Algorithms and Data Structures

Asymptotic Notation and Divide and Conquer Fundamentals

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j $\leftarrow$ 2 **TO** length[A] **n times**
    DO key $\leftarrow A[j]$ **n-1 times**
        {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
        $i \leftarrow j-1$ **n-1 times**
        WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$ $\sum_{j=2}^{n}(t_j-1)$ **times**
        $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case? **Sorted array,** $t_j = 1$ Bounded by some cn for some constant c

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some cn^2 for some constant c

Asymptotic Notation

- When n becomes large, it makes less of a difference if an algorithm takes $2n$ or $3n$ steps to finish.
- In particular, $3\lg n$ steps are fewer than $2n$ steps.
- We would like to avoid having to calculate the precise constants.
- We use asymptotic notation.

Asymptotic Notation

O-notation. $O(g(n)) = f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Ω-notation. $\Omega(g(n)) = f(n)$: there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$.

Θ-notation. $\Theta(g(n)) = f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.

Asymptotic Notation

O-notation. $O(g(n)) = f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

"The rate of growth of $f(n)$ is at most that of $g(n)$."

Asymptotic Notation

O-notation. $O(g(n)) = f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

For sufficiently large inputs, there is a constant such that $c \cdot g(n)$ is not smaller than $f(n)$.

For example, for sufficiently large inputs, $2n$ is larger than $3 \lg n$. Therefore, $3 \lg n = O(2n) = O(n)$.

Use: If we can upper bound the running time of an algorithm by $c \cdot g(n)$, where c is some constant and $g(\cdot)$ is a function of the input, then we can say that the running time is $O(g(n))$.

Asymptotic Notation

Ω-notation. $\Omega(g(n)) = f(n)$: there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$.

"The rate of growth of $f(n)$ is at least that of $g(n)$."

Θ-notation. $\Theta(g(n)) = f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$. "The rate of growth of $f(n)$ is at most that of $g(n)$."

"The rate of growth of $f(n)$ is the same as that of $g(n)$."

Little-O, Little-Omega

o-notation. $o(g(n)) = f(n)$: for any constant c, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$.

"The rate of growth of $f(n)$ is smaller than that of $g(n)$."

ω-notation. $\omega(g(n)) = f(n)$: for any constant c, there exists a constant $n_0 > 0$ such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$.

"The rate of growth of $f(n)$ is larger than that of $g(n)$."

Little-O

o-notation. $o(g(n)) = f(n)$: for any constant c, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Equivalent (but less formal) definition: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

As n approaches infinity, $f(n)$ becomes insignificant compared to $g(n)$.

Example: $2n = o(n^2)$.

Little-Omega

ω-notation. $\omega(g(n)) = f(n)$: for any constant c, there exists a constant $n_0 > 0$ such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$.

Equivalent (but less formal) definition: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

As n approaches infinity, $g(n)$ becomes insignificant compared to $f(n)$.

Example: $4n^2 = \omega(n)$.

Examples

Examples

$5n^3 + 100 = O(n^3)$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

$$5n^3 + 100 = o(n^4)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

$$5n^3 + 100 = o(n^4)$$

$$5n^3 + 100 = \Omega(n^2)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

$$5n^3 + 100 = o(n^4)$$

$$5n^3 + 100 = \Omega(n^2)$$

$$5n^3 + 100 = \omega(n^2)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

$$5n^3 + 100 = o(n^4)$$

$$5n^3 + 100 = \Omega(n^2)$$

$$5n^3 + 100 = \omega(n^2)$$

$$\lg n = o(n^5)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

$$5n^3 + 100 = o(n^4)$$

$$5n^3 + 100 = \Omega(n^2)$$

$$5n^3 + 100 = \omega(n^2)$$

$$\lg n = o(n^5)$$

$$n^5 = o(2^n)$$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

$$5n^3 + 100 = o(n^4)$$

$$5n^3 + 100 = \Omega(n^2)$$

$$5n^3 + 100 = \omega(n^2)$$

$$\lg n = o(n^5)$$

$$n^5 = o(2^n)$$

$$\lg(4n) = \lg n + \lg 4 = O(\lg n)$$

Examples

$5n^3 + 100 = O(n^3)$

$5n^3 + 100 = \Omega(n^3)$

$5n^3 + 100 = \Theta(n^3)$

$5n^3 + 100 = O(n^4)$

$5n^3 + 100 = o(n^4)$

$5n^3 + 100 = \Omega(n^2)$

$5n^3 + 100 = \omega(n^2)$

$\lg n = o(n^5)$

$n^5 = o(2^n)$

$\lg(4n) = \lg n + \lg 4 = O(\lg n)$

$\lg(n^4) = 4 \lg n = O(\lg n)$

Examples

$$5n^3 + 100 = O(n^3)$$

$$5n^3 + 100 = \Omega(n^3)$$

$$5n^3 + 100 = \Theta(n^3)$$

$$5n^3 + 100 = O(n^4)$$

$$5n^3 + 100 = o(n^4)$$

$$5n^3 + 100 = \Omega(n^2)$$

$$5n^3 + 100 = \omega(n^2)$$

$$\lg n = o(n^5)$$

$$n^5 = o(2^n)$$

$$\lg(4n) = \lg n + \lg 4 = O(\lg n)$$

$$\lg(n^4) = 4 \lg n = O(\lg n)$$

$$(4n)^3 = 64n^3 = \Theta(n^3)$$

In class quiz

Running time hierarchy

$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^\alpha)$	$O(c^n)$
logarithmic	linear		quadratic	polynomial	exponential
The algorithm does not even read the whole input.	The algorithm accesses the input only a constant number of times.	The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions.	The algorithm considers pairs of elements.	The algorithm performs many nested loops.	The algorithm considers many subsets of the input elements.

constant	$O(1)$	superlinear	$\omega(n)$
superconstant	$\omega(1)$	superpolynomial	$\omega(n^\alpha)$
sublinear	$o(n)$	subexponential	$o(c^n)$

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j ← 2 **TO** length[A] **n times**
  DO key ← $A[j]$ **n-1 times**
    {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
    $i \leftarrow j-1$ **n-1 times**
    WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
      DO $A[i+1] \leftarrow A[i]$
        $i \leftarrow i-1$ $\sum_{j=2}^{n} (t_j - 1)$ **times**
    $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some cn^2 for some constant c

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR $j \leftarrow 2$ **TO** length[A] **n times**
    DO key $\leftarrow A[j]$ **n-1 times**
        {Put $A[j]$ into the sorted sequence $A[1 \,.\,.\, j-1]$}
        $i \leftarrow j - 1$ **n-1 times**
        WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i - 1$ $\sum_{j=2}^{n} (t_j - 1)$ **times**
        $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Example: Running Time of InsertionSort

```
INSERTION_SORT (A)

1.    FOR j ← 2 TO length[A]                      n times
2.         DO  key ← A[j]                         n-1 times
3.             {Put A[j] into the sorted sequence A[1 . . j − 1]}
4.             i ← j − 1                          n-1 times
5.             WHILE i > 0 and A[i] > key         ∑_{j=2}^{n} t_j  times
6.                   DO A[i +1] ← A[i]            ∑_{j=2}^{n} (t_j − 1)  times
7.                      i ← i − 1
8.             A[i + 1] ← key                     n-1 times
```

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

A bit more formally:

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j $\leftarrow$ 2 **TO** length[A] **n times**
  DO key $\leftarrow A[j]$ **n-1 times**
    {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
    $i \leftarrow j-1$ **n-1 times**
    WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
      DO $A[i+1] \leftarrow A[i]$
        $i \leftarrow i-1$
    $\sum_{j=2}^{n} (t_j - 1)$ **times**
    $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

A bit more formally: How large can t_j be in the worst case?

Example: Running Time of InsertionSort

```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]                         n times
      DO key ← A[j]                             n-1 times
         {Put A[j] into the sorted sequence A[1 . . j − 1]}
         i ← j − 1                              n-1 times
         WHILE i > 0 and A[i] > key             Σ_{j=2}^{n} t_j times
                DO A[i +1] ← A[i]               Σ_{j=2}^{n} (t_j − 1) times
                   i ← i − 1
         A[i + 1] ← key                         n-1 times
```

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

A bit more formally: How large can t_j be in the worst case?

The loop can run at most i times, and $i \leq j$

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j $\leftarrow$ 2 **TO** length[A] **n times**
  DO key $\leftarrow A[j]$ **n-1 times**
    {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
    $i \leftarrow j - 1$ **n-1 times**
    WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
      DO $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$ $\sum_{j=2}^{n} (t_j - 1)$ **times**
    $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

A bit more formally: How large can t_j be in the worst case?

The loop can run at most i times, and $i \leq j$

This means that $t_j \leq j$

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j $\leftarrow$ 2 **TO** length[A] **n times**
    DO key $\leftarrow A[j]$ **n-1 times**
        {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
        $i \leftarrow j-1$ **n-1 times**
        WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$
$\sum_{j=2}^{n}(t_j-1)$ **times**
        $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j-1) + c_6 \sum_{j=2}^{n}(t_j-1) + c_7(n-1)$$

This means that $t_j \le j$

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j $\leftarrow$ 2 **TO** length$[A]$ **n times**
  DO key $\leftarrow A[j]$ **n-1 times**
   $\{$Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]\}$
   $i \leftarrow j-1$ **n-1 times**
   WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
     DO $A[i+1] \leftarrow A[i]$
      $i \leftarrow i-1$ $\sum_{j=2}^{n} (t_j - 1)$ **times**
   $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

This means that $t_j \leq j$

$$T(n) \leq c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7(n-1)$$

Example: Running Time of InsertionSort

```
INSERTION_SORT (A)

FOR j ← 2 TO length[A]   n times
    DO key ← A[j]   n-1 times
       {Put A[j] into the sorted sequence A[1 . . j − 1]}
       i ← j − 1   n-1 times
       WHILE i > 0 and A[i] > key   Σ_{j=2}^{n} t_j times
             DO A[i +1] ← A[i]
                i ← i − 1          Σ_{j=2}^{n} (t_j − 1) times
       A[i + 1] ← key   n-1 times
```

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

This means that $t_j \leq j$

$$T(n) \leq c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7(n-1)$$

$$T(n) \leq C \cdot n + C' \cdot \frac{n(n+1)}{2} = O(n^2)$$

Upper Bounds

Upper Bounds

We proved that on any possible input, **InsertionSort** takes time $O(n^2)$.

Upper Bounds

We proved that on any possible input, **InsertionSort** takes time $O(n^2)$.

This is an **upper bound**, because the running time cannot be more than this (asymptotically).

Upper Bounds

We proved that on any possible input, **InsertionSort** takes time $O(n^2)$.

This is an **upper bound**, because the running time cannot be more than this (asymptotically).

Sometimes we can be happy and stop there.

Upper Bounds

We proved that on any possible input, **InsertionSort** takes time $O(n^2)$.

This is an **upper bound**, because the running time cannot be more than this (asymptotically).

Sometimes we can be happy and stop there.

But what if our analysis was very “loose”?

Upper Bounds

We proved that on any possible input, **InsertionSort** takes time $O(n^2)$.

This is an **upper bound**, because the running time cannot be more than this (asymptotically).

Sometimes we can be happy and stop there.

But what if our analysis was very "loose"?

We bounded $t_j \leq j$. Is this possible for this to happen or are we being too "generous"?

Upper and Lower (Worst-Case) Bounds

Upper and Lower (Worst-Case) Bounds

Upper Bound $O(g_1(n))$: On *any possible input* to the problem, our algorithm will take time (at most) $O(g_1(n))$.

Upper and Lower (Worst-Case) Bounds

Upper Bound $O(g_1(n))$: On *any possible input* to the problem, our algorithm will take time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: There *exists at least one input* to the problem, on which our algorithm will take time (at least) $\Omega(g_2(n))$.

Upper and Lower (Worst-Case) Bounds

Upper Bound $O(g_1(n))$: On *any possible input* to the problem, our algorithm will take time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: There *exists at least one input* to the problem, on which our algorithm will take time (at least) $\Omega(g_2(n))$.

When $g_1(n) = g_2(n)$, we say that our running time analysis is *tight*, and we have fully understood the (asymptotic, worst-case) running time of the algorithm.

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j ← 2 TO length[A] **n times**
DO key ← $A[j]$ **n-1 times**
{Put $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$}
$i \leftarrow j-1$ **n-1 times**
WHILE $i > 0$ and $A[i]$ > key $\sum_{j=2}^{n} t_j$ **times**
DO $A[i+1] \leftarrow A[i]$
$i \leftarrow i-1$
$\sum_{j=2}^{n} (t_j - 1)$ **times**
$A[i+1]$ ← key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some cn^2 for some constant c

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j $\leftarrow$ 2 **TO** length[A] **n times**
    DO key $\leftarrow A[j]$ **n-1 times**
        {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
        $i \leftarrow j-1$ **n-1 times**
        WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$ $\sum_{j=2}^{n} (t_j - 1)$ **times**
        $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some cn^2 for some constant c

To show the lower bound, we construct explicitly a reverse sorted array (choosing numbers) and explain how the algorithm will make *j* comparisons in each step *j*.

Example: Running Time of InsertionSort

INSERTION_SORT (A)

FOR j ← 2 TO length[A] **n times**
    DO key ← $A[j]$ **n-1 times**
        {Put $A[j]$ into the sorted sequence $A[1\,..\,j-1]$}
        $i \leftarrow j-1$ **n-1 times**
        WHILE $i > 0$ and $A[i] >$ key $\sum_{j=2}^{n} t_j$ **times**
            DO $A[i+1] \leftarrow A[i]$
                $i \leftarrow i-1$ $\sum_{j=2}^{n}(t_j-1)$ **times**
        $A[i+1] \leftarrow$ key **n-1 times**

for loops, the tests are executed one more time than the loop body

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$$

Worst case? **Reverse sorted array,** $t_j = j$ Bounded by some cn^2 for some constant c

To show the lower bound, we construct explicitly a reverse sorted array (choosing numbers) and explain how the algorithm will make j comparisons in each step j.

Try it at home!

Upper and Lower (Worst-Case) Bounds

Upper Bound $O(g_1(n))$: On *any possible input* to the problem, our algorithm will take time (at most) $O(g_1(n))$.

Lower Bound $\Omega(g_2(n))$: There *exists at least one input* to the problem, on which our algorithm will take time (at least) $\Omega(g_2(n))$.

When $g_1(n) = g_2(n)$, we say that our running time analysis is *tight*, and we have fully understood the (asymptotic, worst-case) running time of the algorithm.

Introduction to Divide and Conquer

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**.

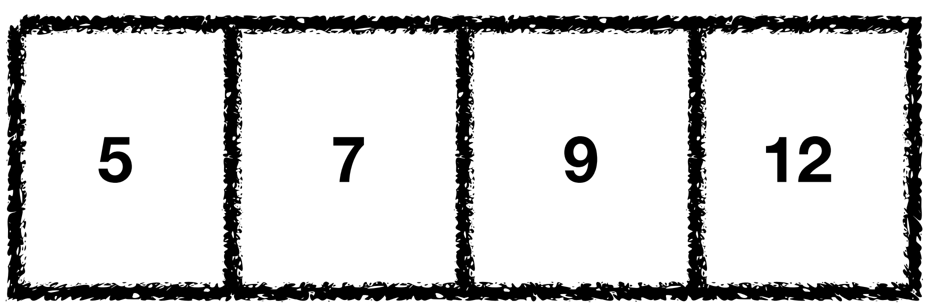
Merging two sorted arrays

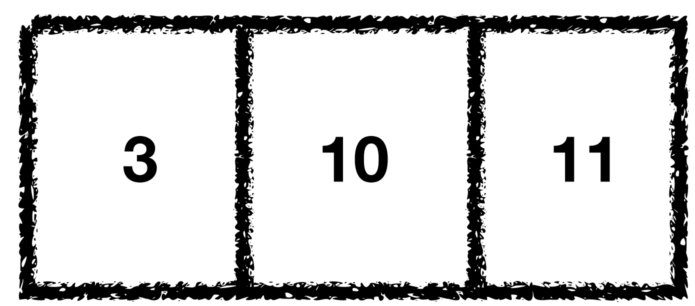
Given two sorted arrays **A**[*1,…,n*] and **B**[*1,…,m*], produce a sorted array **C**[*1, …, n+m*] containing all the elements of **A** and **B**.

5	7	9	12

Merging two sorted arrays

Given two sorted arrays **A**[$1,\ldots,n$] 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

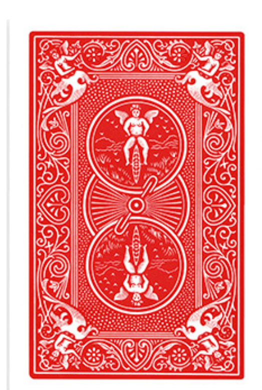
Given two sorted arrays **A**[*1,…,n*] and **B**[*1,…,m*], produce a sorted array **C**[*1, …, n+m*] containing all the elements of **A** and **B**.

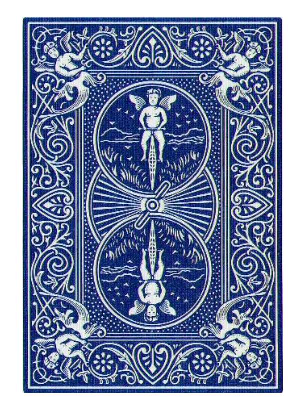
5	7	9	12

3	10	11

3	5	7	9	10	11	12

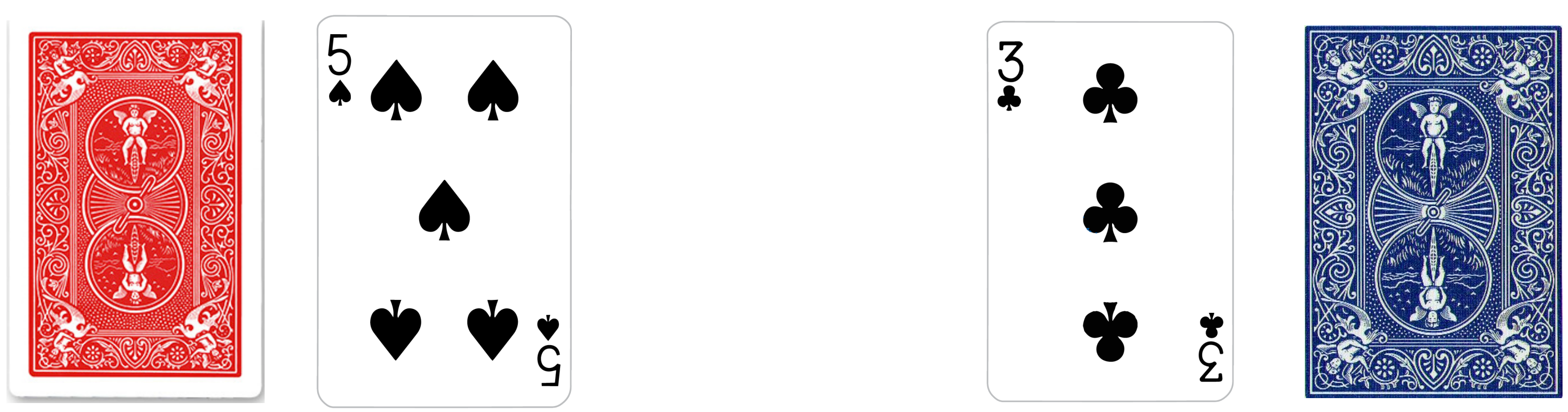
How would you do this?



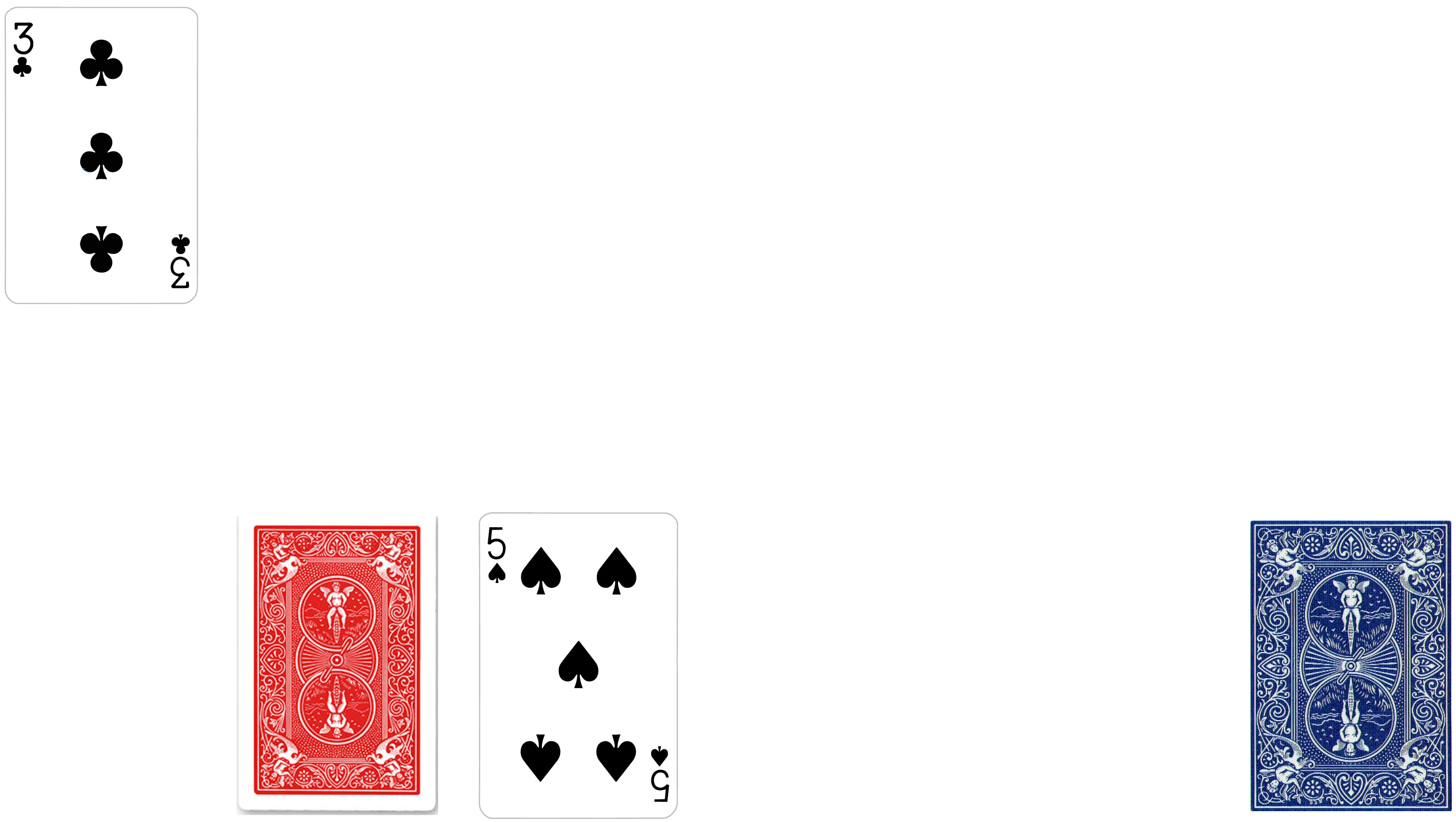


How would you do this?

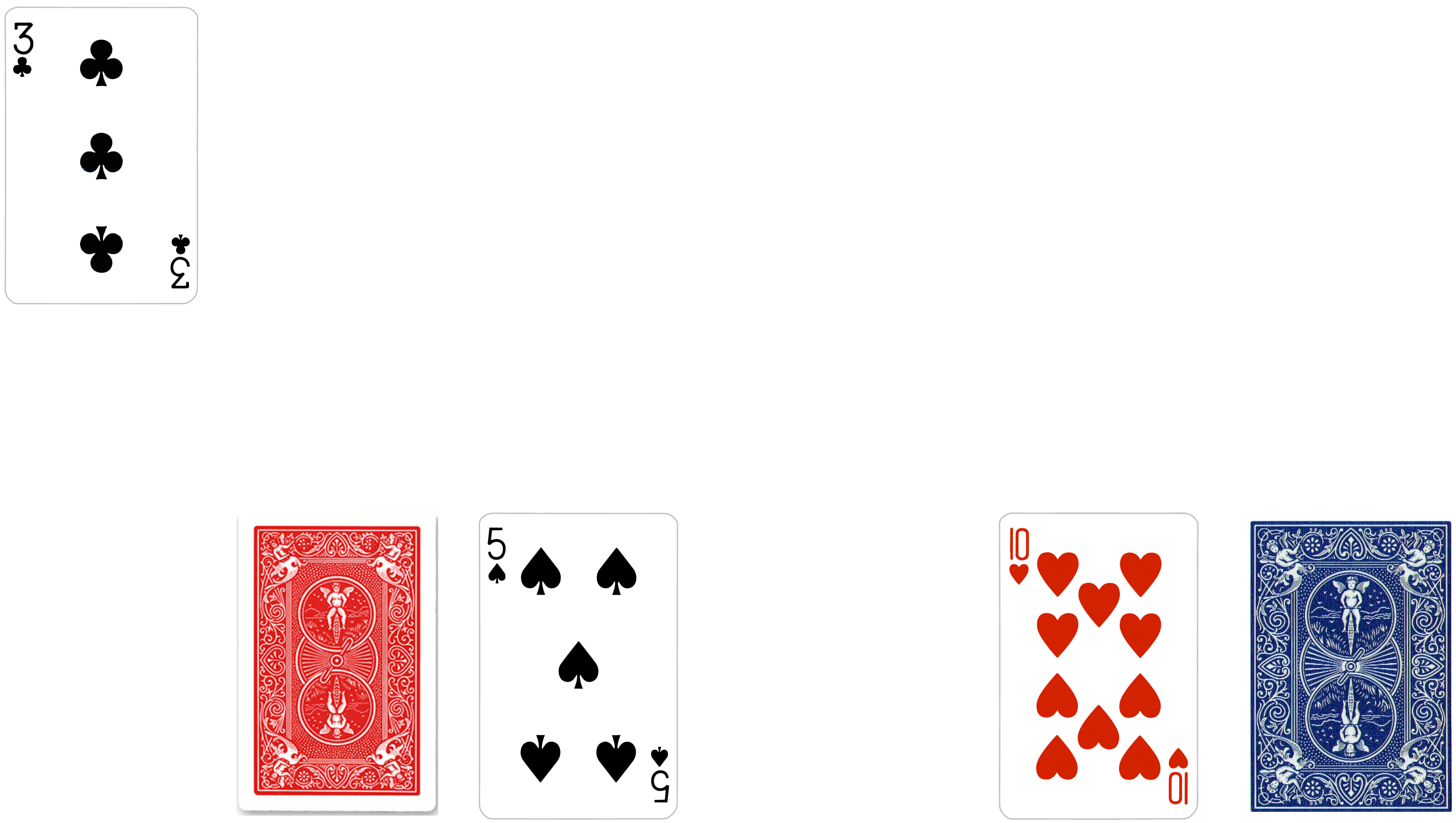
How would you do this?



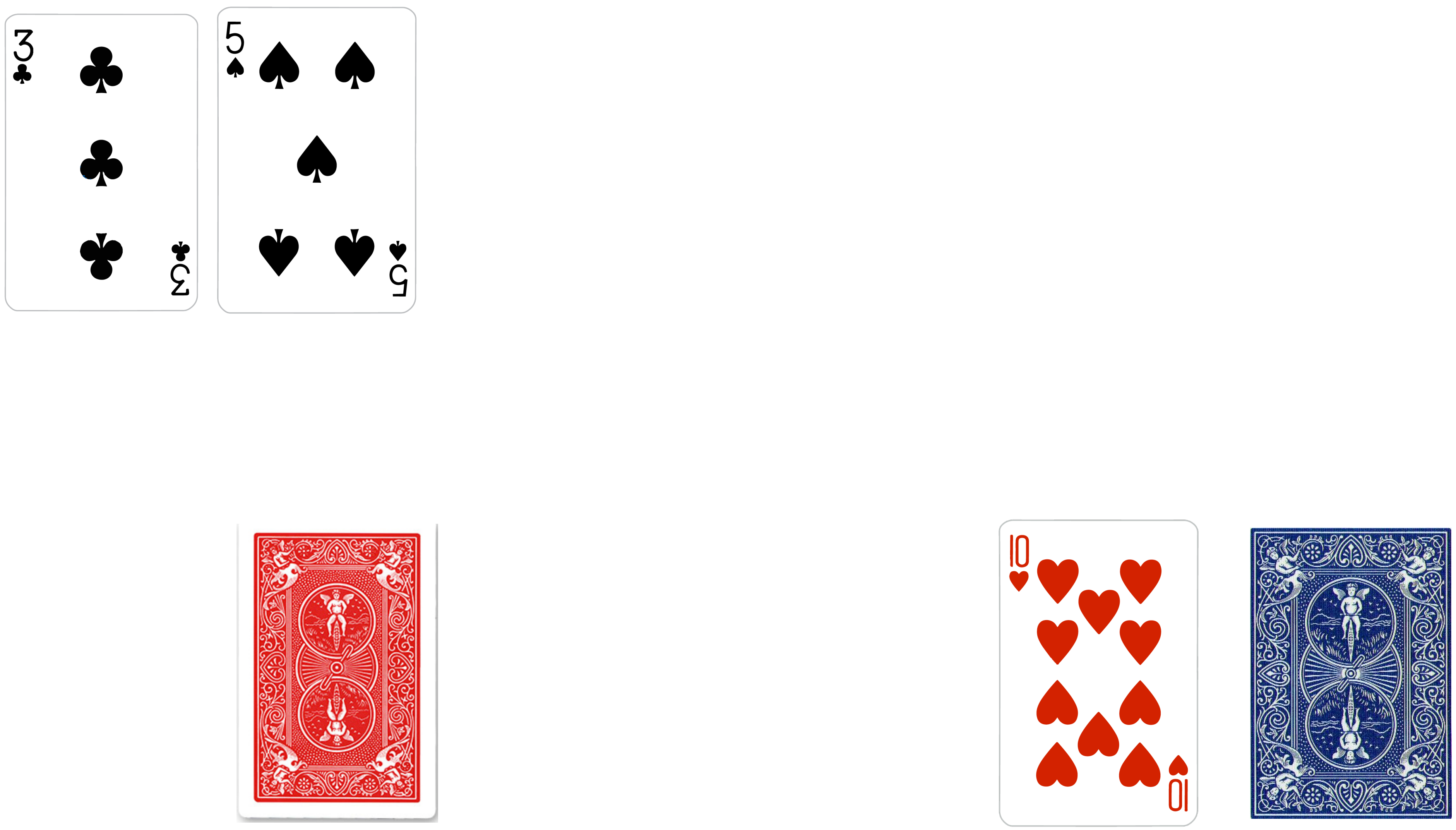
How would you do this?



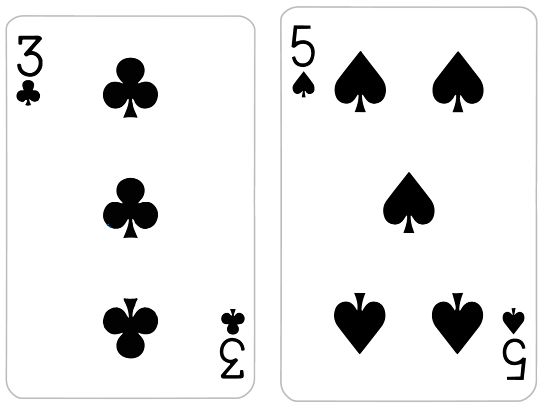
How would you do this?

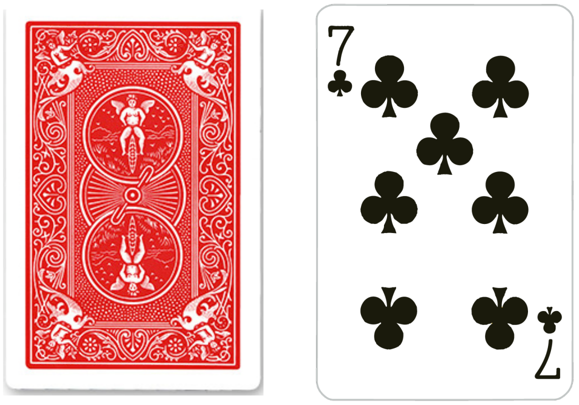


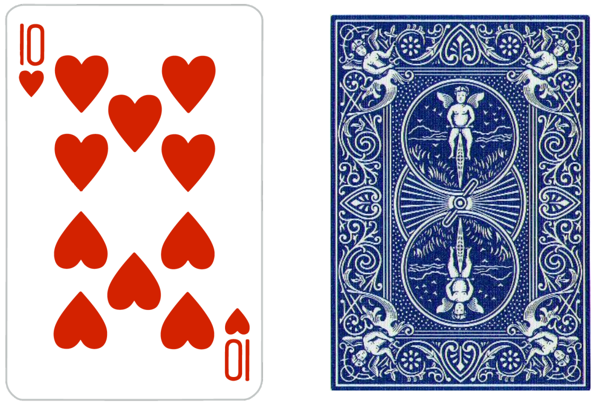
How would you do this?



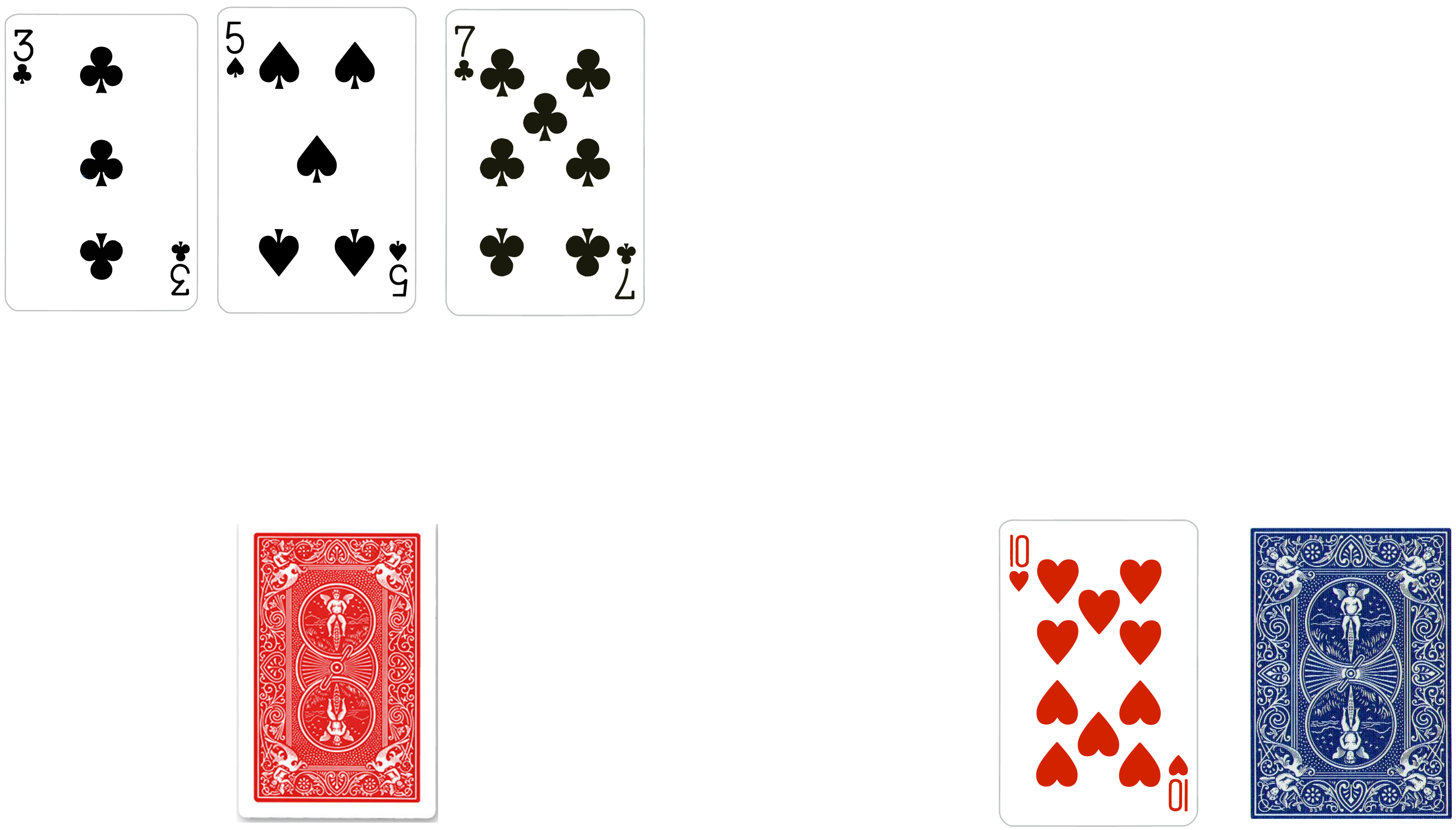
How would you do this?



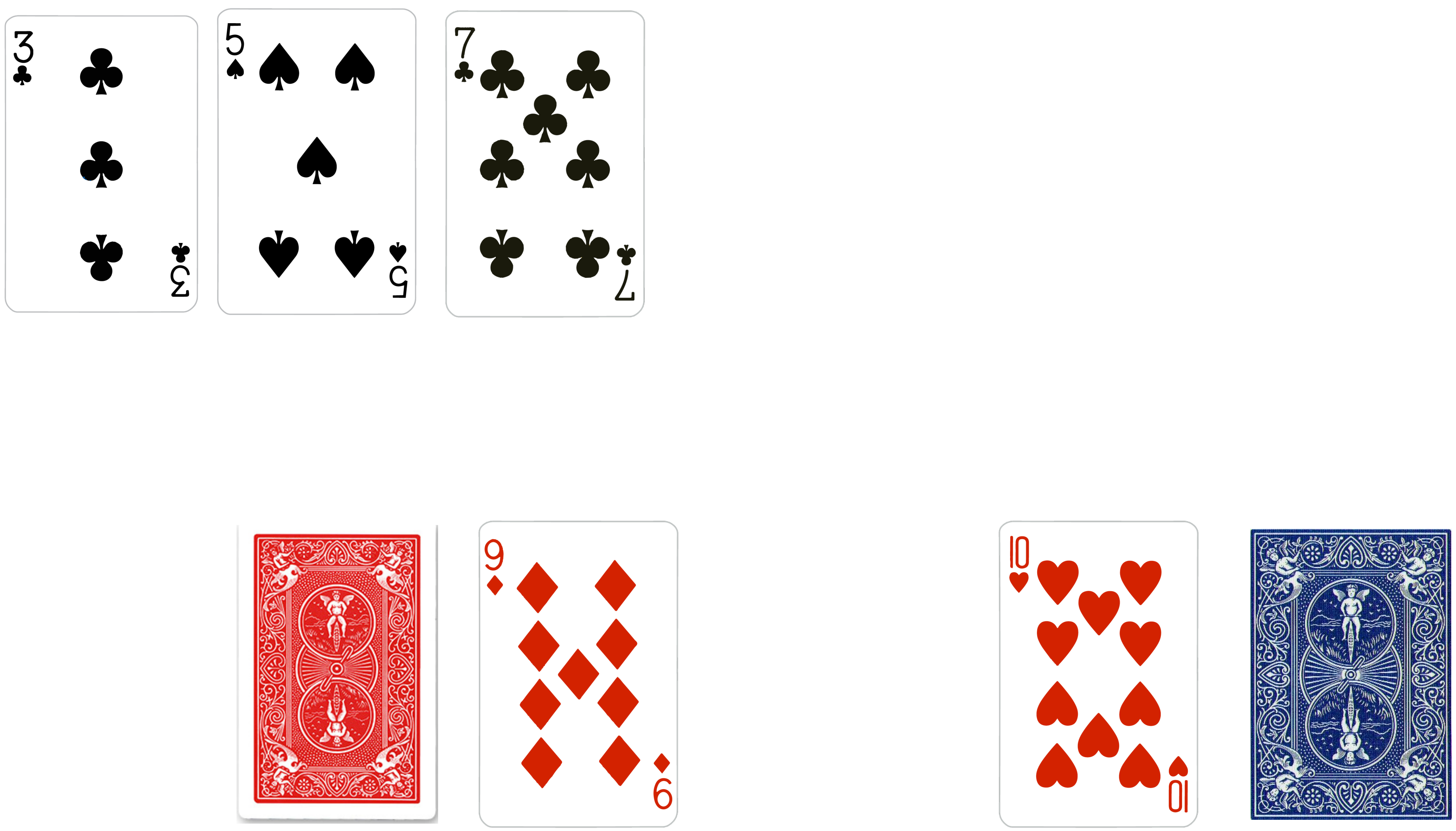




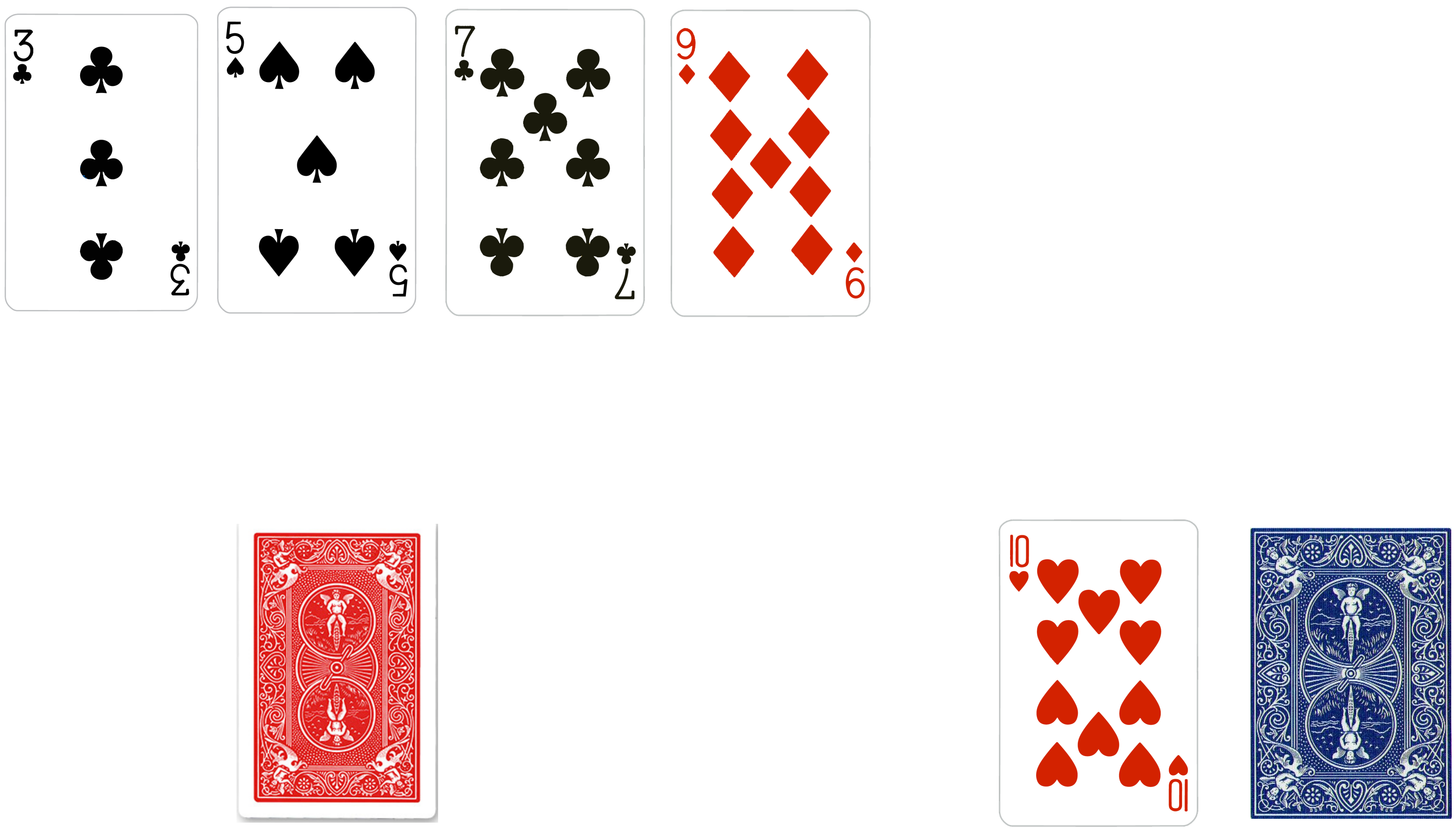
How would you do this?



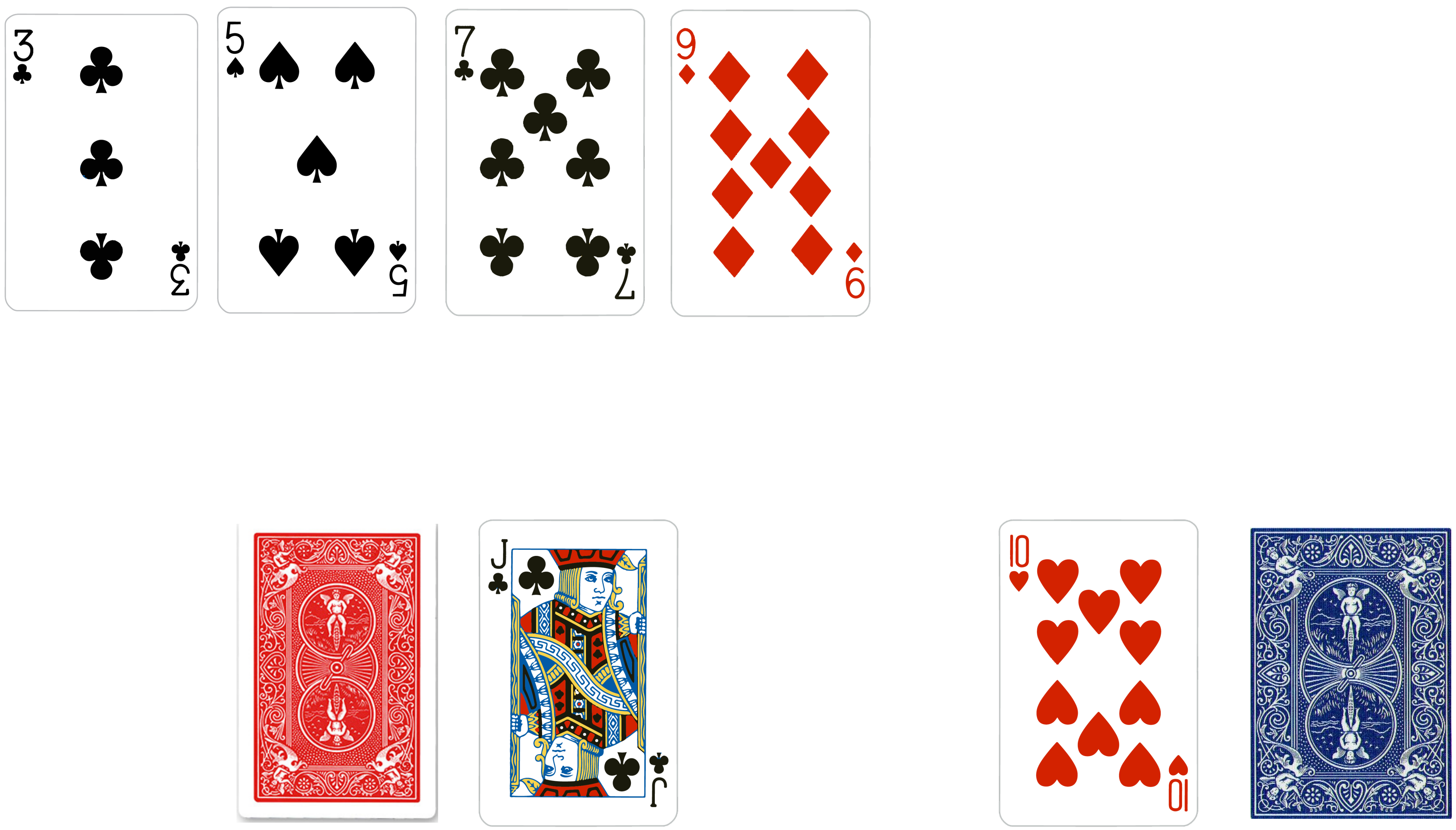
How would you do this?



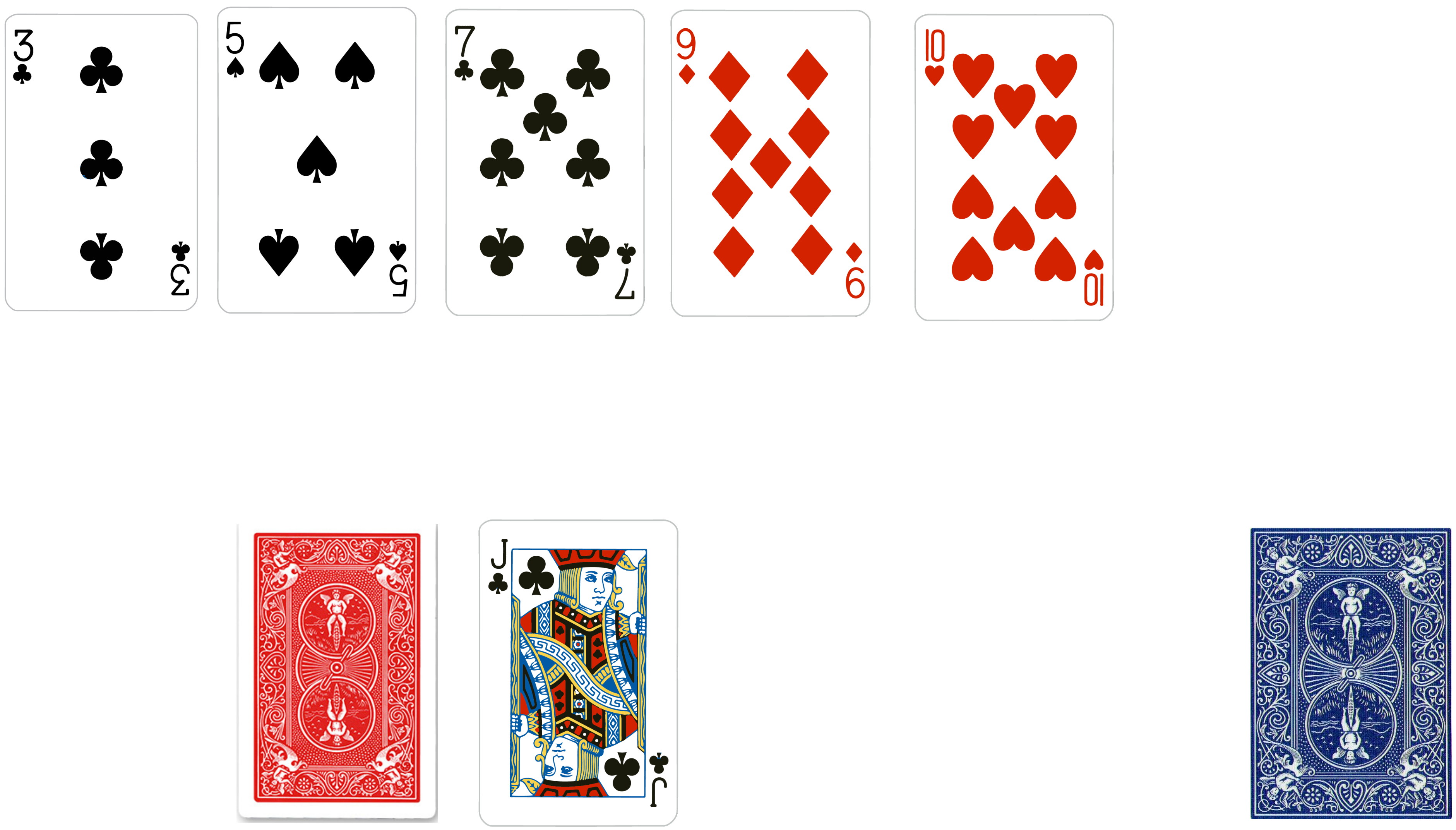
How would you do this?



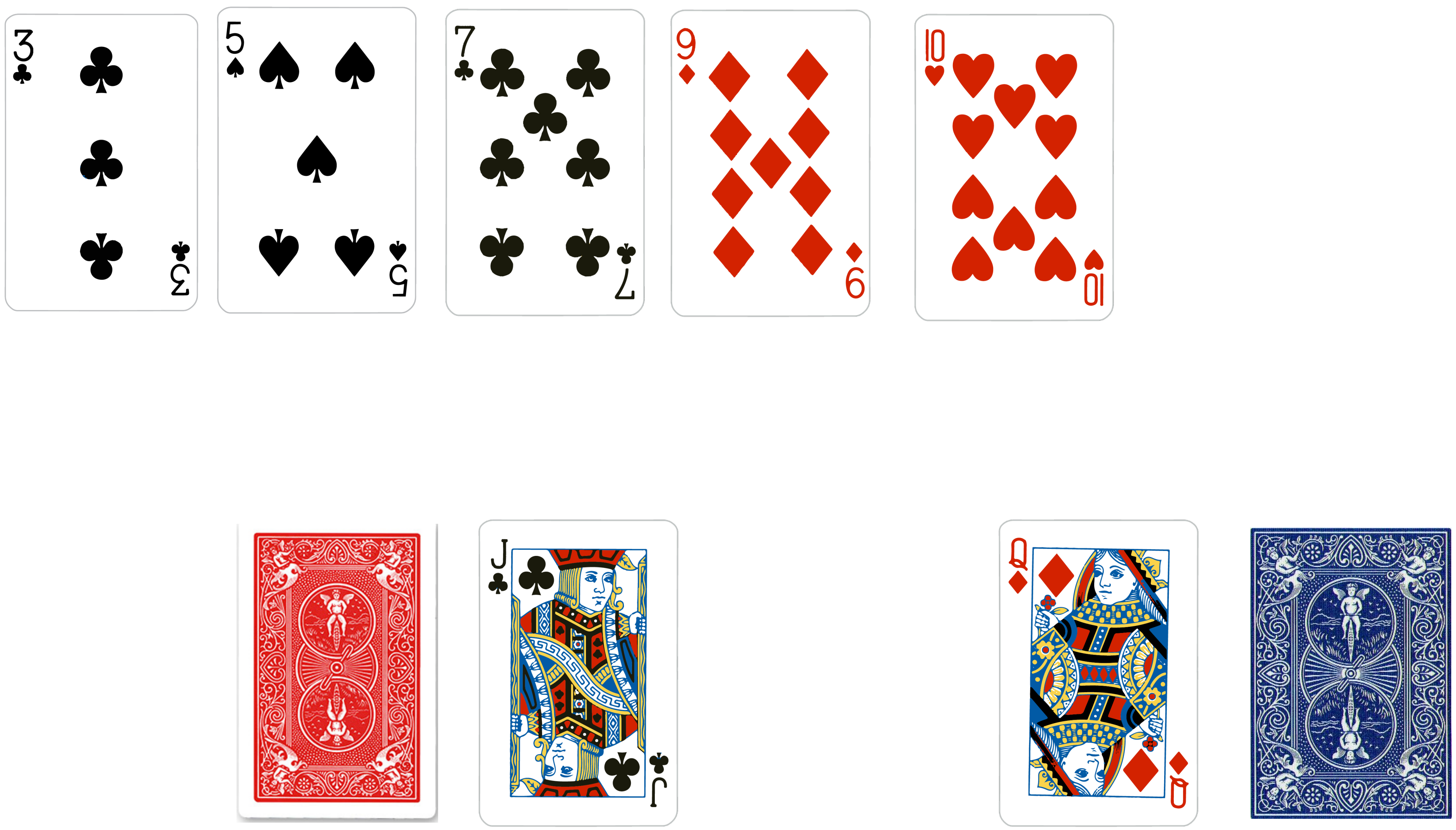
How would you do this?



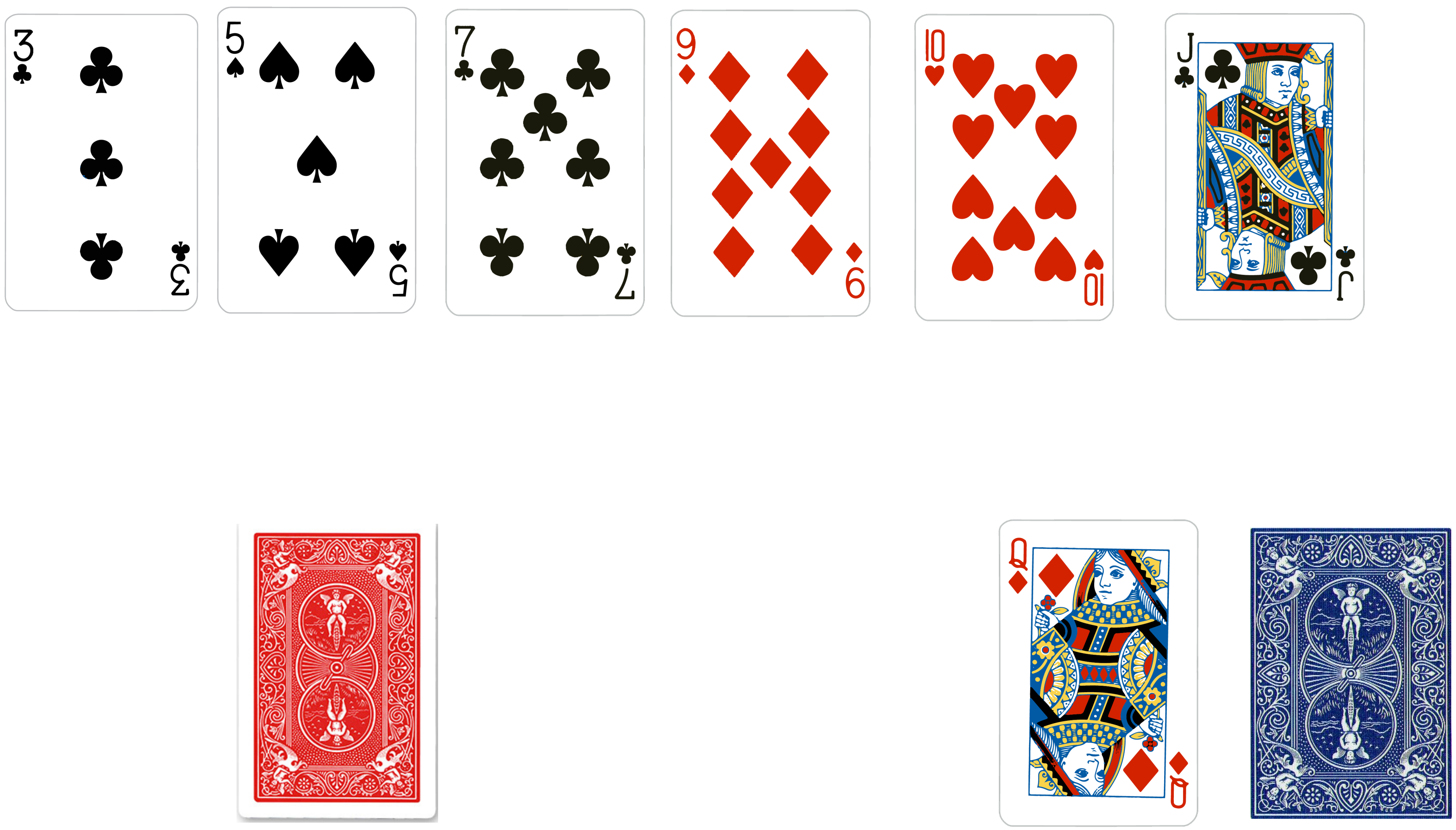
How would you do this?



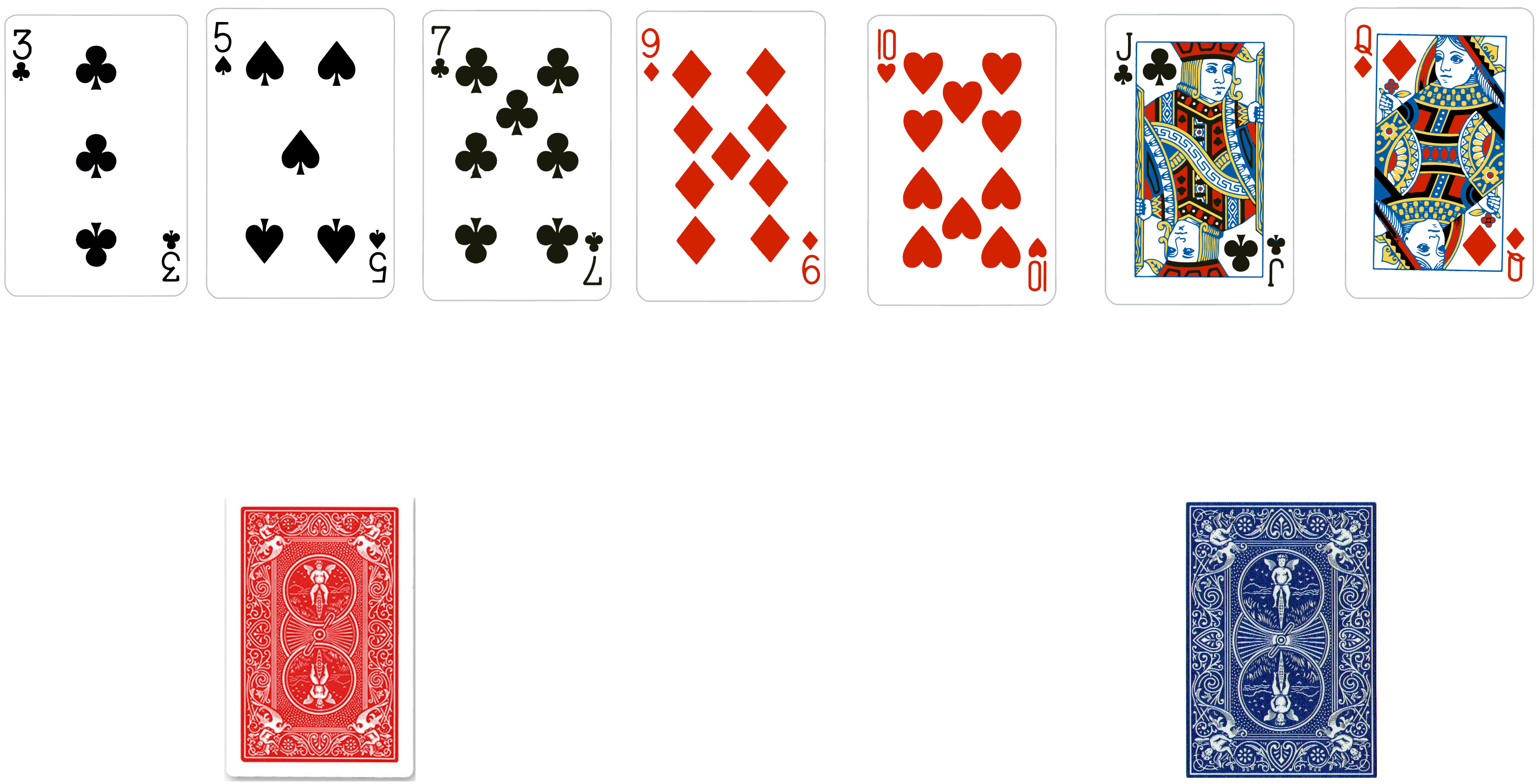
How would you do this?



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] ≤ **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

The Mergesort algorithm

Divide and conquer algorithm.

The Mergesort algorithm

Divide and conquer algorithm.

Split the array **A**[*1,…,n*] to two subarrays,
A[*1,…,n/2*] and **A**[*n/2+1, …, n*]

The Mergesort algorithm

Divide and conquer algorithm.

Split the array **A**[*1,…,n*] to two subarrays,
A[*1,…,n/2*] and **A**[*n/2+1, …, n*]

Sort each subarray using **Mergesort**.

The Mergesort algorithm

Divide and conquer algorithm.

Split the array **A**[*1,…,n*] to two subarrays,
A[*1,…,n/2*] and **A**[*n/2+1, …, n*]

Sort each subarray using **Mergesort**.

Stop the recursion when the subarray contains only one element.

The Mergesort algorithm

Divide and conquer algorithm.

Split the array **A**[$1,\ldots,n$] to two subarrays,
A[$1,\ldots,n/2$] and **A**[$n/2+1, \ldots, n$]

Sort each subarray using **Mergesort**.

Stop the recursion when the subarray contains only one element.

Merge the sorted subarrays **A**[$1,\ldots,n/2$] and **A**[$n/2+1, \ldots, n$] using the **Merge** procedure.

Mergesort pseudocode

Algorithm **Mergesort**(**A**[$i,...,j$])

If $i=j$, return i

$q=(i+j)/2$

$\mathbf{A}_{left}$=**Mergesort**(**A**[$i,...,q$])

$\mathbf{A}_{right}$=**Mergesort**(**A**[$q+1,...,n$])

return **Merge**($\mathbf{A}_{left}$, $\mathbf{A}_{right}$)

Mergesort pseudocode

Algorithm **Mergesort**(**A**[$i,\ldots,j$])

If $i=j$, return i

$q=(i+j)/2$

$\mathbf{A}_{\text{left}}$=**Mergesort**(**A**[$i,\ldots,q$])

$\mathbf{A}_{\text{right}}$=**Mergesort**(**A**[$q+1,\ldots,n$])

return **Merge**($\mathbf{A}_{\text{left}}$, $\mathbf{A}_{\text{right}}$)

Initial call: **Mergesort**(**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

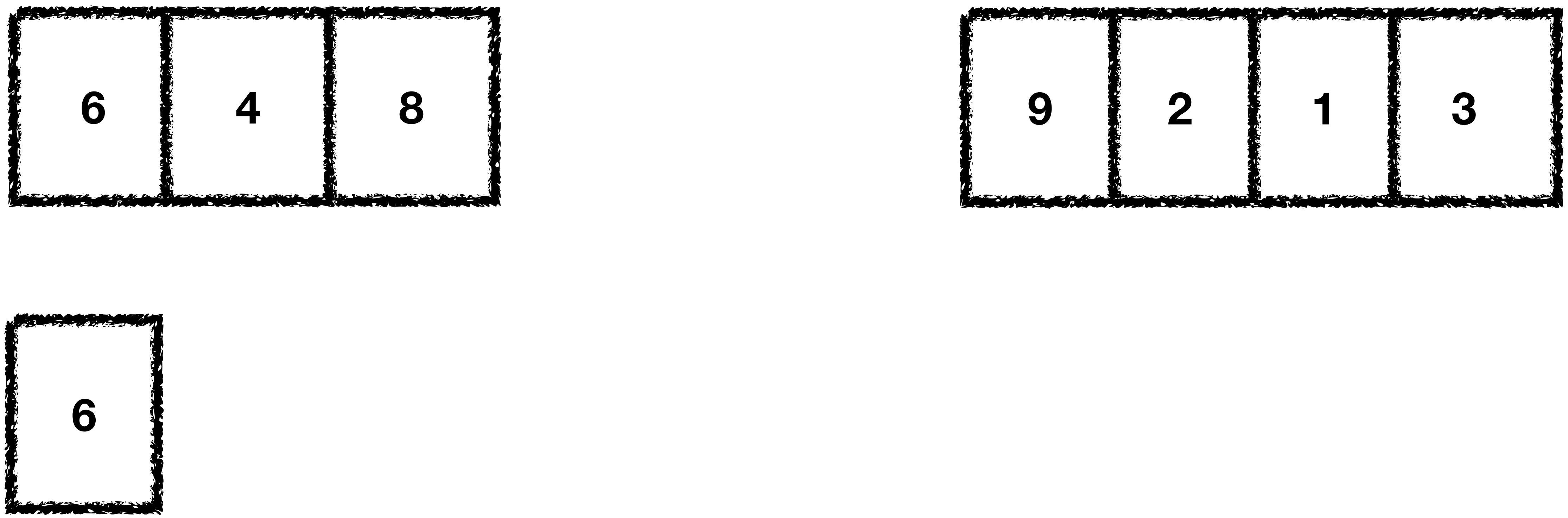
divide

6	4	8

9	2	1	3

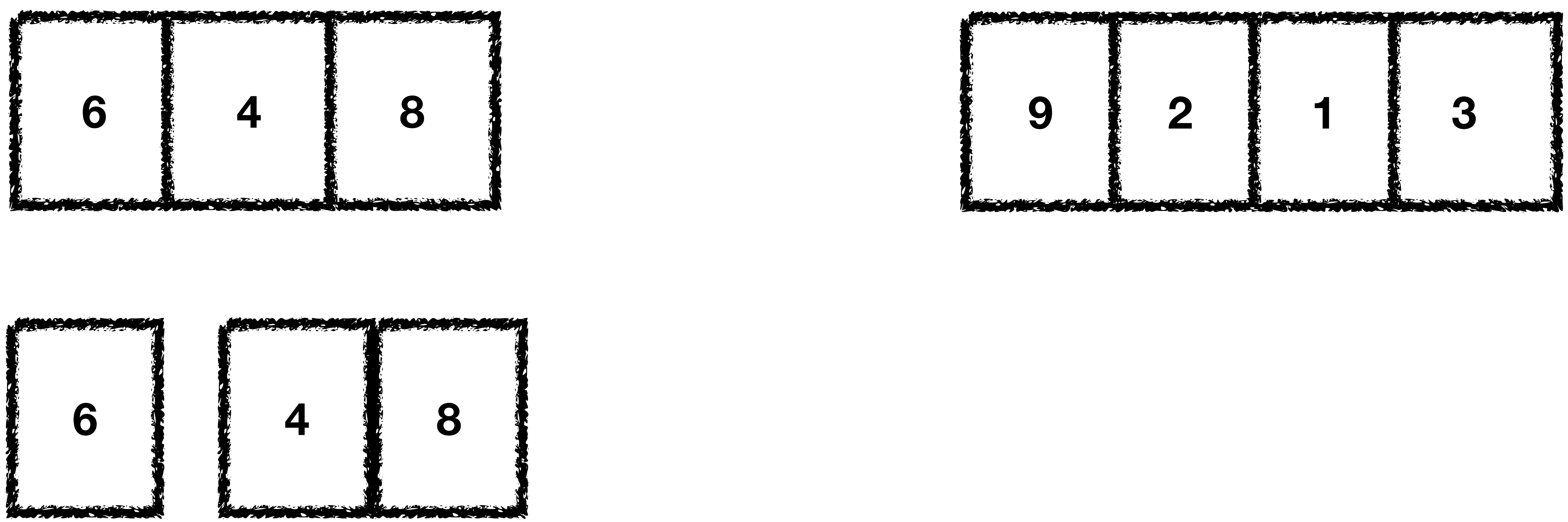
Mergesort example

divide



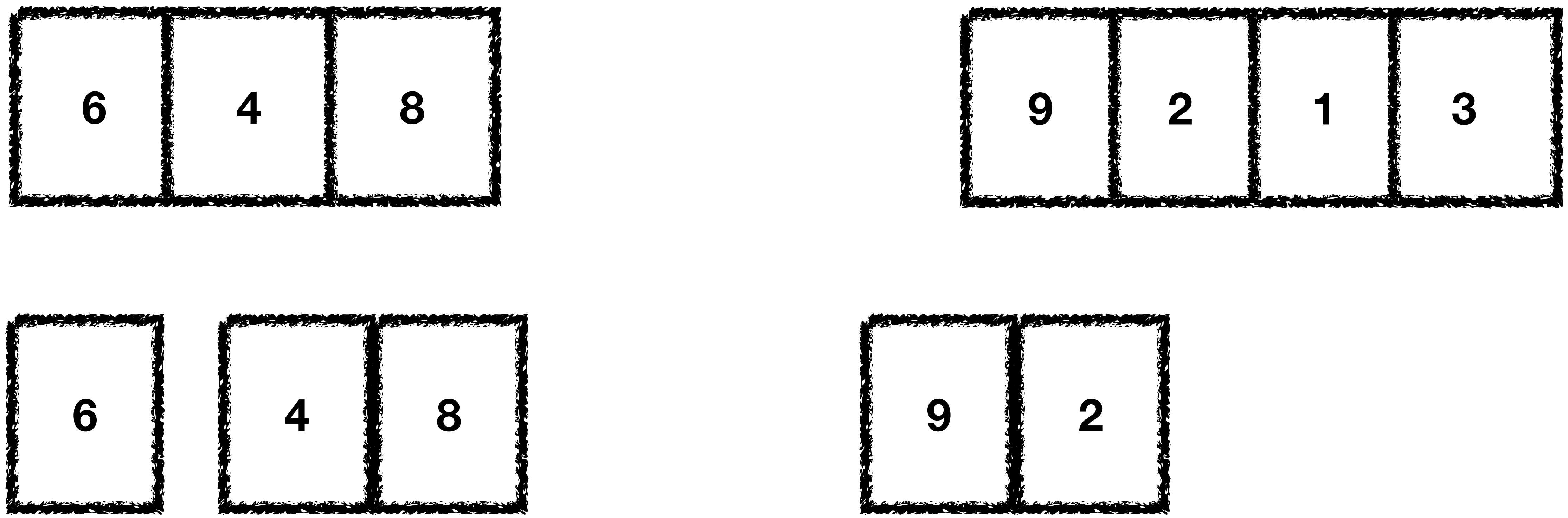
Mergesort example

divide



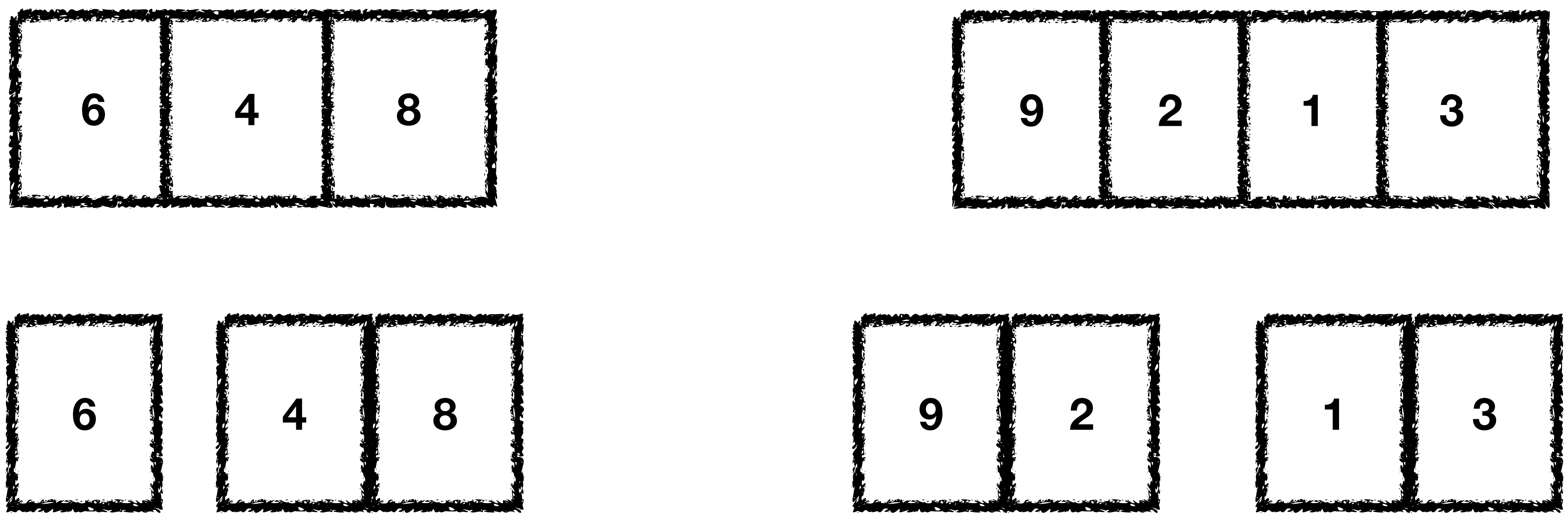
Mergesort example

divide



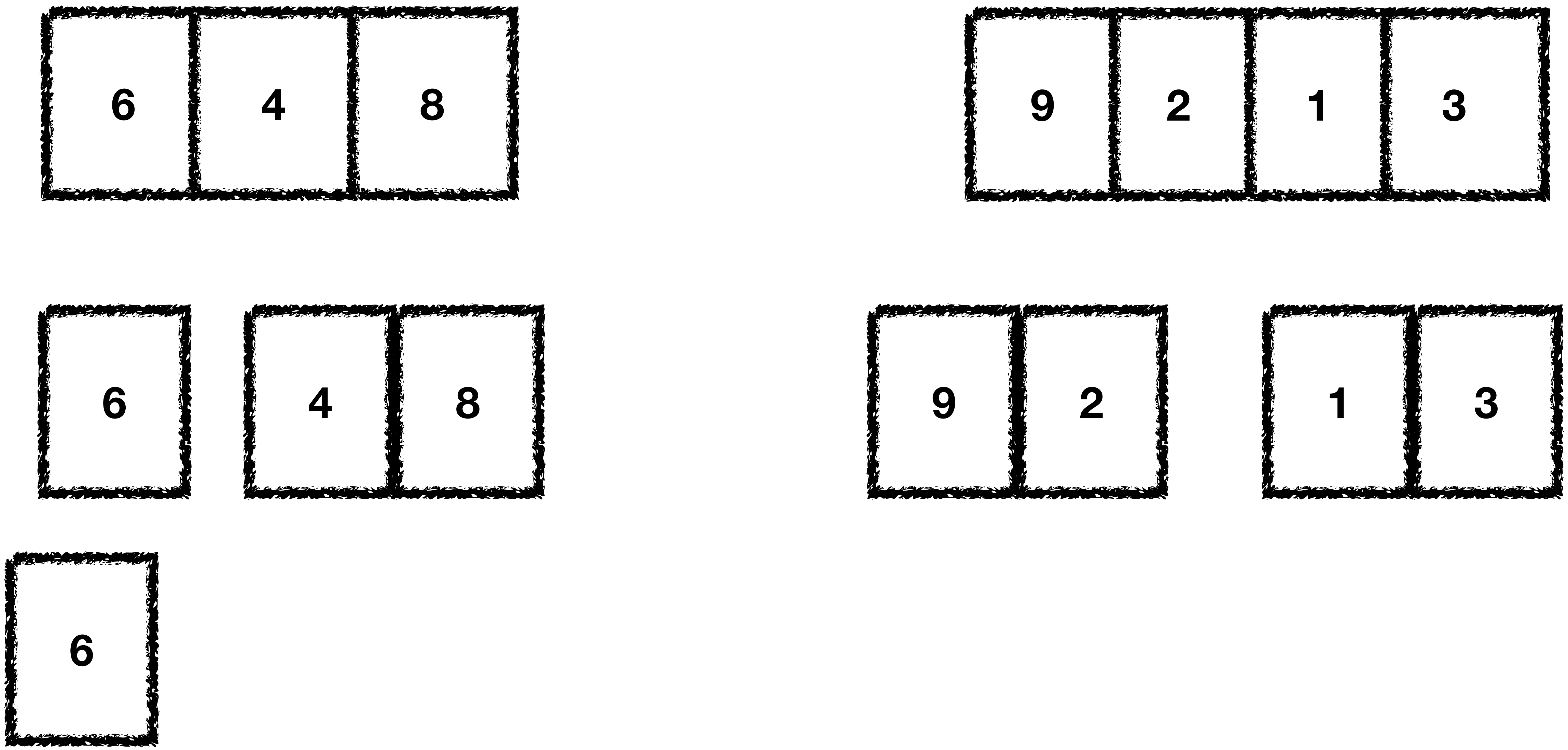
Mergesort example

divide



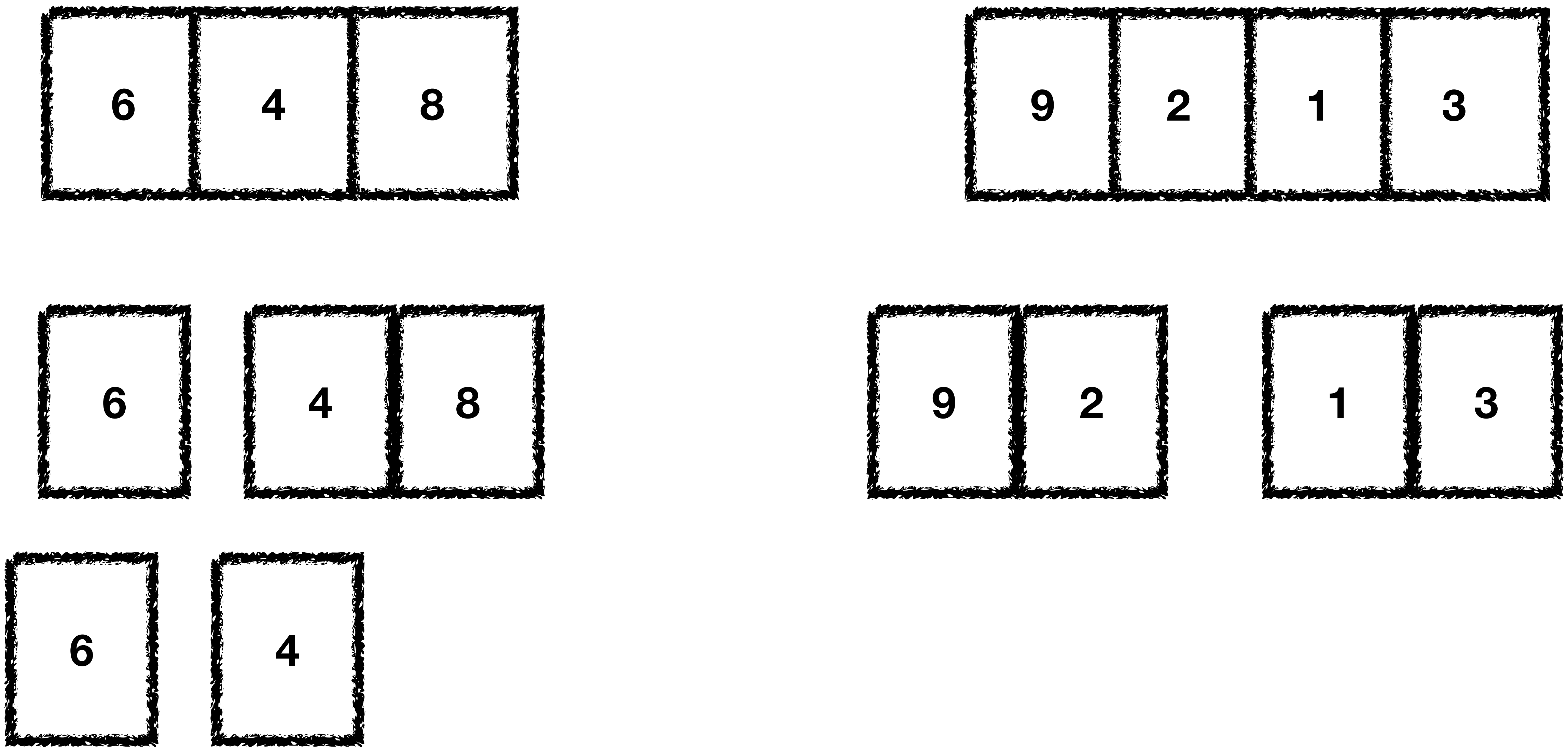
Mergesort example

divide



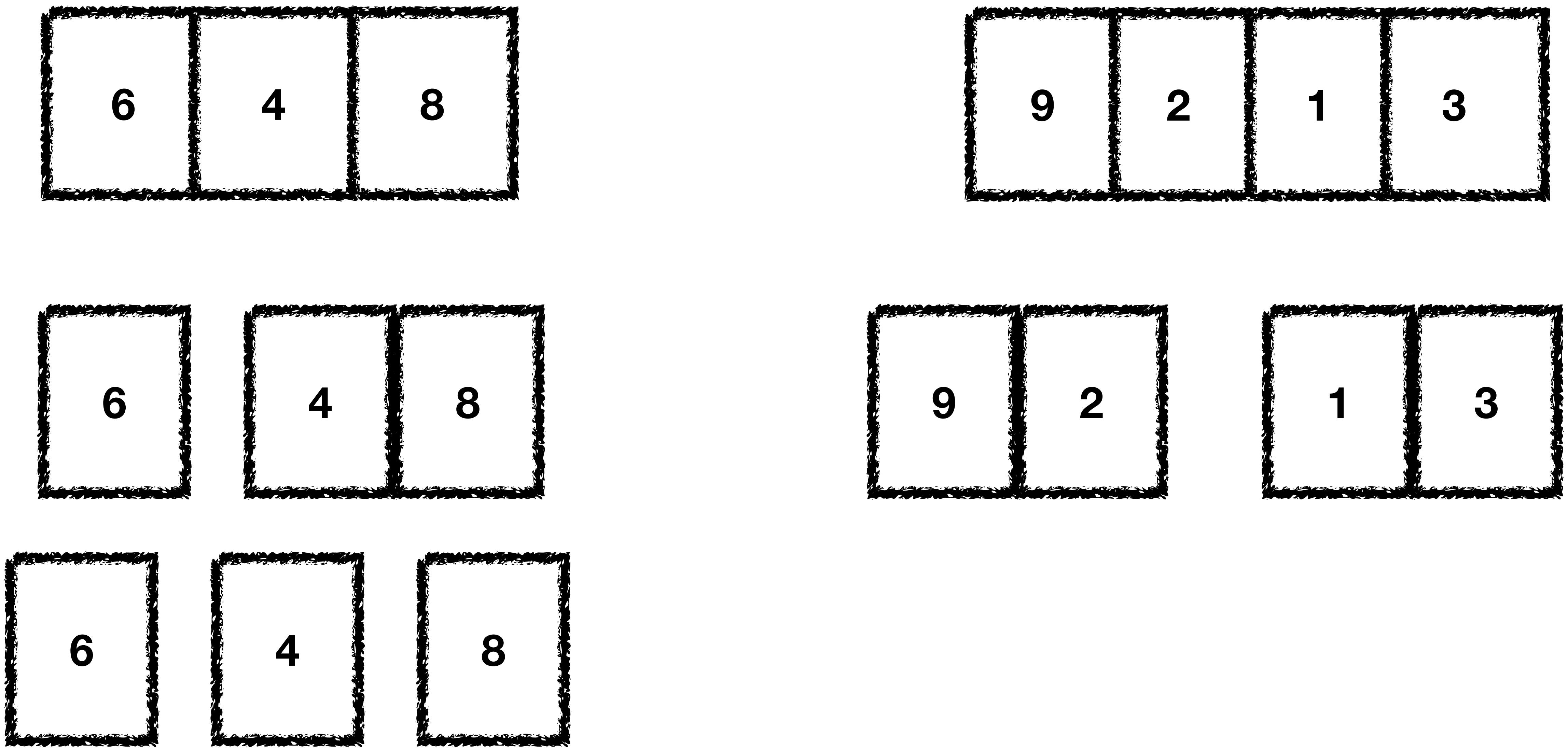
Mergesort example

divide



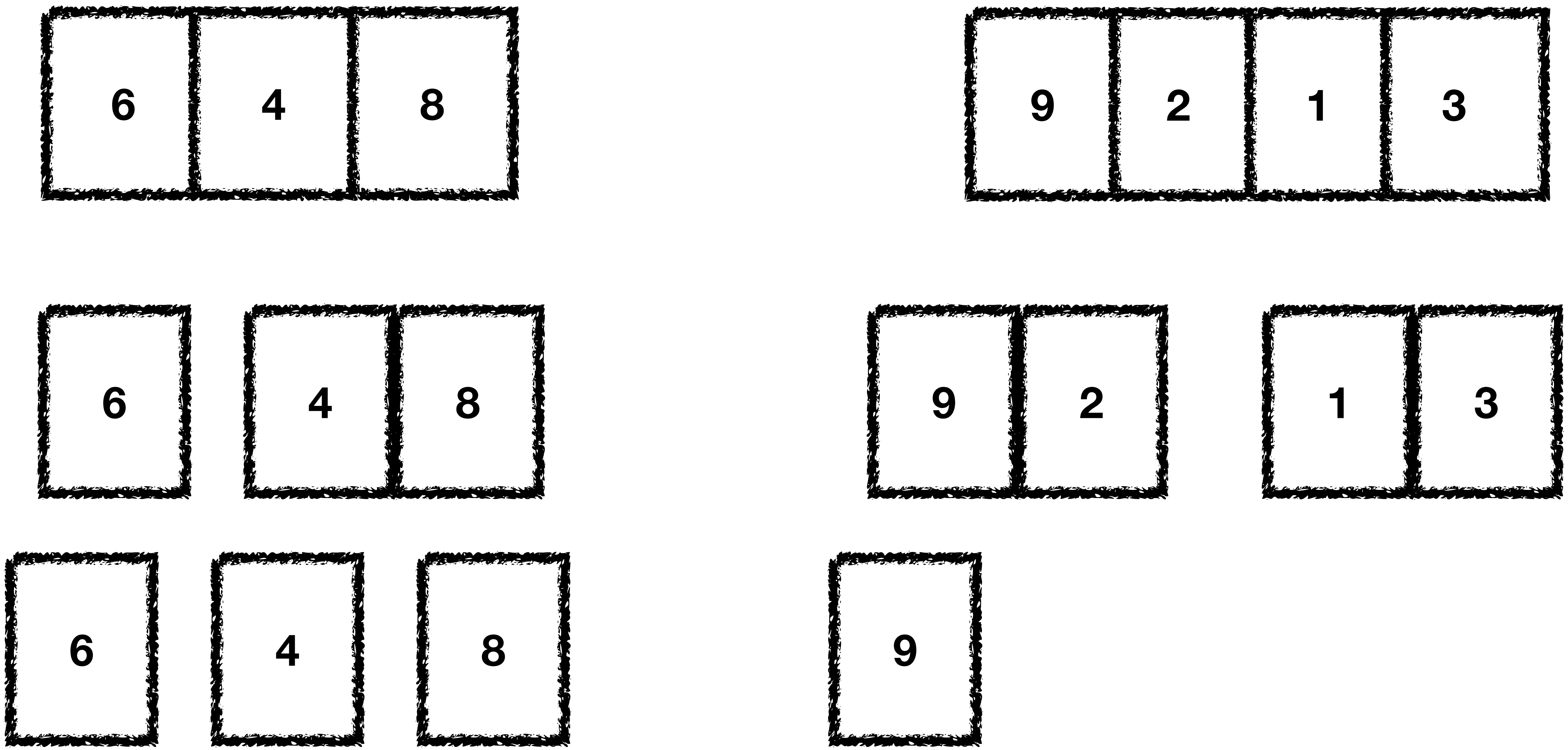
Mergesort example

divide



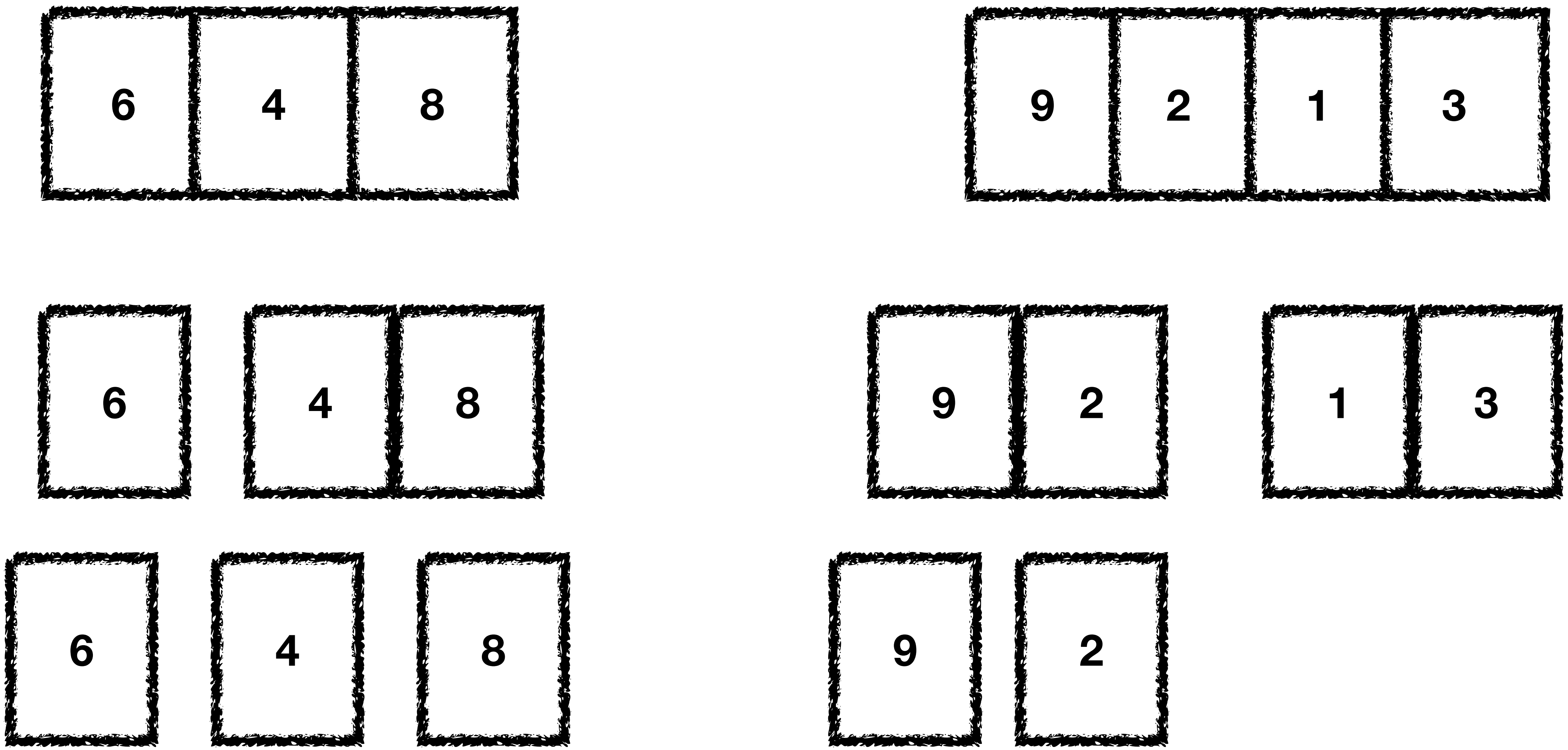
Mergesort example

divide



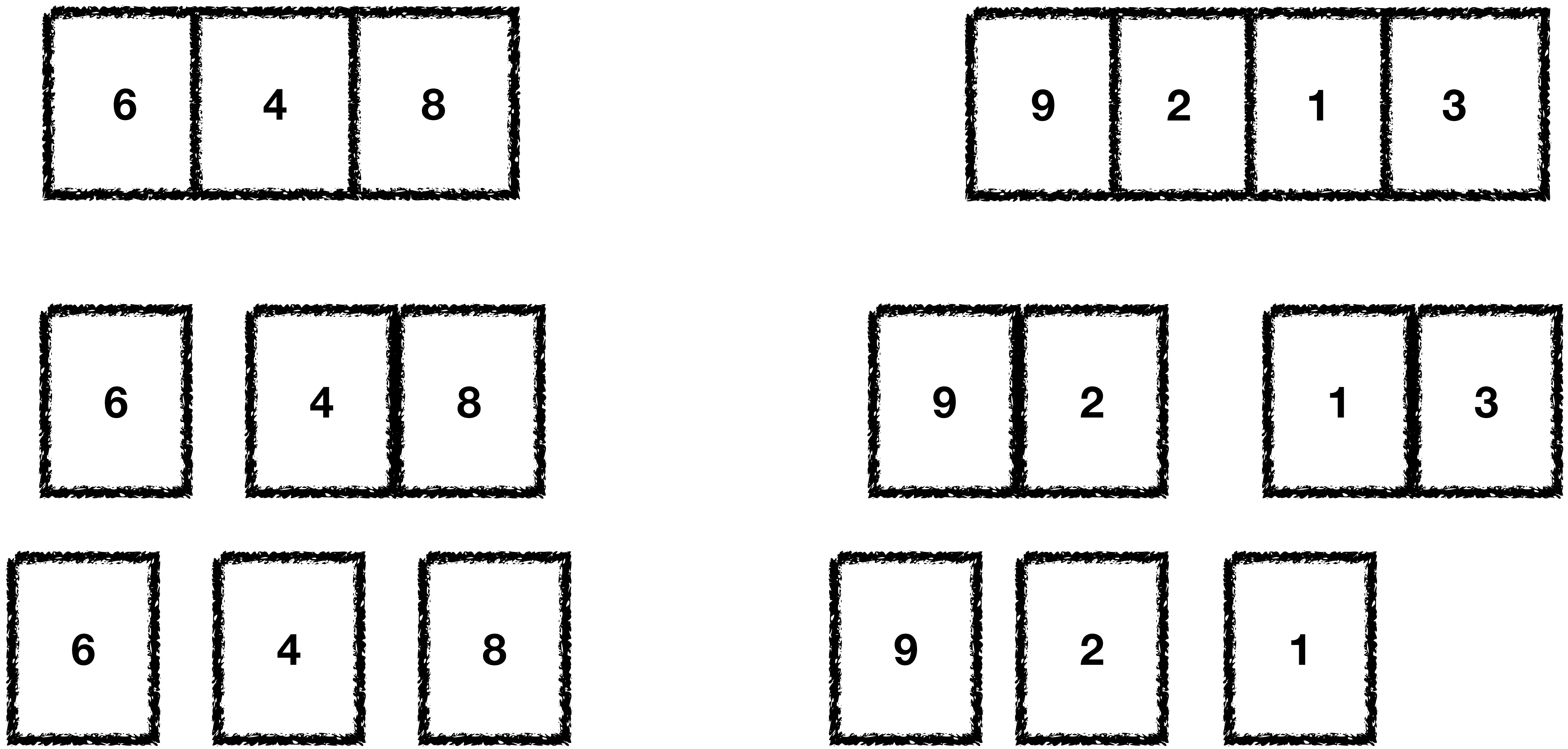
Mergesort example

divide



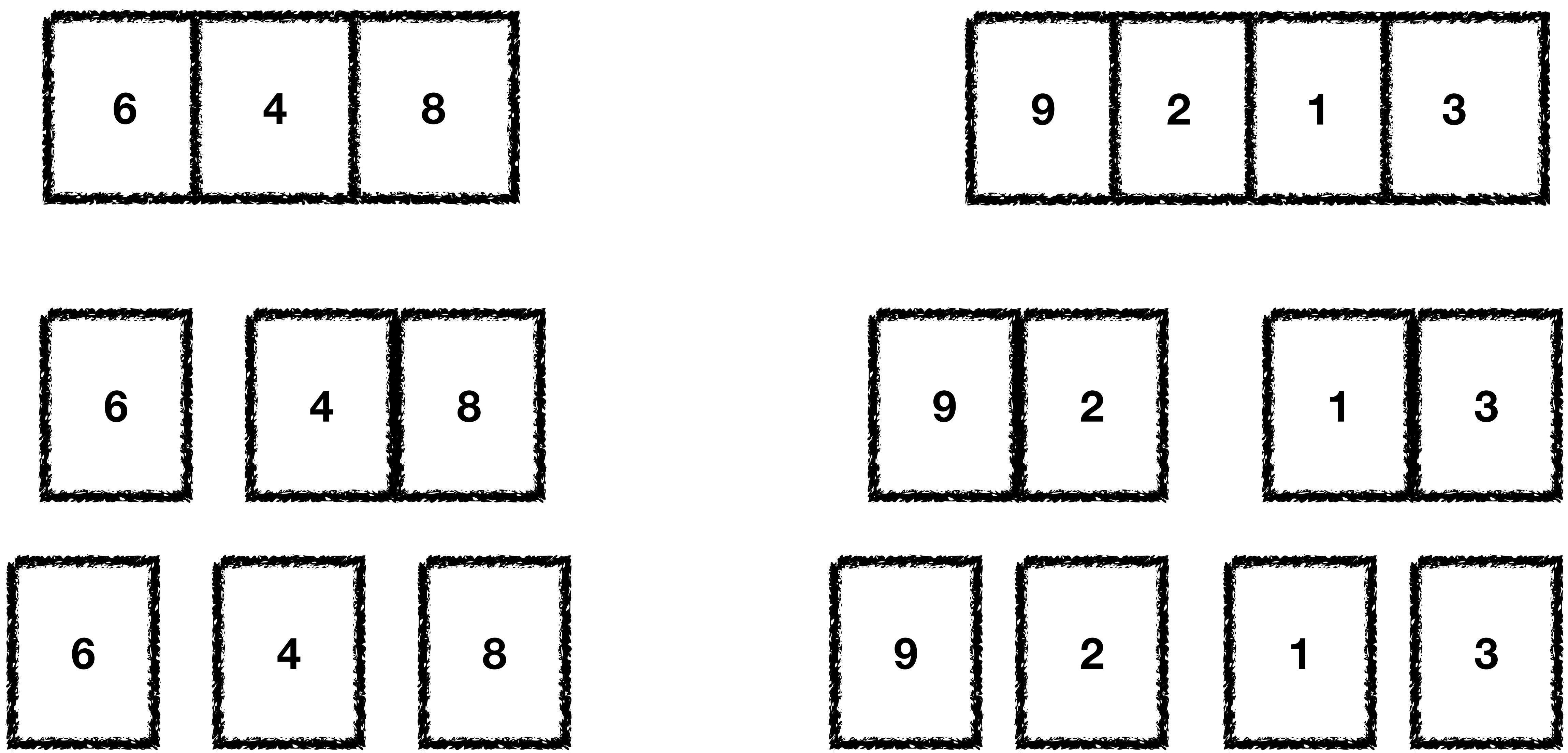
Mergesort example

divide



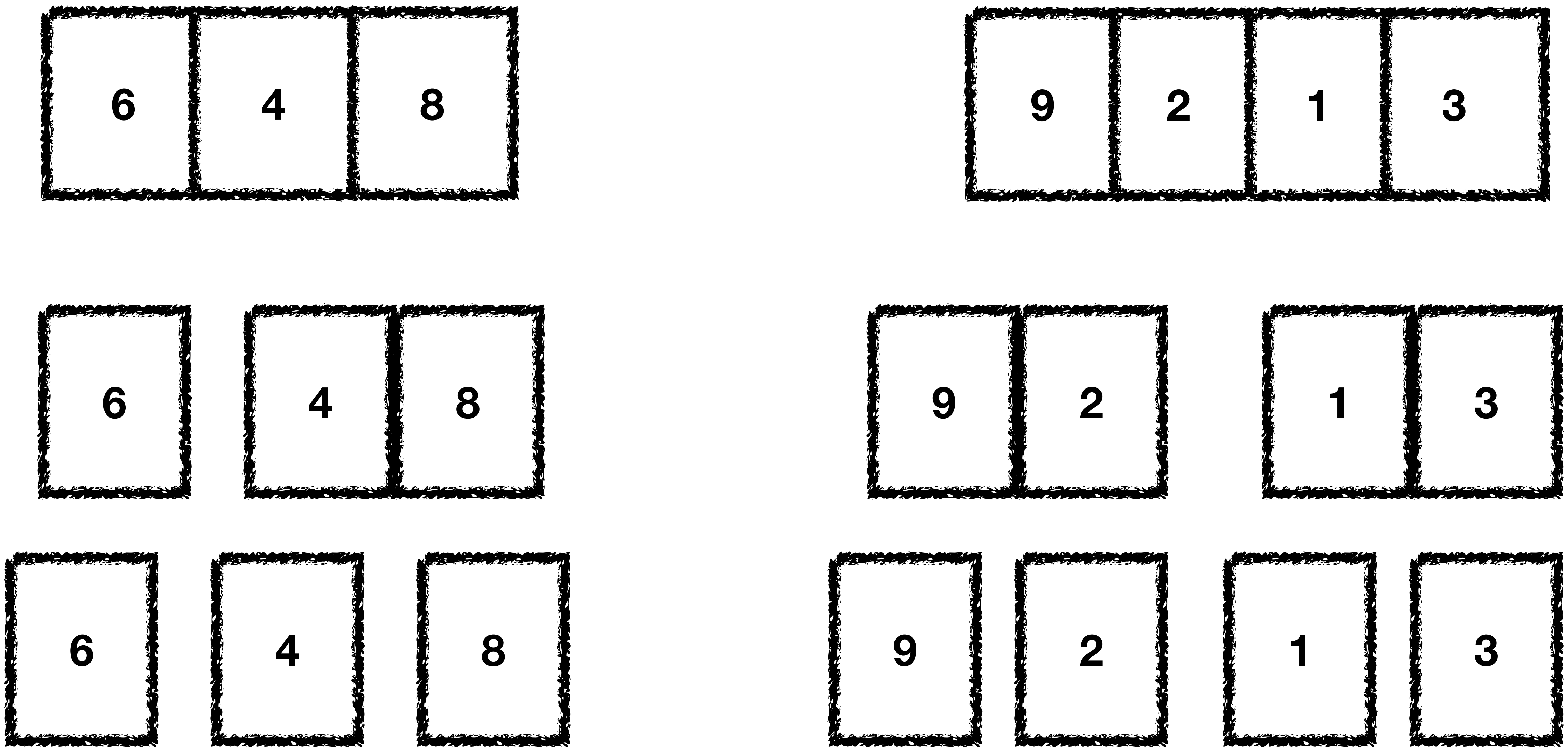
Mergesort example

divide

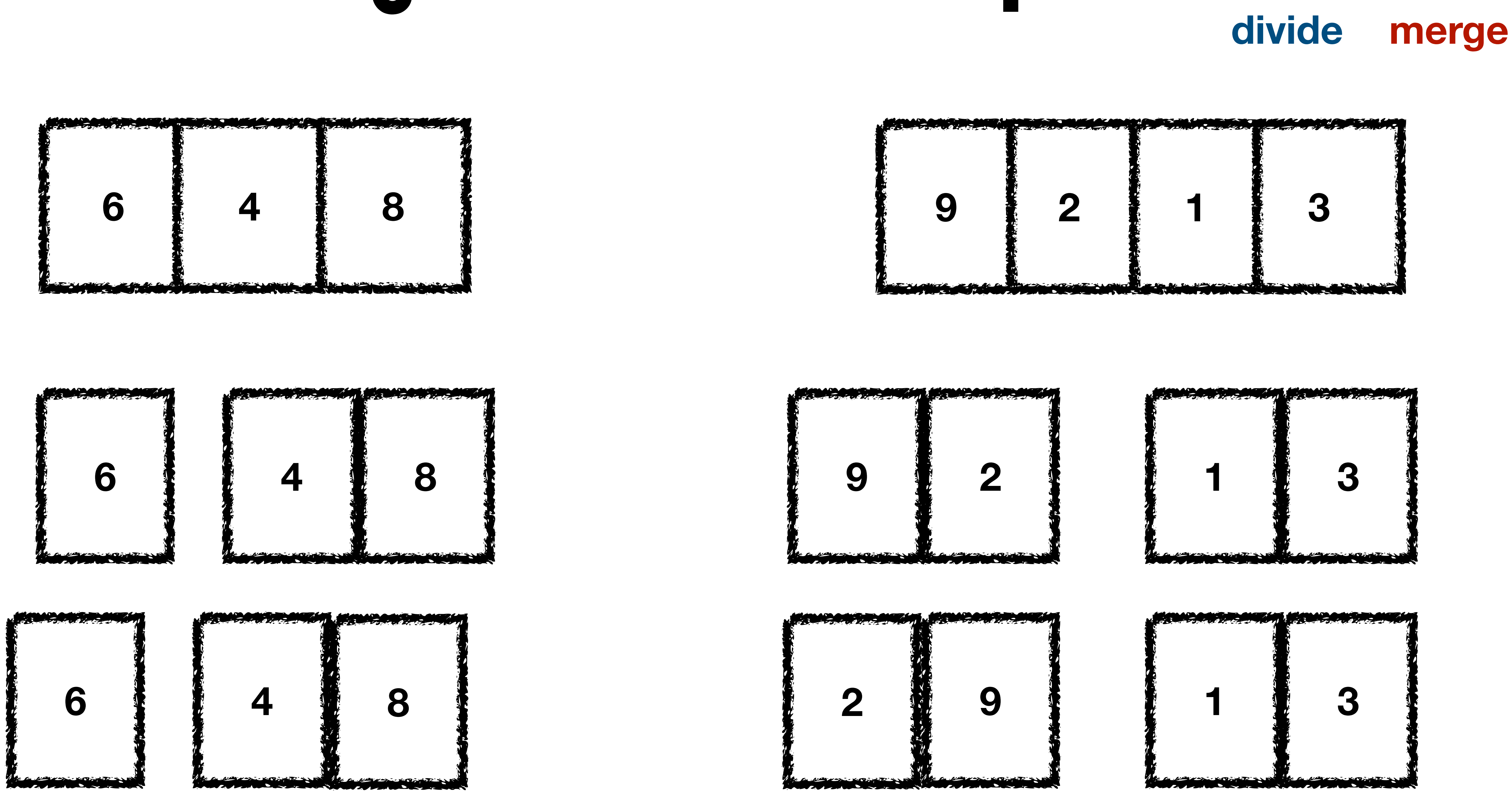


Mergesort example

divide merge

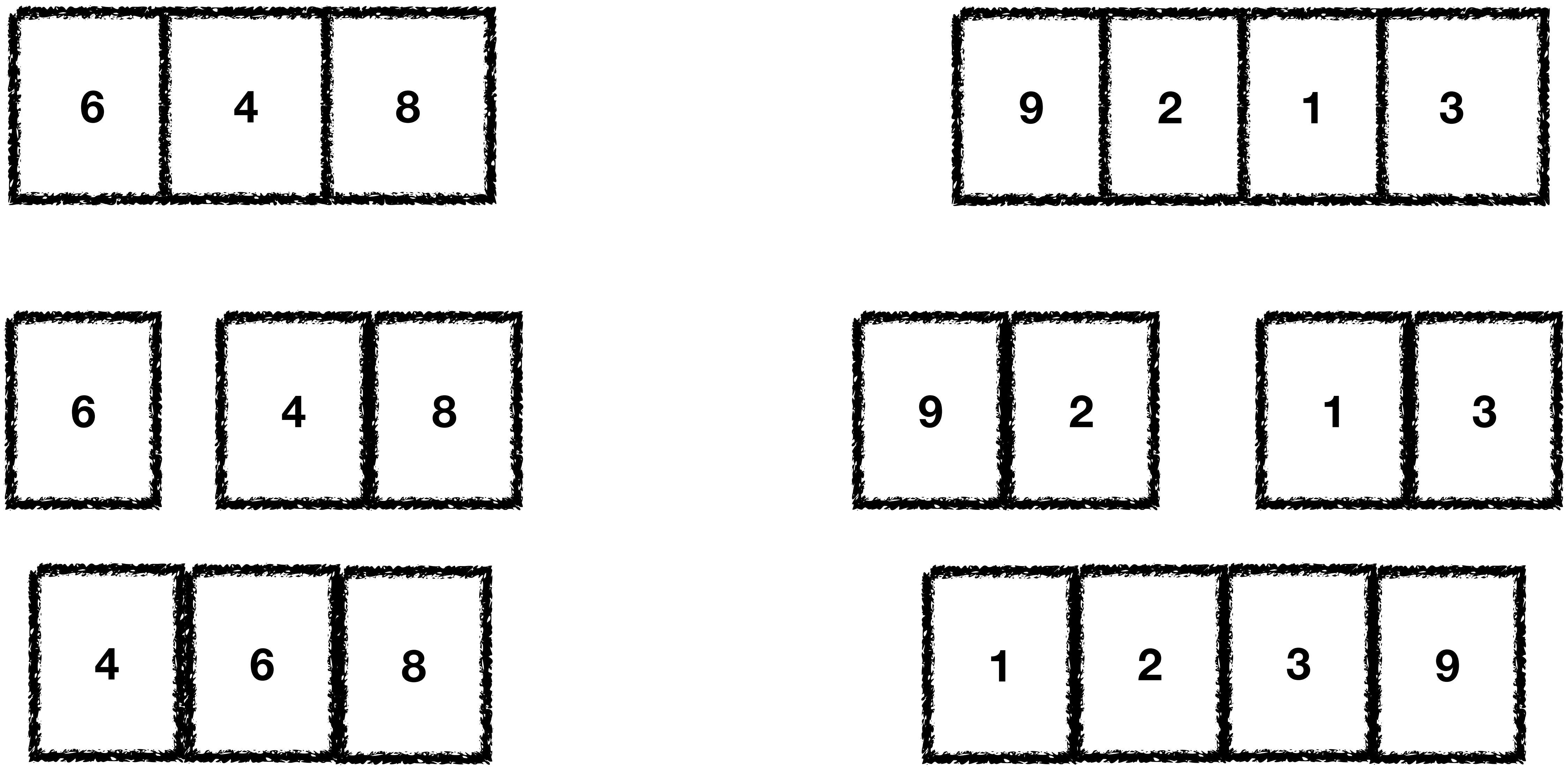


Mergesort example



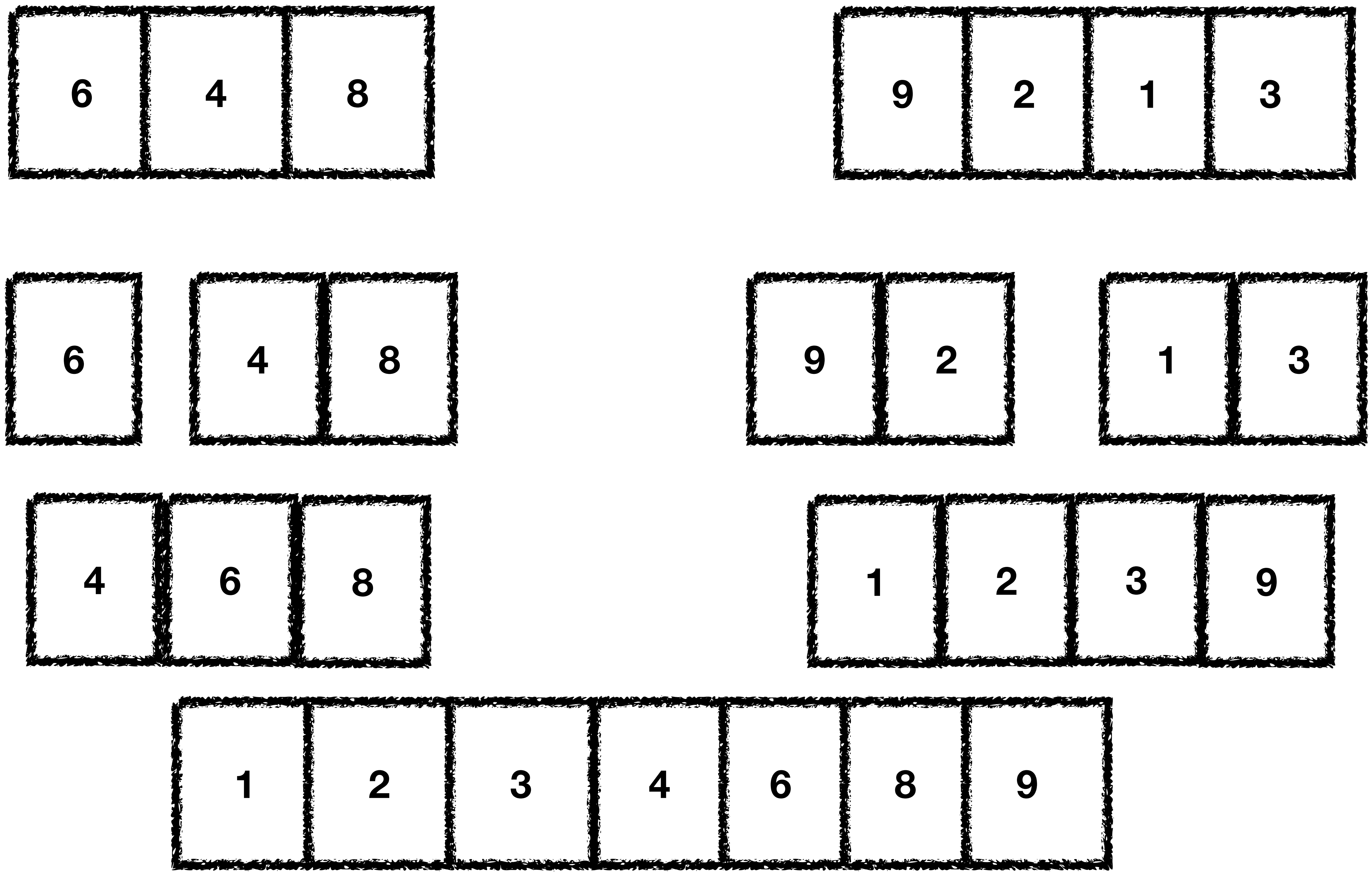
Mergesort example

divide merge



Mergesort example

divide merge



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**[$i,\ldots,j$])

Choose a **pivot element** **x** of **A**

$k = i$

For $h = i$ to j do

If **A**[h] < **x**

Swap **A**[k] with **A**[h]
$k = k + 1$

Swap **A**[k] with **A**[h]

Return k

The Partition procedure

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

Choose a **pivot element** **x** of **A**

$k = i$

For $h = i$ to j do

If **A**[h] < **x**

Swap **A**[k] with **A**[h]
$k = k + 1$

Swap **A**[k] with **A**[h]

Return k

Correctness of **Partition**:

(CLRS p. 171-173)

The Partition procedure

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

Choose a **pivot element** **x** of **A**

$k = i$

For $h = i$ to j do

If **A**[h] < **x**

Swap **A**[k] with **A**[h]
$k = k + 1$

Swap **A**[k] with **A**[h]

Return k

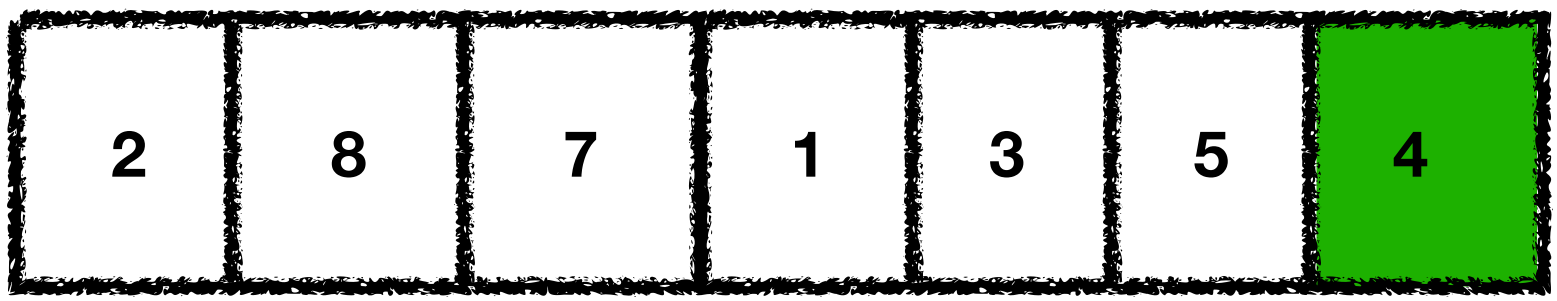
Correctness of **Partition**:
(CLRS p. 171-173)

Running time **O(n)**

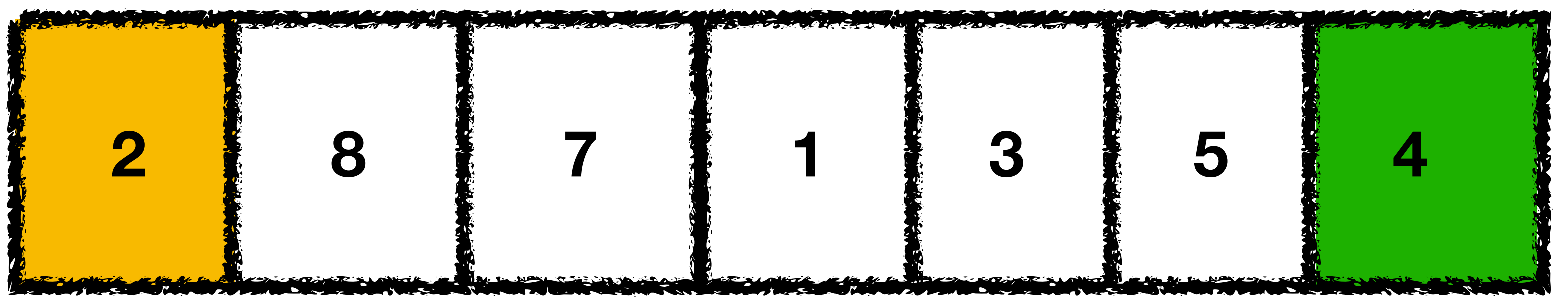
The Quicksort algorithm

2	8	7	1	3	5	4

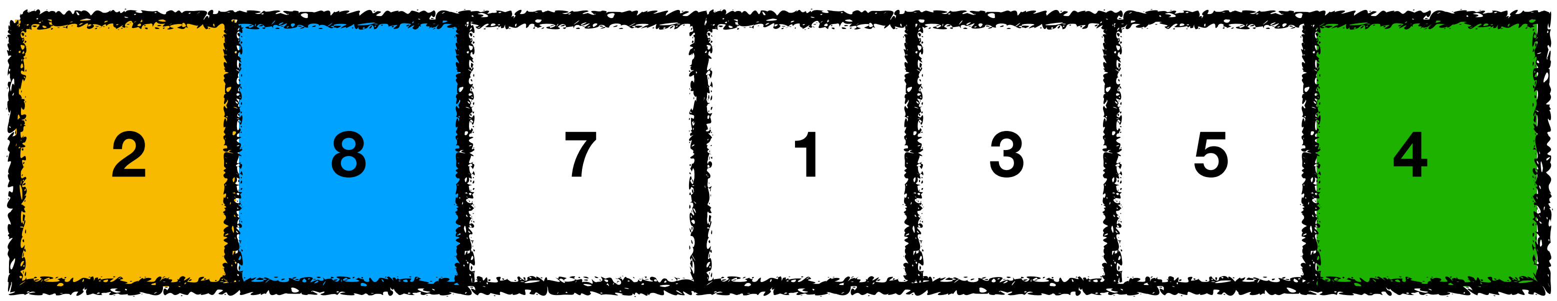
The Quicksort algorithm



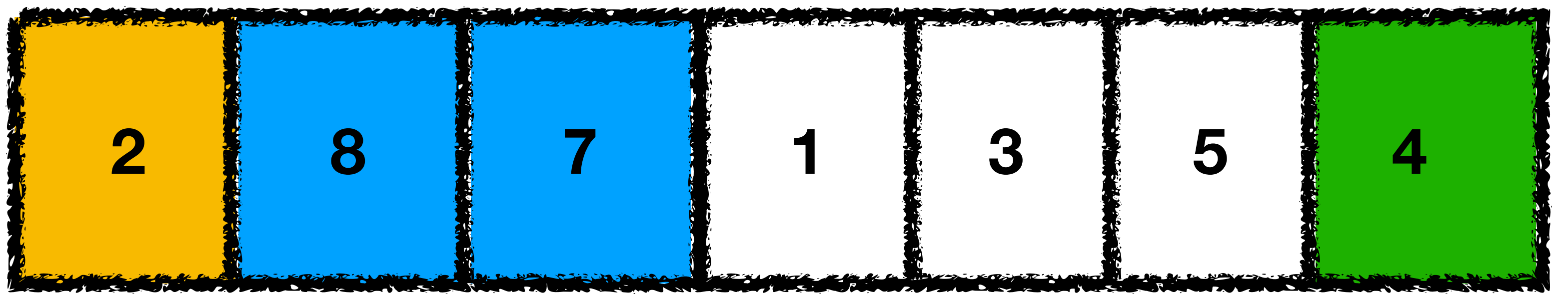
The Quicksort algorithm



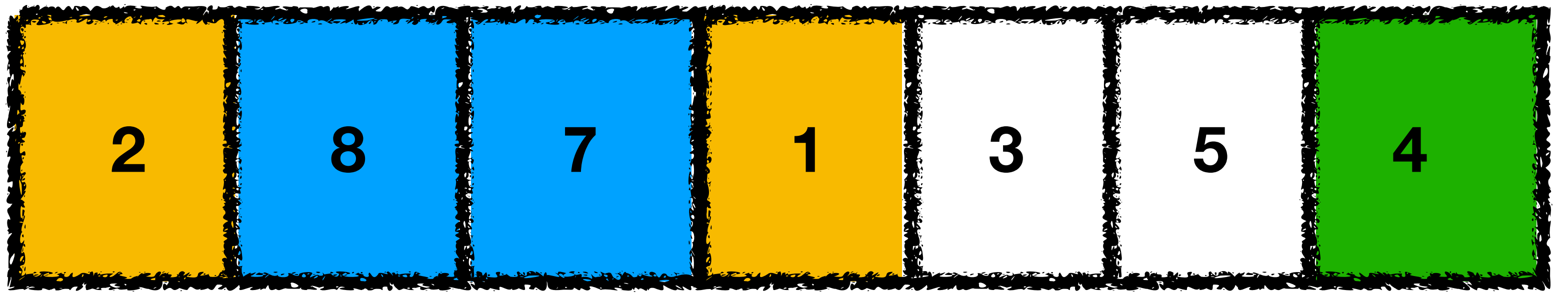
The Quicksort algorithm



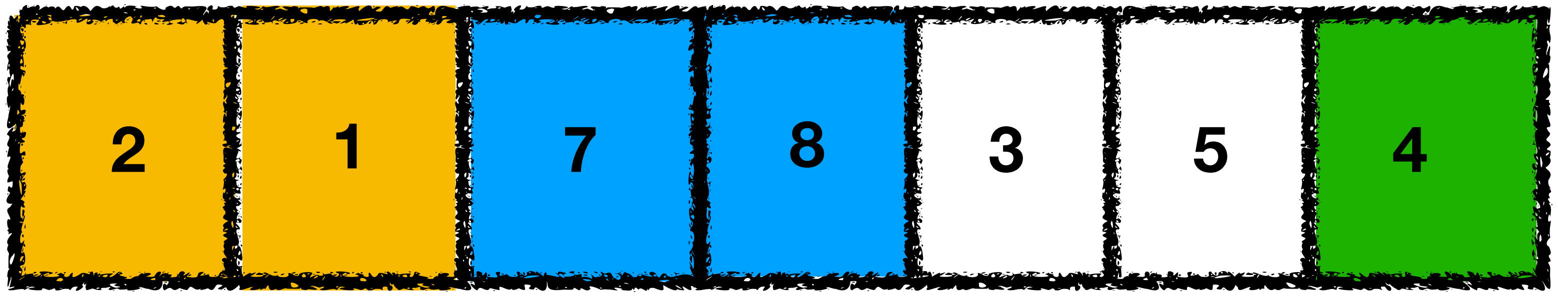
The Quicksort algorithm



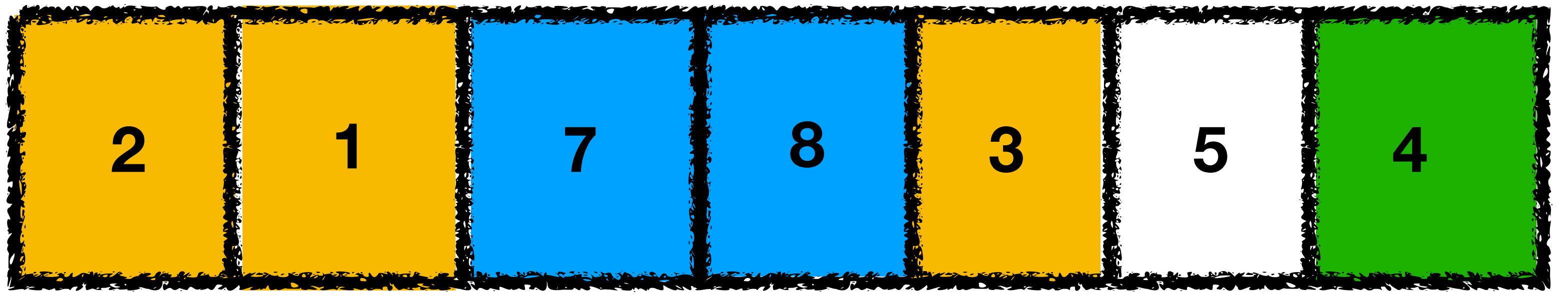
The Quicksort algorithm



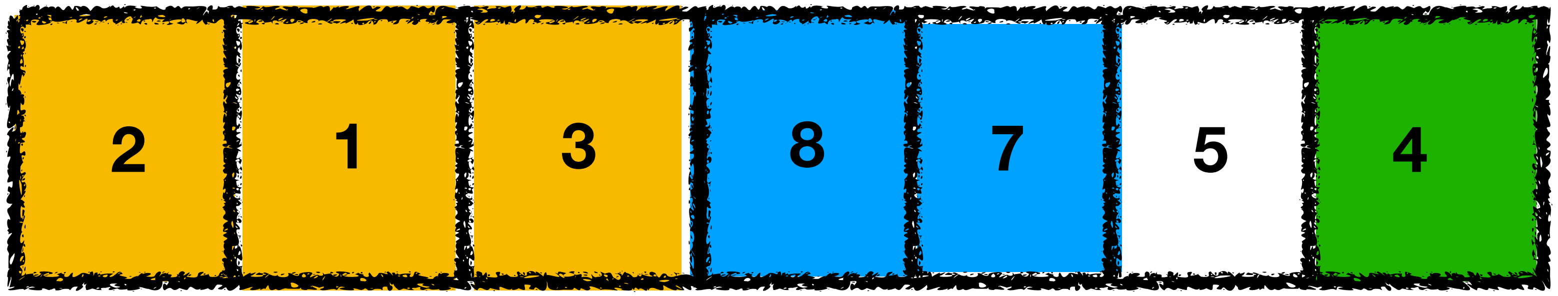
The Quicksort algorithm



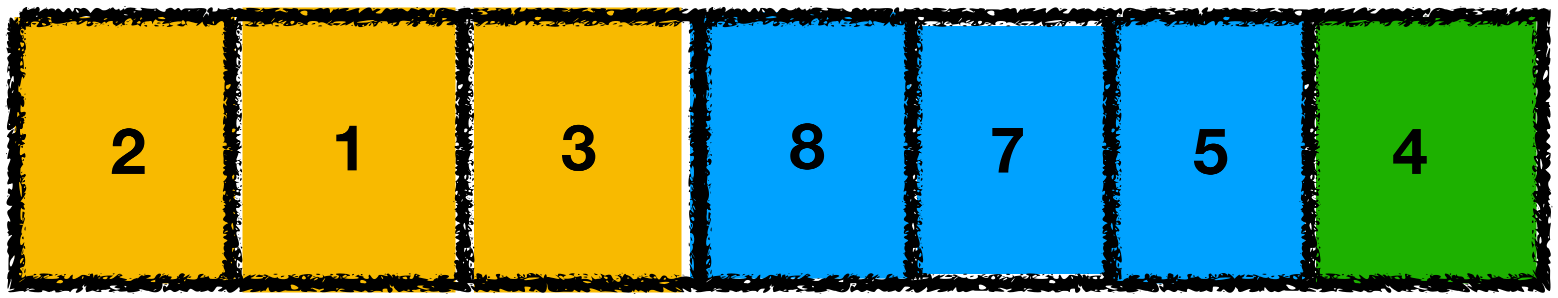
The Quicksort algorithm



The Quicksort algorithm



The Quicksort algorithm

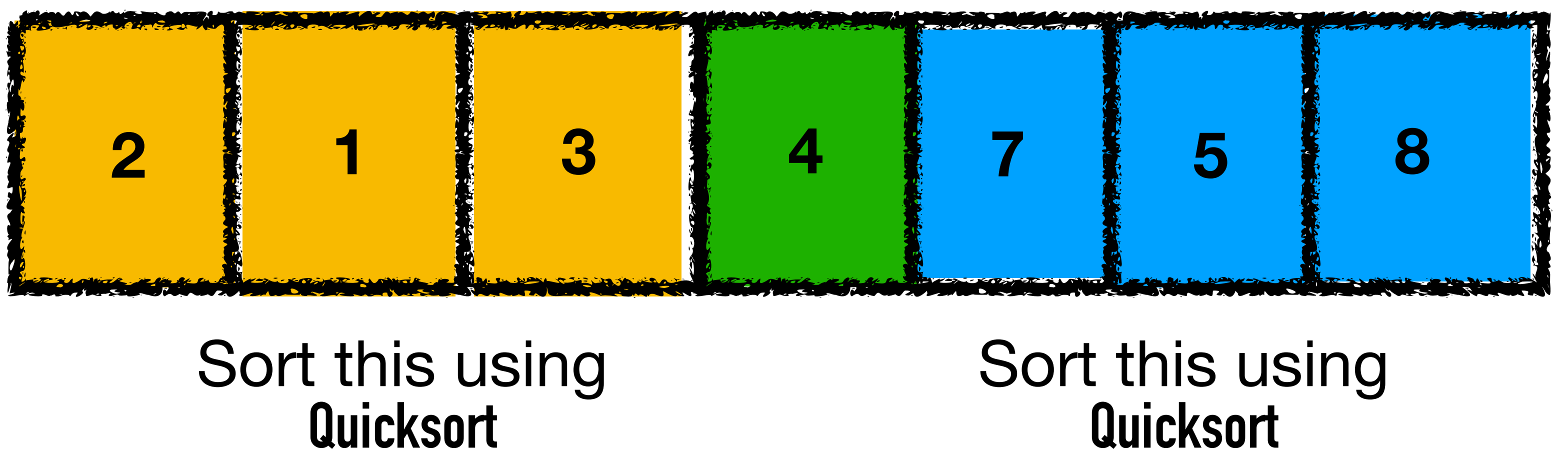


The Quicksort algorithm

The Quicksort algorithm

Sort this using
Quicksort

The Quicksort algorithm



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**[$y+1,\ldots,j$])

Divide-and-Conquer

Divide-and-Conquer

Split the input into smaller sub-instances.

Divide-and-Conquer

Split the input into smaller sub-instances.

Solve each sub-instance separately.

Divide-and-Conquer

Split the input into smaller sub-instances.

Solve each sub-instance separately.

Combine the solutions of the sub-instances into a solution for the problem.

Divide-and-Conquer

Split the input into smaller sub-instances.

Solve each sub-instance separately.

Combine the solutions of the sub-instances into a solution for the problem.

Often: For each sub-instance, the algorithm calls itself to solve it **(recursion).**

The instances become so small that they can be solved via a **brute force** algorithm.

Worst-case Running Times

Worst-case Running Times

What is the worst-case running time of **Mergesort**?

Worst-case Running Times

What is the worst-case running time of **Mergesort**?

$\Theta(n \lg n)$

Worst-case Running Times

What is the worst-case running time of **Mergesort**?

$\Theta(n \lg n)$

What is the worst-case running time of **Quicksort**?

Worst-case Running Times

What is the worst-case running time of **Mergesort**?

$\Theta(n \lg n)$

What is the worst-case running time of **Quicksort**?

$\Theta(n^2)$

Worst-case Running Times

What is the worst-case running time of **Mergesort**?

$\Theta(n \lg n)$

What is the worst-case running time of **Quicksort**?

$\Theta(n^2)$

How do we prove these?

Worst-case Running Times, Upper Bounds

Algorithm **Mergesort**(**A**[$i,\ldots,j$])

If $i=j$, return i

$q=(i+j)/2$

$\mathbf{A}_{\text{left}}$=**Mergesort**(**A**[$i,\ldots,q$])

$\mathbf{A}_{\text{right}}$=**Mergesort**(**A**[$q+1,\ldots,n$])

return **Merge**($\mathbf{A}_{\text{left}}$, $\mathbf{A}_{\text{right}}$)

Worst-case Running Times, Upper Bounds

Algorithm **Mergesort**(**A**[$i,\ldots,j$])

If $i=j$, return i

$q=(i+j)/2$

$\mathbf{A}_{\text{left}}$=**Mergesort**(**A**[$i,\ldots,q$])

$\mathbf{A}_{\text{right}}$=**Mergesort**(**A**[$q+1,\ldots,n$])

return **Merge**($\mathbf{A}_{\text{left}}$, $\mathbf{A}_{\text{right}}$)

Recurrence relation:

Worst-case Running Times, Upper Bounds

Algorithm **Mergesort**(**A**[$i,\ldots,j$])

If i=j, return i

q=(i+j)/2

$\mathbf{A}_{\text{left}}$=**Mergesort**(**A**[$i,\ldots,q$])

$\mathbf{A}_{\text{right}}$=**Mergesort**(**A**[$q+1,\ldots,n$])

return **Merge**($\mathbf{A}_{\text{left}}$, $\mathbf{A}_{\text{right}}$)

Recurrence relation:

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

Worst-case Running Times, Upper Bounds

Algorithm **Mergesort**(**A**[$i,\ldots,j$])

If $i=j$, return i

$q=(i+j)/2$

$\mathbf{A}_{\text{left}}$=**Mergesort**(**A**[$i,\ldots,q$])

$\mathbf{A}_{\text{right}}$=**Mergesort**(**A**[$q+1,\ldots,n$])

return **Merge**($\mathbf{A}_{\text{left}}$, $\mathbf{A}_{\text{right}}$)

Recurrence relation:

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

where $f(n) = O(n)$

Worst-case Running Times, Upper Bounds

Algorithm Mergesort(**A**[$i,\ldots,j$])

If i=j, return i

q=(i+j)/2

A$_{\text{left}}$=Mergesort(**A**[$i,\ldots,q$])

A$_{\text{right}}$=Mergesort(**A**[$q+1,\ldots,n$])

return Merge(**A**$_{\text{left}}$, **A**$_{\text{right}}$)

Recurrence relation:

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

where $f(n) = O(n)$

If we solve the recurrence relation we obtain $T(n) = O(n \lg n)$

Worst-case Running Times, Upper Bounds

Algorithm **Mergesort**(**A**[$i, \ldots, j$])

If $i=j$, return i

$q=(i+j)/2$

$\mathbf{A}_{\text{left}}$=**Mergesort**(**A**[$i, \ldots, q$])

$\mathbf{A}_{\text{right}}$=**Mergesort**(**A**[$q+1, \ldots, n$])

return **Merge**($\mathbf{A}_{\text{left}}$, $\mathbf{A}_{\text{right}}$)

Recurrence relation:

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

where $f(n) = O(n)$

If we solve the recurrence relation we obtain

$T(n) = O(n \lg n)$

(next lecture)