Advanced Algorithmic Techniques (COMP523)

Recursion and Divide and Conquer Techniques #2

Recap and plan

Recap and plan

- **Last lecture:**
    - Asymptotic Complexity.
    - Searching in logarithmic time.
    - Finding majority in an array.

Recap and plan

- **Last lecture:**
  - Asymptotic Complexity.
  - Searching in logarithmic time.
  - Finding majority in an array.
- **This lecture:**
  - Sorting with the **MergeSort** algorithm.
  - Sorting with the **QuickSort** algorithm.
  - The limitations of comparison-based sorting.

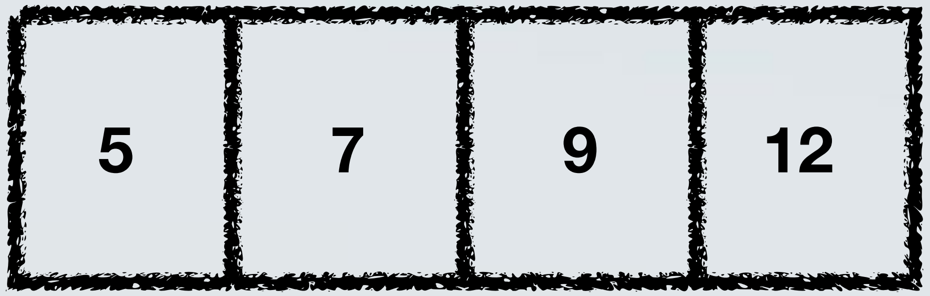
Sorting with Mergesort

Merging two sorted arrays

- Given two sorted arrays **A**[$1,\ldots,n$] and **B**[$1,\ldots,m$], produce a sorted array **C**[$1, \ldots, n+m$] containing all the elements of **A** and **B**.

Merging two sorted arrays

- Given two sorted arrays **A**[*1,…,n*] and **B**[*1,…,m*], produce a sorted array **C**[*1, …, n+m*] containing all the elements of **A** and **B**.



5	7	9	12

Merging two sorted arrays

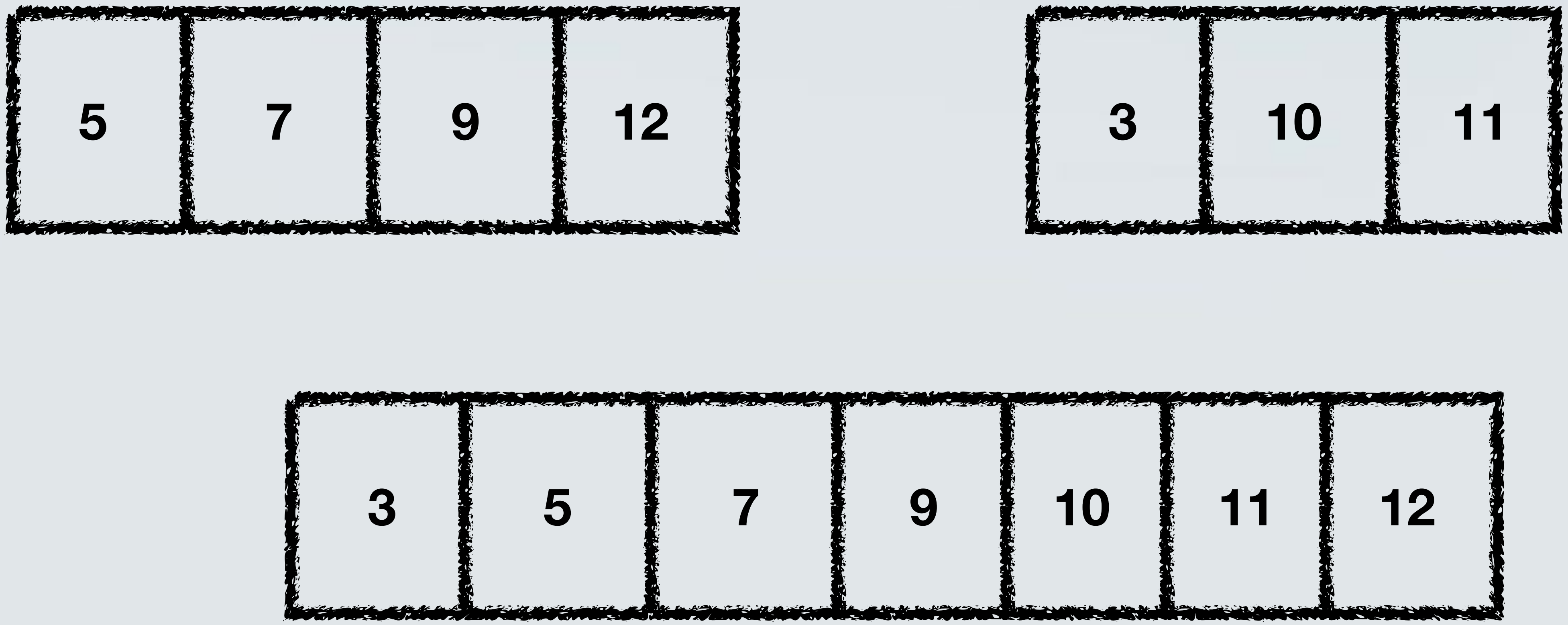
- Given two sorted arrays **A**[*1,…,n*] and **B**[*1,…,m*], produce a sorted array **C**[*1, …, n+m*] containing all the elements of **A** and **B**.

5	**7**	**9**	**12**

3	**10**	**11**

Merging two sorted arrays

- Given two sorted arrays **A**[*1,…,n*] and **B**[*1,…,m*], produce a sorted array **C**[*1, …, n+m*] containing all the elements of **A** and **B**.



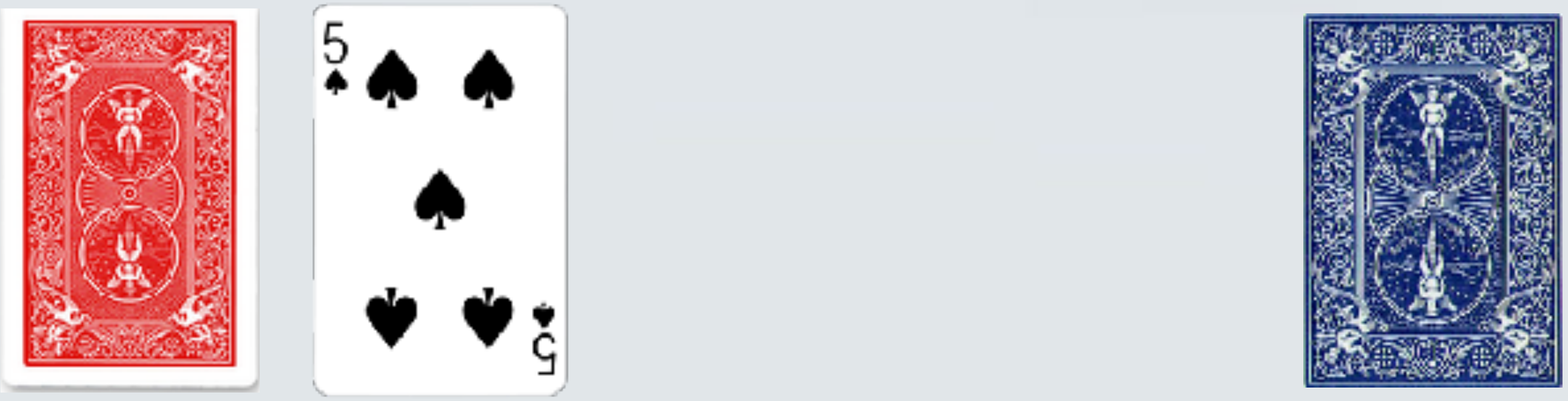
5	7	9	12

3	10	11

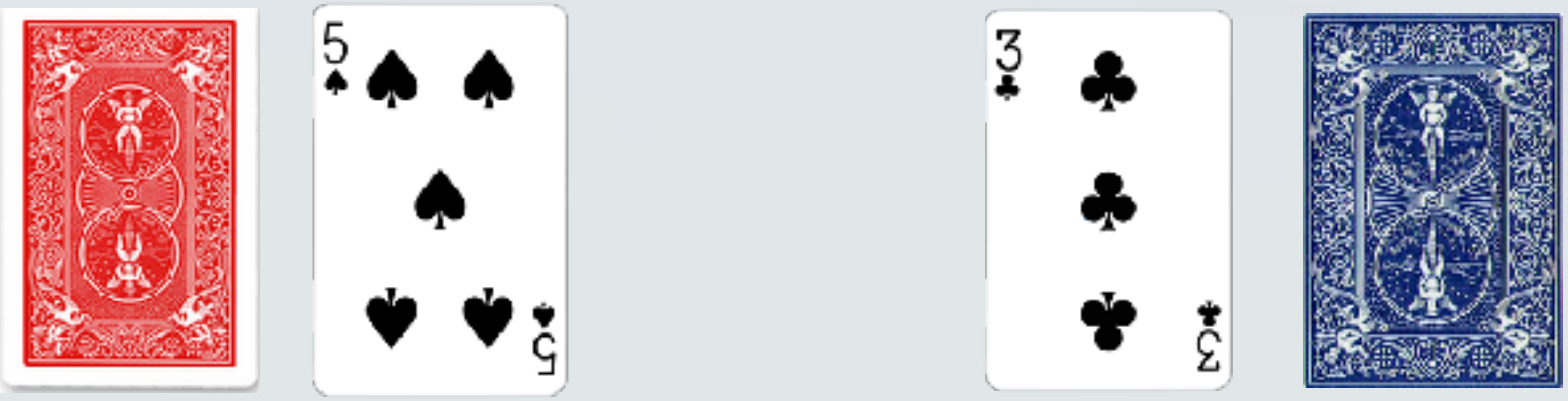
3	5	7	9	10	11	12

How would you do this?

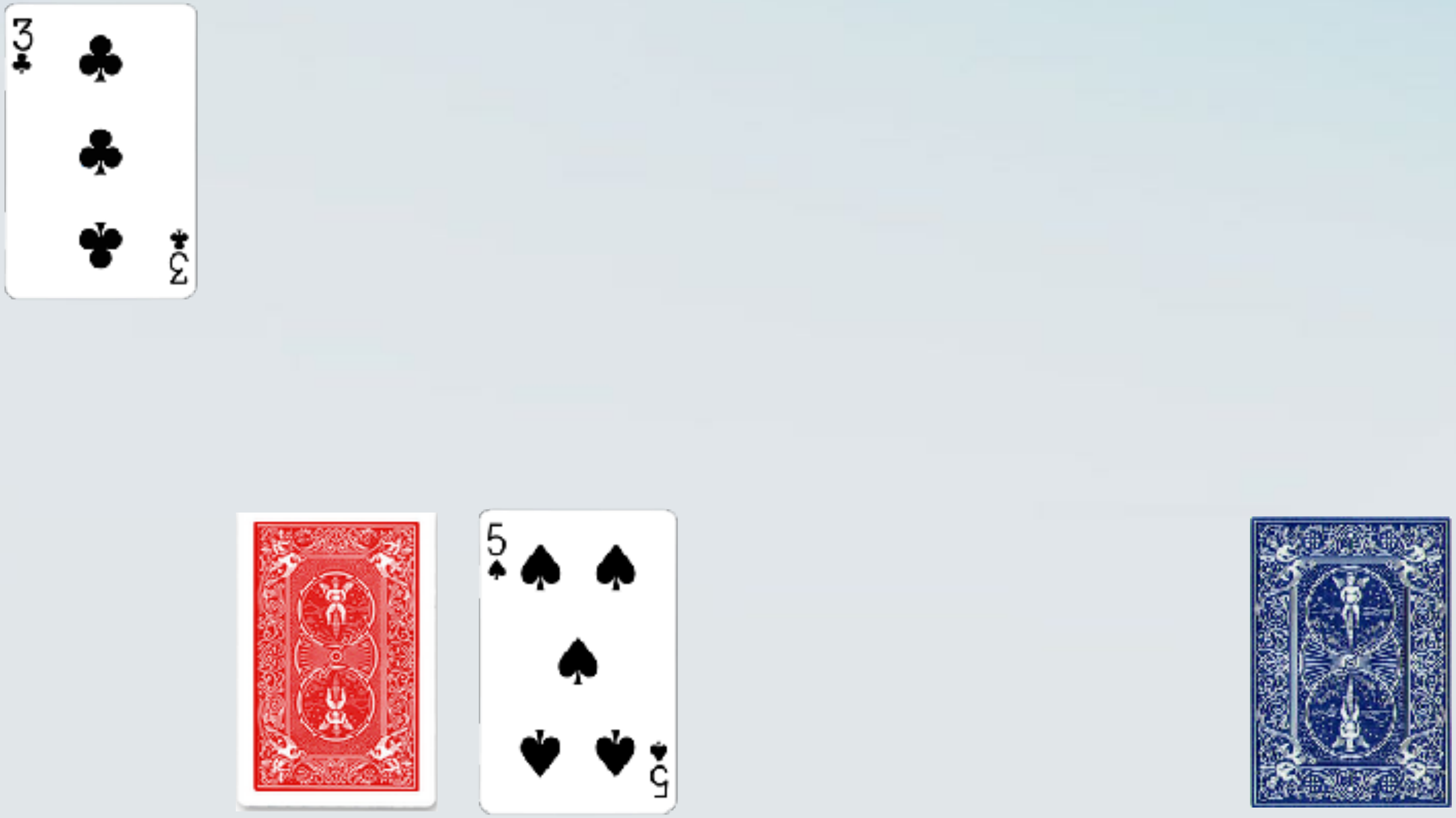
How would you do this?



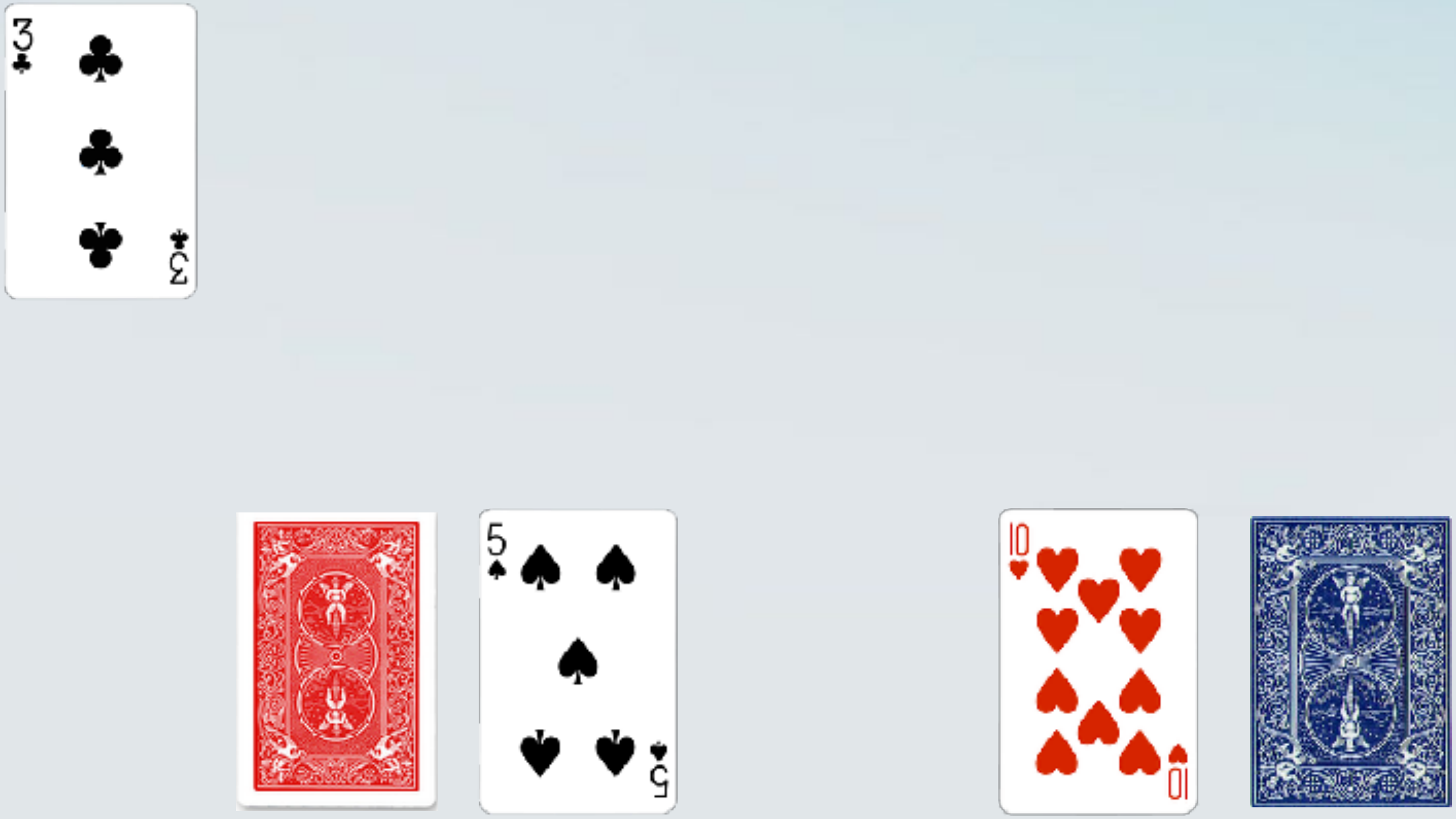
How would you do this?



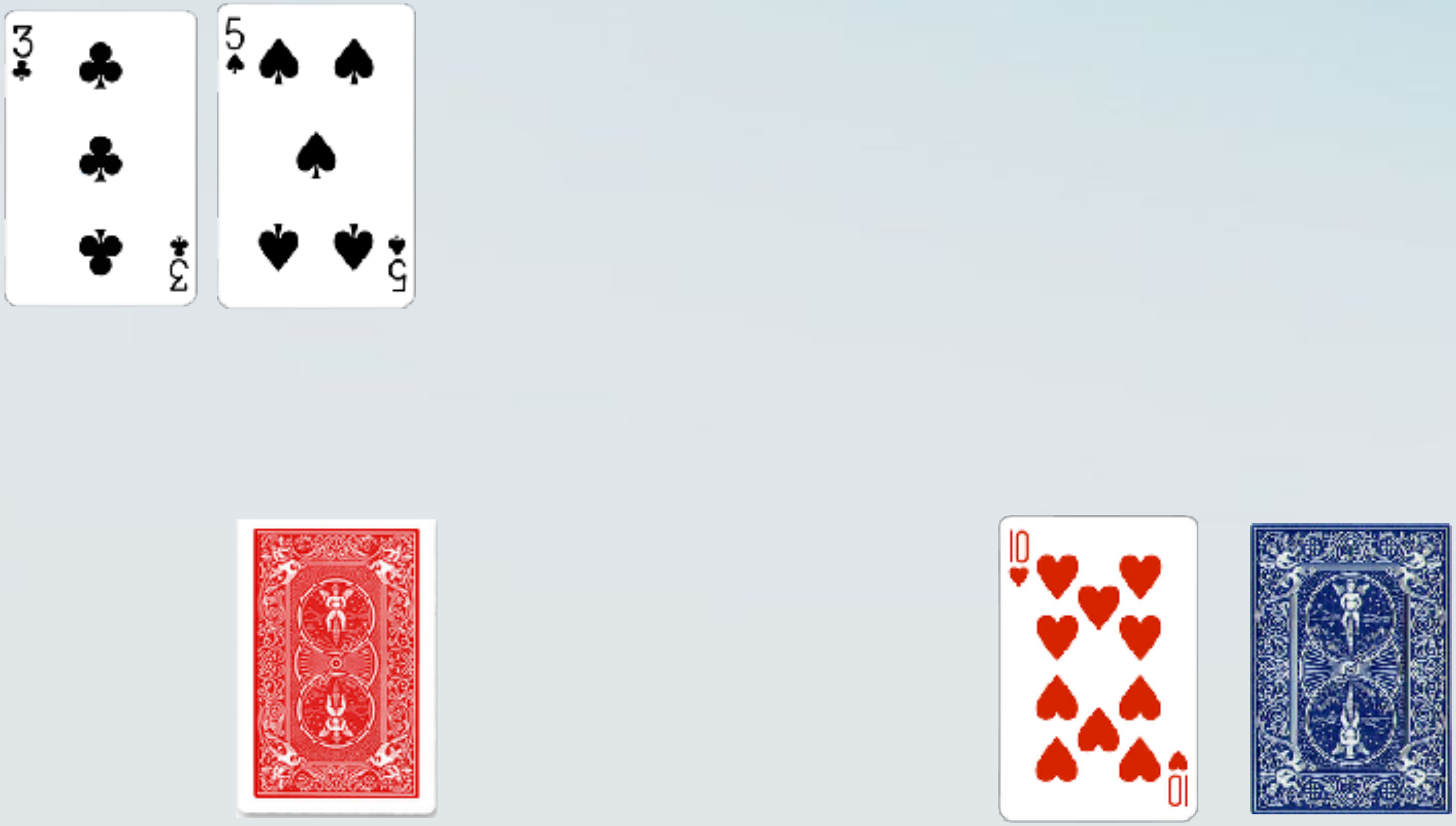
How would you do this?



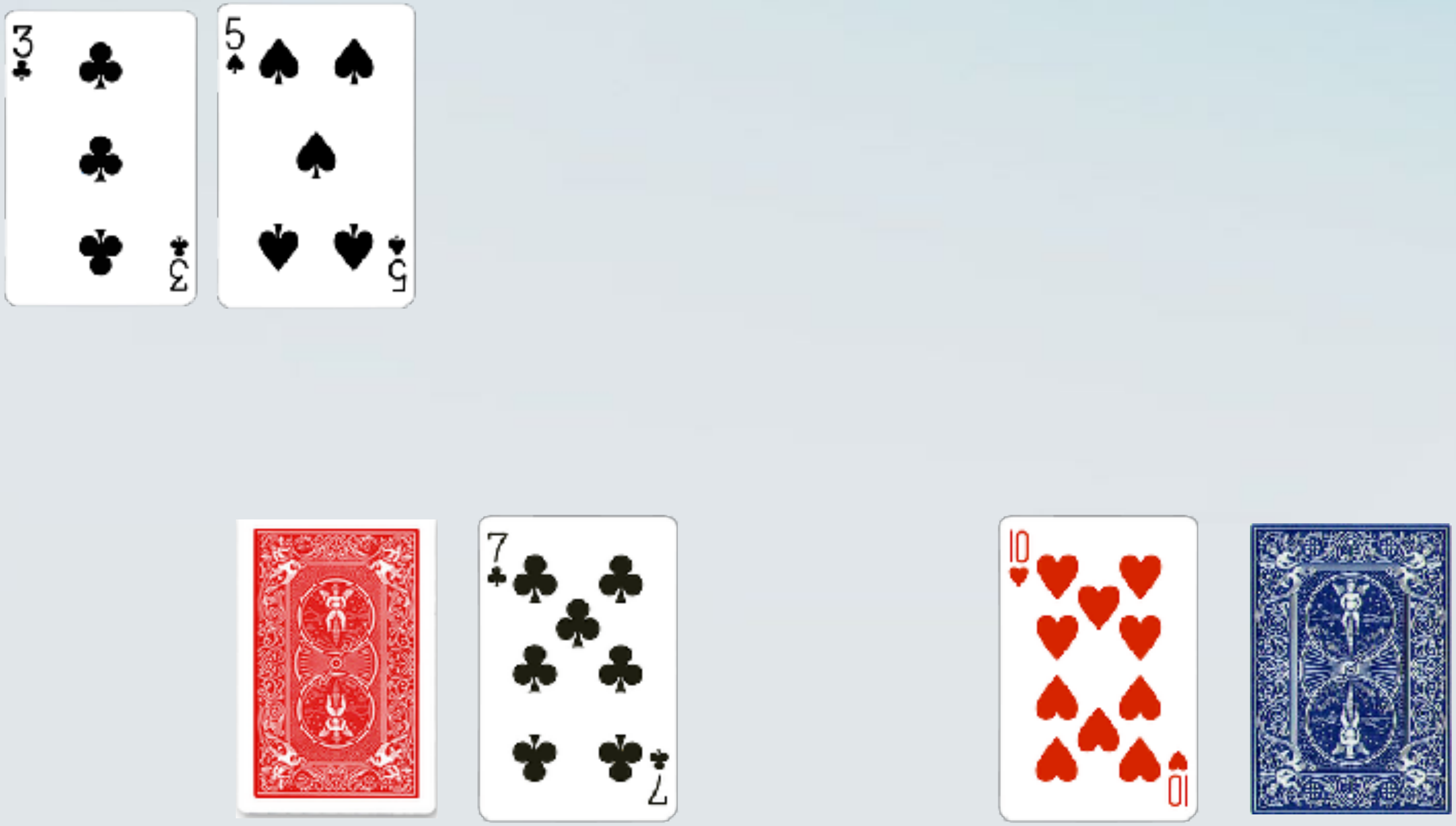
How would you do this?



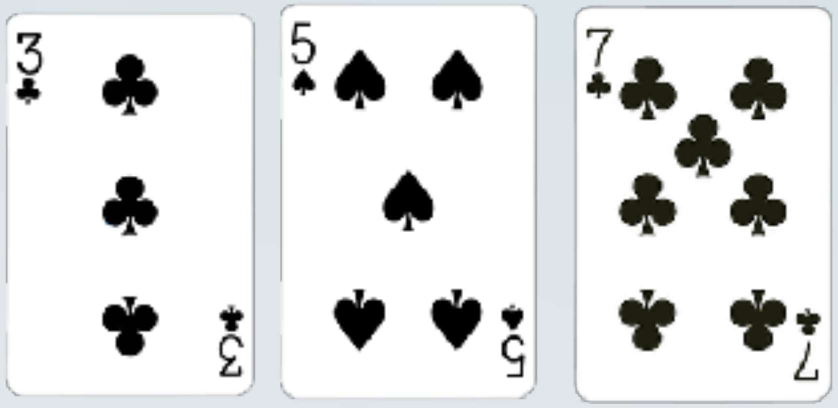
How would you do this?



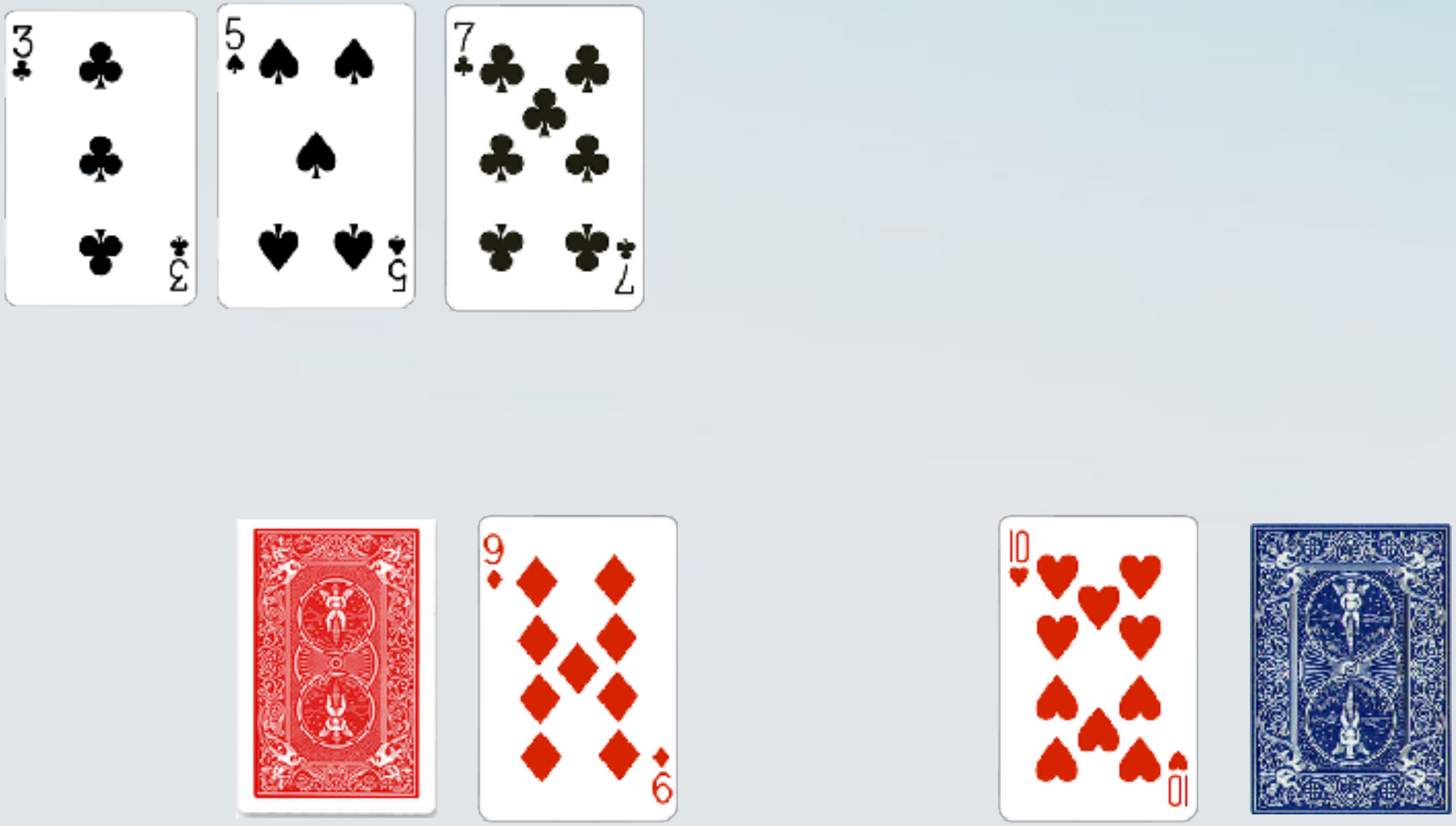
How would you do this?



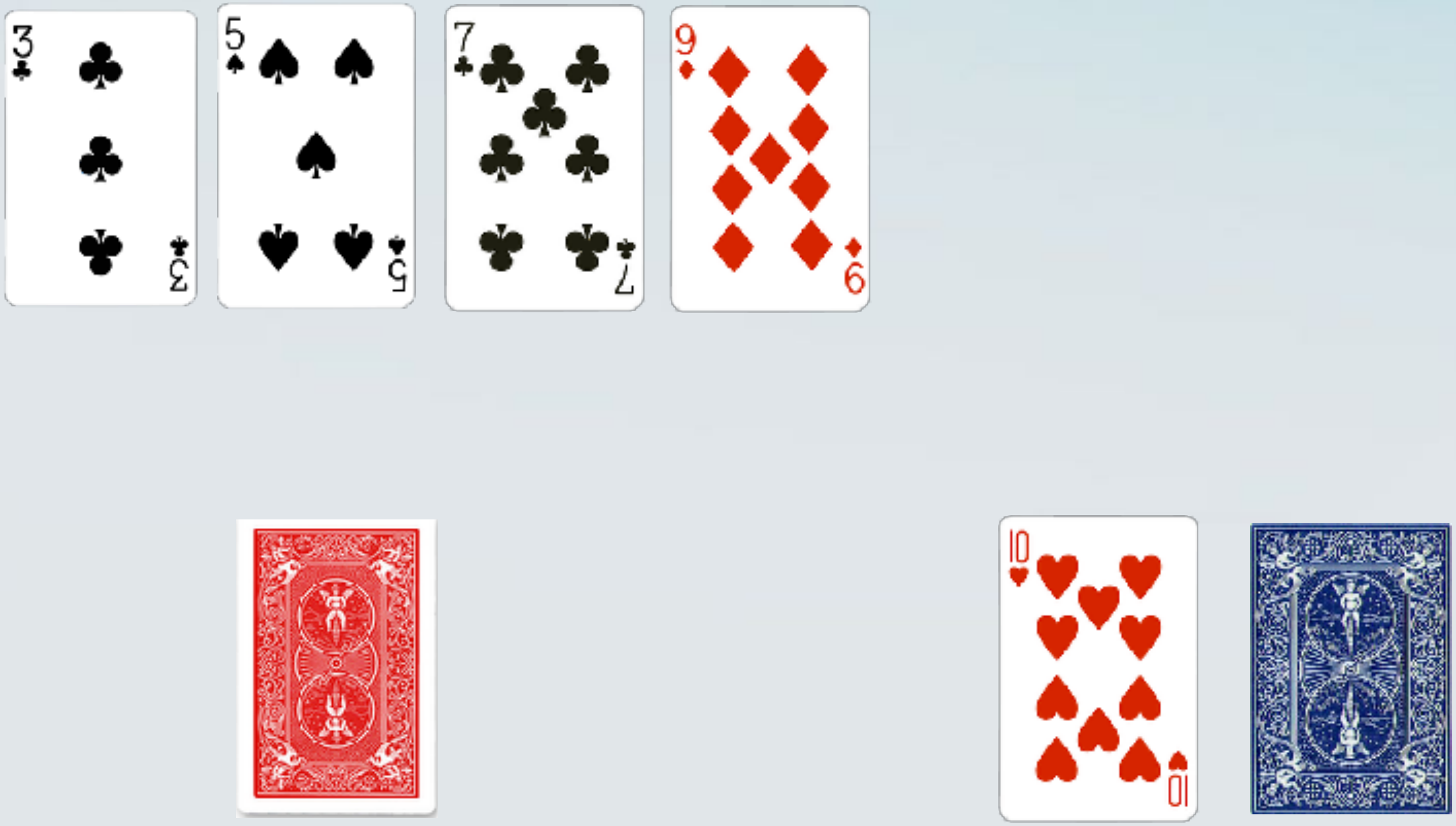
How would you do this?



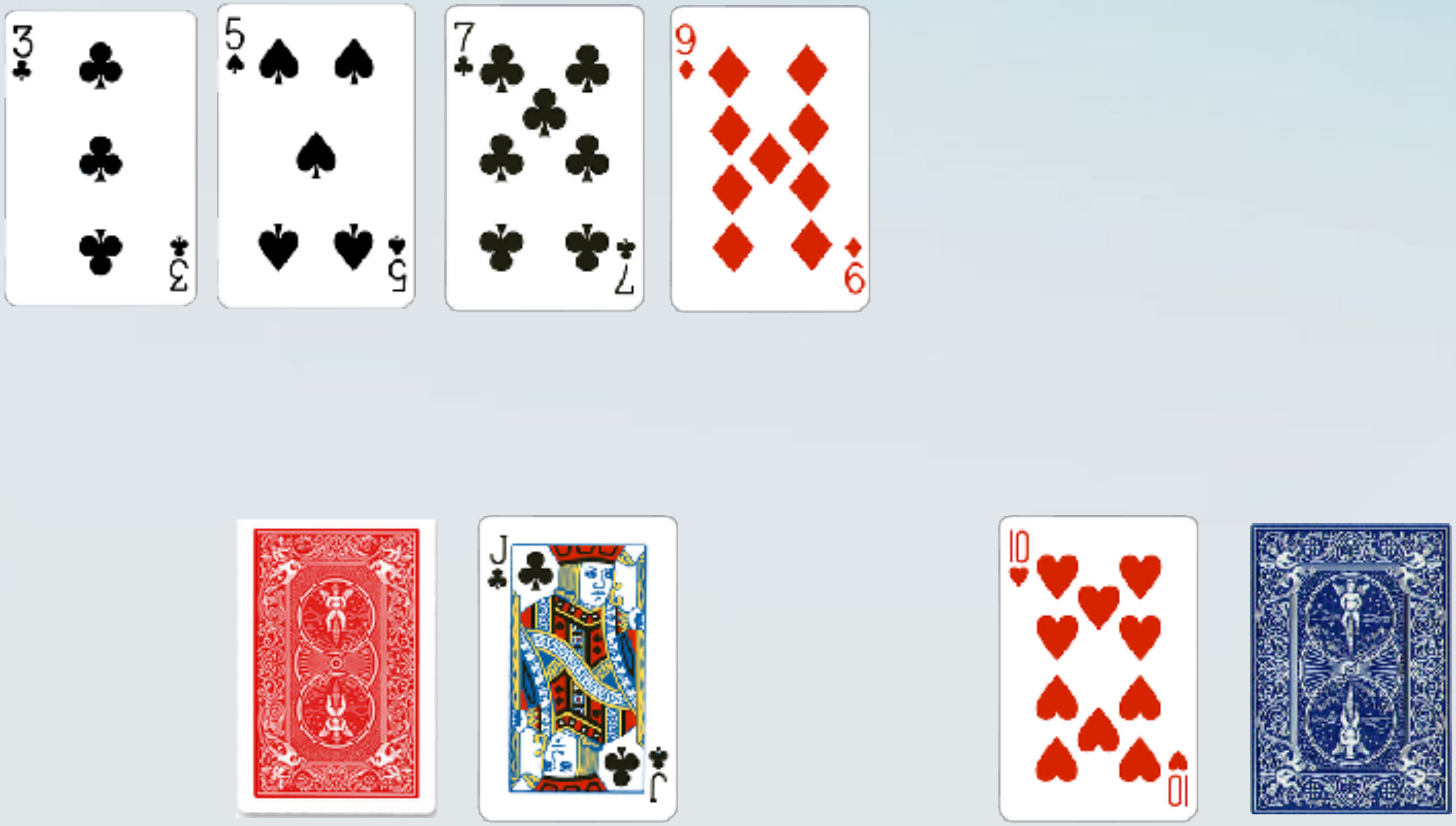
How would you do this?



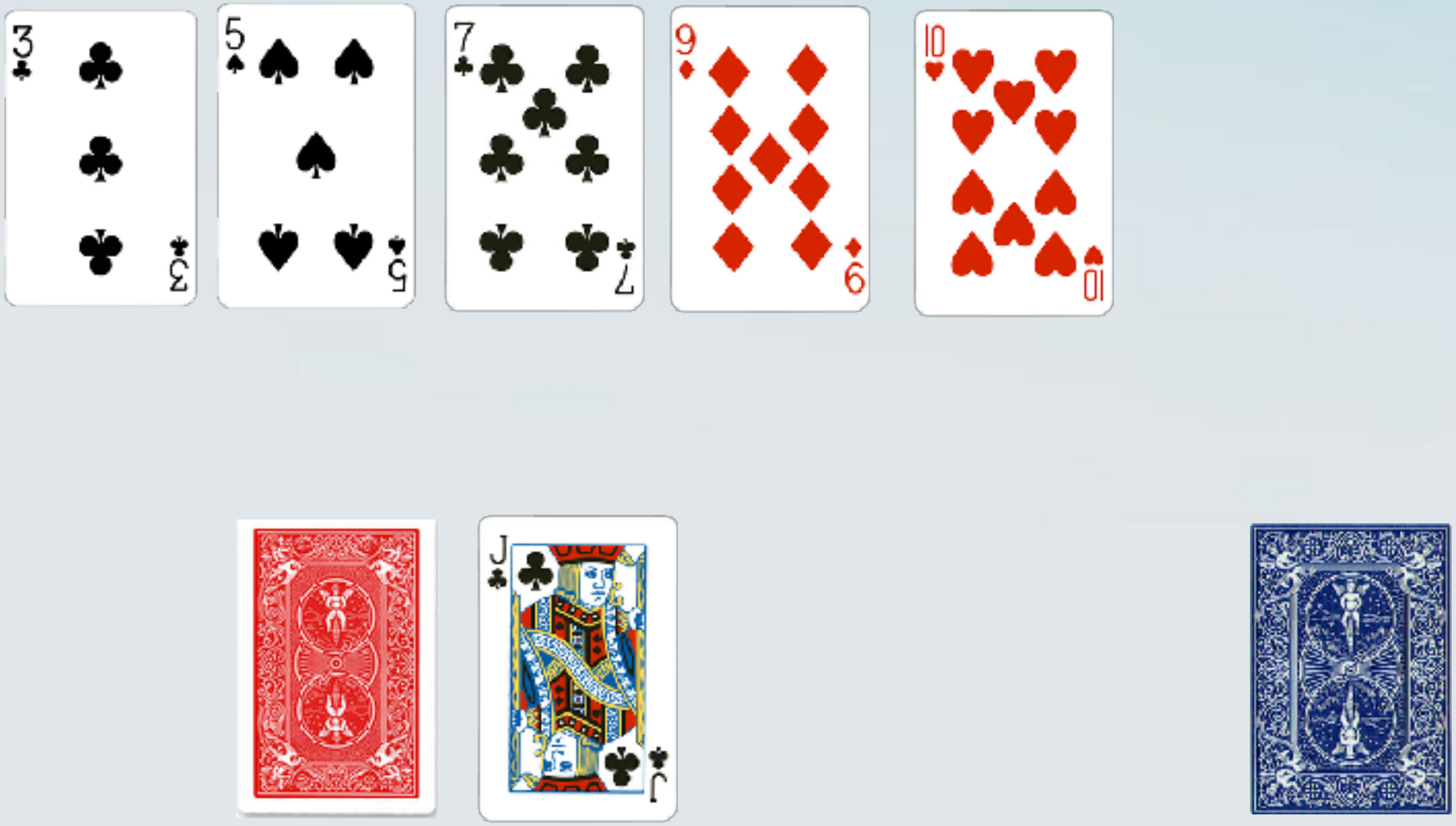
How would you do this?



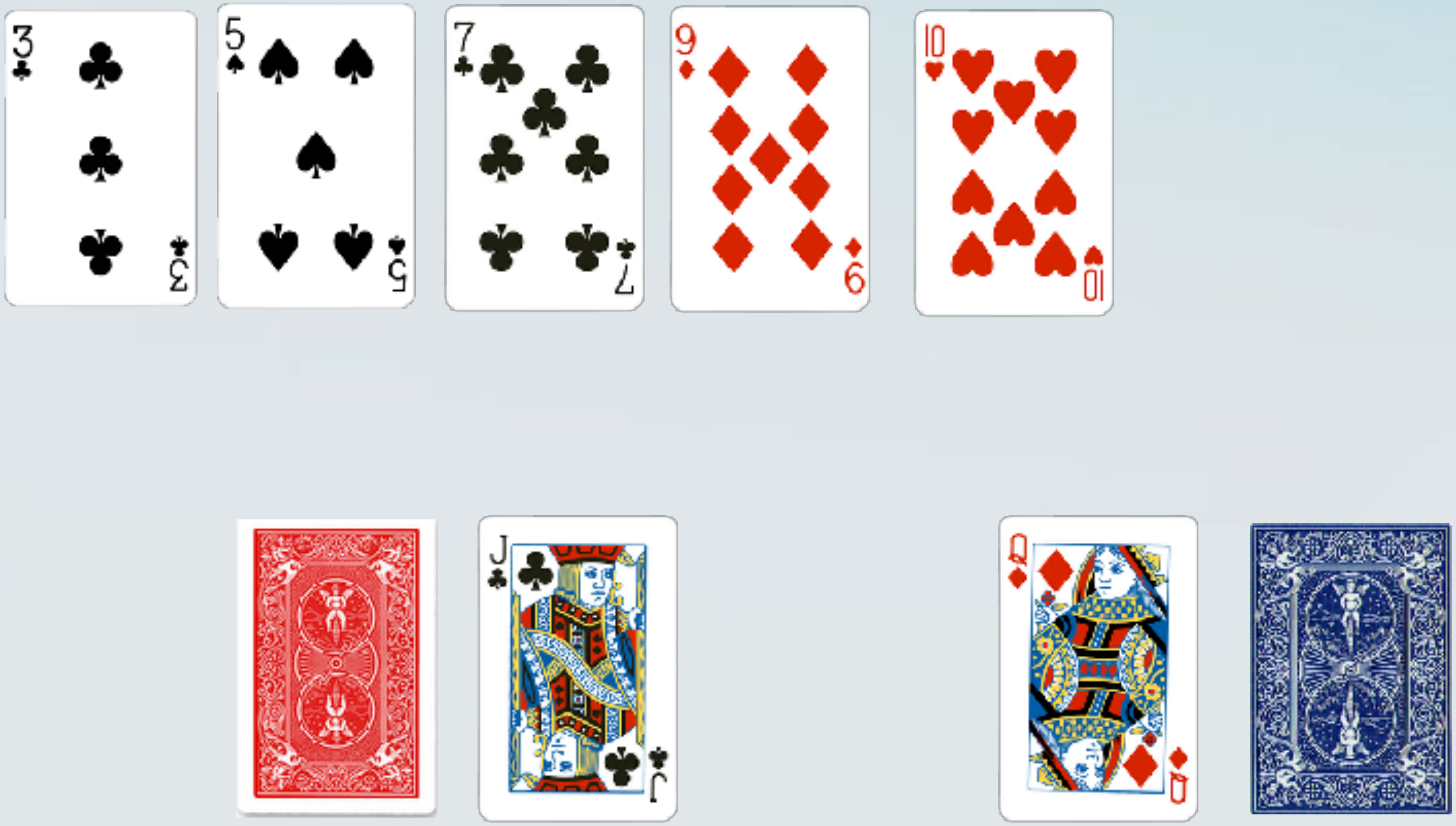
How would you do this?



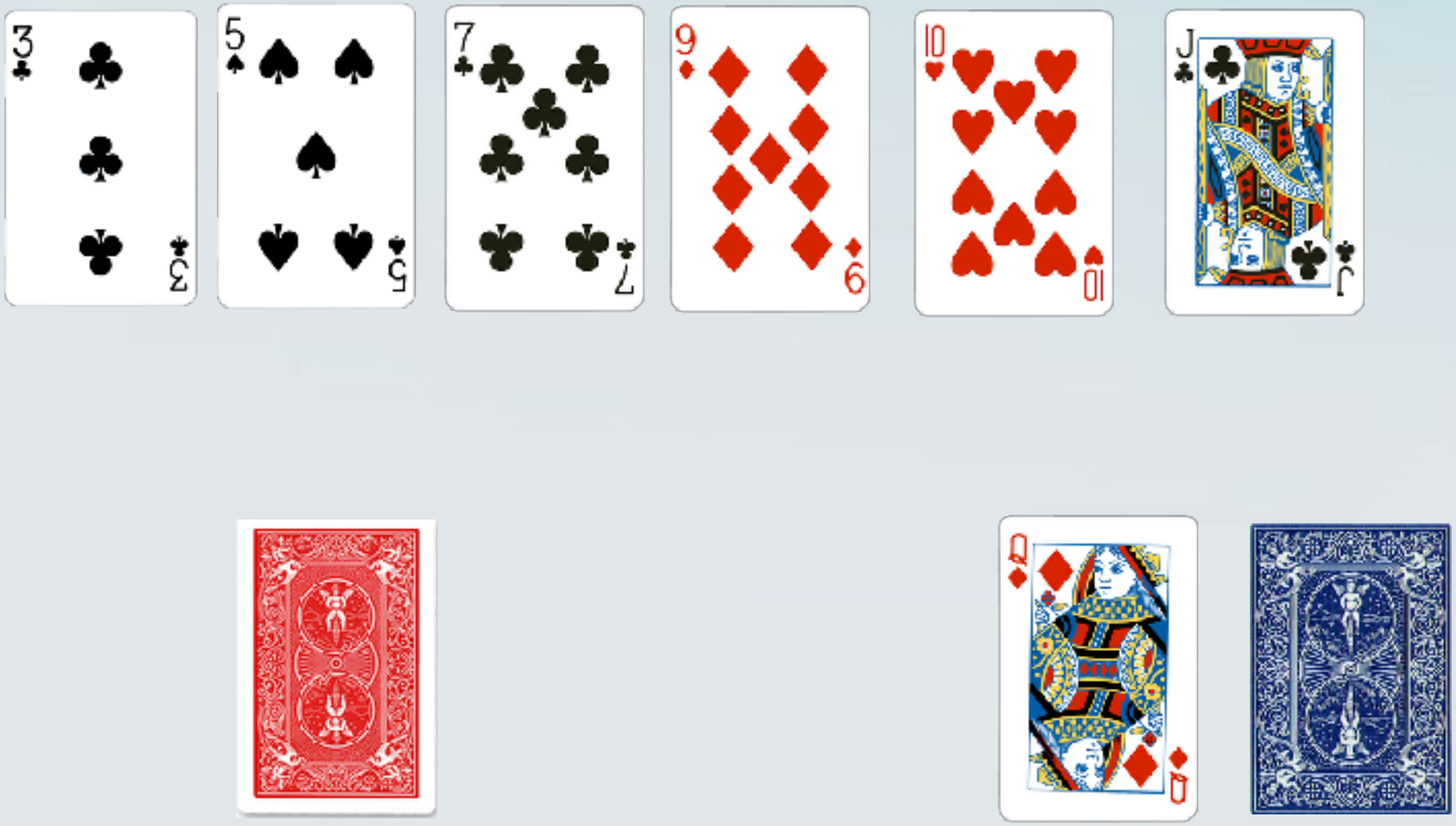
How would you do this?



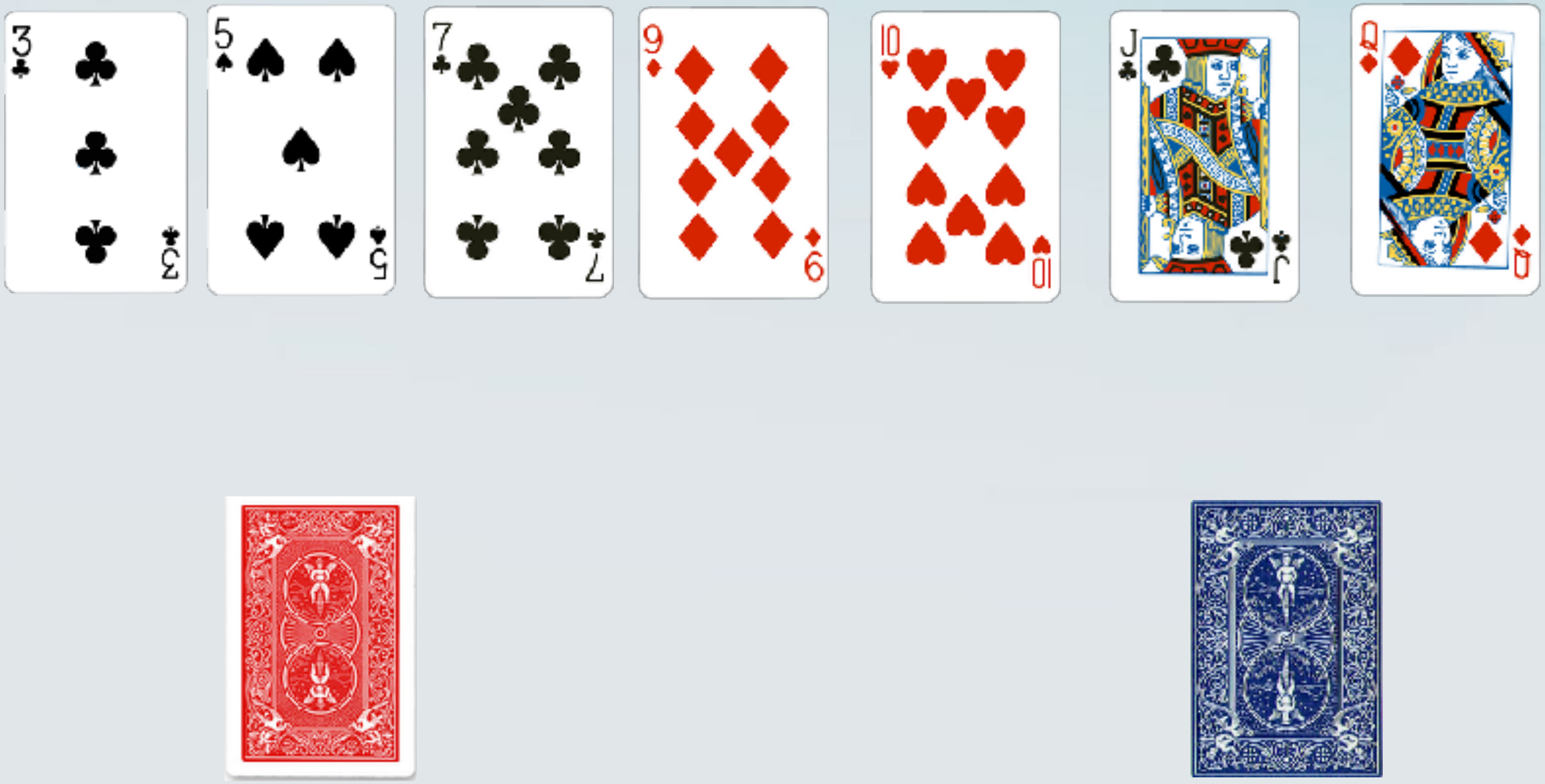
How would you do this?



How would you do this?



How would you do this?



Procedure Merge

```
Procedure Merge(A, B)
/* Recall that |A| = n and |B| = m */

Initialise array C of size n+m

i=1, j=1

For k=1, … , m+n-1

    If A[i] ≤ B[j]
        C[k] = A[i]
        i=i+1
    Else
        C[k] = B[j]
        j=j+1
```

Procedure Merge

```
Procedure Merge(A, B)
/* Recall that |A| = $n$ and |B| = $m$ */

Initialise array C of size n+m

$i$=1, $j$=1

For $k$=1, … , m+n-1

    If A[$i$] ≤ B[$j$]
        C[$k$] = A[$i$]
        $i$=$i$+1
    Else
        C[$k$] = B[$j$]
        $j$=$j$+1
```

Procedure **Merge**

```
Procedure Merge(A, B)
/* Recall that |A| = $n$ and |B| = $m$ */

Initialise array C of size n+m

$i$=1, $j$=1

For $k$=1, … , m+n-1

    If A[$i$] ≤ B[$j$]
        C[$k$] = A[$i$]
        $i$=$i$+1
    Else
        C[$k$] = B[$j$]
        $j$=$j$+1
```

What is the running time of **Merge**?

Procedure Merge

```
Procedure Merge(A, B)
/* Recall that |A| = n and |B| = m */

Initialise array C of size n+m

i=1, j=1

For k=1, … , m+n-1

    If A[i] ≤ B[j]
        C[k] = A[i]
        i=i+1
    Else
        C[k] = B[j]
        j=j+1
```

What is the running time of **Merge**?

How many times can an element be compared in the worst case?

Procedure Merge

```
Procedure Merge(A, B)
/* Recall that |A| = $n$ and |B| = $m$ */

Initialise array C of size n+m

$i$=1, $j$=1

For $k$=1, … , m+n-1

    If A[$i$] $\leq$ B[$j$]
        C[$k$] = A[$i$]
        $i$=$i$+1
    Else
        C[$k$] = B[$j$]
        $j$=$j$+1
```

What is the running time of **Merge**?

How many times can an element be compared in the worst case?

1, 3 ,5 , … , 2n-1

n, n+2 ,n+4 , … , 3n-2

Procedure **Merge**

```
Procedure Merge(A, B)
/* Recall that |A| = $n$ and |B| = $m$ */

Initialise array C of size n+m

$i$=1, $j$=1

For $k$=1, … , m+n-1

    If A[$i$] ≤ B[$j$]
        C[$k$] = A[$i$]
        $i$=$i$+1
    Else
        C[$k$] = B[$j$]
        $j$=$j$+1
```

What is the running time of **Merge**?

How many times can an element be compared in the worst case?

1, 3 ,5 , … , 2n-1

n, n+2 ,n+4 , … , 3n-2

Charging argument: The cost of each iteration is “charged” to the “winner” of the comparison.

Procedure Merge

```
Procedure Merge(A, B)
/* Recall that |A| = n and |B| = m */

Initialise array C of size n+m

i=1, j=1

For k=1, … , m+n-1

    If A[i] ≤ B[j]
        C[k] = A[i]
        i=i+1
    Else
        C[k] = B[j]
        j=j+1
```

What is the running time of Merge?

How many times can an element be compared in the worst case?

1, 3 ,5 , … , 2n-1

n, n+2 ,n+4 , … , 3n-2

Charging argument: The cost of each iteration is “charged” to the “winner” of the comparison.

O(m+n)

The **Mergesort** algorithm

- Divide and conquer algorithm.
- Split the array **A**[$1,\dots,n$] to two subarrays, **A**[$1,\dots,n/2$] and **A**[$n/2+1, \dots, n$]
- Sort each subarray using **Mergesort**.
  - Stop the recursion when the subarray contains only one element.
- Merge the sorted subarrays **A**[$1,\dots,n/2$] and **A**[$n/2+1, \dots, n$] using the **Merge** procedure.

Mergesort pseudocode

```
Algorithm Mergesort($\mathbf{A}[i,\ldots,j]$)

If $i=j$, return $i$

$q=(i+j)/2$

$\mathbf{A}_{left}$=Mergesort($\mathbf{A}[i,\ldots,q]$)
$\mathbf{A}_{right}$=Mergesort($\mathbf{A}[q+1,\ldots,n]$)
return Merge( $\mathbf{A}_{left}$ , $\mathbf{A}_{right}$ )
```

Mergesort pseudocode

```
Algorithm Mergesort($\mathbf{A}[i,\ldots,j]$)

If $i=j$, return $i$

$q=(i+j)/2$

$\mathbf{A}_{left}=$Mergesort($\mathbf{A}[i,\ldots,q]$)
$\mathbf{A}_{right}=$Mergesort($\mathbf{A}[q+1,\ldots,n]$)
return Merge( $\mathbf{A}_{left}$ , $\mathbf{A}_{right}$ )
```

Initial call: **Mergesort($\mathbf{A}[i,\ldots,n]$)**

Mergesort example

6	4	8	9	2	1	3

Mergesort example

divide

6	4	8	9	2	1	3

Mergesort example

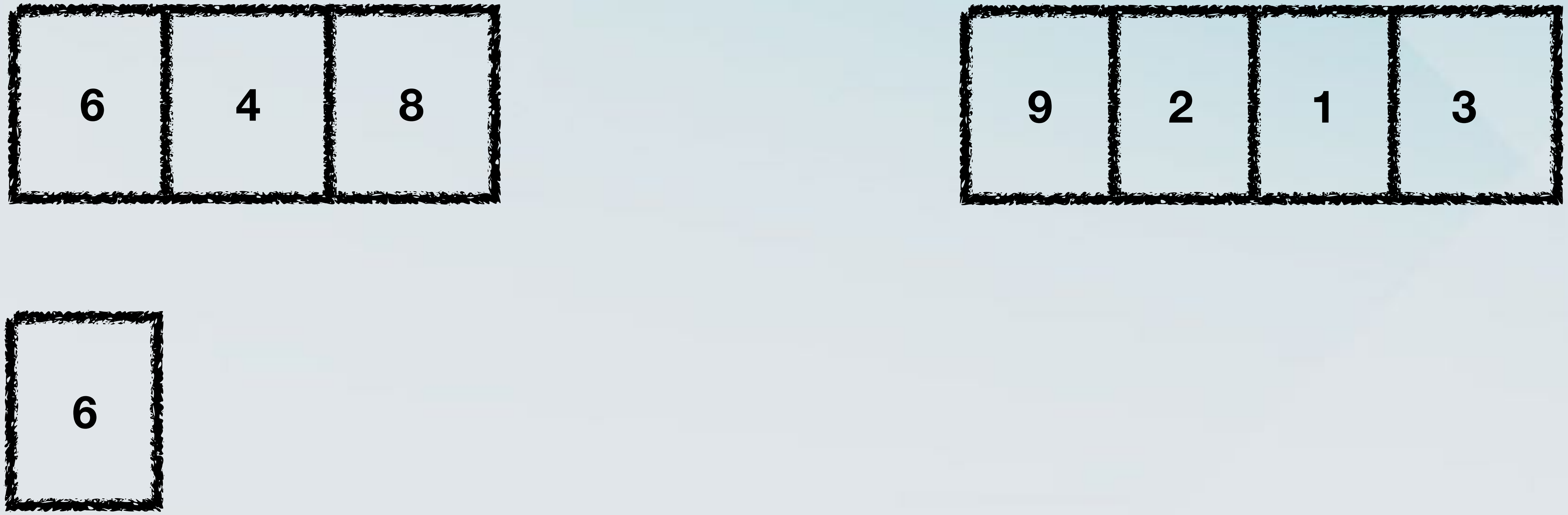
divide

6	4	8

9	2	1	3

Mergesort example

divide



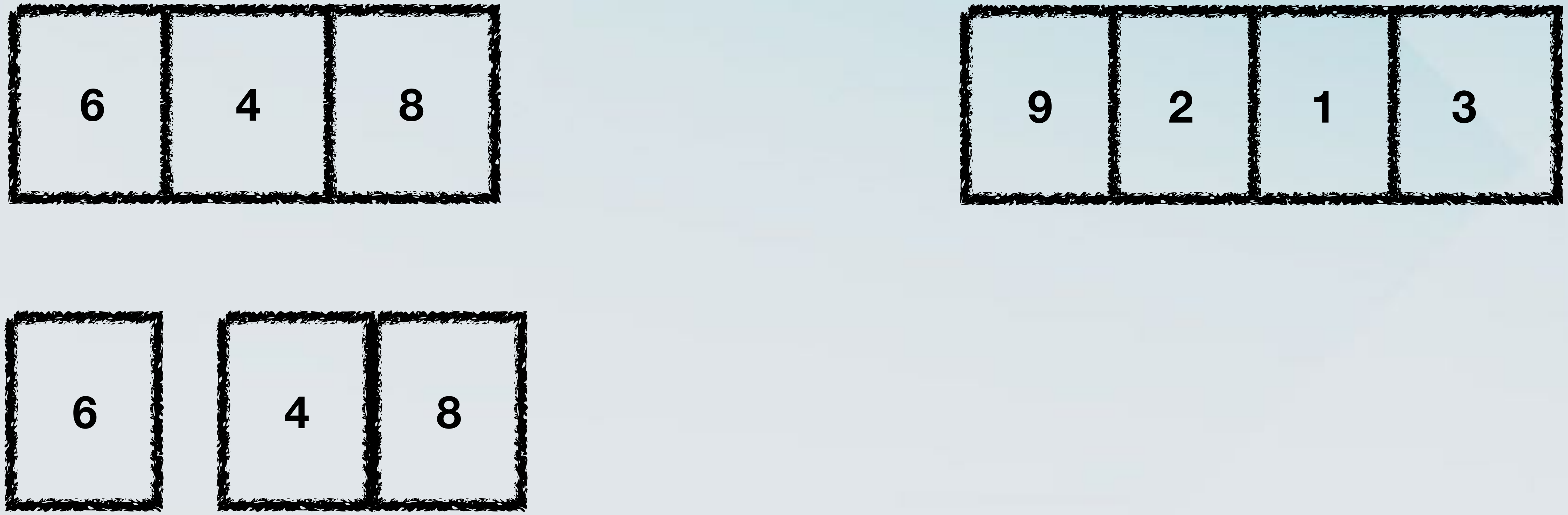
6	4	8

9	2	1	3

6

Mergesort example

divide



6	4	8

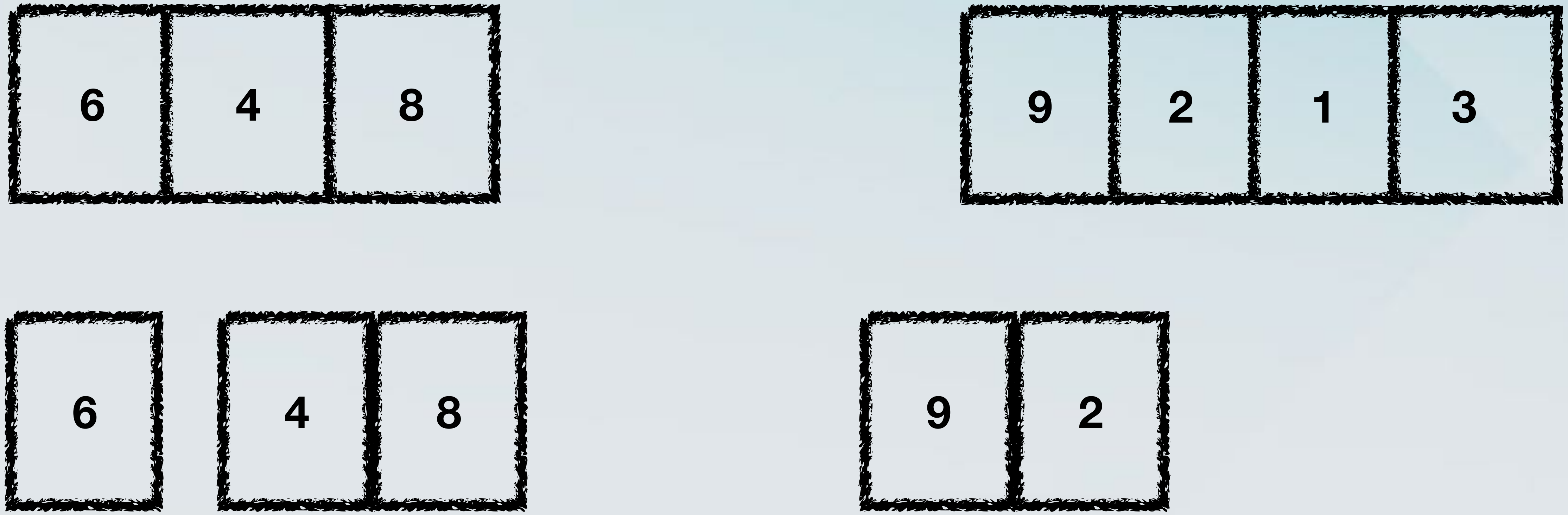
9	2	1	3

6

4	8

Mergesort example

divide



6	4	8

9	2	1	3

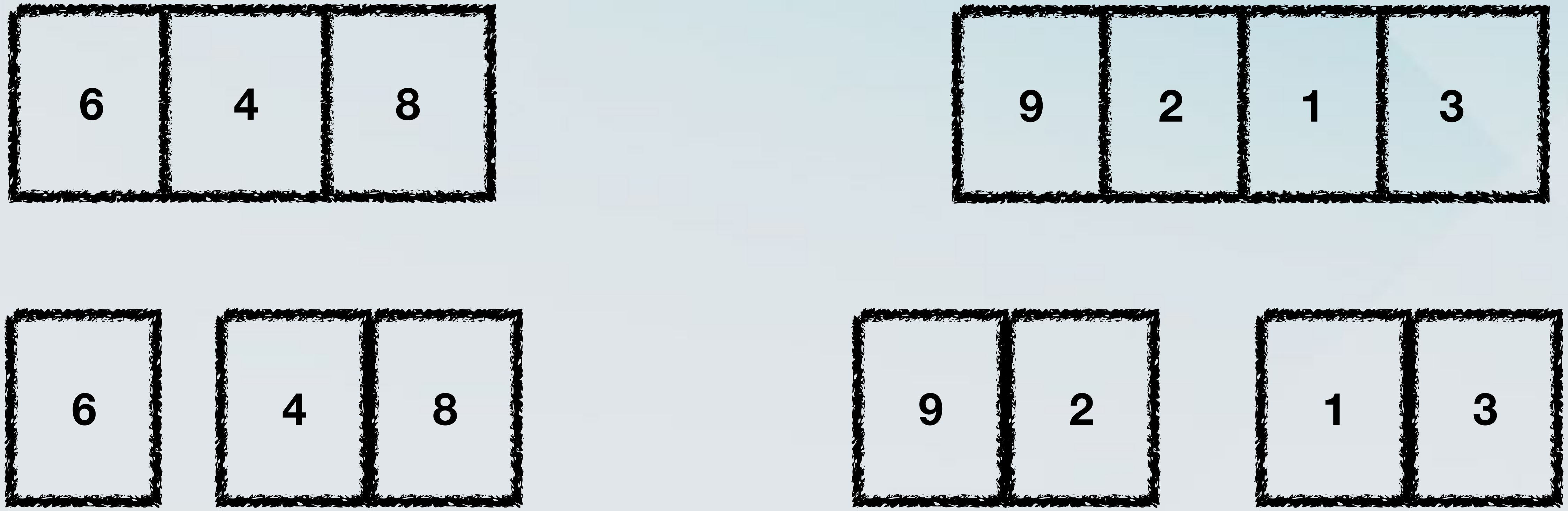
6

4	8

9	2

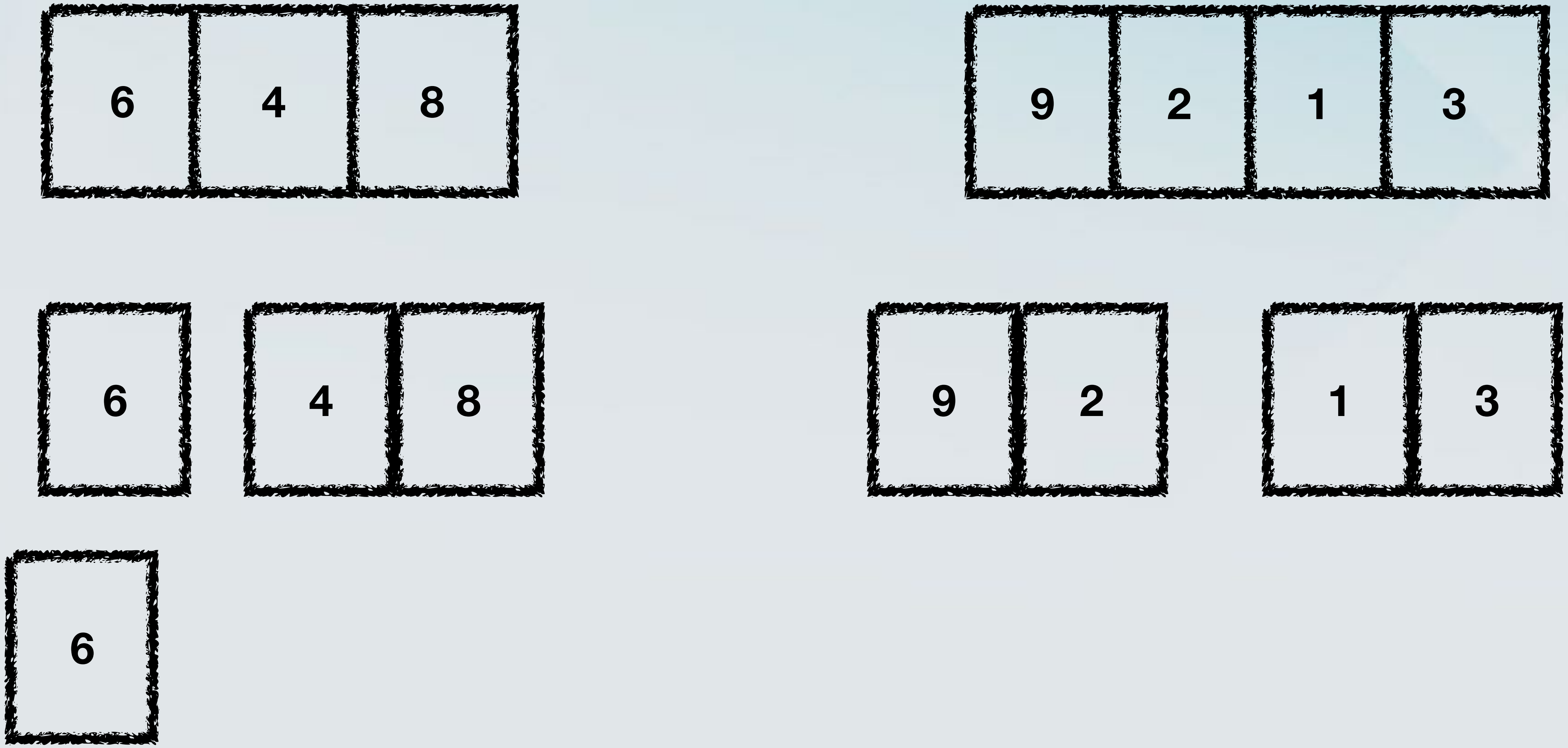
Mergesort example

divide



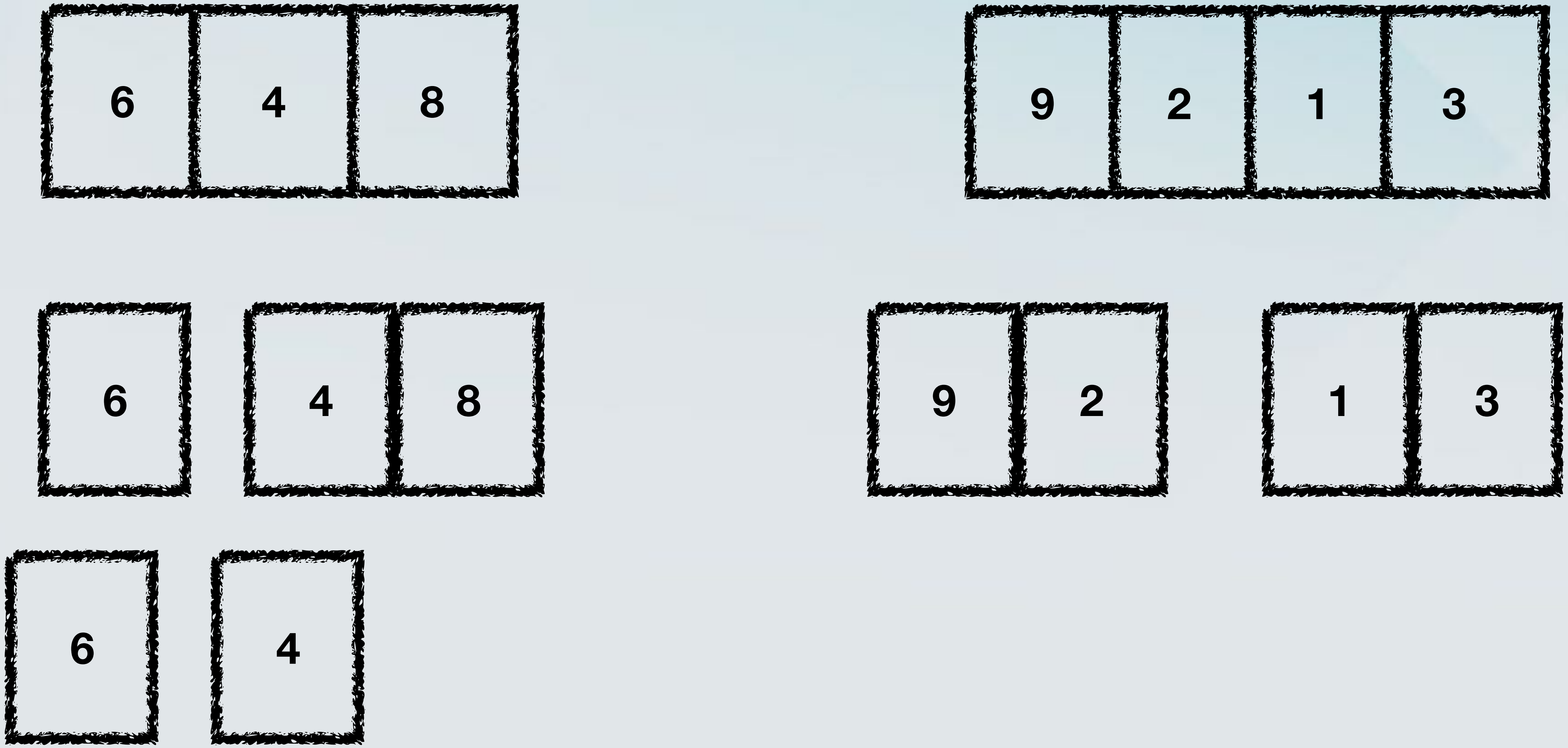
Mergesort example

divide



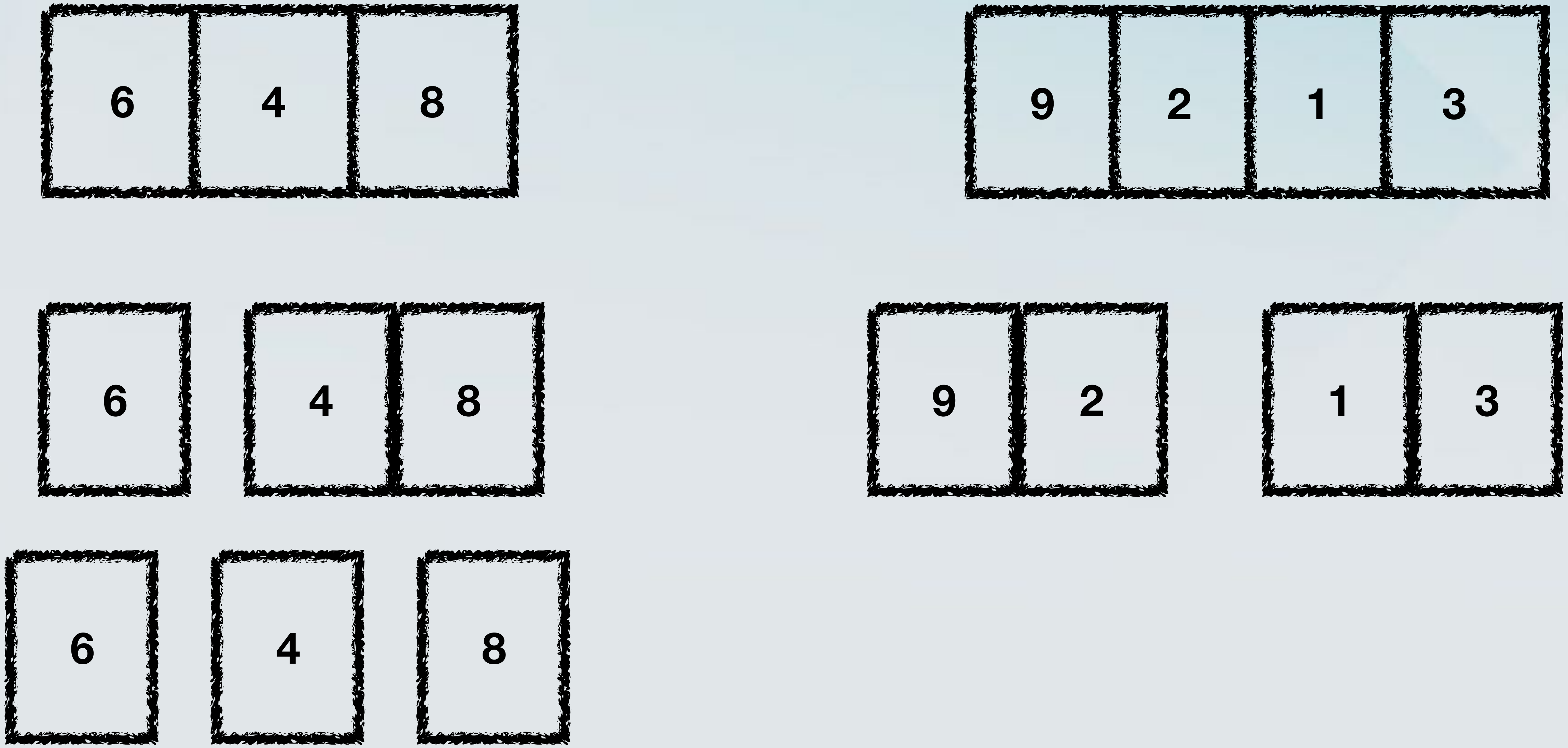
Mergesort example

divide



Mergesort example

divide



6	4	8

9	2	1	3

6

4	8

9	2

1	3

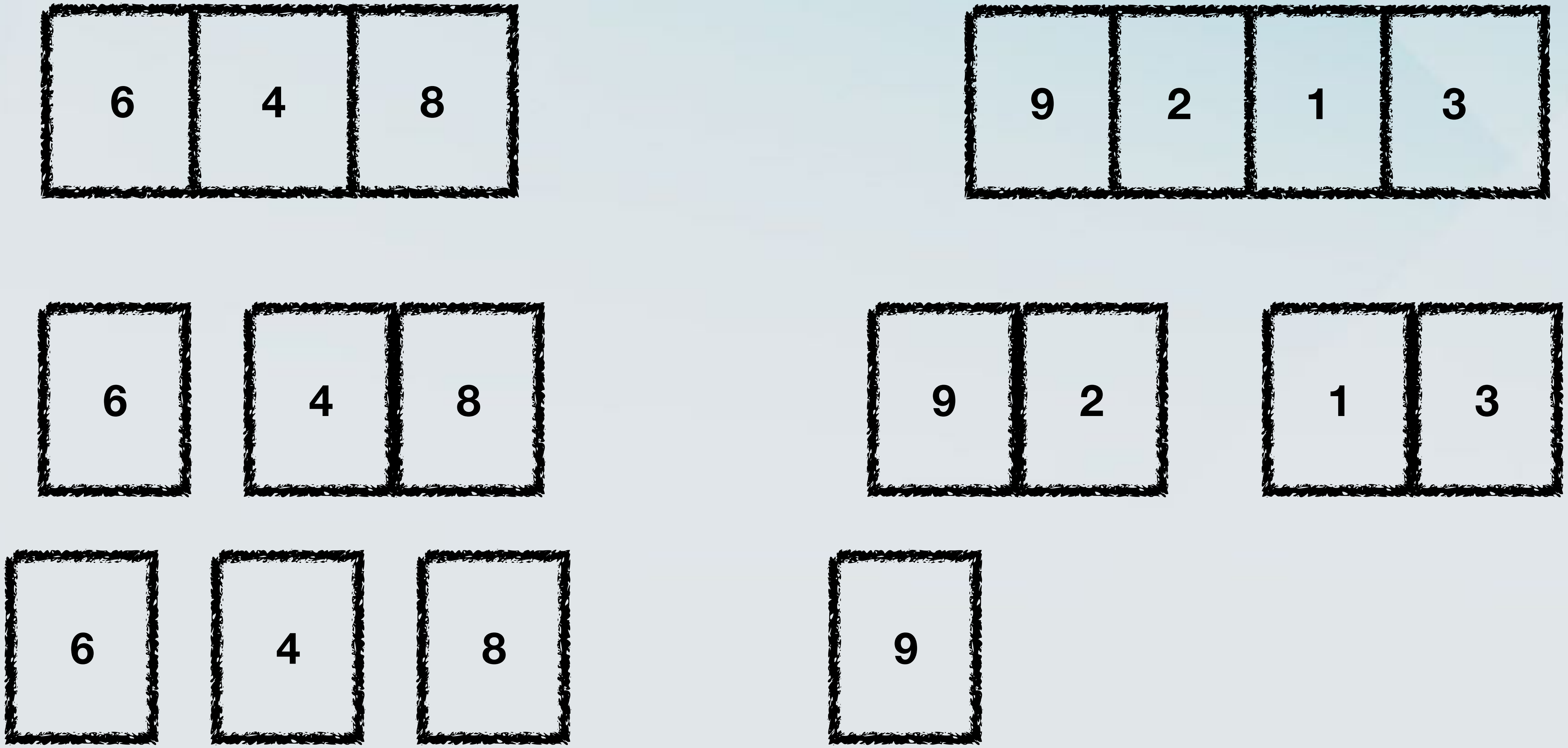
6

4

8

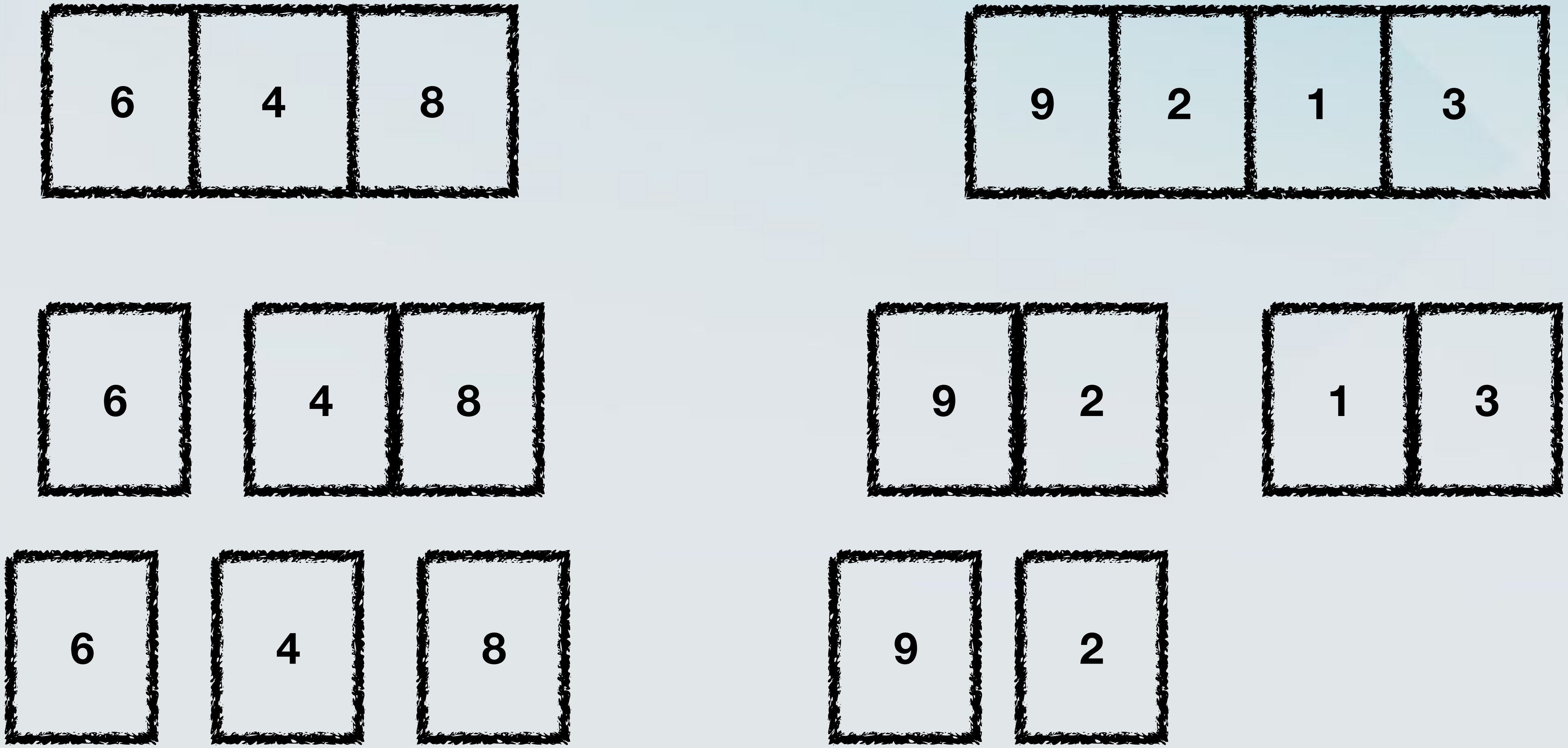
Mergesort example

divide



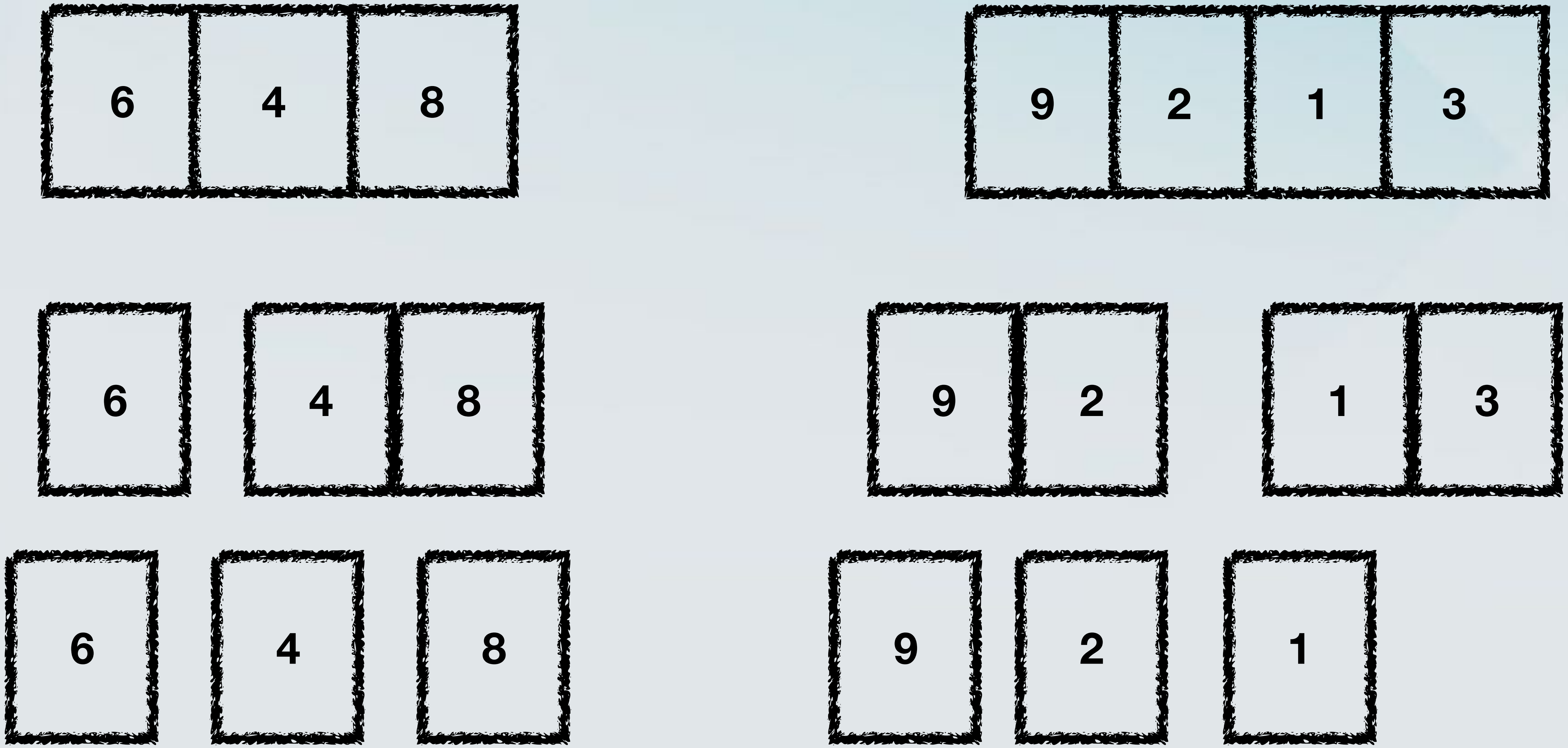
Mergesort example

divide



Mergesort example

divide



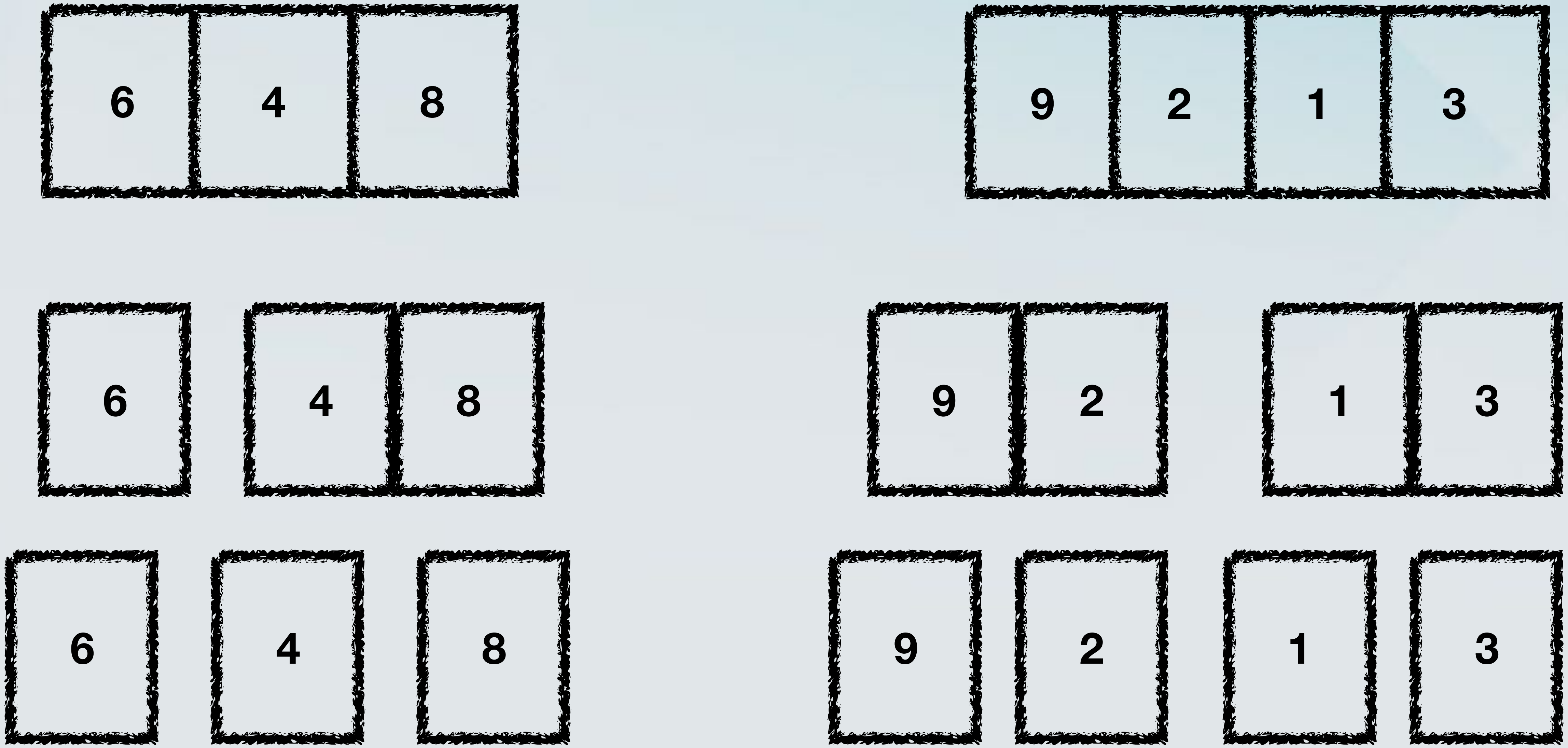
6 4 8 9 2 1 3

6 4 8 9 2 1 3

6 4 8 9 2 1

Mergesort example

divide



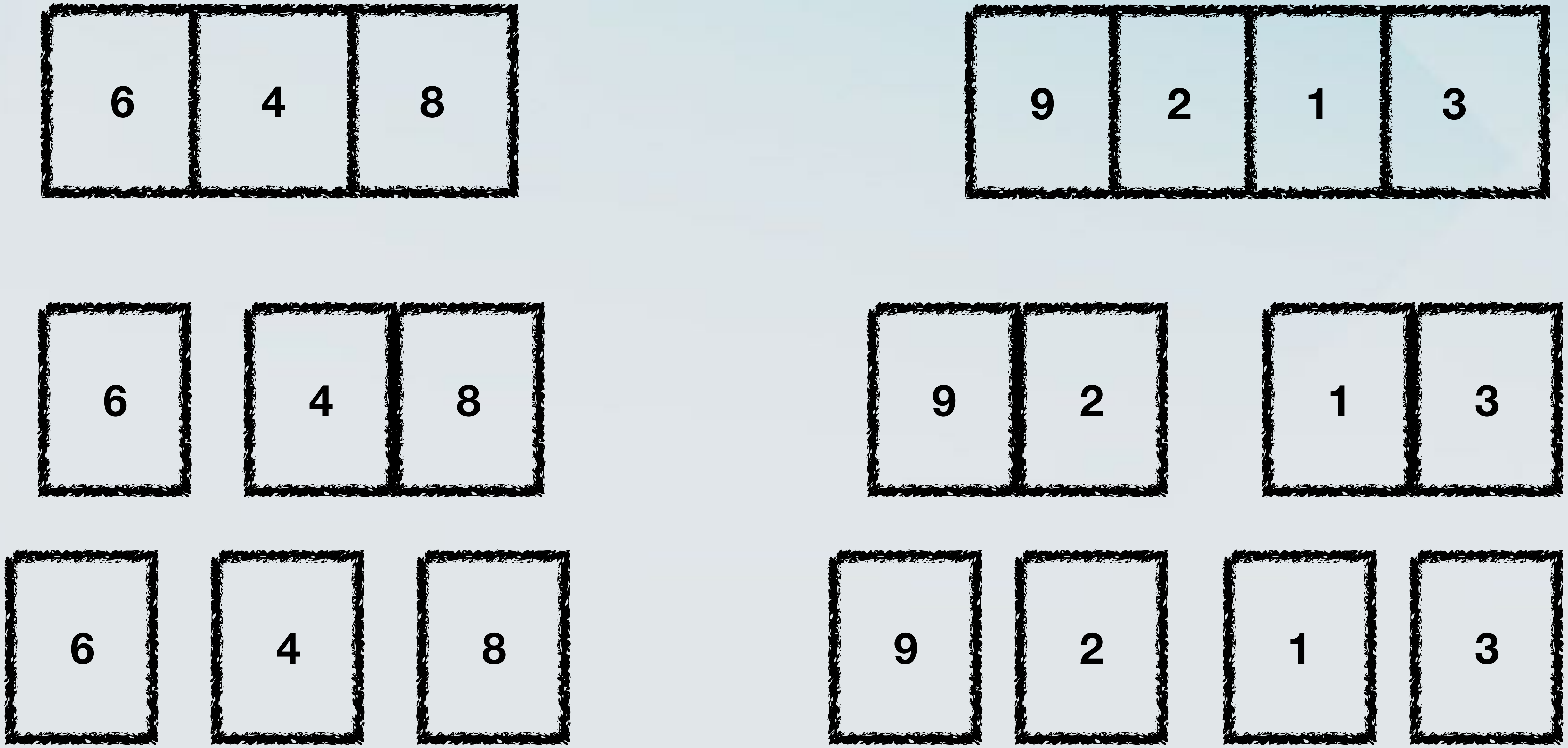
6 4 8   9 2 1 3

6 | 4 8   9 2 | 1 3

6 | 4 | 8   9 | 2 | 1 | 3

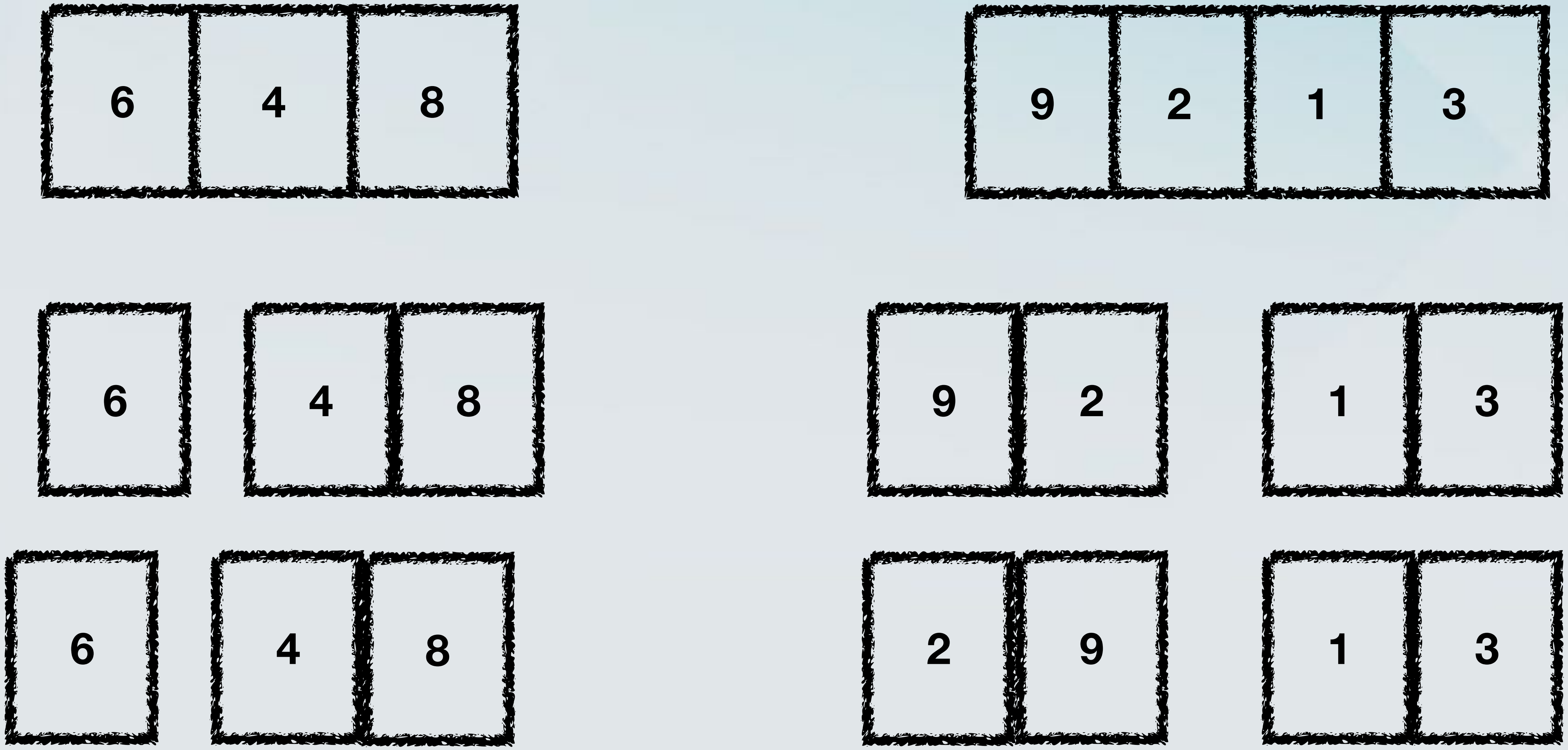
Mergesort example

divide merge



Mergesort example

divide merge



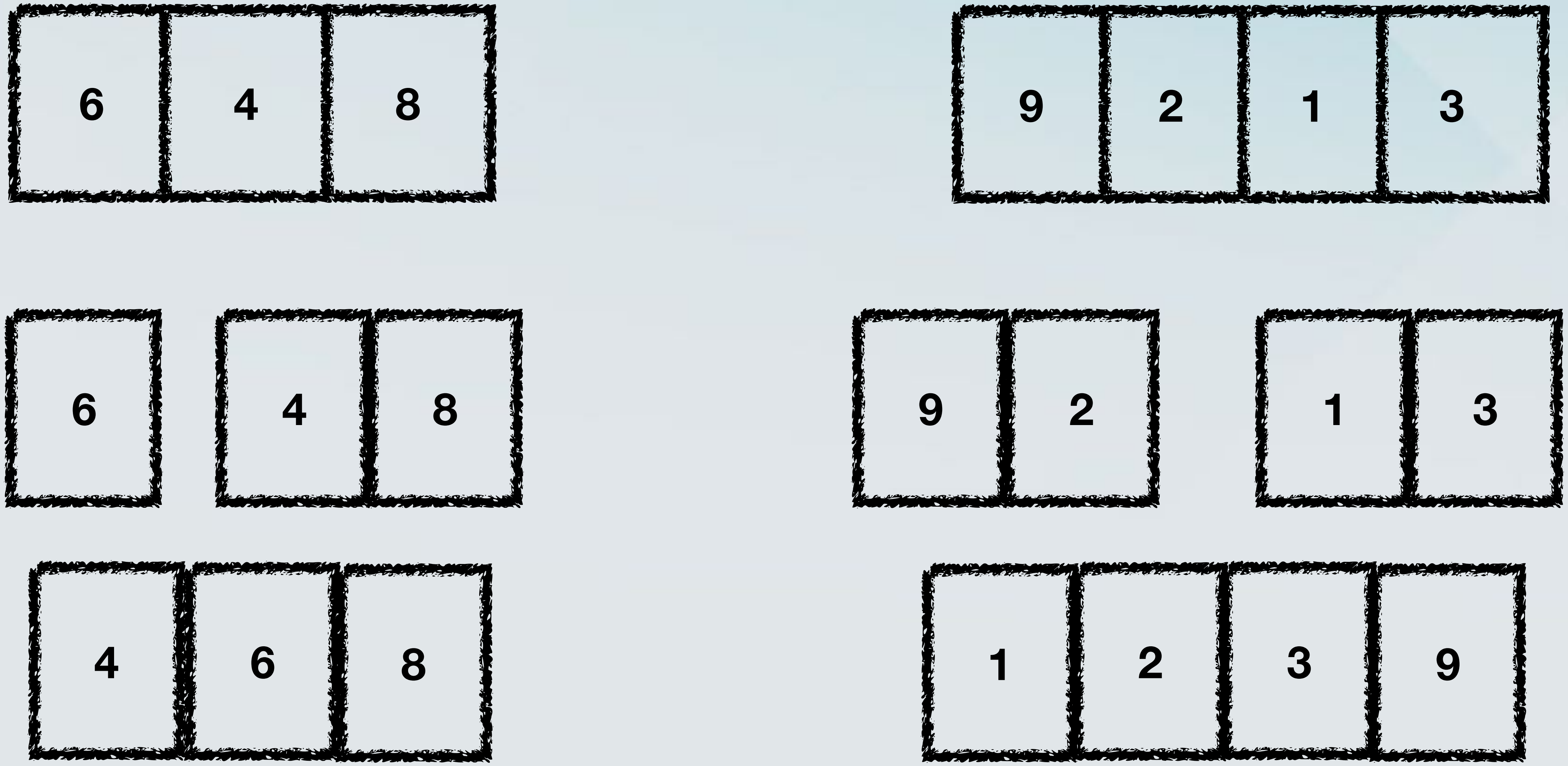
6 4 8 9 2 1 3

6 | 4 8 9 2 | 1 3

6 | 4 8 2 9 | 1 3

Mergesort example

divide merge



6	4	8

9	2	1	3

6

4	8

9	2

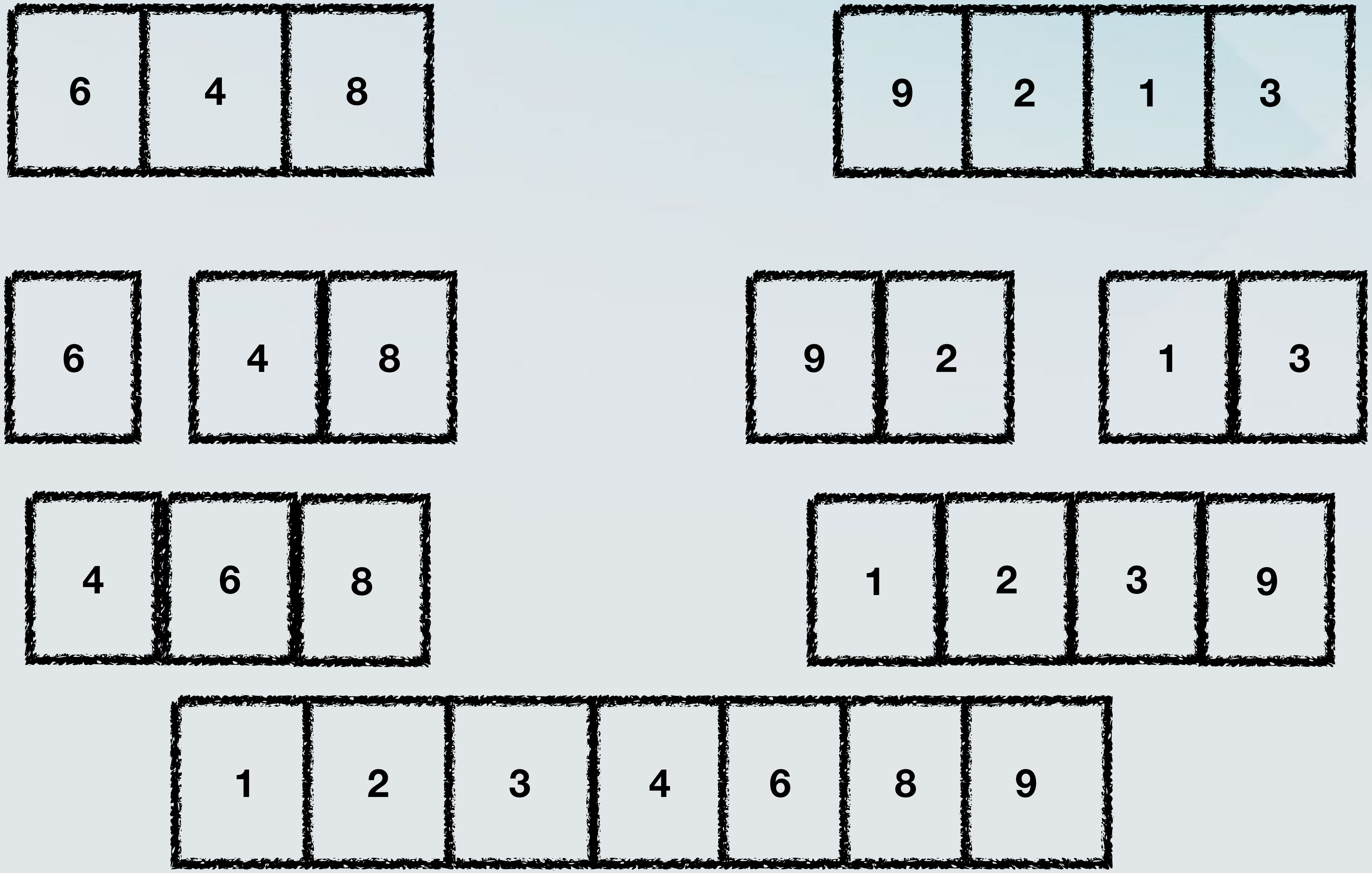
1	3

4	6	8

1	2	3	9

Mergesort example

divide merge



6 4 8   9 2 1 3

6   4 8   9 2   1 3

4 6 8   1 2 3 9

1 2 3 4 6 8 9

Running time of Mergesort

- We could guess the running time and prove it using induction, as we saw in the previous lecture.
- Instead, we will try to "figure out" what the running time should be.
- We will use the method of **recursion trees.**
- The running time is generally

$$T(n) = 2T(n/2) + f(n)$$

Running time of Mergesort

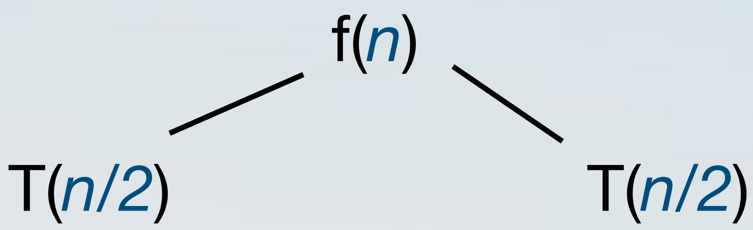
Running time of Mergesort

For simplicity, assume $n = 2^k$

Running time of Mergesort

For simplicity, assume $n = 2^k$

First iteration: Price of $f(n)$ plus the cost of two subproblems of size n/2



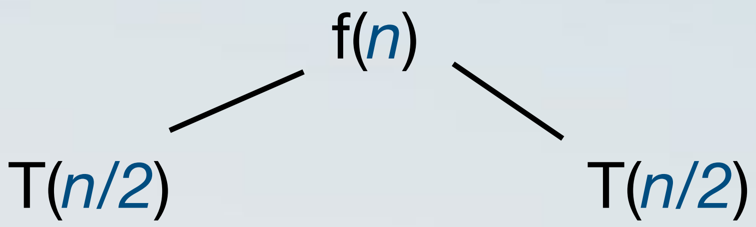
$f(n)$

$T(n/2)$ $T(n/2)$

Running time of Mergesort

For simplicity, assume $n = 2^k$

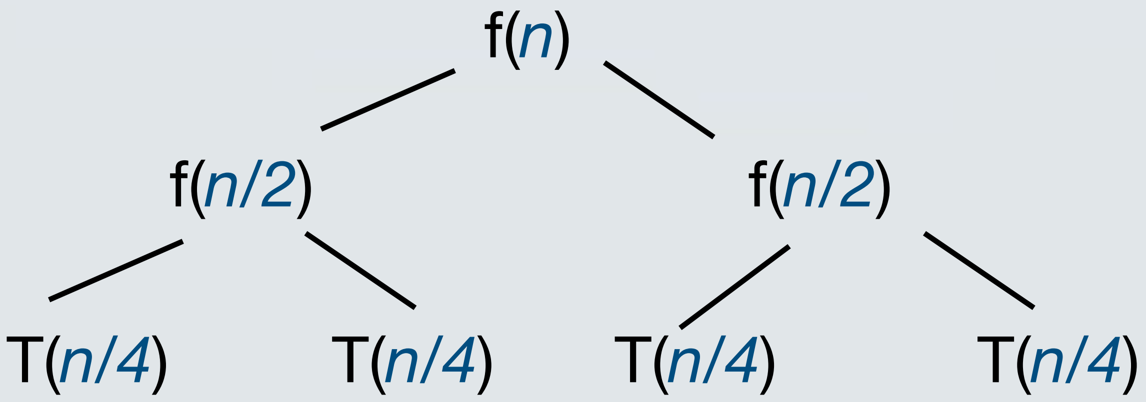
First iteration: Price of $f(n)$ plus the cost of two subproblems of size n/2



$f(n)$

$T(n/2)$ $T(n/2)$

Second iteration: Price of $f(n/2)$ for each subproblem, plus the cost of two subproblems of size n/4



$f(n)$

$f(n/2)$ $f(n/2)$

$T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

Running time of Mergesort

- In total, there will be log n + 1 levels (input halved every time).
  - Level 0 has cost $C_0(n) = f(n)$
  - Level 1 has cost $C_1(n) = 2f(n/2)$
  - Level 2 has cost $C_2(n) = 4f(n/4)$
  - Level j has cost $C_j(n) = 2^j f(n/2^j)$
  - The last level has cost $f(n)$

Running time of Mergesort

Running time of Mergesort

- Recurrence relation:

$$T(n) = \sum_{j=1}^{\log n - 1} C_j(n) + f(n)$$

Running time of Mergesort

- Recurrence relation:

$$T(n) = \sum_{j=1}^{\log n - 1} C_j(n) + f(n)$$

Running time of Mergesort

- Recurrence relation:

$$T(n) = \sum_{j=1}^{\log n - 1} C_j(n) + f(n)$$

Running time of Mergesort

- Recurrence relation:

$$T(n) = \sum_{j=1}^{\log n - 1} C_j(n) + f(n)$$

- It also holds that f(*n*) ≤ d*n* for some large enough d.

Running time of Mergesort

- Recurrence relation:

$$T(n) = \sum_{j=1}^{\log n - 1} C_j(n) + f(n)$$

- It also holds that f(n) ≤ dn for some large enough d.

- It also holds that $C_j(n) = 2^j f(n/2^j)$ is at most dn for some large enough d.

Running time of Mergesort

- Recurrence relation:

$$T(n) = \sum_{j=1}^{\log n - 1} C_j(n) + f(n)$$

- It also holds that $f(n) \leq dn$ for some large enough d.

- It also holds that $C_j(n) = 2^j f(n/2^j)$ is at most dn for some large enough d.

- The overall running time is **O(n log n)**.

Sorting with Quicksort

The Quicksort algorithm

The **Quicksort** algorithm

- **Mergesort** was based on the **Merge** procedure for joining the sorted sub-arrays into a sorted array.

The Quicksort algorithm

- **Mergesort** was based on the **Merge** procedure for joining the sorted sub-arrays into a sorted array.
- **Quicksort** first divides the array into two parts, such that the first part is “smaller” than the second part.

The Quicksort algorithm

- **Mergesort** was based on the **Merge** procedure for joining the sorted sub-arrays into a sorted array.
- **Quicksort** first divides the array into two parts, such that the first part is “smaller” than the second part.
  - This is done via the **Partition** procedure.

The Quicksort algorithm

- **Mergesort** was based on the **Merge** procedure for joining the sorted sub-arrays into a sorted array.

- **Quicksort** first divides the array into two parts, such that the first part is “smaller” than the second part.

    - This is done via the **Partition** procedure.

- Then it calls itself recursively.

The Quicksort algorithm

- **Mergesort** was based on the **Merge** procedure for joining the sorted sub-arrays into a sorted array.

- **Quicksort** first divides the array into two parts, such that the first part is “smaller” than the second part.

    - This is done via the **Partition** procedure.

- Then it calls itself recursively.

- The two parts are joined, but this is trivial.

The Partition procedure

```
Procedure Partition(A[$i,\dots,j$])

    Choose a pivot element x of A

    $k = i-1$

    For $h = i$ to $j-1$ do

        If A[$h$] $\leq$ x
            $k = k + 1$
            Swap A[$k$] with A[$h$]

    Swap A[$k+1$] with A[$j$]

Return $k+1$
```

The Partition procedure

```
Procedure Partition(A[$i,\ldots,j$])

    Choose a pivot element x of A

    $k = i-1$

    For $h = i$ to $j-1$ do

        If A[$h$] ≤ x
            $k = k + 1$
            Swap A[$k$] with A[$h$]

    Swap A[$k+1$] with A[$j$]

Return $k+1$
```

Correctness of **Partition**:

(CLRS p. 171-173)

The Partition procedure

```
Procedure Partition(A[$i$,…,$j$])

    Choose a pivot element x of A

    $k = i-1$

    For $h = i$ to $j-1$ do

        If A[$h$] $\leq$ x
            $k = k + 1$
            Swap A[$k$] with A[$h$]

    Swap A[$k+1$] with A[$j$]

Return $k+1$
```

Correctness of **Partition**:

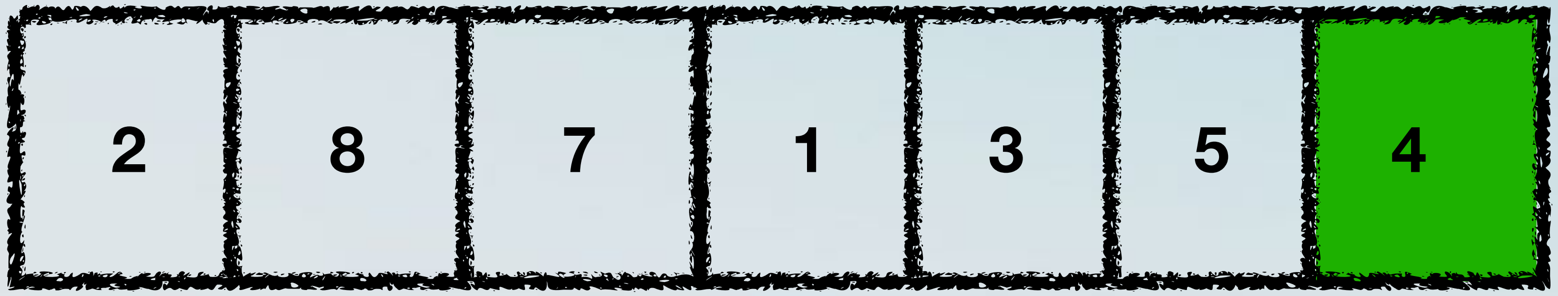
(CLRS p. 171-173)

Running time **O(n)**

The Quicksort algorithm

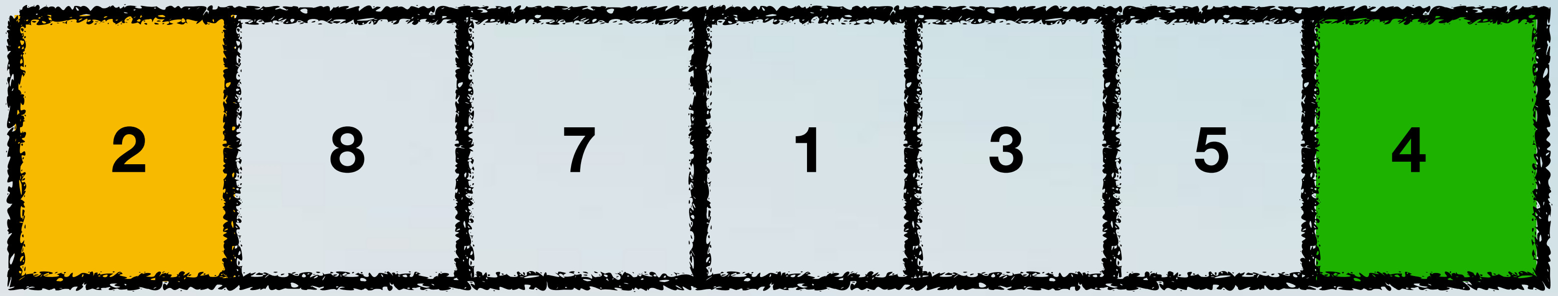
2	8	7	1	3	5	4

The Quicksort algorithm



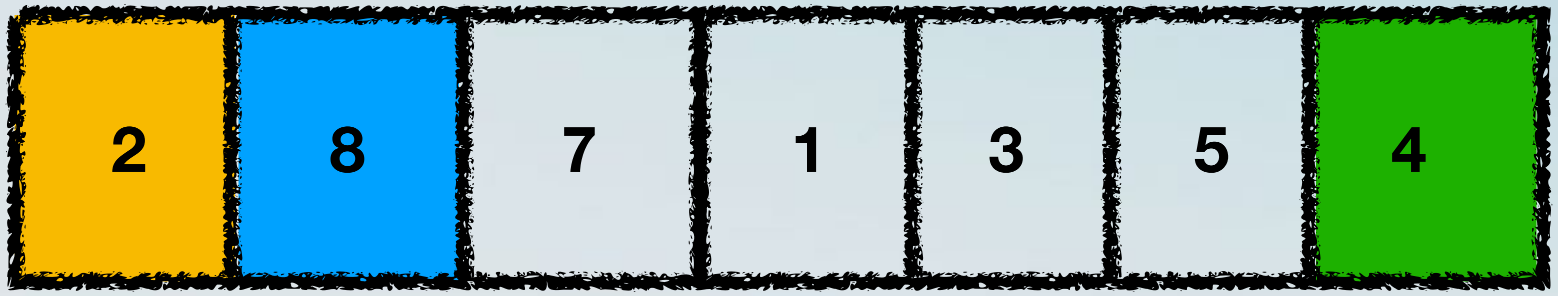
2	8	7	1	3	5	4

The Quicksort algorithm



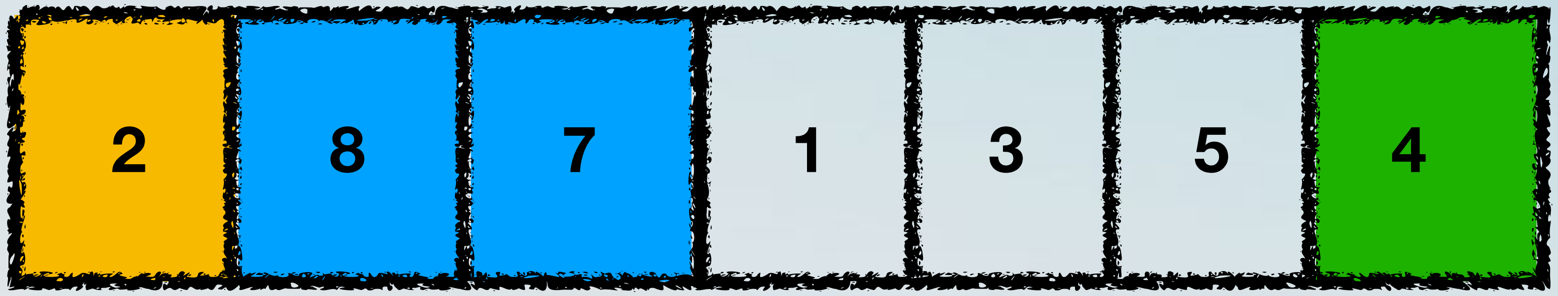
2	8	7	1	3	5	4

The Quicksort algorithm



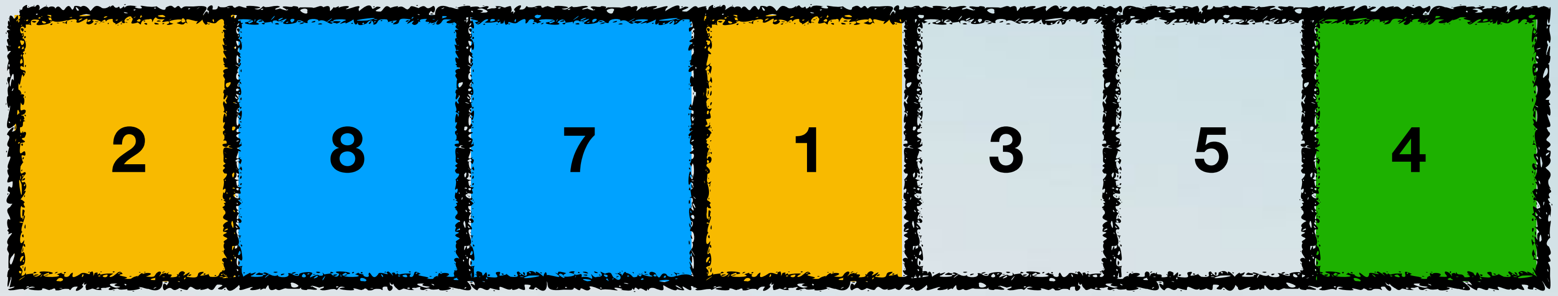
2	8	7	1	3	5	4

The Quicksort algorithm



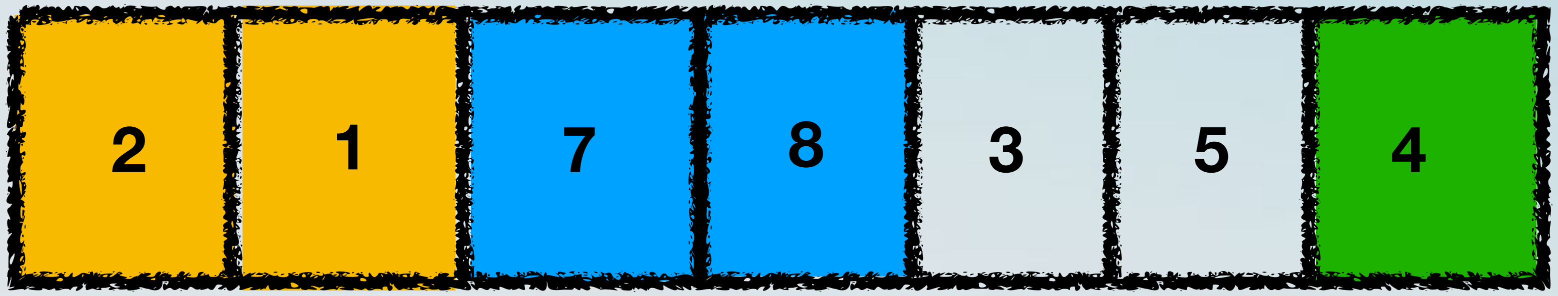
2	8	7	1	3	5	4

The Quicksort algorithm



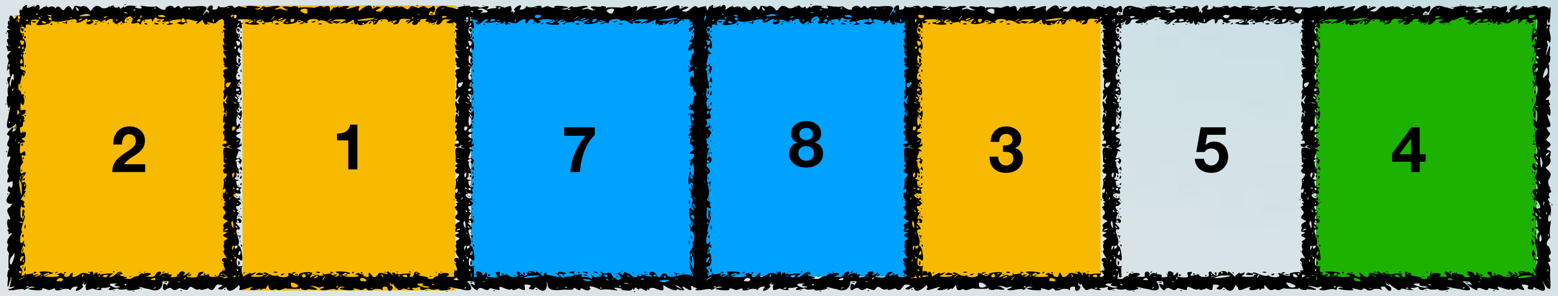
2	8	7	1	3	5	4

The Quicksort algorithm



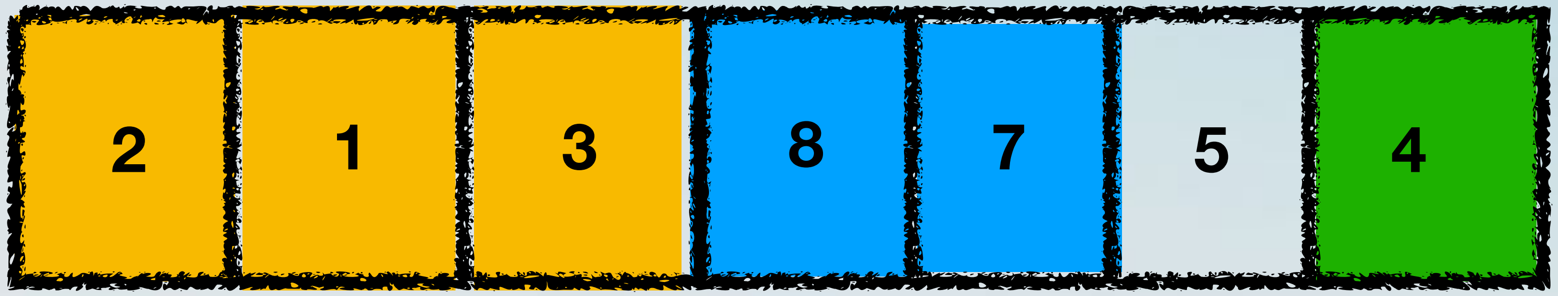
2	1	7	8	3	5	4

The Quicksort algorithm



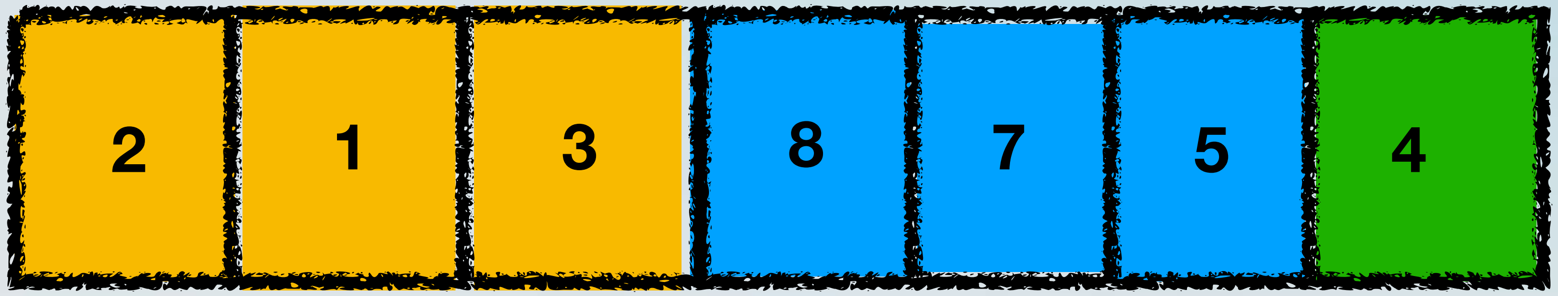
2	1	7	8	3	5	4

The Quicksort algorithm



2	1	3	8	7	5	4

The Quicksort algorithm



2	1	3	8	7	5	4

The Quicksort algorithm

2	1	3	4	7	5	8

The **Quicksort** algorithm

2	1	3	4	7	5	8

Sort this using
Quicksort

The **Quicksort** algorithm

2	1	3	4	7	5	8

Sort this using
Quicksort

Sort this using
Quicksort

The Quicksort algorithm

Sort this using **Quicksort**

Sort this using **Quicksort**

```
Algorithm Quicksort(A[$i,\ldots,j$])

        $y$ = Partition(A[$i,\ldots,j$])
              Quicksort(A[$i,\ldots,y-1$])
              Quicksort(A[$y+1,\ldots,j$])
```

Running time of **Quicksort**

- Can it be as fast as **Mergesort**?
- Can it be slower than **Mergesort**?
- Can it be faster than **Mergesort**?

Running time of **Quicksort**

- Can it be as fast as **Mergesort**?
- Can it be slower than **Mergesort**?
- Can it be faster than **Mergesort**?

Running time of **Quicksort**

- Can it be as fast as **Mergesort**?
- Can it be slower than **Mergesort**?
- Can it be faster than **Mergesort**?

- This will depend on the pivot element!

Running time of Quicksort

Running time of Quicksort

Mergesort: $T(n) \leq 2T(n/2) + cn$

Running time of **Quicksort**

Mergesort: $T(n) \leq 2T(n/2) + cn$

Quicksort: $T(n) \leq T(n_1) + T(n_2) + cn$

Running time of Quicksort

Mergesort: $T(n) \leq 2T(n/2) + cn$

Quicksort: $T(n) \leq T(n_1) + T(n_2) + cn$

When $n_1 = n_2$, the running time is the same as **Mergesort.**

Running time of Quicksort

Mergesort: $T(n) \leq 2T(n/2) + cn$

Quicksort: $T(n) \leq T(n_1) + T(n_2) + cn$

When $n_1 = n_2$, the running time is the same as **Mergesort.**

What is the worst possible running time?

Running time of Quicksort

Running time of Quicksort

- Consider the case where we have an unbalanced partitioning in every step.

  $n_1 = n-1$
  $n_2 = 0$

Running time of Quicksort

- Consider the case where we have an unbalanced partitioning in every step.

  $n_1 = n-1$
  $n_2 = 0$

- We get $T(n) = T(n-1) + cn$

Running time of Quicksort

- Consider the case where we have an unbalanced partitioning in every step.

  $n_1 = n-1$
  $n_2 = 0$

- We get $T(n) = T(n-1) + cn$

- What is the solution to this recurrence?

Running time of Quicksort

- Consider the case where we have an unbalanced partitioning in every step.

  $n_1 = n-1$
  $n_2 = 0$

- We get $T(n) = T(n-1) + cn$

- What is the solution to this recurrence?

  - $T(n) = \Theta(n^2)$

Running time of Quicksort

Running time of **Quicksort**

- Can it be as fast as **Mergesort**? **Yes**

Running time of **Quicksort**

- Can it be as fast as **Mergesort**? **Yes**

- Can it be slower than **Mergesort**? **Yes**

Running time of **Quicksort**

- Can it be as fast as **Mergesort**? **Yes**
- Can it be slower than **Mergesort**? **Yes**
- Can it be faster than **Mergesort**? **??**

Lower bound for (comparison-based) sorting

Lower bound for (comparison-based) sorting

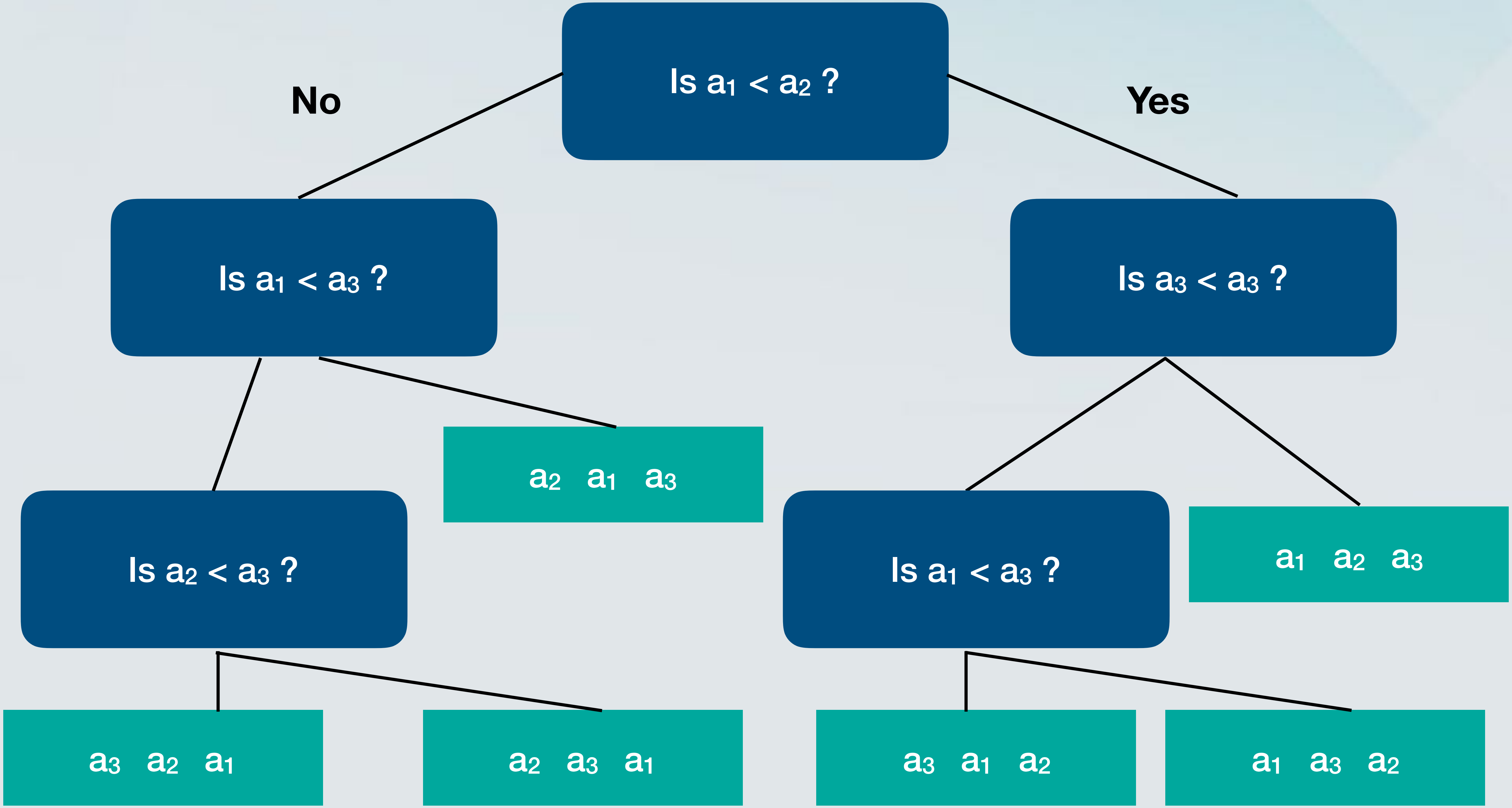
- We will prove that no algorithm that is based on performing comparisons between the elements of the array can not run in time *asymptotically* faster than n log n

Lower bound for (comparison-based) sorting

- We will prove that no algorithm that is based on performing comparisons between the elements of the array can not run in time *asymptotically* faster than n log n

- In other words, we will prove that for any such algorithm, the running time is **Ω(n log n).**

Lower bound for (comparison-based) sorting



Is $a_1 < a_2$?

No

Yes

Is $a_1 < a_3$?

Is $a_3 < a_3$?

a_2 a_1 a_3

Is $a_2 < a_3$?

Is $a_1 < a_3$?

a_1 a_2 a_3

a_3 a_2 a_1

a_2 a_3 a_1

a_3 a_1 a_2

a_1 a_3 a_2

Lower bound for (comparison-based) sorting

Lower bound for (comparison-based) sorting

- We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).

Lower bound for (comparison-based) sorting

- We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).
- The decision tree has n! leaves

Lower bound for (comparison-based) sorting

- We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).
- The decision tree has n! leaves
  - A leaf is a permutation of $\{a_1, a_2, \dots , a_n\}$

Lower bound for (comparison-based) sorting

- We need as many comparisons as the *depth* of the tree (length of the longest path from the root to the leaves).
- The decision tree has n! leaves
  - A leaf is a permutation of $\{a_1, a_2, \dots, a_n\}$
  - Every possible permutation can appear as a leaf, since every possible permutation is a valid output.

Lower bound for (comparison-based) sorting

Lower bound for (comparison-based) sorting

- Fact: Every binary tree of depth d has at most 2^d leaves.

Lower bound for (comparison-based) sorting

- Fact: Every binary tree of depth d has at most 2^d leaves.
- Therefore the minimum number of comparisons is $\log(n!)$

Lower bound for (comparison-based) sorting

- Fact: Every binary tree of depth d has at most 2^d leaves.
- Therefore the minimum number of comparisons is $\log(n!)$
- We claim that $\log(n!) = \Omega(n \log n)$

Lower bound for (comparison-based) sorting

- Fact: Every binary tree of depth d has at most 2^d leaves.
- Therefore the minimum number of comparisons is $\log(n!)$
- We claim that $\log(n!) = \Omega(n \log n)$
  - Why is that the case? Ideas?

Lower bound for (comparison-based) sorting

- Fact: Every binary tree of depth d has at most 2^d leaves.
- Therefore the minimum number of comparisons is $\log(n!)$
- We claim that $\log(n!) = \Omega(n \log n)$
  - Why is that the case? Ideas?

$$
\begin{aligned}
\log(n!) = \log(1 * 2 * \ldots * n) &= \log(1) + \log(2) + \ldots + \log(n) \\
&\geq \log(n/2) + \ldots + \log(n) \text{ (half)} \\
&\geq \log(n/2) + \ldots + \log(n/2) \\
&= (n/2) * \log(n/2)
\end{aligned}
$$

Proving lower bounds

Proving lower bounds

- Consider some criterion A that we would like to minimise (could be running time, memory, etc).

Proving lower bounds

- Consider some criterion A that we would like to minimise (could be running time, memory, etc).
- We want to find the best algorithm (asymptotically) for criterion A.

Proving lower bounds

- Consider some criterion A that we would like to minimise (could be running time, memory, etc).
- We want to find the best algorithm (asymptotically) for criterion A.
- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

Proving lower bounds

- Consider some criterion A that we would like to minimise (could be running time, memory, etc).
- We want to find the best algorithm (asymptotically) for criterion A.
- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.
- **Upper bound:** We construct an algorithm that has performance $O(g(n))$ for criterion A.

Proving lower bounds

- Consider some criterion A that we would like to minimise (could be running time, memory, etc).
- We want to find the best algorithm (asymptotically) for criterion A.
- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.
- **Upper bound:** We construct an algorithm that has performance $O(g(n))$ for criterion A.
- **Lower bound:** We show that for any algorithm, the performance for criterion A is $\Omega(g(n))$.

Proving lower bounds

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.
- How do we find this function?

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.
- How do we find this function?
  - No easy answer!

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.
- How do we find this function?
  - No easy answer!
  - We try to design algorithms which are as good as possible and when we feel that we can not improve more, we try to prove the matching lower bound.

Proving lower bounds

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

- We want to find the best algorithm **(asymptotically)** for criterion A.

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.
- We want to find the best algorithm **(asymptotically)** for criterion A.
- This is only true if $g(n)$ is not a constant function.

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.
- We want to find the best algorithm **(asymptotically)** for criterion A.
- This is only true if $g(n)$ is not a constant function.
  - This will be more relevant when we talk about approximation algorithms and online algorithms.

Proving lower bounds

- The best possible achievable performance is $\Theta(g(n))$ for some function $g(n)$.

- We want to find the best algorithm **(asymptotically)** for criterion A.

- This is only true if $g(n)$ is not a constant function.

    - This will be more relevant when we talk about approximation algorithms and online algorithms.

    - Stay tuned!