **Conclusion**: If you don't need stability, quicksort may be fastest. If you do, consider merge sort

Asymptotic runtime isn't everything

- When choosing a sorting algorithm, it's important to consider whether stability is important to you

Asymptotic runtime isn’t everything

- Are there other factors to consider, too…?
- Consider your situation carefully

Other factors — receiving one new item

- Say you currently have a list of books, sorted by title
- Then someone hands you a new book to add to the list. What should you do?
    - **Option 1**: Iterate through the list until you find the correct spot for the book, and put it there
    - **Option 2**: Stick the book at the end, and then re-sort the whole list

Other factors — receiving one new item

- No need to re-sort the whole thing, so option 1 is clearly best
- This is basically **insertion sort**
- Conclusion: If you receive items one-by-one occasionally, rather than all at once, you essentially have no choice except to insertion sort

Other factors — consider the nature of your data

- Say you need to sort a list of million 32-bit integers, but you happened to know all of the integers were either 2015, 2014, or 2013
- Can we take advantage of this fact to speed up the sorting?

Counting sort

- I propose a simple algorithm called **counting sort**
- It will sound kinda dumb, but sometimes the simplest solution is best

Counting sort

- The algorithm:
  - Tally up each type of item
  - Then create a new list with however many copies of each item

Counting sort walkthrough

- Say we want to sort this list of numbers:

  2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

- We'll maintain a tally:

2013	**2014**	**2015**

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

		I
2013	2014	2015

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

2013 2014 2015

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

2013 **2014** **2015**

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

2013	2014	2015
I	I	II

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

2013 **2014** **2015**

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

Counting sort walkthrough

- Iterate through the numbers one-by-one, and tally

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

Counting sort walkthrough

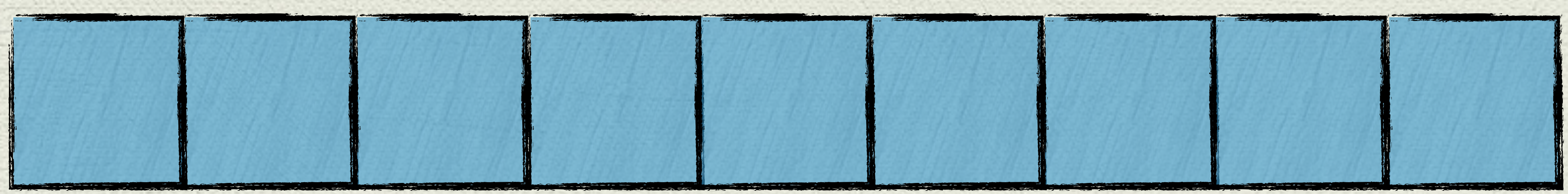
- What can we tell from this information?
- We know the sorted list will look like two 2013s, followed by four 2014s, followed by three 2015s

	2	4	5
counts	2013	2014	2015

Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

- First, create an empty array big enough to hold all of the numbers:

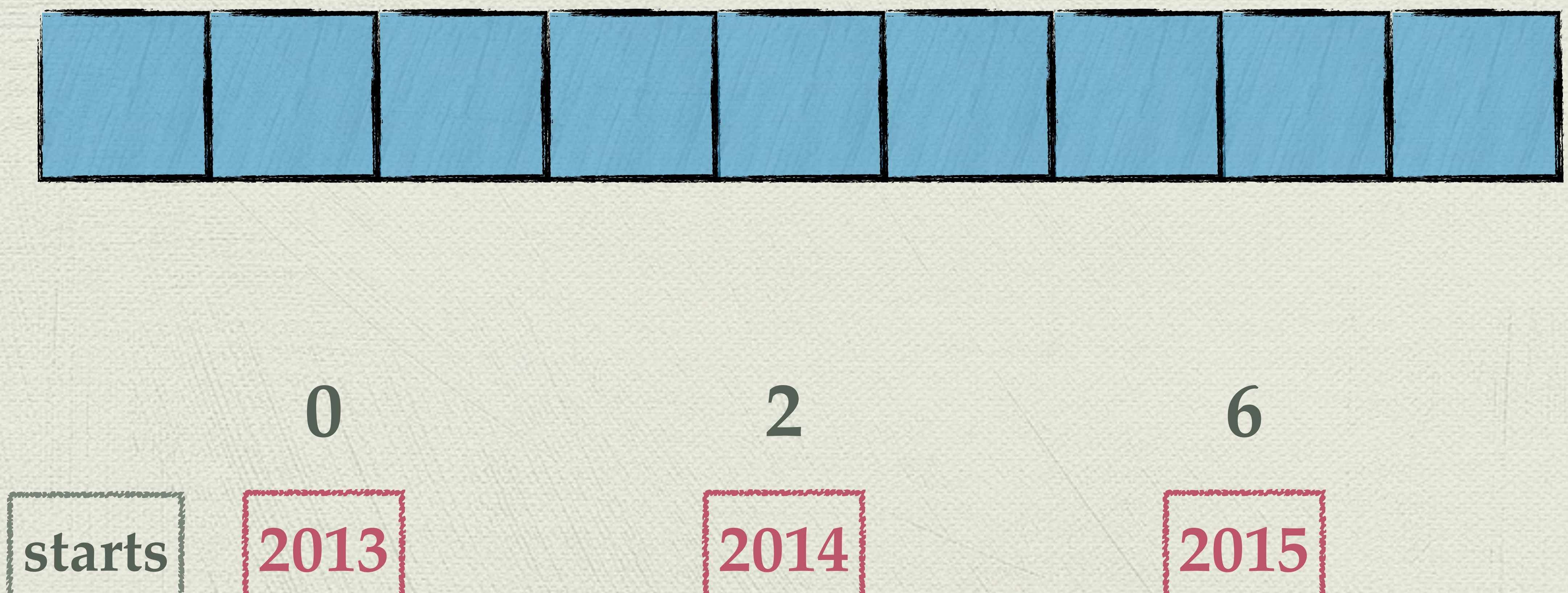


	2	4	5
counts	**2013**	**2014**	**2015**

Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

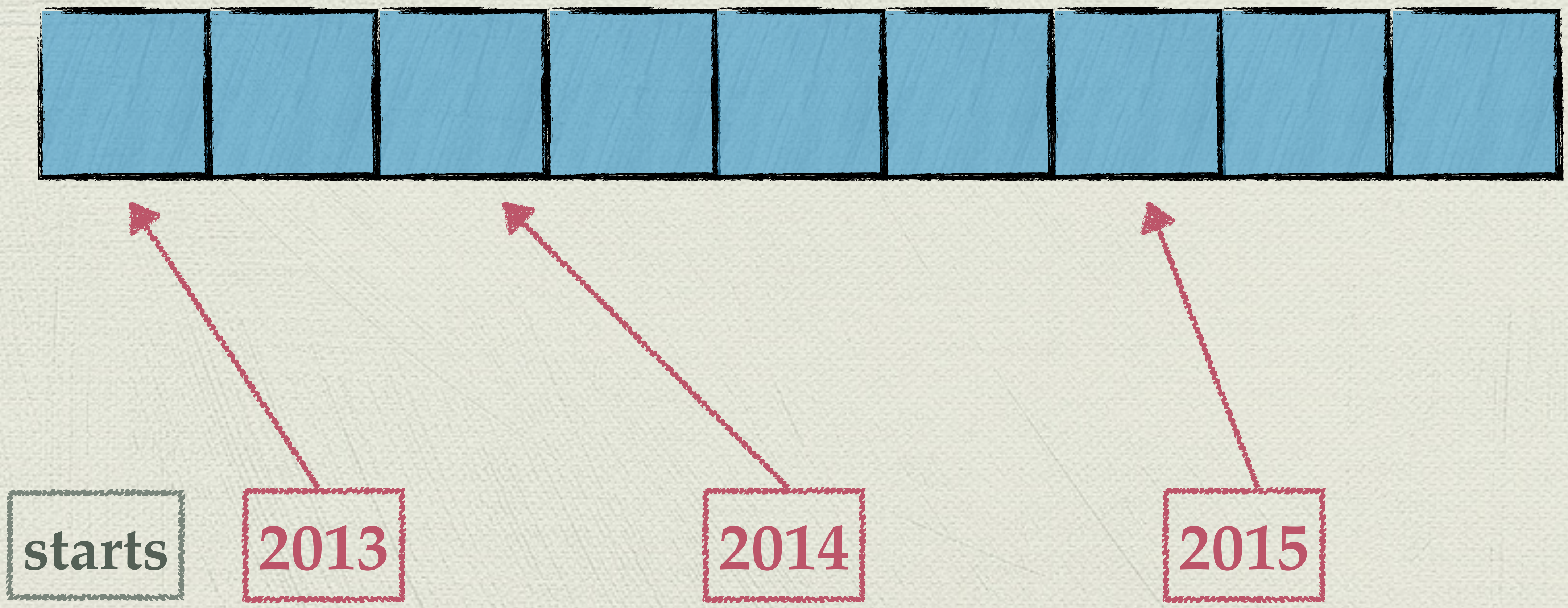
- From the counts, we can figure out what the **starting position** of each kind of year is



Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

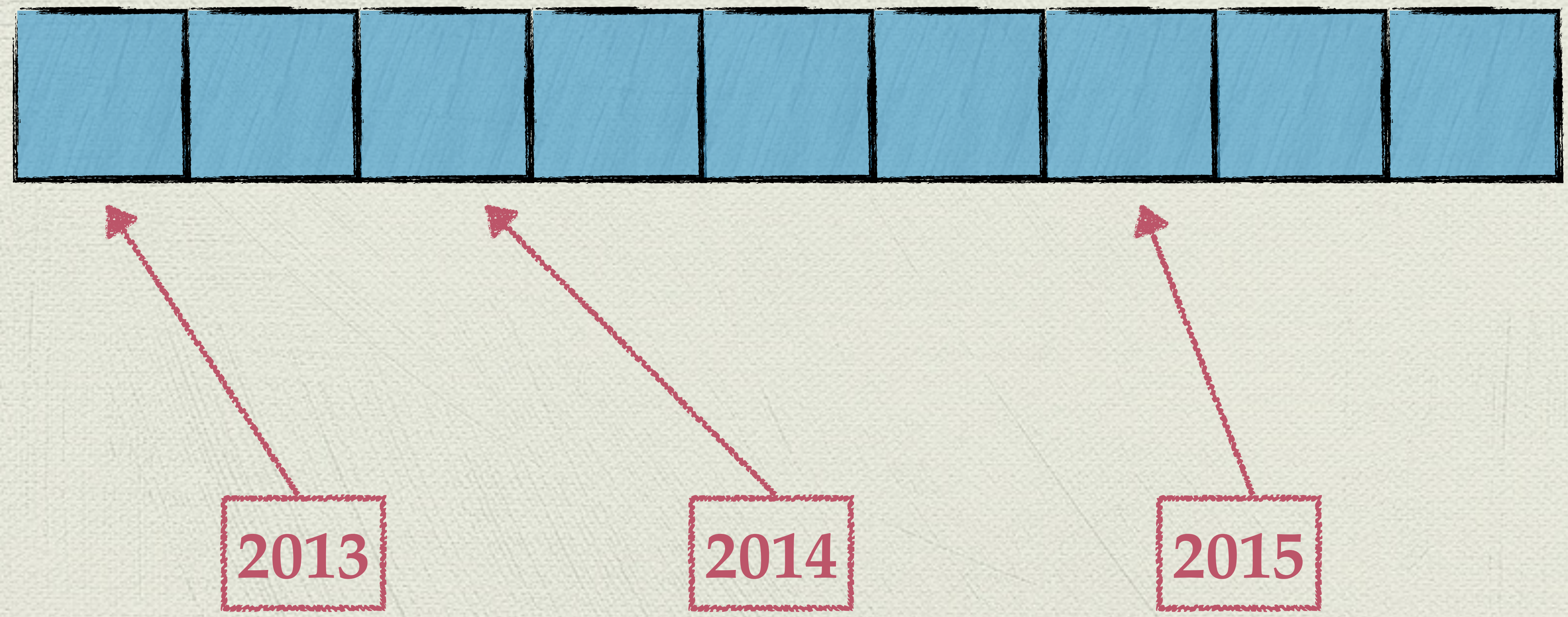
- From the counts, we can figure out what the **starting position** of each year is



Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

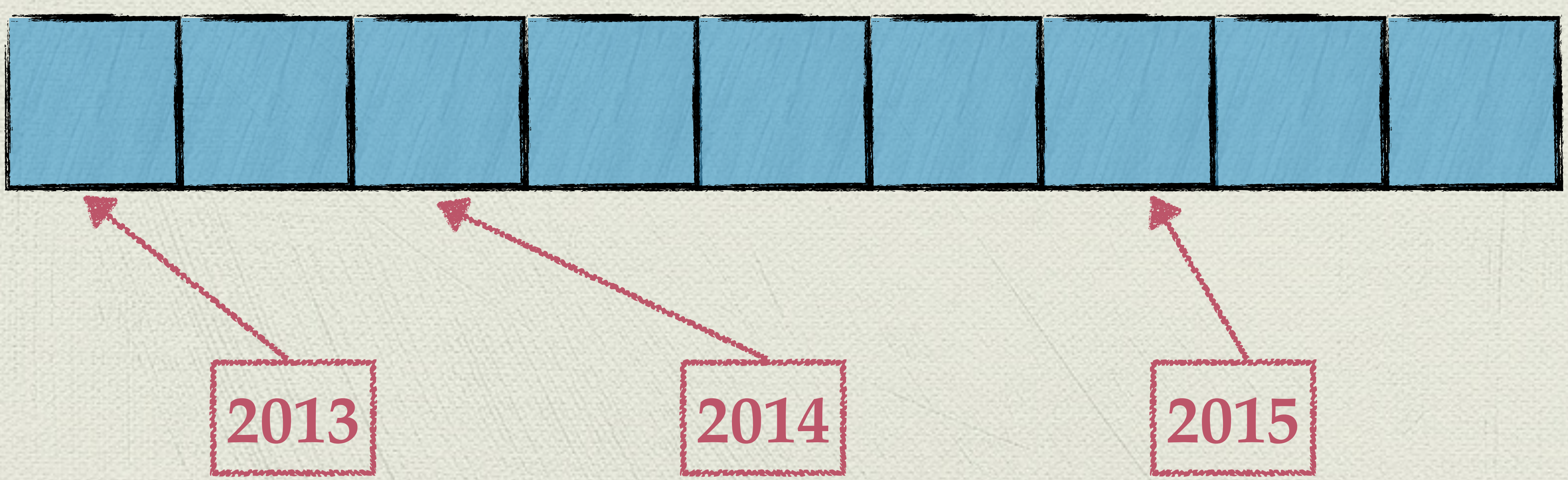
- Now we just iterate through our original list, and put items in the correct spots



Counting sort walkthrough

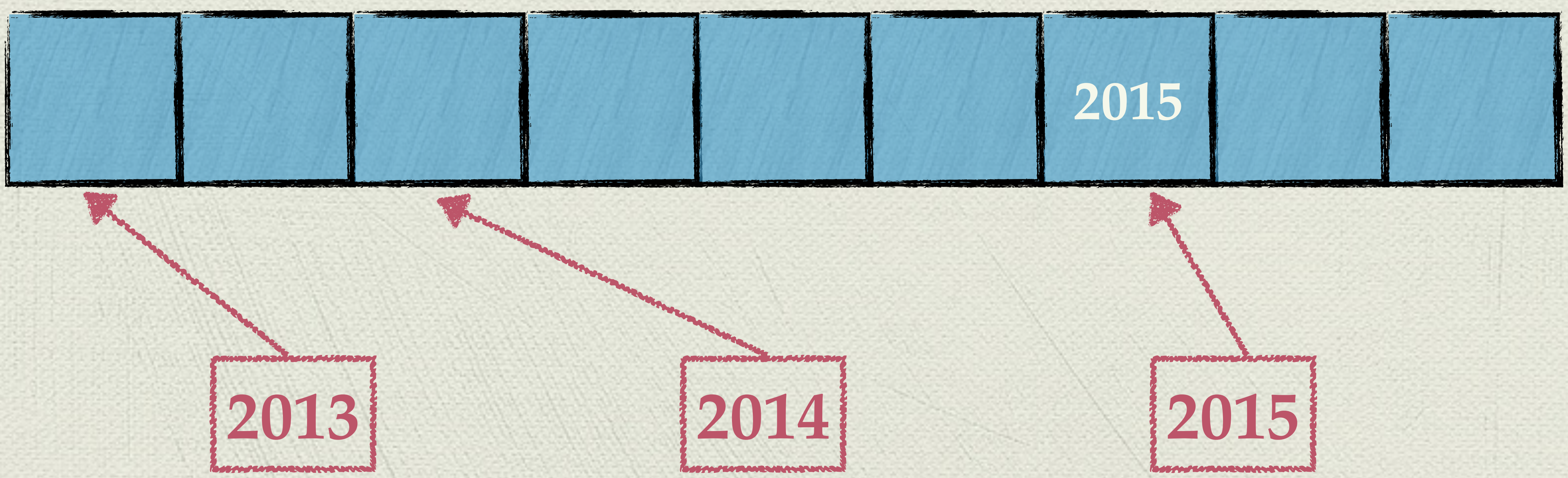
2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

It's the first 2015, so we know where it must go in the array



Counting sort walkthrough

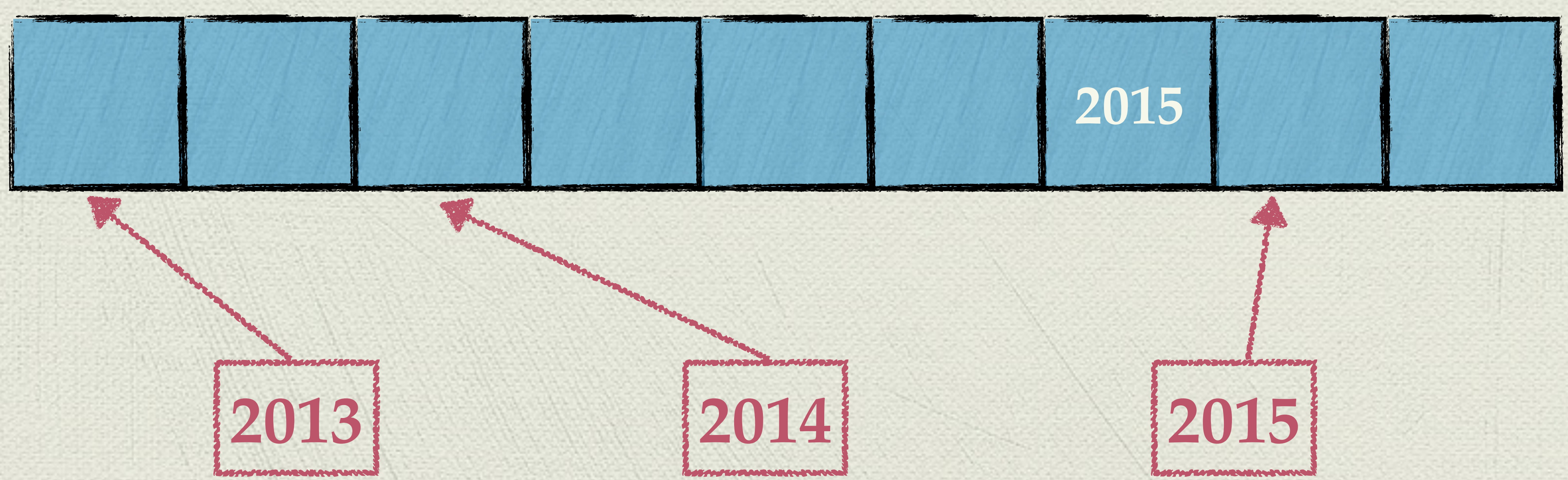
2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



Counting sort walkthrough

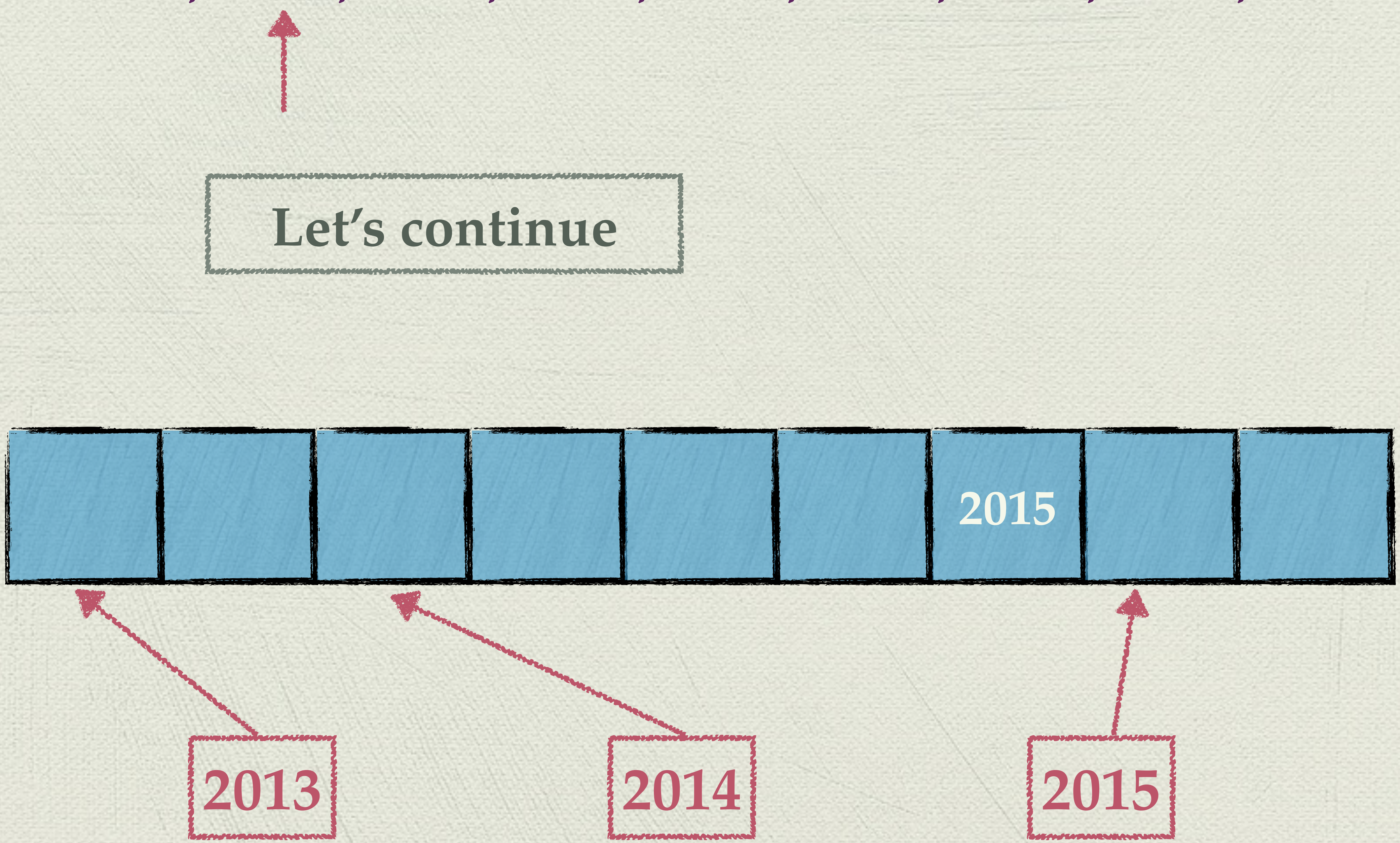
2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

Now the starting positions of 2015s has moved



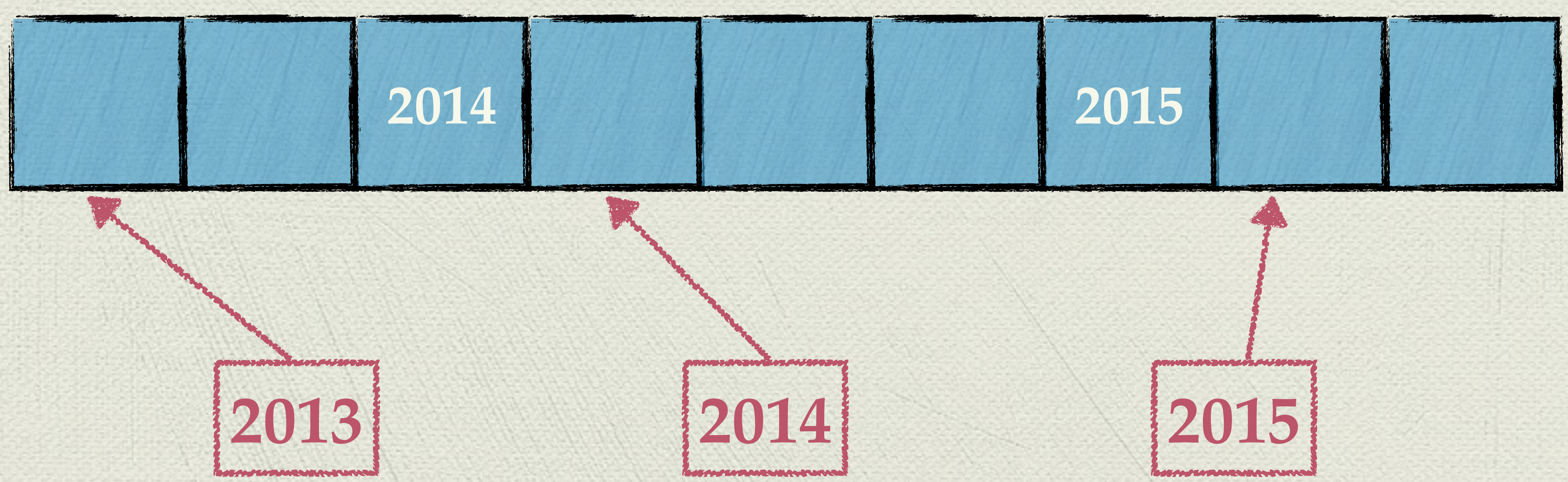
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



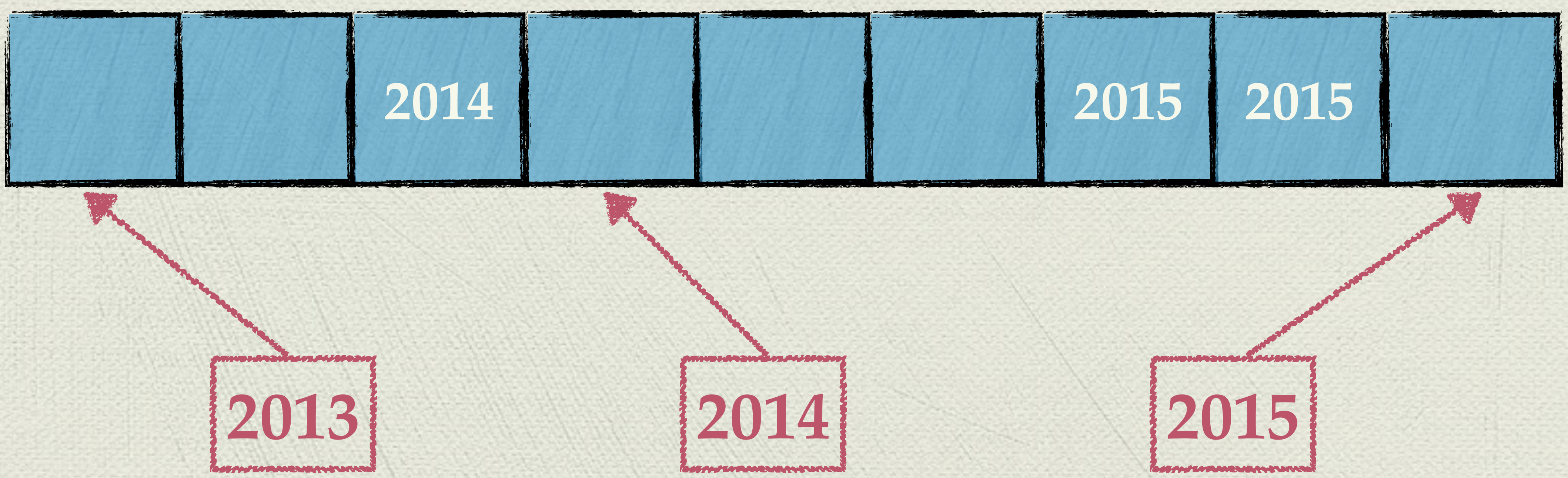
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



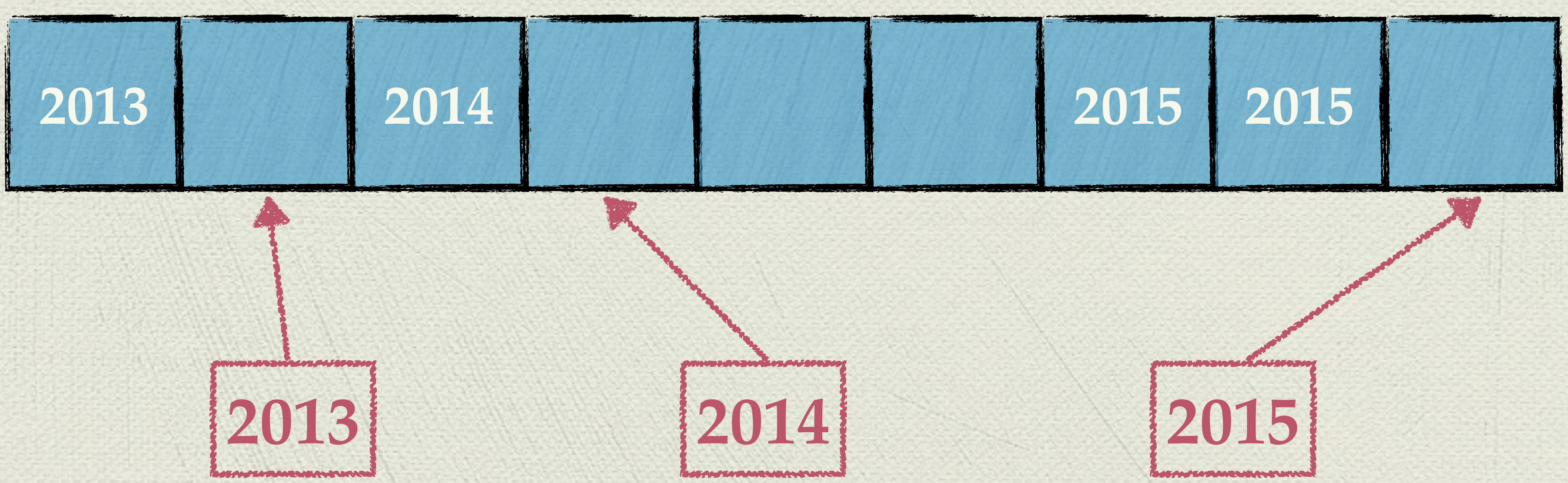
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



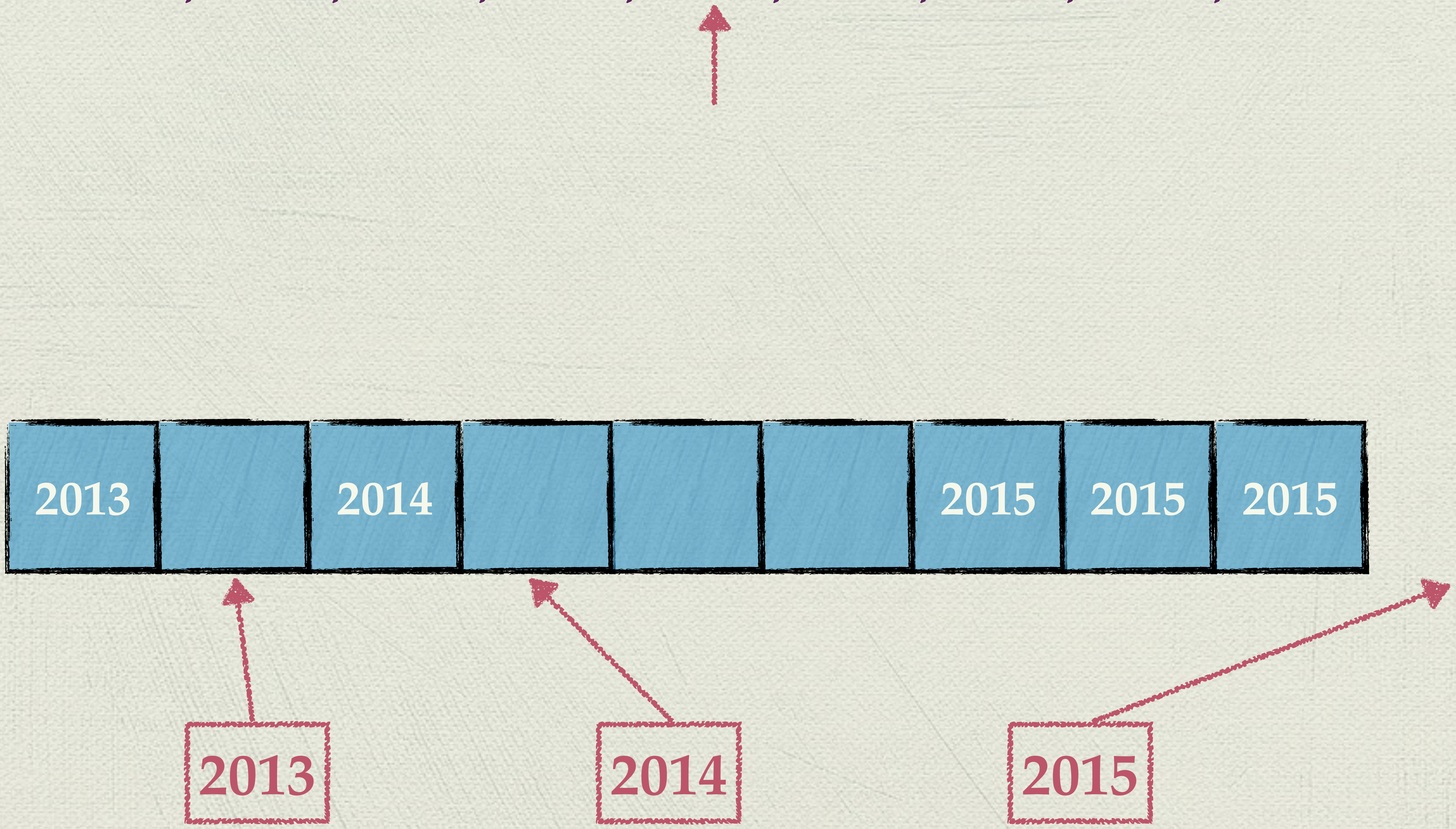
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



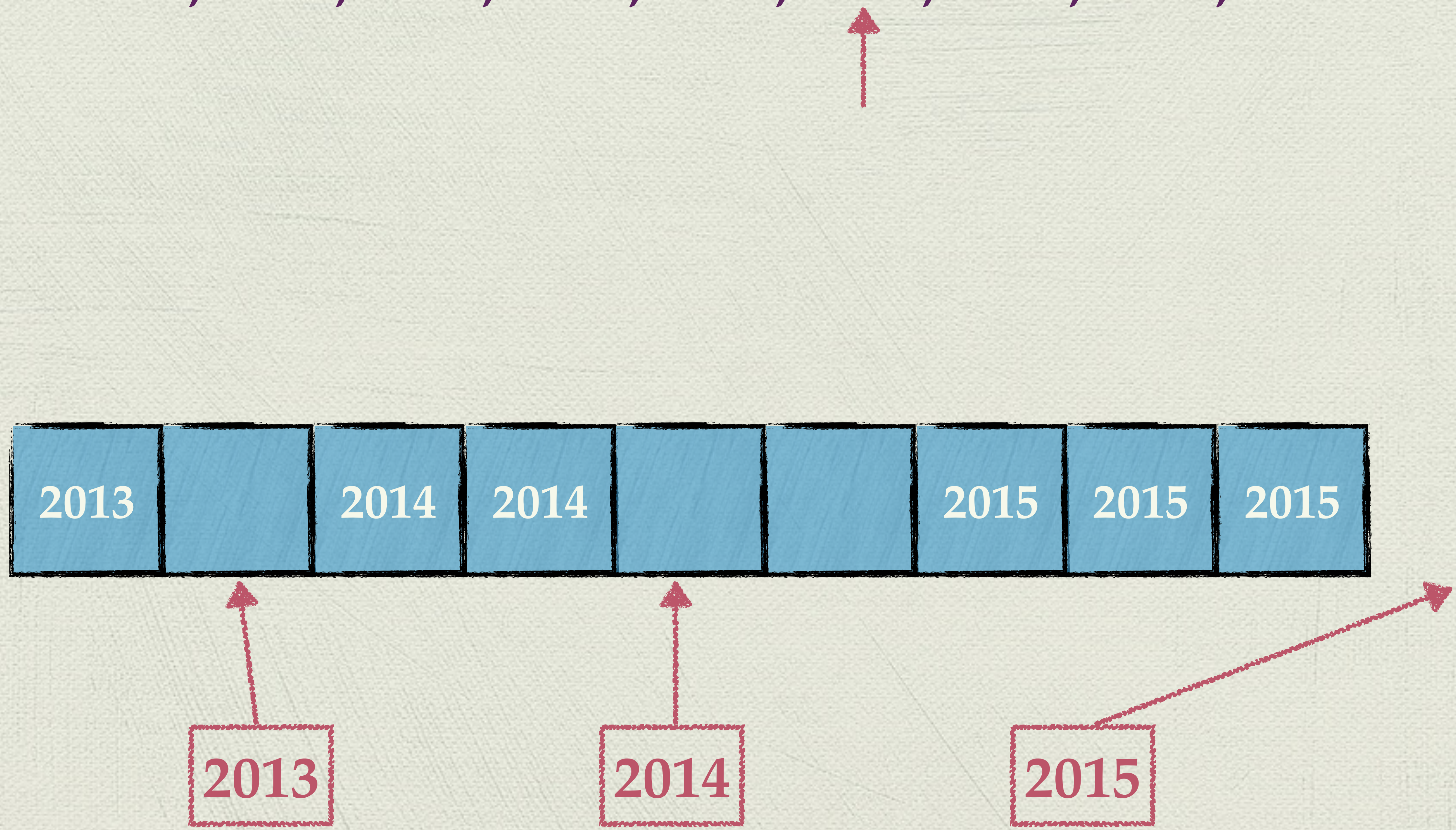
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



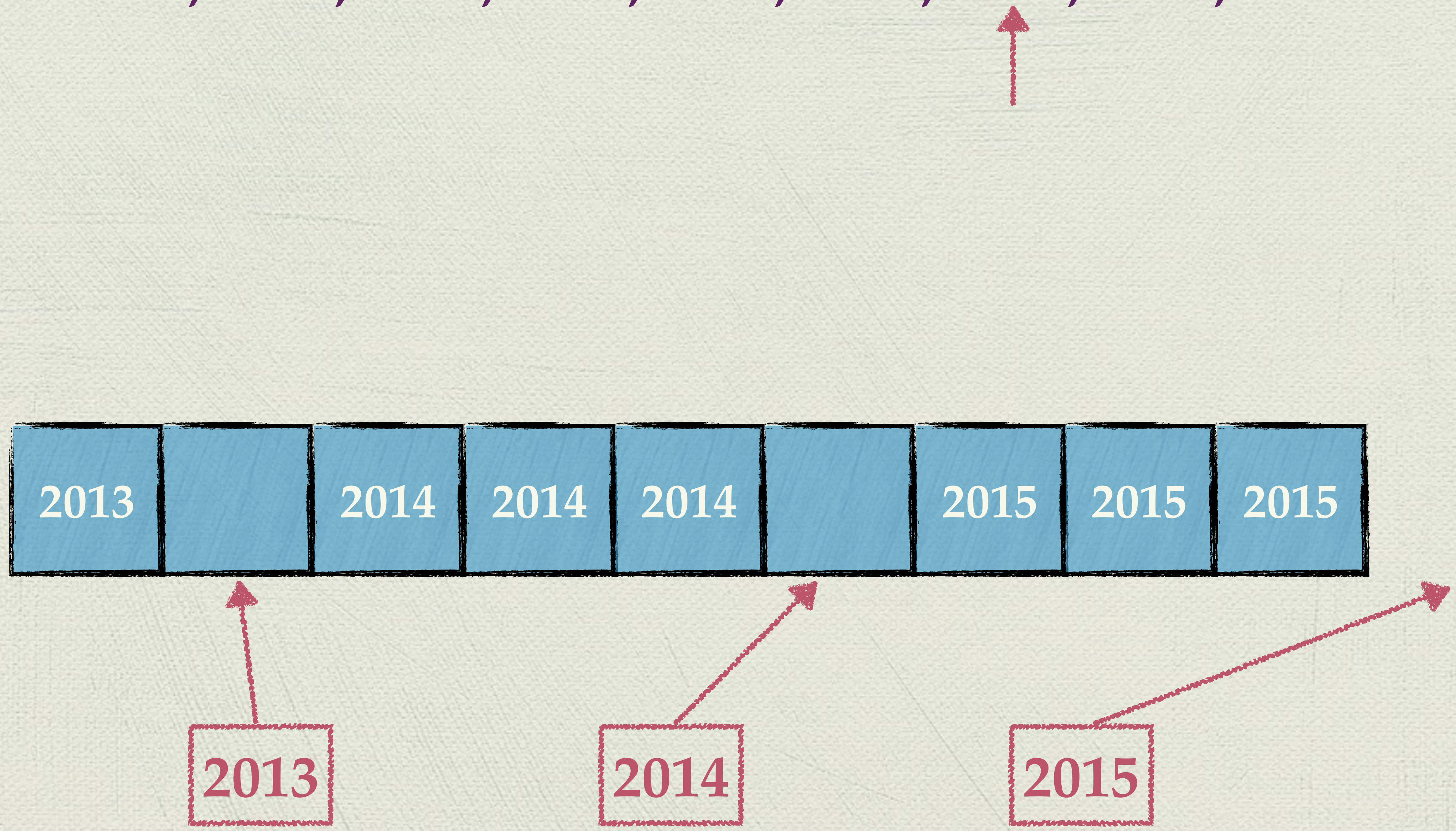
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



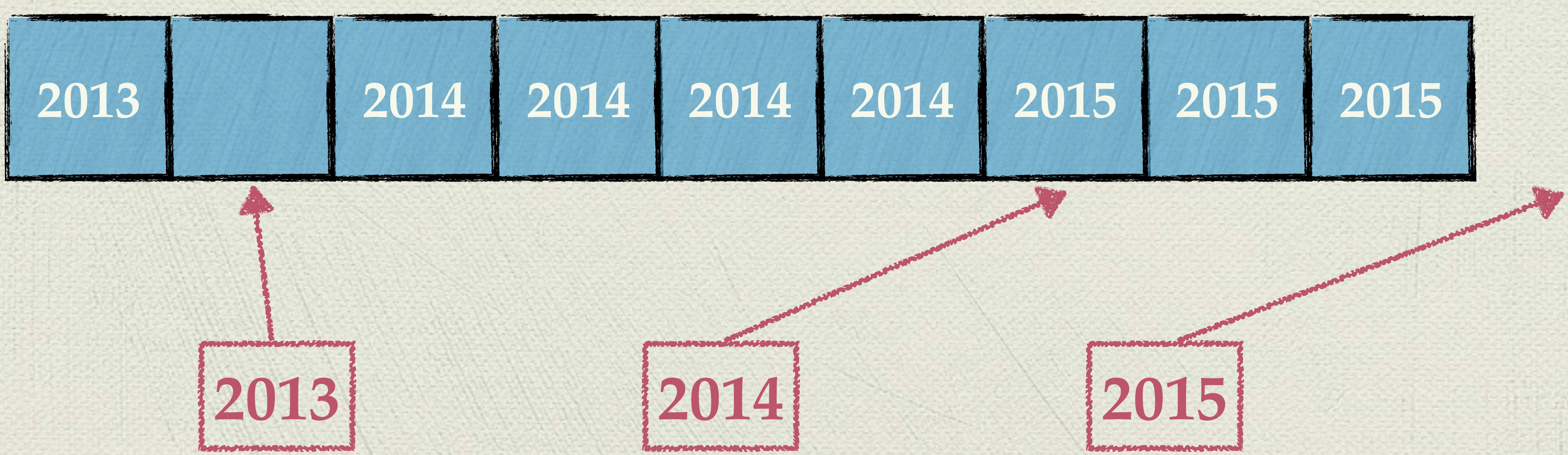
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



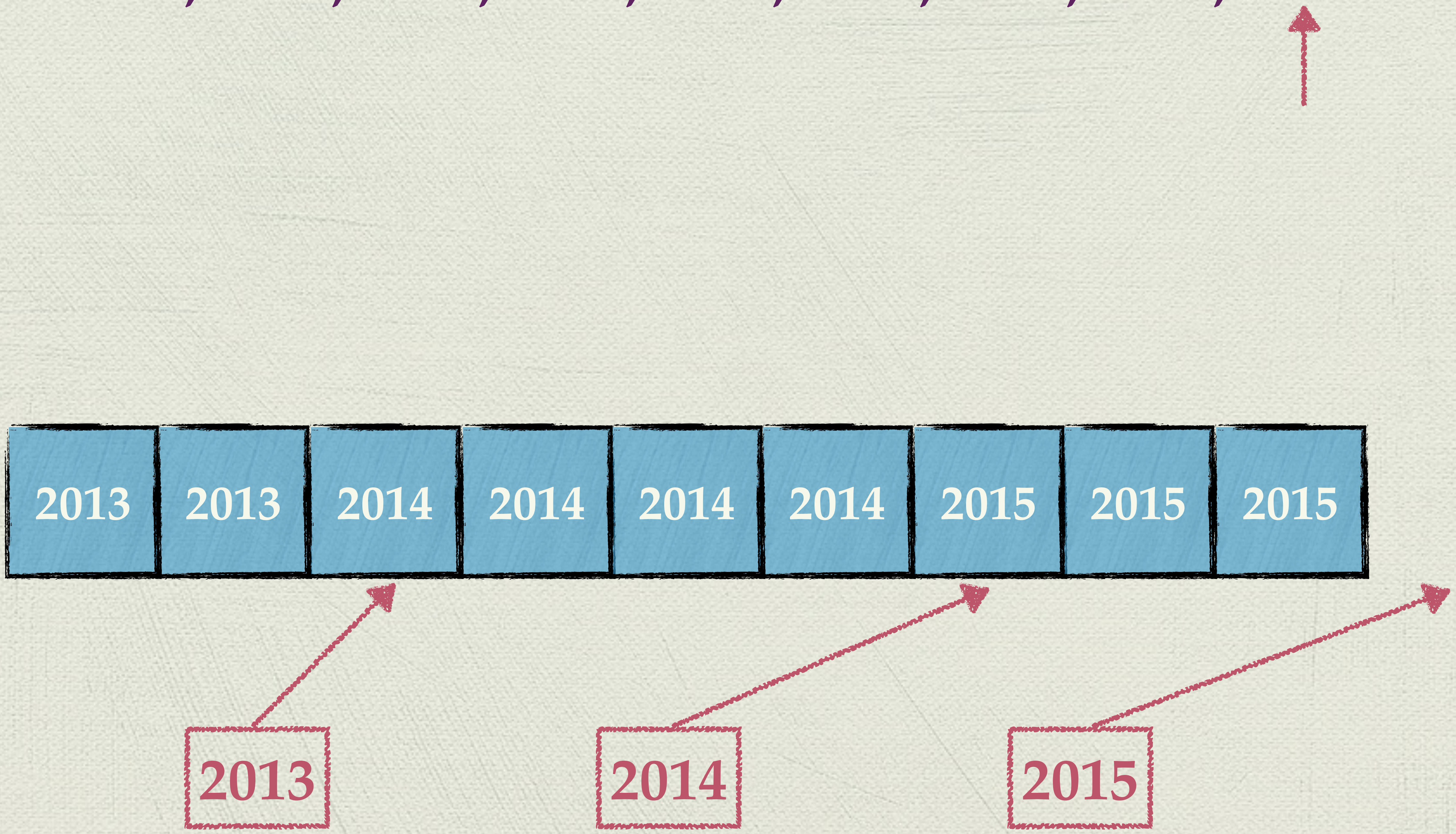
Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



Counting sort walkthrough

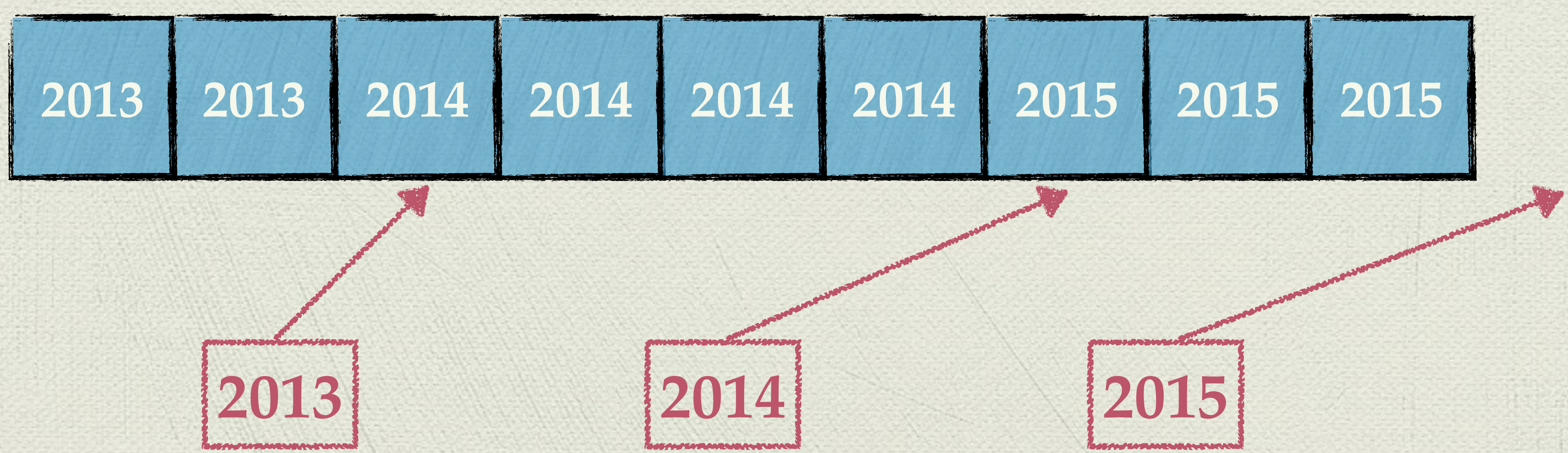
2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013



Counting sort walkthrough

2015, 2014, 2015, 2013, 2015, 2014, 2014, 2014, 2013

Done!



Counting sort runtime?

- We just iterated through our list twice, once to count up the items, and once to place items
- So this is O(2N), or **O(N)**!

Counting sort runtime?

- **O(N)** seems too good to be true
- What's the catch?
- We essentially had to sort our tallies — 2013, 2014, or 2015 — beforehand. But since there were only three things, this could be considered constant time

Counting sort runtime?

- **Conclusion:** If the *variety* of things we're sorting is small, counting sort is by far the fastest

Sorting, what's the point?

- Sorting is essentially a solved problem
- If you need to sort things in your own code, just call standard library functions

Sorting, what's the point?

- We study sorting as a case study of algorithm design
- What's important is the thought process behind analyzing which algorithms are appropriate in which situations
  - Do I need properties like stability, or can I get away without them?
  - If I know something special about my data, can I take advantage of that somehow?