Data Structure and Algorithm

(CSE 225)

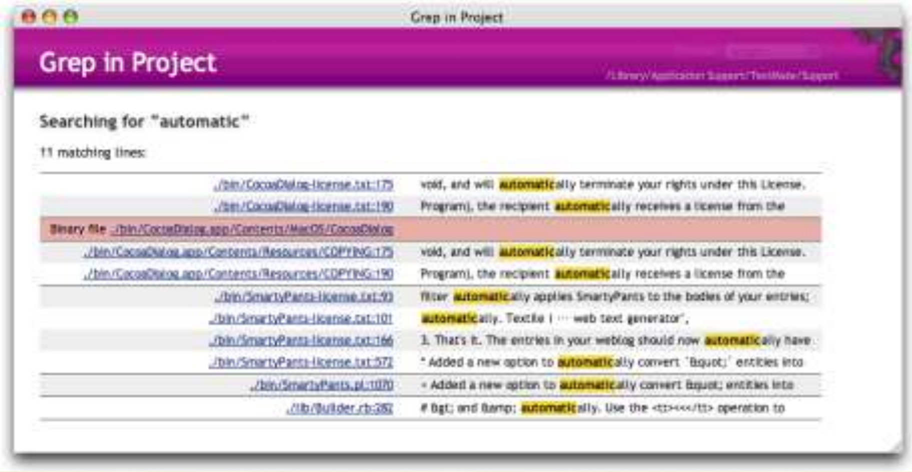
Lecture 20

Sorting and Searching

- Fundamental problems in computer science and programming
- Sorting done to make searching easier
- Multiple different algorithms to solve the same problem
  - How do we know which algorithm is "better"?
- Let review searching first (We have already covered it.)

Searching

recursive backtracking



Searching for "automatic"

11 matching lines:

```
              ./bin/CocoaDialog-license.txt:175    void, and will automatically terminate your rights under this License.
              ./bin/CocoaDialog-license.txt:190    Program), the recipient automatically receives a license from the
Binary file ./bin/CocoaDialog.app/Contents/MacOS/CocoaDialog
./bin/CocoaDialog.app/Contents/Resources/COPYING:175    void, and will automatically terminate your rights under this License.
./bin/CocoaDialog.app/Contents/Resources/COPYING:190    Program), the recipient automatically receives a license from the
              ./bin/SmartyPants-license.txt:93     filter automatically applies SmartyPants to the bodies of your entries;
              ./bin/SmartyPants-license.txt:101    automatically. Textile I ··· web text generator",
              ./bin/SmartyPants-license.txt:166    3. That's it. The entries in your weblog should now automatically have
              ./bin/SmartyPants-license.txt:571    * Added a new option to automatically convert `&quot;` entities into
                   ./bin/SmartyPants.pl:1070       - Added a new option to automatically convert &quot; entities into
                        ./lib/Builder.rb:282       # &gt; and &amp; automatically. Use the <tt><<</tt> operation to
```

Searching

- **Given a list of data find the location of a particular value or report that value is not present**
- **linear search**
  - intuitive approach
  - start at first item
  - is it the one I am looking for?
  - if not go to next item
  - repeat until found or all items checked
- **If items not sorted or unsortable this approach is necessary**

Searching in a Sorted List

- If items are sorted then we can *divide and conquer*
- dividing your work in half with each step
  - generally a good thing
- The Binary Search on List in Ascending order
  - Start at middle of list
  - is that the item?
  - If not is it less than or greater than the item?
  - less than, move to second half of list
  - greater than, move to first half of list
  - repeat until found or sub list size = 0

Sorting

- **A fundamental application for computers**
- **Done to make finding data (searching) faster**
- **Many different algorithms for sorting**
- **One of the difficulties with sorting is working with a fixed size storage container (array)**
  - if resize, that is expensive (slow)
- **The "simple" sorts run in quadratic time O(N^2)**
  - bubble sort
  - selection sort
  - insertion sort

The Problem of Sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion Sort

- Commonly used by card players: As each card is picked up, it is placed into the proper sequence in their hand.

- Divide the list into a sorted sublist and an unsorted sublist.

- In each pass, one or more pieces of data are removed from the unsorted sublist and inserted into their correct position in a sorted sublist.

2
4
5
7
10

Insertion Sort Pseudocode

A[0] unused, valid elements: A[1] … A[n]

Algorithm Name with parameters (like a C function-header)

Comment

```
INSERTION-SORT (A, n)          ▷ A[1 . . n]

for j ← 2 to n
    do ▷ Insert A[ j ] into the sorted suba
       ▷ in such a position that A[1..j] bec
       key ← A[ j]
       i ← j – 1
       while i > 0 and A[i] > key
           do A[i+1] ← A[i]
              i ← i – 1
       A[i+1] ← key
```

Algorithm body

For loop body

While

Loop body

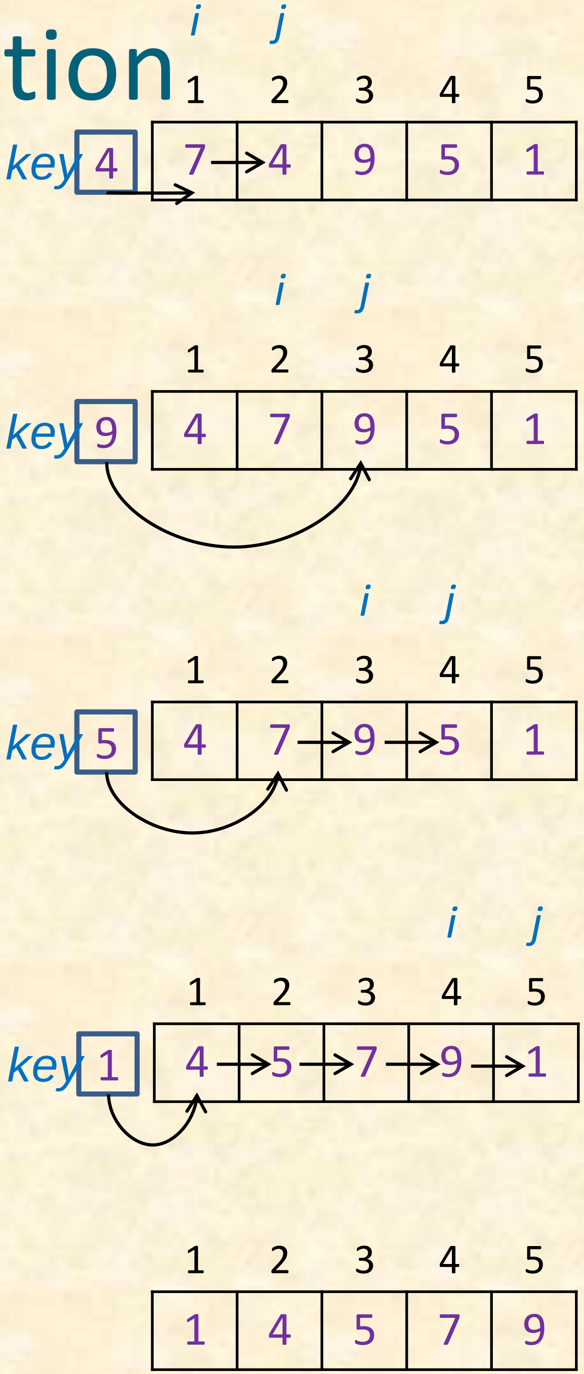
Indentation/spacing determines where the algorithm/loop/if/else-body ends

Equivalent CPP function

```
void insertionSort (int A[], int n)
{   //here A[0 . . n] is an int array
    int i, j;
    for (j = 2; j<=n; j++) {
        key = A[ j];
        i = j – 1;
        while(i > 0 && A[i] > key){
            A[i+1] = A[i];
            i = i – 1;
        }//while
        A[i+1] = key;
    }//for
}
```

Insertion Sort Simulation

```
INSERTION-SORT $(A, n)$        ⊳ $A[1 \,.\,.\, n]$
    for $j \leftarrow 2$ to $n$
        do ⊳ Insert A[ j ] into the sorted subarray A[1..j -1]
           ⊳ in such a position that A[1..j] becomes sorted
           $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] \leftarrow key$
```



i j
1 2 3 4 5
key 4 | 7 4 9 5 1

i j
1 2 3 4 5
key 9 | 4 7 9 5 1

i j
1 2 3 4 5
key 5 | 4 7 9 5 1

i j
1 2 3 4 5
key 1 | 4 5 7 9 1

1 2 3 4 5
1 4 5 7 9

Insertion Sort

5	1	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort

5	1	3	4	6	2

🟨	Comparison
🟩	Data Movement
🟦	Sorted

Insertion Sort

5	1	3	4	6	2

🟨	Comparison
🟩	Data Movement
🟦	Sorted

Insertion Sort

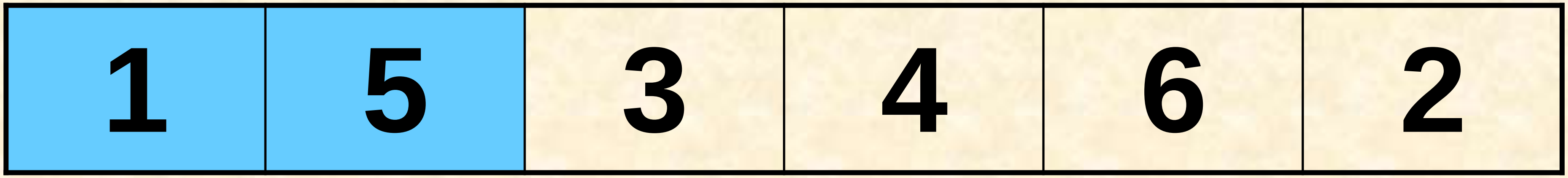
1	5	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



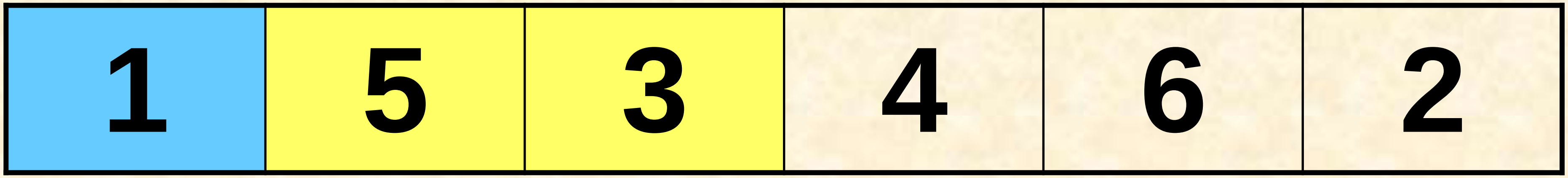
1	5	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



1	5	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort

1	3	5	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort

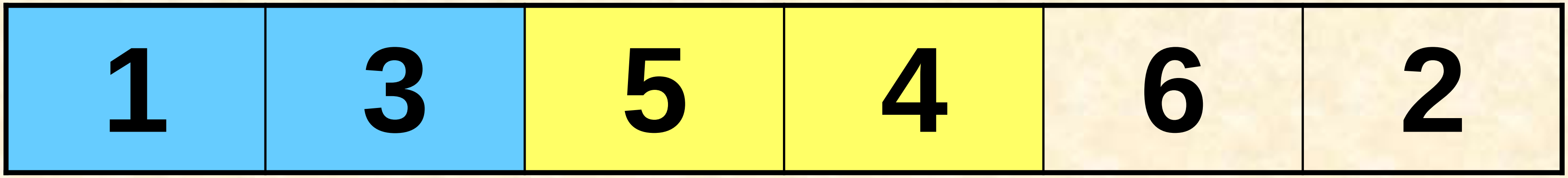
1	3	5	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



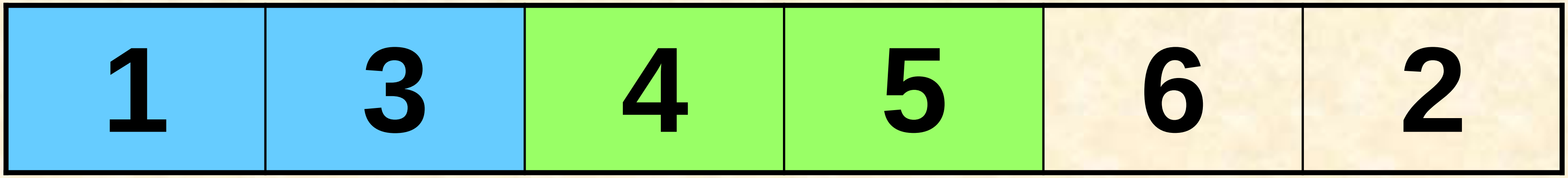
1	3	5	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



1	3	4	5	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort

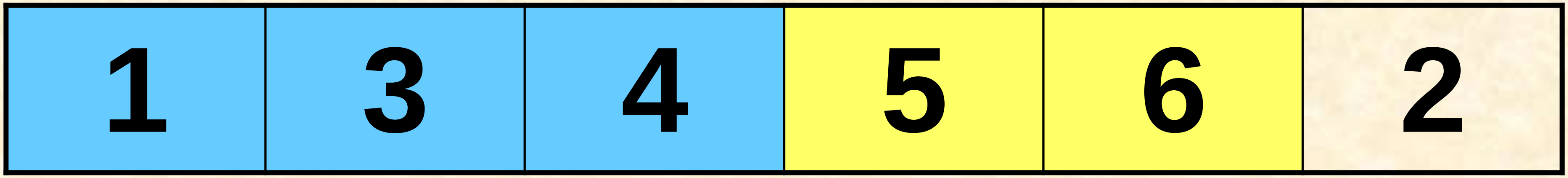
1	3	4	5	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



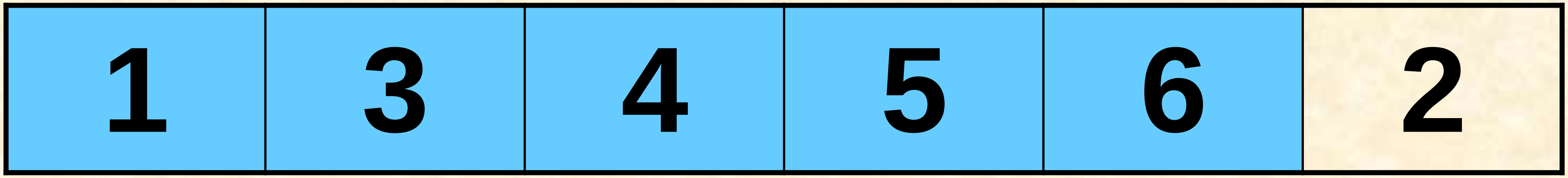
1	3	4	5	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



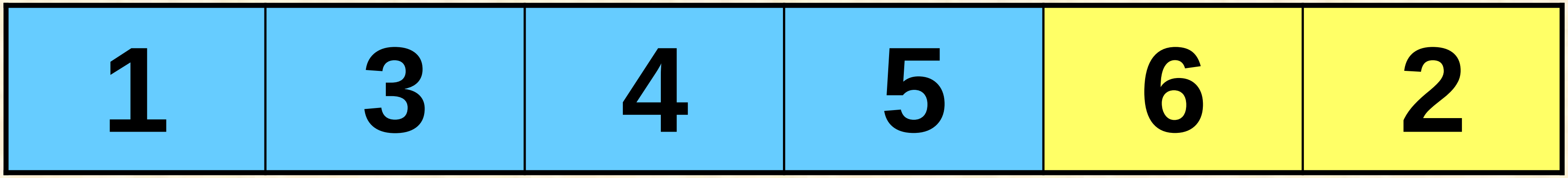
1	3	4	5	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



1	3	4	5	6	2

🟨 Comparison

🟩 Data Movement

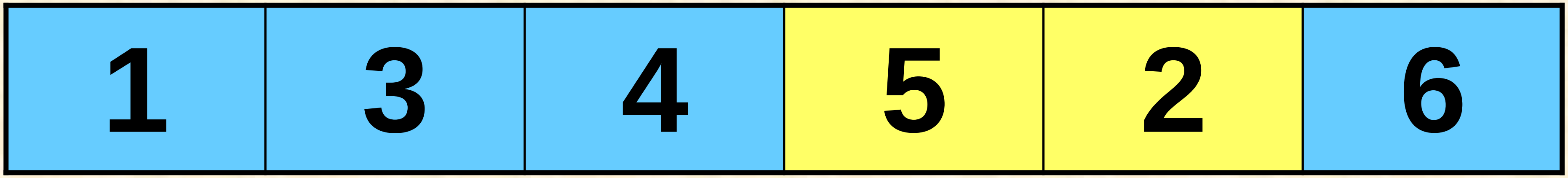
🟦 Sorted

Insertion Sort

1	3	4	5	2	6

🟨	Comparison
🟩	Data Movement
🟦	Sorted

Insertion Sort



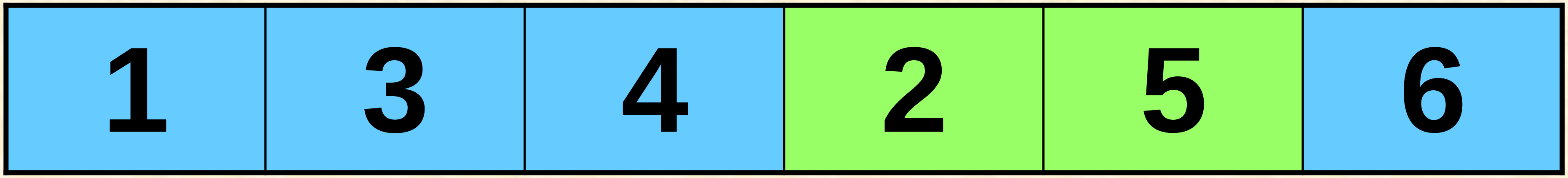
1	3	4	5	2	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



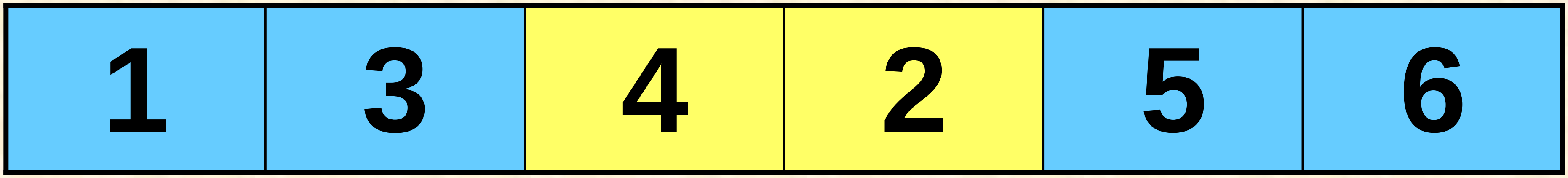
1	3	4	2	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



1	3	4	2	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort

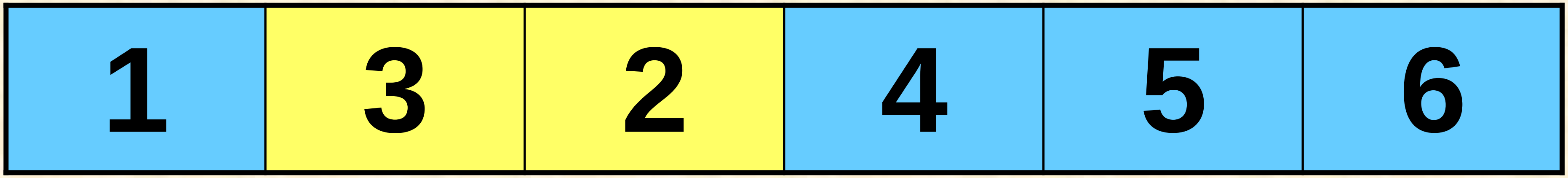
🟦 1	🟦 3	🟩 2	🟩 4	🟦 5	🟦 6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



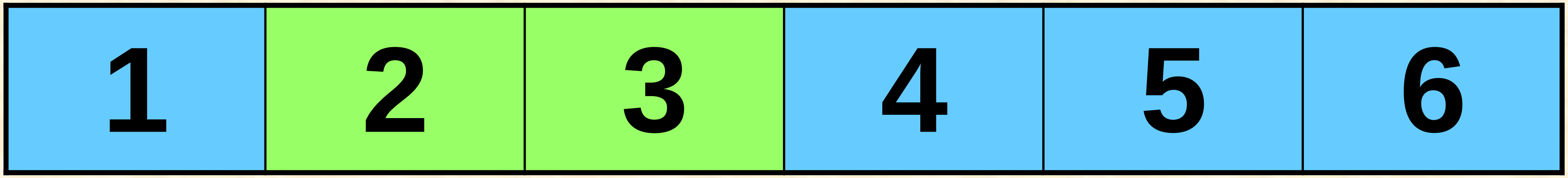
1	3	2	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort



1	2	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Insertion Sort

1	2	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Sorting Fun
Why Not Bubble Sort?

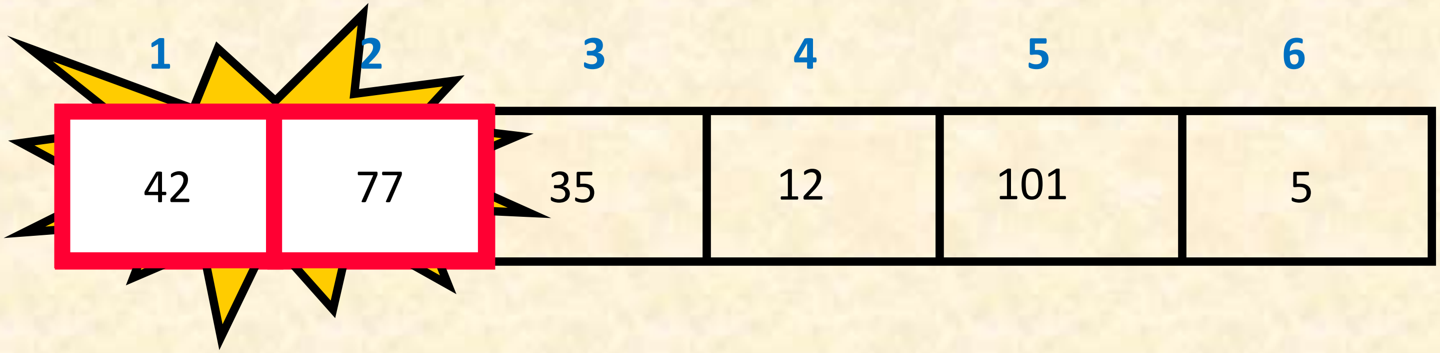
"Bubbling Up" the Largest Element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
77	42	35	12	101	5

"Bubbling Up" the Largest Element

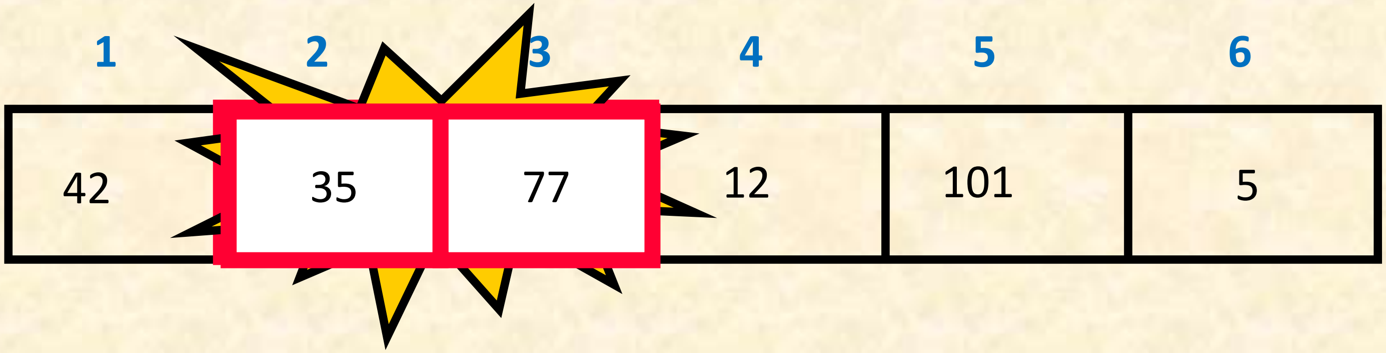
- Traverse a collection of elements
  - Move from the front to the end
  - “Bubble” the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	77	35	12	101	5

"Bubbling Up" the Largest Element

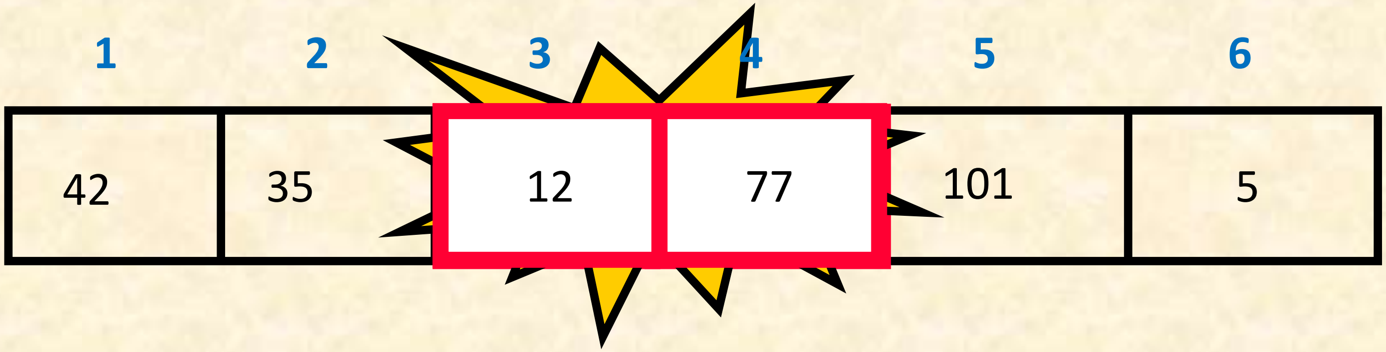
- `Traverse a collection of elements`
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	77	12	101	5

"Bubbling Up" the Largest Element

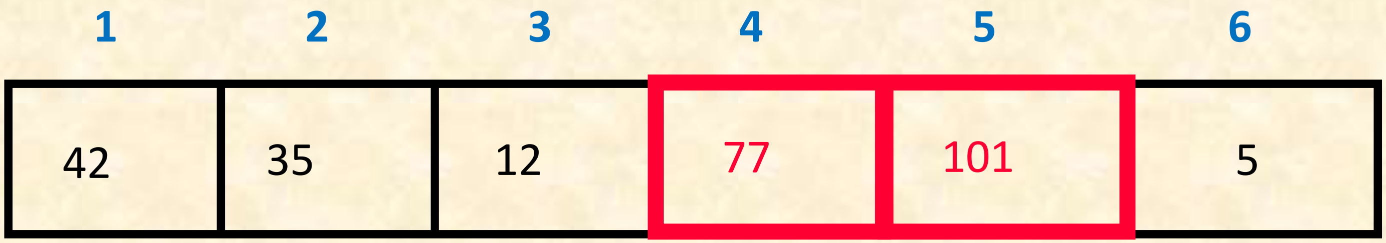
- `Traverse a collection of elements`
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	12	77	101	5

"Bubbling Up" the Largest Element

- `Traverse a collection of elements`
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

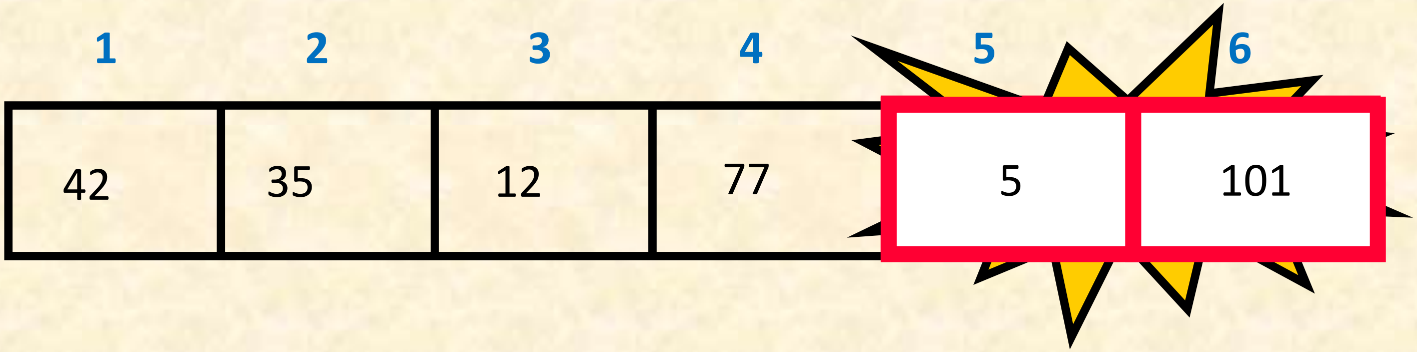


1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling Up" the Largest Element

- `Traverse a collection of elements`
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



1	2	3	4	5	6
42	35	12	77	5	101

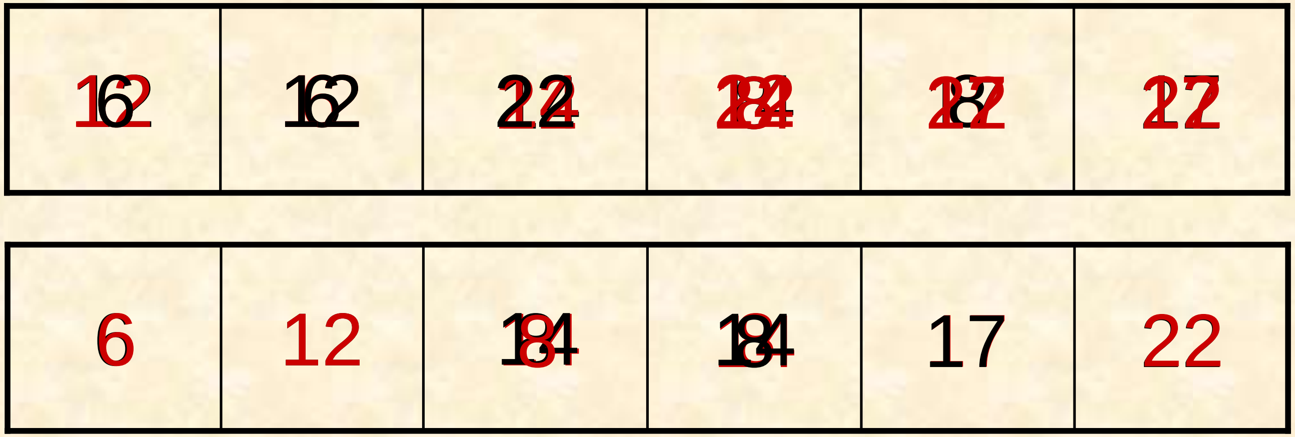
"Bubbling Up" the Largest Element

- **`Traverse a collection of elements`**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Bubble Sort



- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort

6	12 8	8 12	14	17	22

6	8	12	14	17	22

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort

- **Algorithm:** (Bubble Sort) BUBBLE (DATA, N)

  Here DATA is an Array with N elements. This algorithm sorts the elements in DATA.

```
1.  Repeat Steps 2 and 3 for K = 1 to N-1
2.      Set PTR: =1
3.      Repeat while PTR<=N-K
            (a) If DATA[PTR]>DATA[PTR+1], then:
                    Interchange DATA[PTR] and DATA[PTR+1]
                [End of if structure]
            (b) Set PTR: =PTR+1
        [End of inner loop]
    [End of Step 1 Outer loop]
4.  Exit
```

Bubble Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them

- How efficient is bubble sort?
  - In general, given n numbers to sort, it performs n^2 comparisons
  - The same as selection sort

- Is there a simple way to improve on the basic bubble sort?
  - Yes! Stop after going through without making any swaps
  - This will only help some of the time

Bubble Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them

Try one!

15	3	11	19	4	7

Selection Sort

5	1	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	6	2

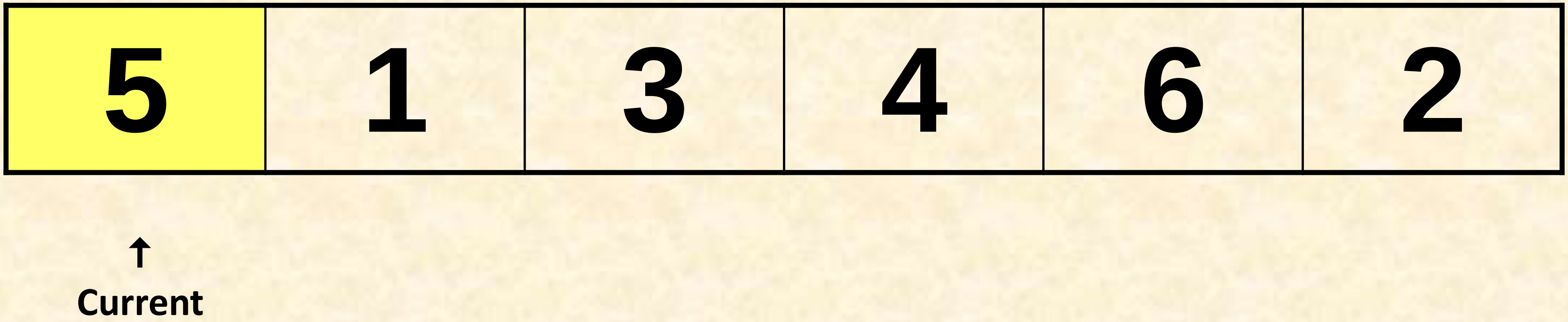
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



5	1	3	4	6	2

↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	6	2
↑ **Current**					

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	🟨 6	2

↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

5	1	3	4	6	2

↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

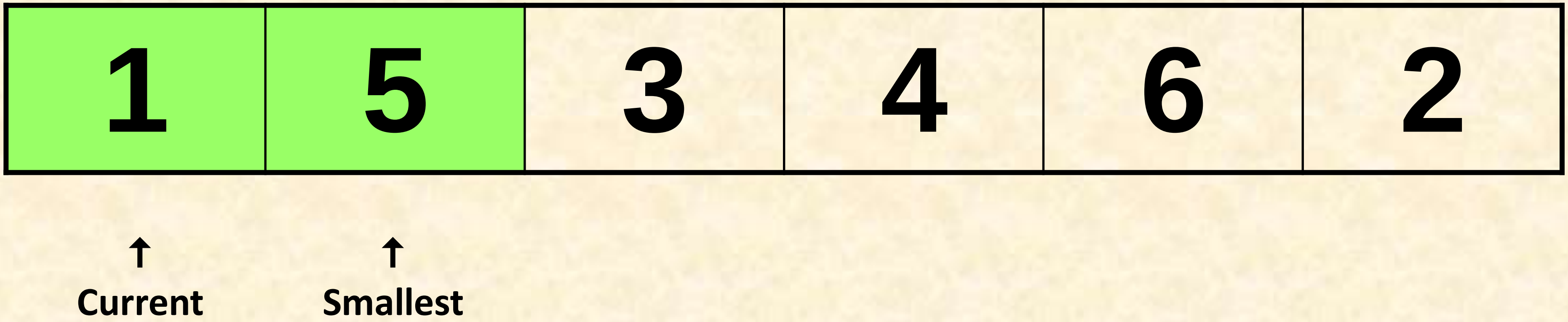
5	1	3	4	6	2
↑ Current	↑ Smallest				

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	5	3	4	6	2

↑ Current

↑ Smallest

🟨 Comparison

🟩 Data Movement

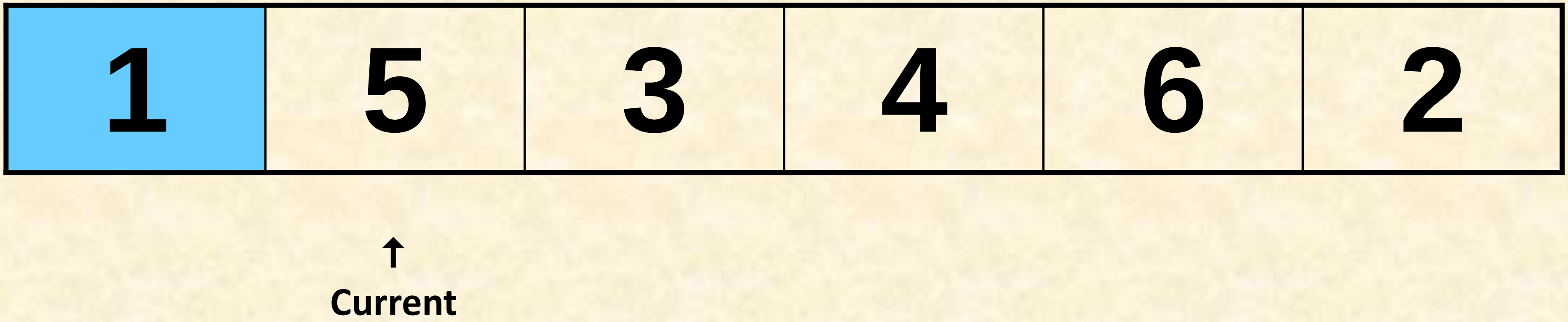
🟦 Sorted

Selection Sort

1	5	3	4	6	2

🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort



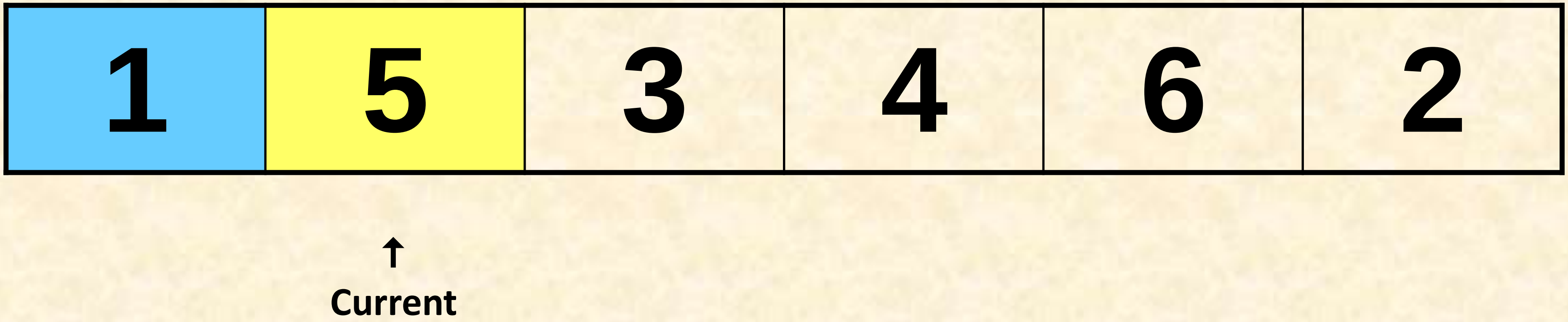
1	5	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	5	3	4	6	2

↑
Current

🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort

1	5	3	4	6	2

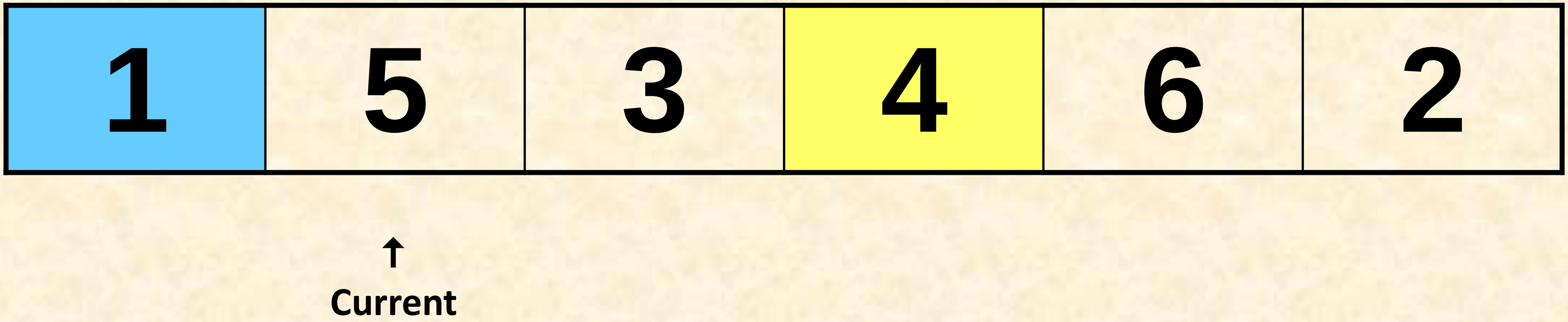
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



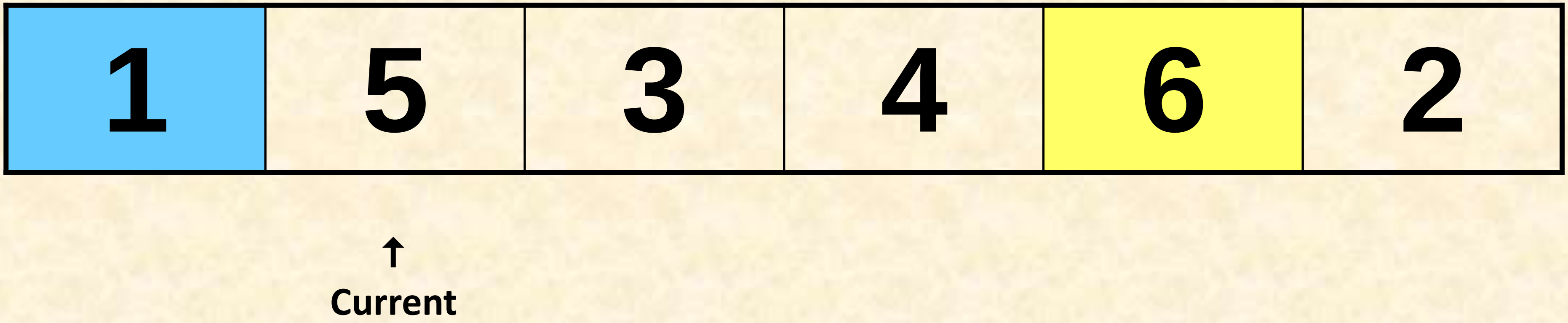
1	5	3	4	6	2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	5	3	4	6	2

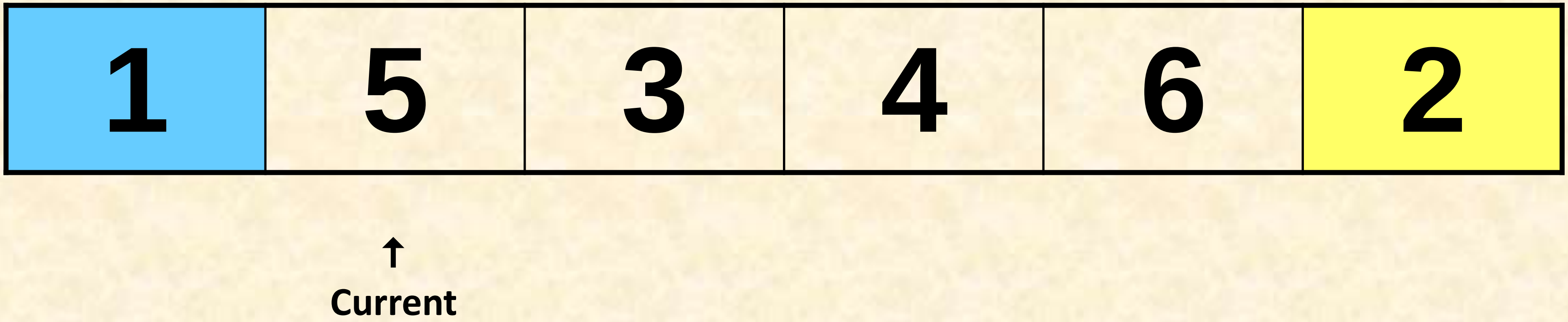
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	5	3	4	6	2

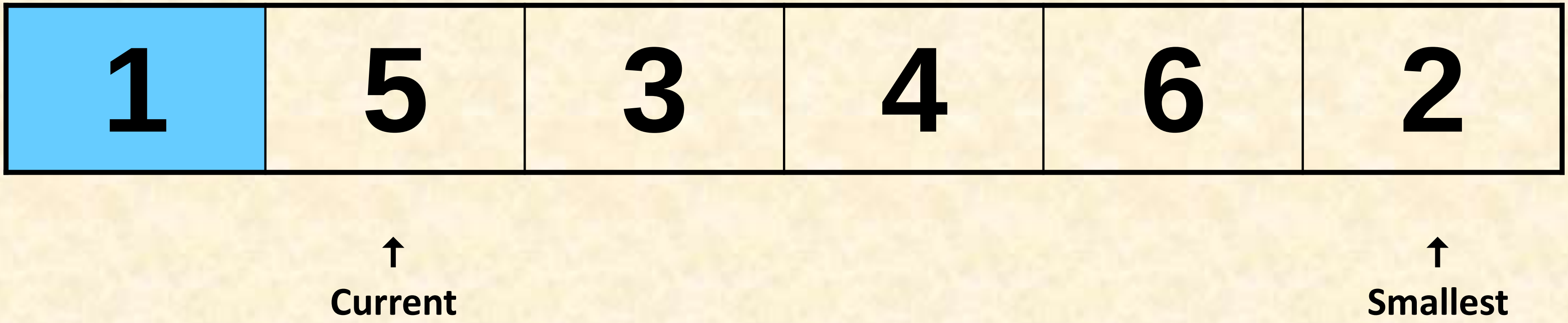
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

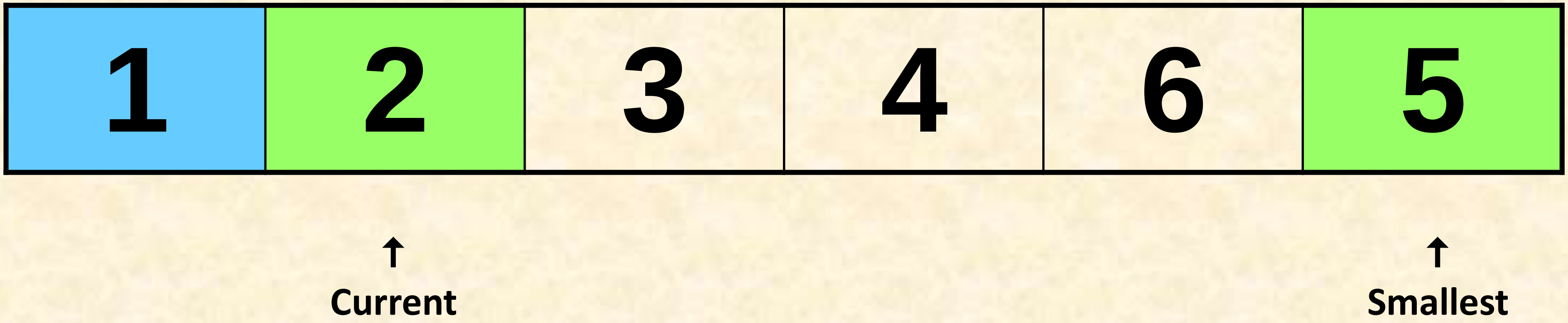
Selection Sort



1	5	3	4	6	2
	↑ Current				↑ Smallest

- 🟨 Comparison
- 🟩 Data Movement
- 🟦 Sorted

Selection Sort



🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

1	2	3	4	6	5

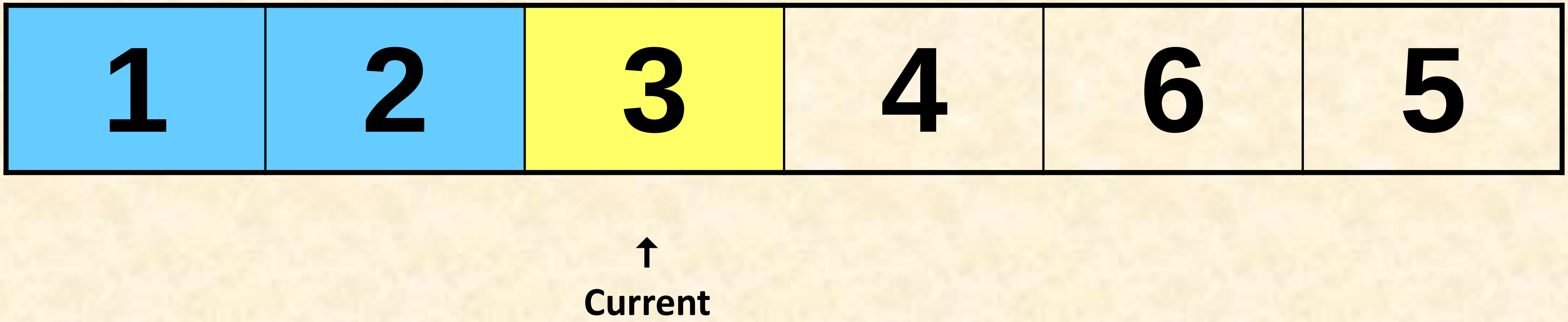
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

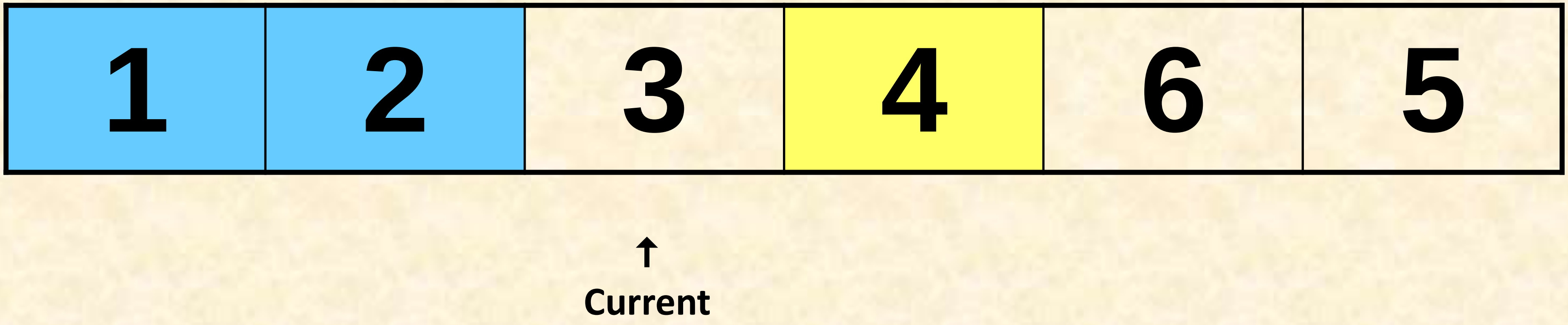
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



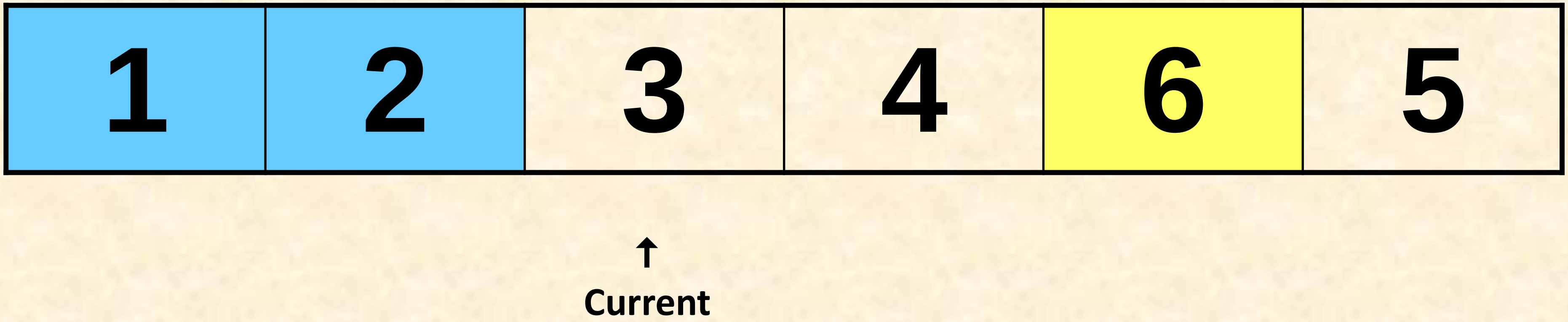
1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

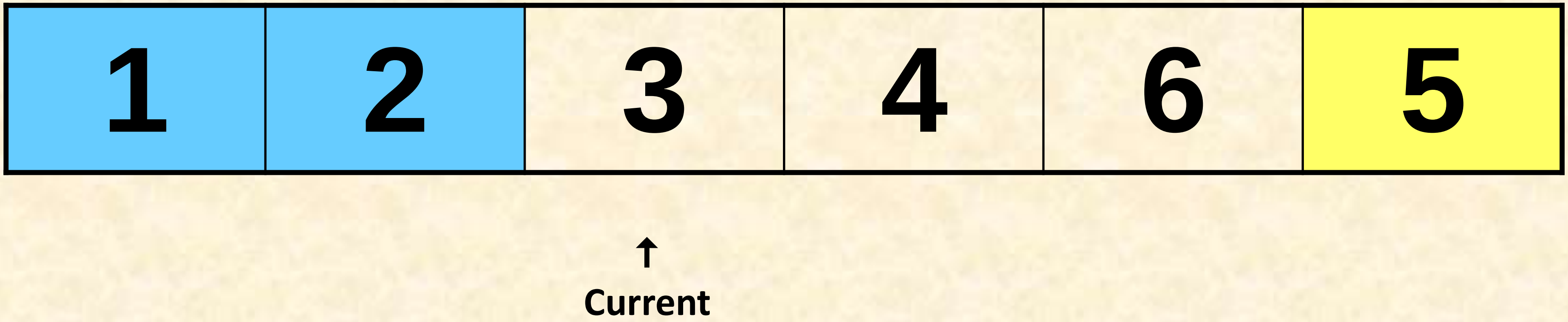
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

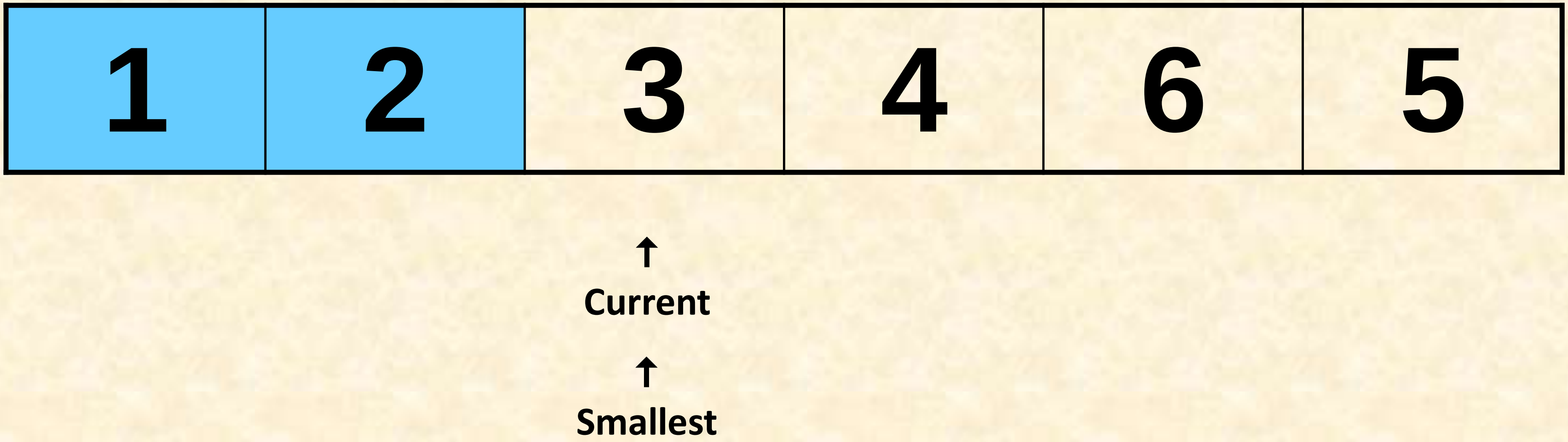
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

↑
Current

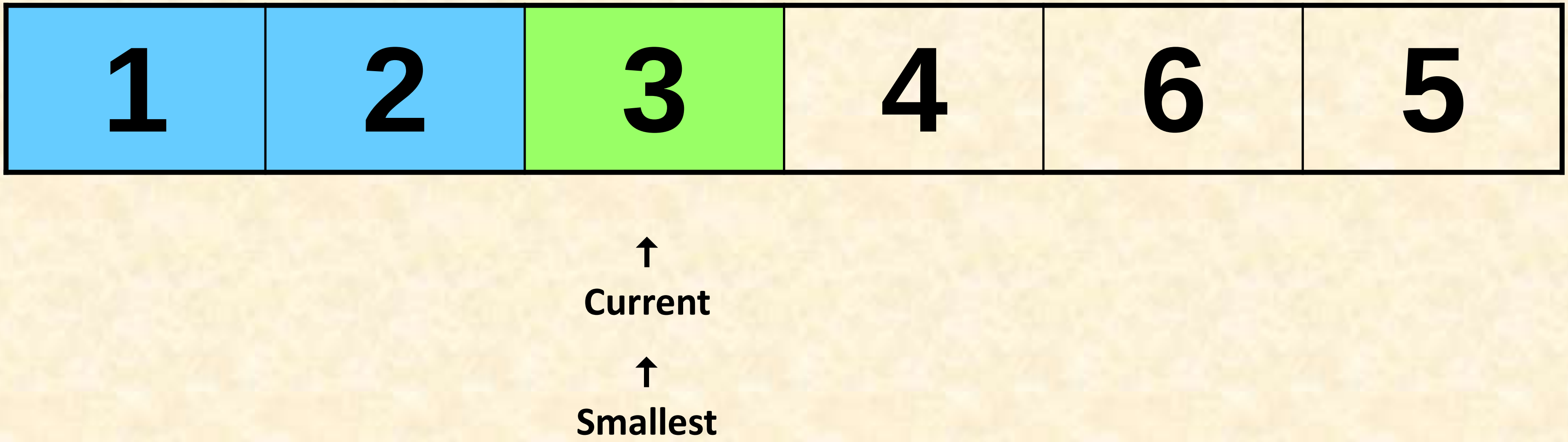
↑
Smallest

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

↑
Current

↑
Smallest

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

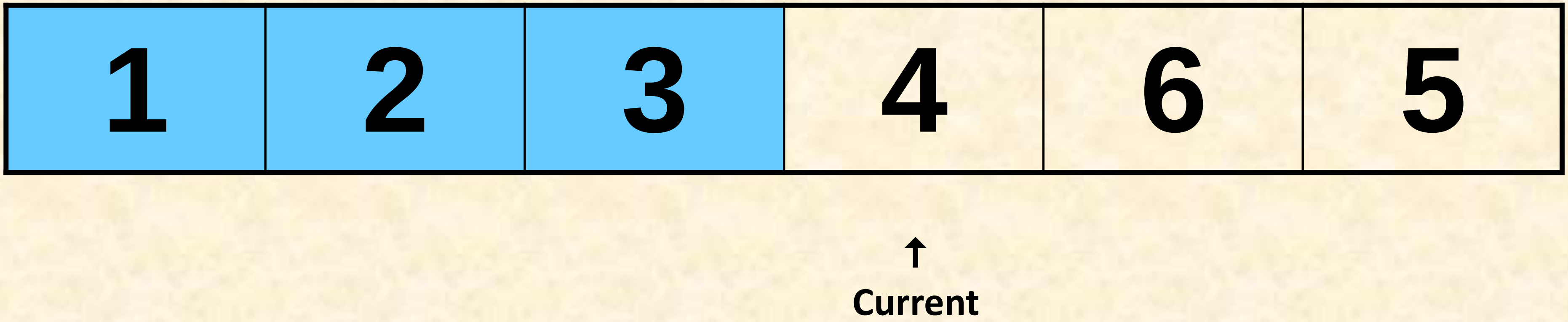
1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

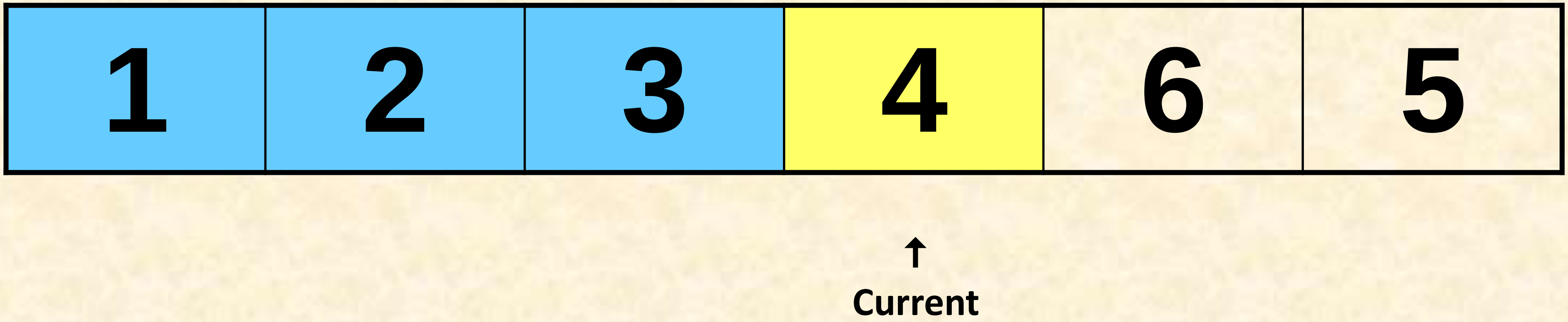
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

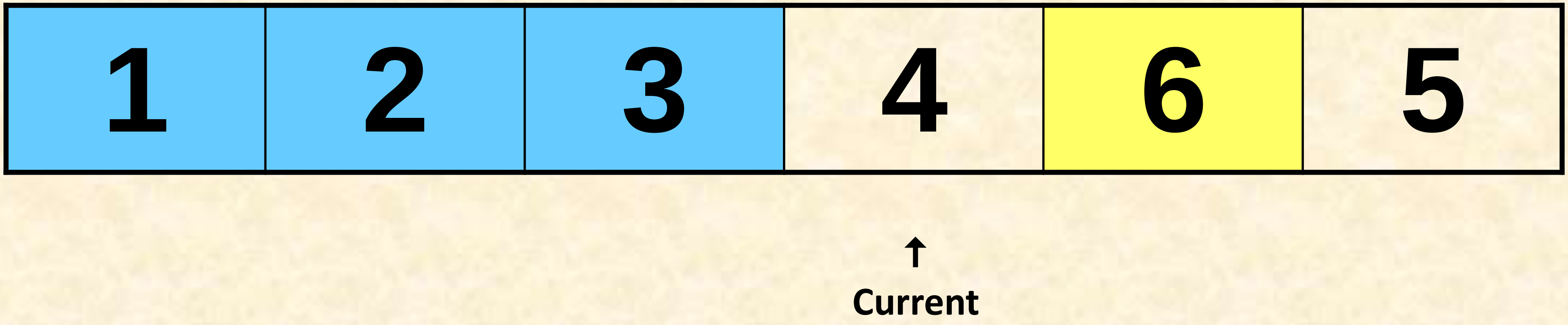


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



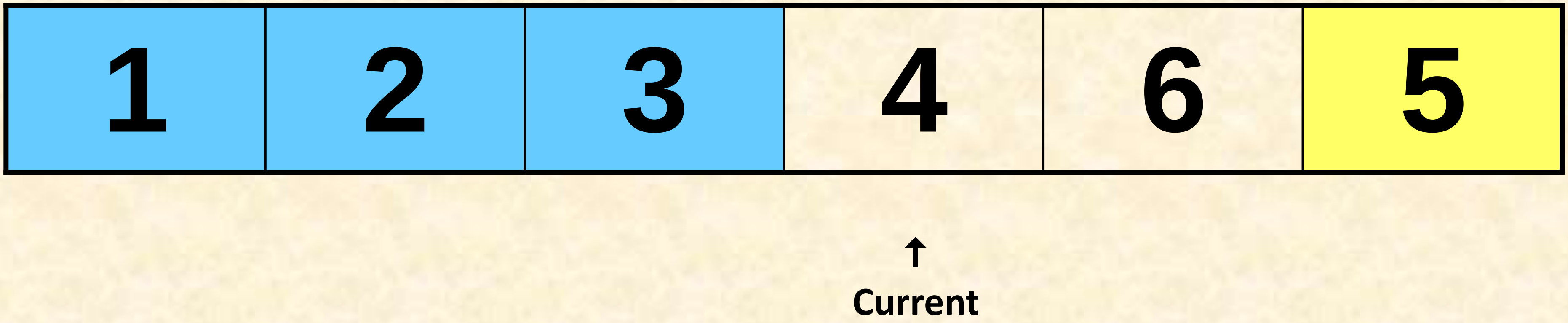
1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1 2 3 4 6 5

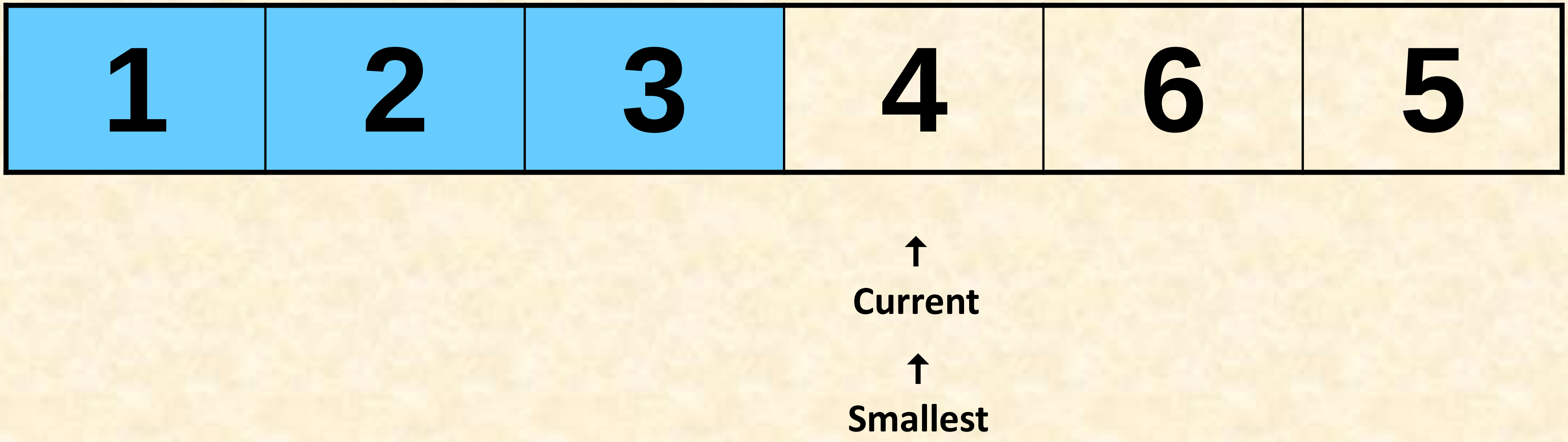
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



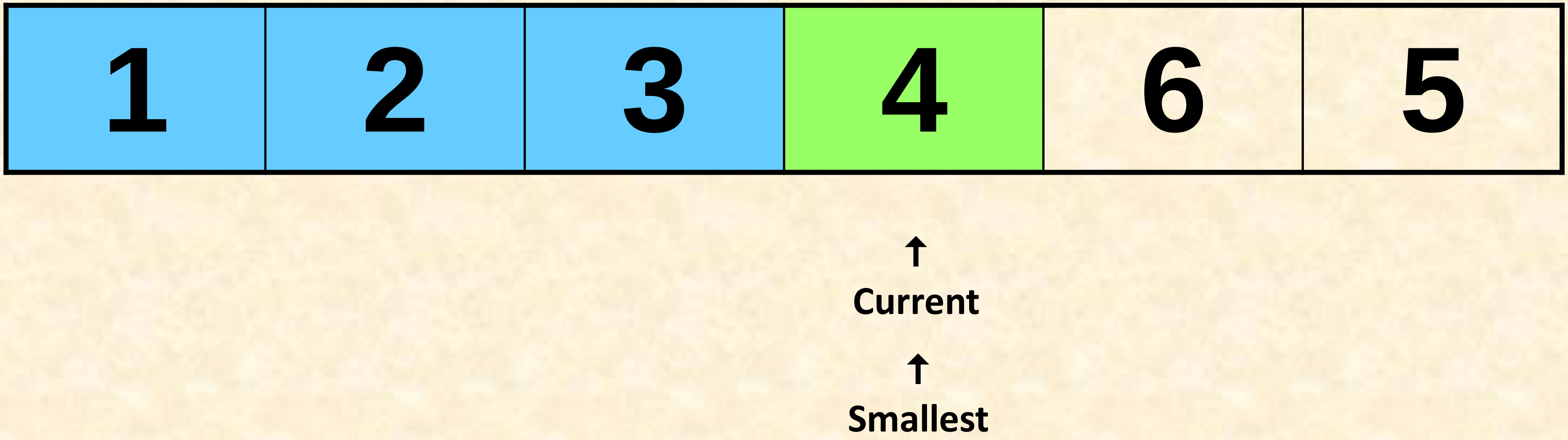
1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

↑
Current

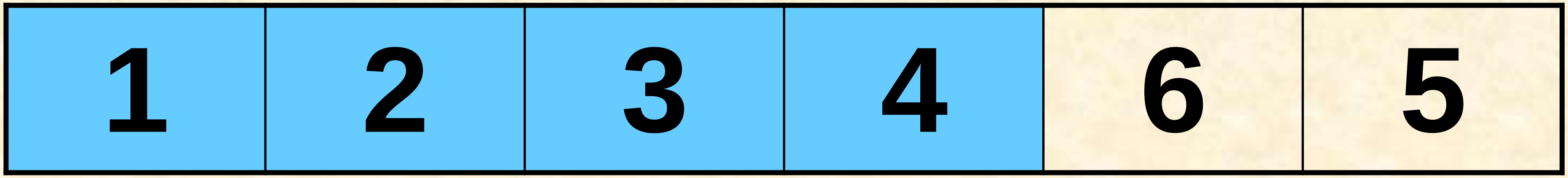
↑
Smallest

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



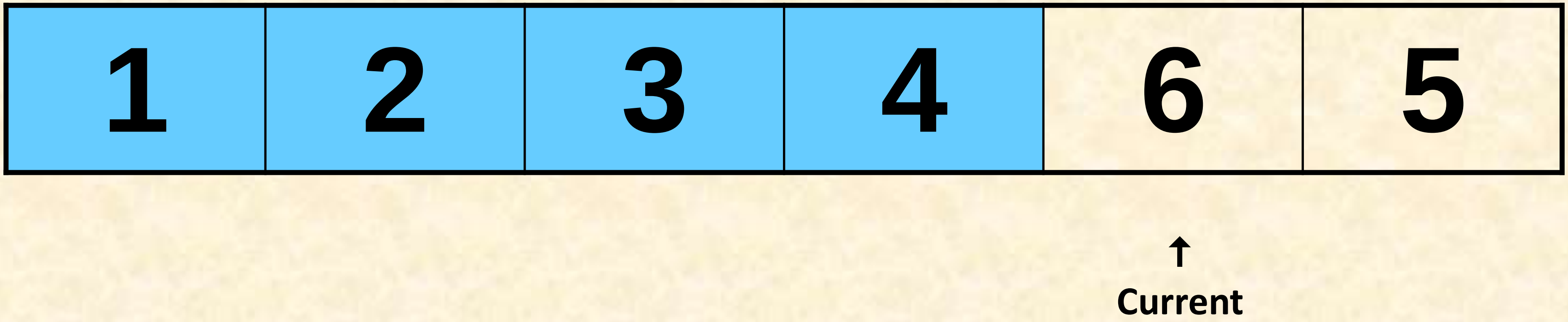
1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



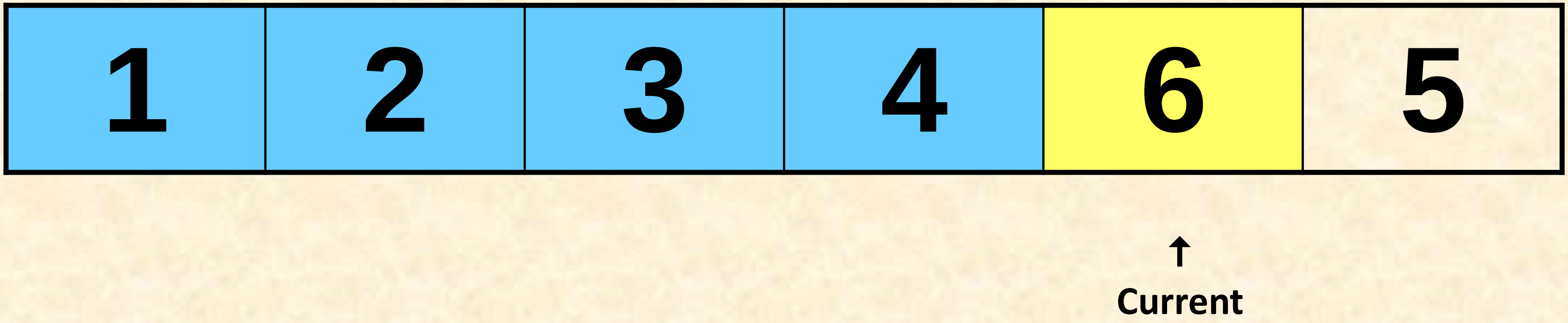
1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

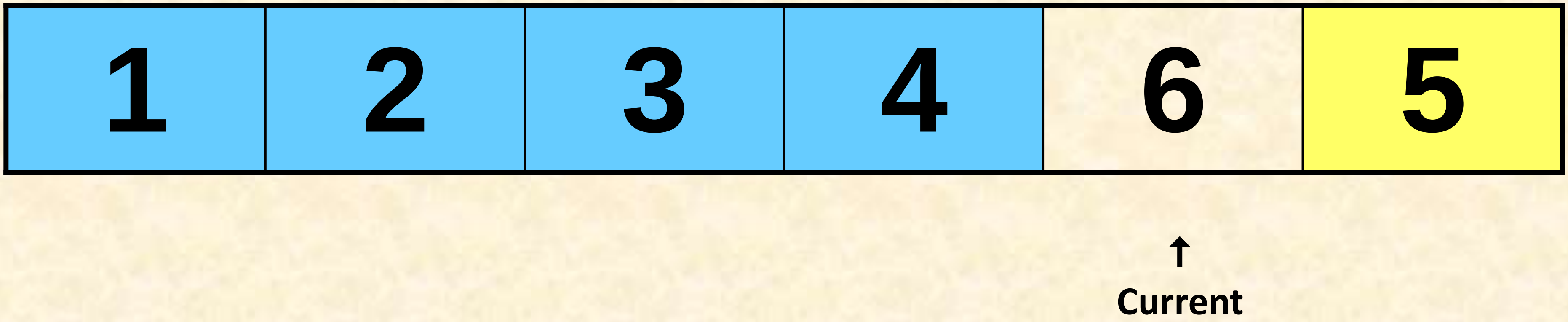
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	6	5

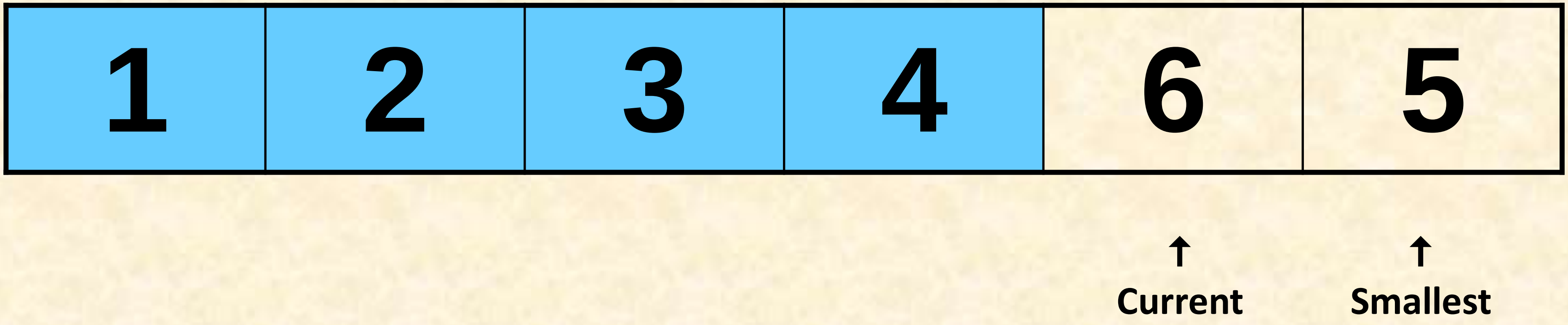
↑
Current

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



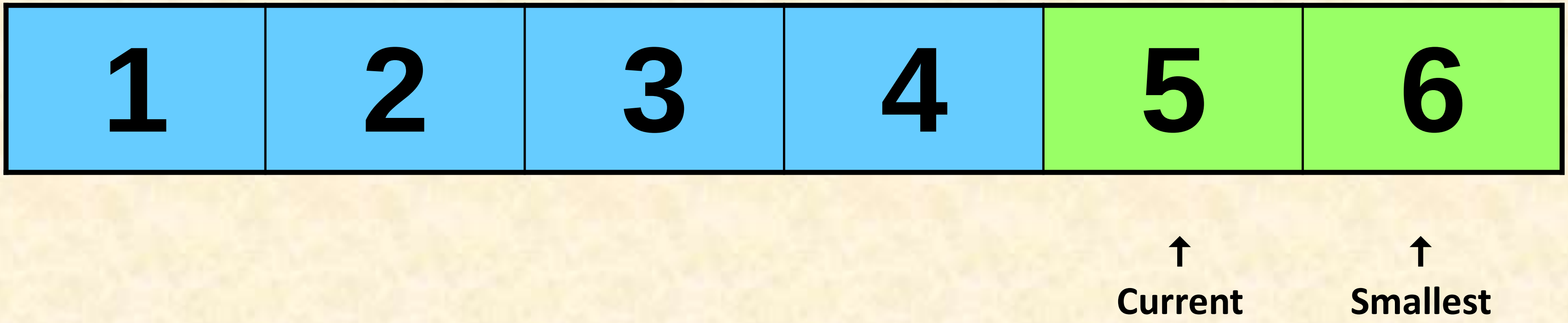
1	2	3	4	6	5

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



1	2	3	4	5	6

- 🟨 Comparison
- 🟩 Data Movement
- 🟦 Sorted

Selection Sort

1	2	3	4	5	6

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

```
template<class ItemType>
int getMinIndex(ItemType values[], int startIndex, int endIndex)
{
        int indexOfMin = startIndex;
        for (int index = startIndex + 1; index <= endIndex; index++)
                if (values[index] < values[indexOfMin])
                        indexOfMin = index;
        return indexOfMin;
}
template<class ItemType>
void SelectionSort(ItemType values[], int numValues)
{
        int endIndex = numValues-1;
        int minIndex;
        for (int current = 0; current < endIndex; current++)
        {
                minIndex = getMinIndex(values, current, endIndex);
                Swap(values[current], values[minIndex]);
        }
}
```

Selection Sort

```
template<class ItemType>
int getMinIndex(ItemType values[], int startIndex, int endIndex)
{
	int indexOfMin = startIndex;
	for (int index = startIndex + 1; index <= endIndex; index++)
		if (values[index] < values[indexOfMin])
			indexOfMin = index;
	return indexOfMin;
}
template<class ItemType>
void SelectionSort(ItemType values[], int numValues)
{
	int endIndex = numValues-1;
	int minIndex;
	for (int current = 0; current < endIndex; current++)
	{
		minIndex = getMinIndex(values, current, endIndex);
		Swap(values[current], values[minIndex]);
	}
}
```

O(N)

O(N)

$O(N^2)$

Divide and Conquer

- **Recursive in structure**
  - ***Divide*** the problem into independent sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - This can be done by reducing the problem until it reaches the base case, which is the solution.
  - ***Combine*** the solutions of the sub-problems to create a solution to the original problem

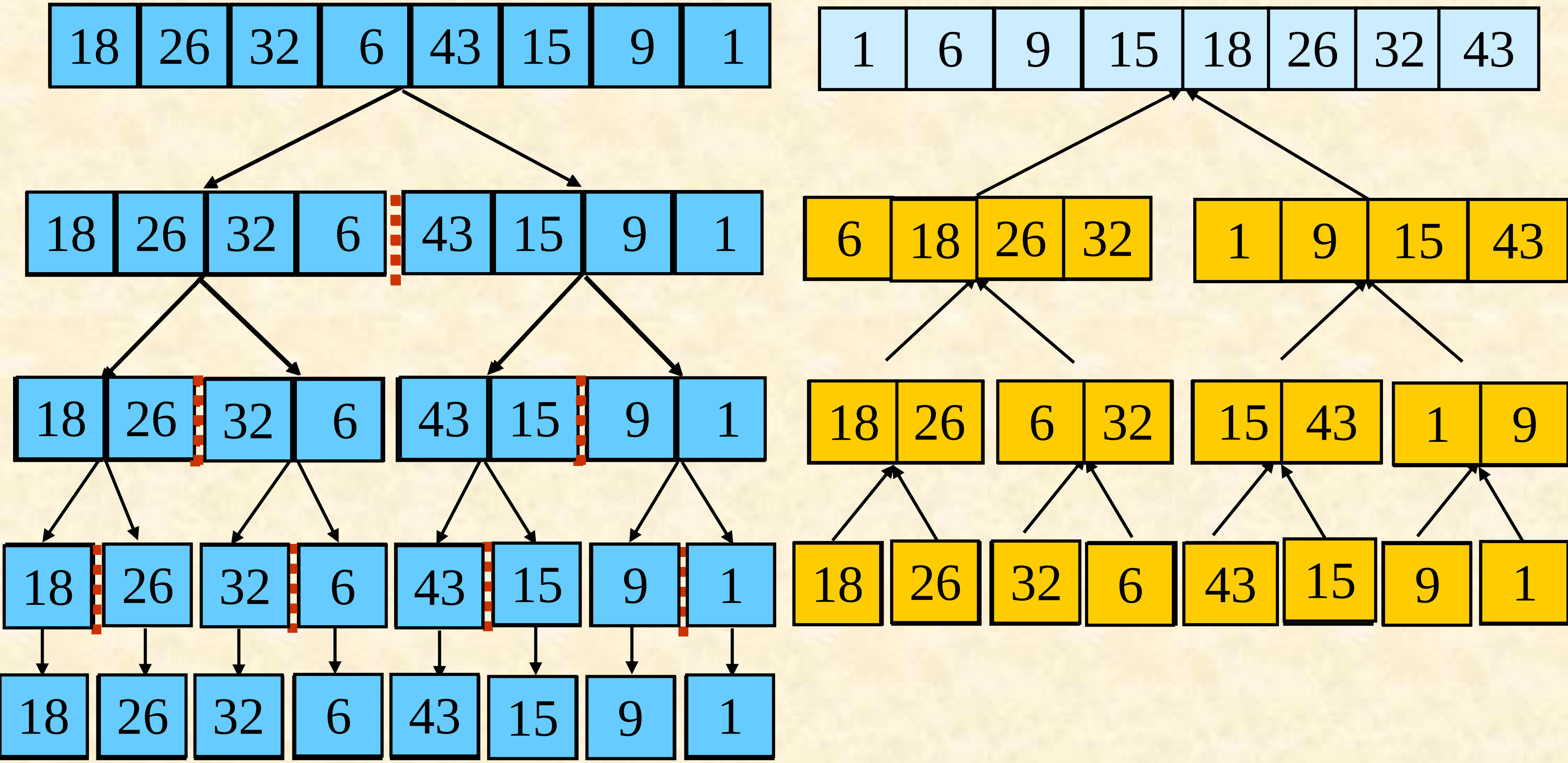
Example: Merge Sort

Sorting Problem: Sort a sequence of *n* elements into non-decreasing order.

- *Divide:* Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- *Conquer:* Sort the two subsequences recursively using merge sort.
- *Combine:* Merge the two sorted subsequences to produce the sorted answer.

Original Sequence

Sorted Sequence



18 26 32 6 43 15 9 1

18 26 32 6 | 43 15 9 1

18 26 | 32 6 | 43 15 | 9 1

18 | 26 | 32 | 6 | 43 | 15 | 9 | 1

18 26 32 6 43 15 9 1

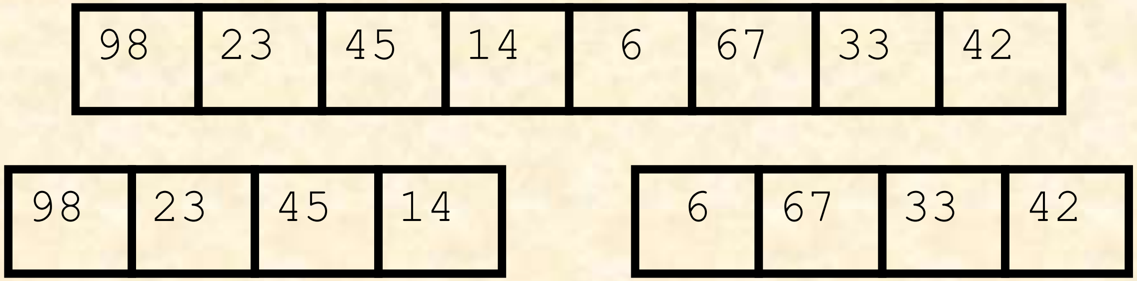
1 6 9 15 18 26 32 43

6 18 26 32 | 1 9 15 43

18 26 | 6 32 | 15 43 | 1 9

18 26 32 6 43 15 9 1

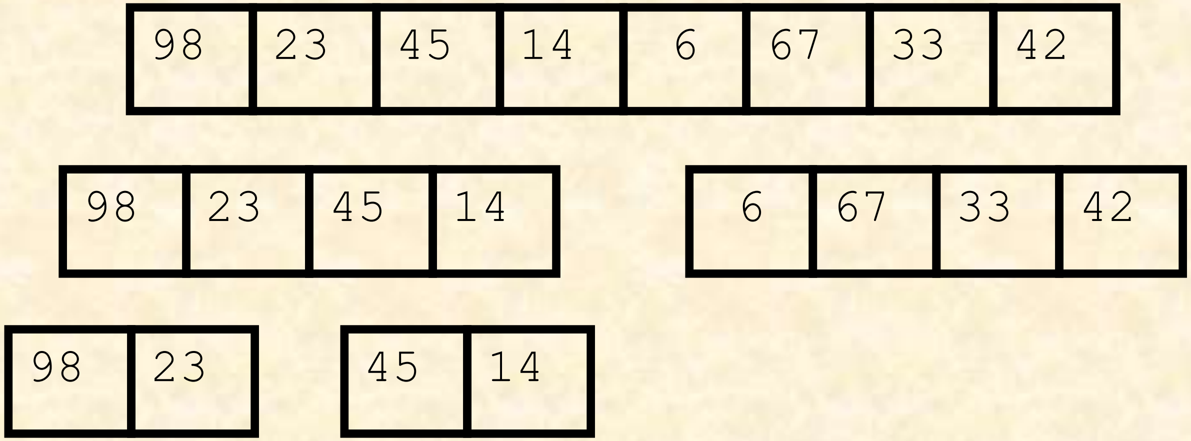
98	23	45	14	6	67	33	42

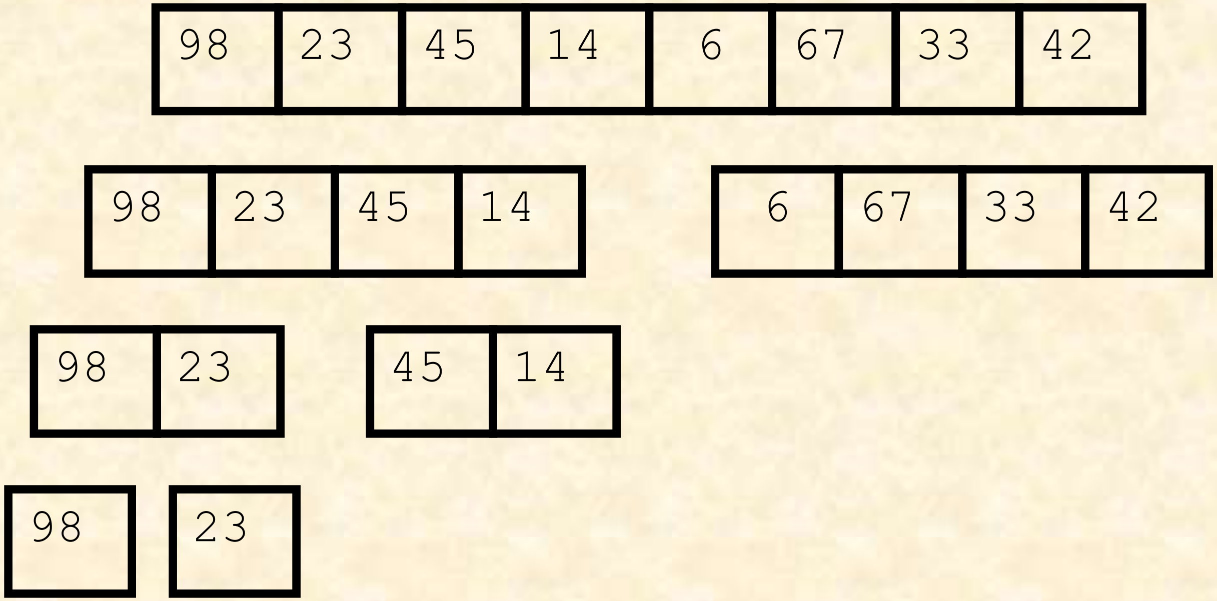


98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42


98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14



98 23 45 14 6 67 33 42

98 23 45 14

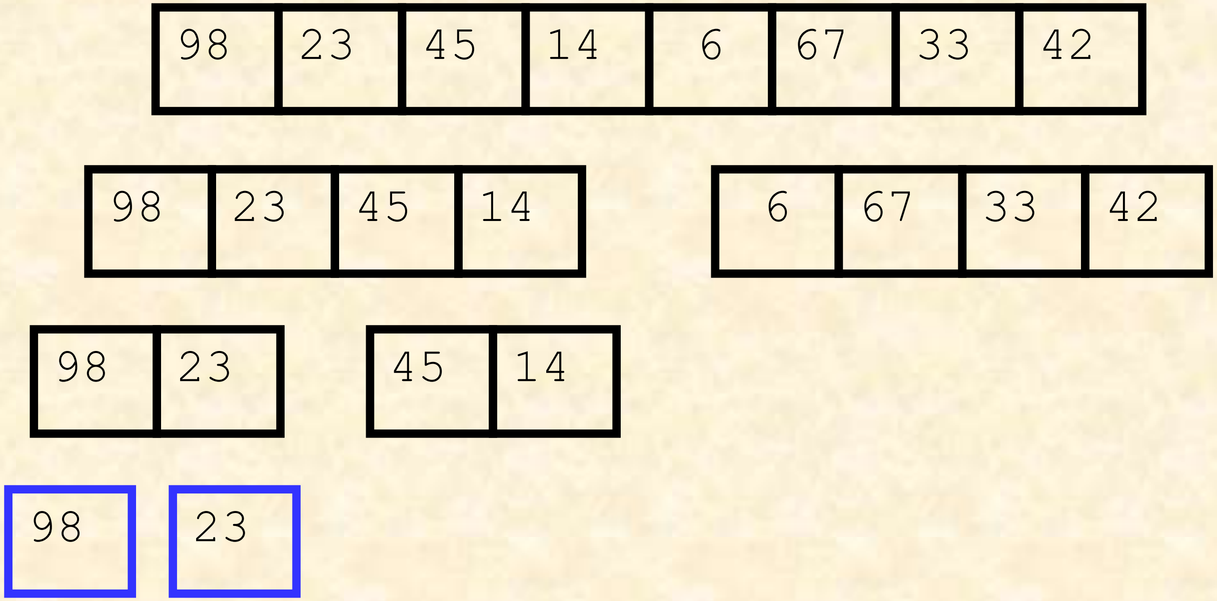
6 67 33 42

98 23

45 14

98

23



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

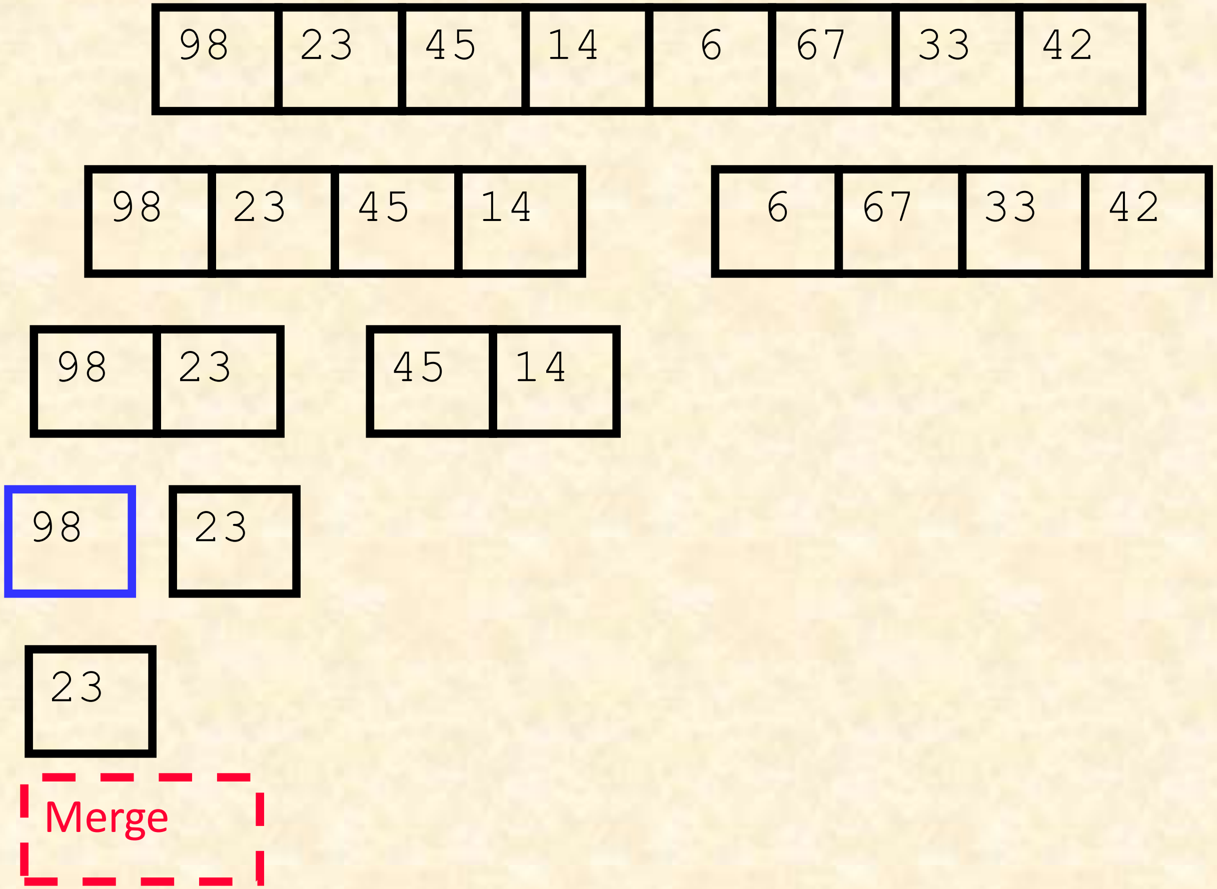
98 23

45 14

98

23

Merge



98	23	45	14	6	67	33	42

98	23	45	14

6	67	33	42

98	23

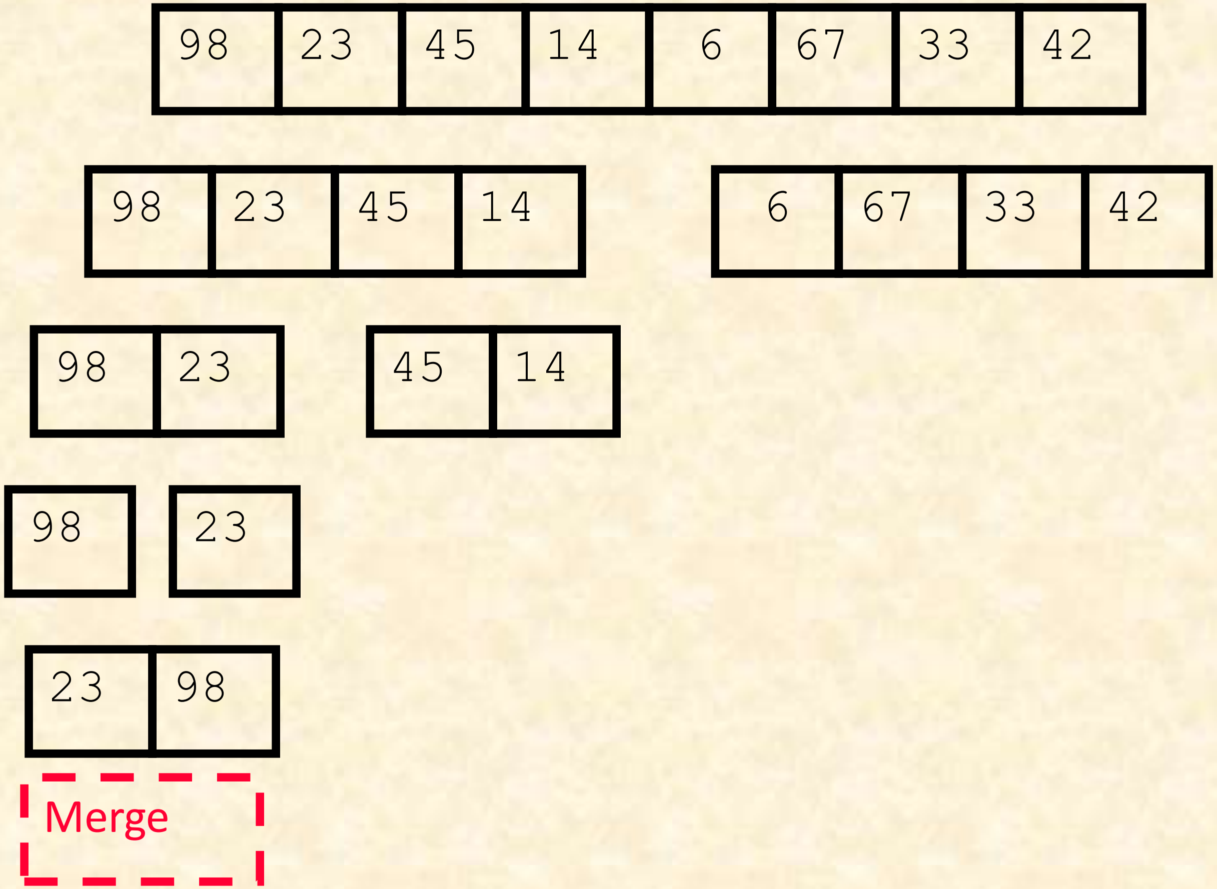
45	14

98

23

23

Merge



98 23 45 14 6 67 33 42

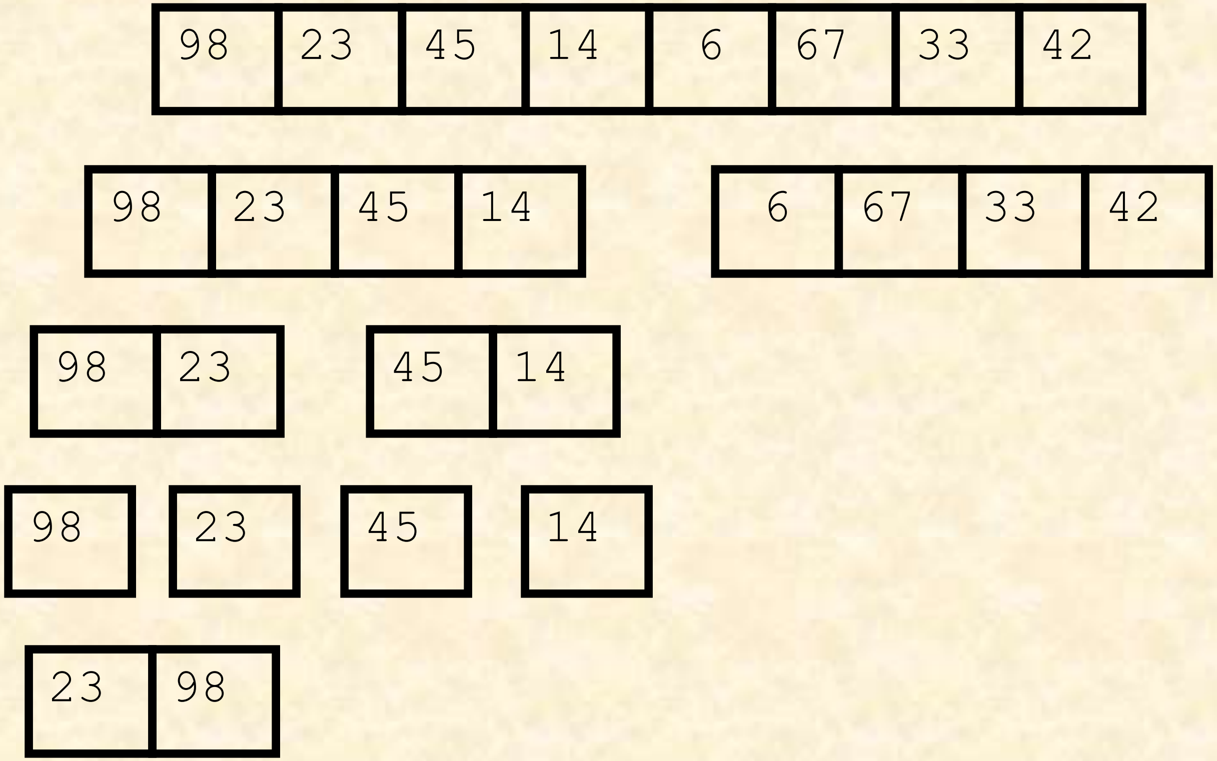
98 23 45 14 | 6 67 33 42

98 23 | 45 14

98 | 23

23 98

Merge



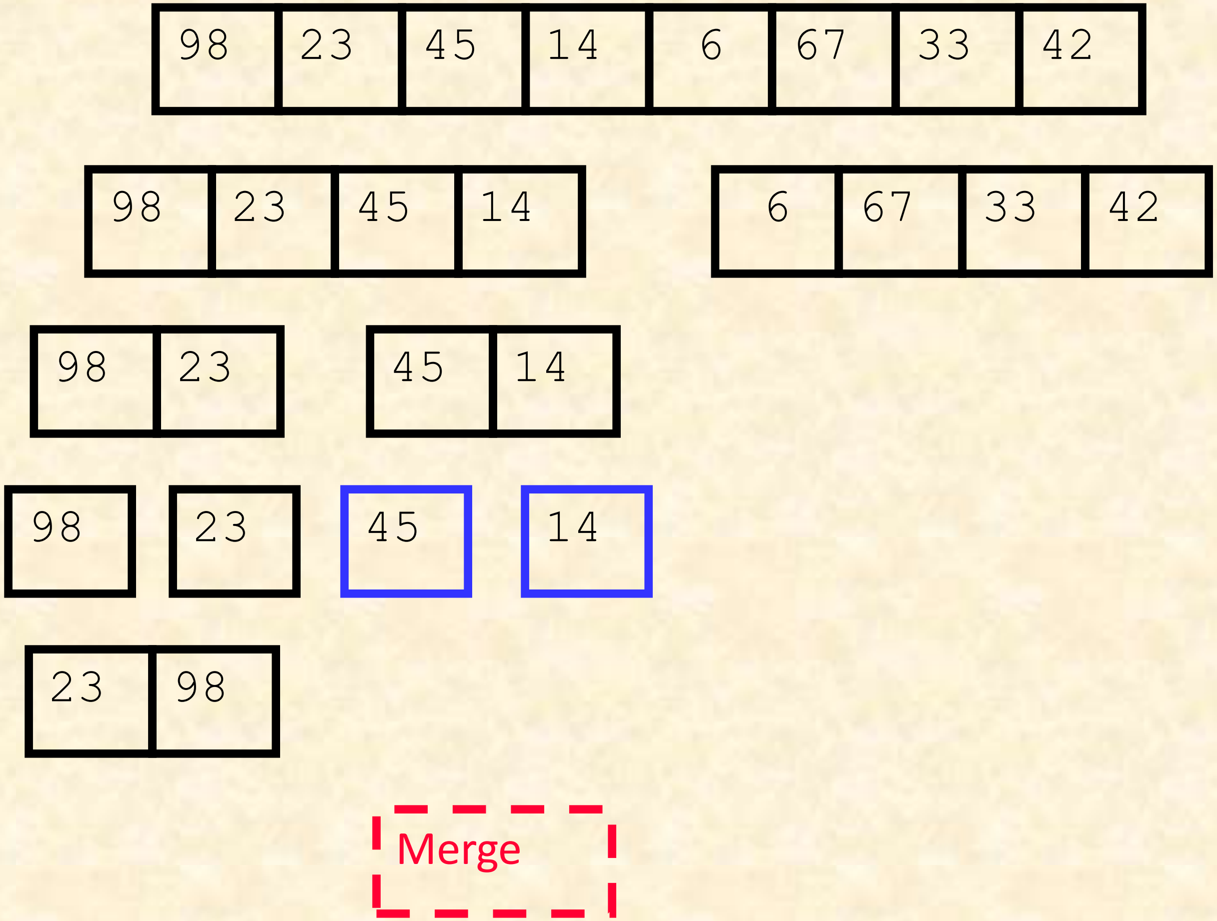
98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

98 23 45 14

98 23 45 14

23 98



98 23 45 14 6 67 33 42

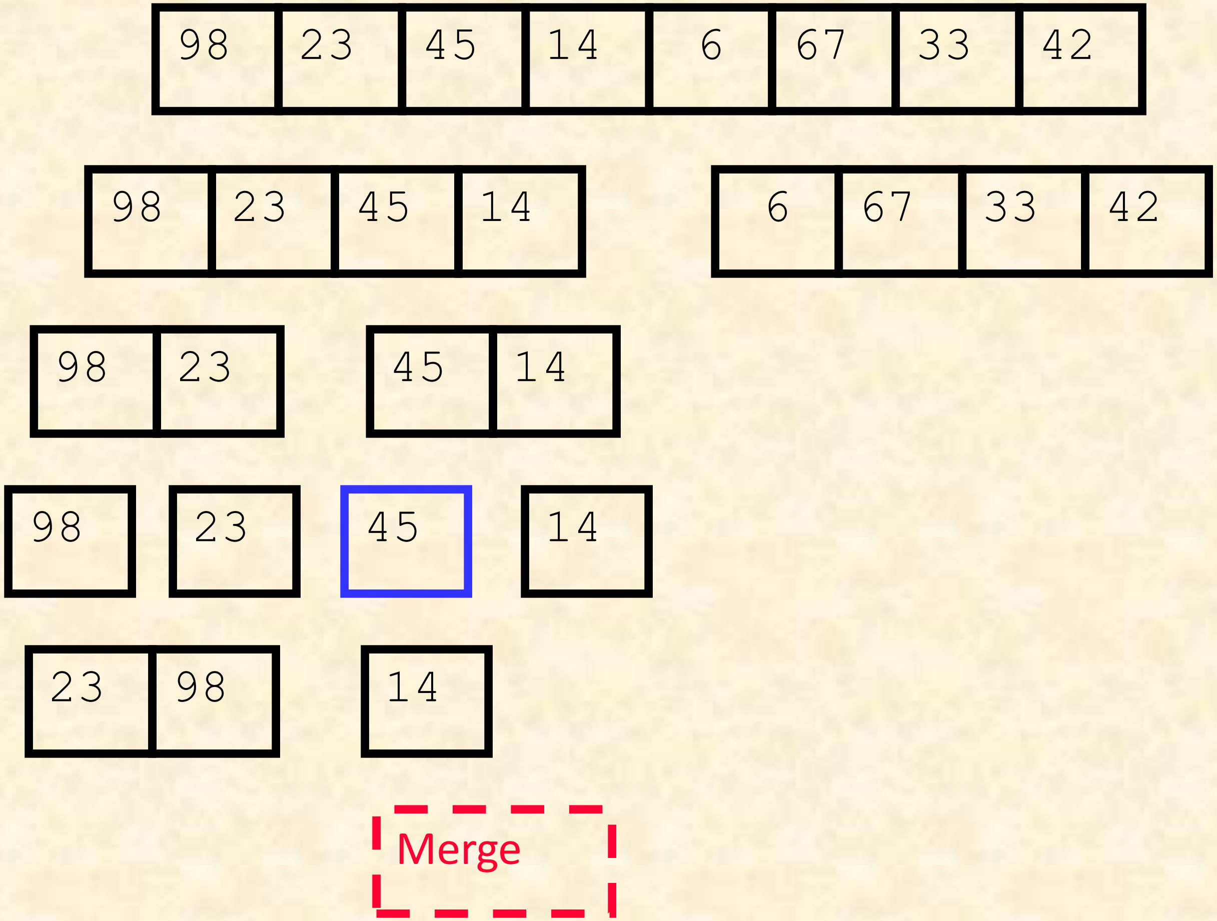
98 23 45 14 6 67 33 42

98 23 45 14

98 23 45 14

23 98

Merge



98 23 45 14 6 67 33 42

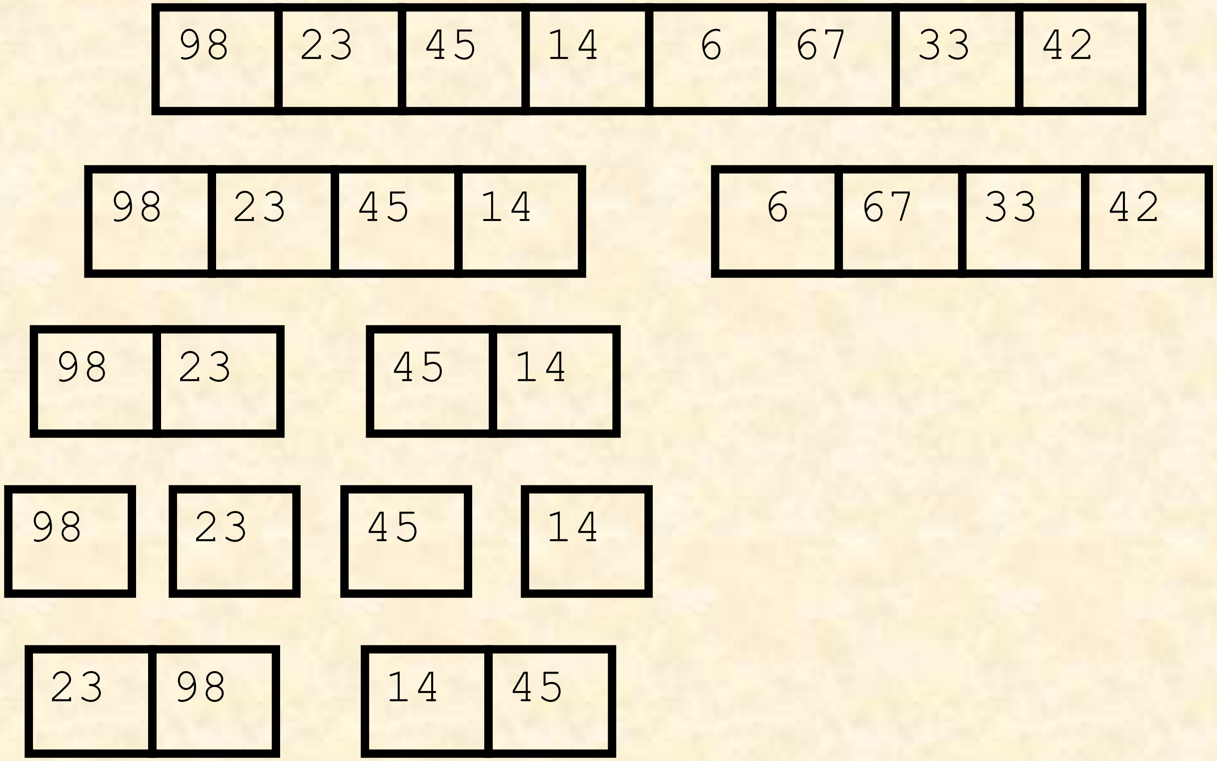
98 23 45 14　　6 67 33 42

98 23　　45 14

98　23　45　14

23 98　　14

Merge



98 23 45 14 6 67 33 42

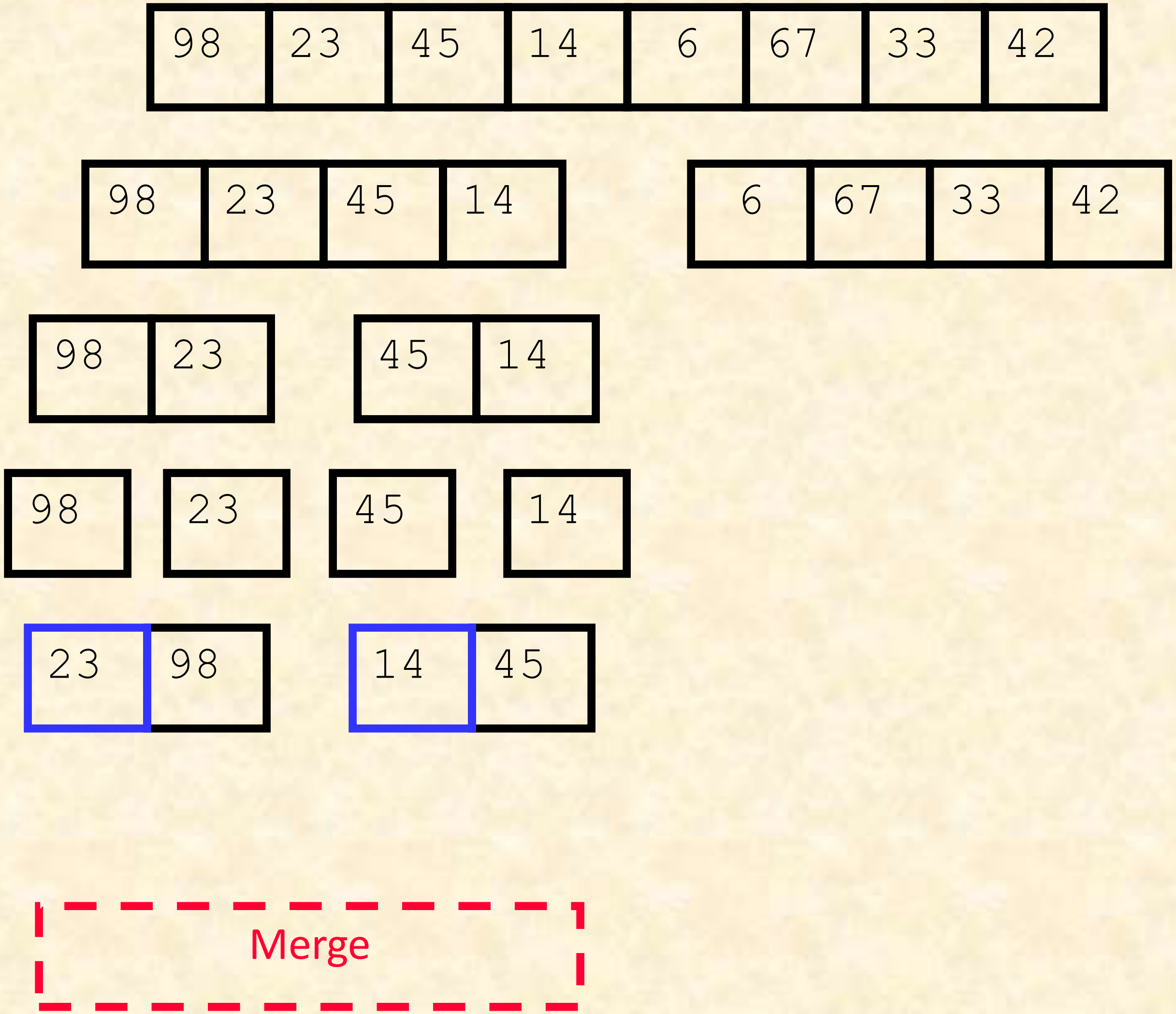
98 23 45 14 | 6 67 33 42

98 23 | 45 14

98 | 23 | 45 | 14

23 98 | 14 45

Merge



98 23 45 14 6 67 33 42

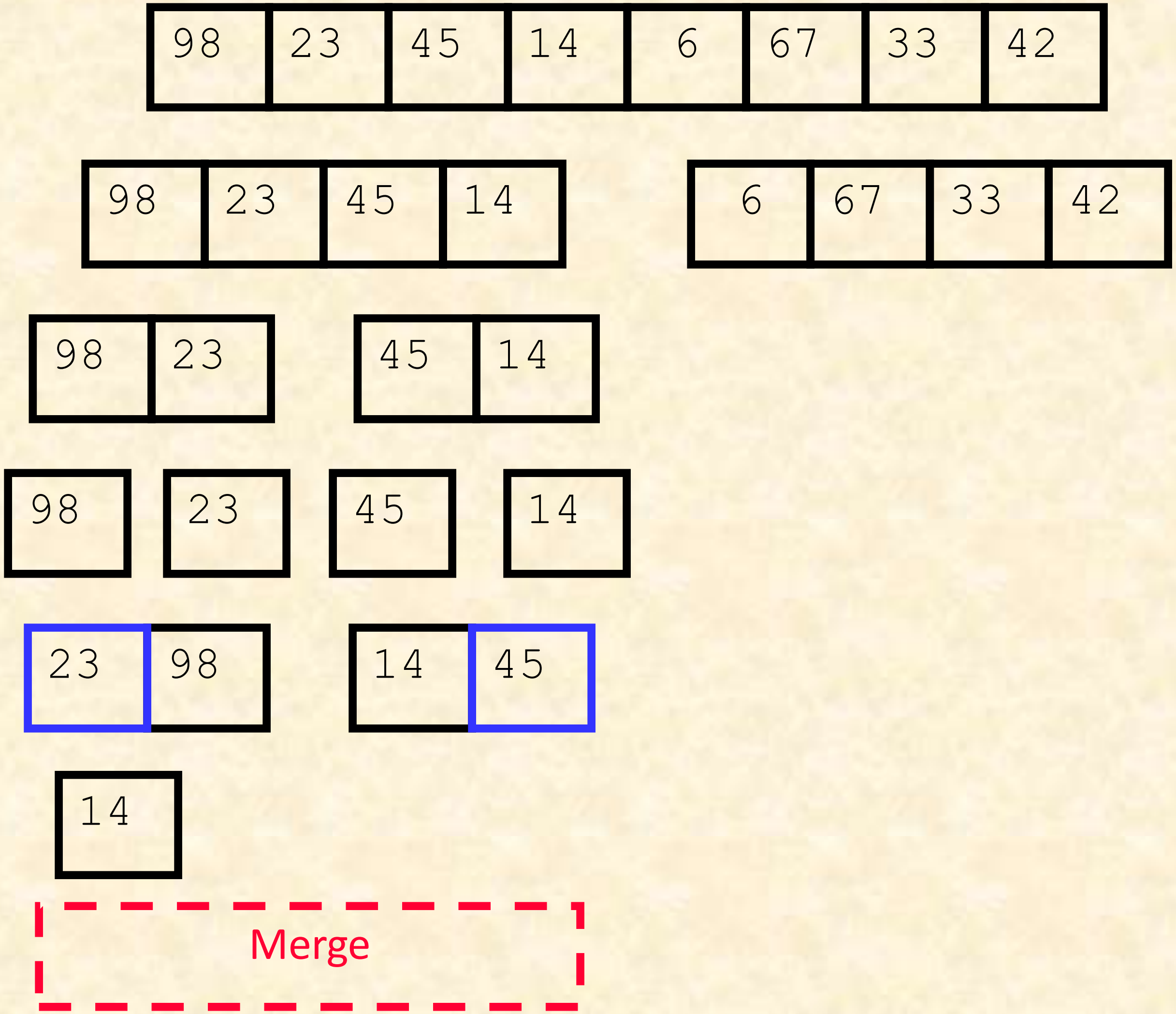
98 23 45 14 | 6 67 33 42

98 23 | 45 14

98 | 23 | 45 | 14

23 98 | 14 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

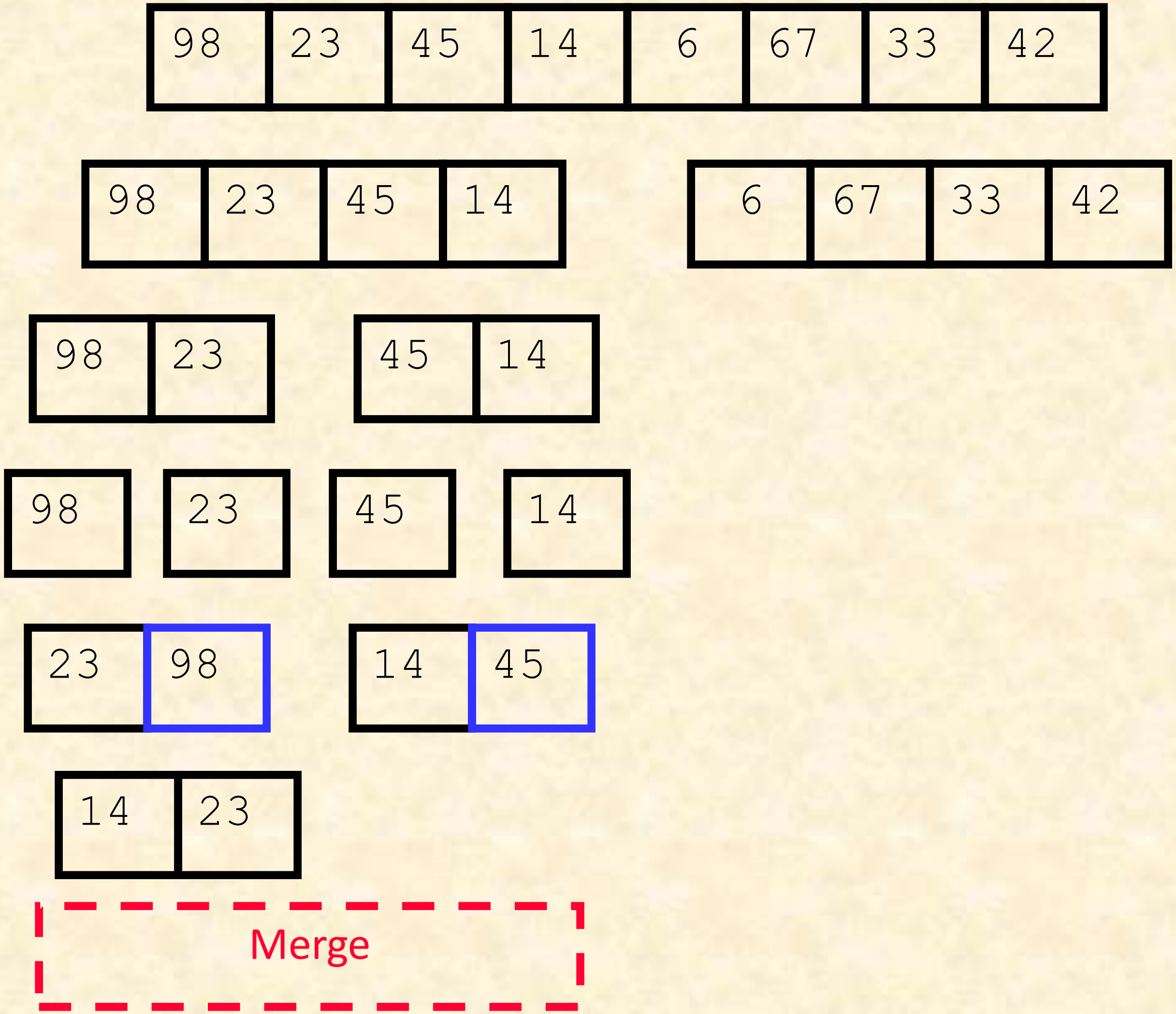
98 23   45 14

98   23   45   14

23 98   14 45

14

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

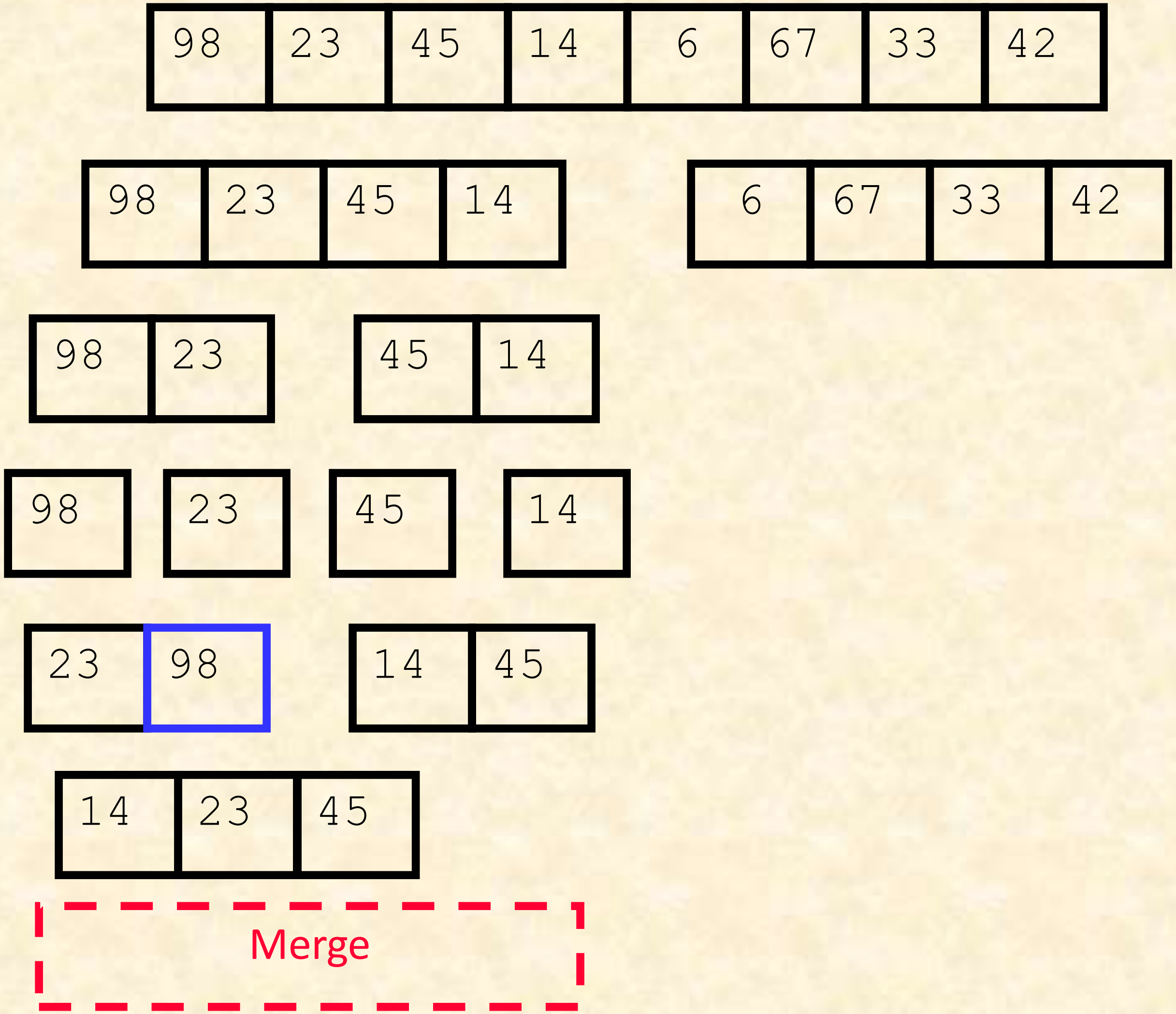
98 23   45 14

98   23   45   14

23 98   14 45

14 23

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

23

45

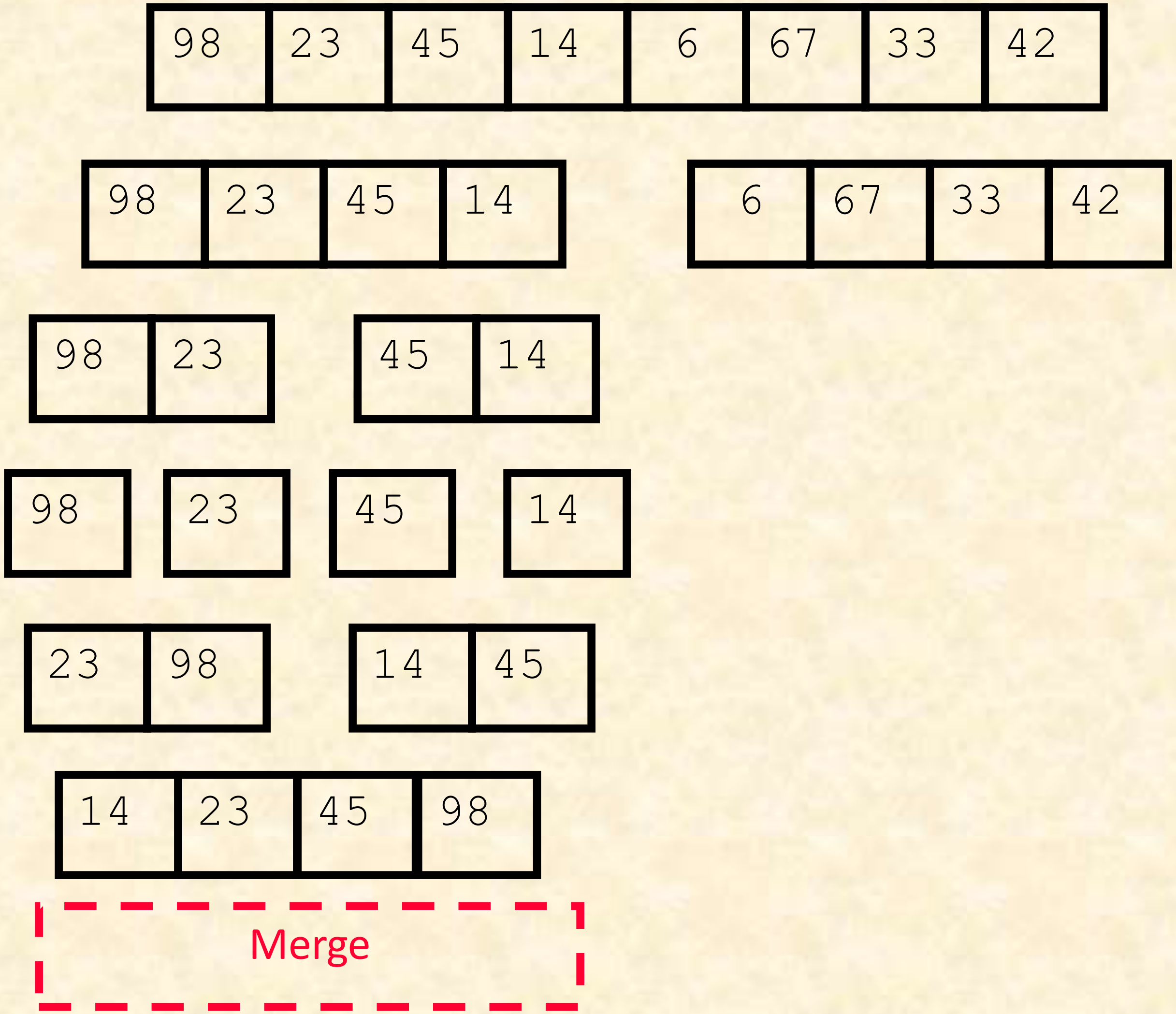
14

23 98

14 45

14 23 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

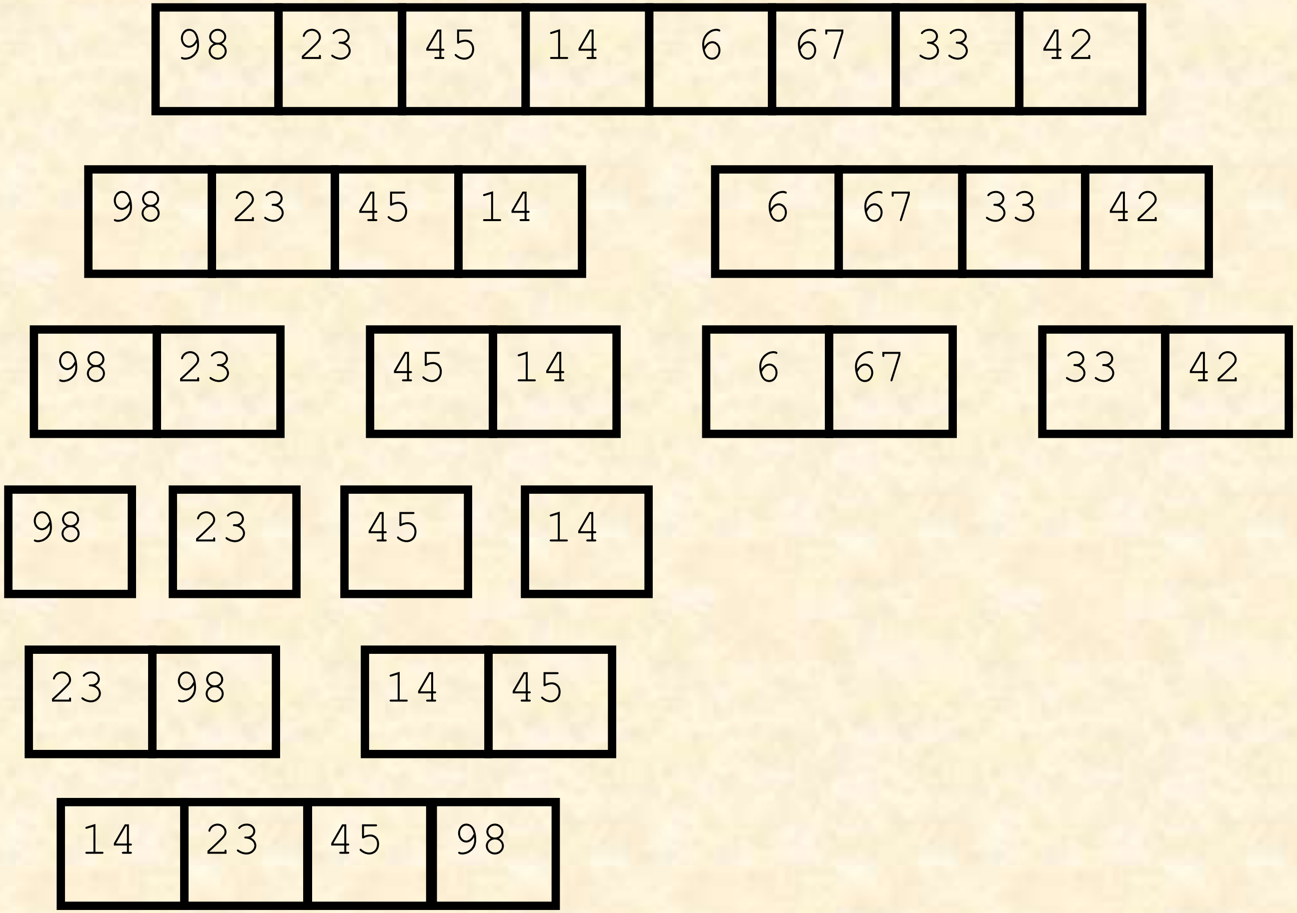
98 23   45 14

98   23   45   14

23 98   14 45

14 23 45 98

Merge



98 23 45 14 6 67 33 42

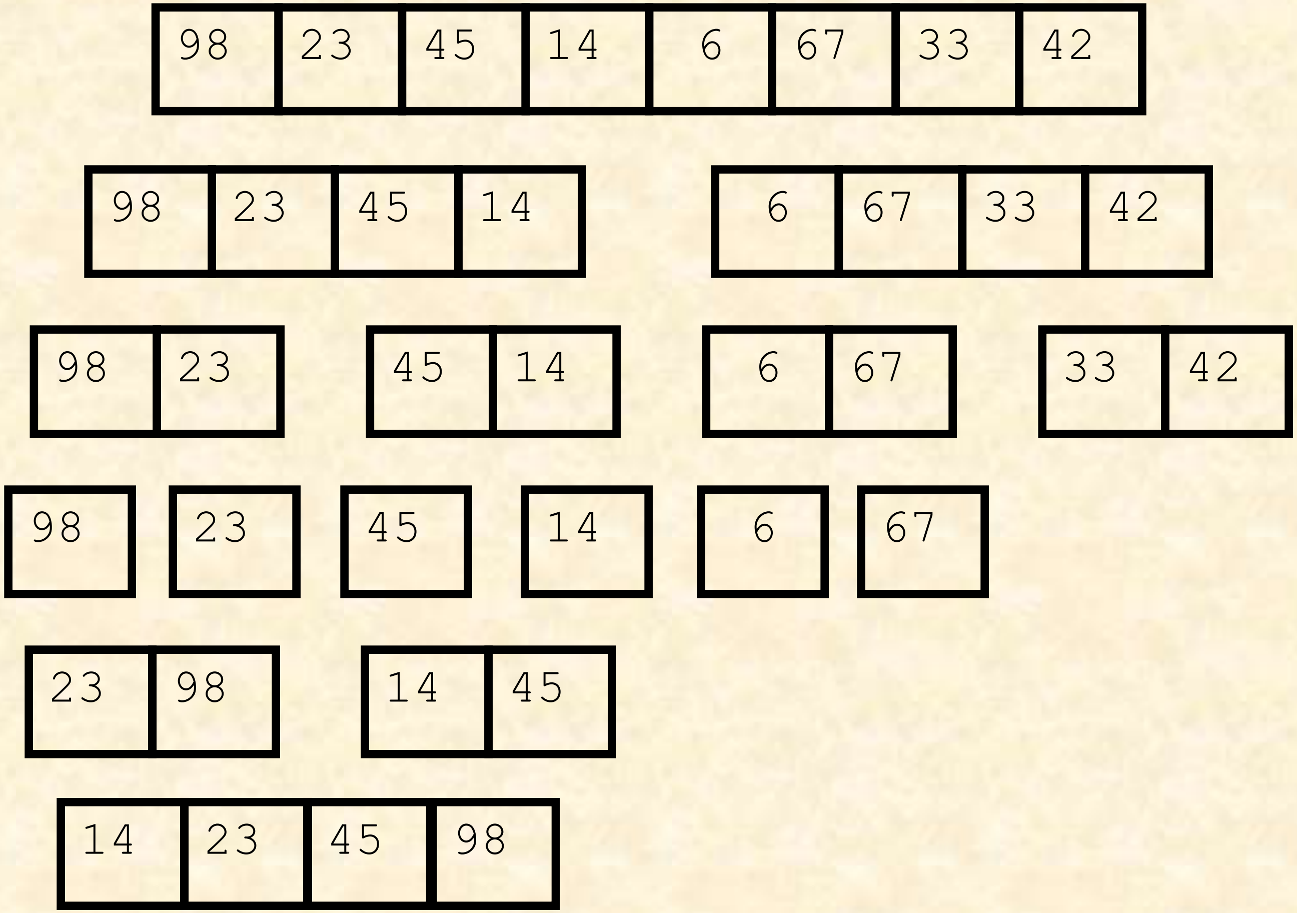
98 23 45 14　6 67 33 42

98 23　45 14　6 67　33 42

98　23　45　14

23 98　14 45

14 23 45 98



98 23 45 14 6 67 33 42

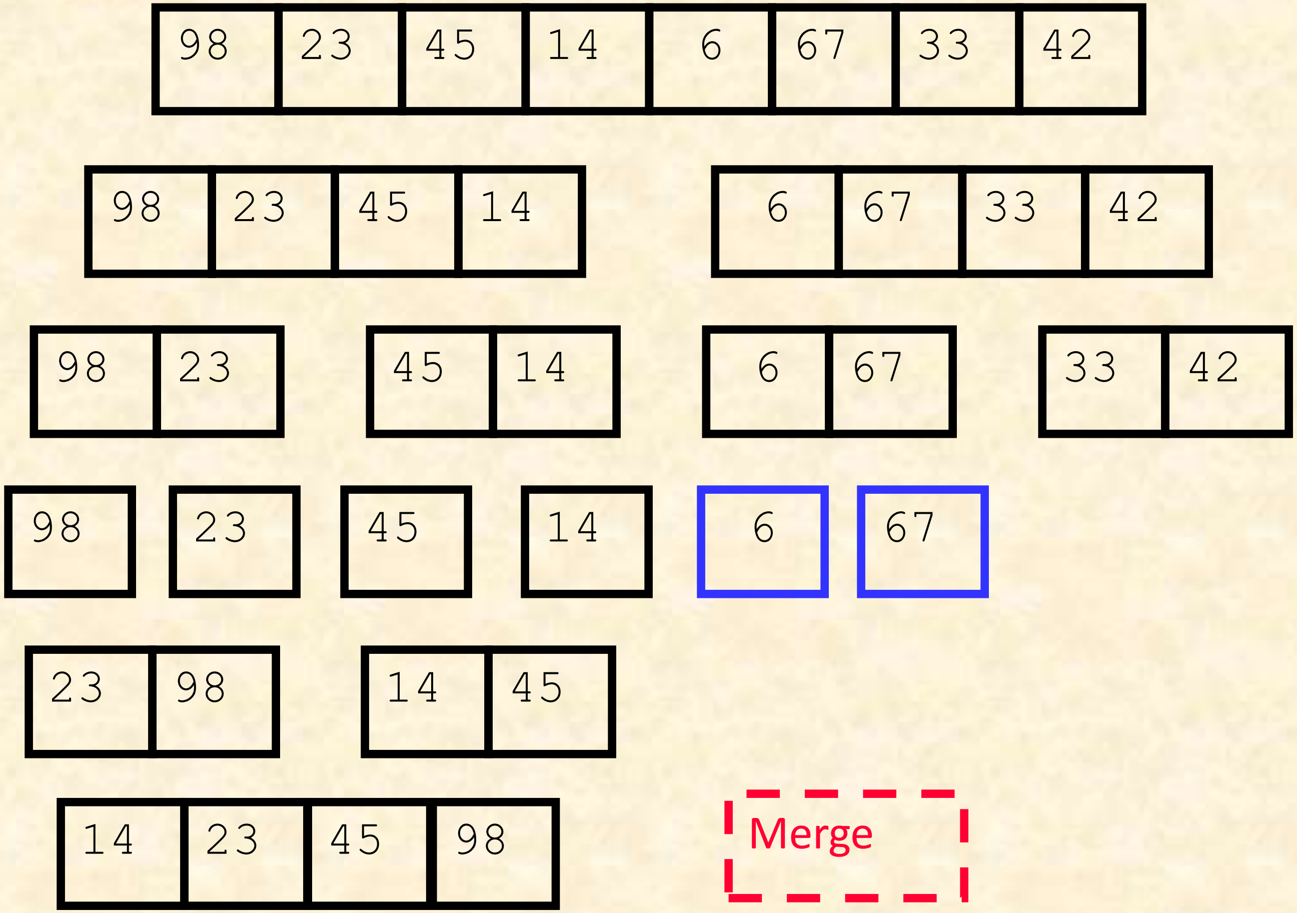
98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67

23 98 | 14 45

14 23 45 98



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

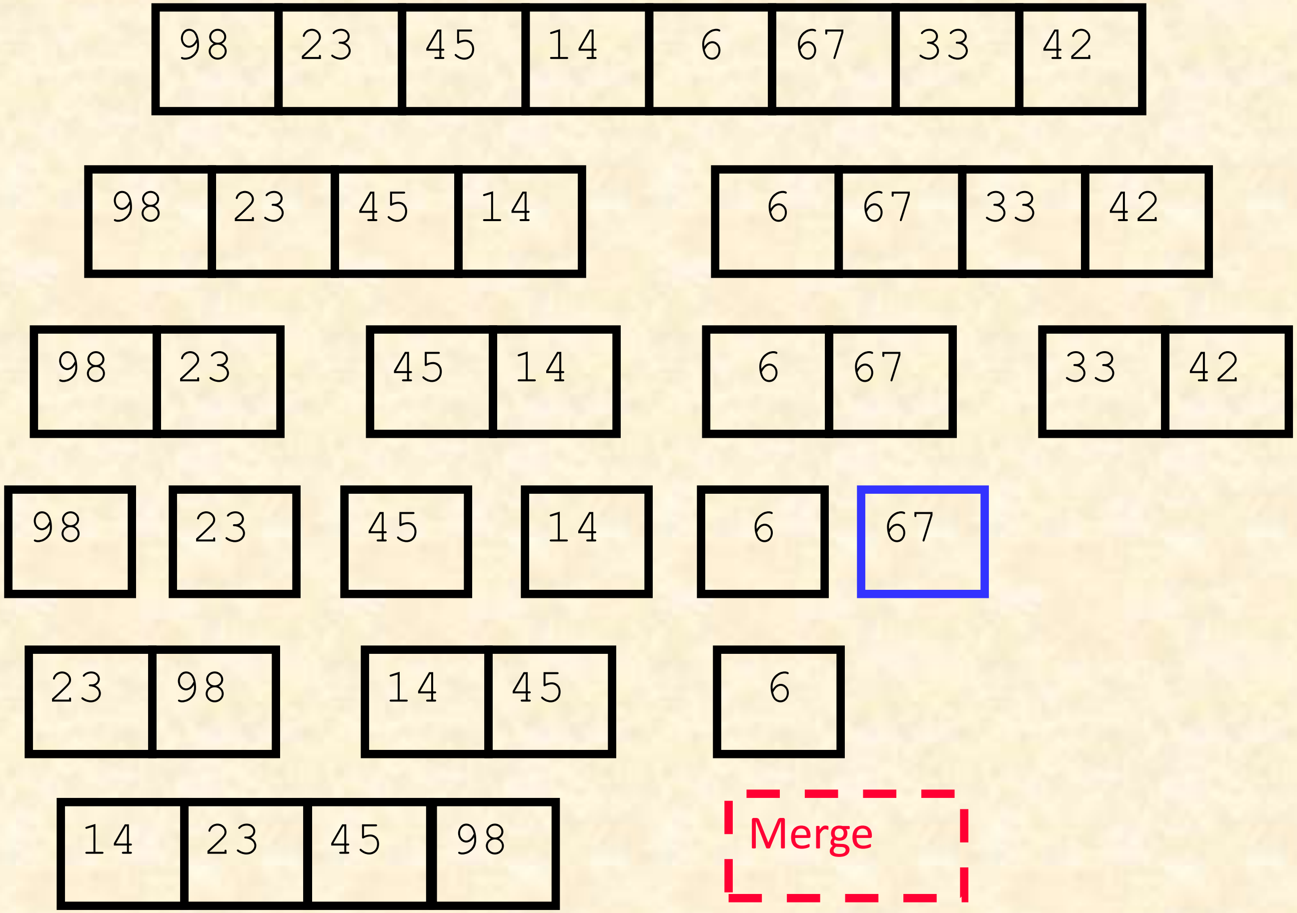
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67

23 98 | 14 45

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

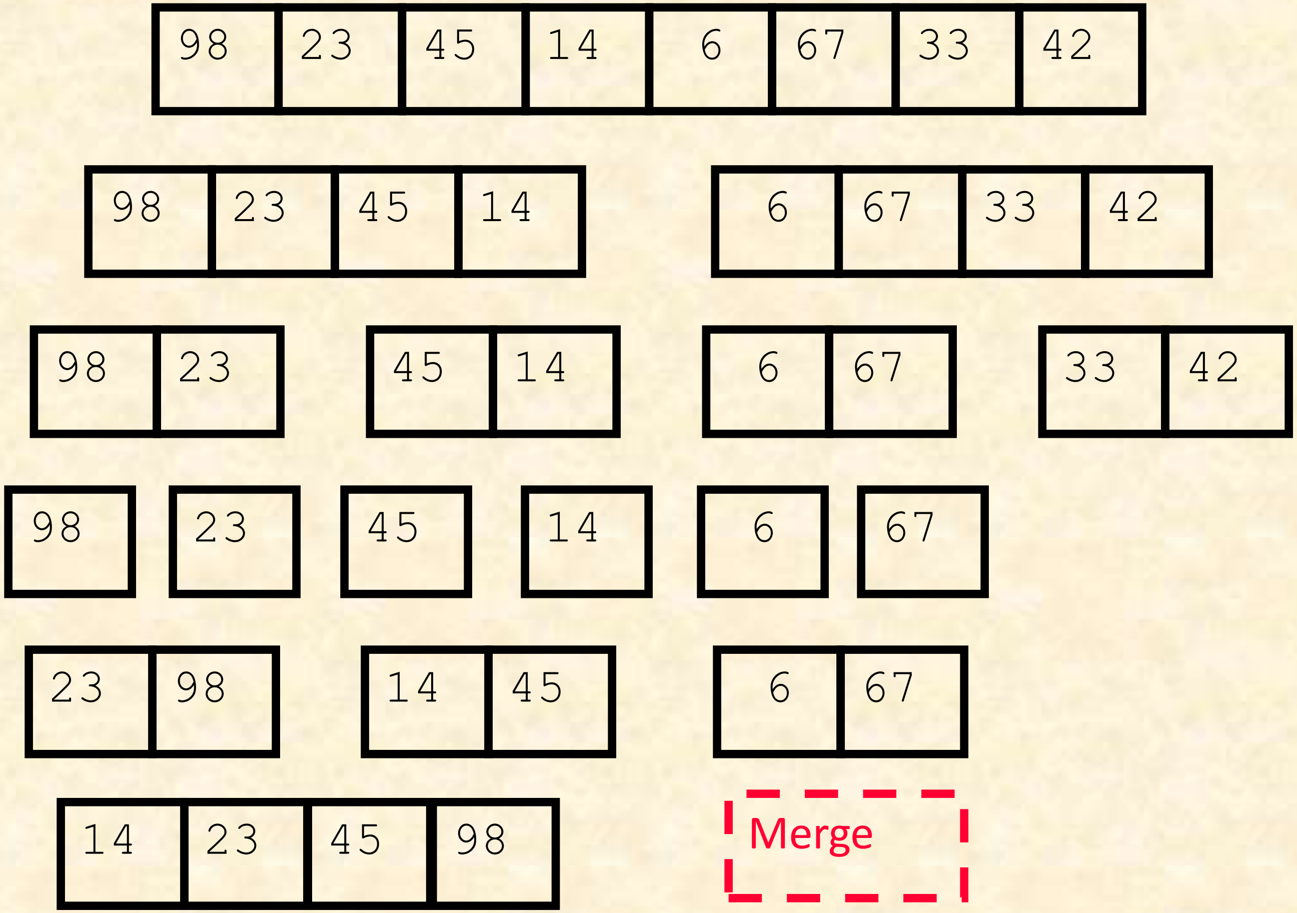
98 23 45 14 6 67 33 42

98 23 45 14 6 67

23 98 14 45 6

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

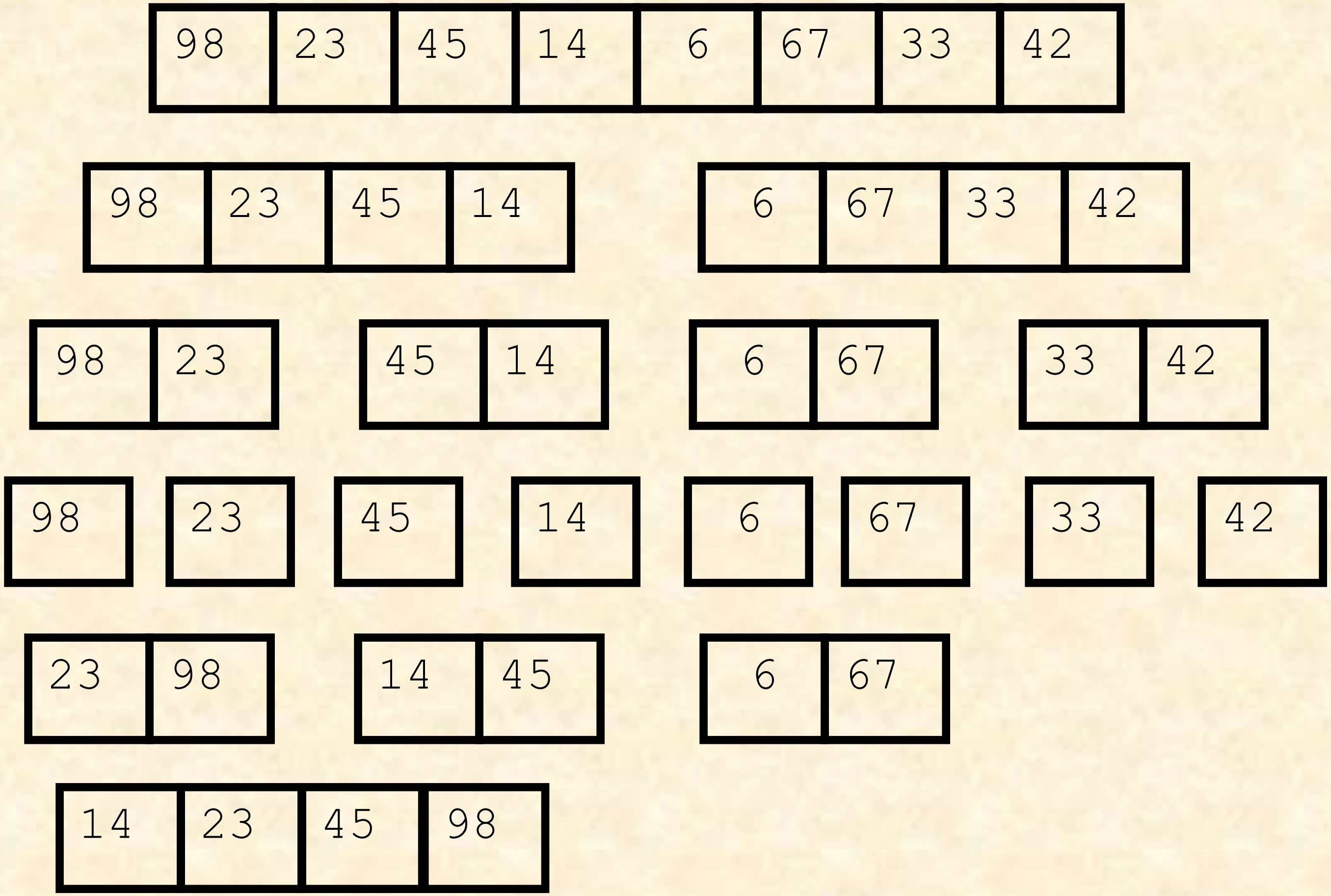
98 23 45 14 6 67 33 42

98 23 45 14 6 67

23 98 14 45 6 67

14 23 45 98

Merge



98 23 45 14 6 67 33 42

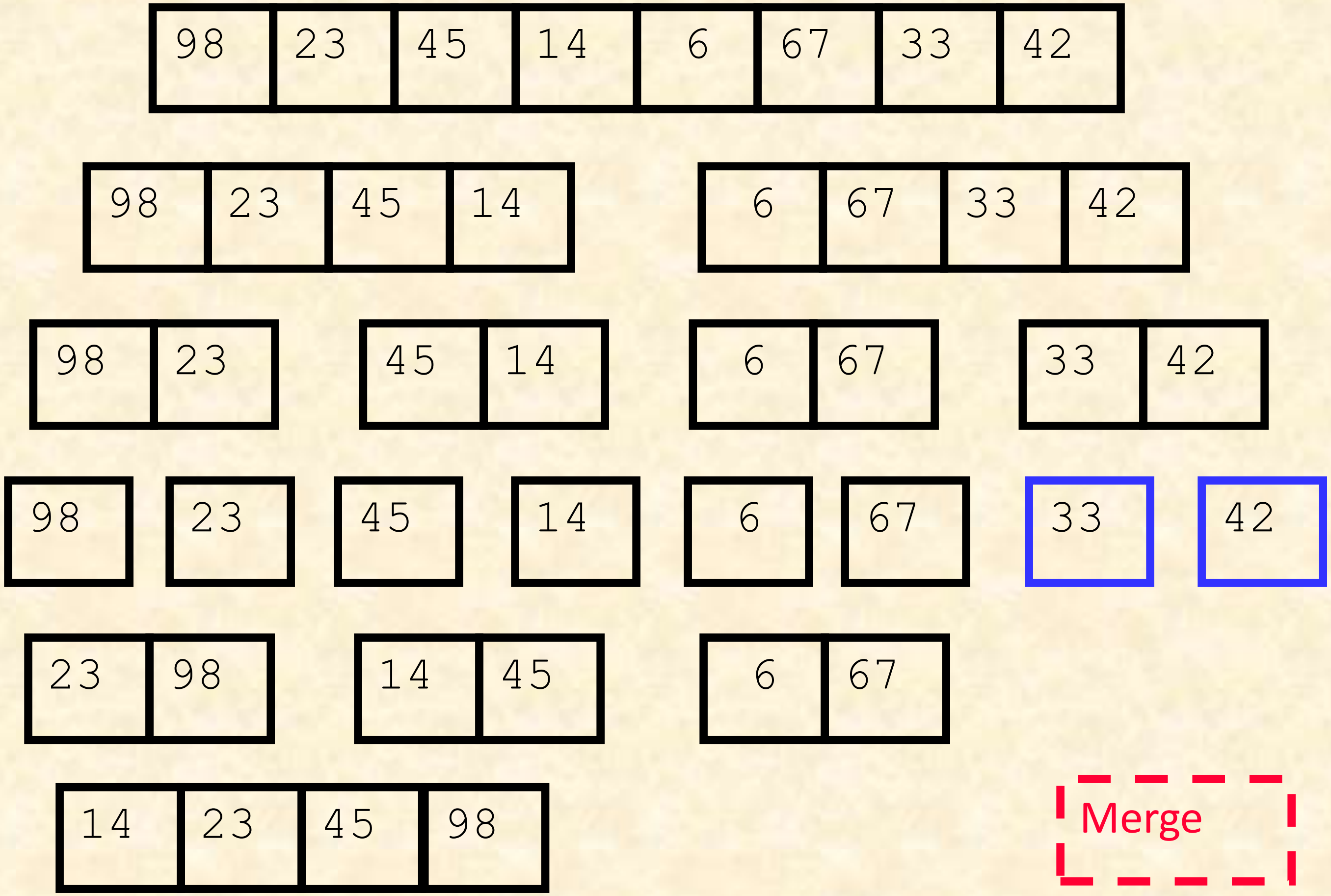
98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67

14 23 45 98



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

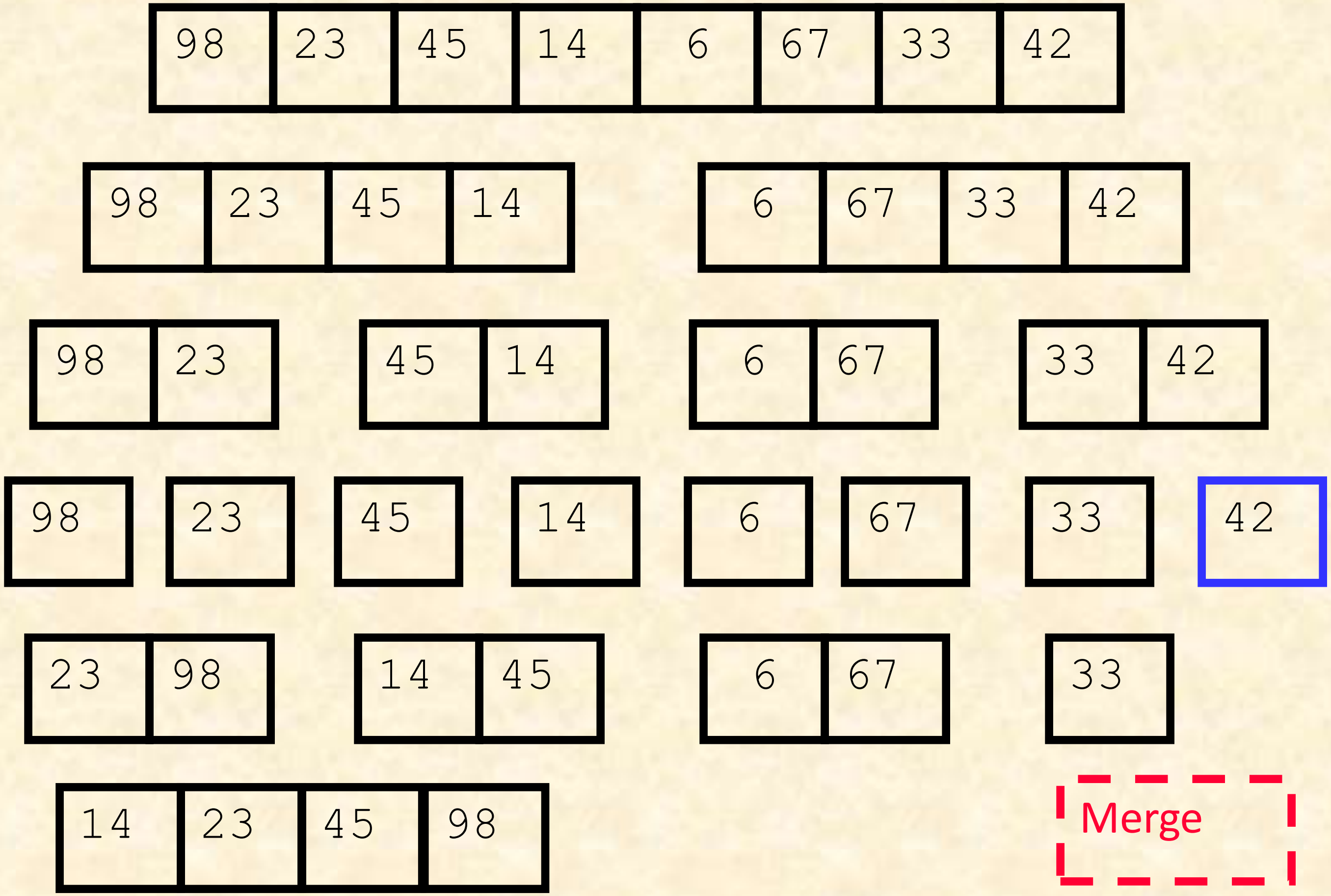
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

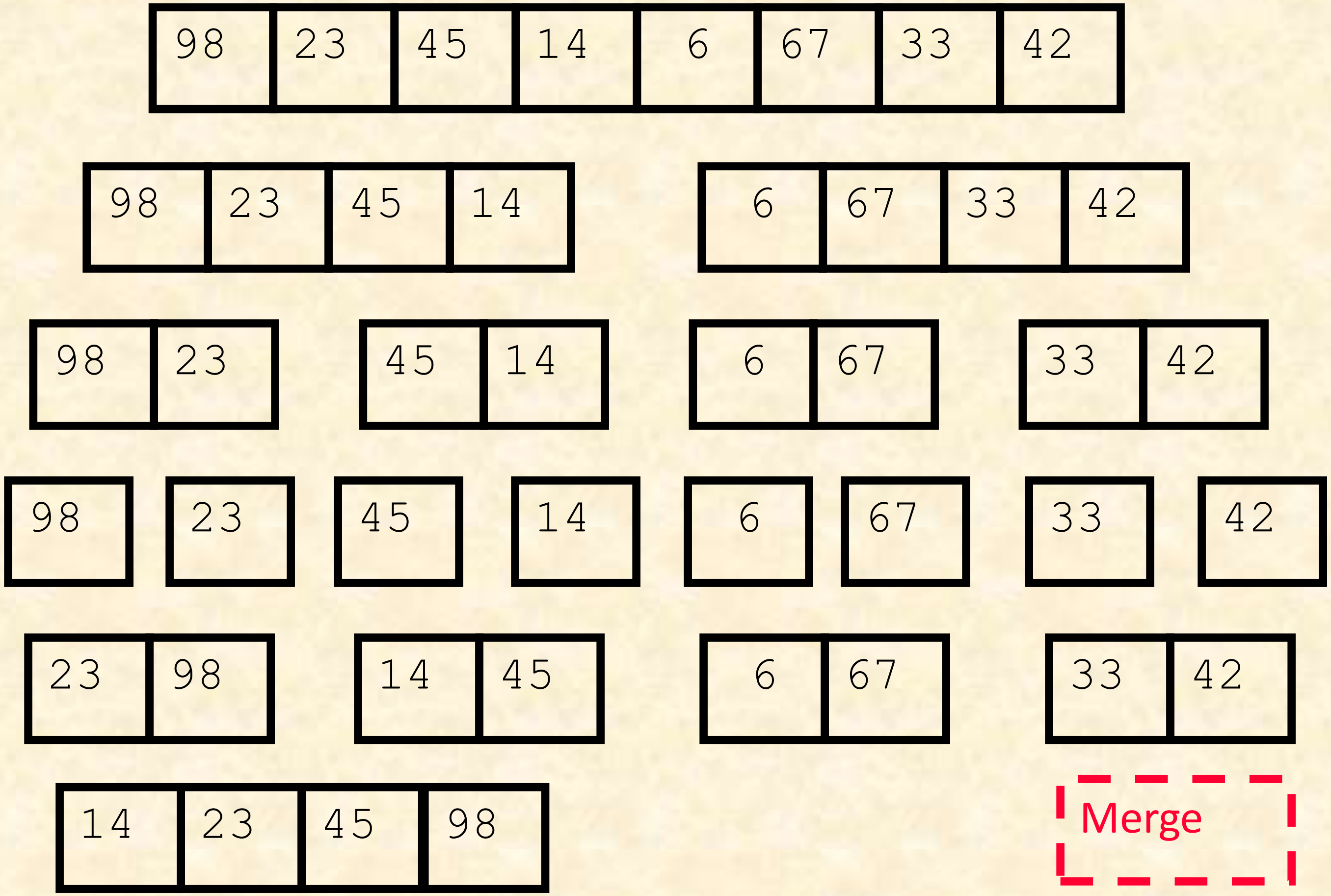
98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67   33

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

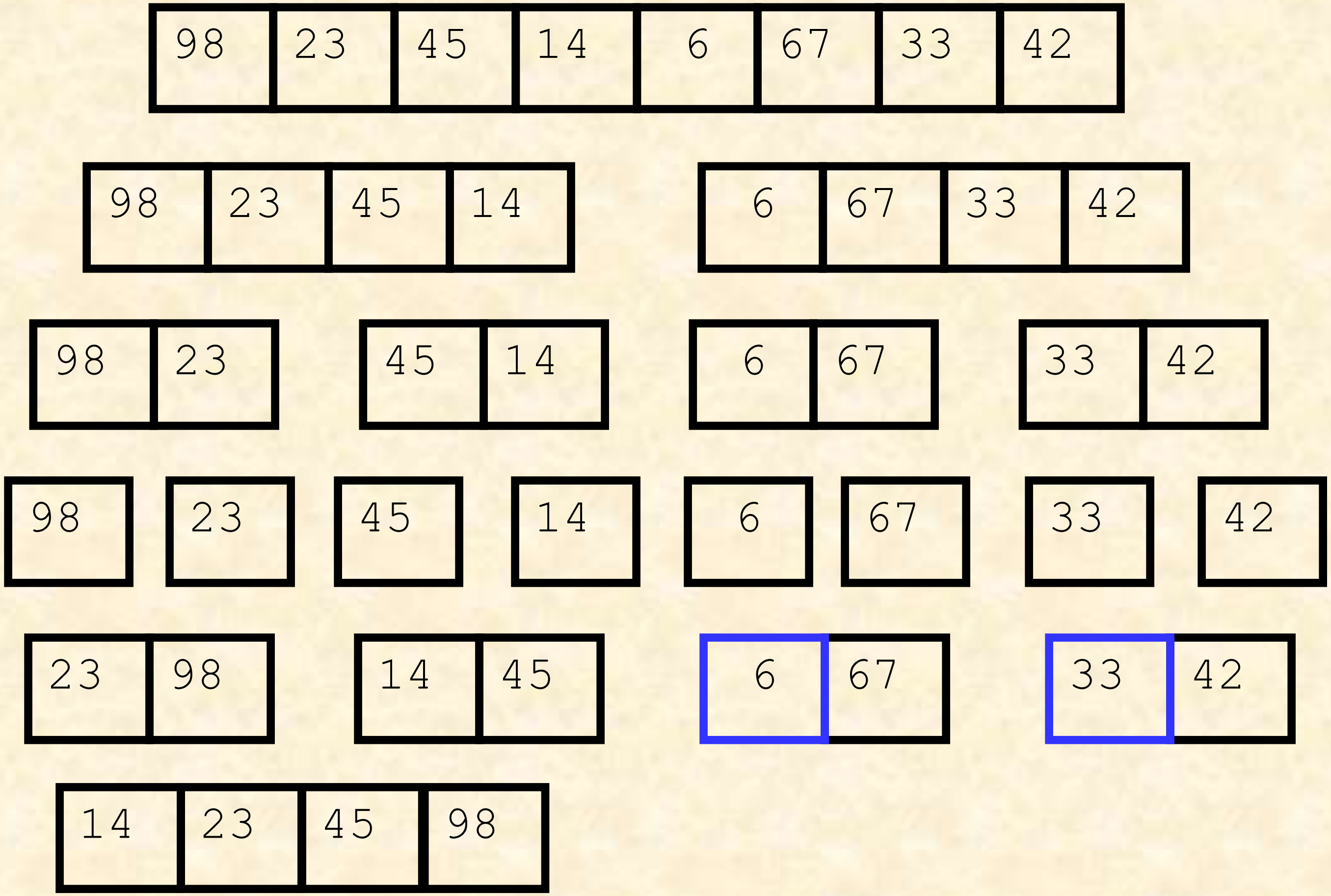
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

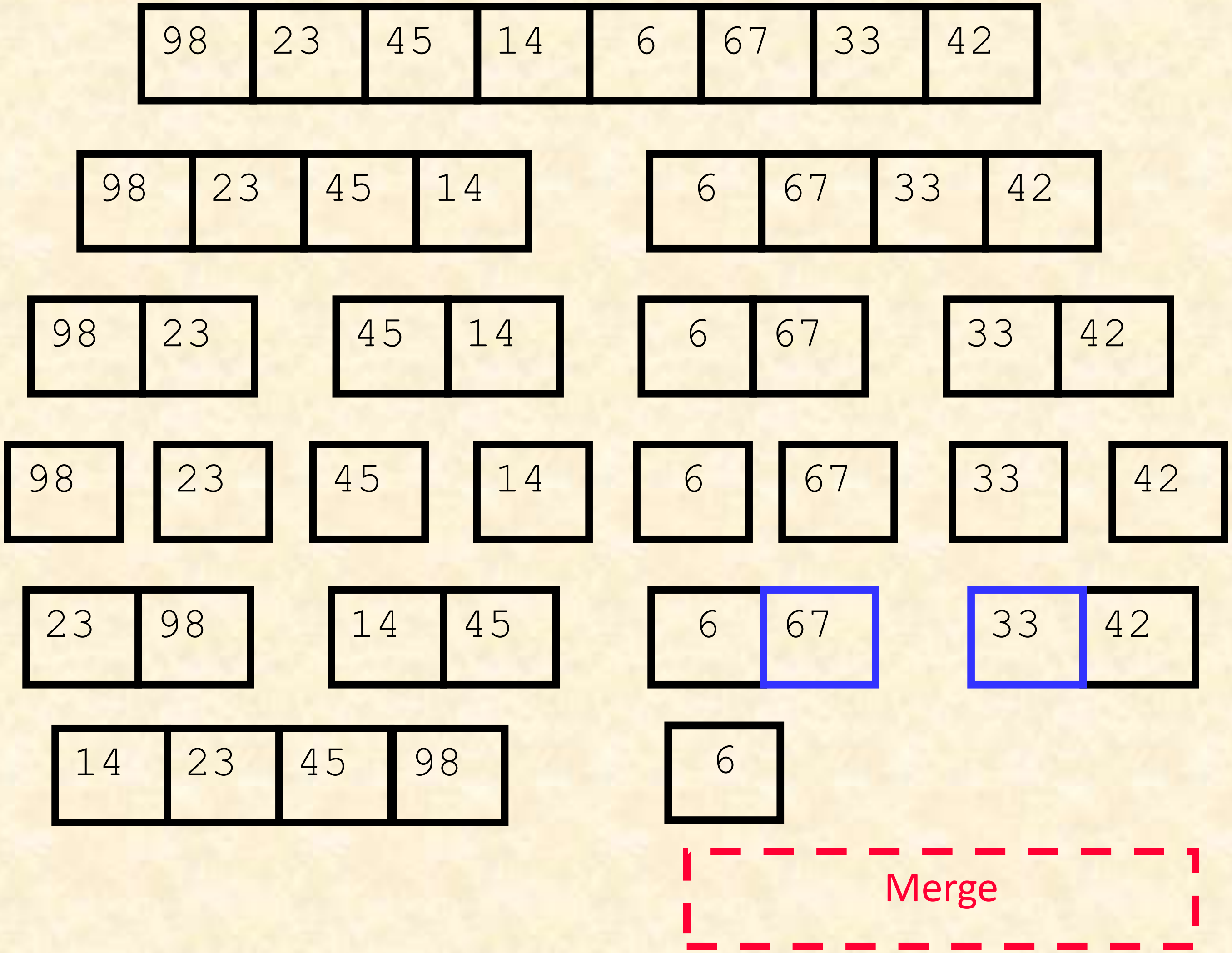
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

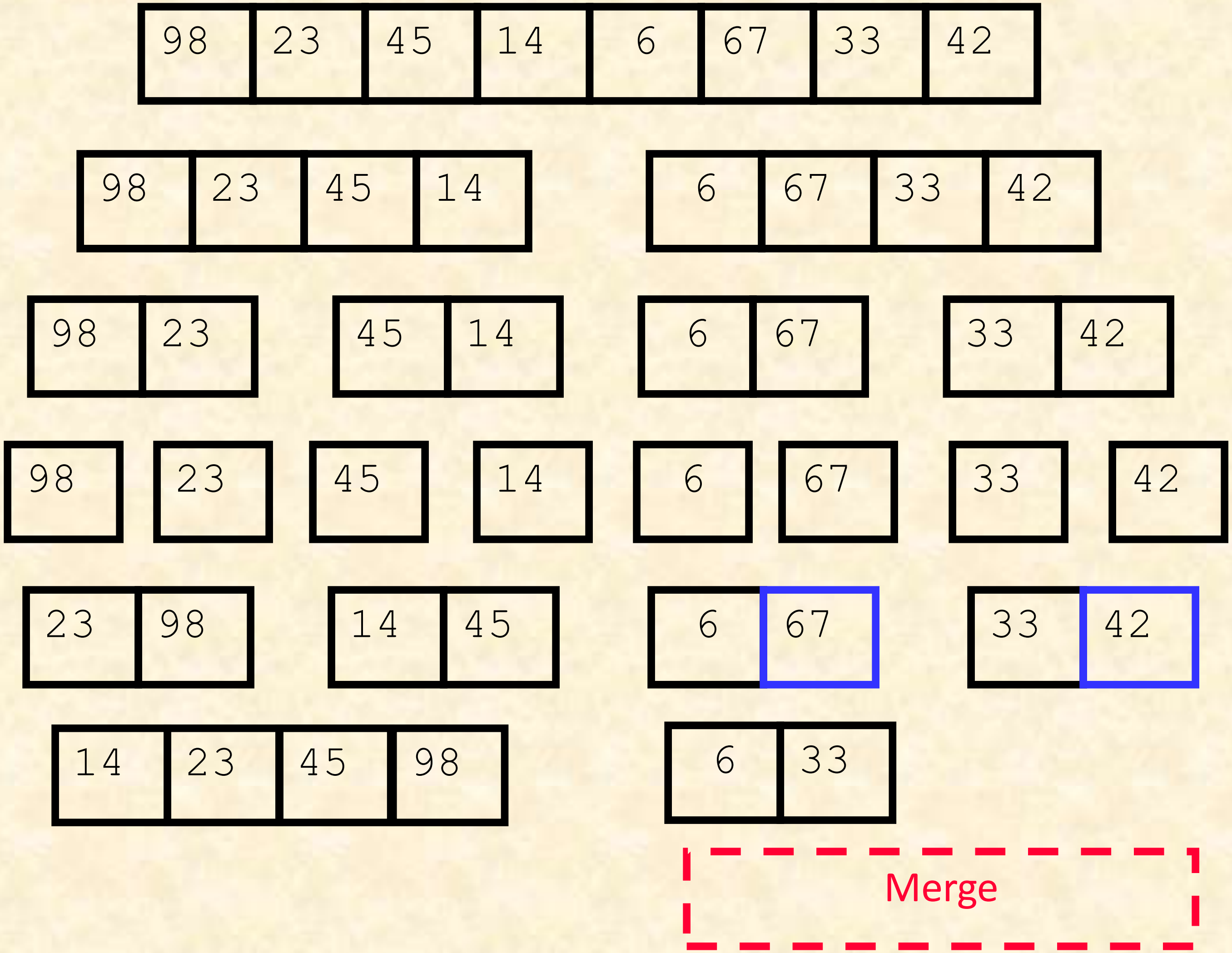
98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

23 98 14 45 6 67 33 42

14 23 45 98 6

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

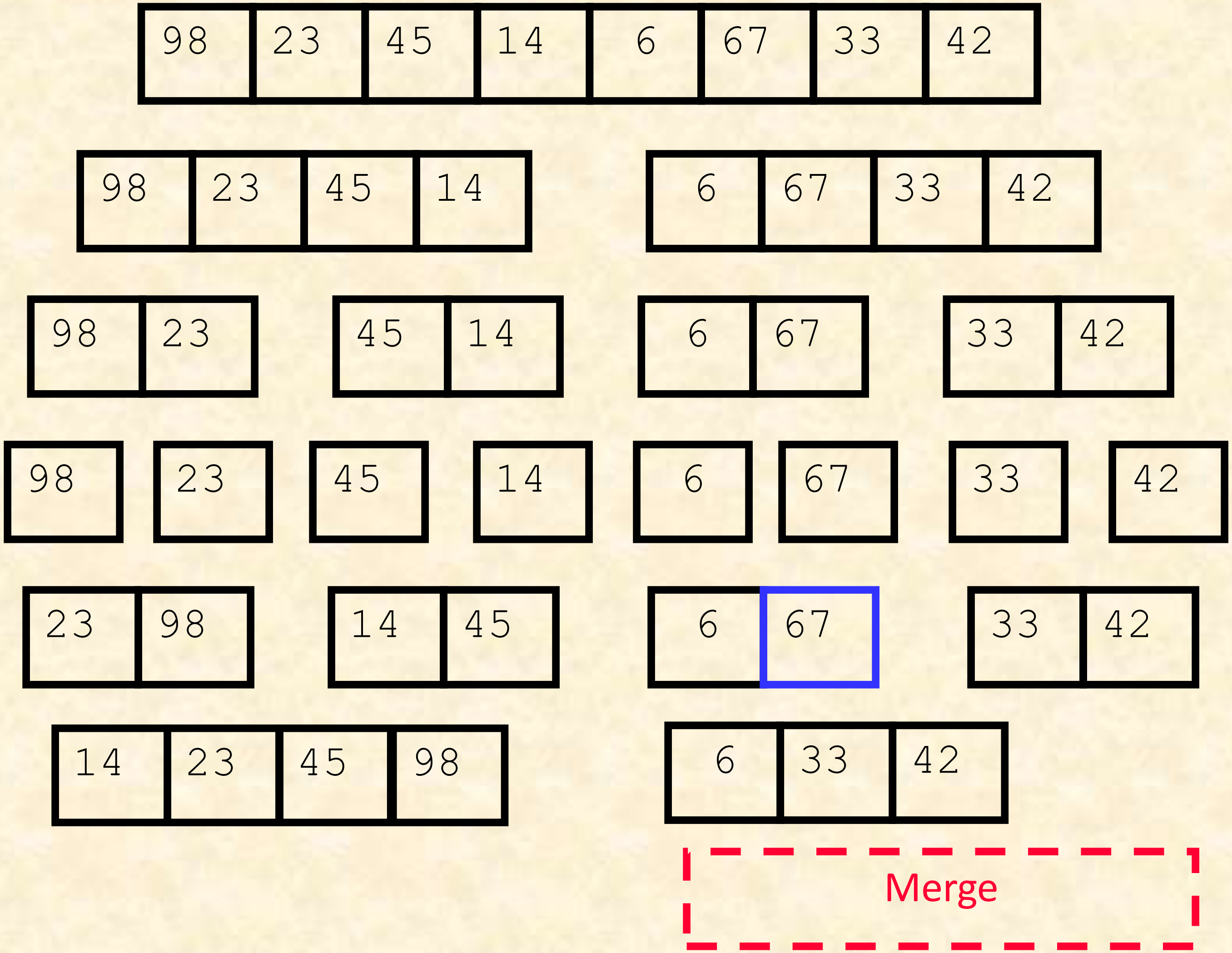
98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

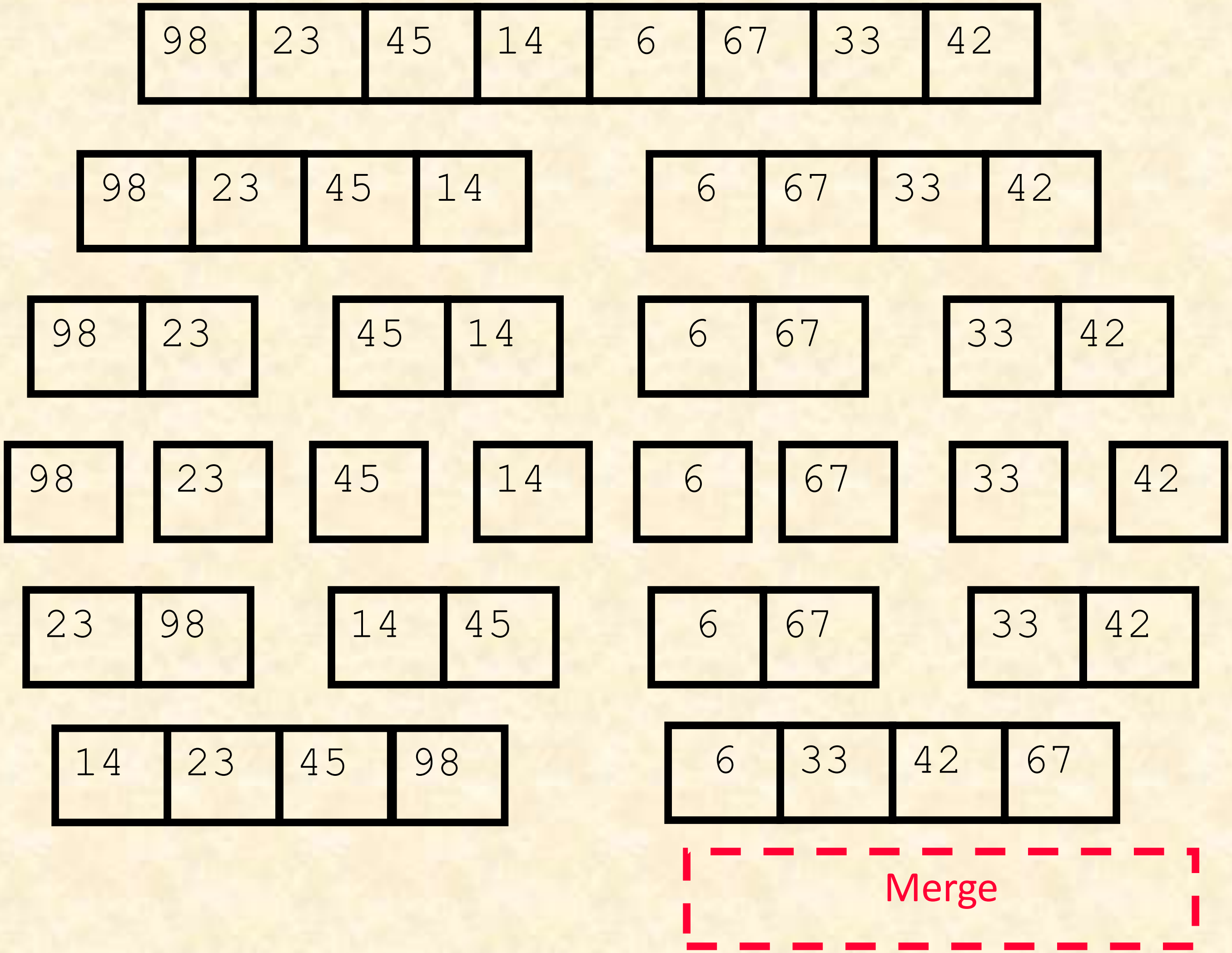
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

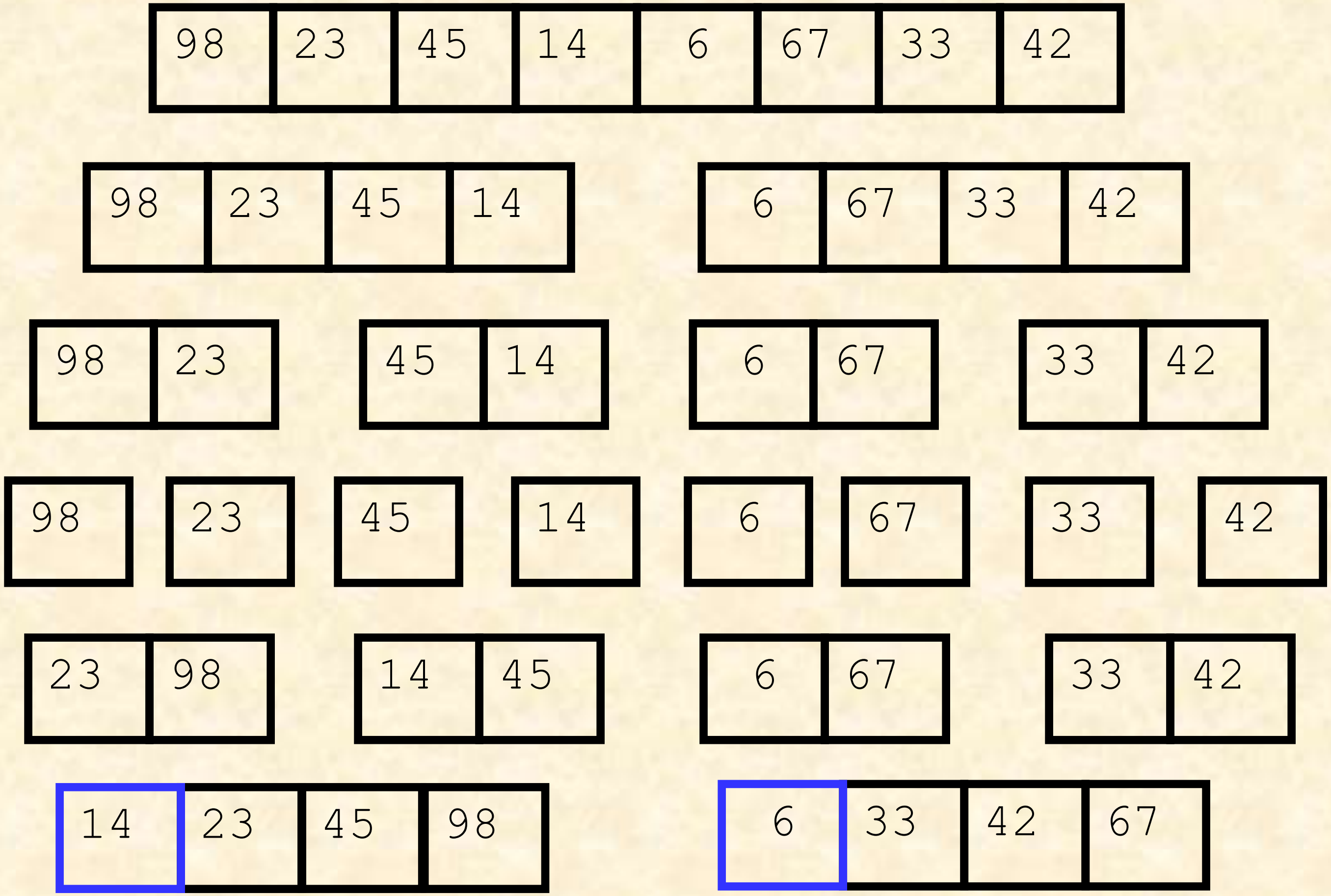
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

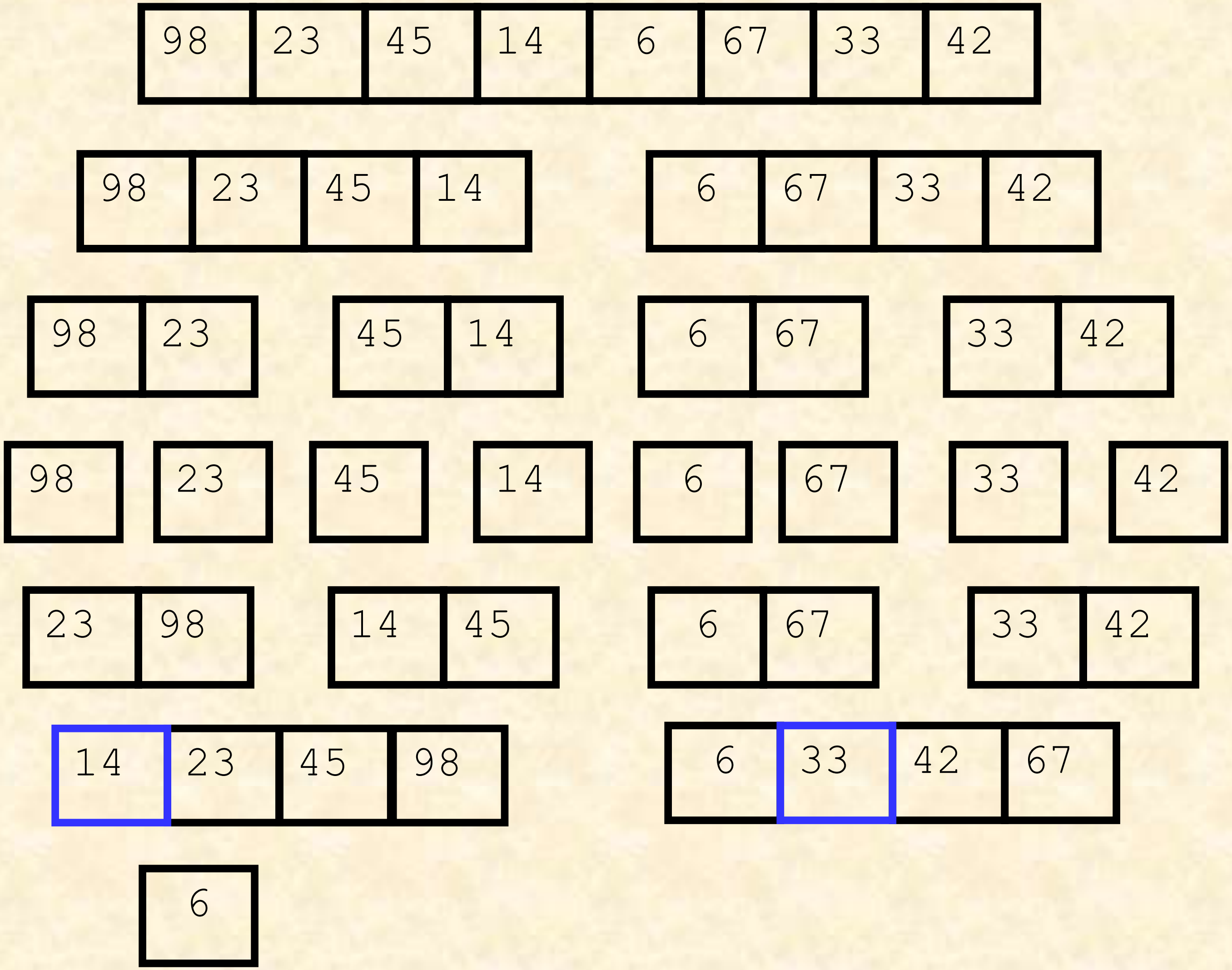
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

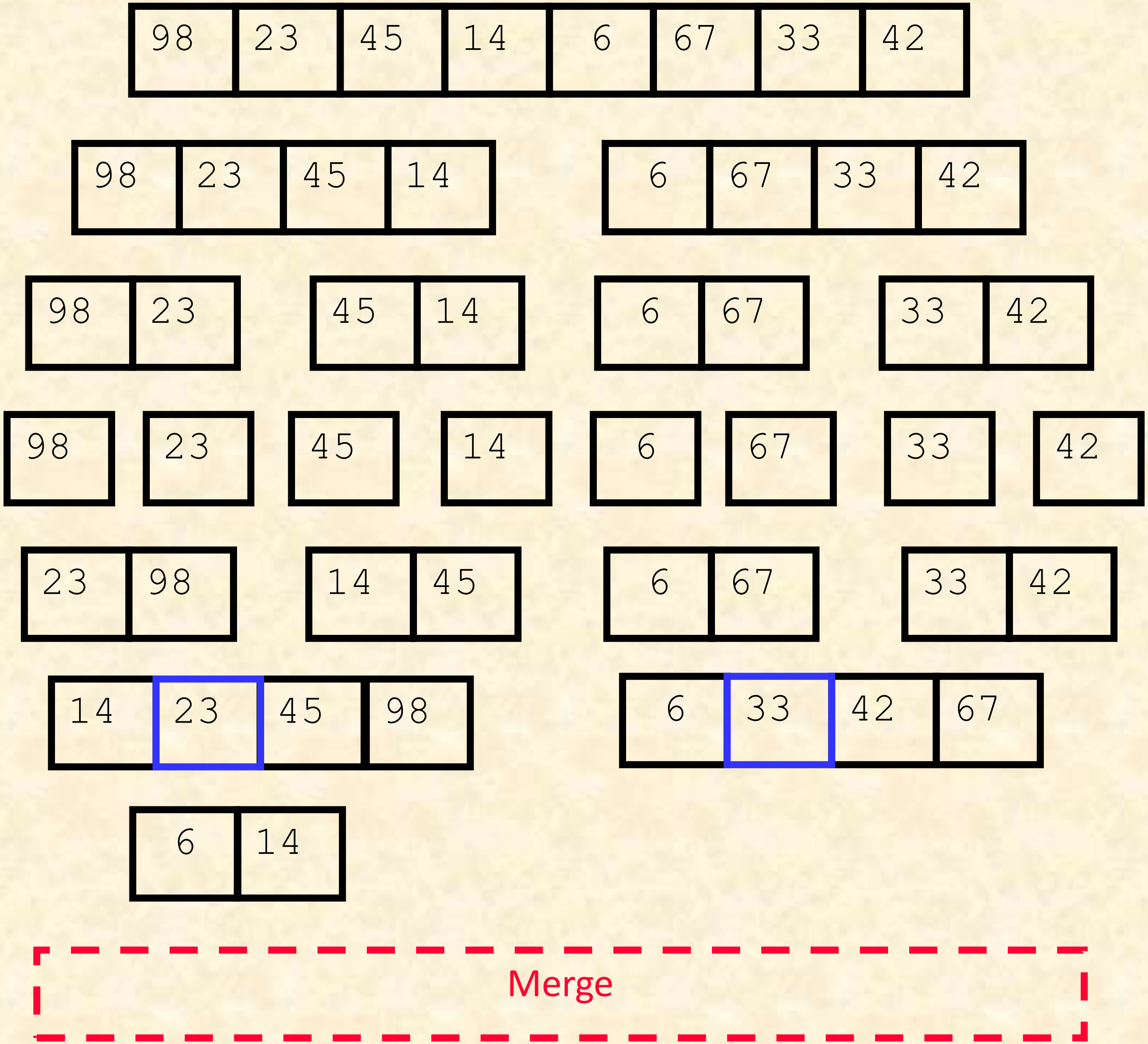
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6

Merge



98 23 45 14 6 67 33 42

98 23 45 14 — 6 67 33 42

98 23 — 45 14 — 6 67 — 33 42

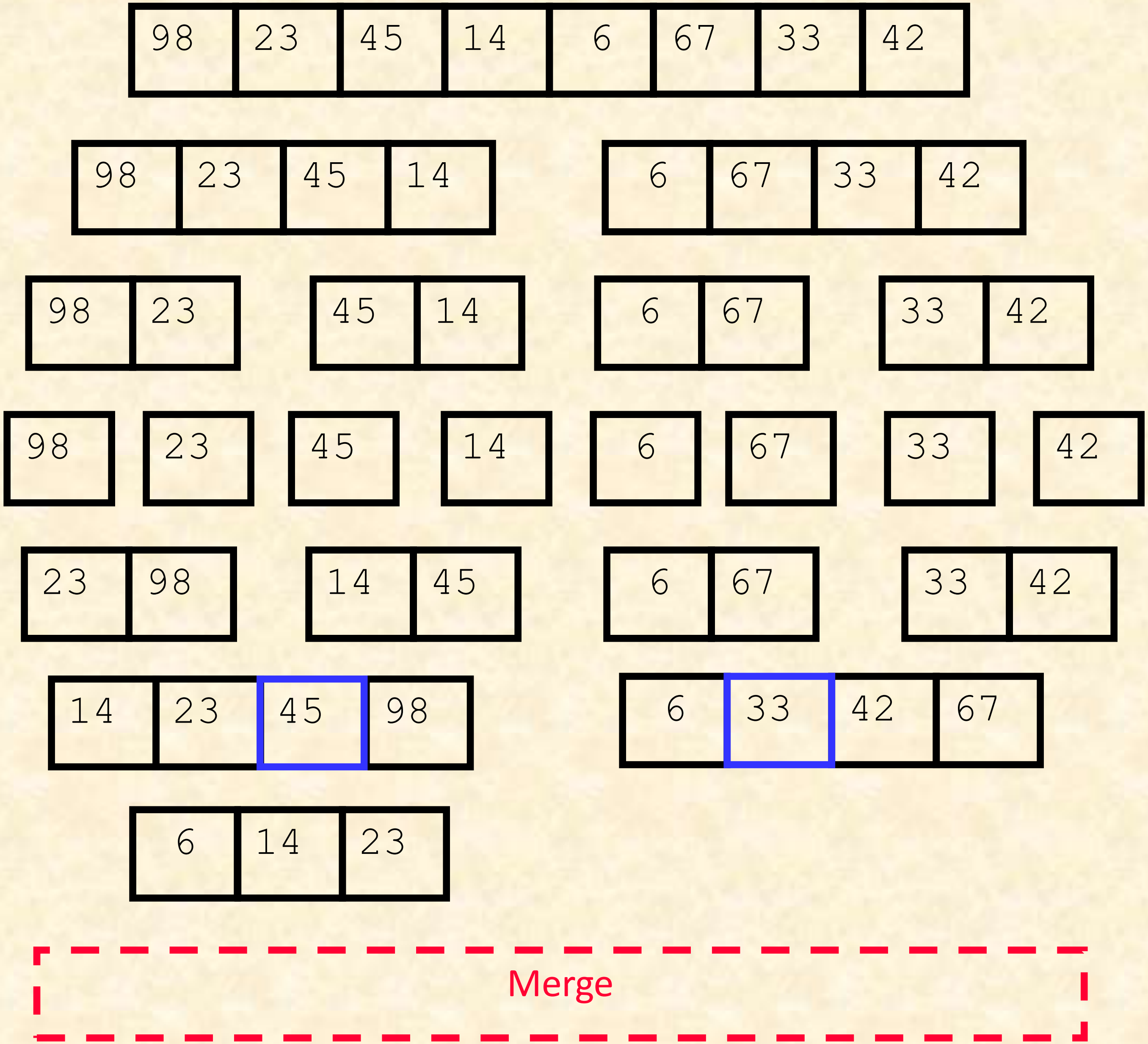
98 — 23 — 45 — 14 — 6 — 67 — 33 — 42

23 98 — 14 45 — 6 67 — 33 42

14 23 45 98 — 6 33 42 67

6 14

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

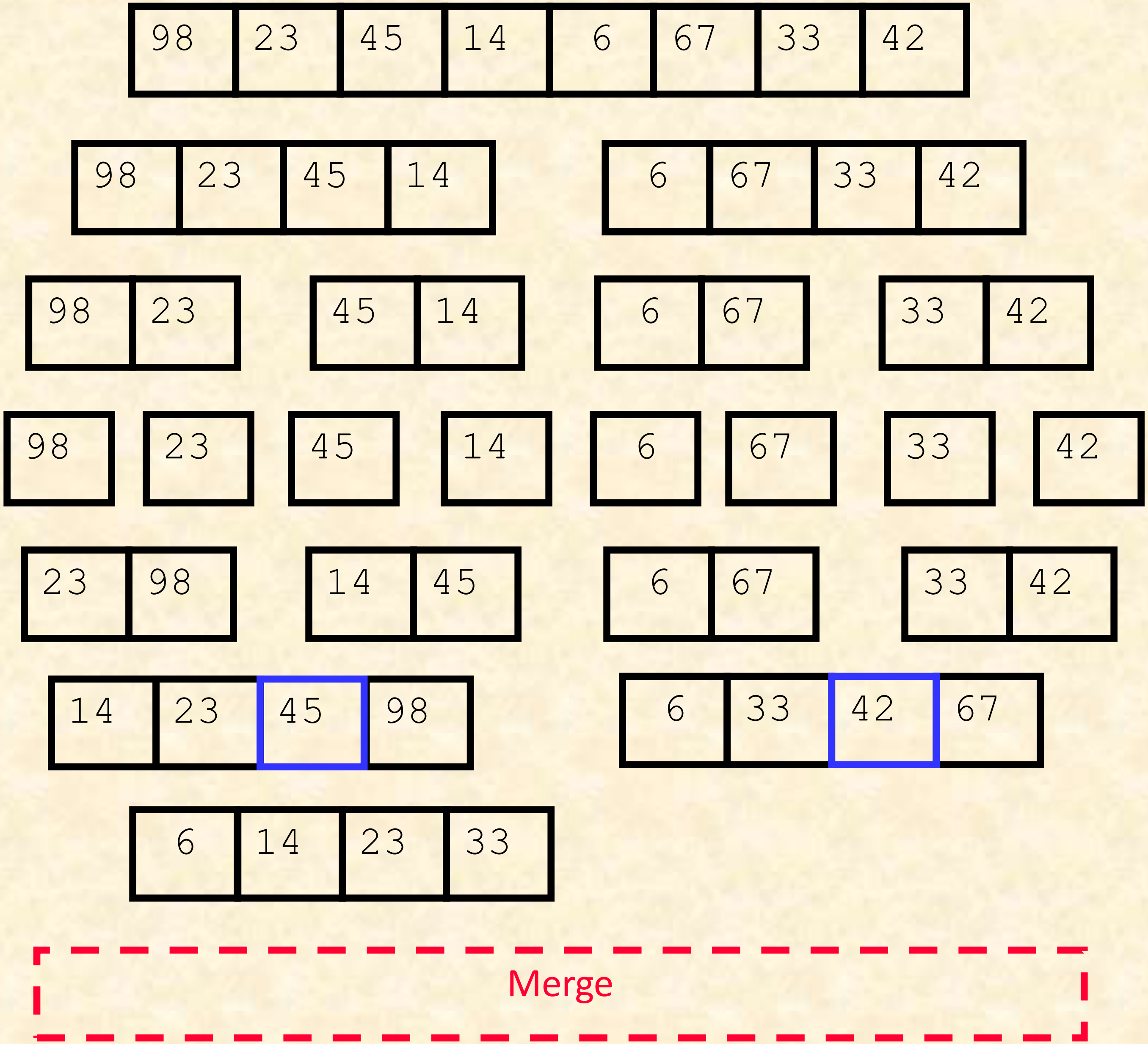
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

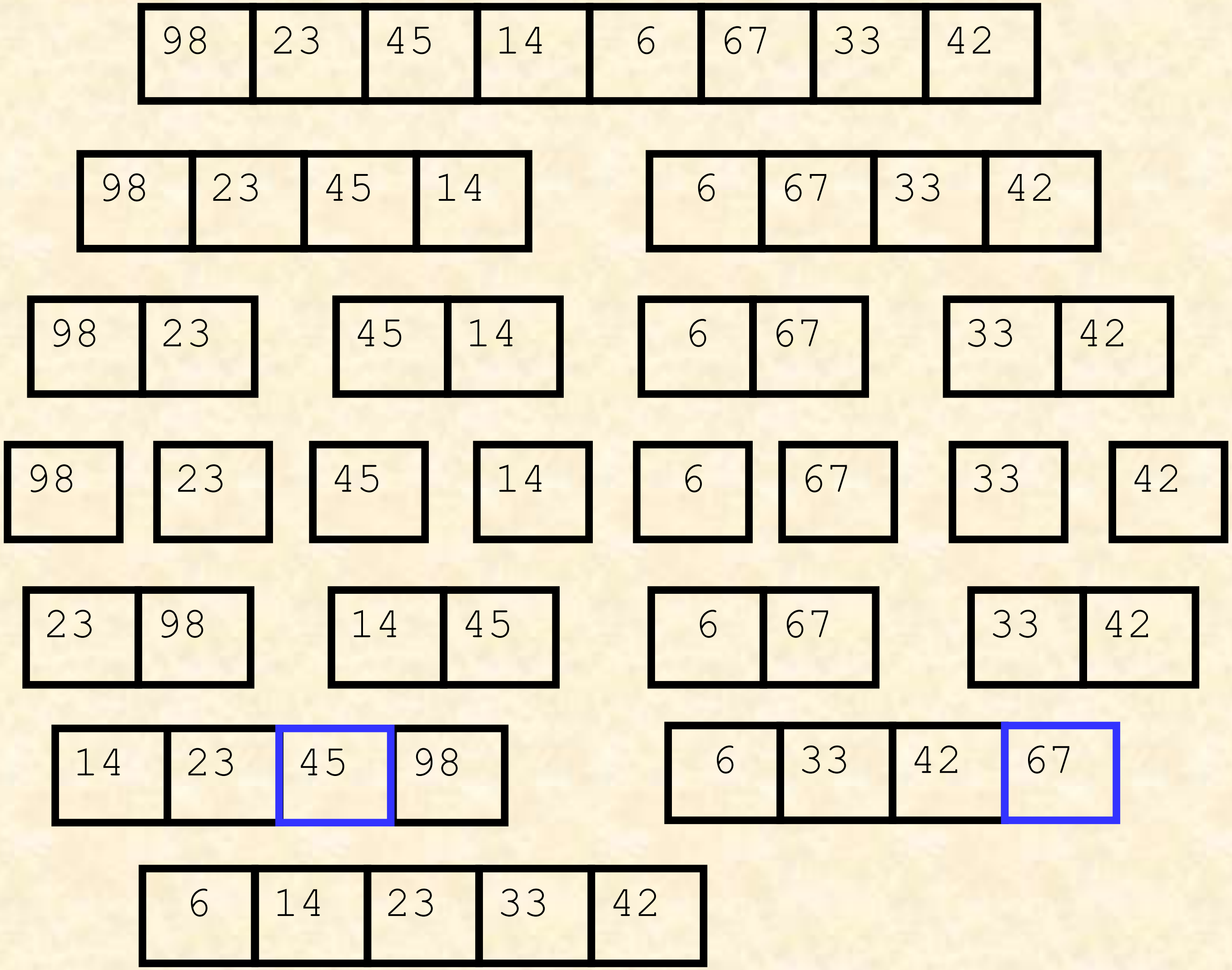
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6 14 23 33

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

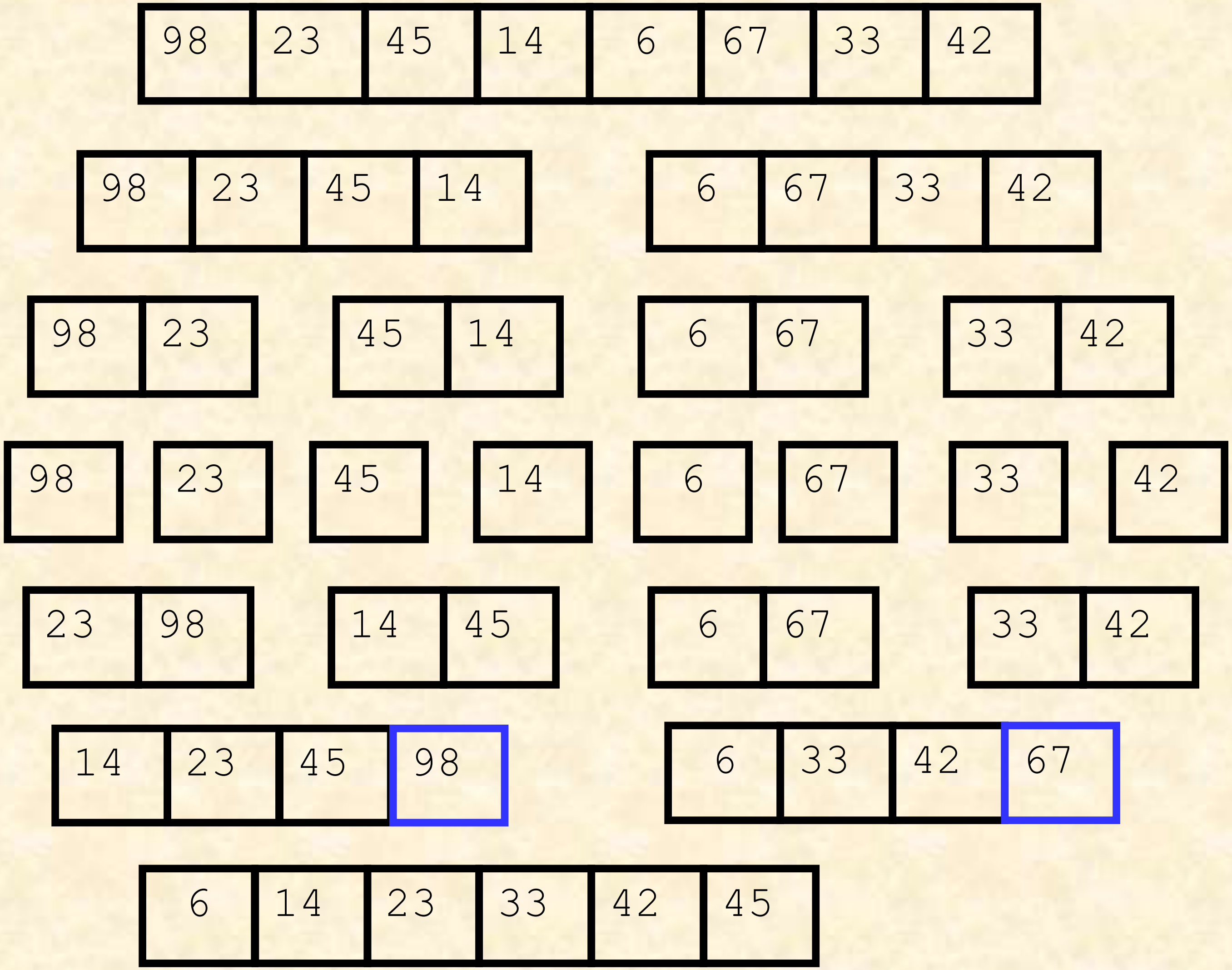
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6 14 23 33 42

Merge



98 23 45 14 6 67 33 42

98 23 45 14
6 67 33 42

98 23
45 14
6 67
33 42

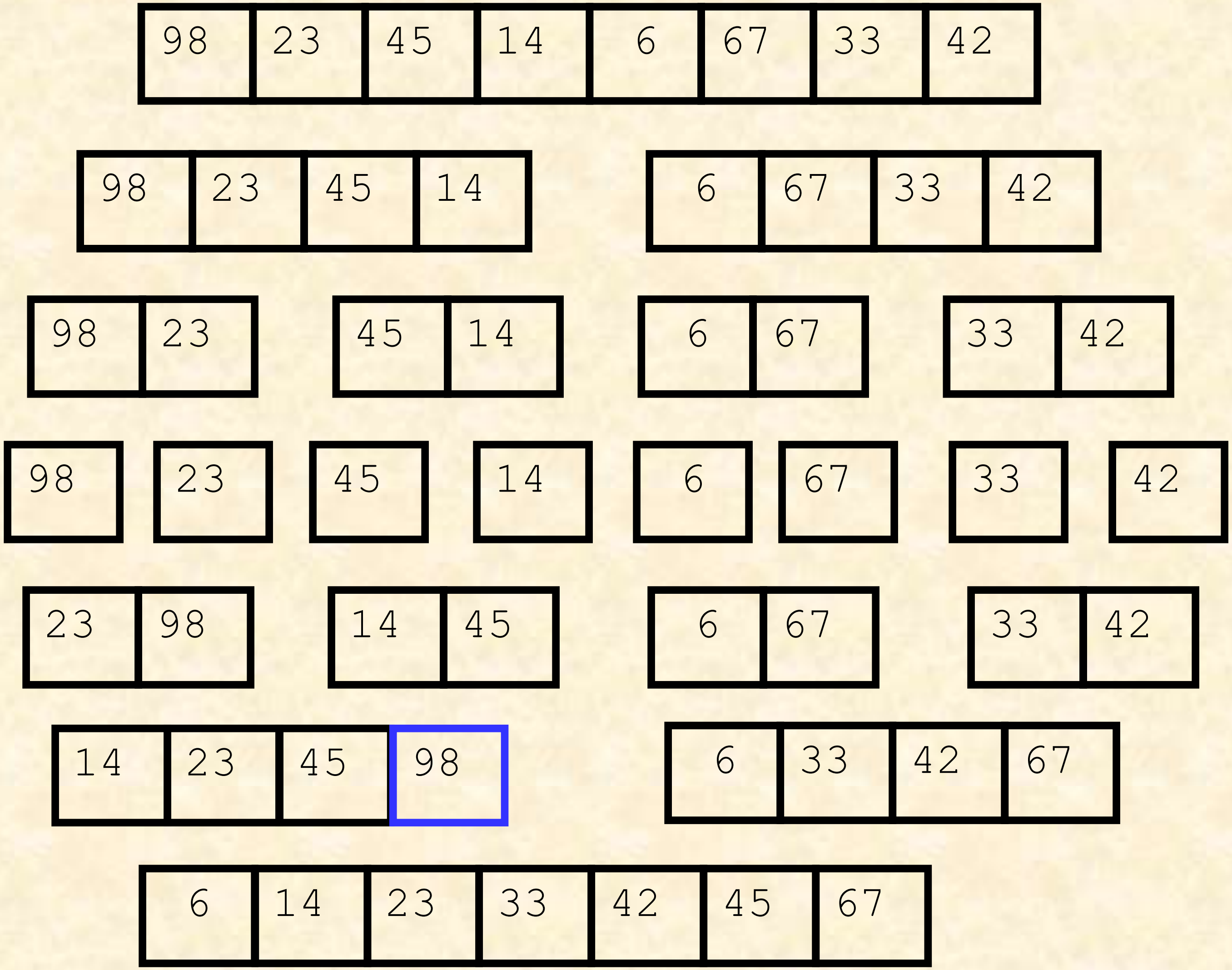
98
23
45
14
6
67
33
42

23 98
14 45
6 67
33 42

14 23 45 98
6 33 42 67

6 14 23 33 42 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

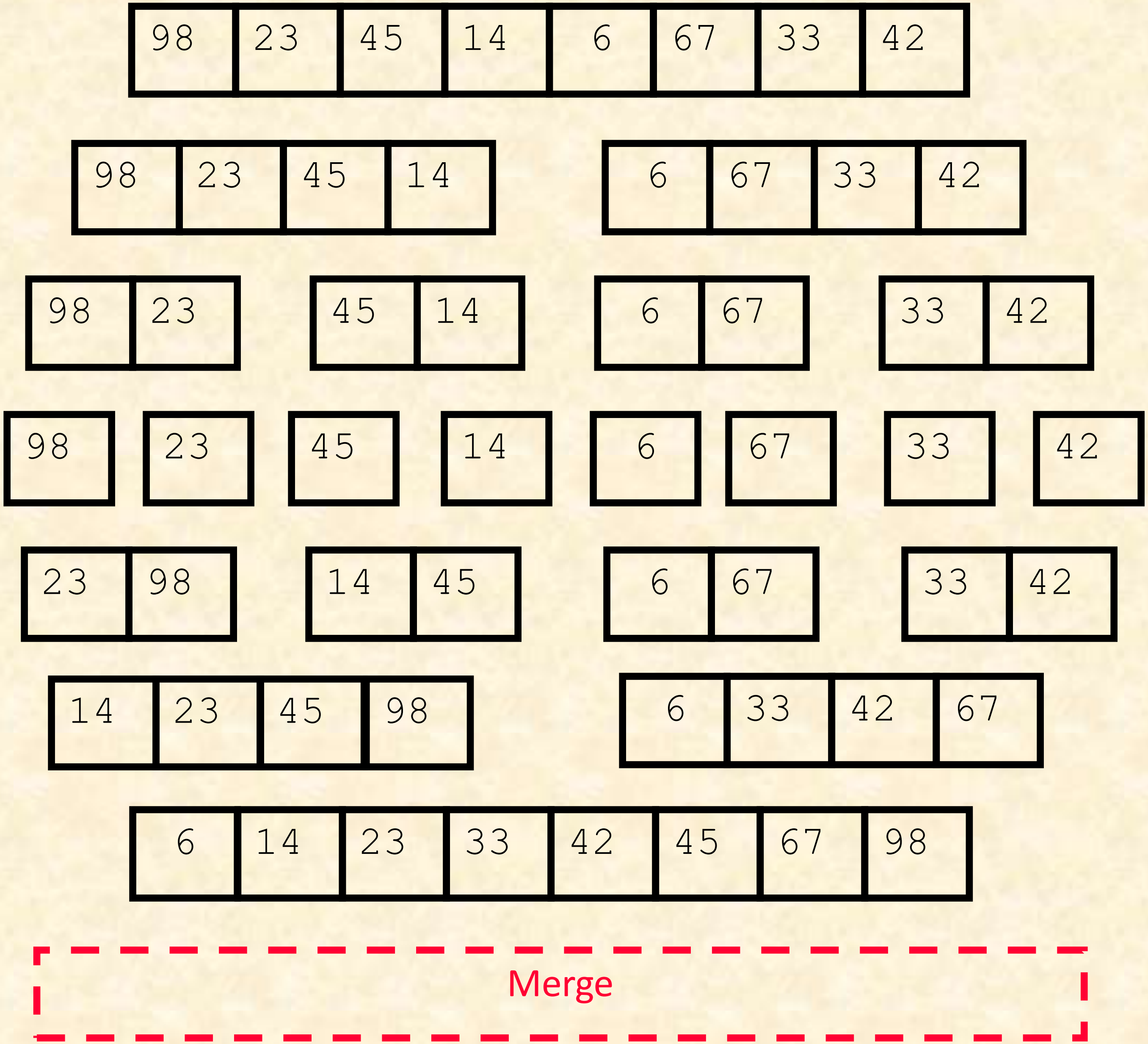
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

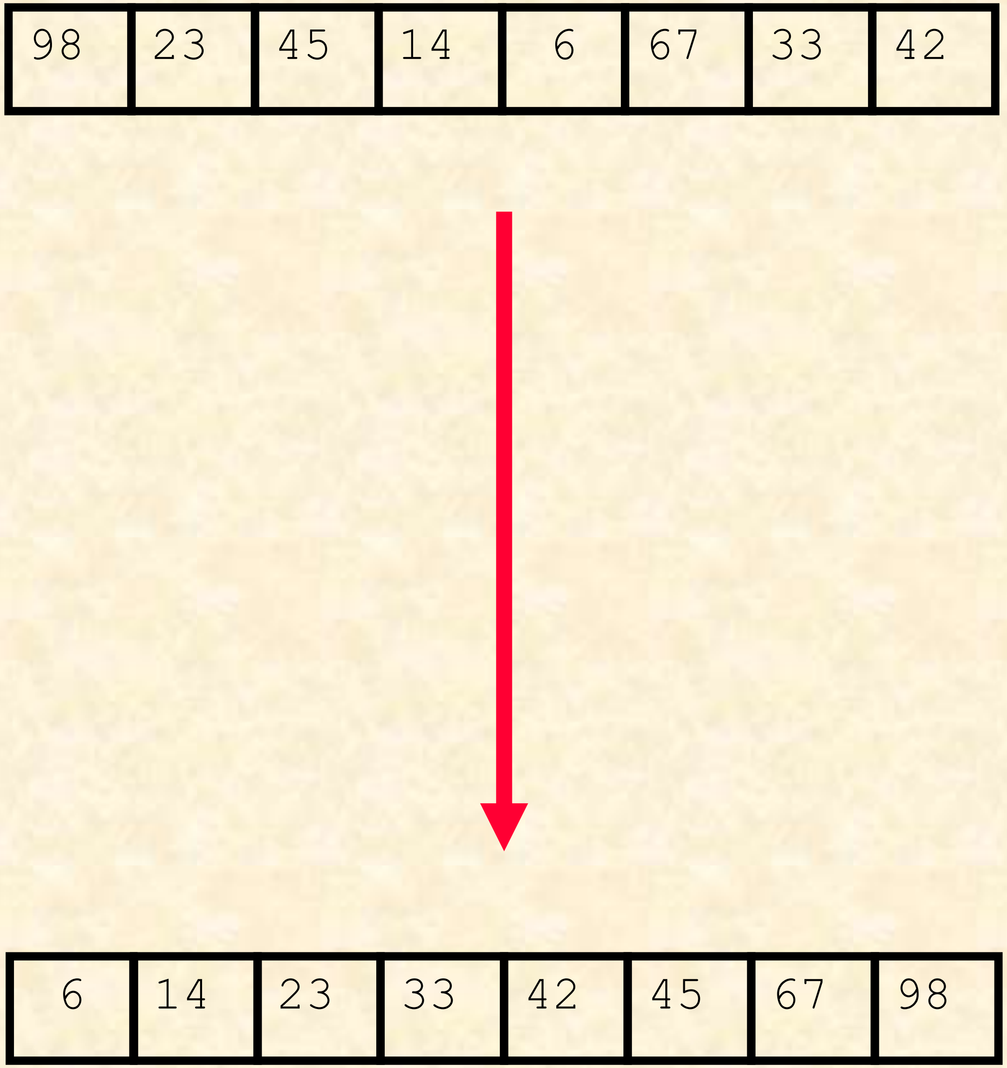
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98

Merge



98	23	45	14	6	67	33	42

6	14	23	33	42	45	67	98

Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n*

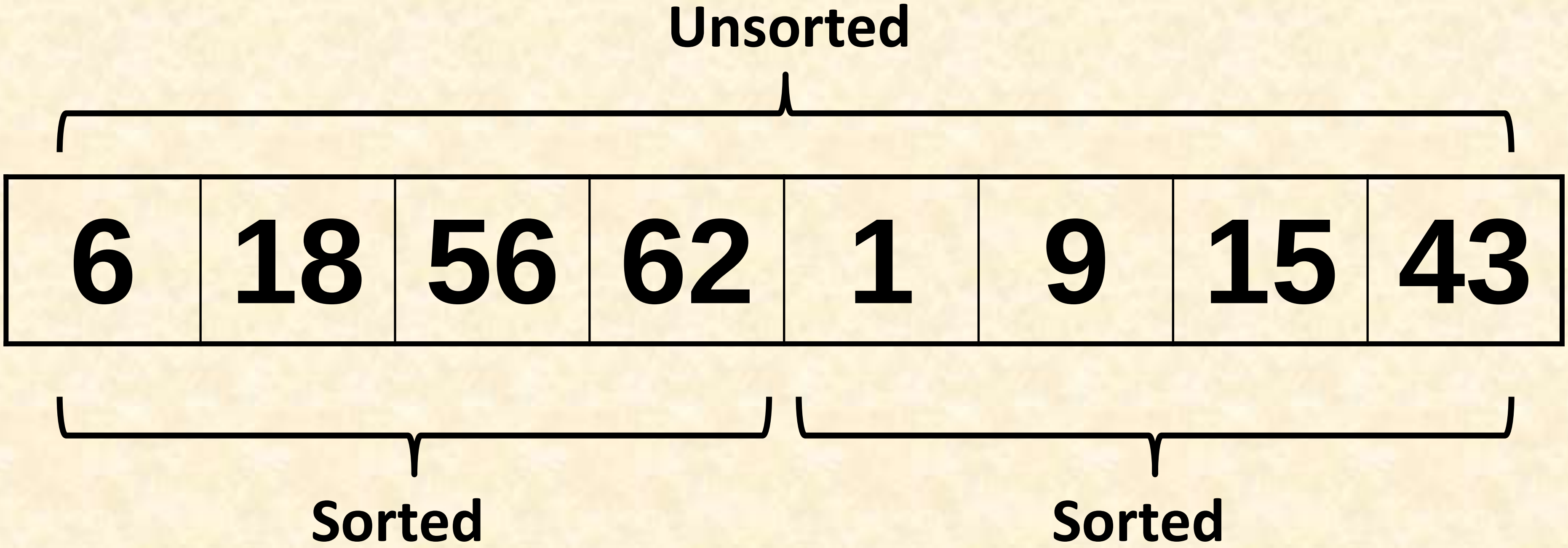
```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
if $p < r$
    then $q \leftarrow \lfloor (p+r)/2 \rfloor$
        MergeSort (A, p, q)
        MergeSort (A, q+1, r)
        Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(*A*, 1, *n*)

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging two sorted subsequeces



Unsorted

6	18	56	62	1	9	15	43

Sorted

Sorted

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1							

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1							

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6						

🟨	Left half
🟩	Right half
🟦	Minimum between first elements in both halves

Merging two sorted subsequeces

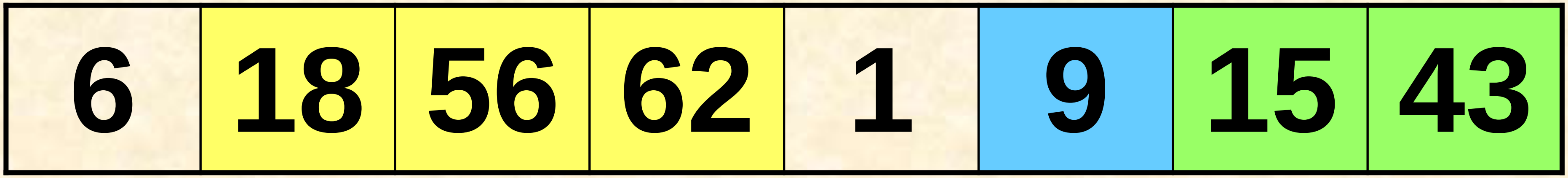
6	18	56	62	1	9	15	43

Merging

1	6						

🟨	Left half
🟩	Right half
🟦	Minimum between first elements in both halves

Merging two sorted subsequeces



6	18	56	62	1	9	15	43

Merging

1	6	9					

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6	9					

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	🟨 18	🟨 56	🟨 62	1	9	🟦 15	🟩 43

Merging

1	6	9	15				

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6	9	15				

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6	9	15	18			

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6	9	15	18			

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6	9	15	18	43		

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6	9	15	18	43		

🟨	Left half
🟩	Right half
🟦	Minimum between first elements in both halves

Merging two sorted subsequeces

6	18	56	62	1	9	15	43

Merging

1	6	9	15	18	43	56	62

🟨	Left half
🟩	Right half
🟦	Minimum between first elements in both halves

Merging two sorted subsequeces

1	6	9	15	18	43	56	62

Merging

1	6	9	15	18	43	56	62

🟨 Left half

🟩 Right half

🟦 Minimum between first elements in both halves

Merging two sorted subsequeces

```
Merge($A$, $p$, $q$, $r$)
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
    for $i \leftarrow 1$ to $n_1$
        do $L[i] \leftarrow A[p + i - 1]$
    for $j \leftarrow 1$ to $n_2$
        do $R[j] \leftarrow A[q + j]$
    $L[n_1+1] \leftarrow \infty$
    $R[n_2+1] \leftarrow \infty$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k \leftarrow p$ to $r$
        do if $L[i] \leq R[j]$
            then $A[k] \leftarrow L[i]$
                 $i \leftarrow i + 1$
            else $A[k] \leftarrow R[j]$
                 $j \leftarrow j + 1$
```

Input: Array containing sorted subarrays A[p..q] and A[q+1..r].

Output: Merged sorted subarray in A[p..r].

***Sentinels**, to avoid having to check if either subarray is fully copied at each step.*

Time complexity of Merge

```
Merge(A, p, q, r)   //Let r-p+1 = n
$n_1 \gets q - p + 1$   //O(1)
$n_2 \gets r - q$   //O(1)
for $i \gets 1$ to $n_1$   //O(q-p+1)
    do $L[i] \gets A[p + i - 1]$
for $j \gets 1$ to $n_2$   //O(r-q)
    do $R[j] \gets A[q + j]$
$L[n_1+1] \gets \infty$
$R[n_2+1] \gets \infty$
$i \gets 1$
$j \gets 1$
for $k \gets p$ to $r$   //O(r-p+1) = O(n)
    do if $L[i] \le R[j]$
        then $A[k] \gets L[i]$
                 $i \gets i + 1$
        else $A[k] \gets R[j]$
                 $j \gets j + 1$
//Total time: O(n)
```

Input: Array containing sorted subarrays A[p..q] and A[q+1..r].

Output: Merged sorted subarray in A[p..r].