Laboratory 12: Sorting Algorithms part-2

CSC205A Data structures and Algorithms Laboratory B. Tech. 2015

Vaishali R Kulkarni

Department of Computer Science and Engineering
Faculty of Engineering and Technology
M. S. Ramaiah University of Applied Sciences
Email: vaishali.cs.et@msruas.ac.in
Tel: +91-80-4906-5555 (2212) WWW: www.msruas.ac.in

Introduction and Purpose of Experiment

- Sorting provide us with means of organising information to facilitate the retrieval of specific data.
- Searching methods are designed to take advantage of the organisation of information.
- By solving these problems students will be able to use sorting algorithms to sort a randomly ordered set of numbers, and search for key element.

Aim and objectives

Aim:

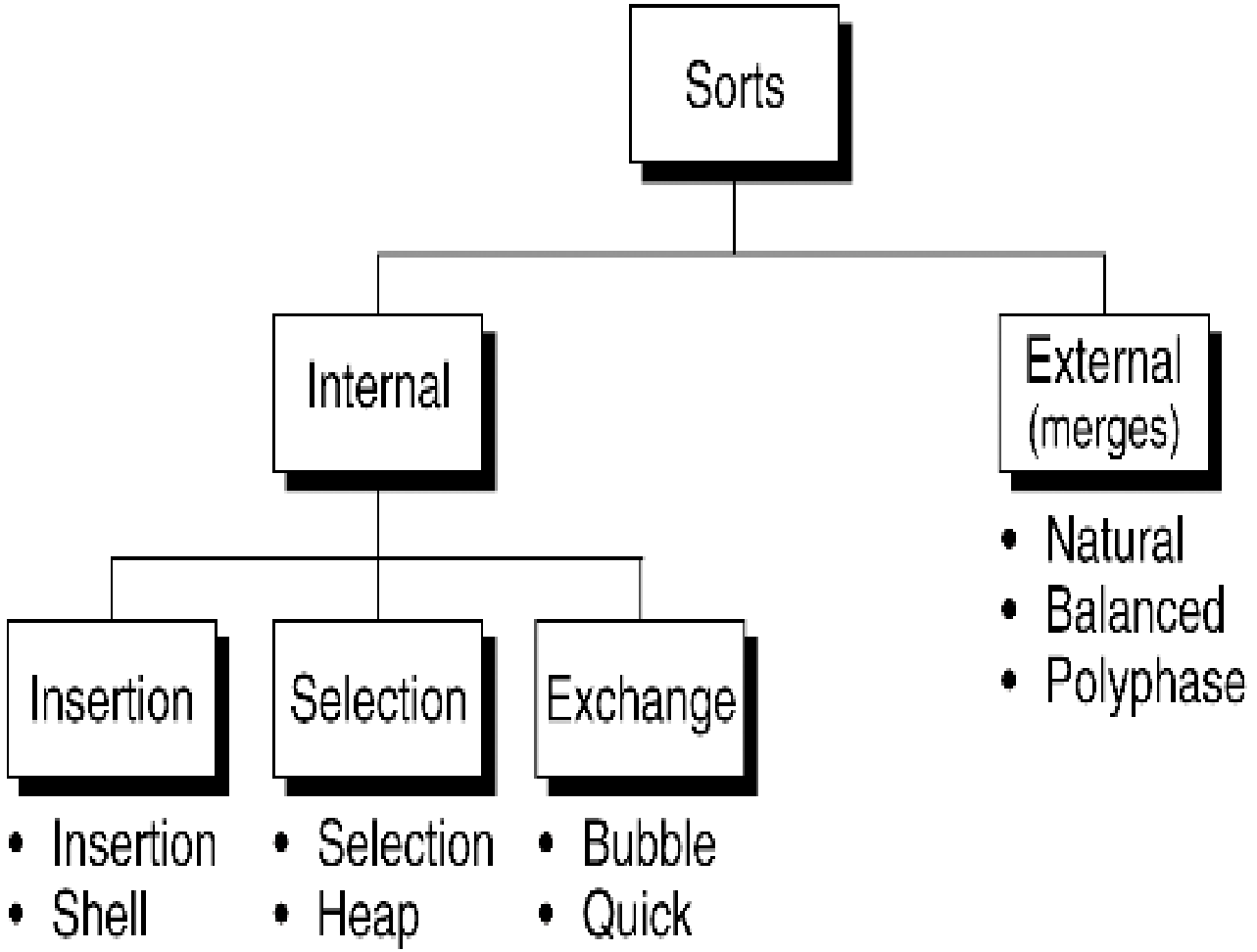
- To design and develop C programs to sort the given data using quick sort and merge sort, different sorting techniques

Objectives:

At the end of this lab, the student will be able to

- Create C programs using sorting algorithms such as **quick sort**
- Create C programs using sorting algorithms such as merge **sort**
- Analyse the efficiency of implemented sort algorithms

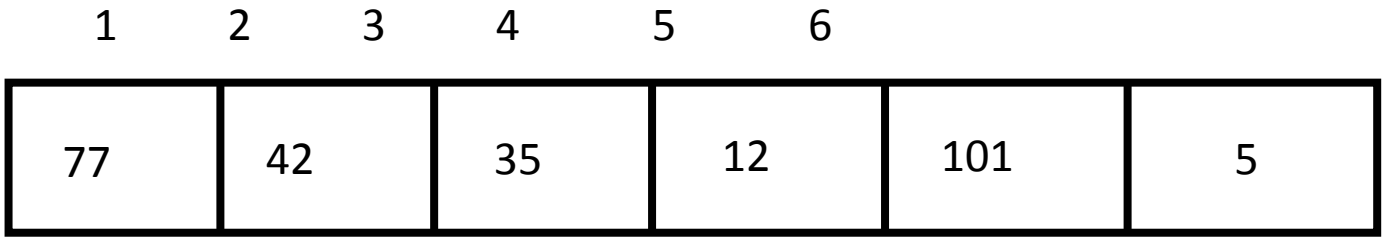
Sort Classifications



Bubble Sort

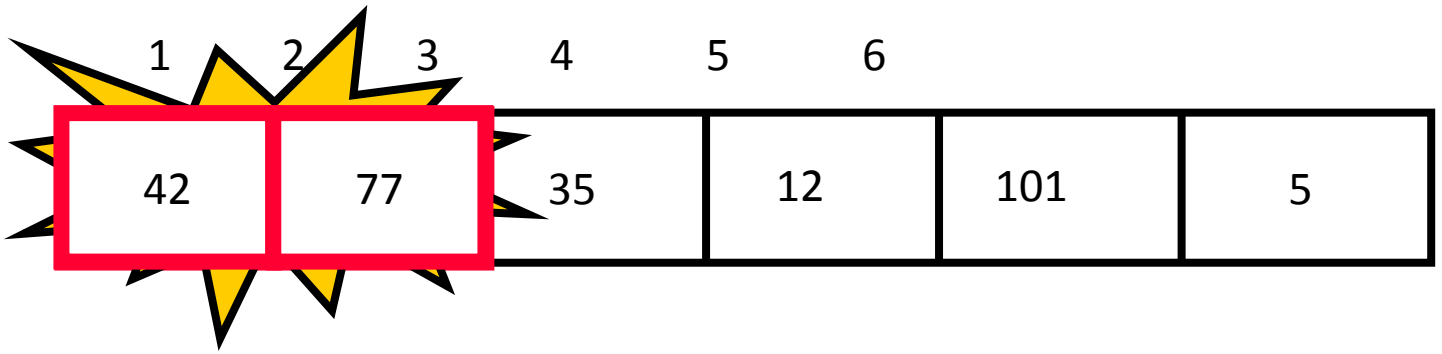
"Bubbling Up" the Largest Element

- **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**



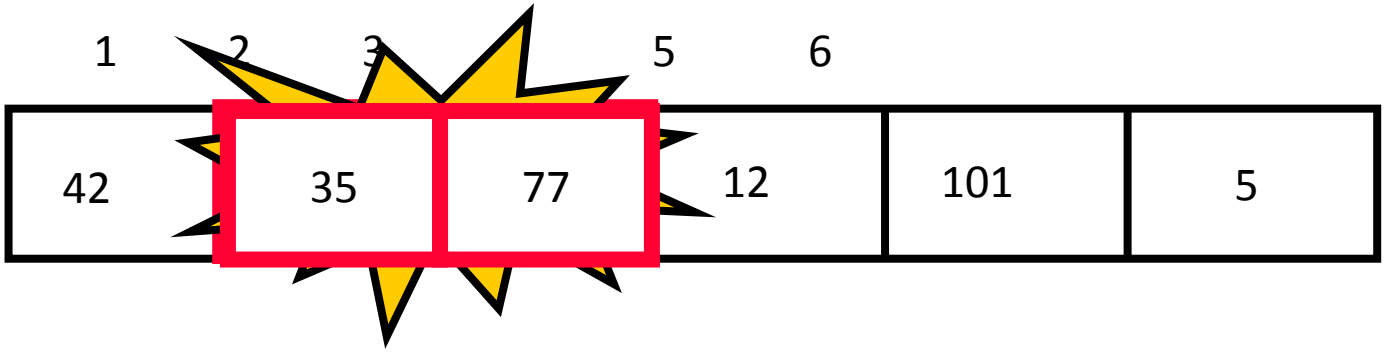
"Bubbling Up" the Largest Element

- **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**



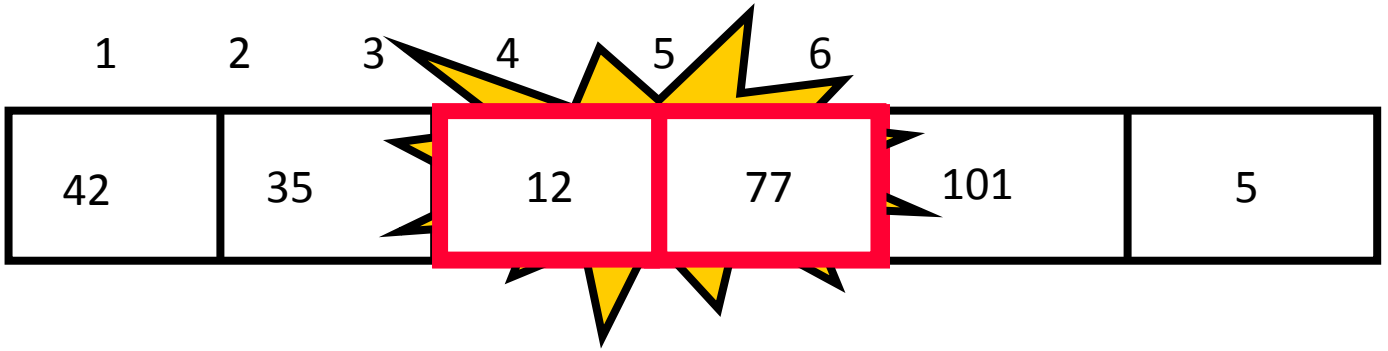
"Bubbling Up" the Largest Element

- **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**



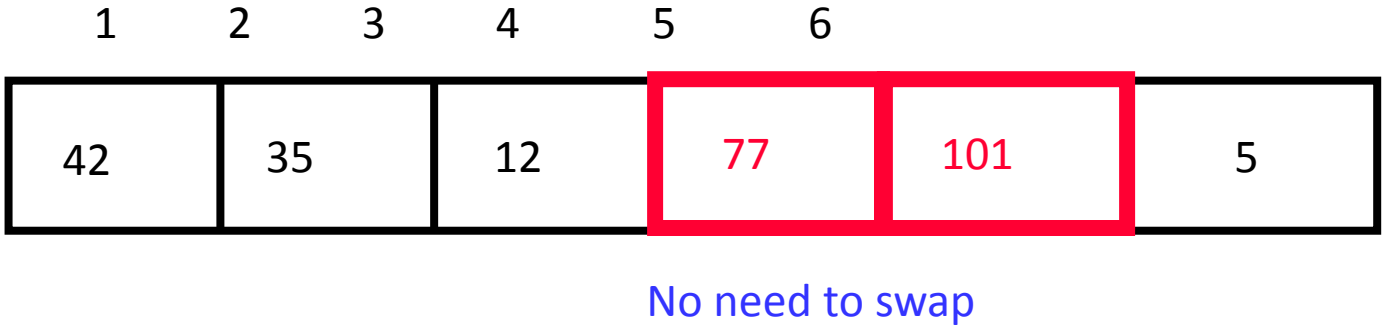
"Bubbling Up" the Largest Element

- **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**



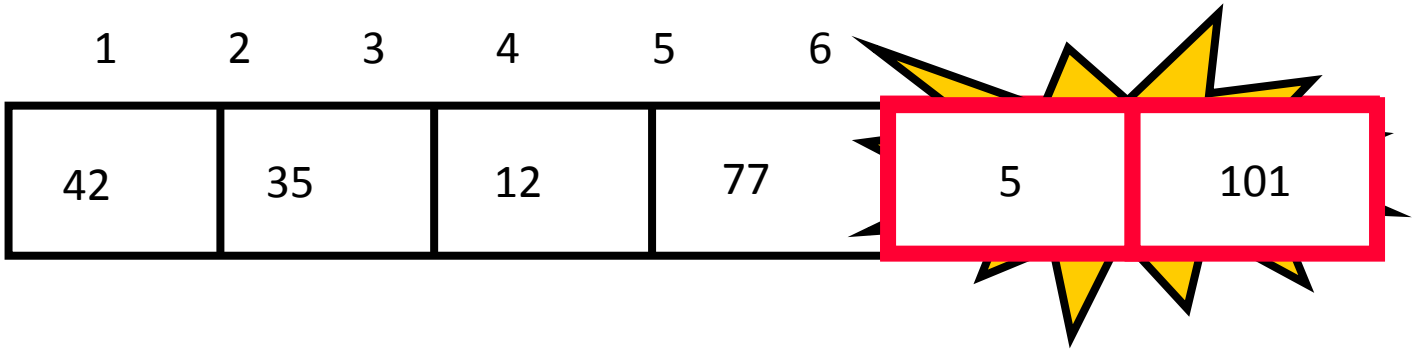
"Bubbling Up" the Largest Element

- **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**



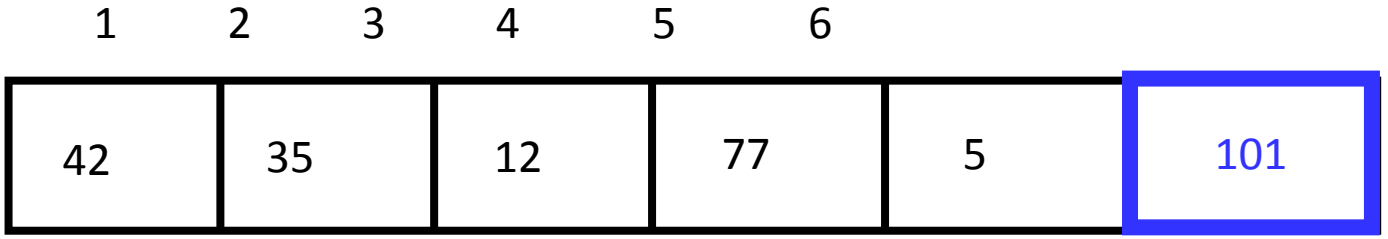
"Bubbling Up" the Largest Element

- **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**



"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**



The “Bubble Up” Algorithm

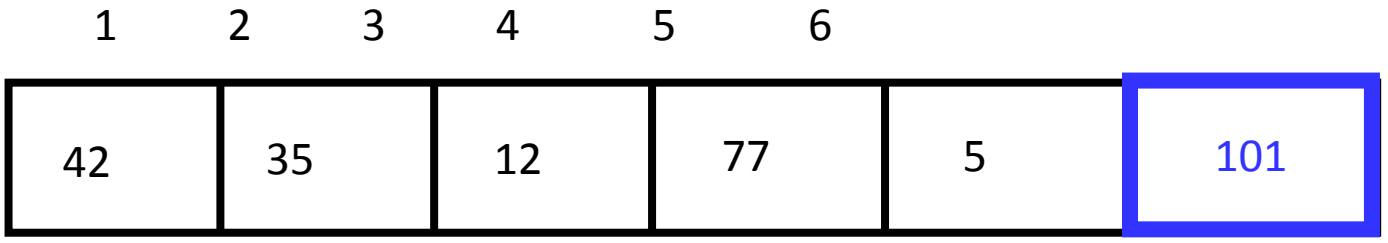
```
index <- 1
last_compare_at <- n - 1

loop
  exitif(index > last_compare_at)
  if(A[index] > A[index + 1]) then
    Swap(A[index], A[index + 1])
  endif
  index <- index + 1
endloop
```

No, Swap isn't built in.

```
Procedure Swap(a, b isoftype in/out
  Num)
   t isoftype Num
   t <- a
   a <- b
   b <- t
endprocedure // Swap
```

Items of Interest

- **Notice that only the largest value is correctly placed**
- **All other values are still out of order**
- **So we need to repeat this process**

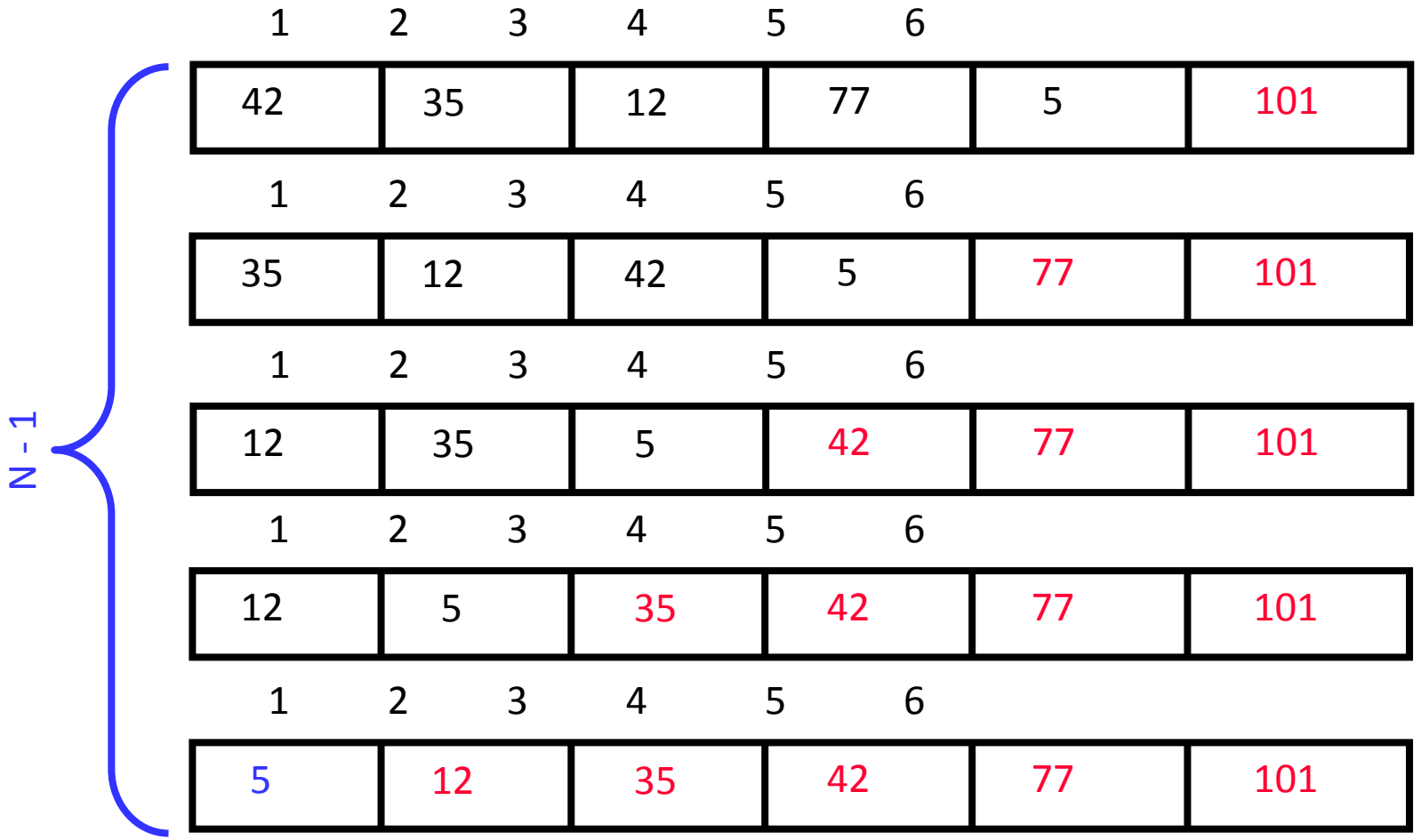


Largest value correctly placed

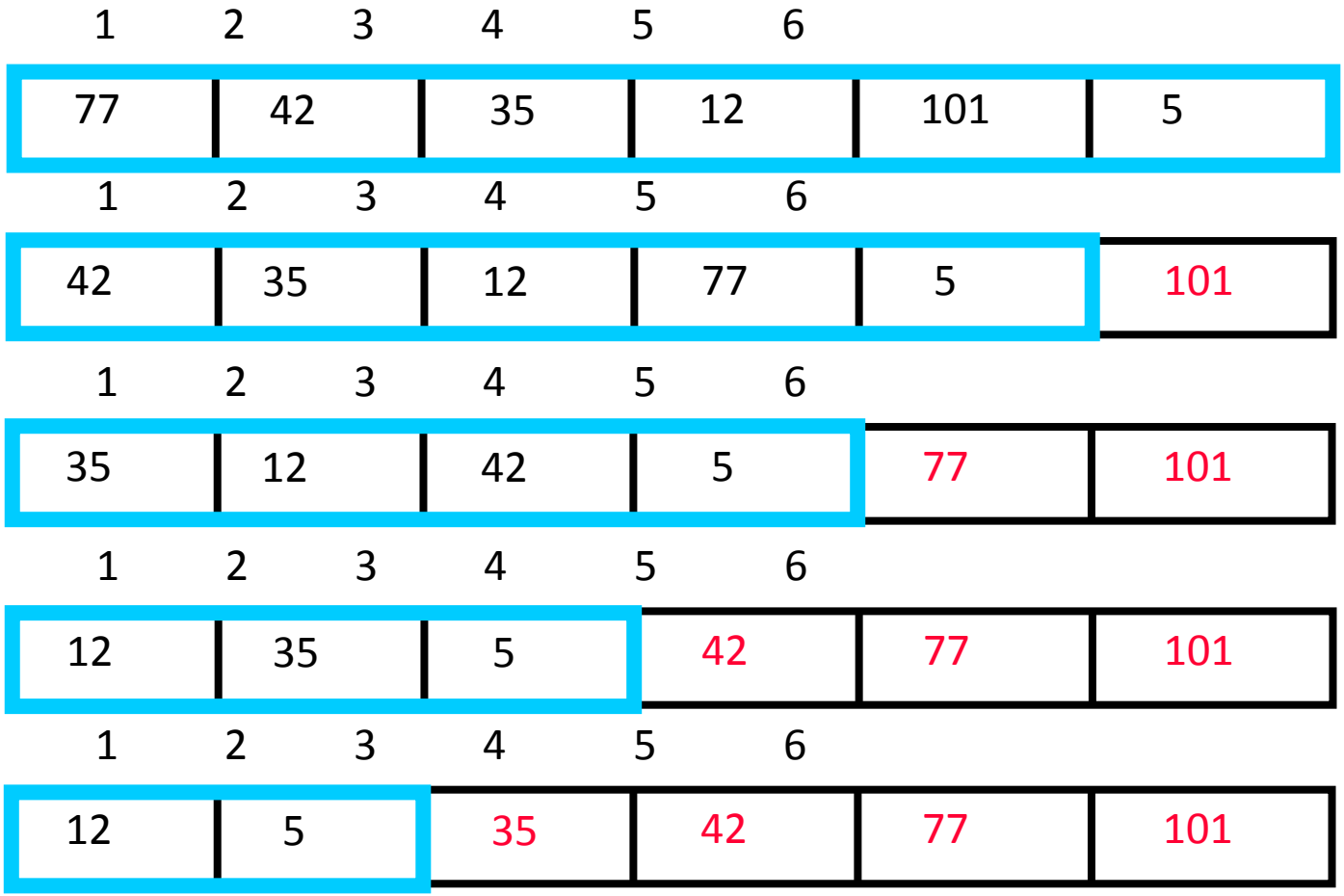
Repeat "Bubble Up" How Many Times?

- **If we have N elements…**
- **And if each time we bubble an element, we place it in its correct location…**
- **Then we repeat the "bubble up" process N – 1 times.**
- **This guarantees we'll correctly place all N elements.**

“Bubbling” All the Elements



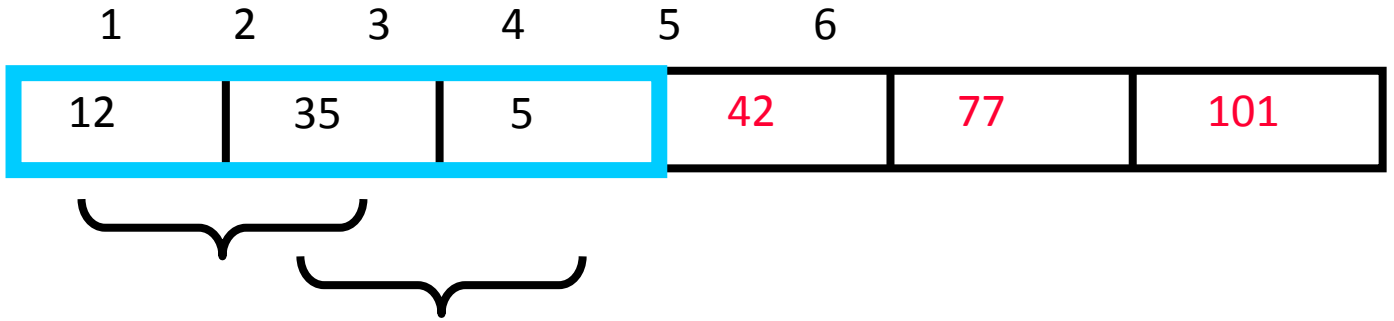
Reducing the Number of Comparisons



Reducing the Number of Comparisons

- **On the Nth "bubble up", we only need to do MAX-N comparisons.**

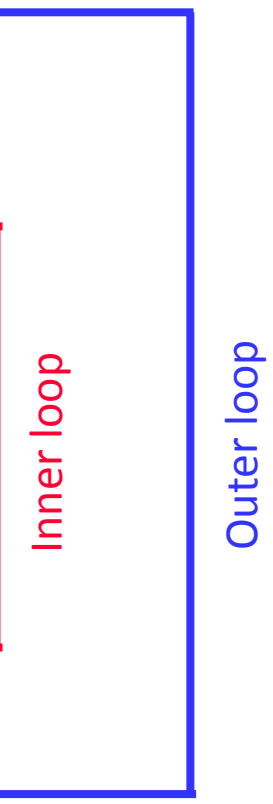
- **For example:**
  - **This is the 4th "bubble up"**
  - **MAX is 6**
  - **Thus we have 2 comparisons to do**



Putting It All Together

```
N is … // Size of Array

Arr_Type defines a Array[1..N] of Num

Procedure Swap(n1, n2 is of type int Num)
  temp is of type Num
  temp <- n1
  n1   <- n2
  n2   <- temp
end procedure // Swap
```

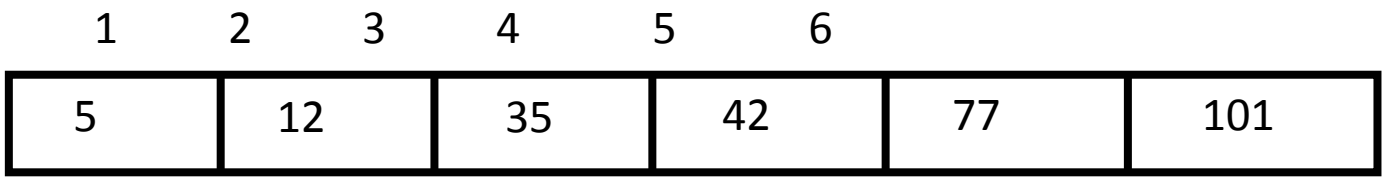
```
procedure Bubblesort(A is of type int Arr_Type)
  to_do, index is of type Num
  to_do <- N - 1

  loop
    exitif(to_do = 0)
    index <- 1
    loop
      exitif(index > to_do)
      if(A[index] > A[index + 1]) then
        Swap(A[index], A[index + 1])
      endif
      index <- index + 1
    endloop
    to_do <- to_do - 1
  endloop
end procedure // Bubblesort
```



Already Sorted Collections?

- **What if the collection was already sorted?**
- **What if only a few elements were out of place and after a couple of "bubble ups," the collection was sorted?**

- **We want to be able to detect this and "stop early"!**

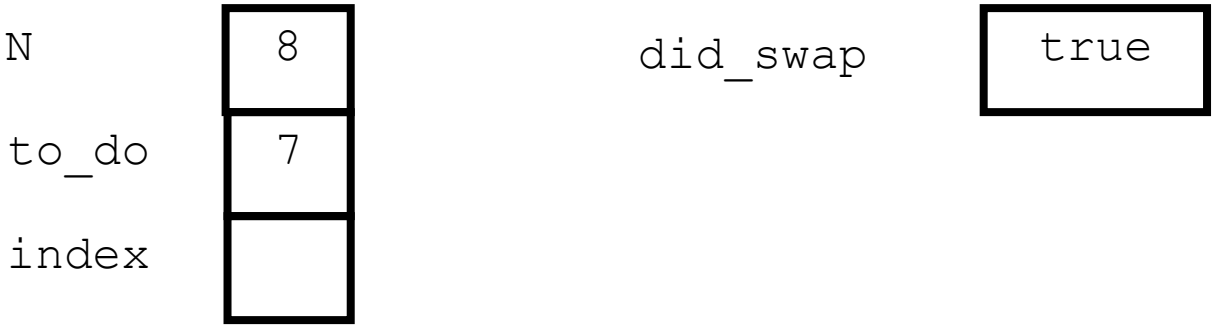


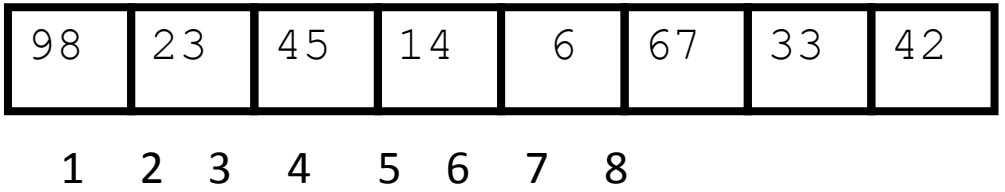
Using a Boolean "Flag"

- **We can use a boolean variable to determine if any swapping occurred during the "bubble up."**
- **If no swapping occurred, then we know that the collection is already sorted!**
- **This boolean "flag" needs to be reset after each "bubble up."**

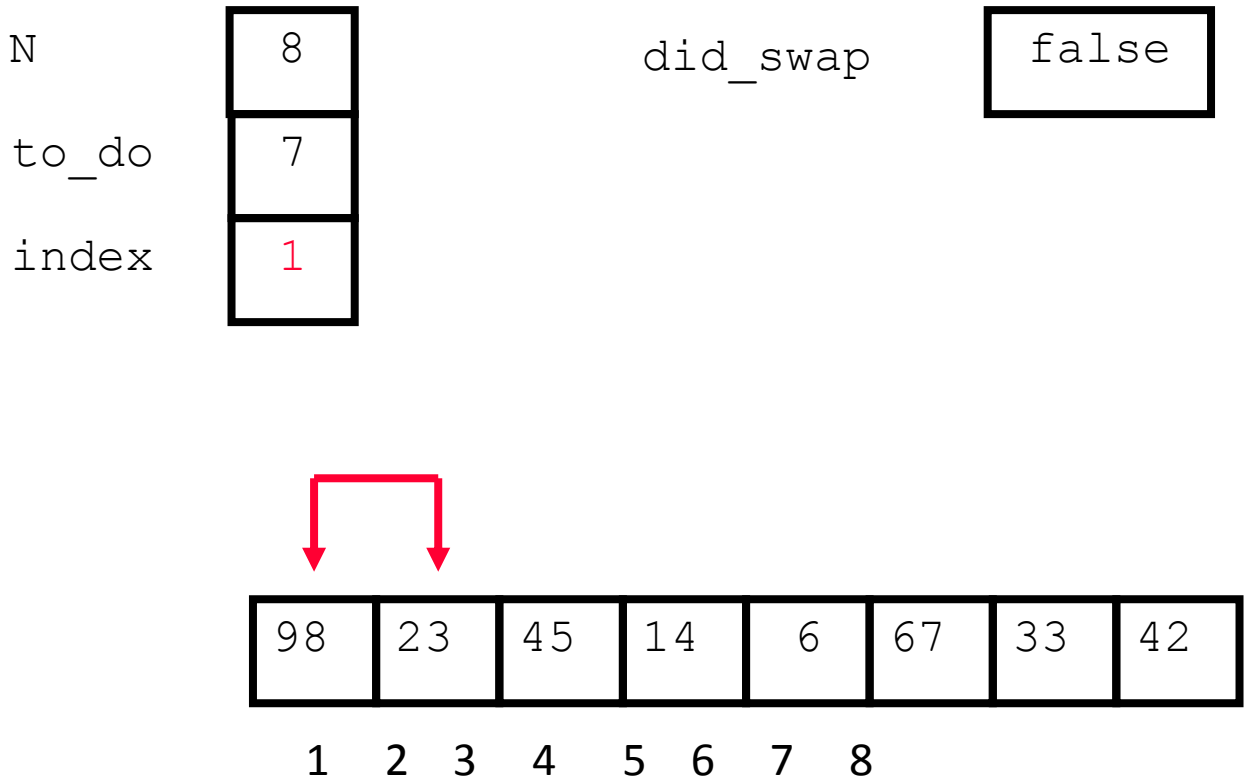
```
did_swap isoftype Boolean
did_swap <- true

loop
  exitif ((to_do = 0) OR NOT(did_swap))
  index <- 1
  did_swap <- false
  loop
    exitif(index > to_do)
    if(A[index] > A[index + 1]) then
      Swap(A[index], A[index + 1])
      did_swap <- true
    endif
    index <- index + 1
  endloop
  to_do <- to_do - 1
endloop
```

An Animated Example

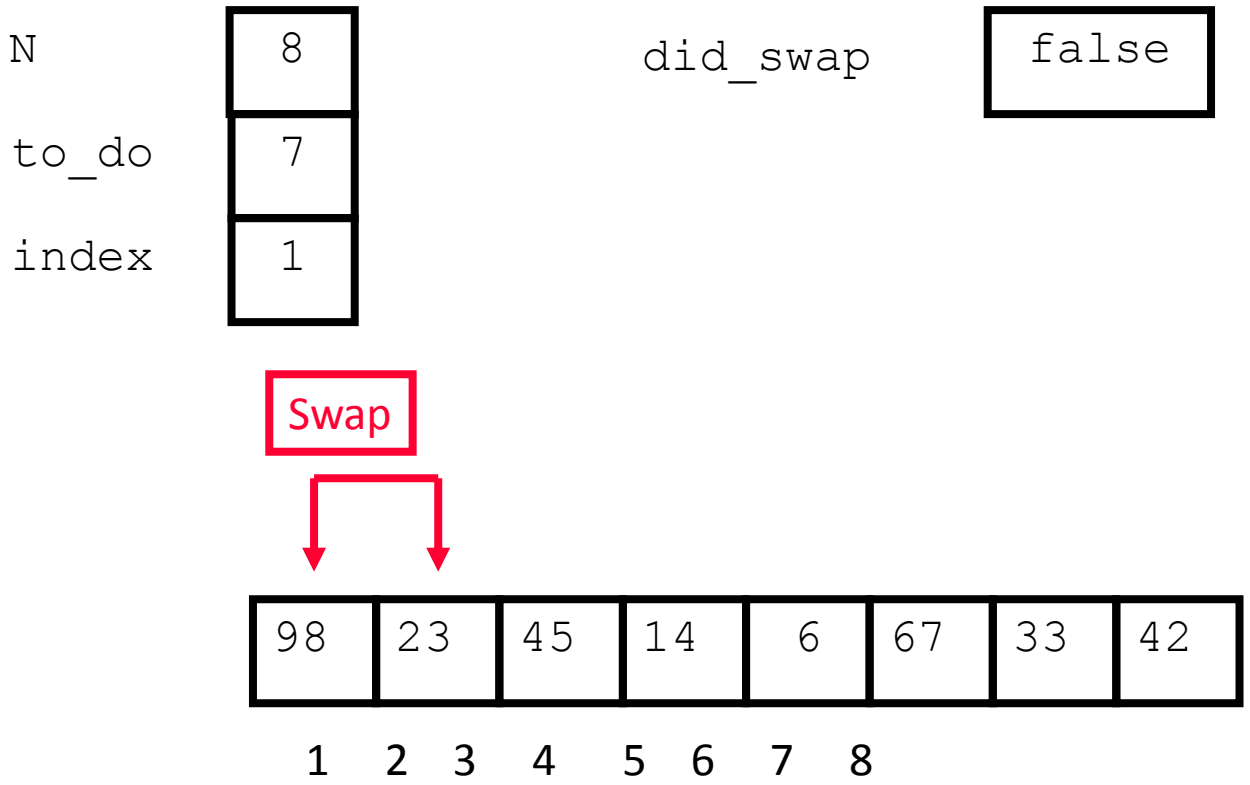




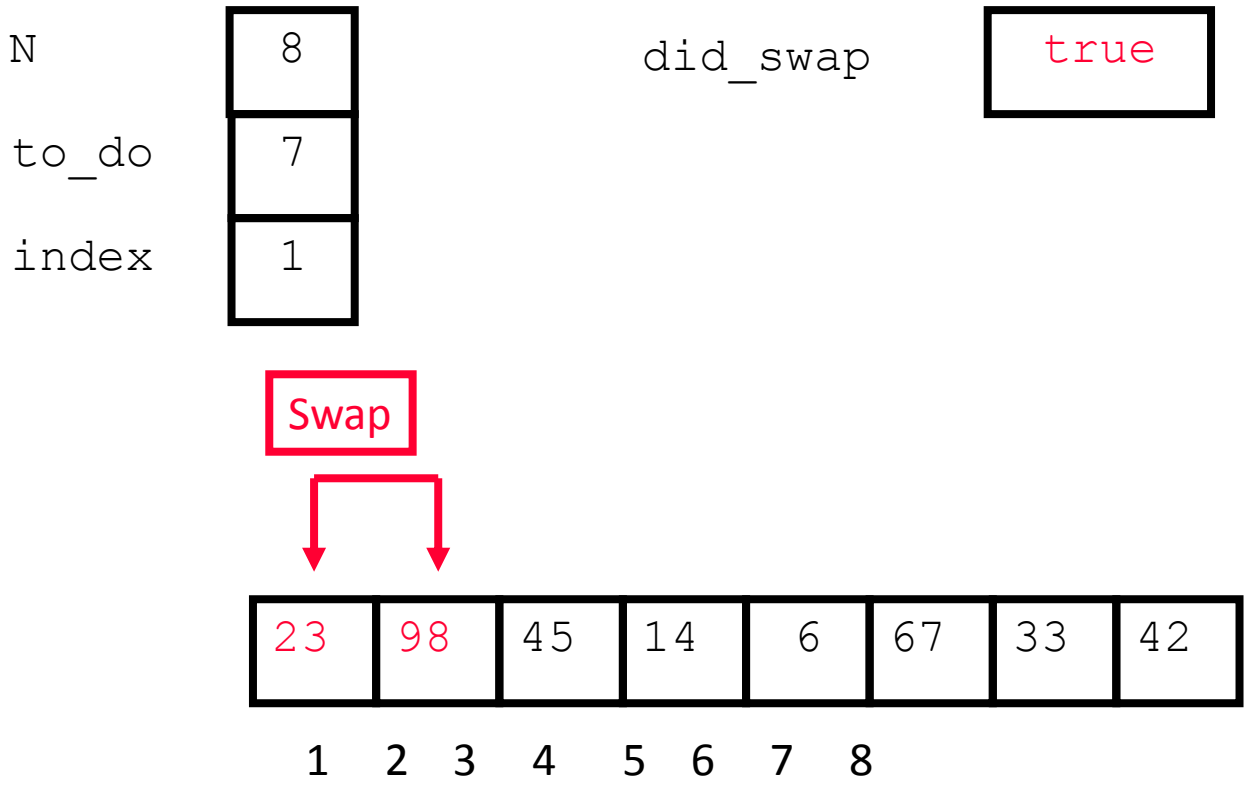
An Animated Example



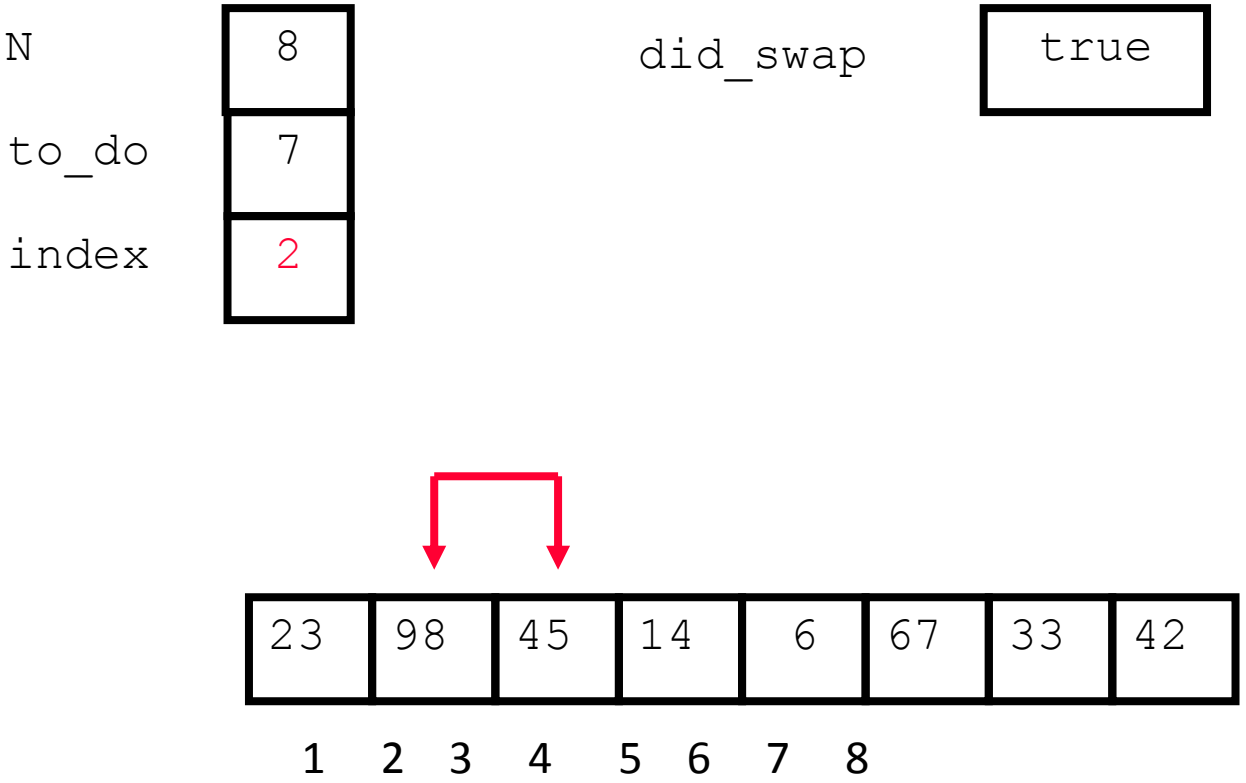
An Animated Example



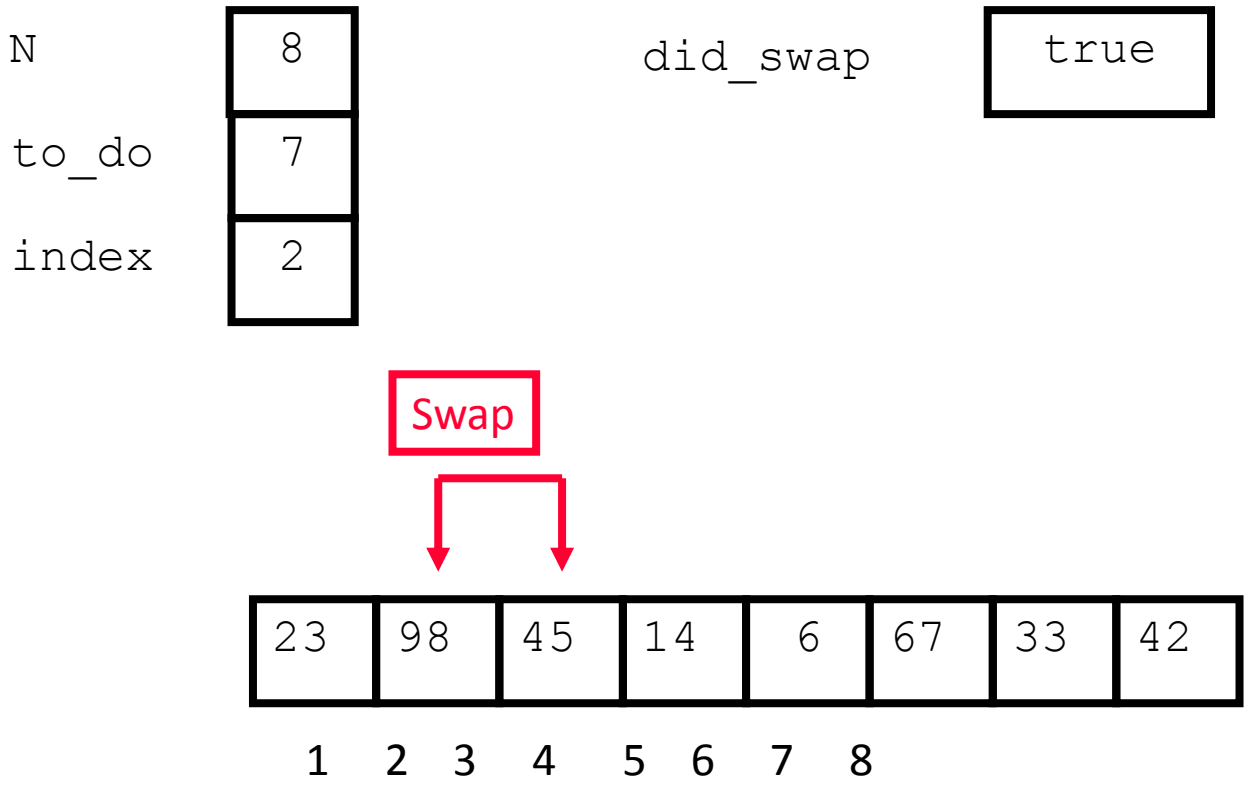
An Animated Example



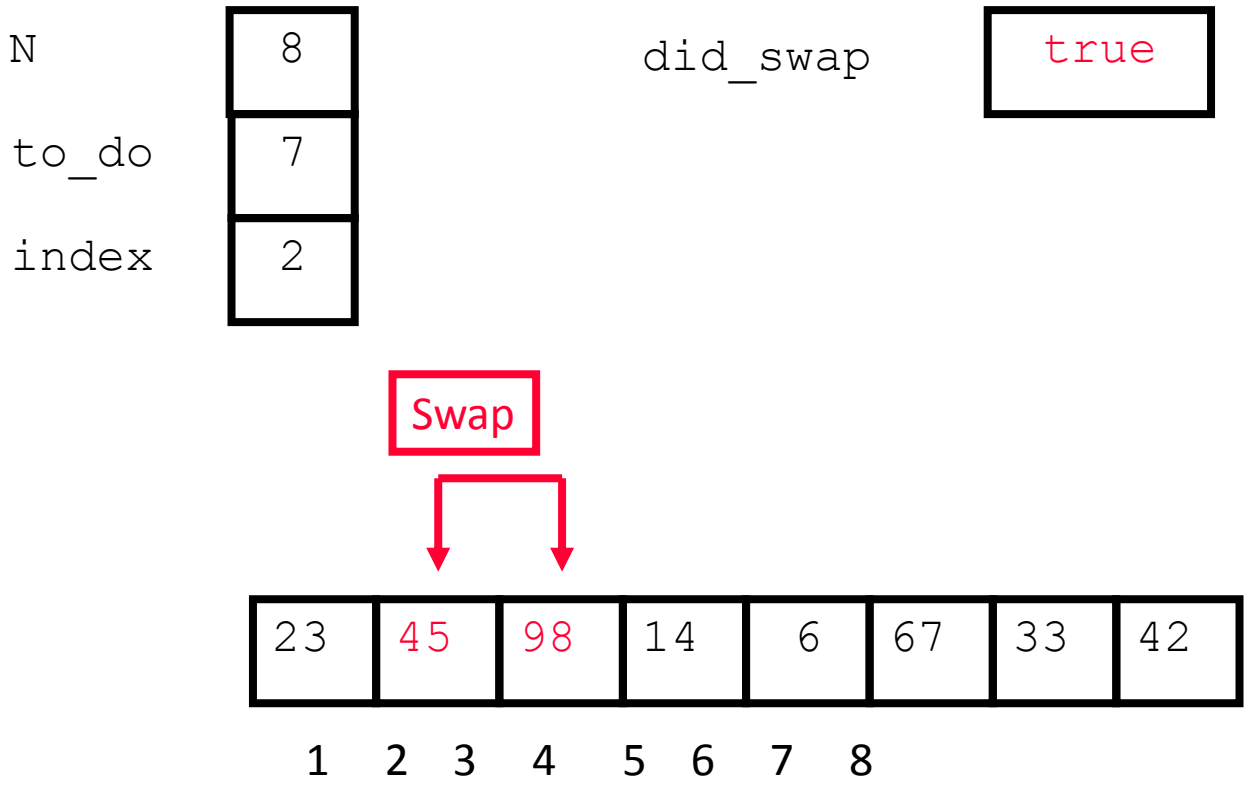
An Animated Example



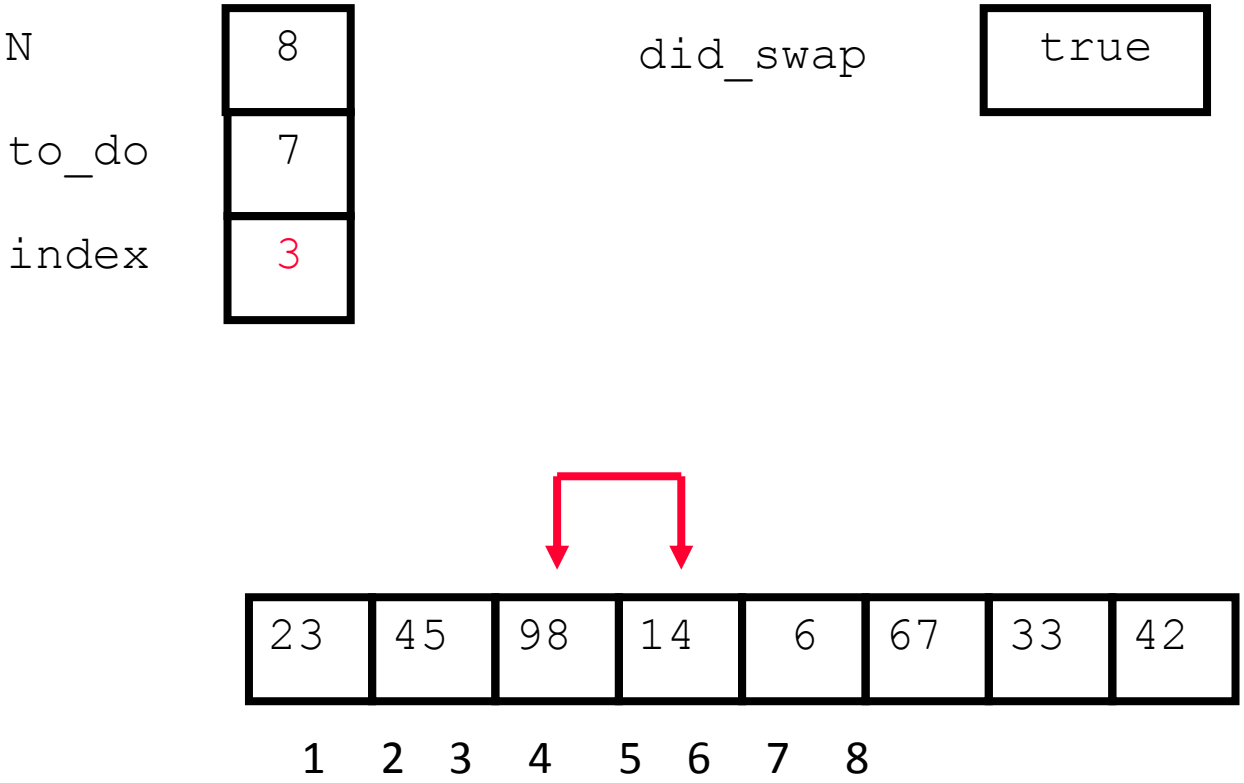
An Animated Example



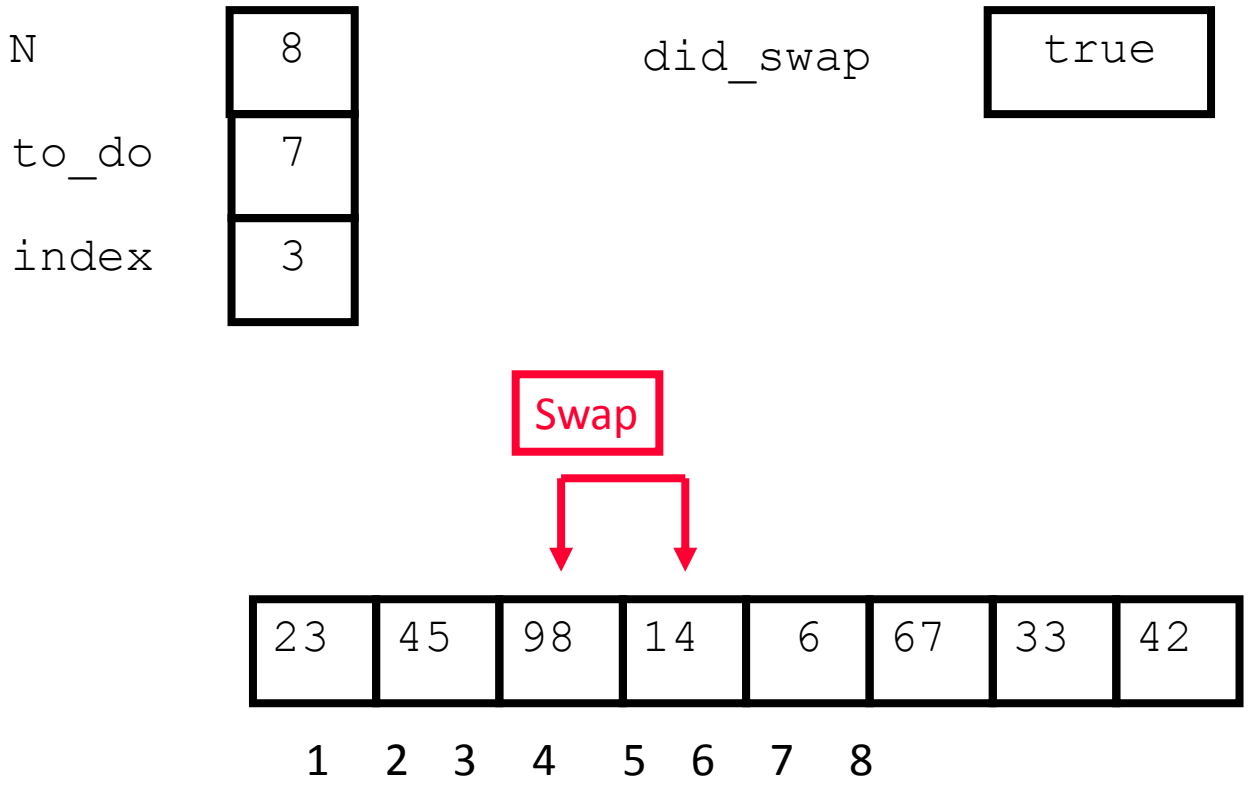
An Animated Example



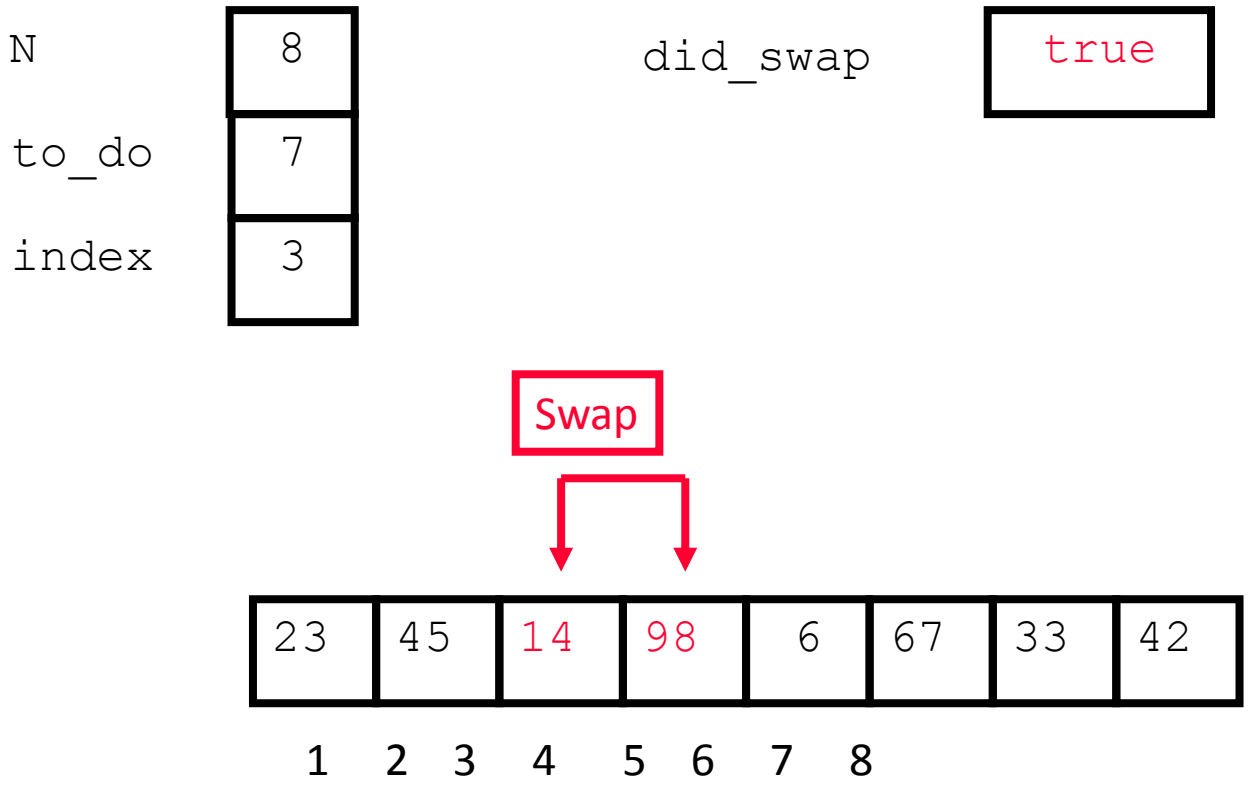
An Animated Example



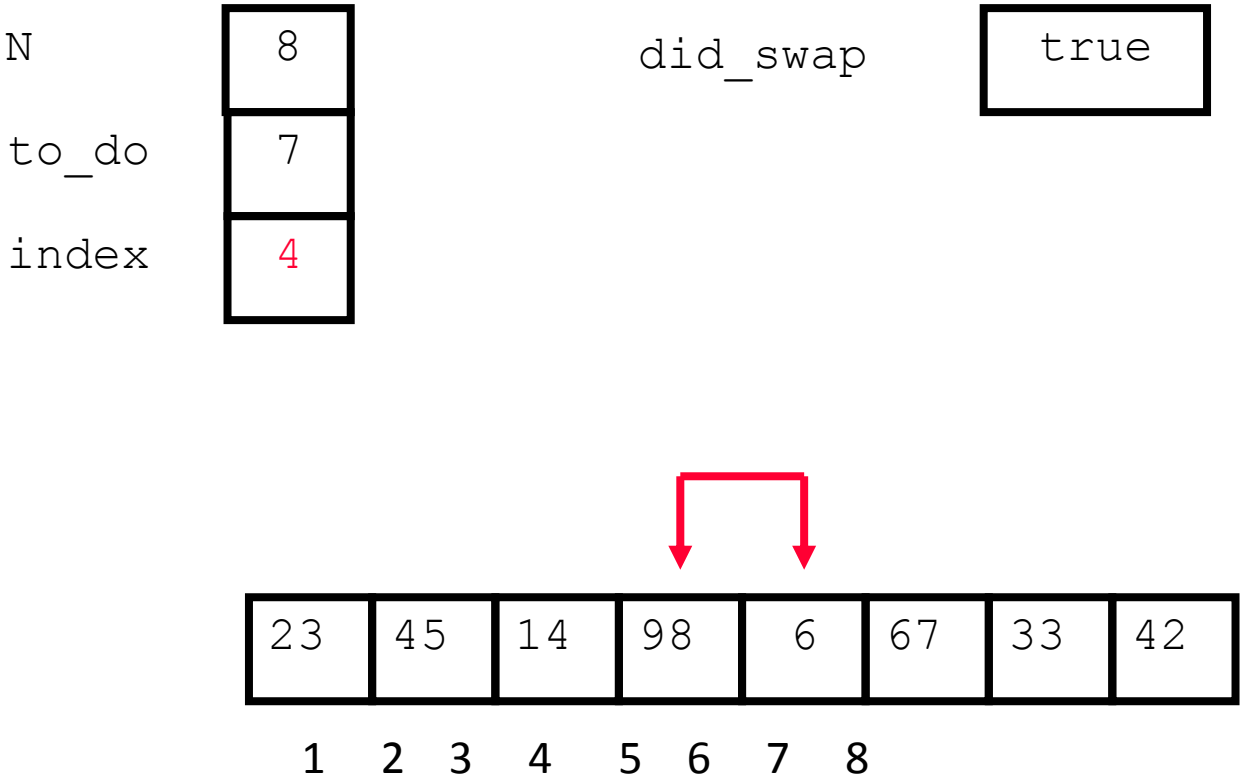
An Animated Example



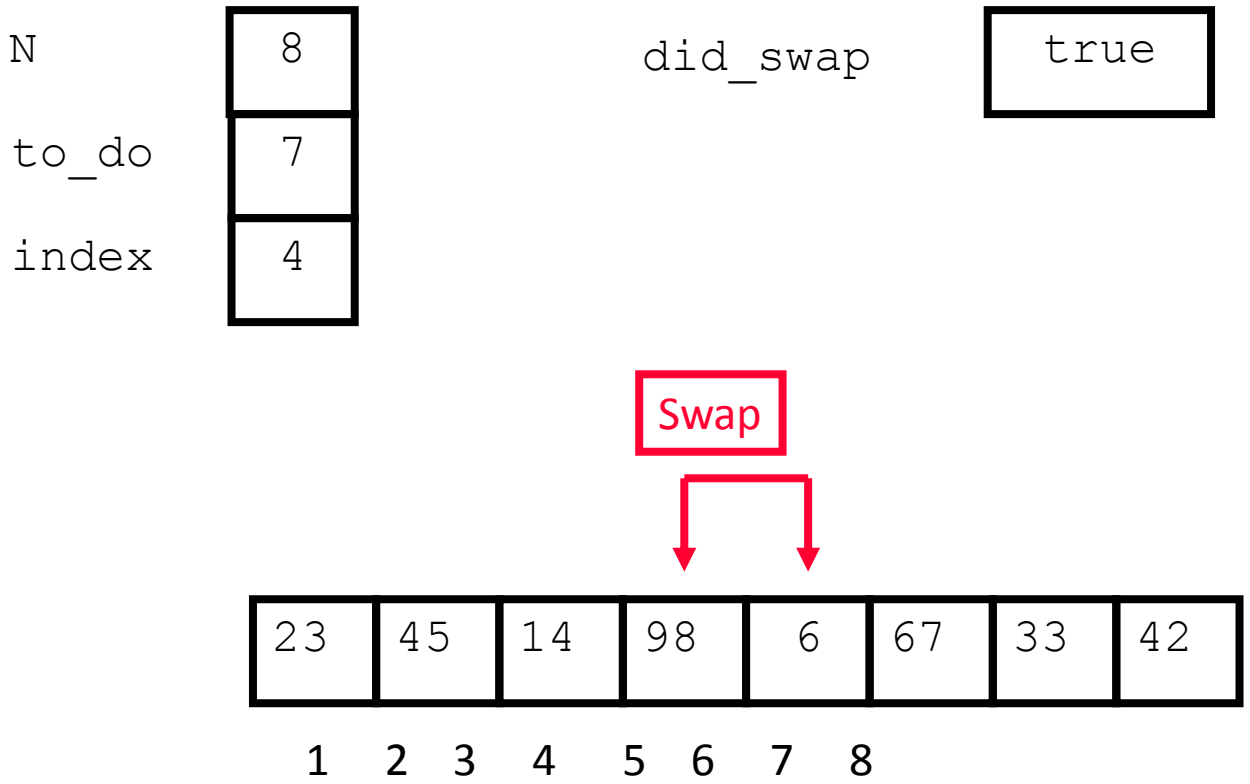
An Animated Example



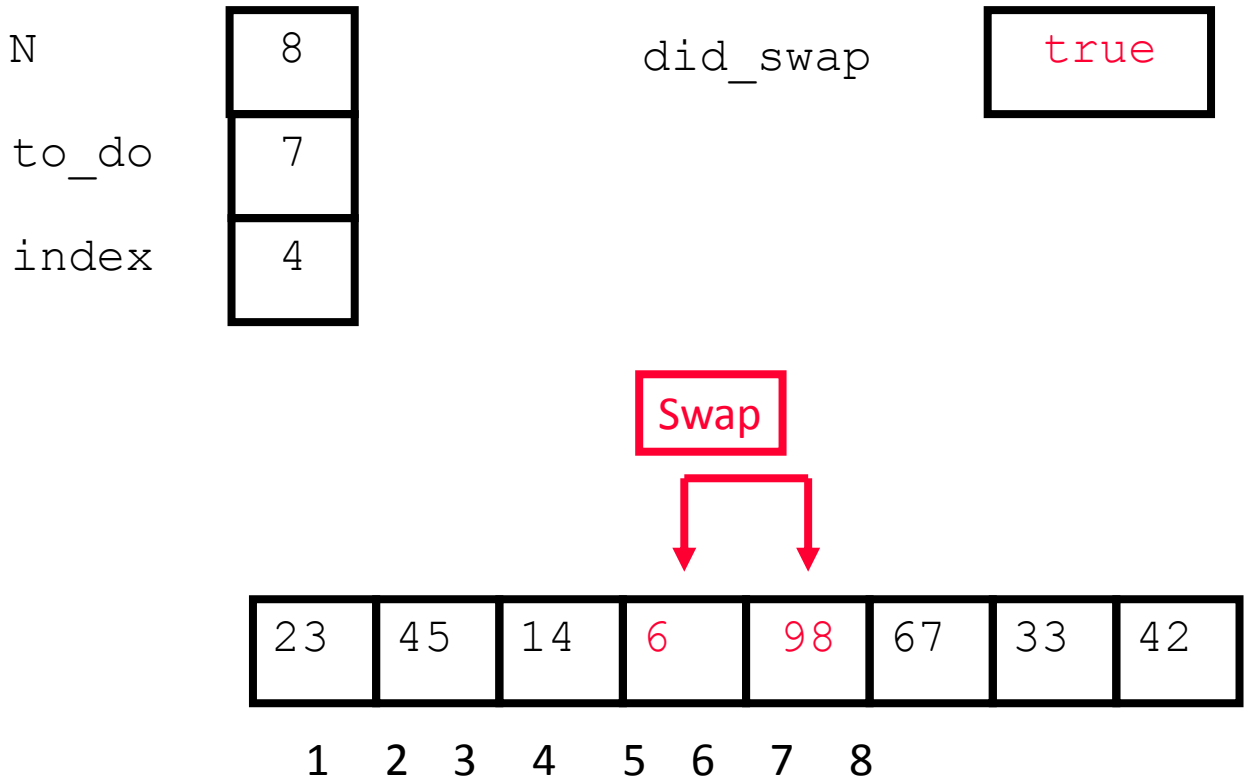
An Animated Example



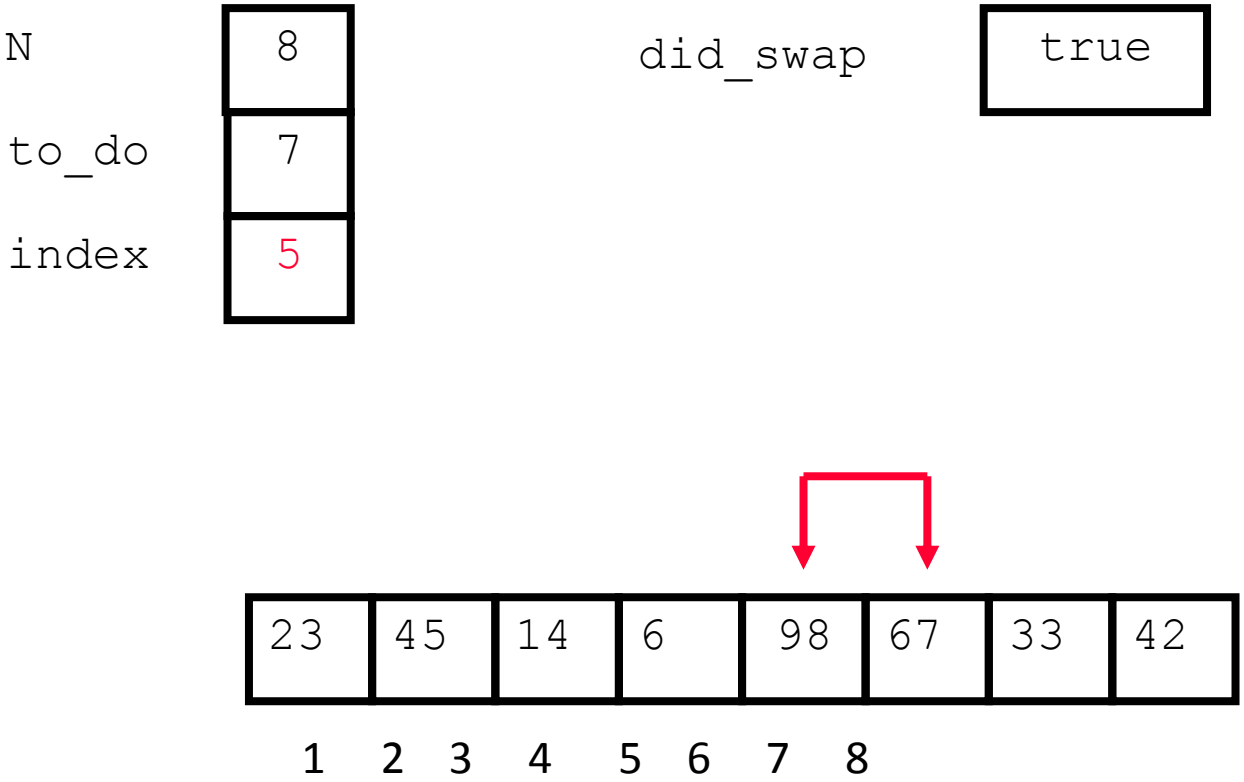
An Animated Example



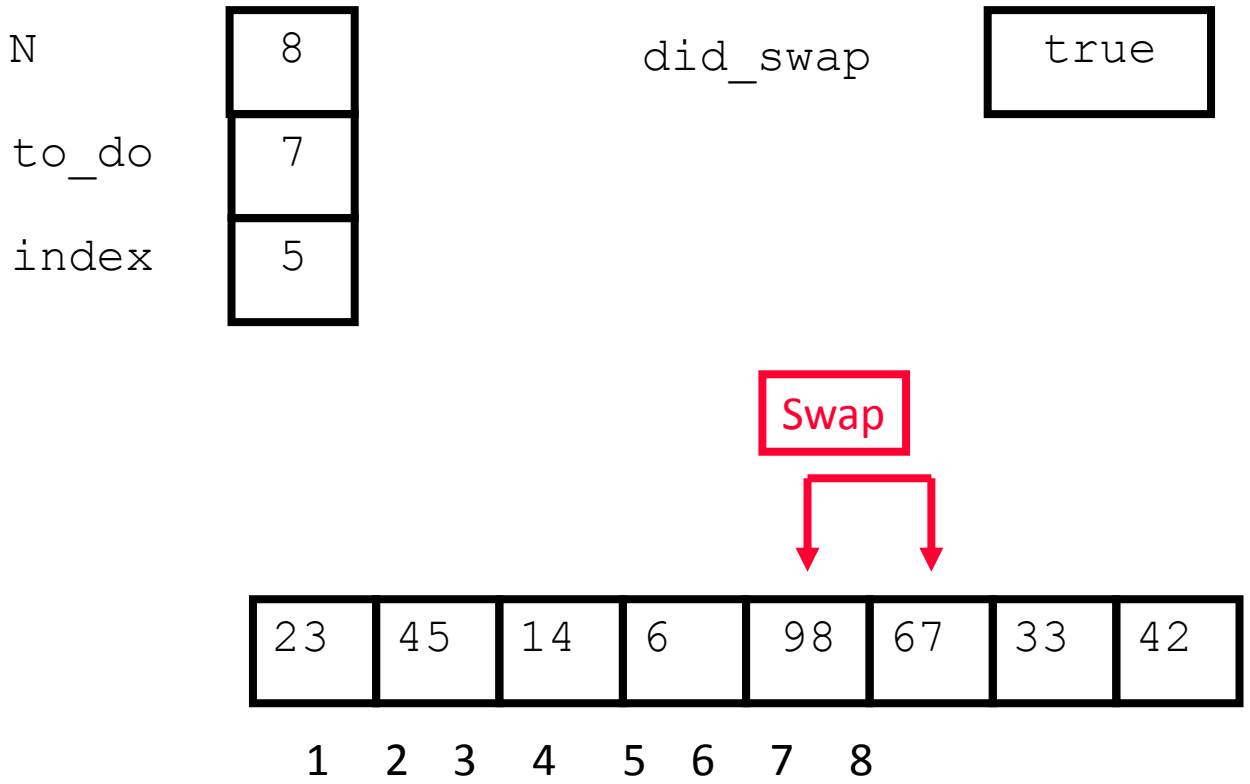
An Animated Example



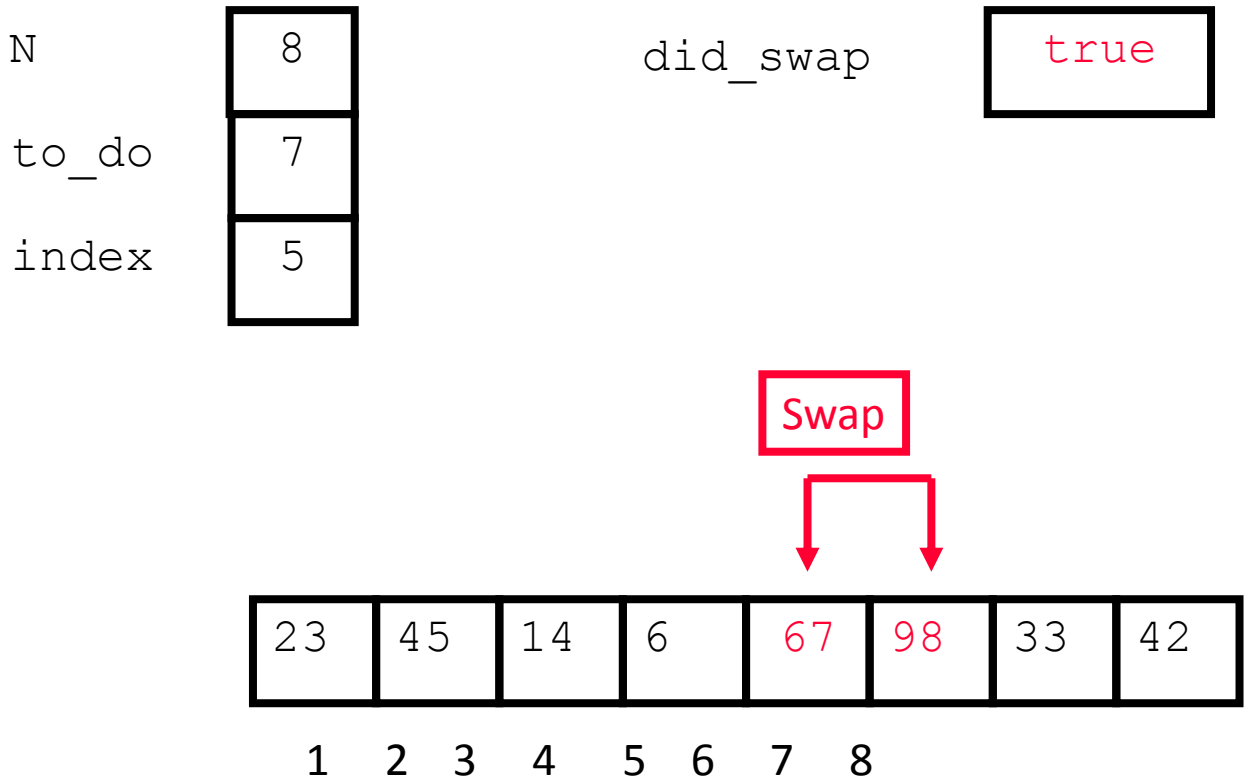
An Animated Example



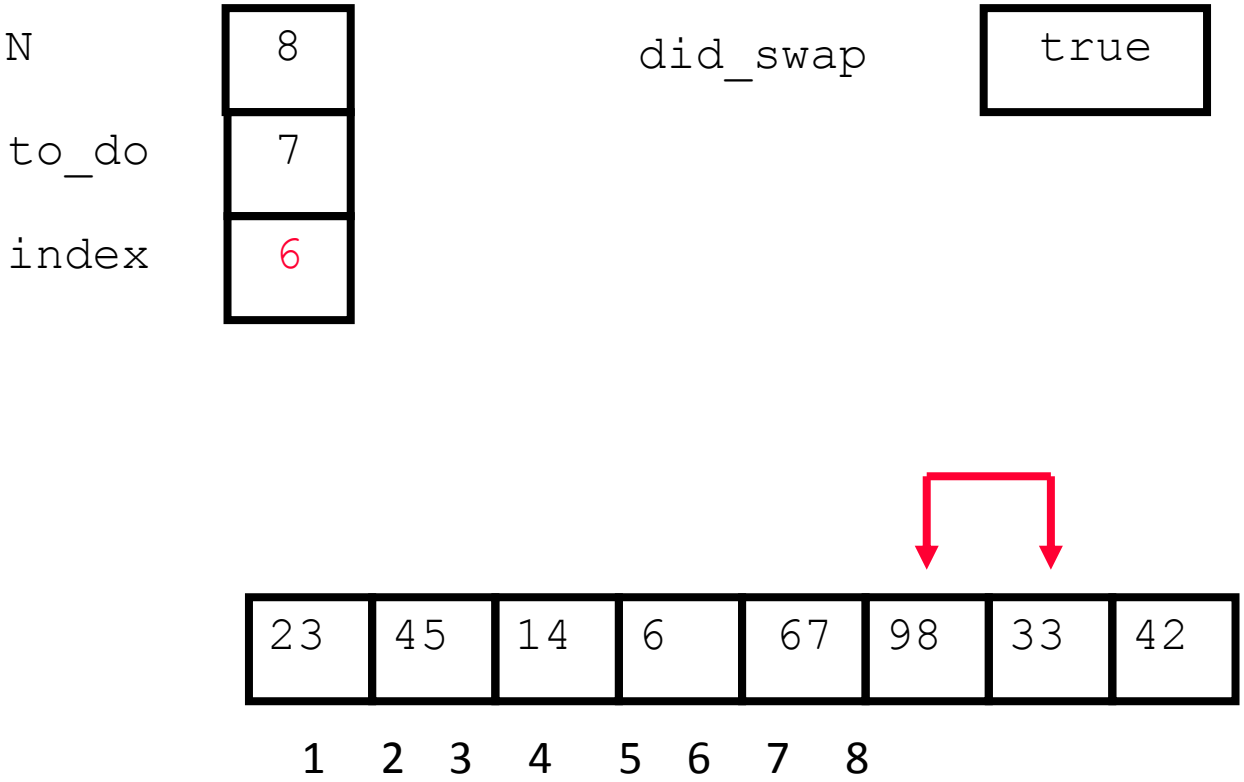
An Animated Example



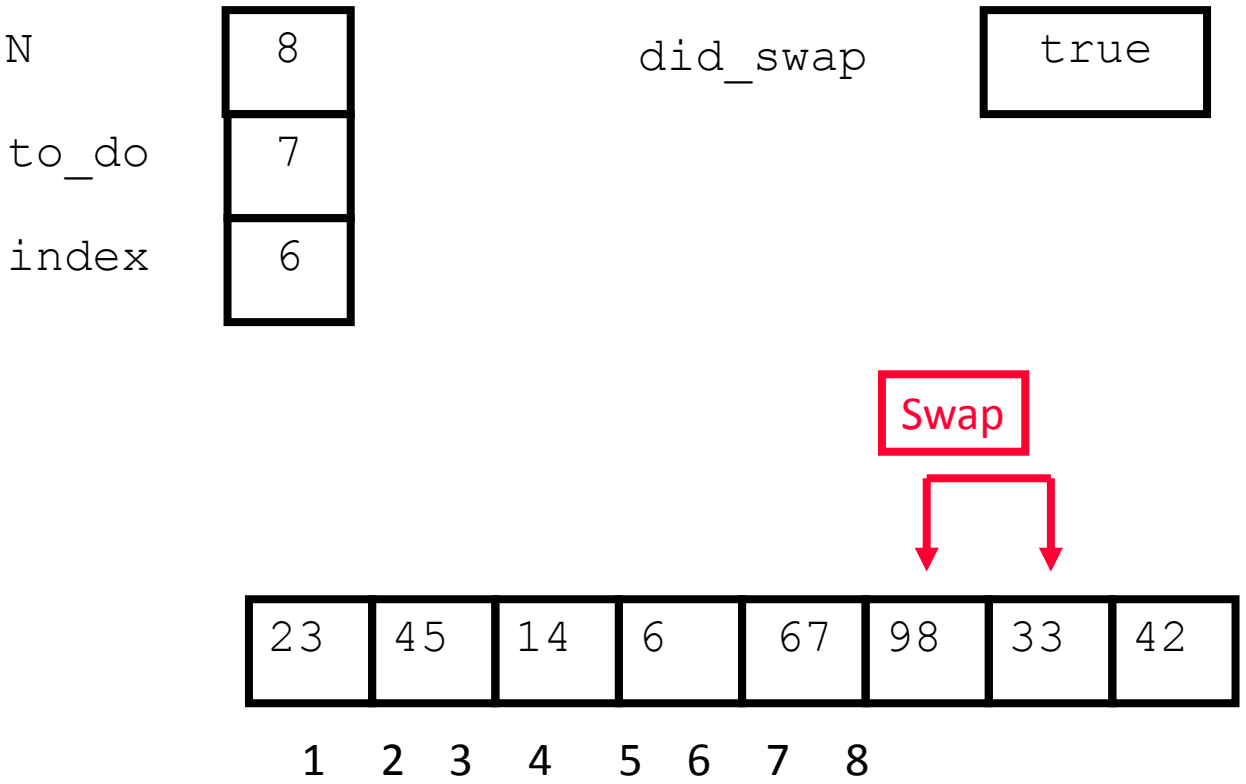
An Animated Example



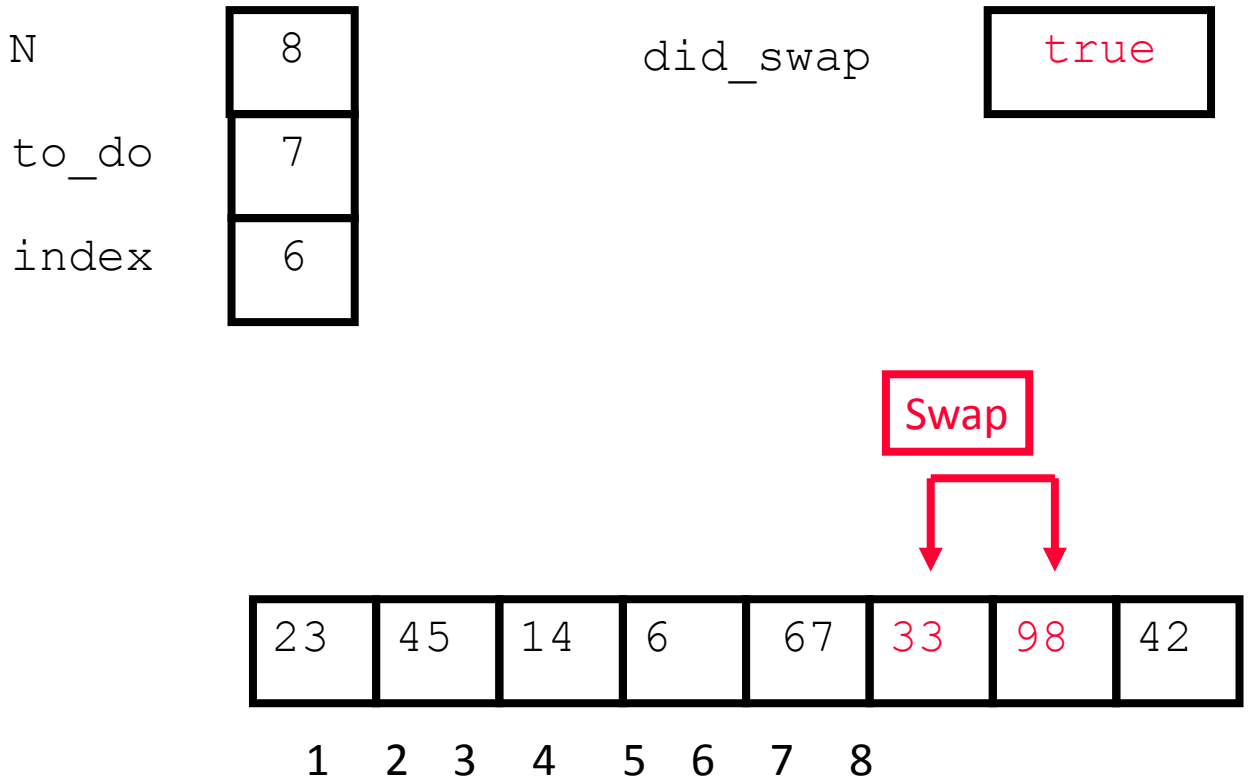
An Animated Example



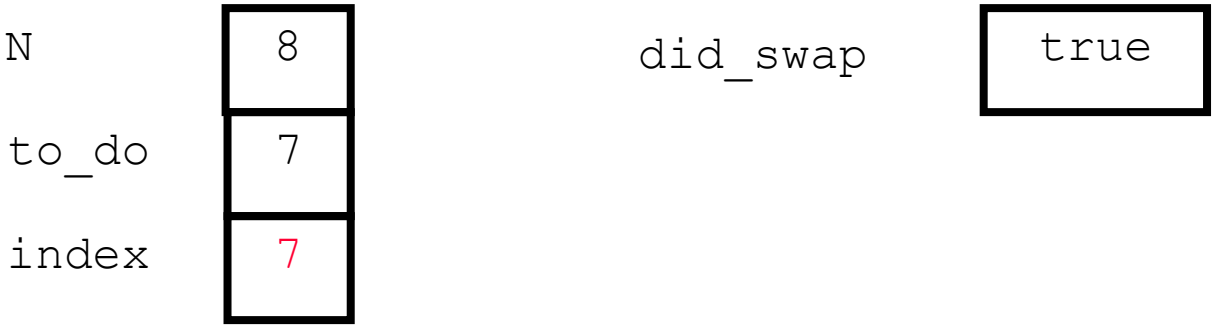
An Animated Example

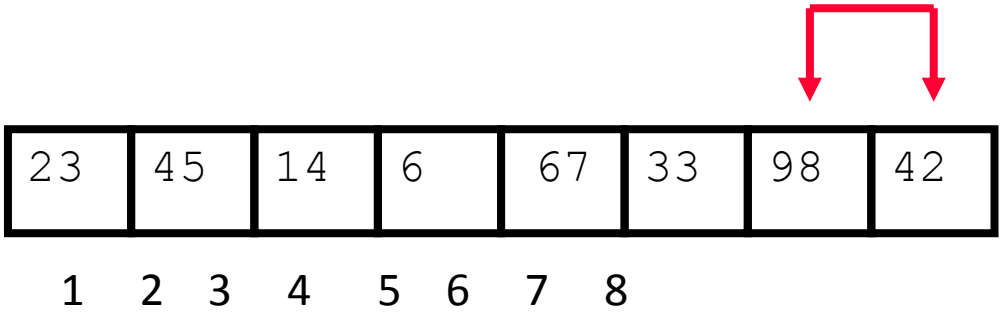


An Animated Example

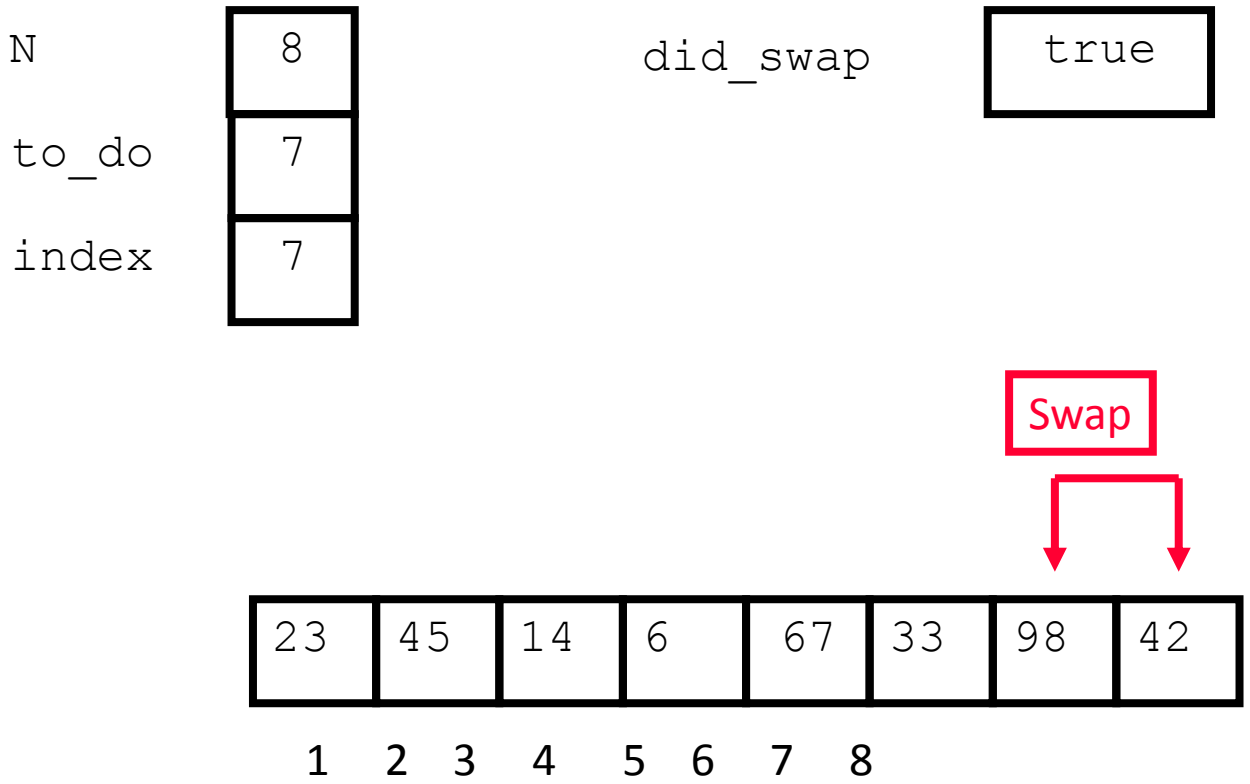


An Animated Example

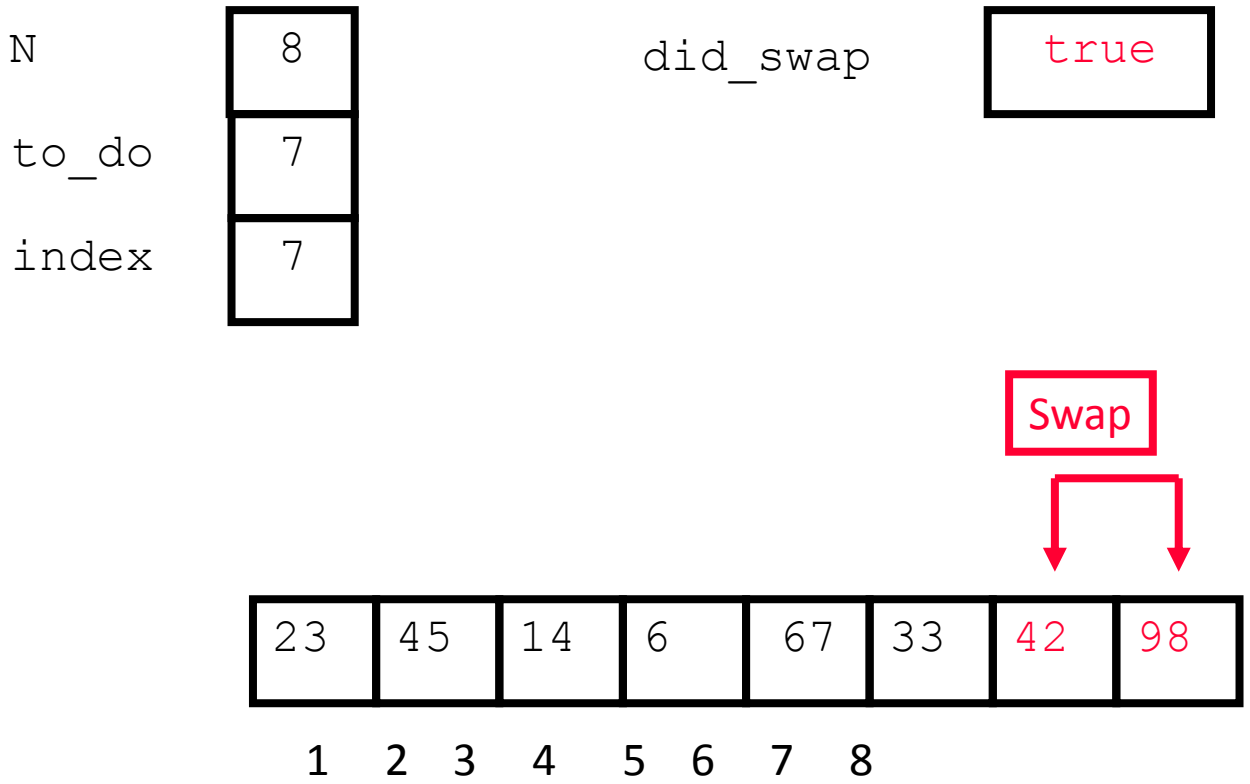




An Animated Example



An Animated Example

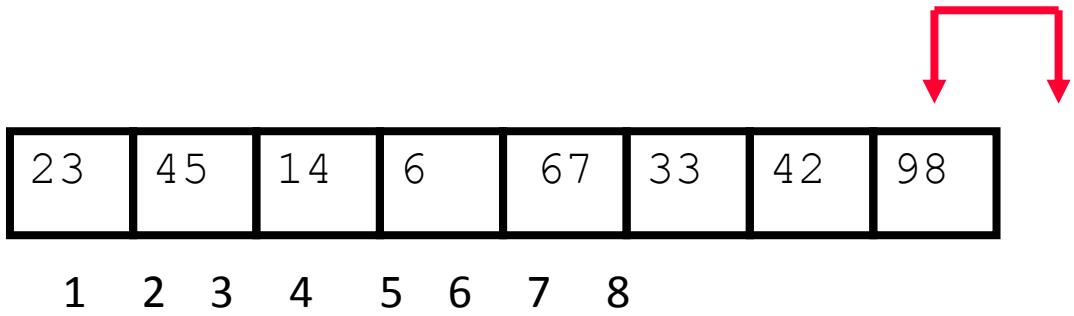


After First Pass of Outer Loop

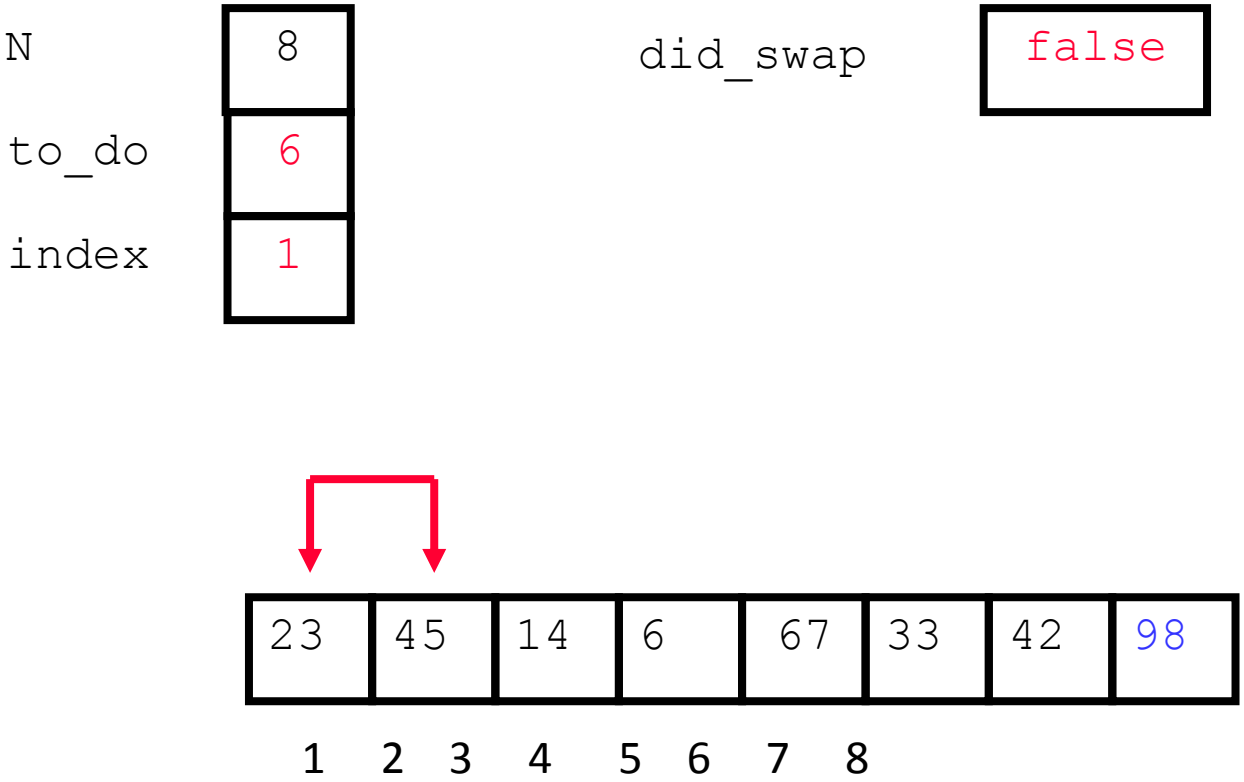
N	8
to_do	7
index	8

did_swap	true

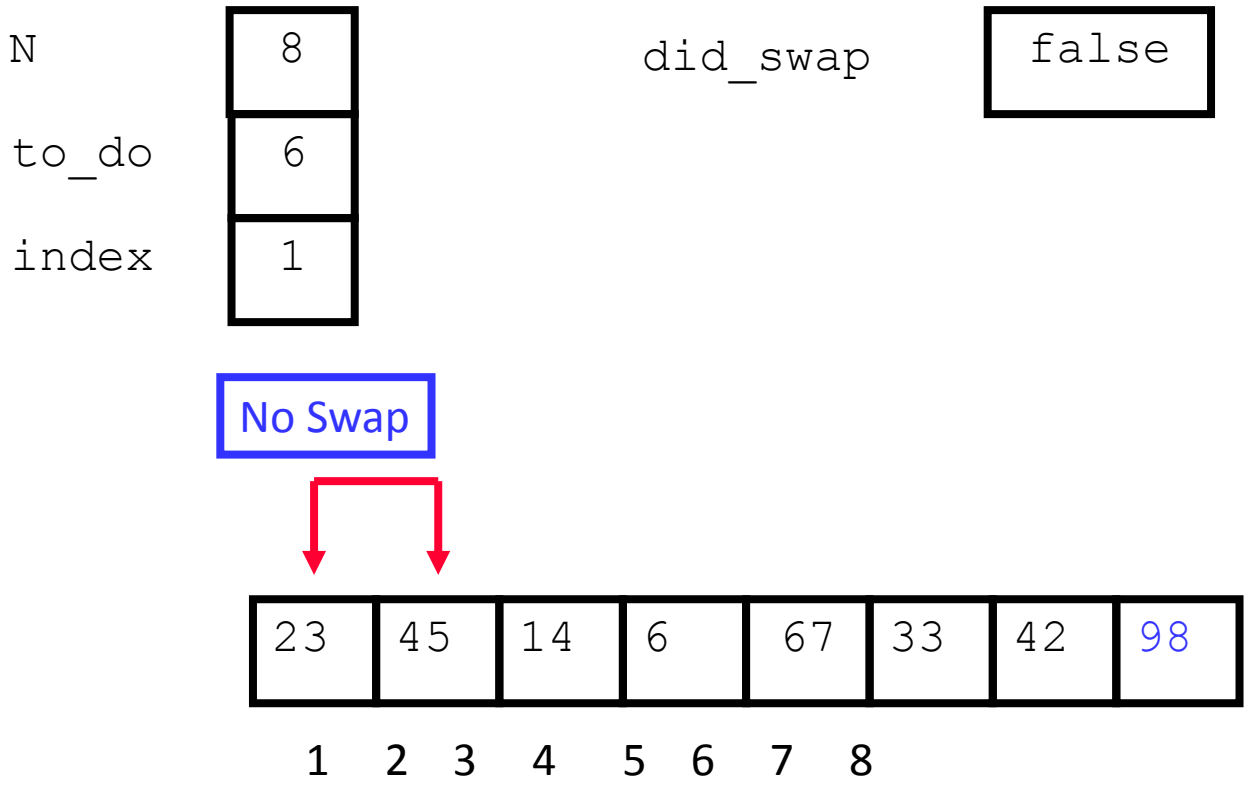
Finished first “Bubble Up”



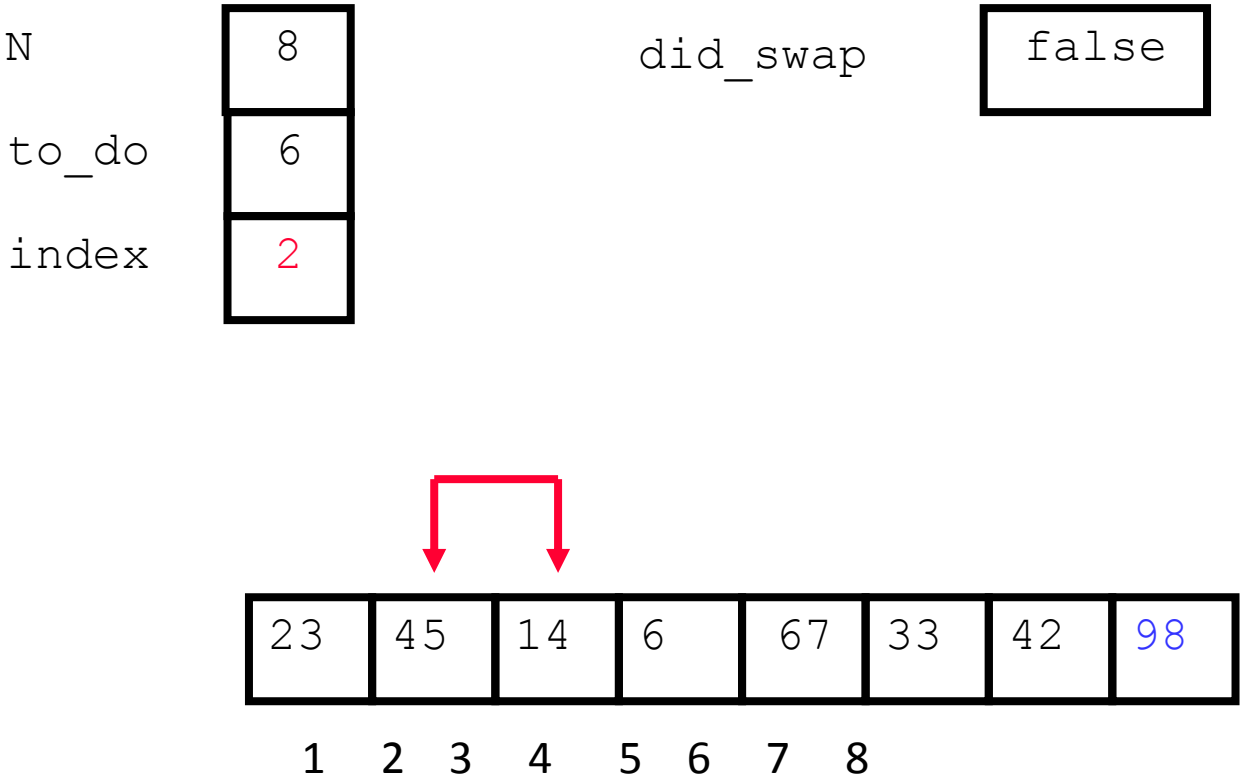
The Second “Bubble Up”



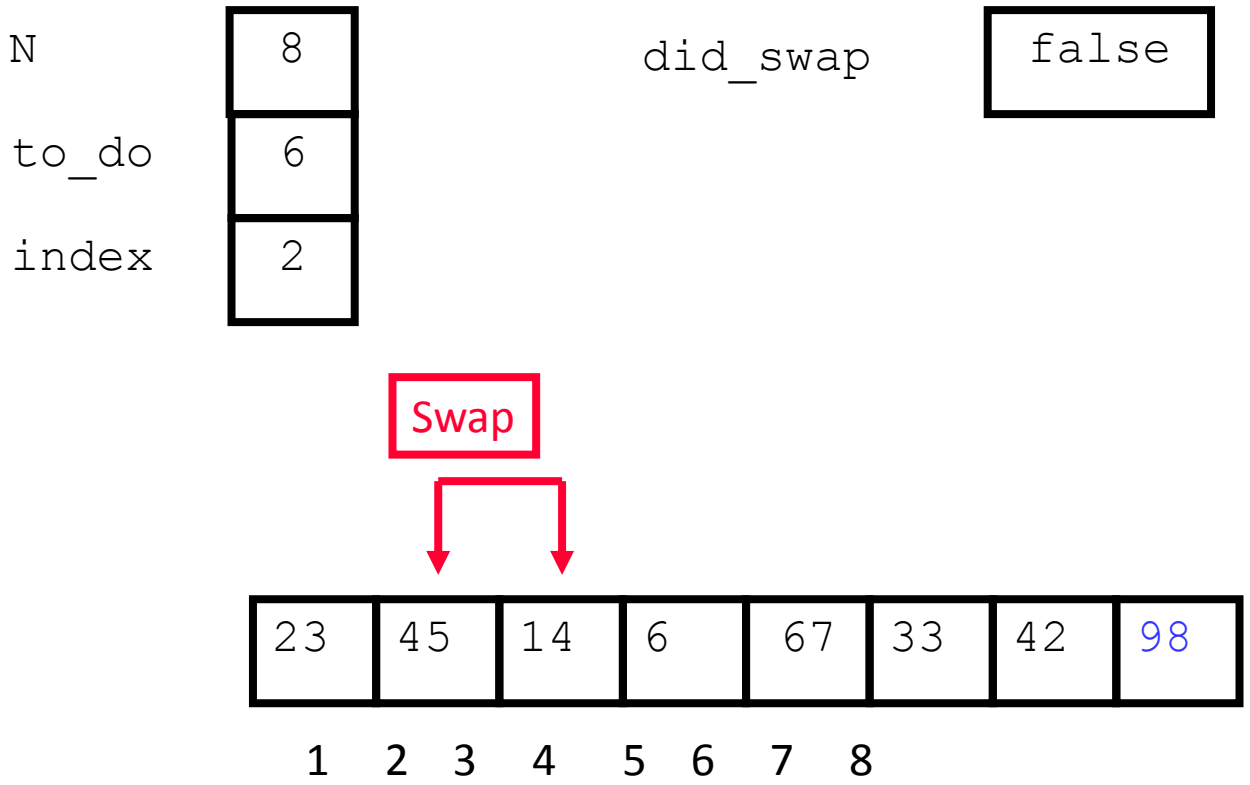
The Second “Bubble Up”



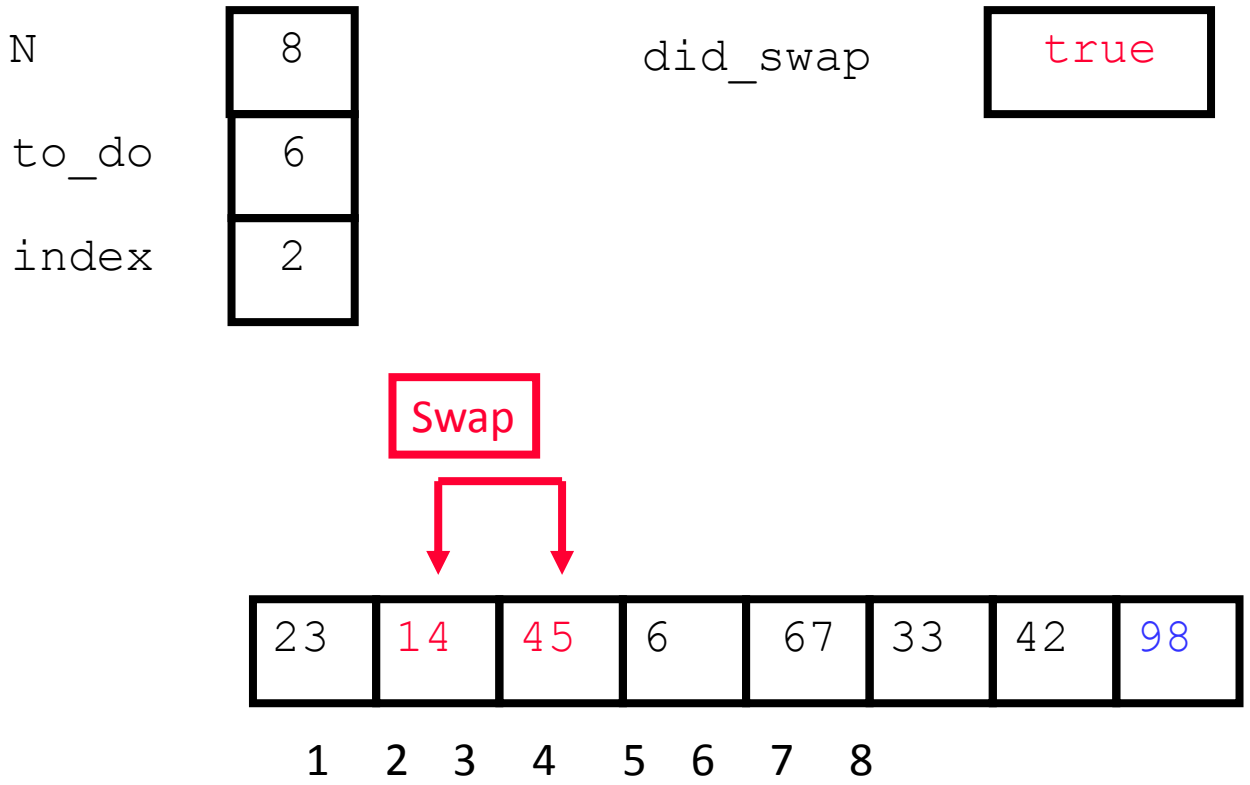
The Second “Bubble Up”



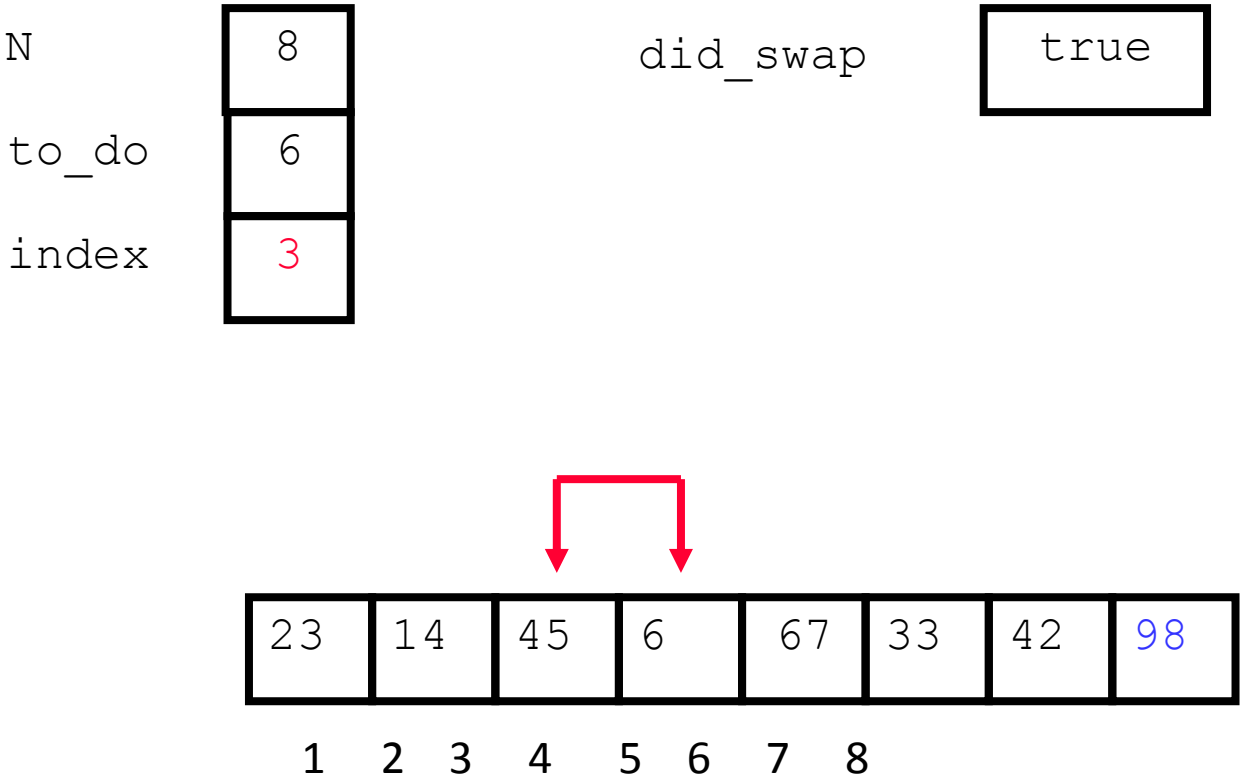
The Second “Bubble Up”



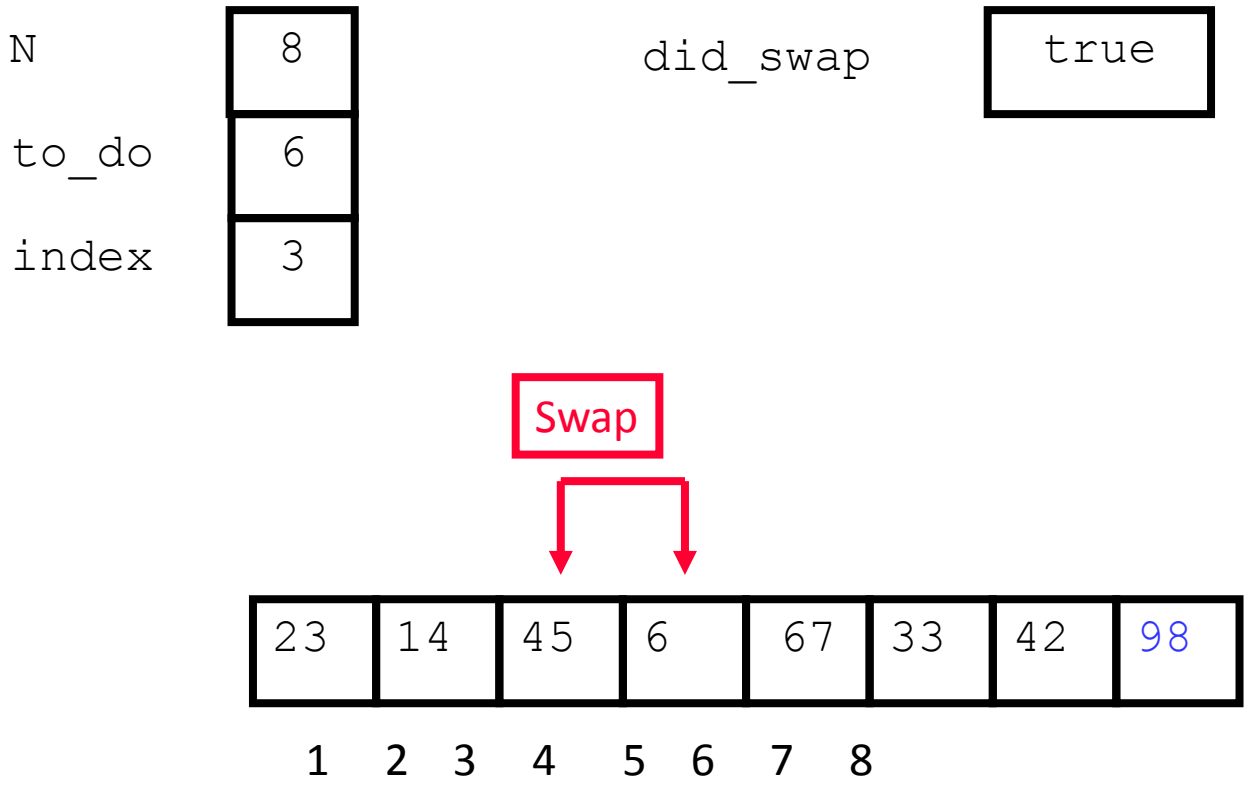
The Second “Bubble Up”



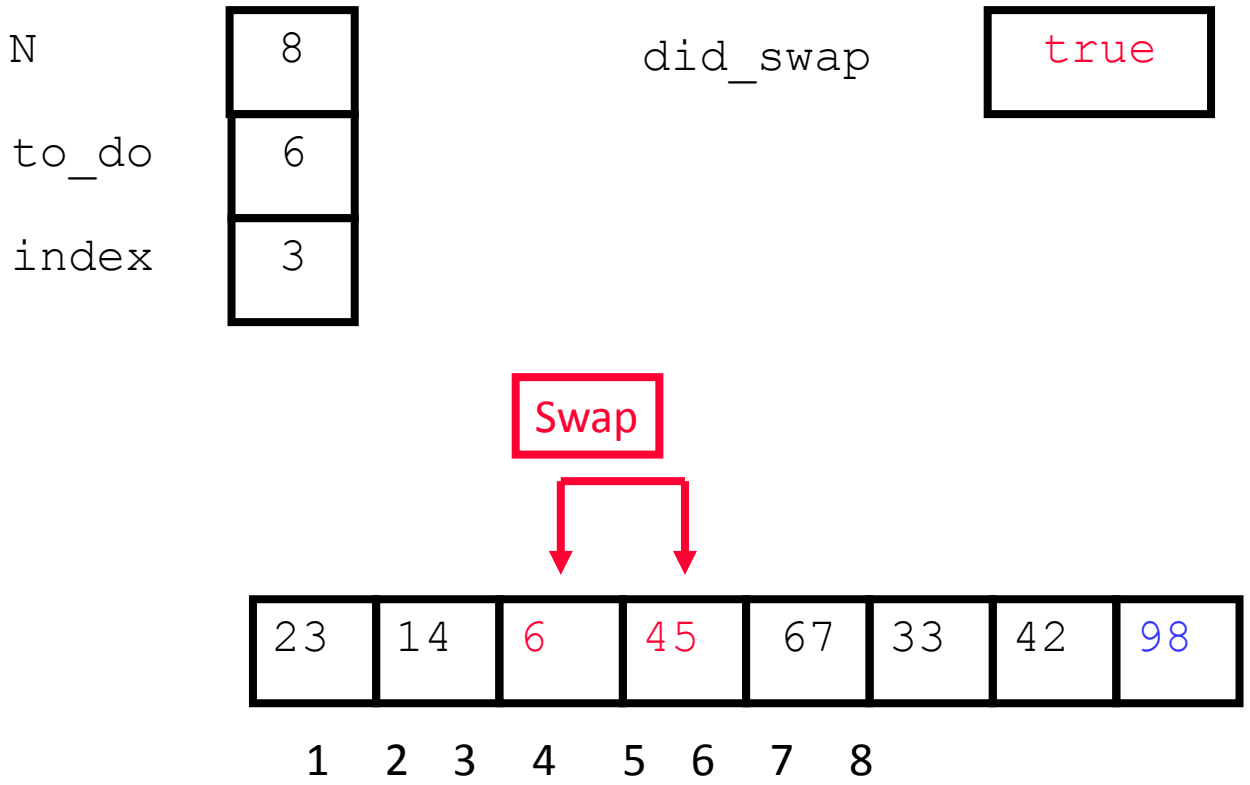
The Second “Bubble Up”



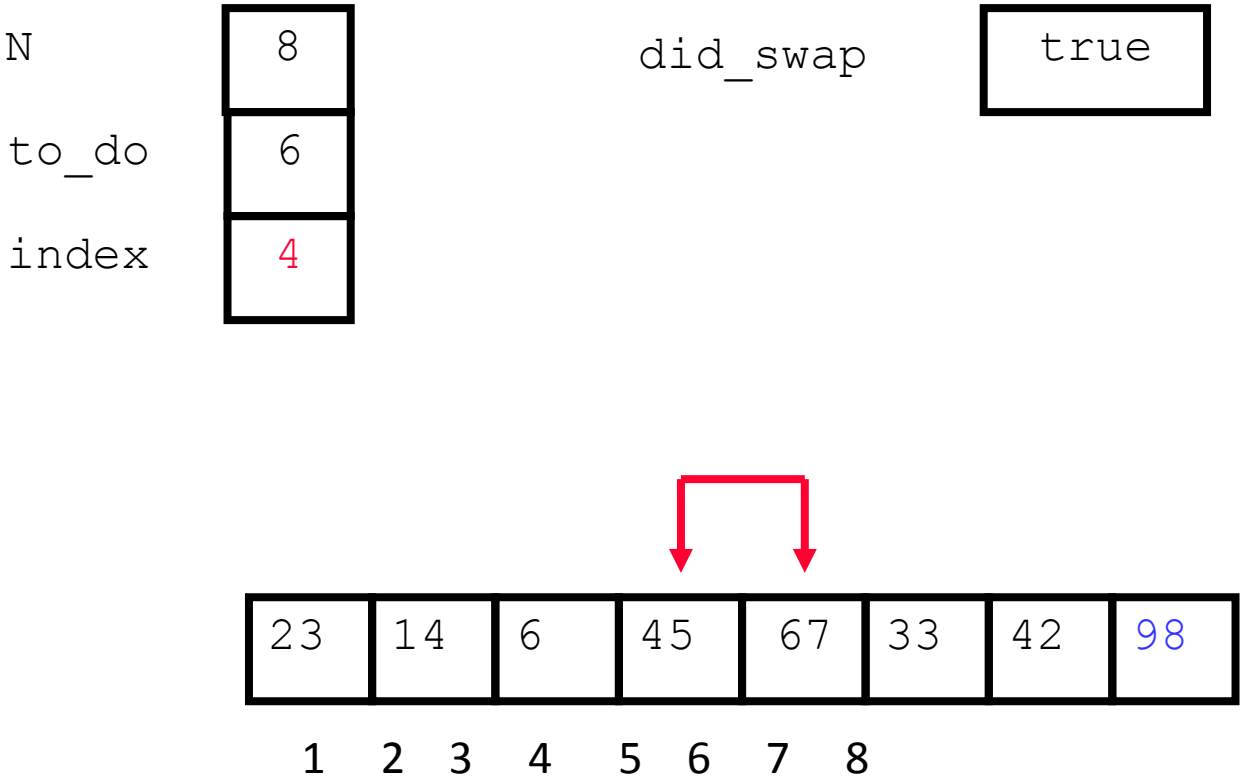
The Second “Bubble Up”



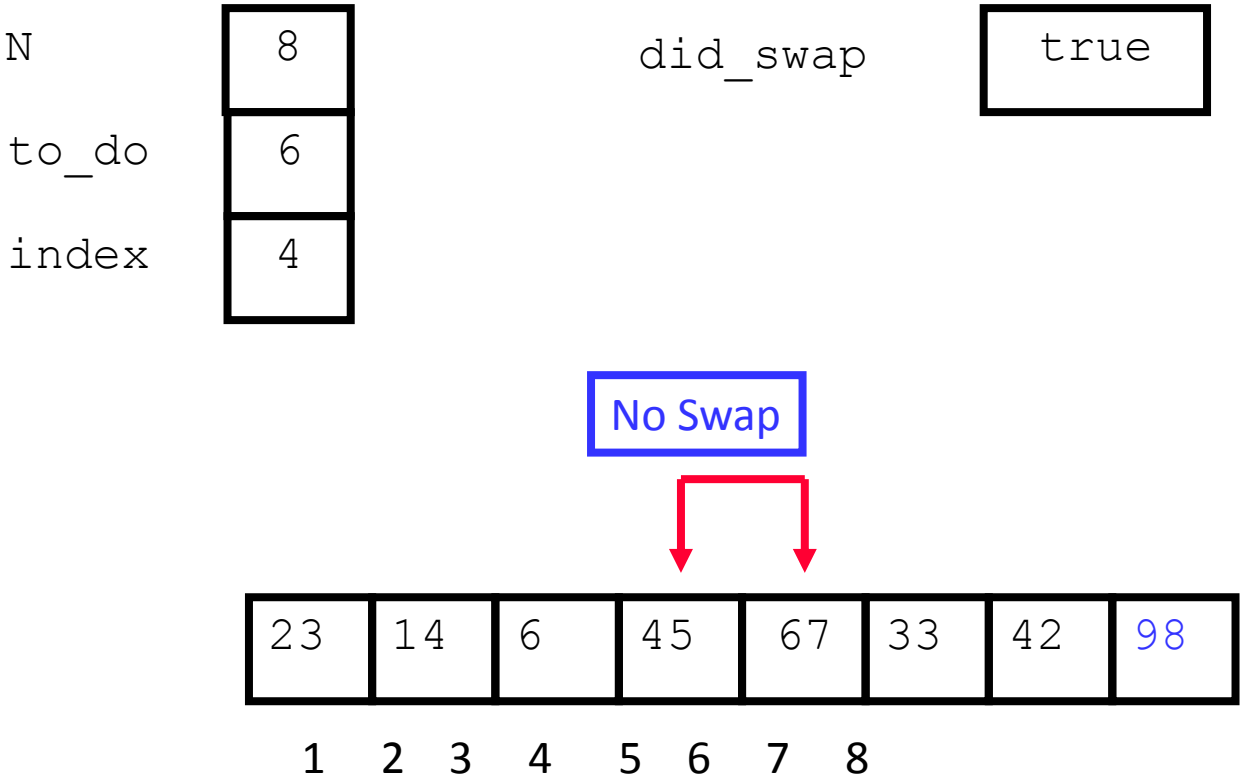
The Second “Bubble Up”



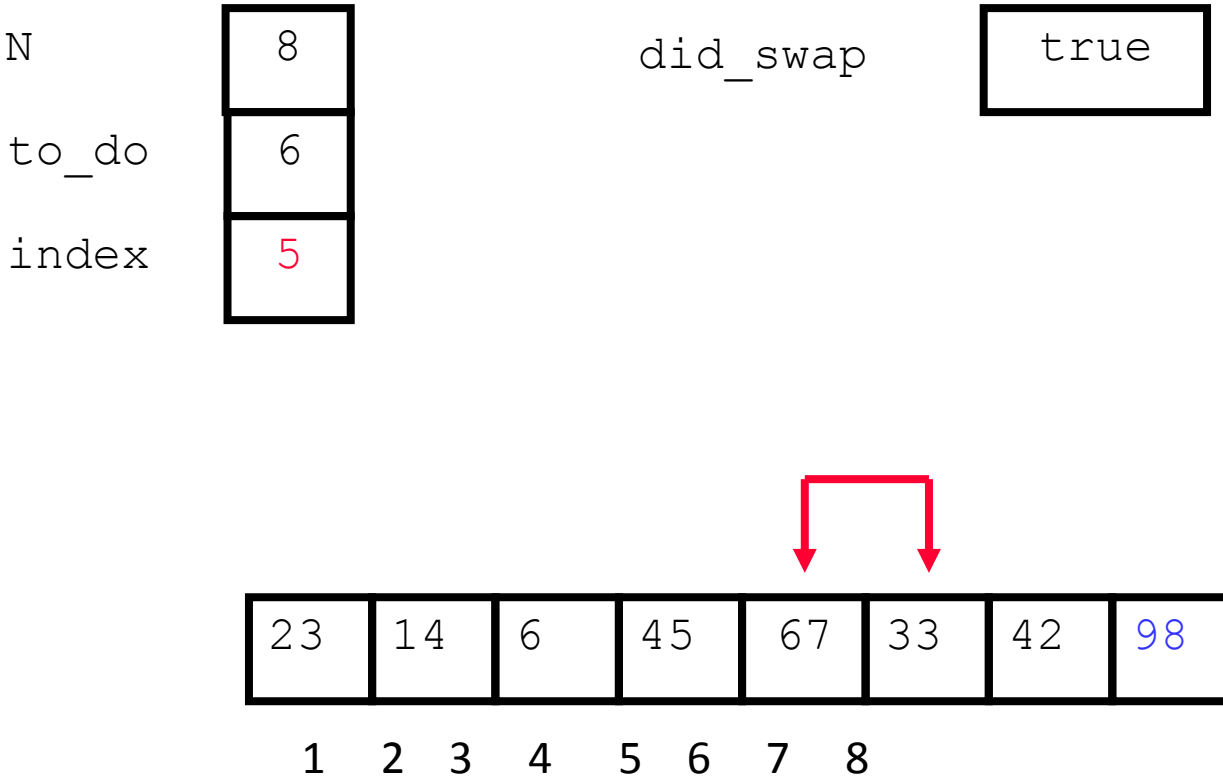
The Second “Bubble Up”



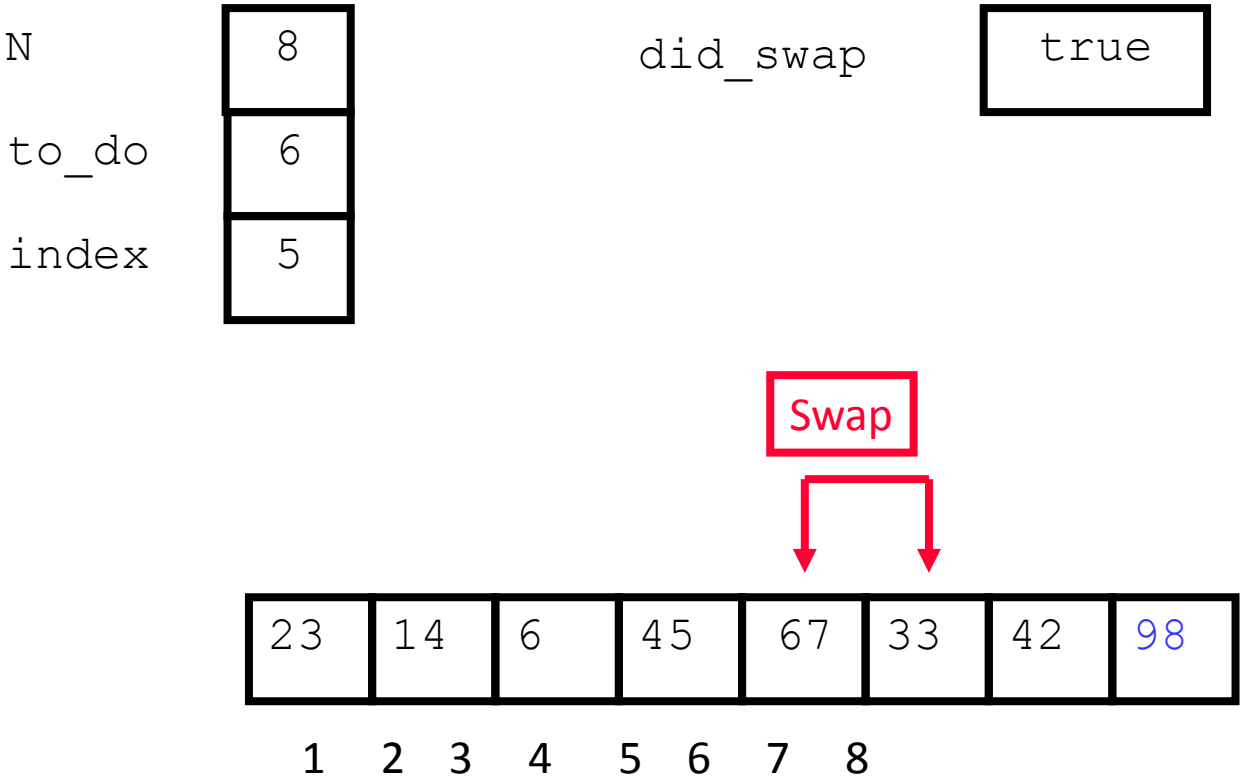
The Second “Bubble Up”



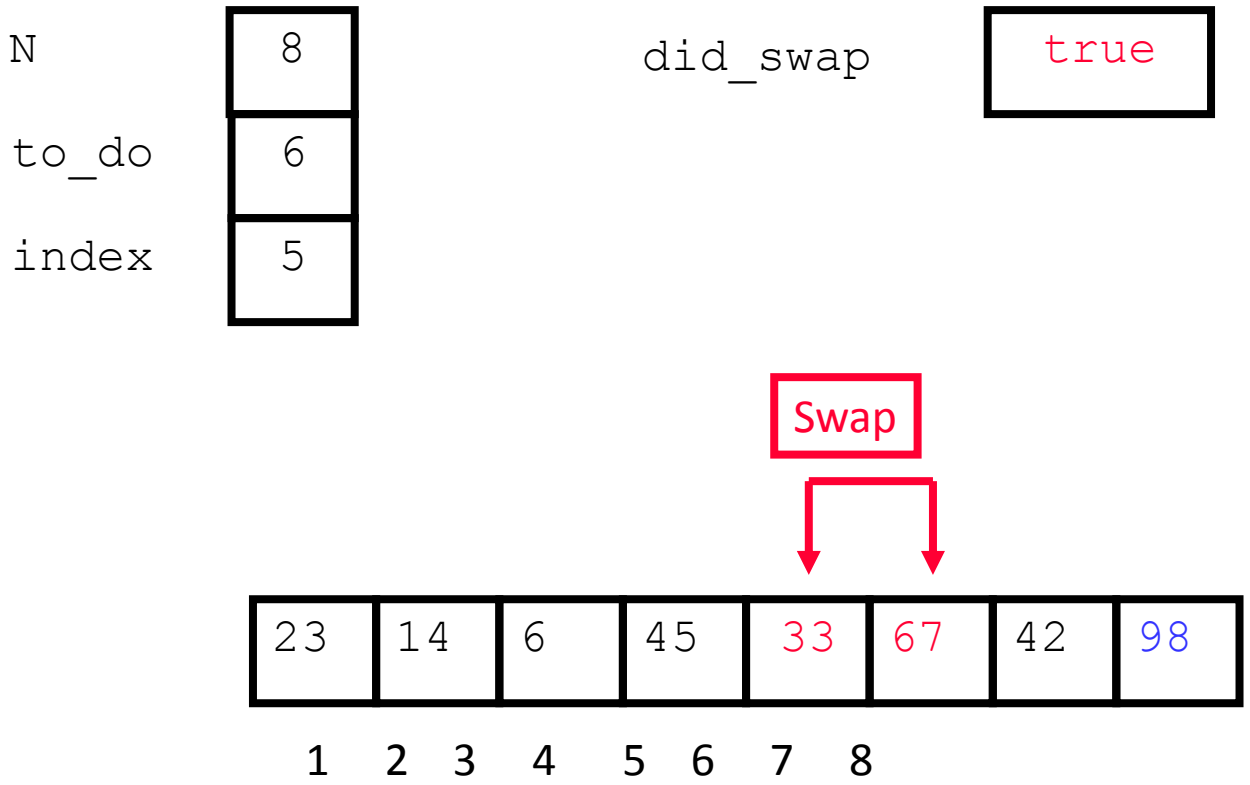
The Second “Bubble Up”



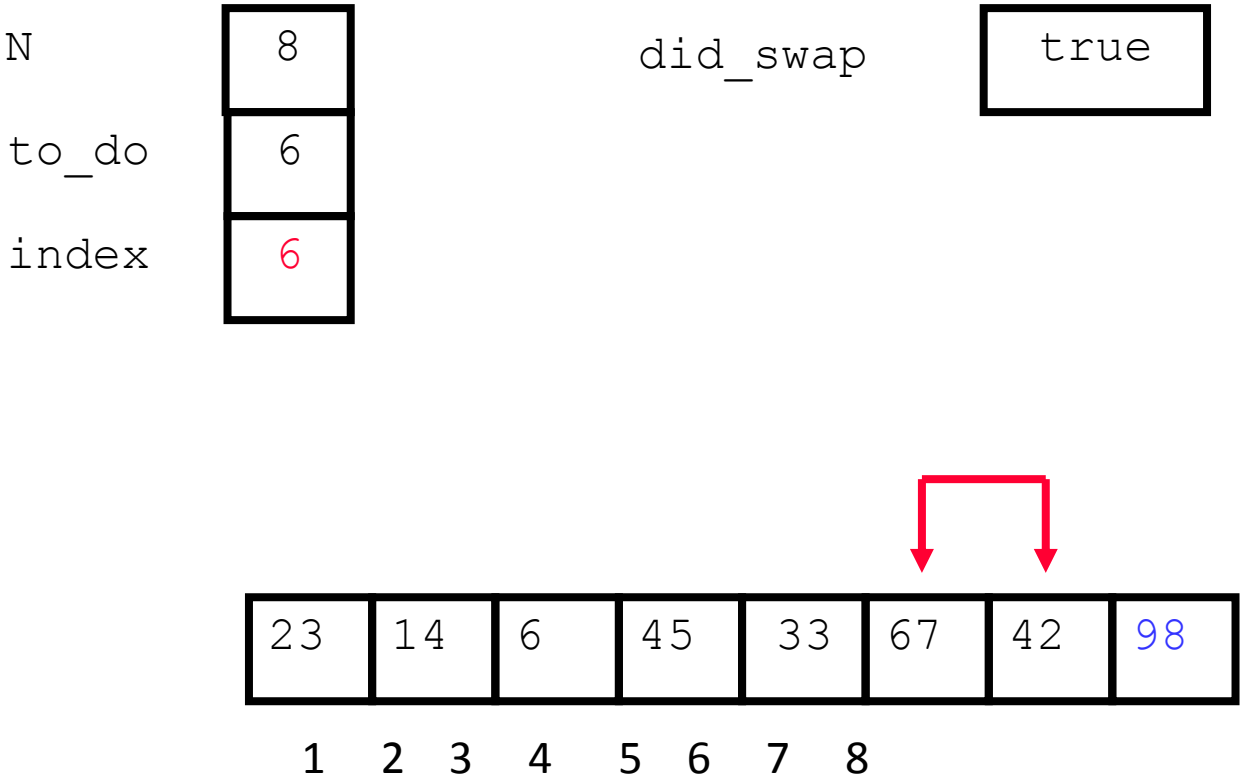
The Second “Bubble Up”



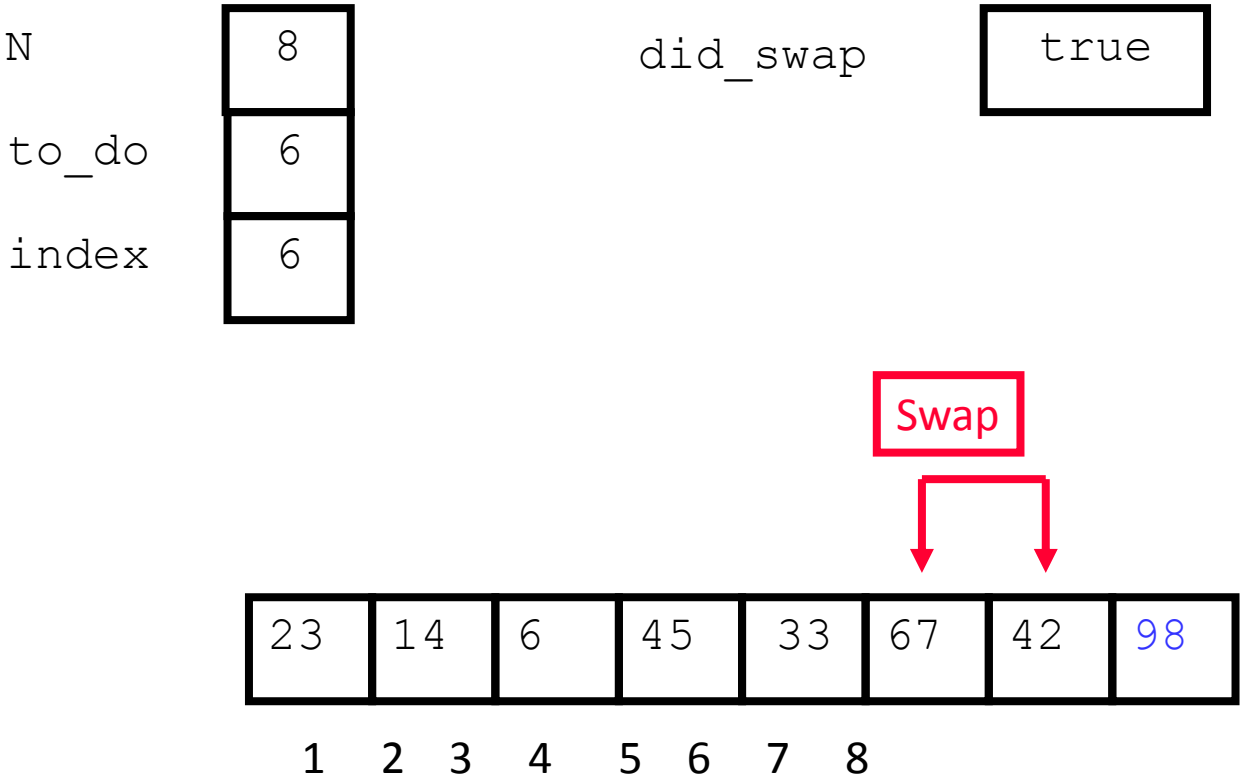
The Second “Bubble Up”



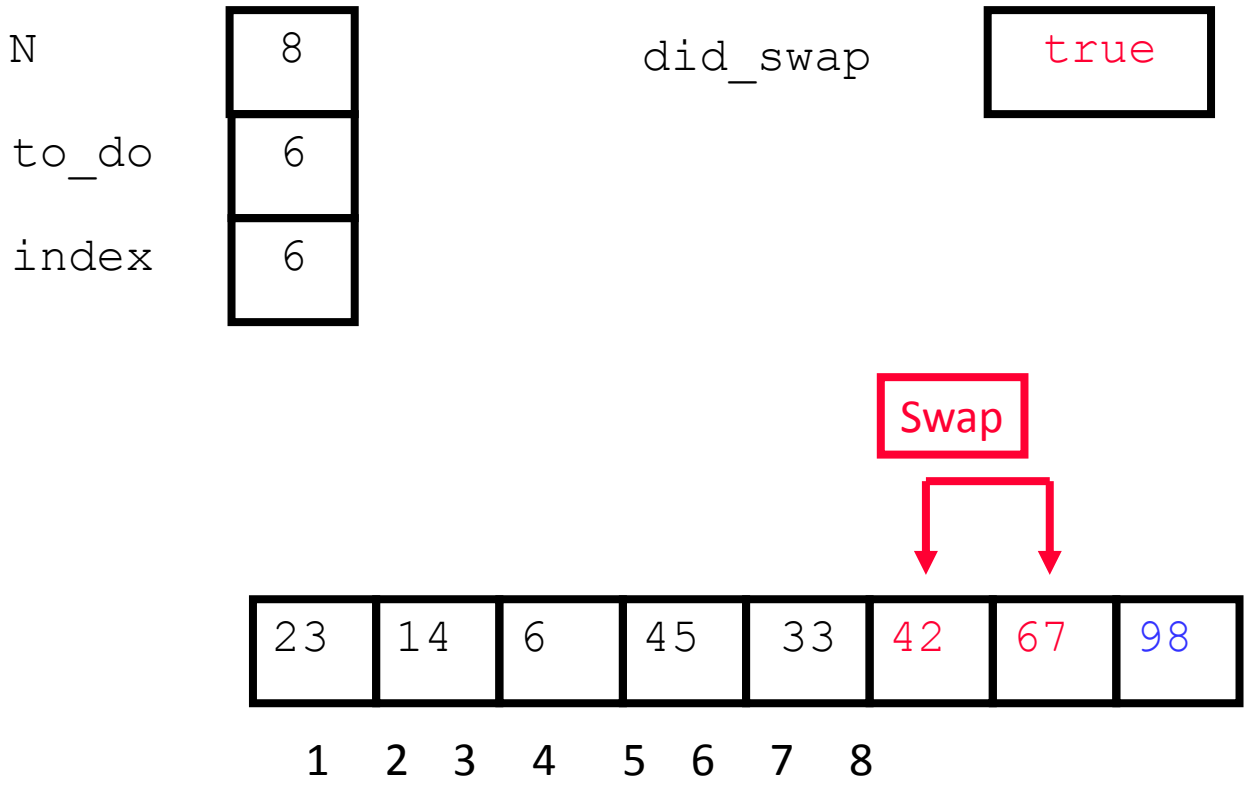
The Second “Bubble Up”



The Second “Bubble Up”



The Second “Bubble Up”

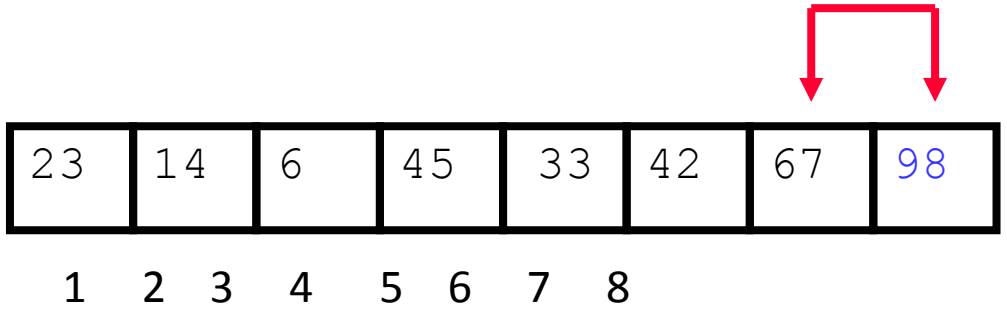


After Second Pass of Outer Loop

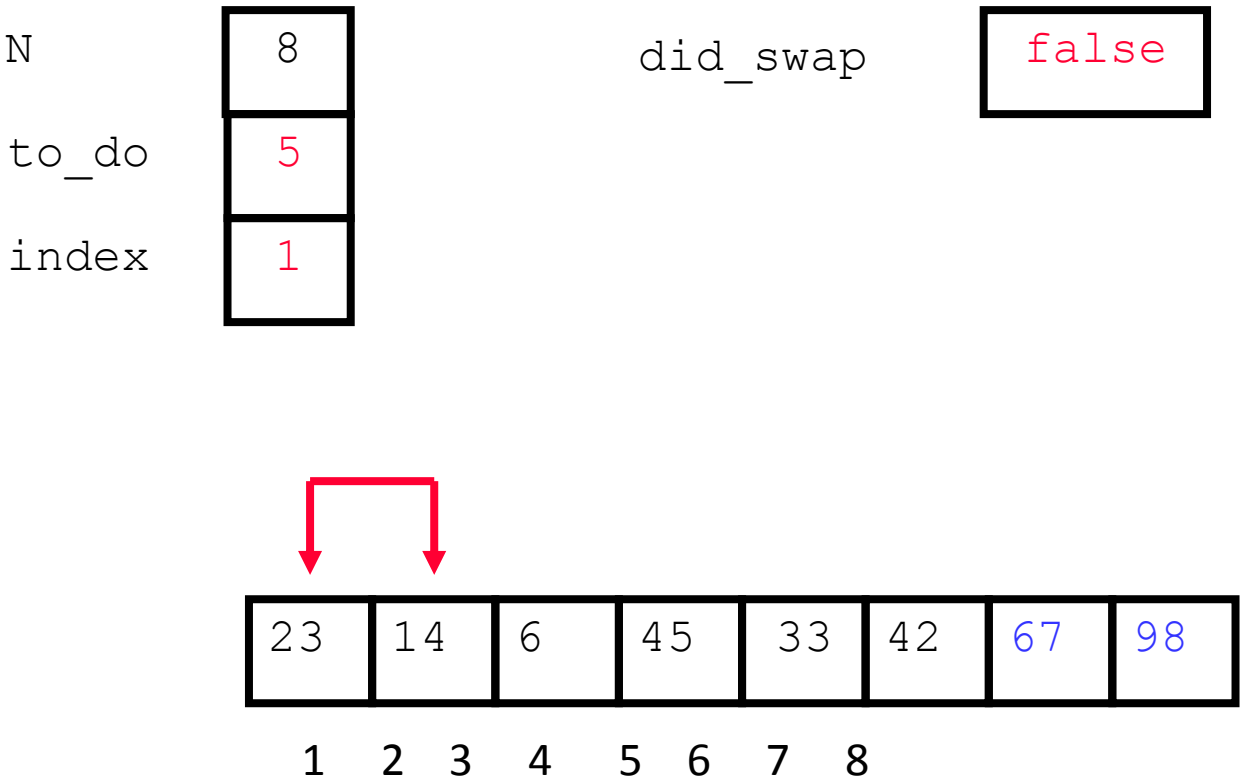
N	8
to_do	6
index	7

did_swap	true

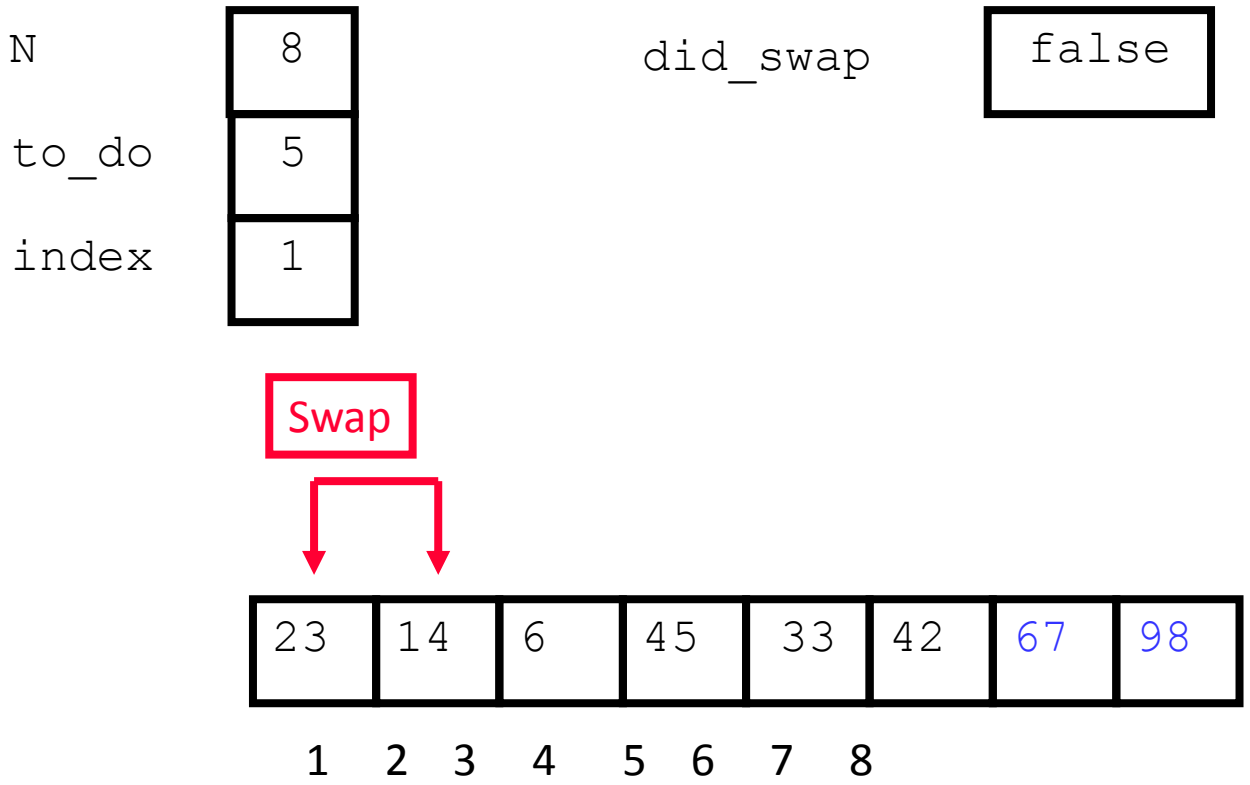
Finished second “Bubble Up”



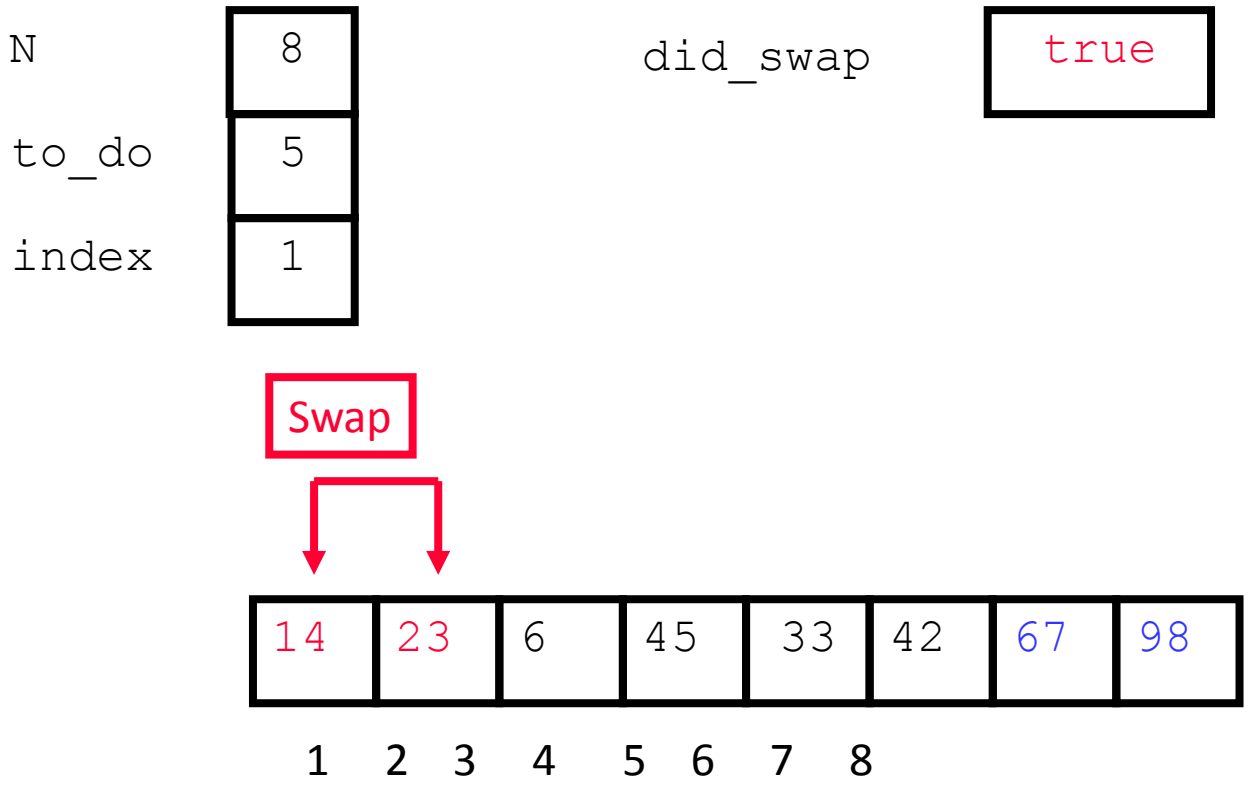
The Third “Bubble Up”



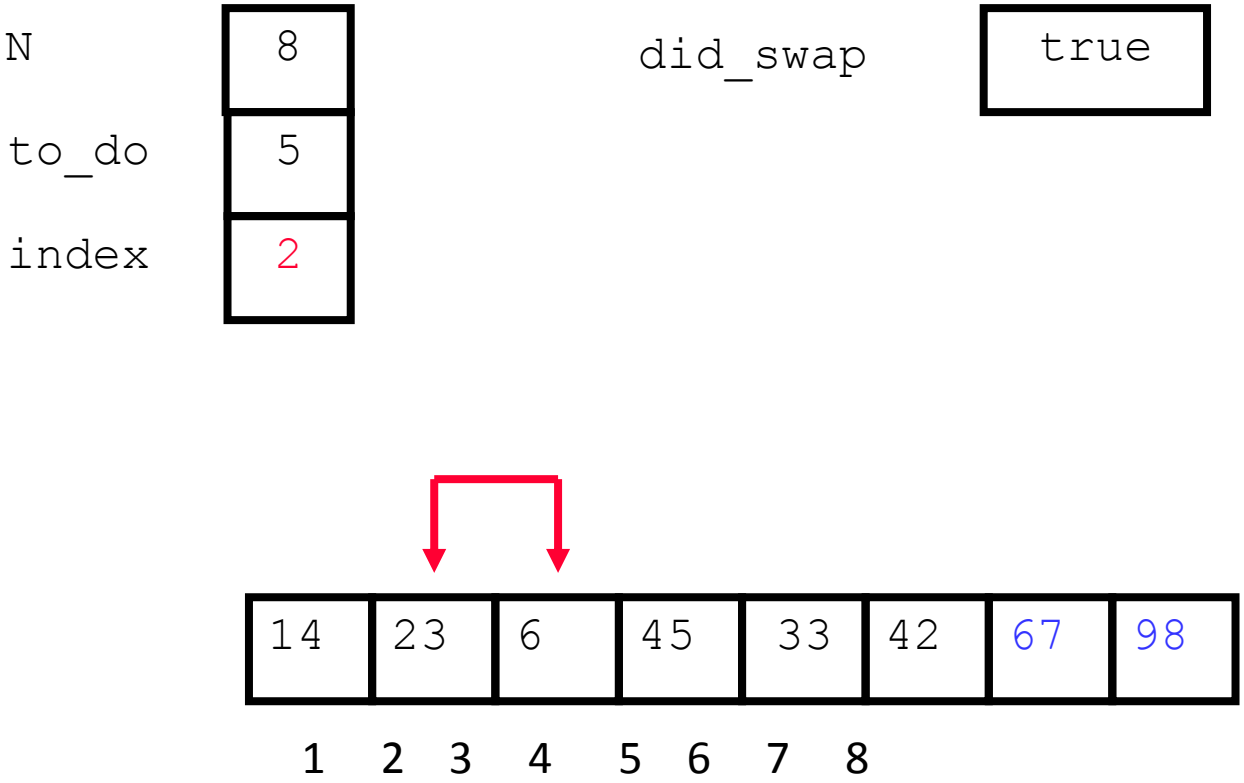
The Third “Bubble Up”



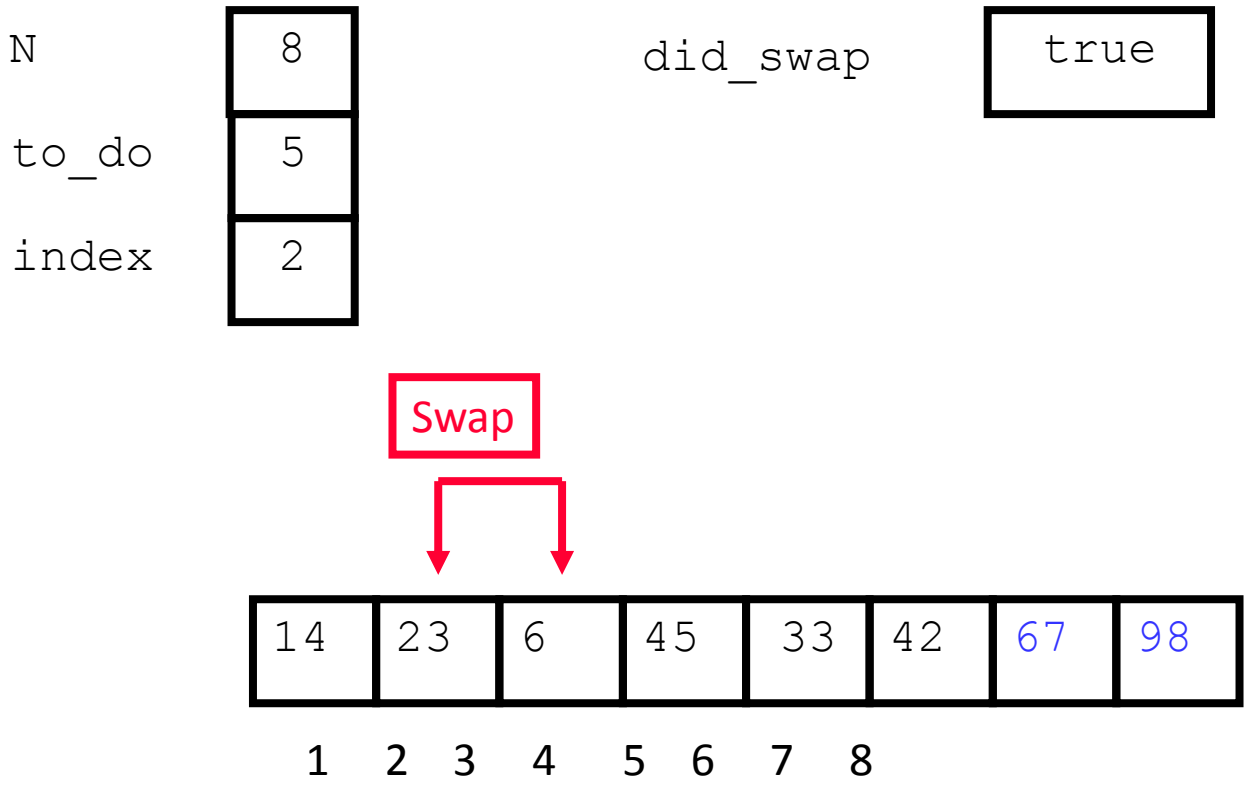
The Third “Bubble Up”



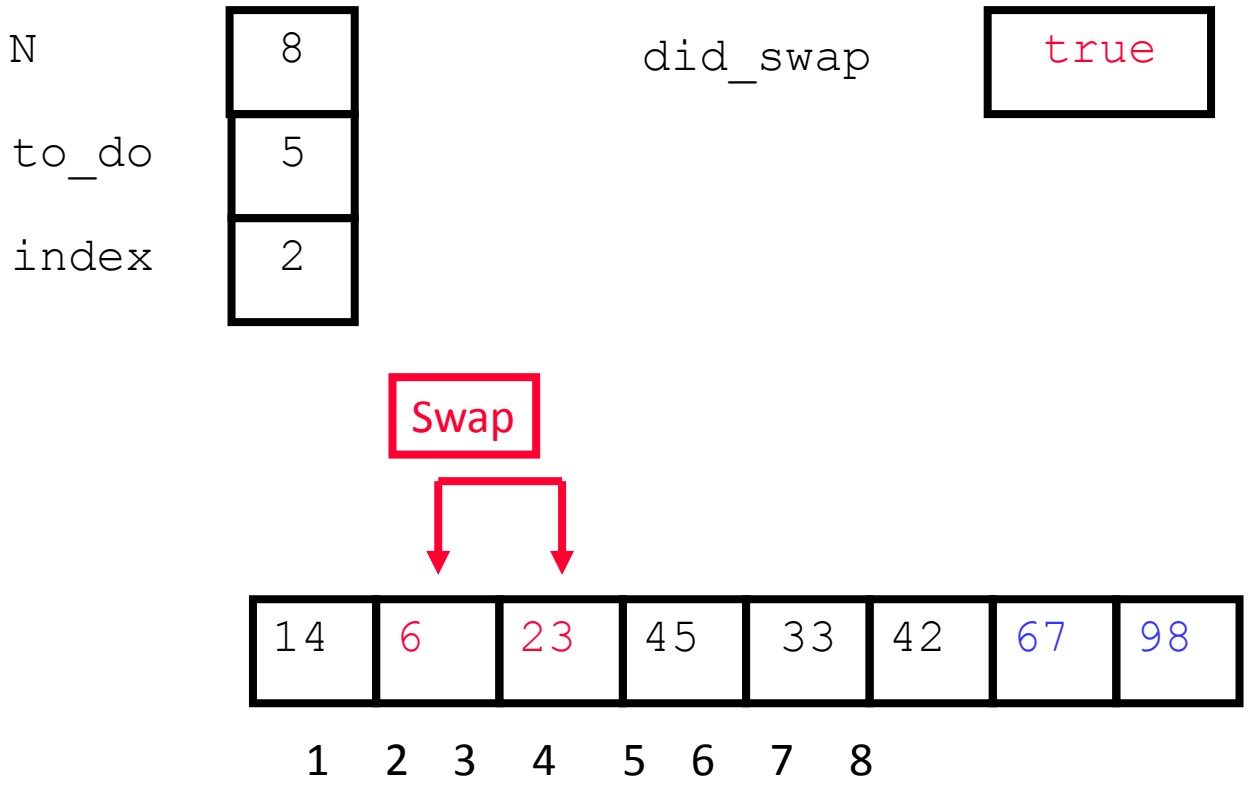
The Third “Bubble Up”



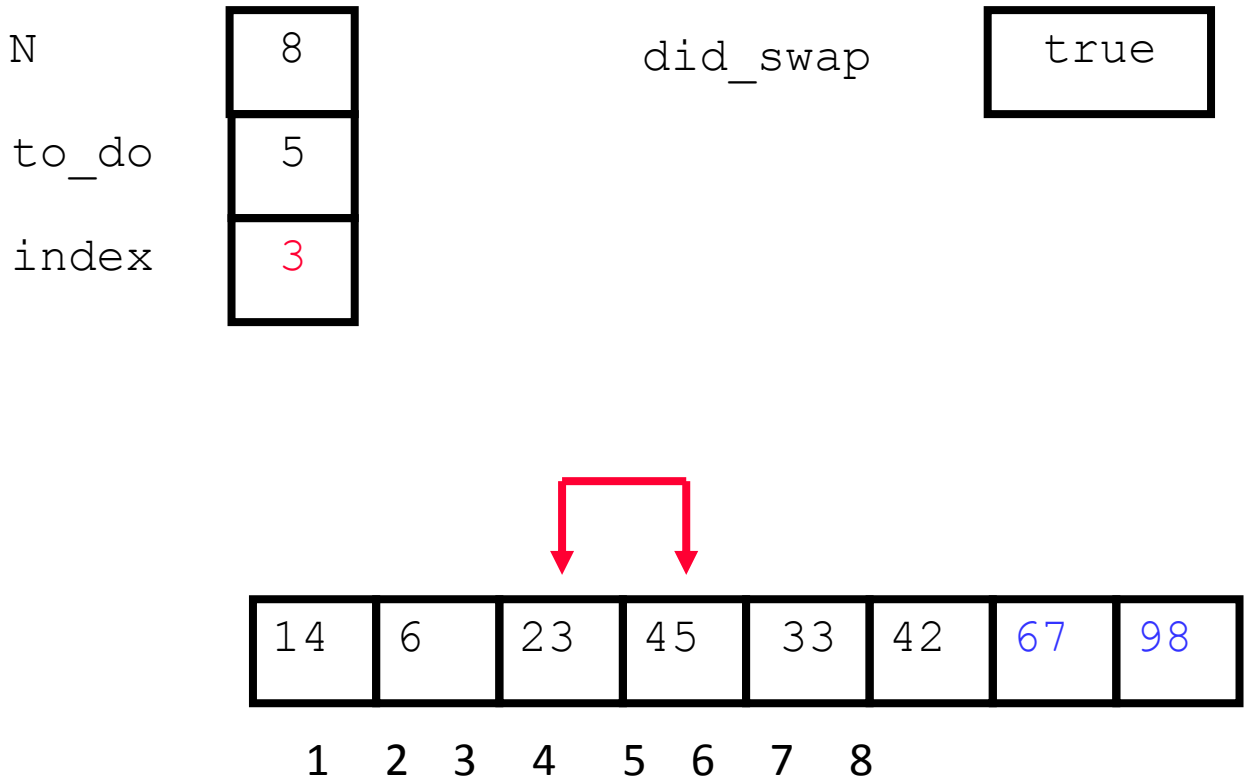
The Third “Bubble Up”



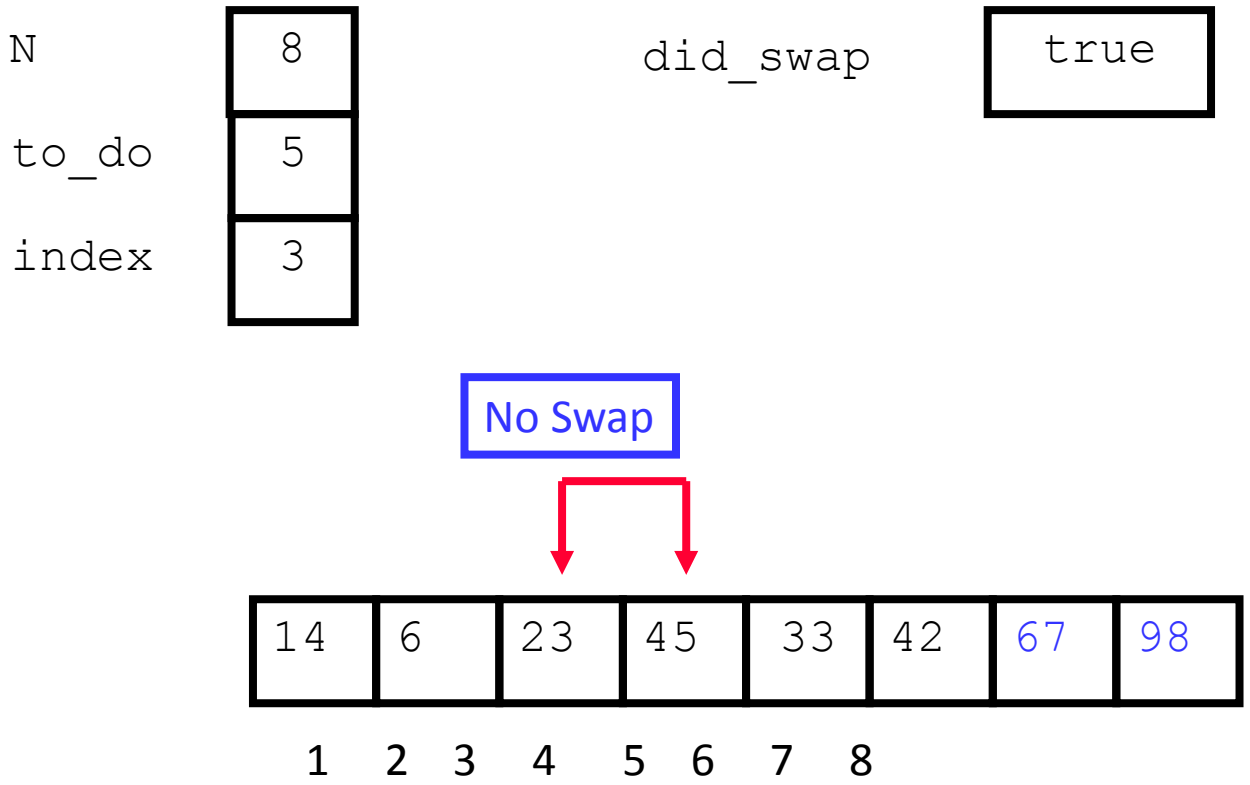
The Third “Bubble Up”



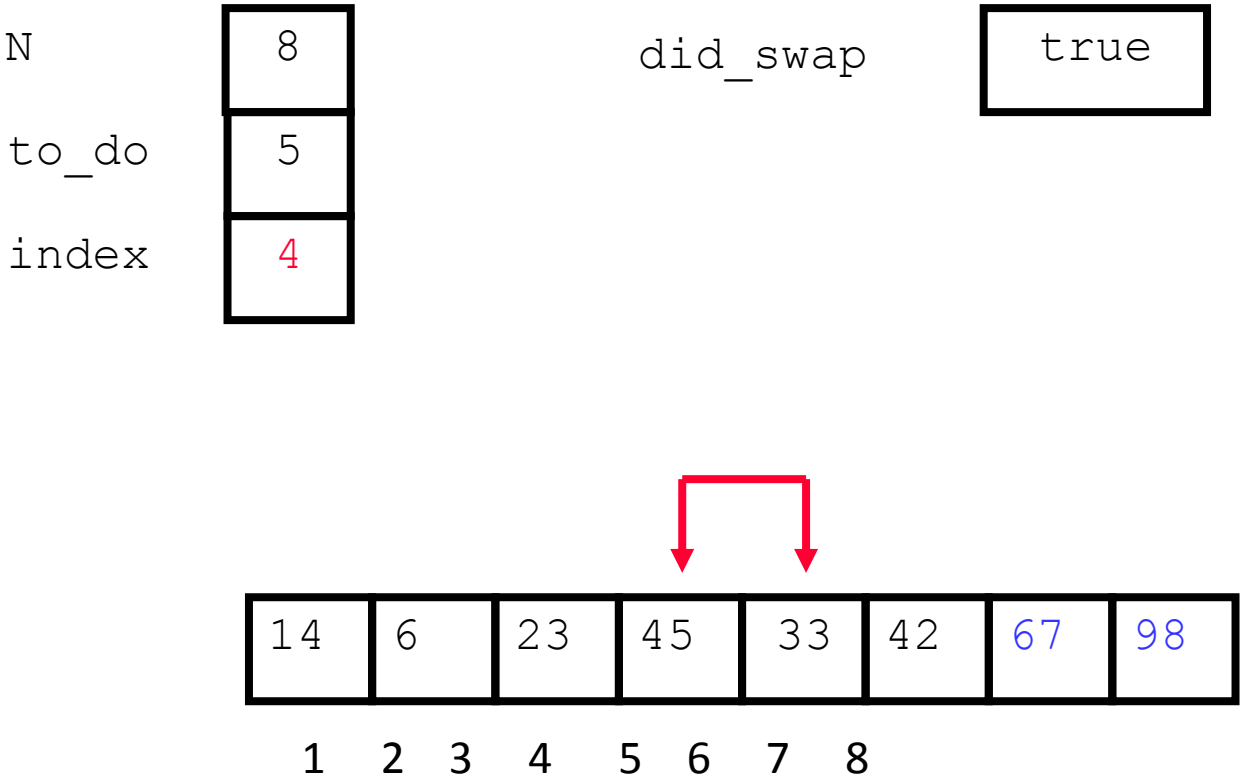
The Third “Bubble Up”



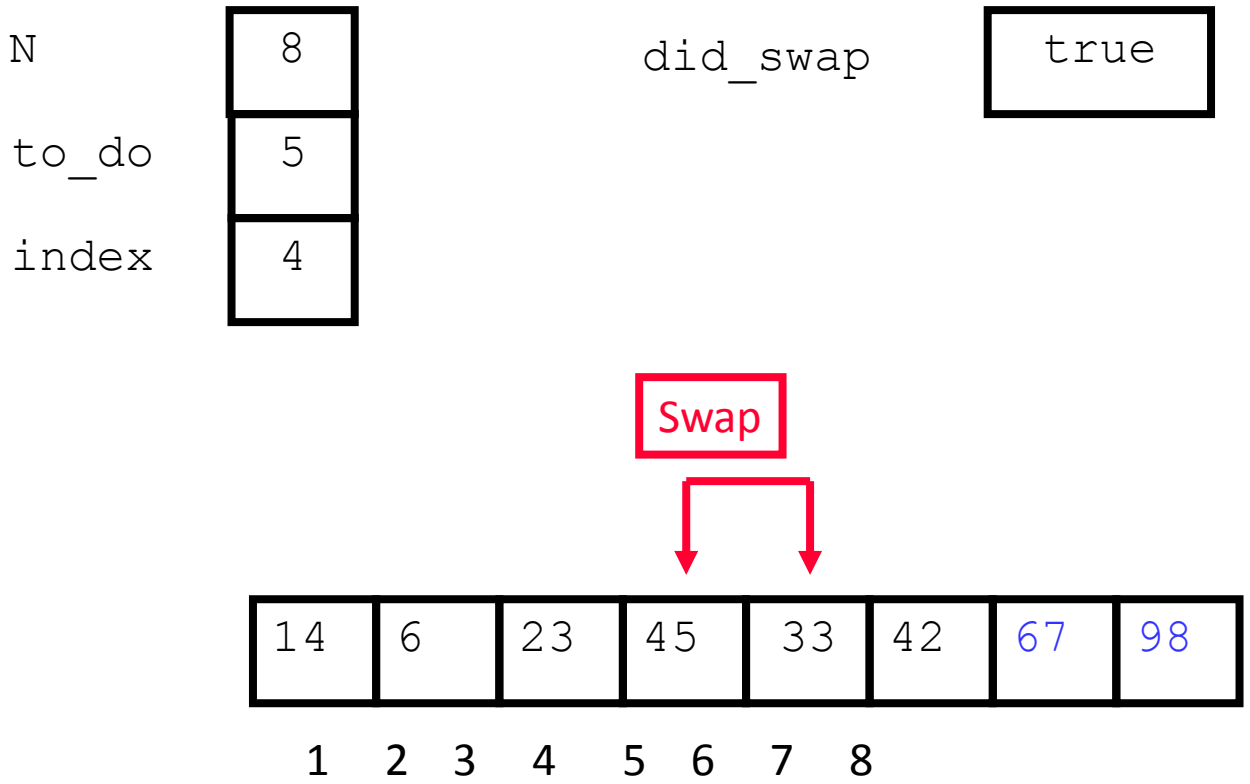
The Third “Bubble Up”



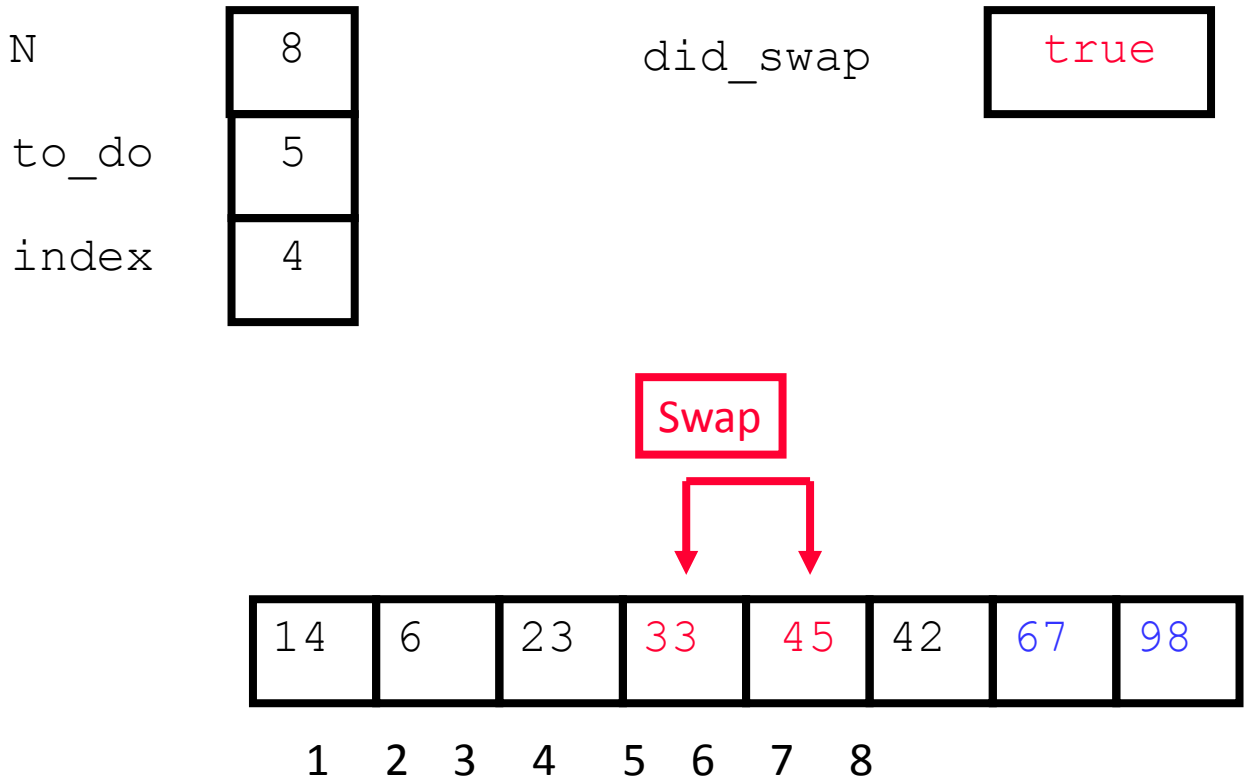
The Third “Bubble Up”



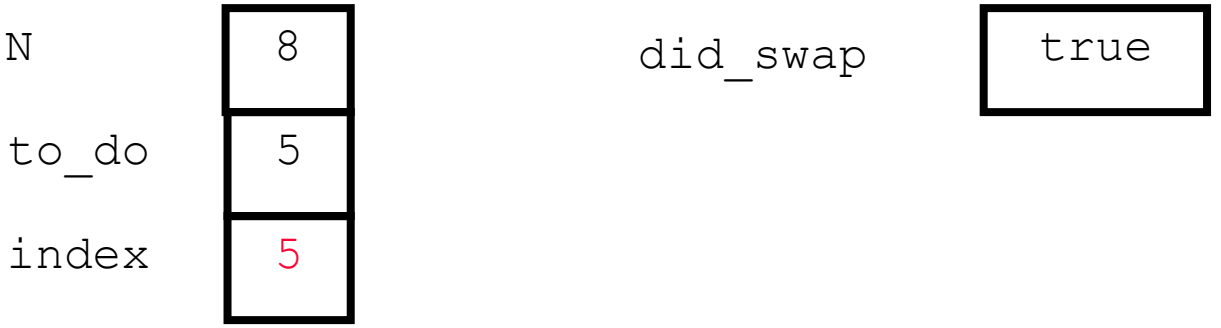
The Third “Bubble Up”

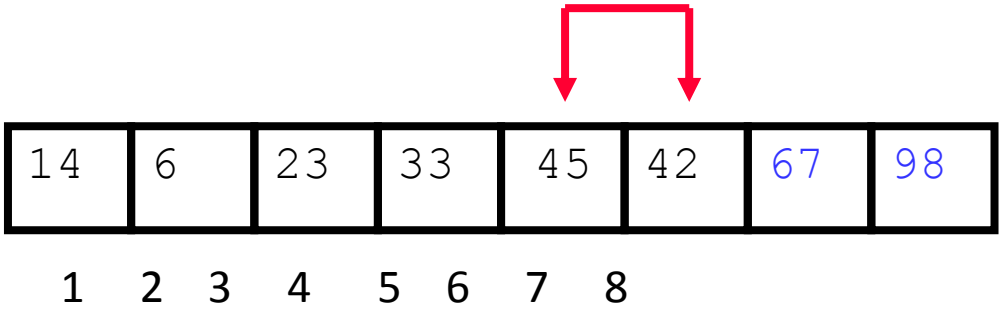


The Third “Bubble Up”

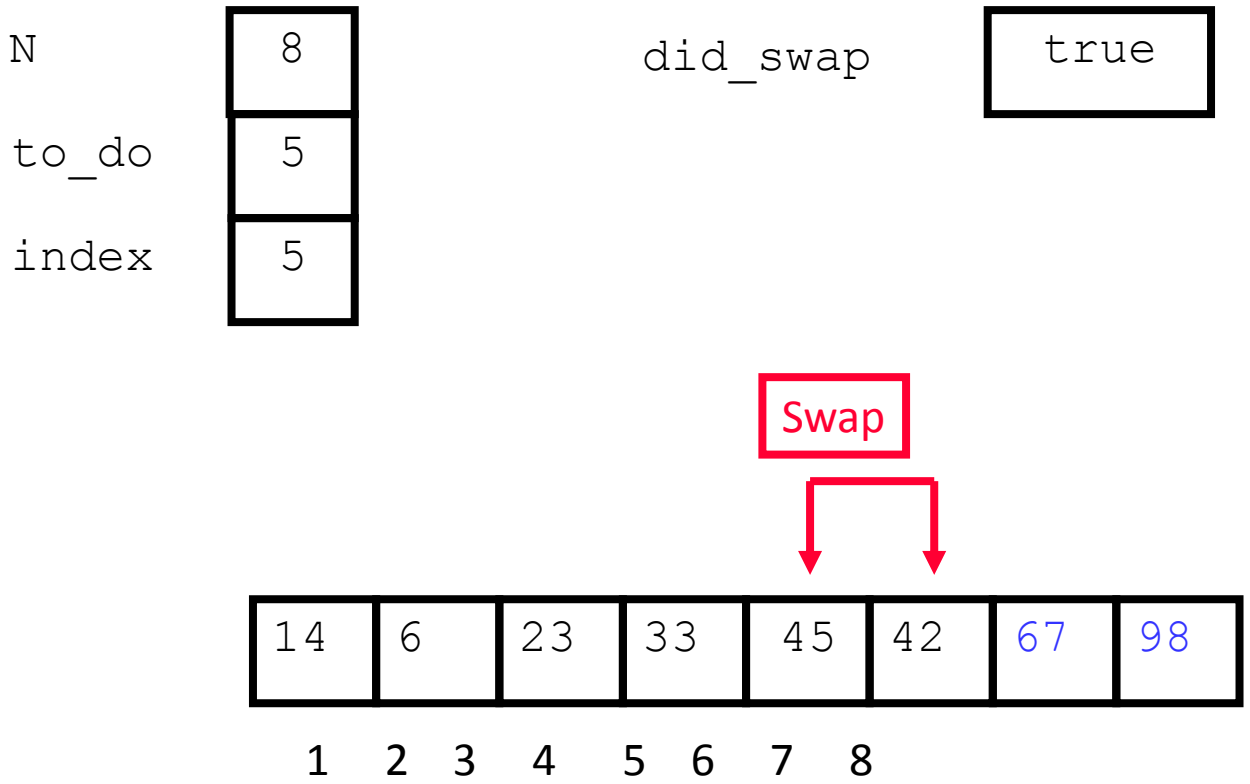


The Third “Bubble Up”

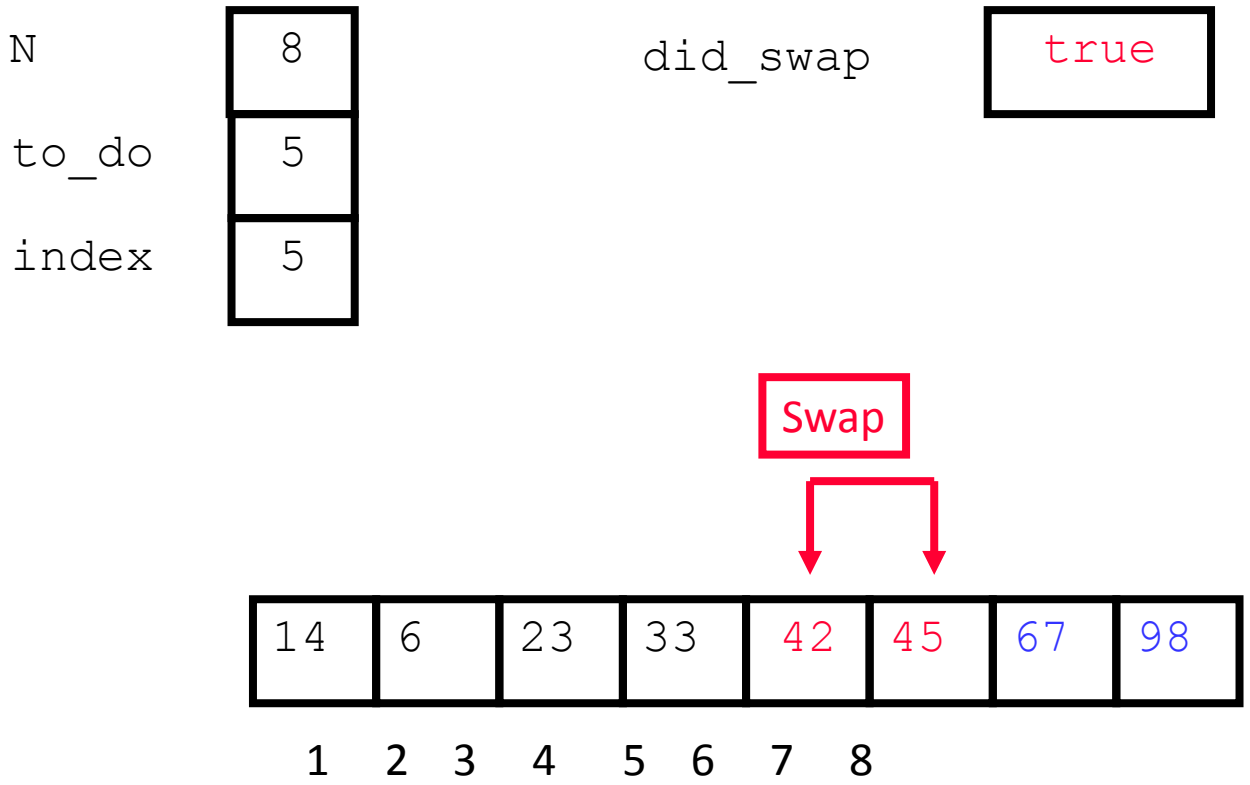




The Third “Bubble Up”



The Third “Bubble Up”

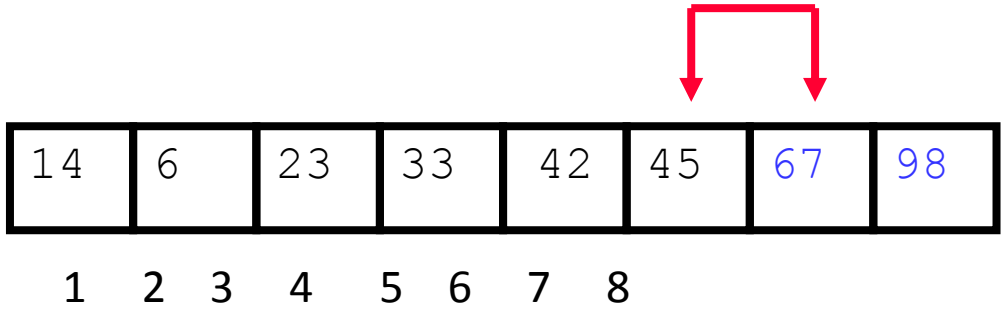


After Third Pass of Outer Loop

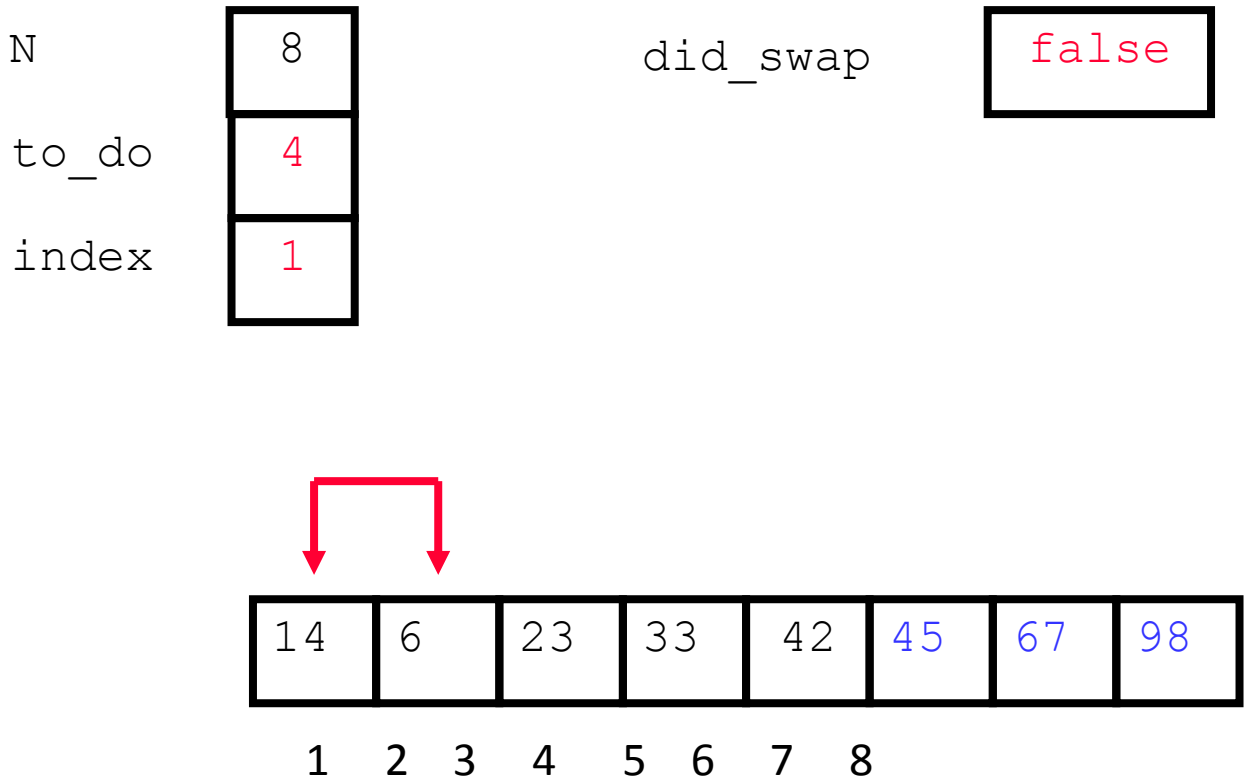
N	8
to_do	5
index	6

did_swap	true

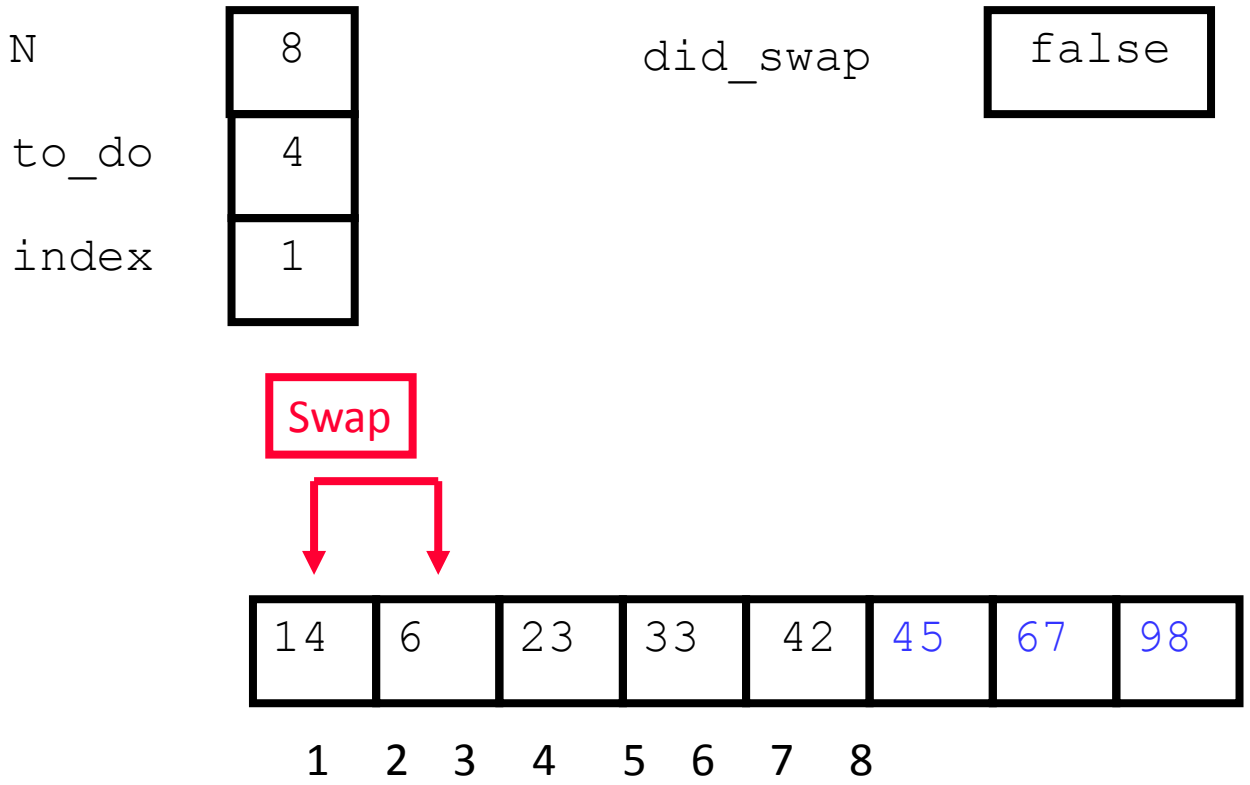
Finished third “Bubble Up”



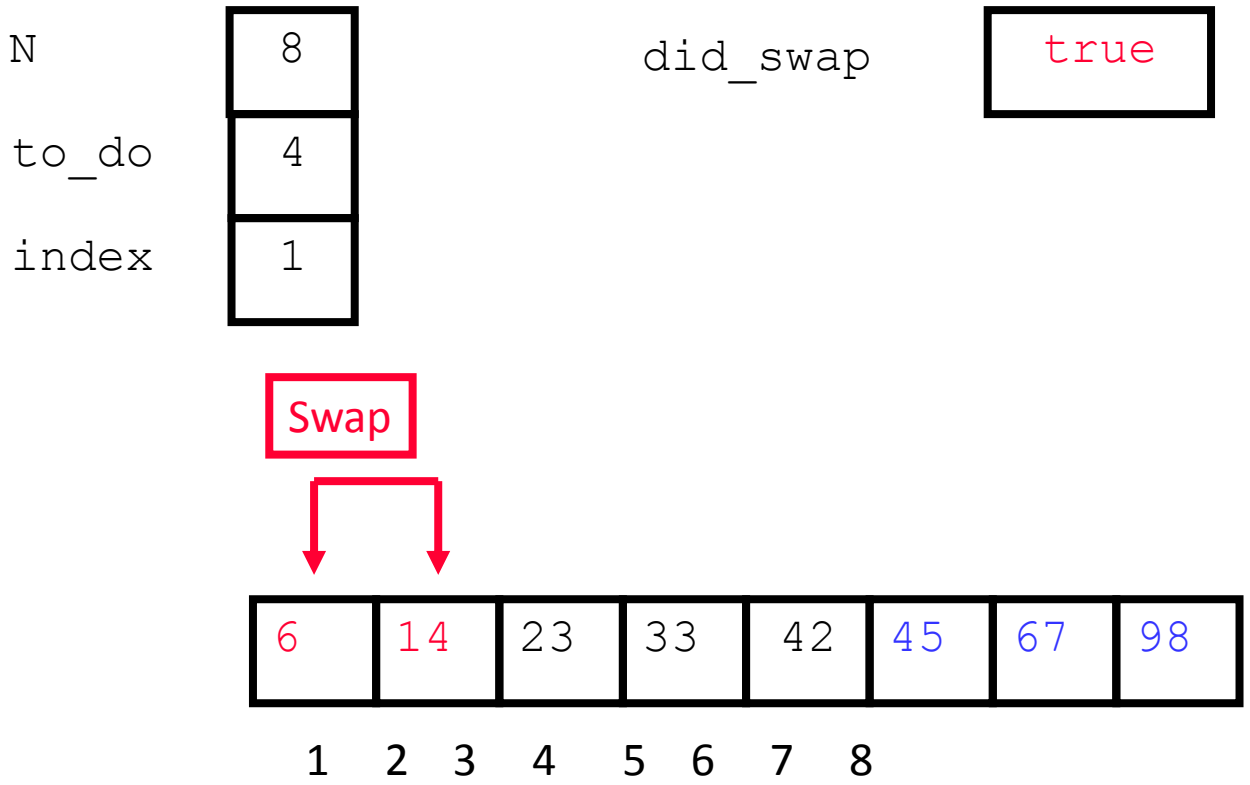
The Fourth “Bubble Up”



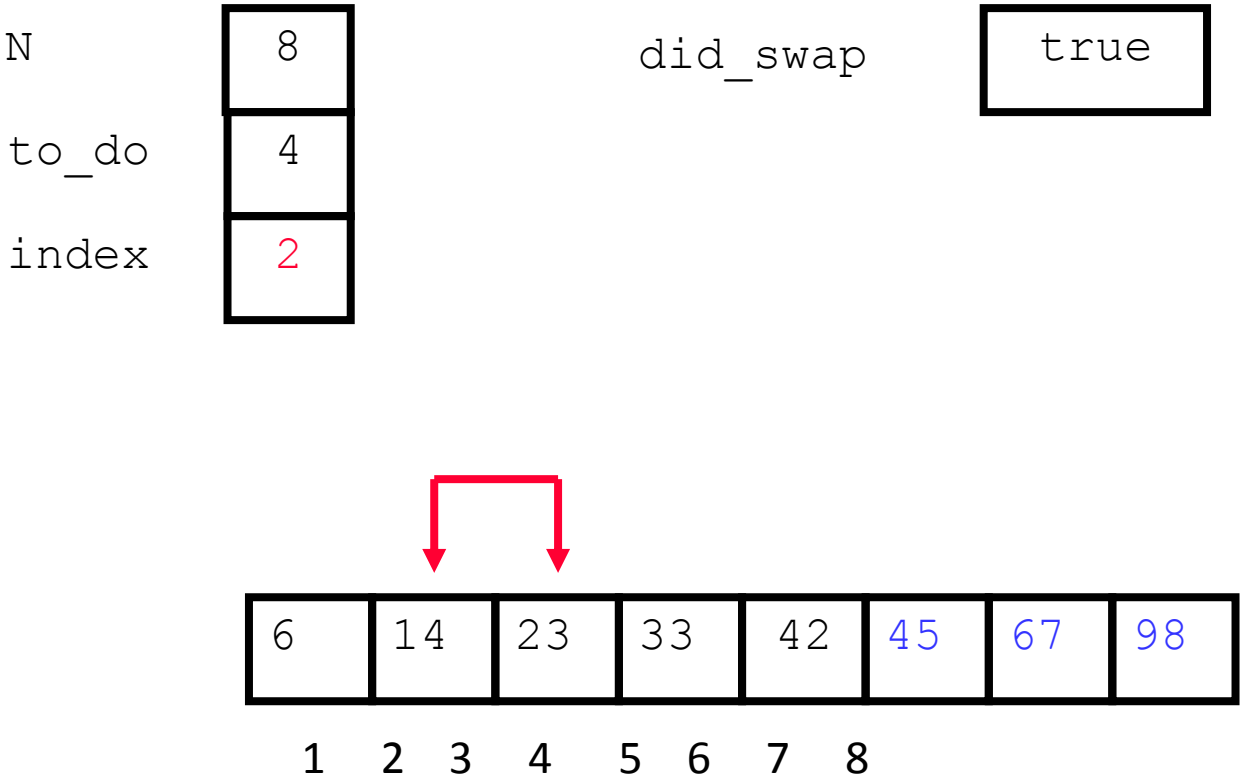
The Fourth “Bubble Up”



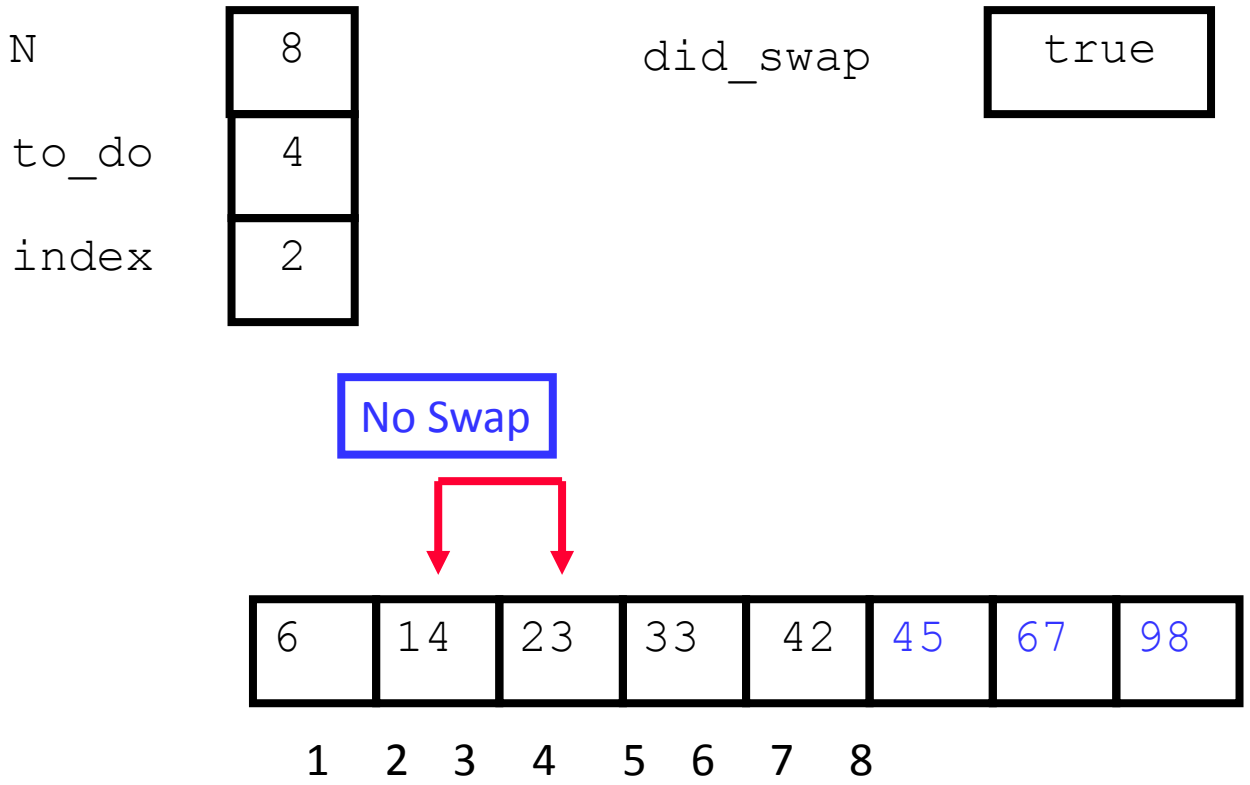
The Fourth “Bubble Up”



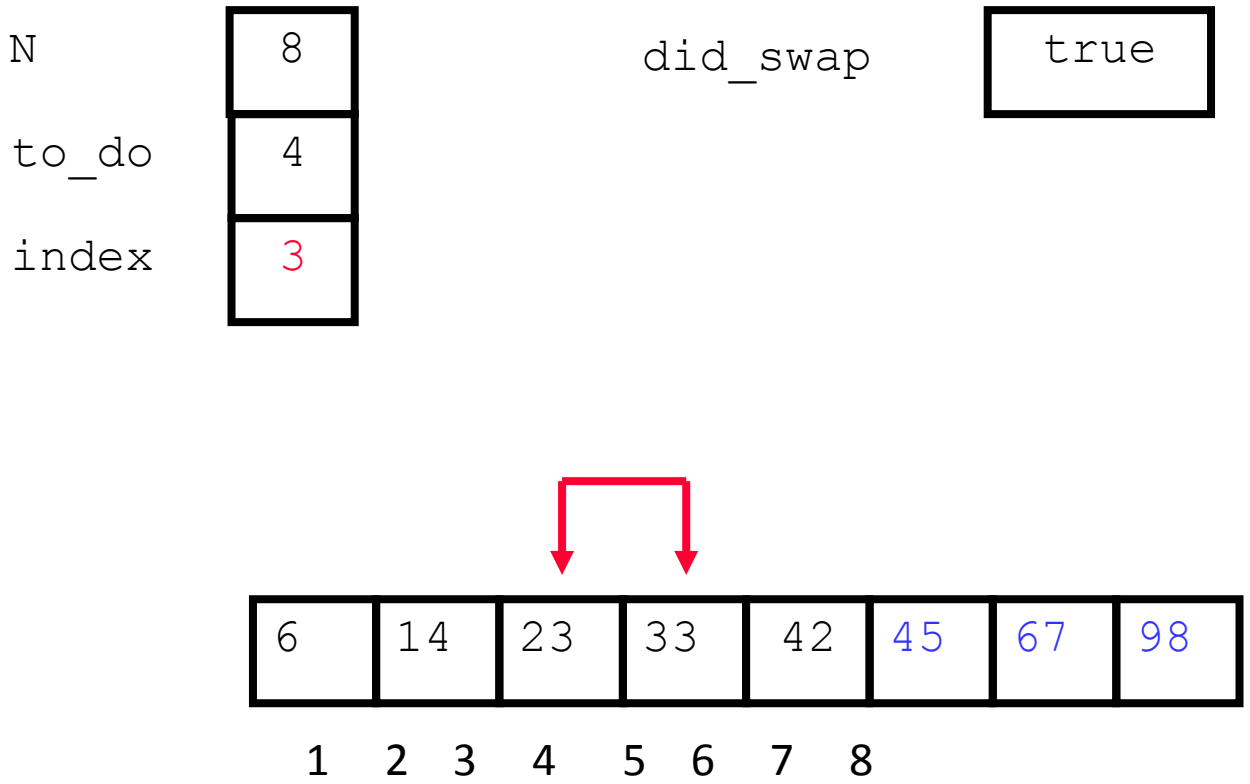
The Fourth “Bubble Up”



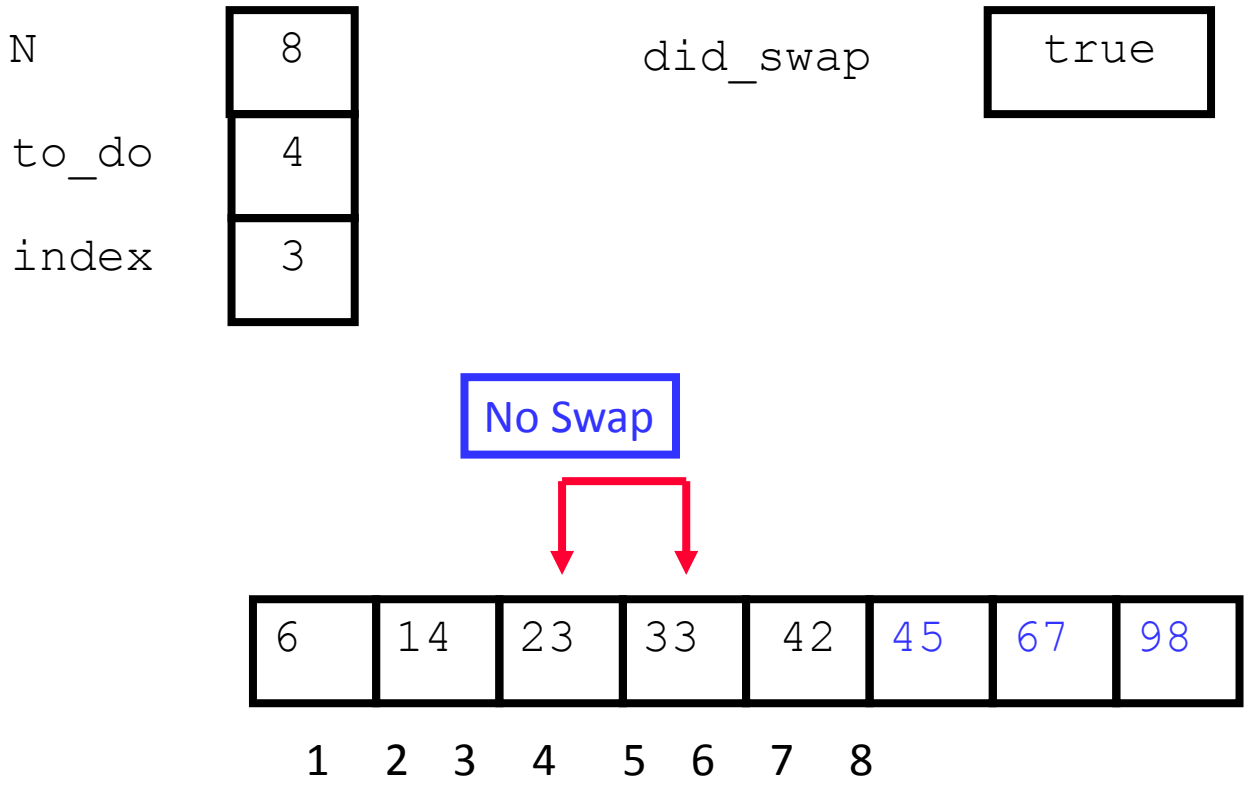
The Fourth “Bubble Up”



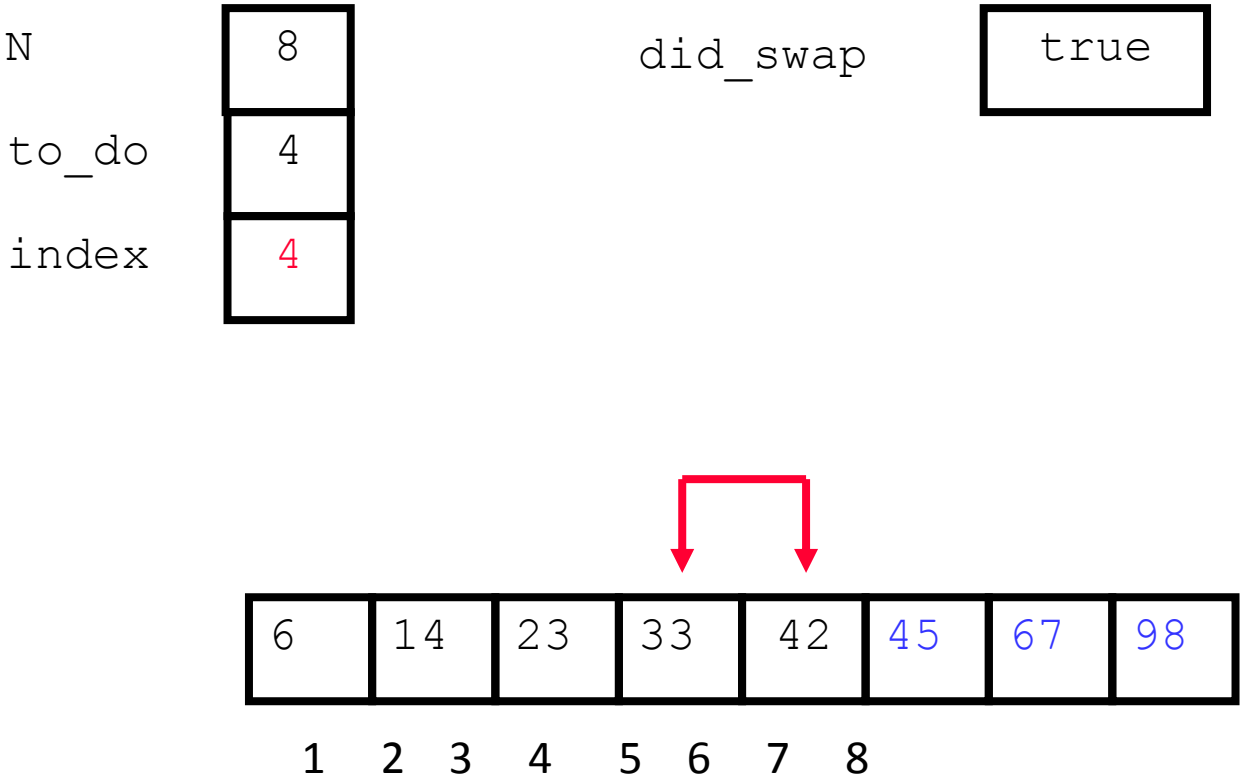
The Fourth “Bubble Up”



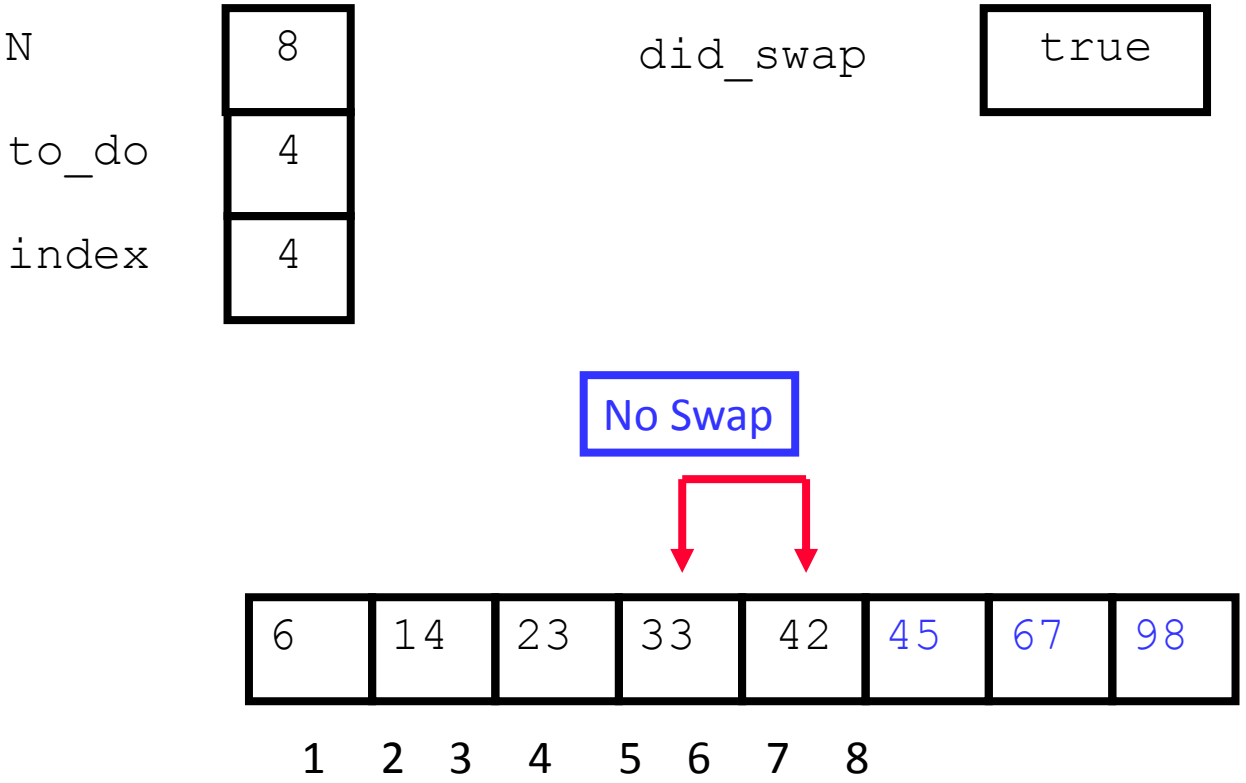
The Fourth “Bubble Up”



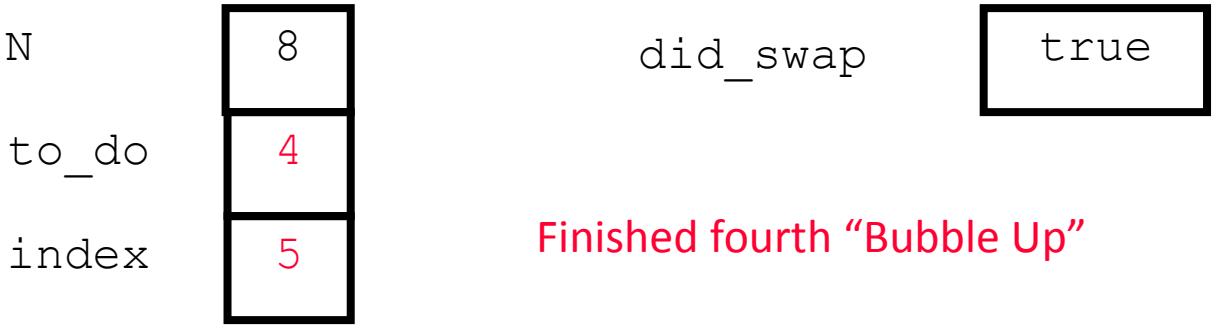
The Fourth “Bubble Up”

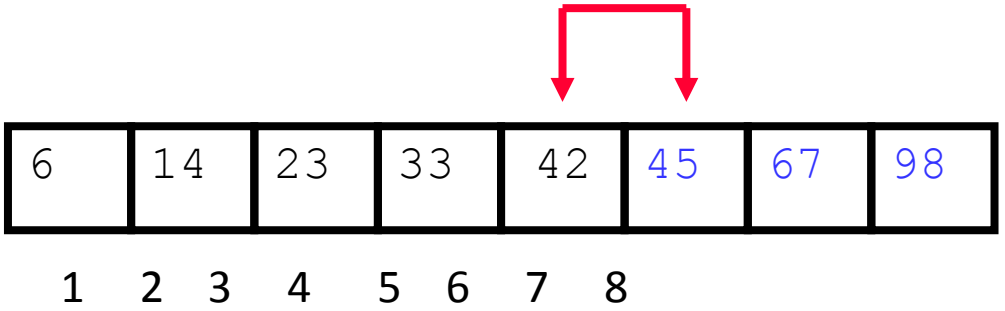


The Fourth “Bubble Up”

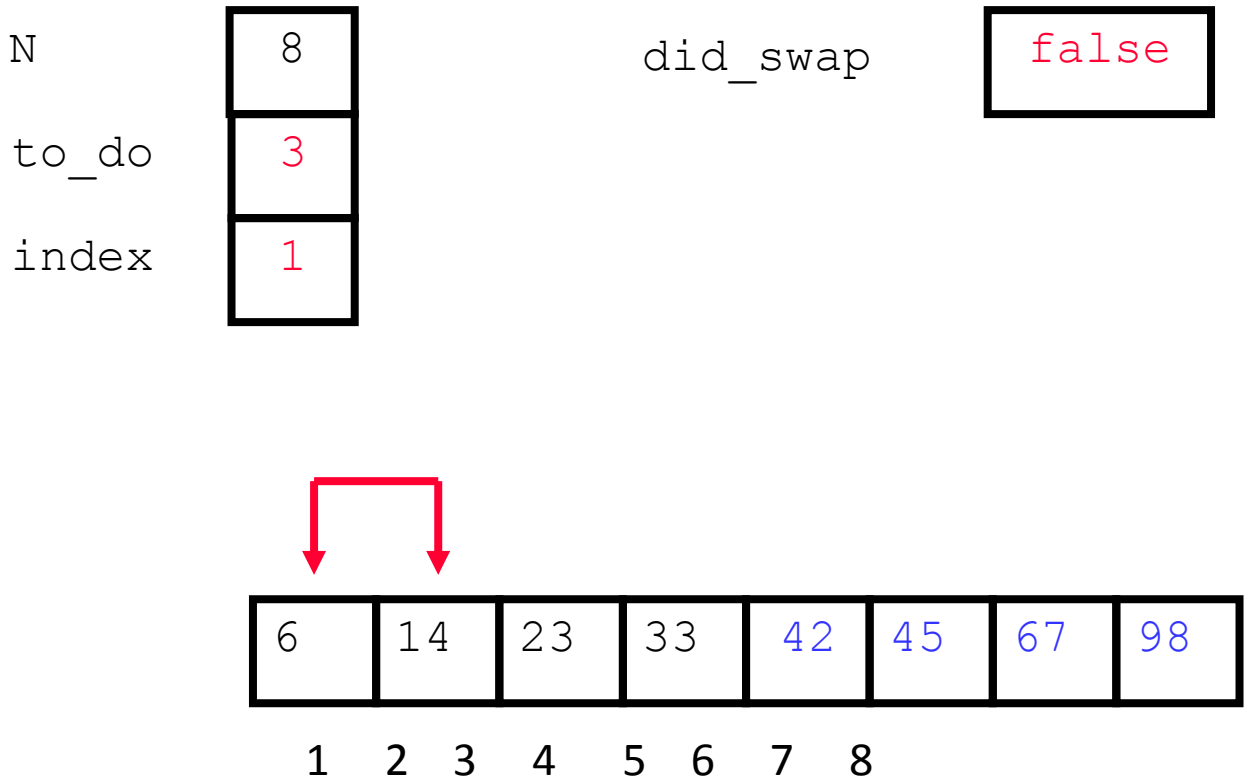


After Fourth Pass of Outer Loop

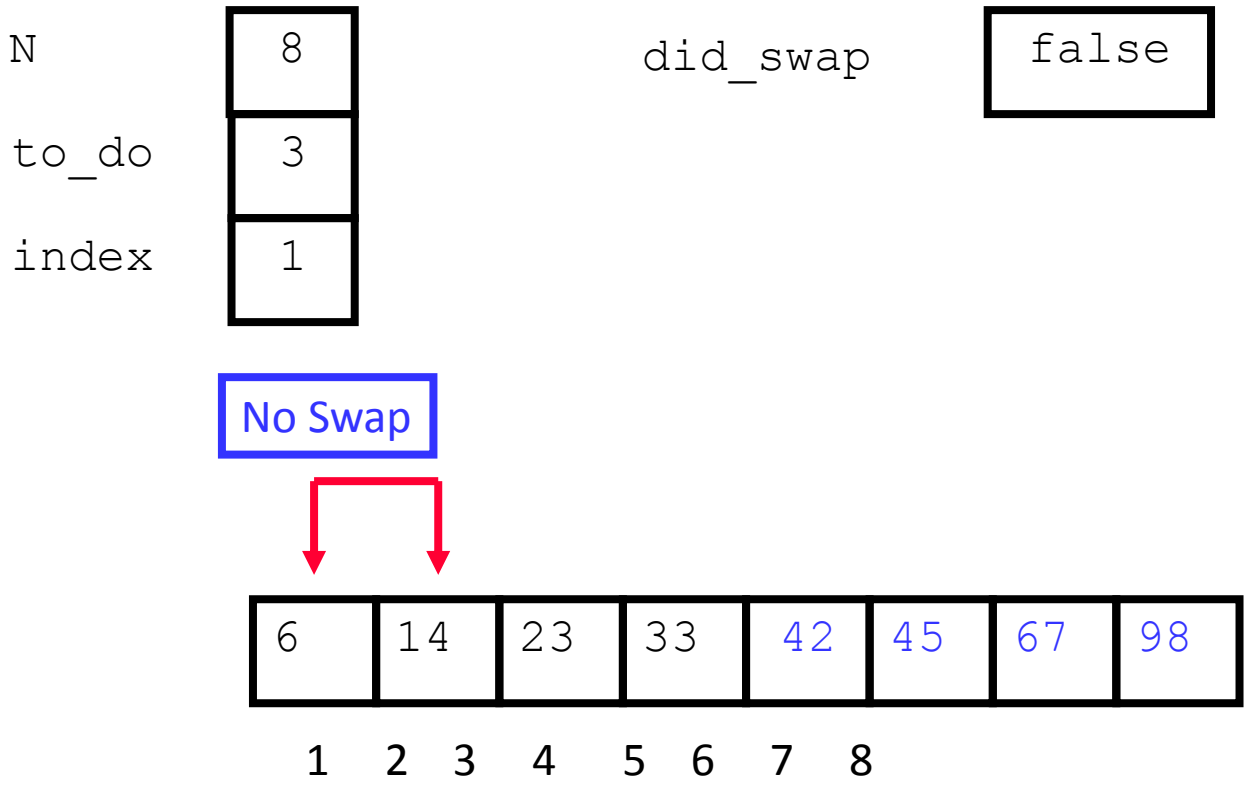




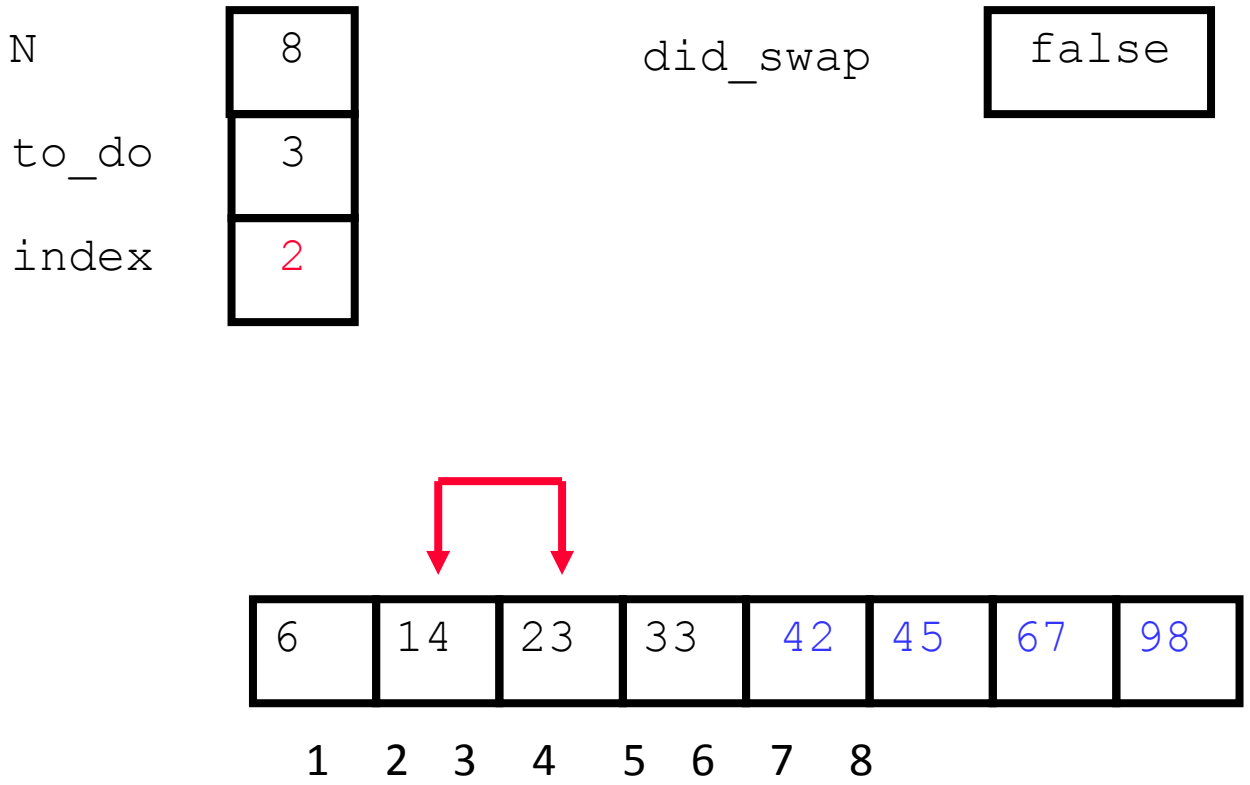
The Fifth “Bubble Up”



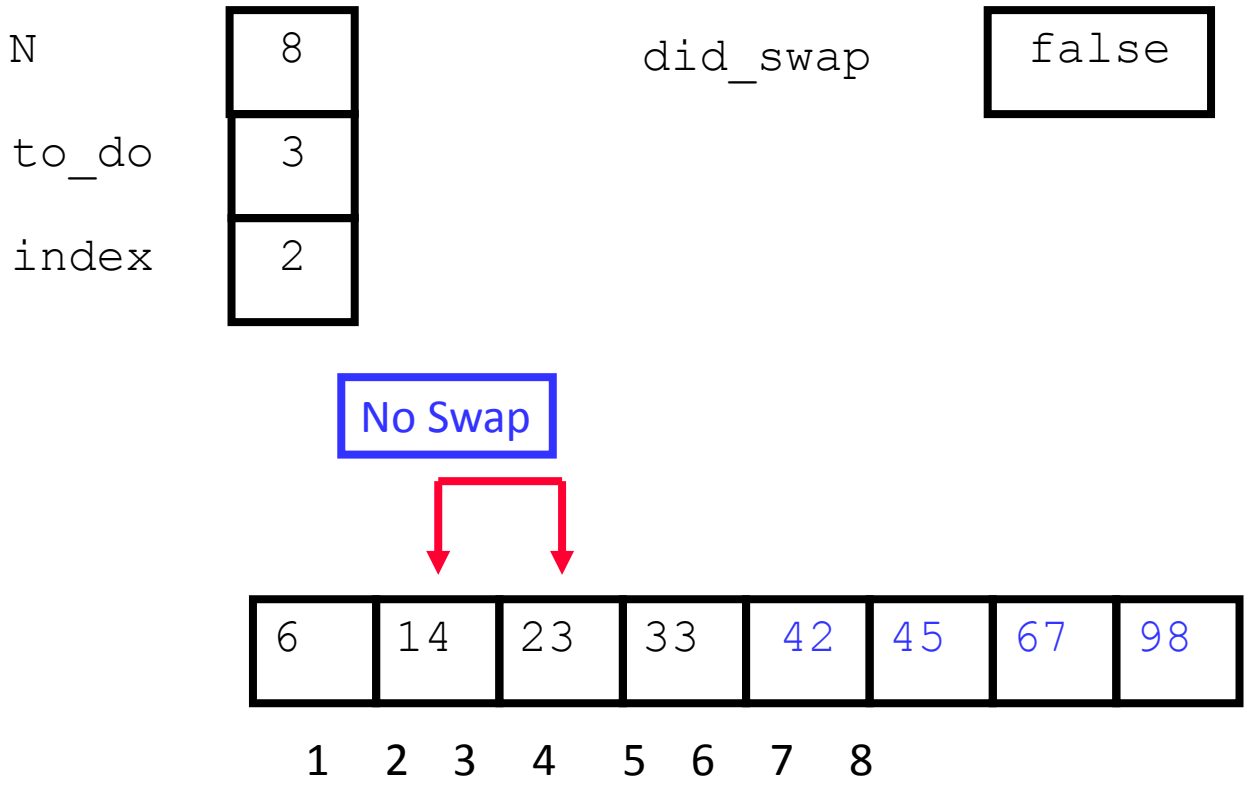
The Fifth “Bubble Up”



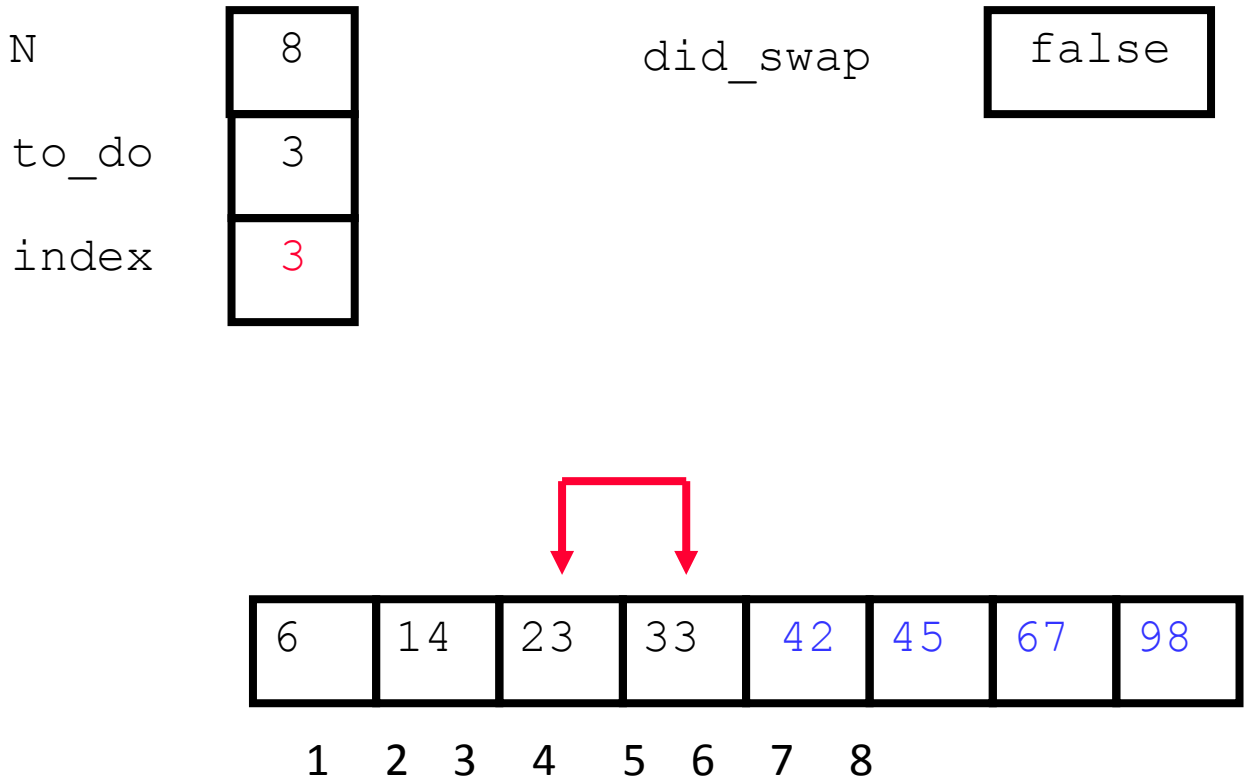
The Fifth “Bubble Up”



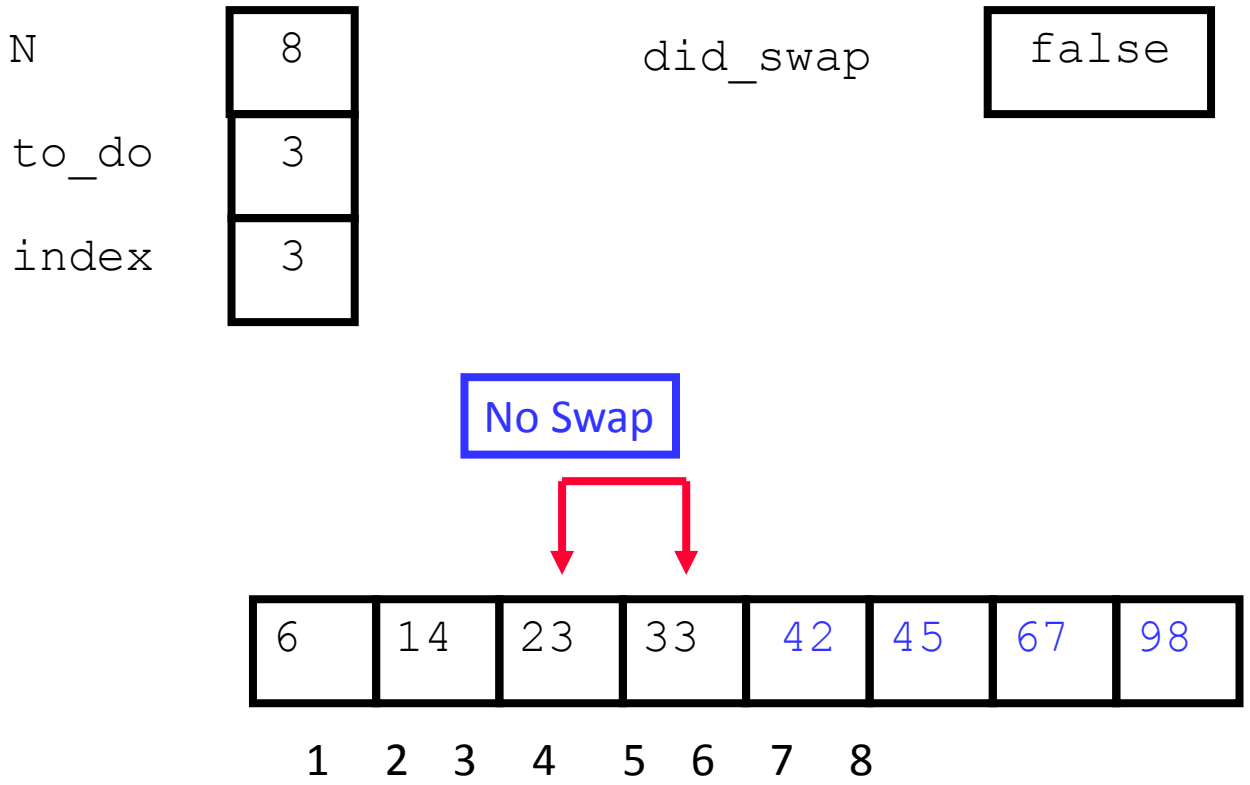
The Fifth “Bubble Up”



The Fifth “Bubble Up”



The Fifth “Bubble Up”

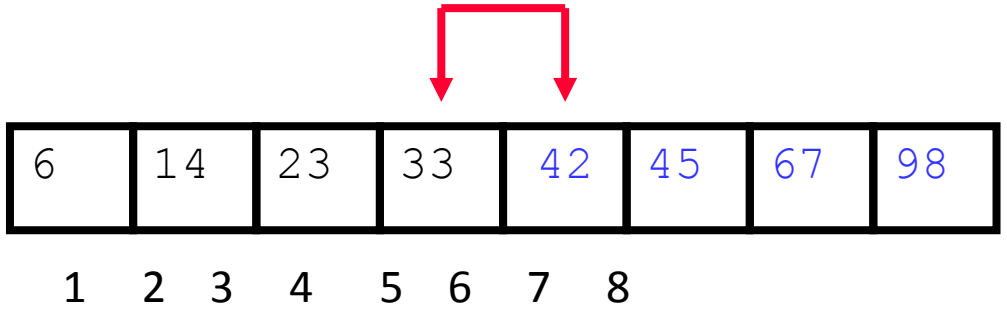


After Fifth Pass of Outer Loop

N	8
to_do	3
index	4

did_swap	false

Finished fifth “Bubble Up”



Finished “Early”

N	8
to_do	3
index	4

did_swap	false

We didn’t do any swapping, so all of the other elements must be correctly placed.

We can “skip” the last two passes of the outer loop.

6	14	23	33	42	45	67	98
1	2	3	4	5	6	7	8

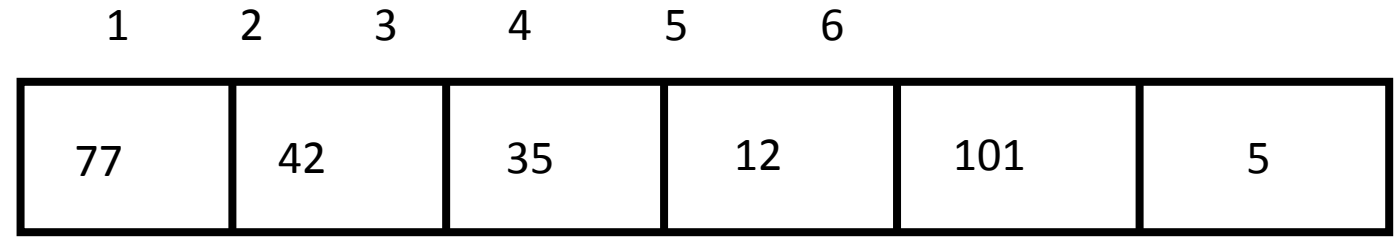
Summary

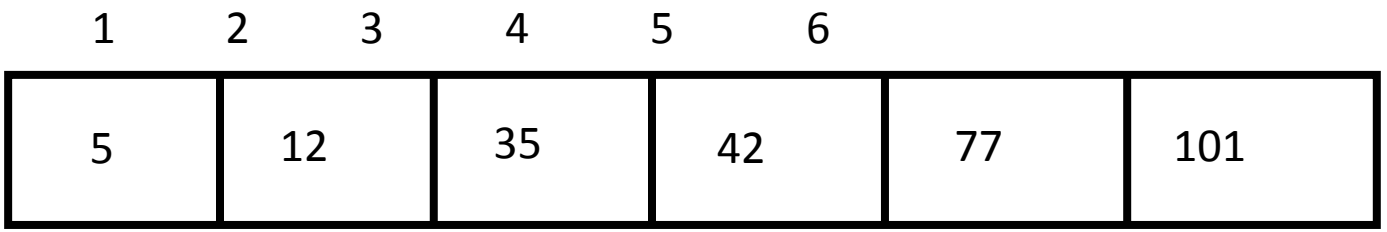
- **"Bubble Up" algorithm will move largest value to its correct location (to the right)**
- **Repeat "Bubble Up" until all elements are correctly placed:**
  - **Maximum of N-1 times**
  - **Can finish early if no swapping occurs**
- **We reduce the number of elements we compare each time one is correctly placed**

Mergesort

Sorting

- **Sorting takes an unordered collection and makes it an ordered one.**





Divide and Conquer

- **Divide and Conquer cuts the problem in half each time, but uses the result of both halves:**
  - **cut the problem in half until the problem is trivial**
  - **solve for both halves**
  - **combine the solutions**

Mergesort

- **A divide-and-conquer algorithm:**
- **Divide the unsorted array into 2 halves until the sub-arrays only contain one element**
- **Merge the sub-problem solutions together:**
  - **Compare the sub-array's first elements**
  - **Remove the smallest element and put it into the result array**
  - **Continue the process until all elements have been put into the result array**

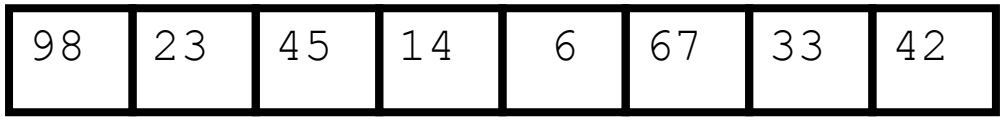
37	23	6	89	15	12	2	19

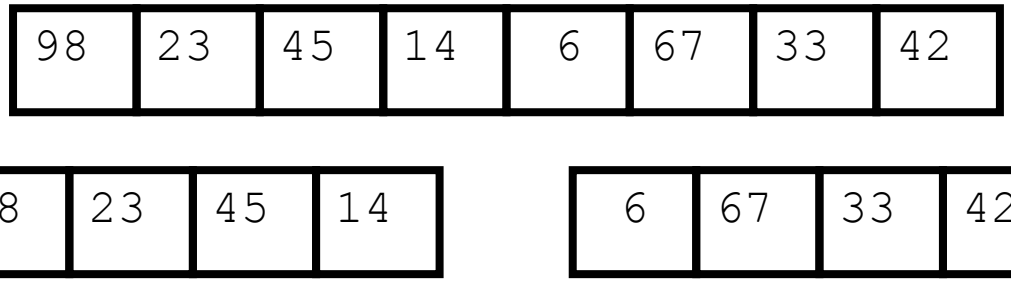
Algorithm

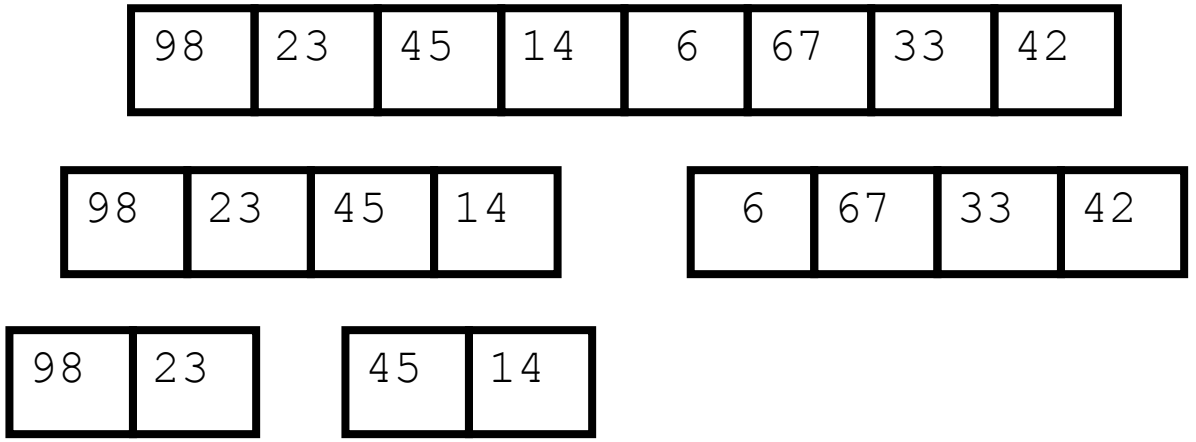
```
Mergesort(Passed an array)
  if array size > 1
    Divide array in half
    Call Mergesort on first half.
    Call Mergesort on second half.
    Merge two halves.

Merge(Passed two arrays)
  Compare leading element in each array
  Select lower and place in new array.
    (If one input array is empty then place
     remainder of other array in output array)
```

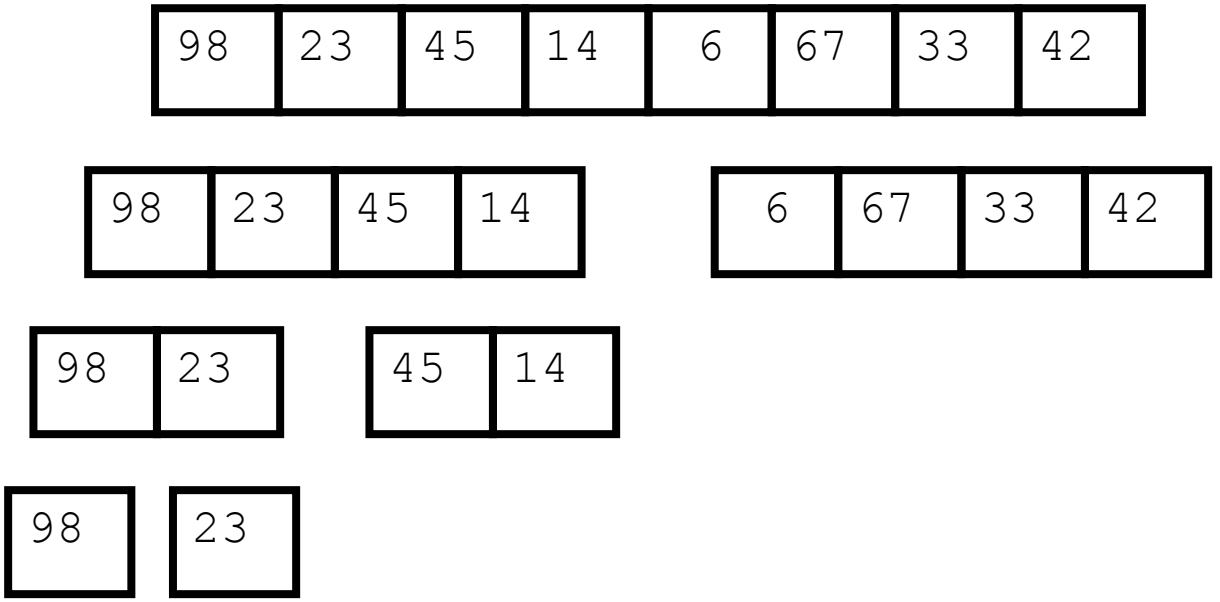
Algorithm

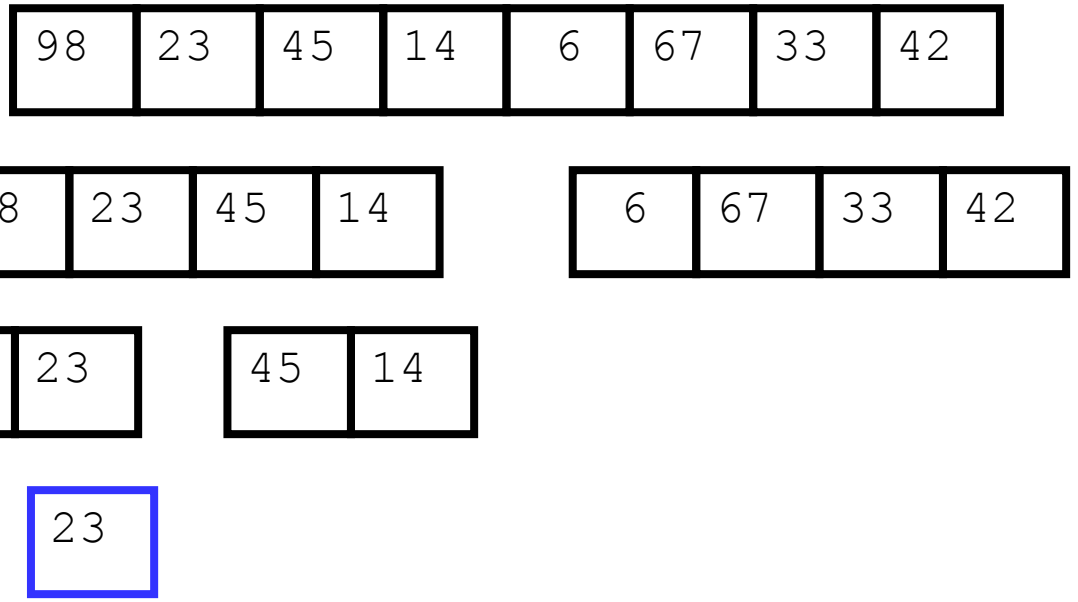
```
Mergesort(Passed an array)
  if array size > 1
    Divide array in half
    Call Mergesort on first half.
    Call Mergesort on second half.
    Merge two halves.

Merge(Passed two arrays)
  Compare leading element in each array
  Select lower and place in new array.
    (If one input array is empty then place
     remainder of other array in output array)
```

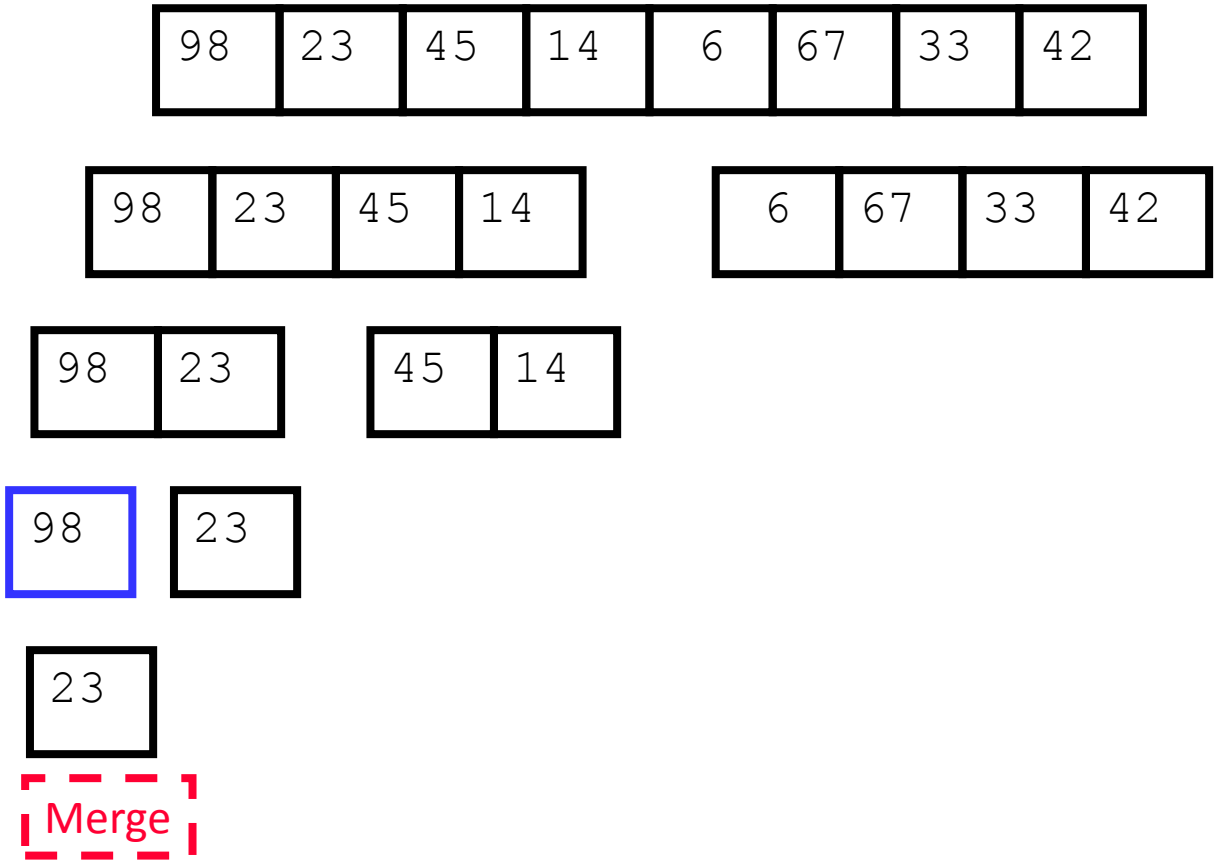


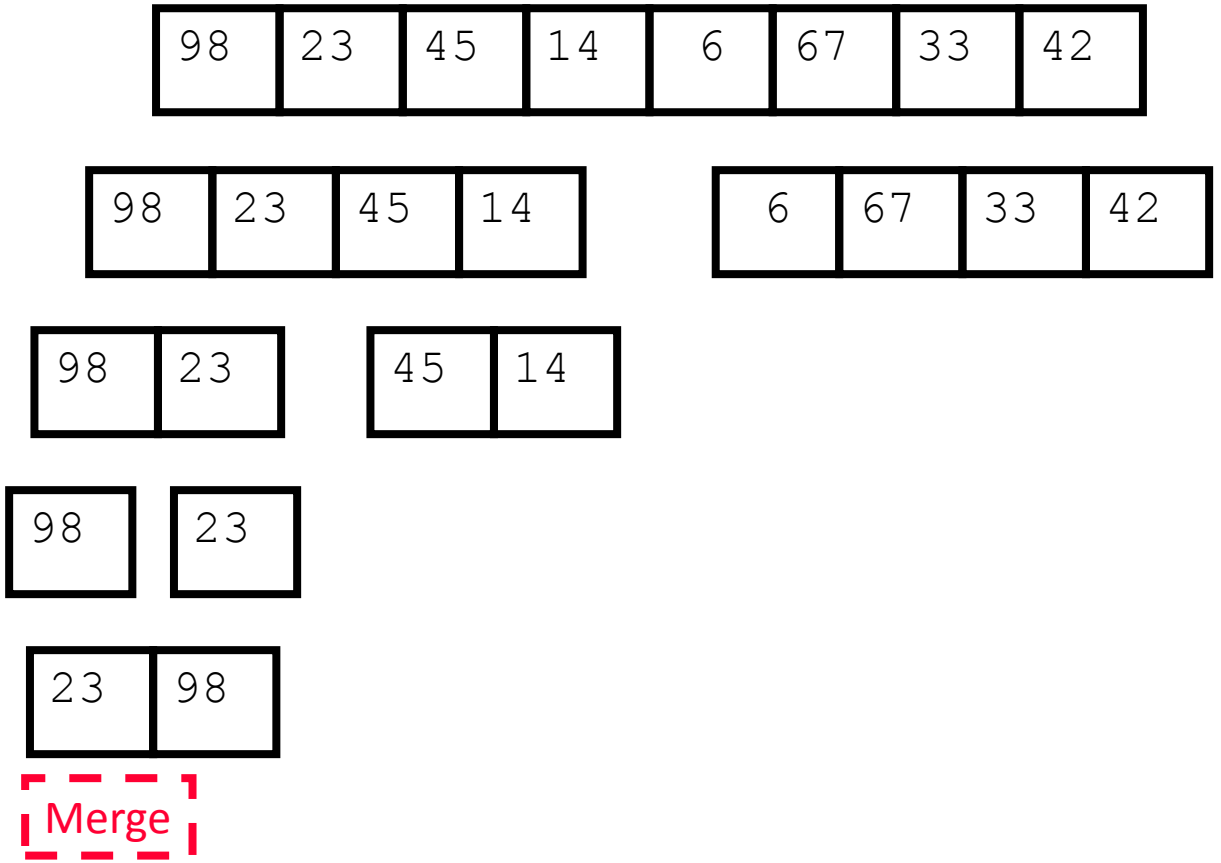


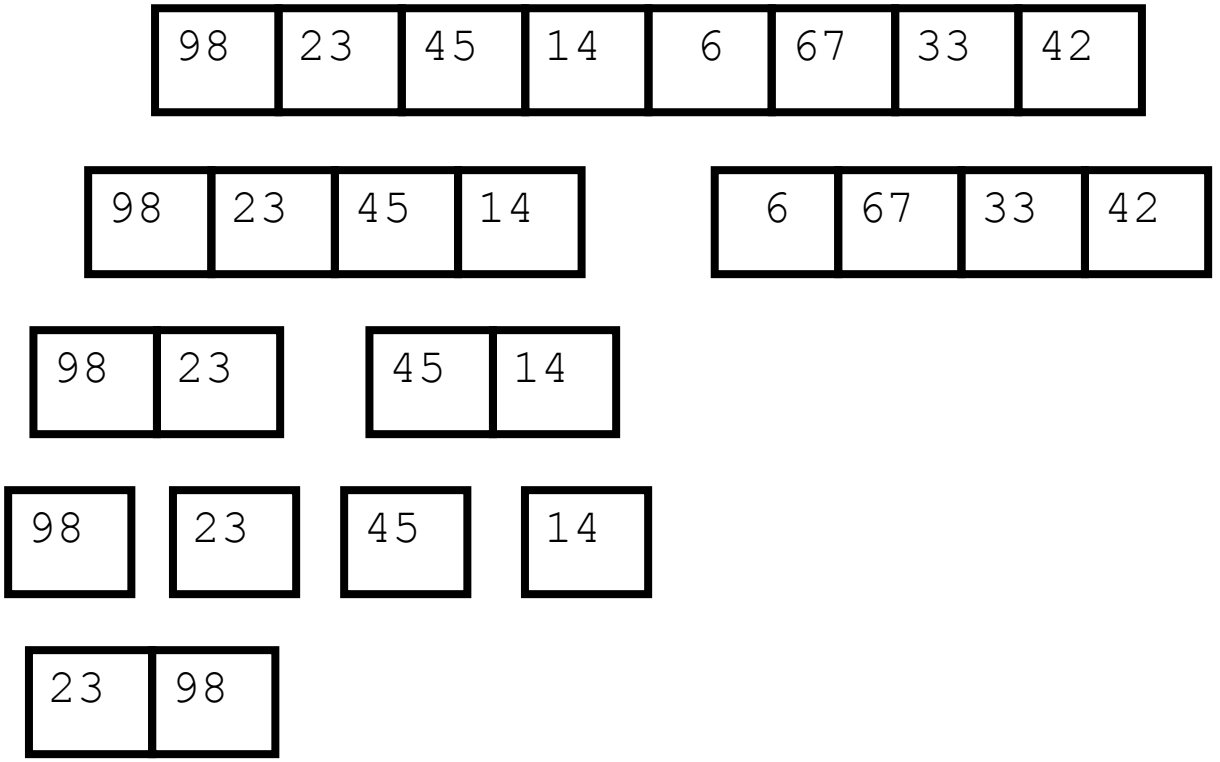


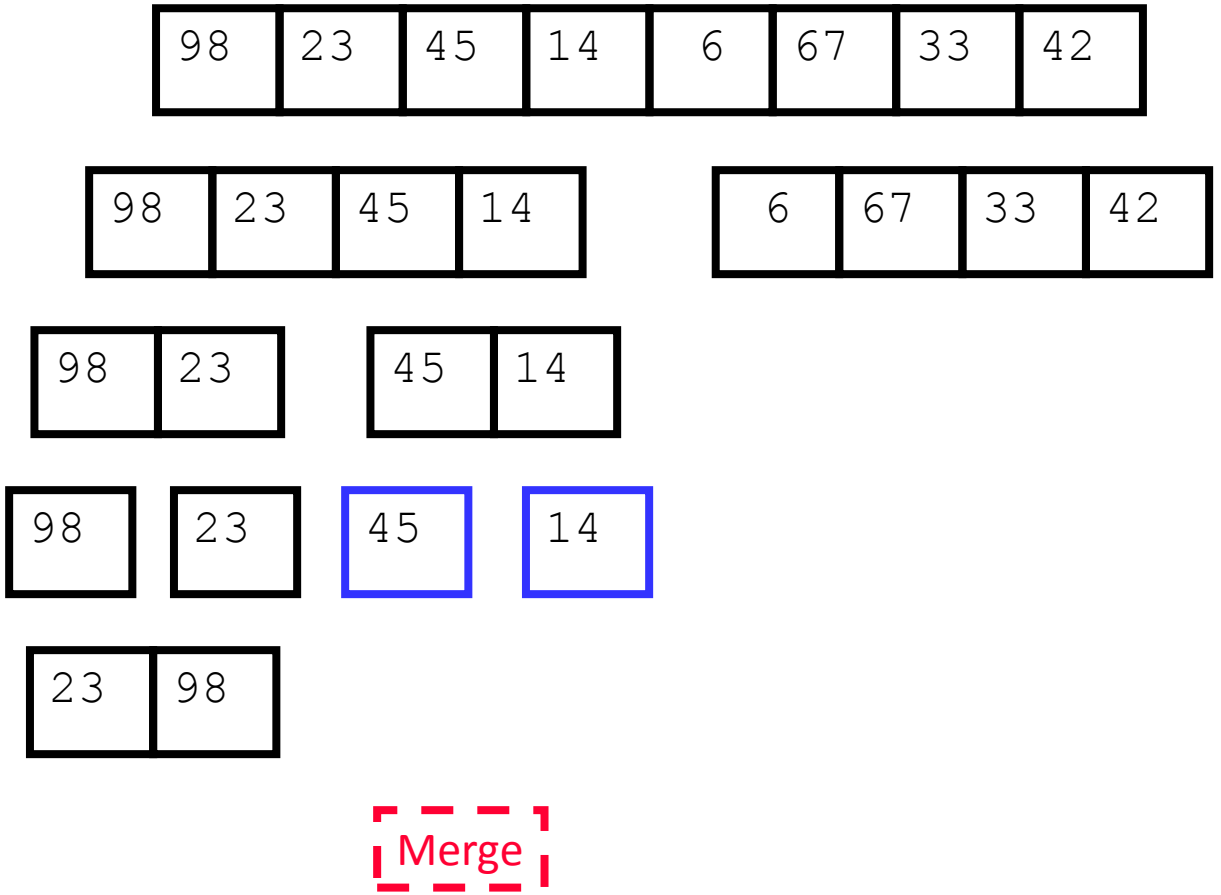


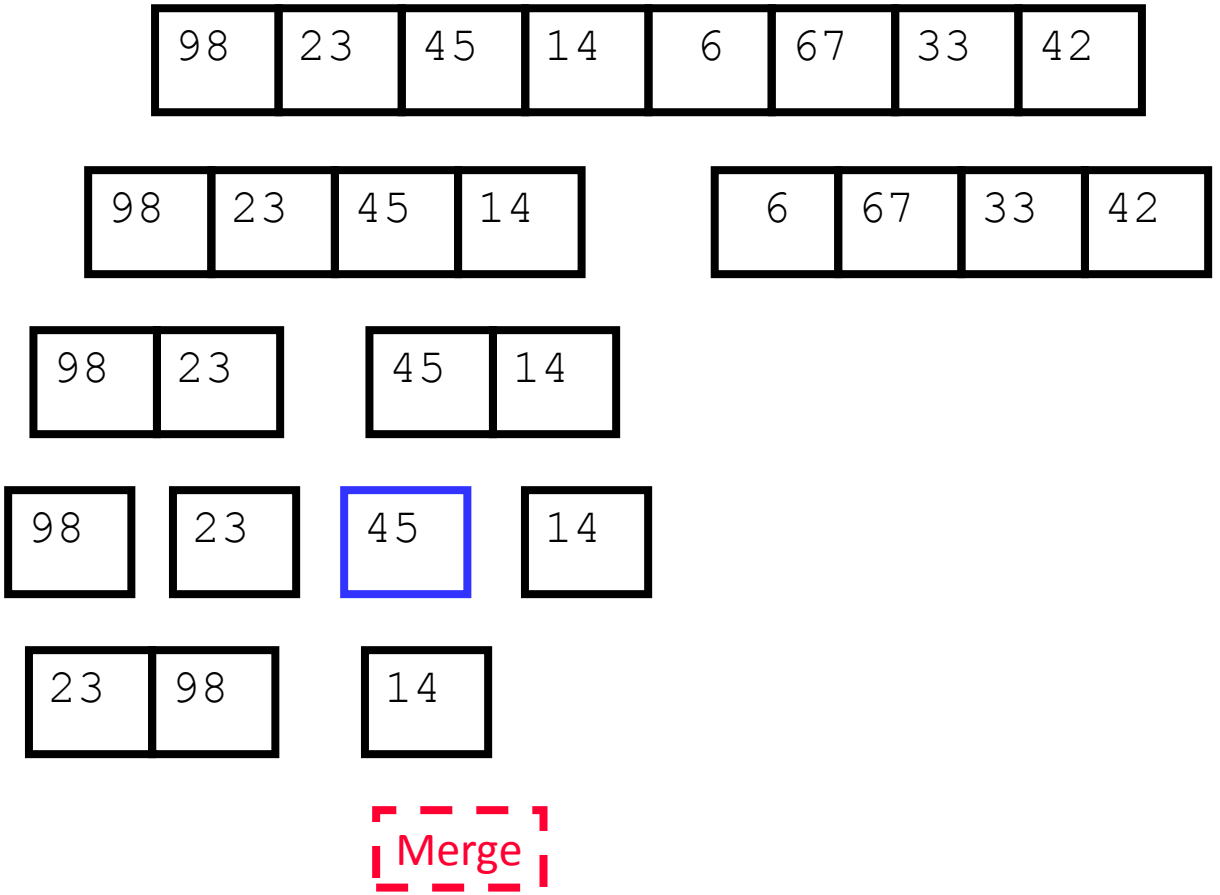


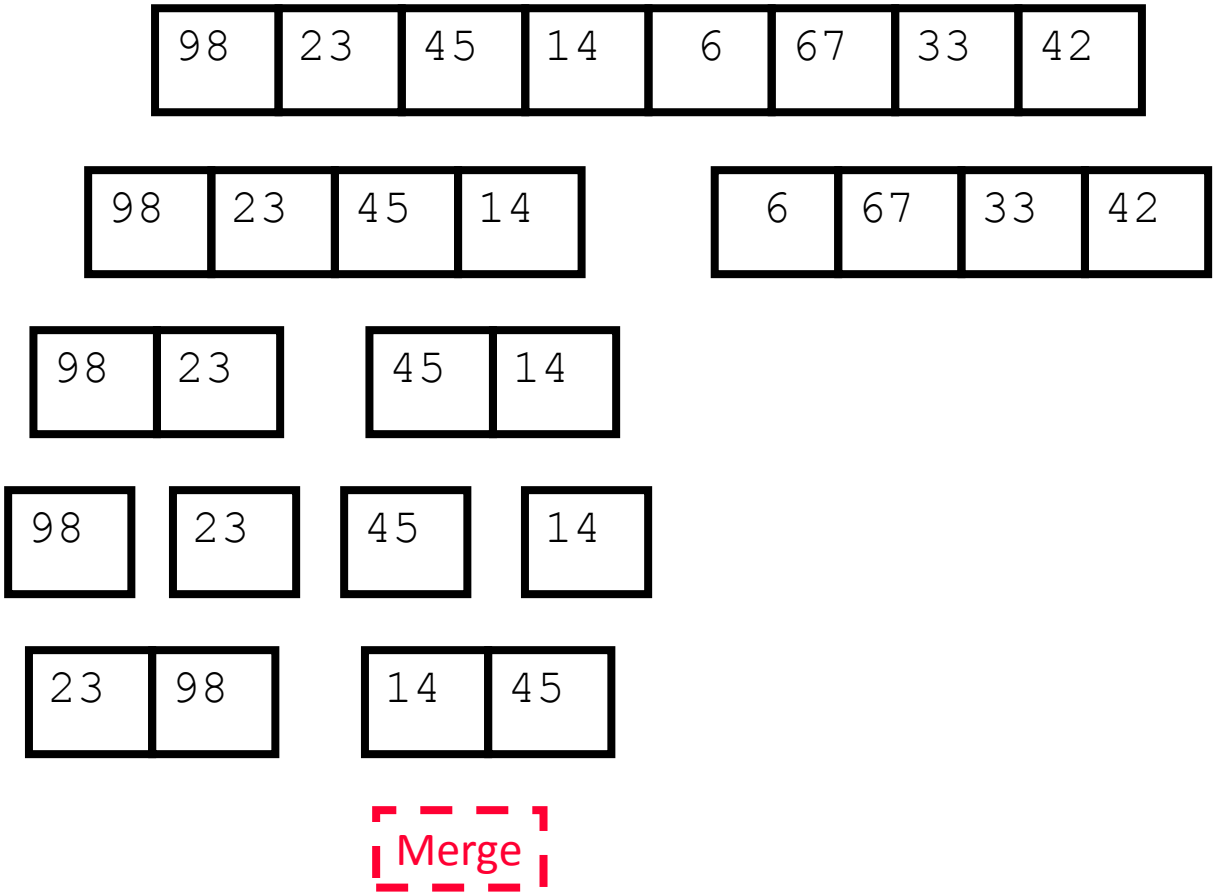


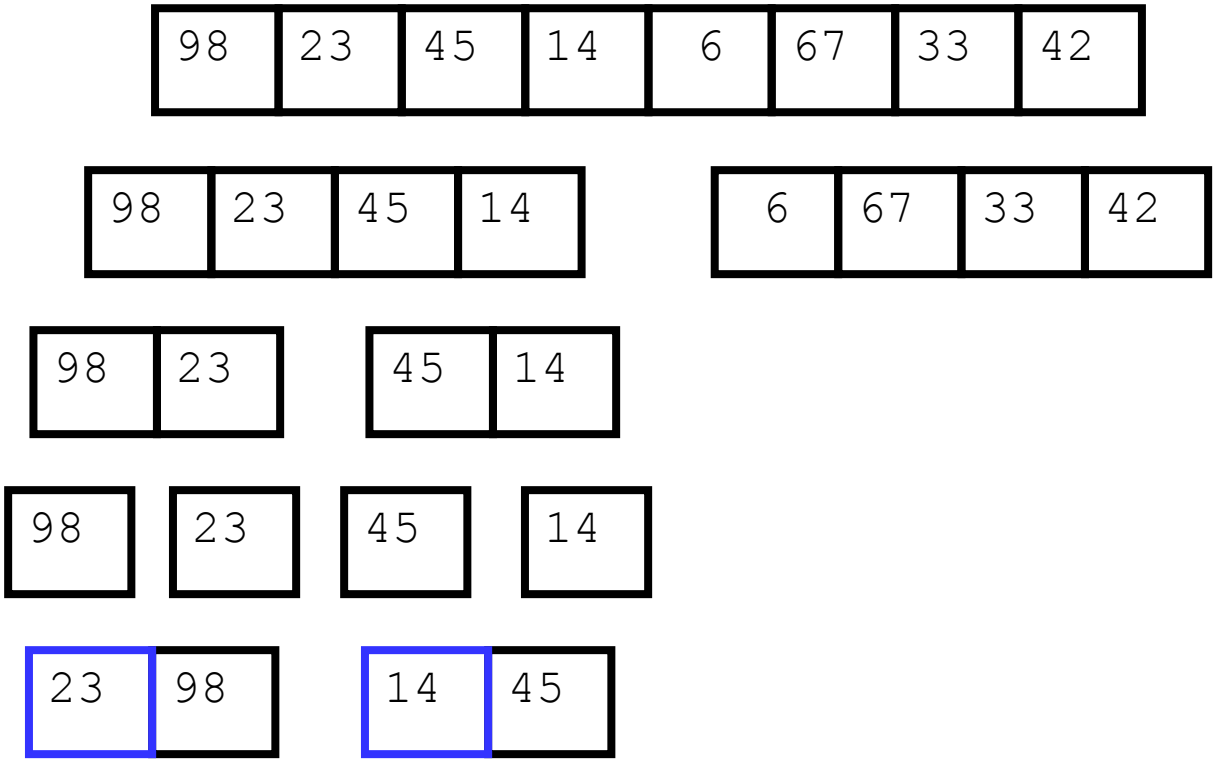


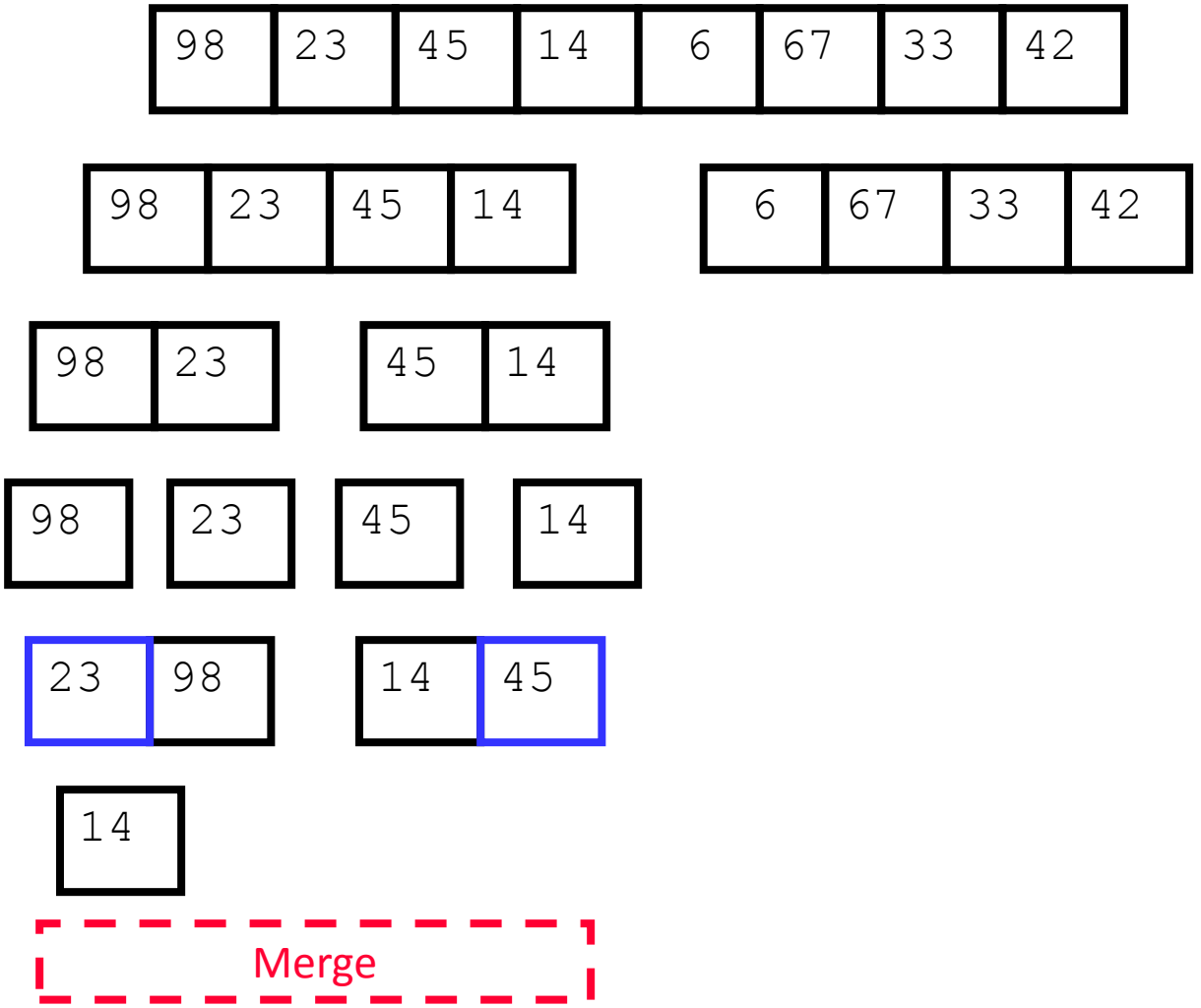


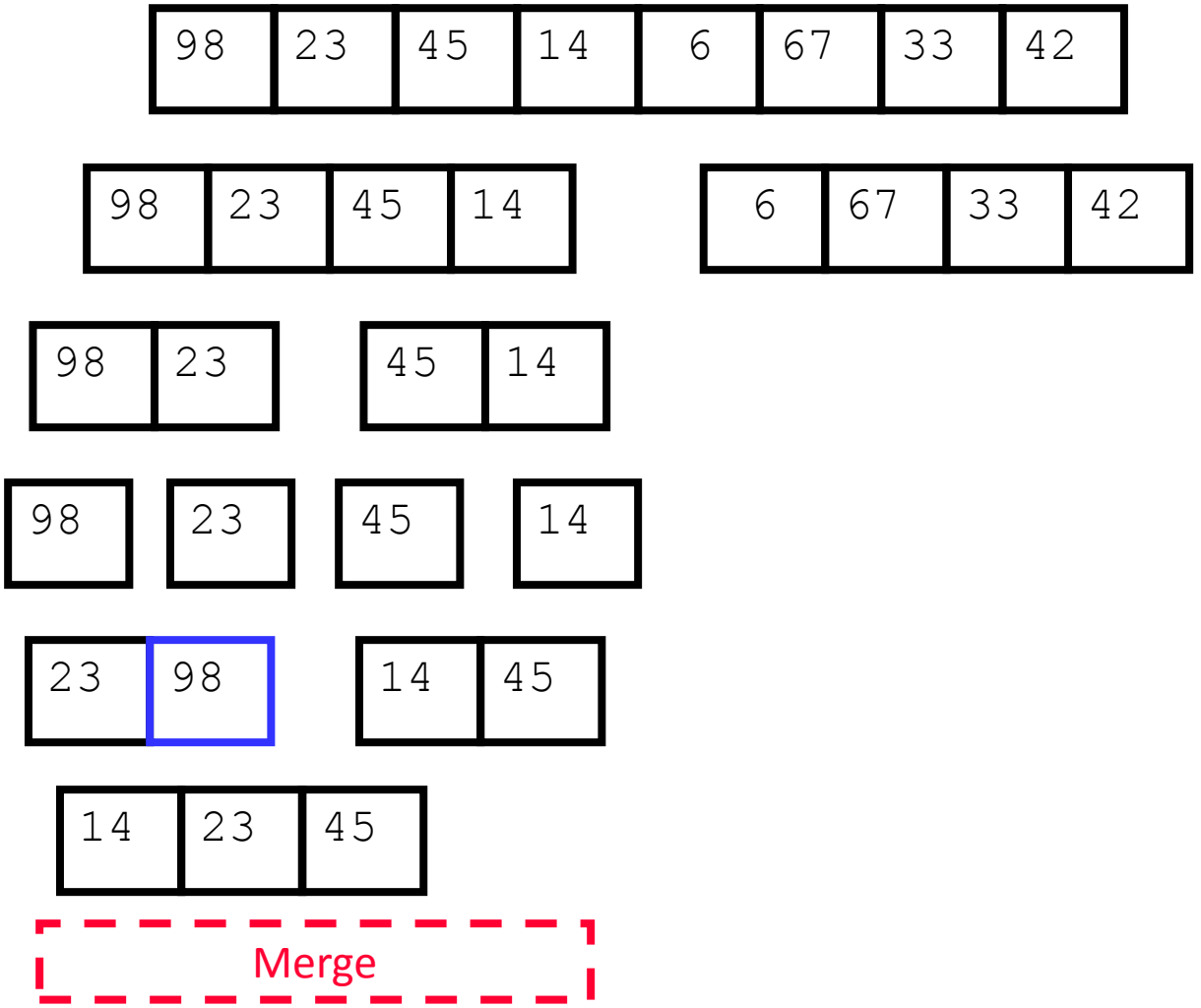


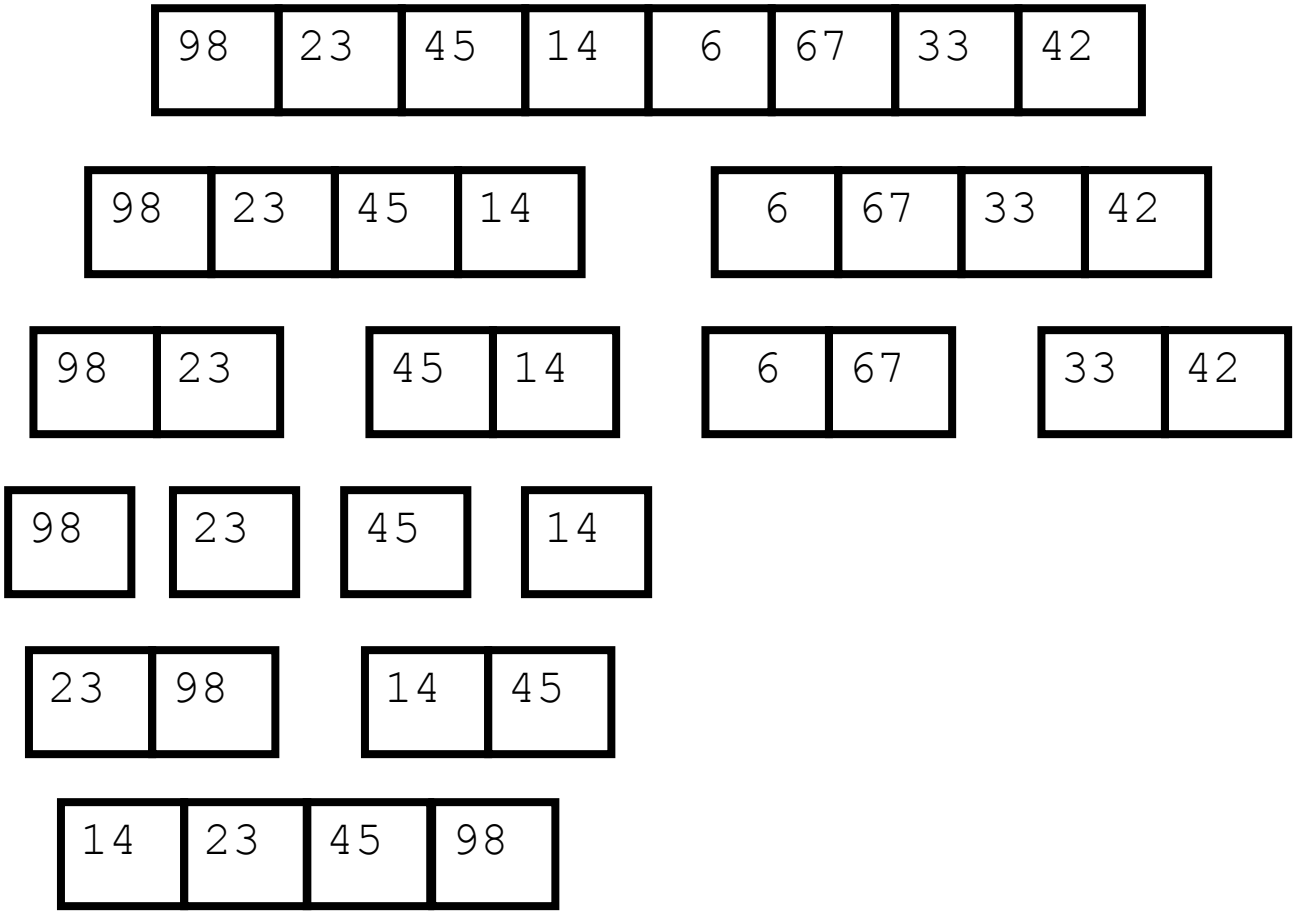


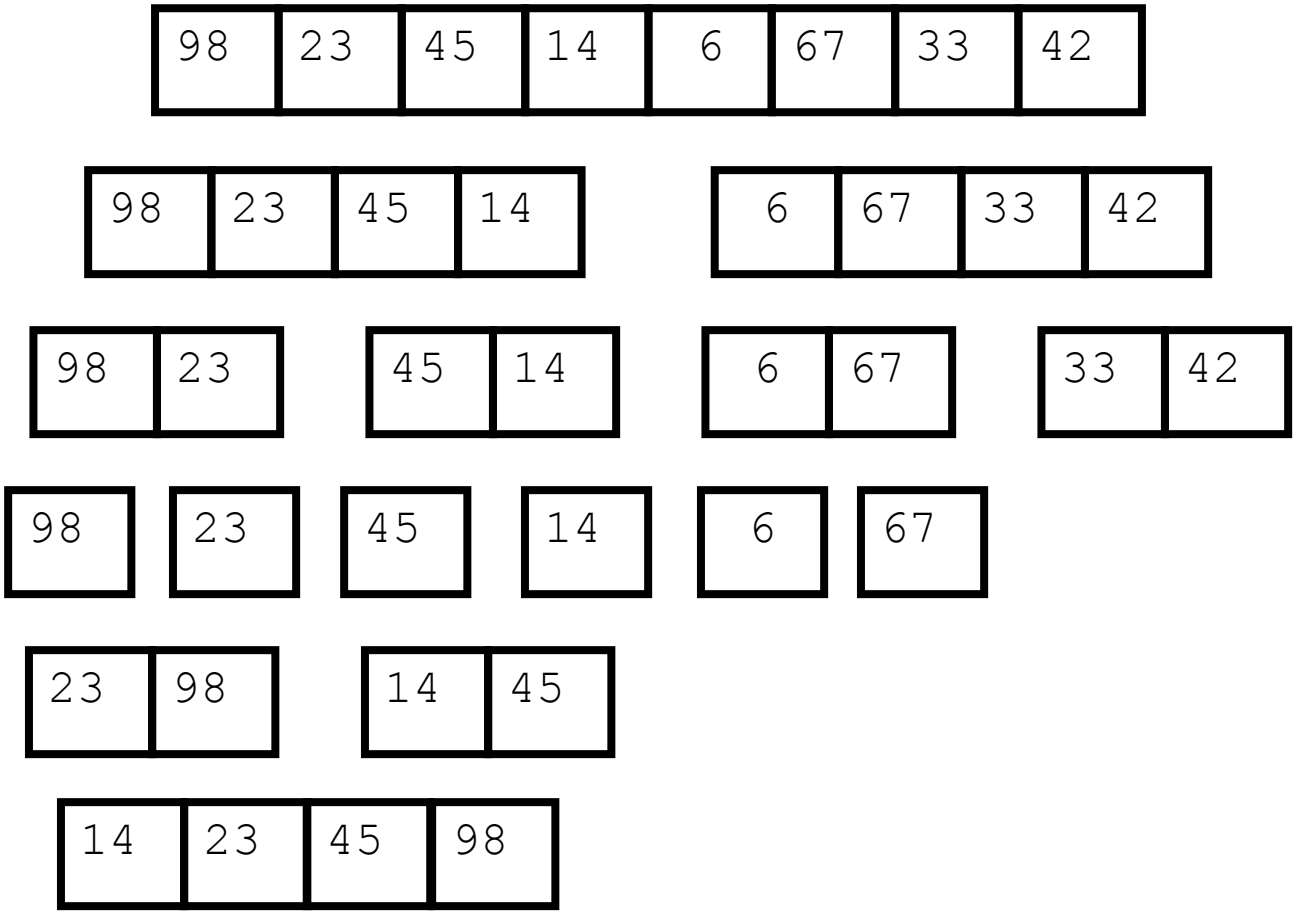


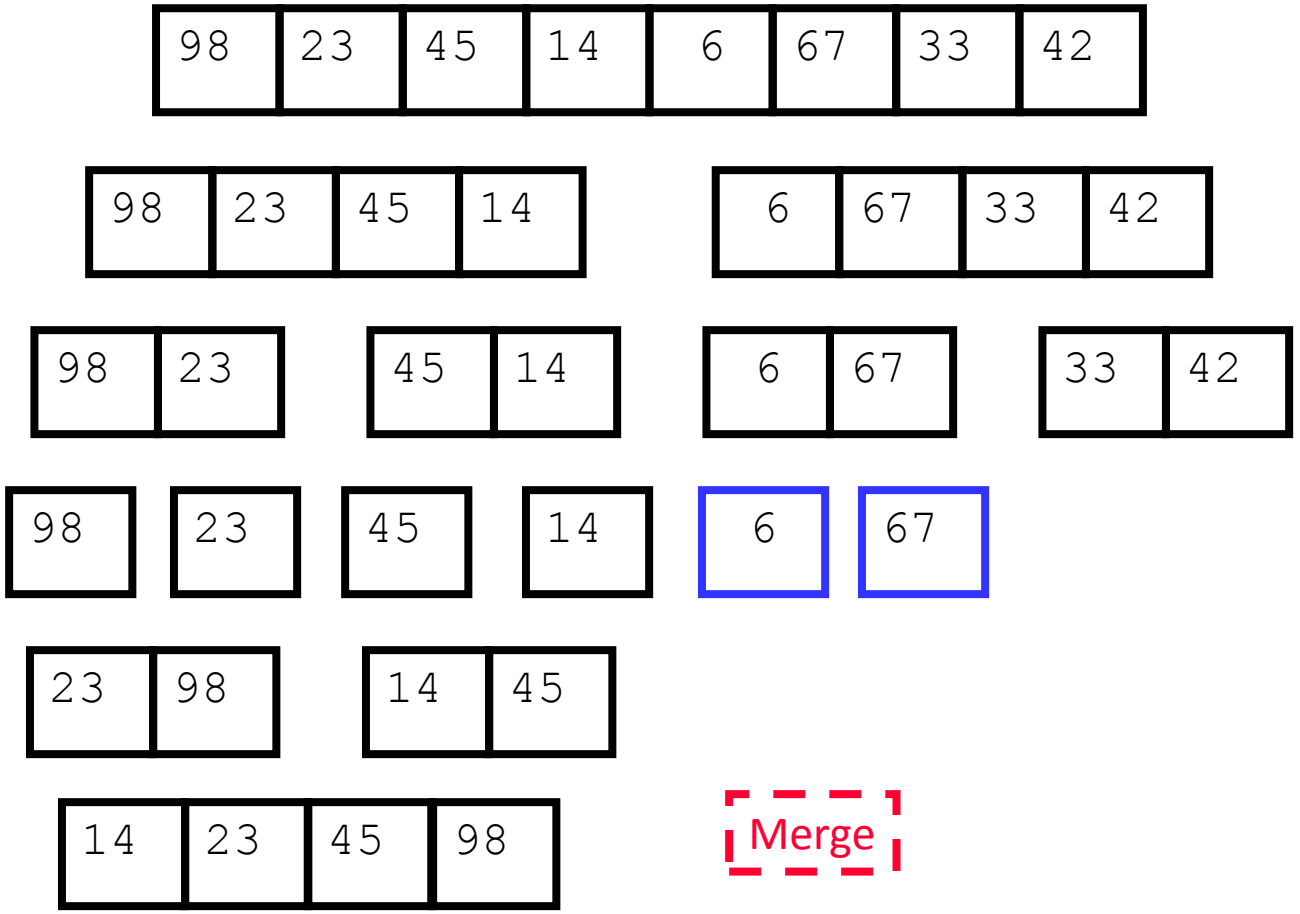


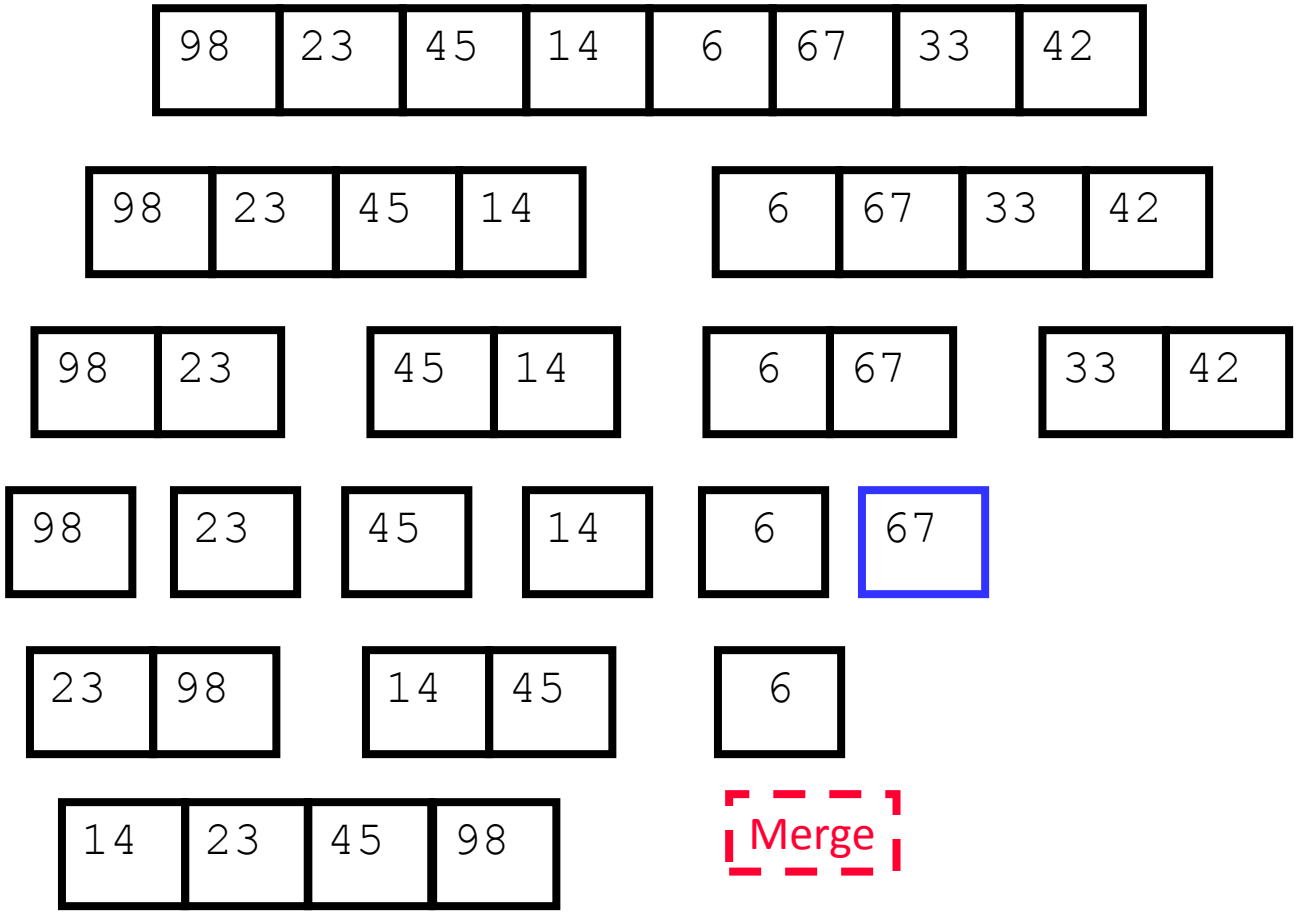


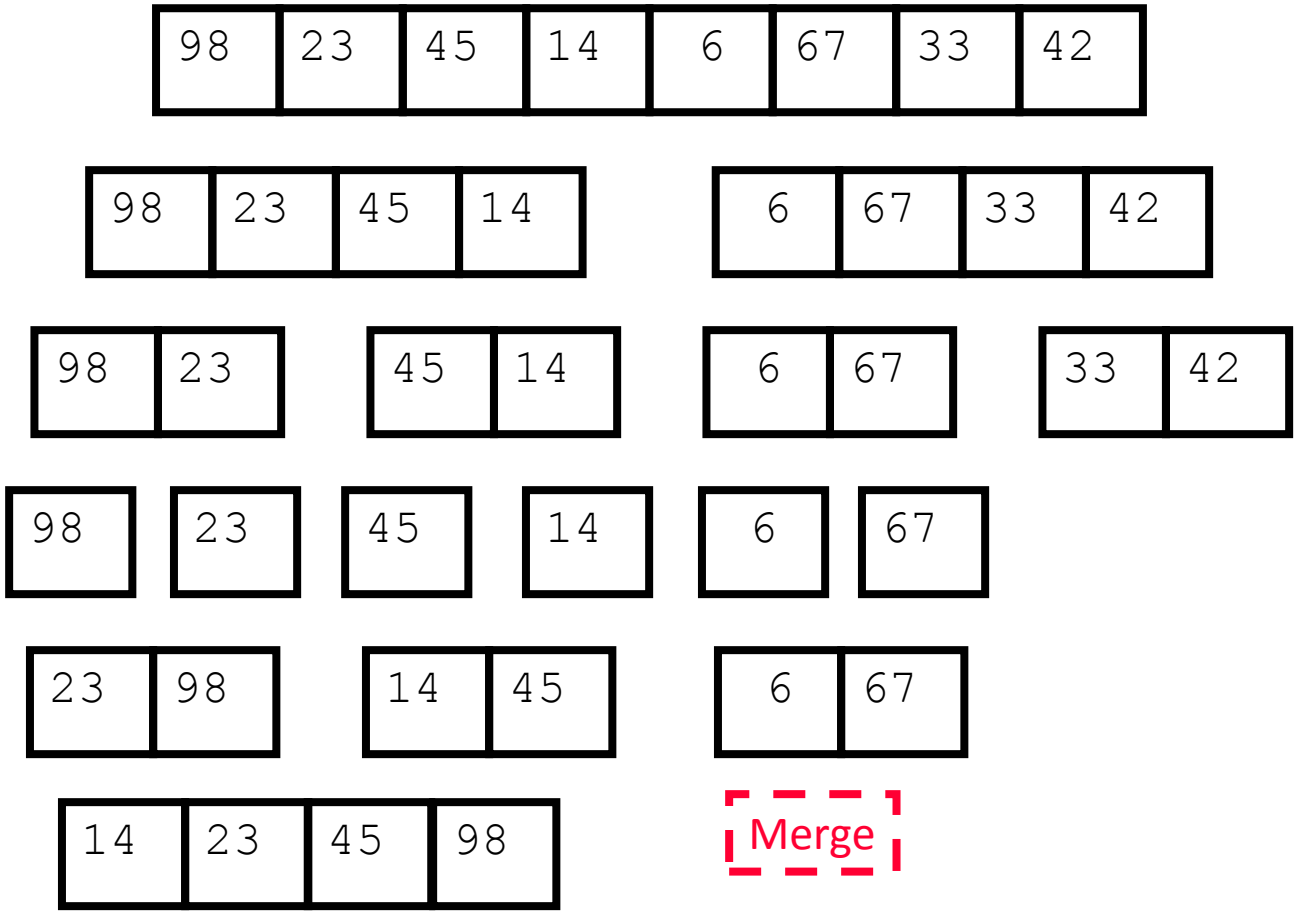


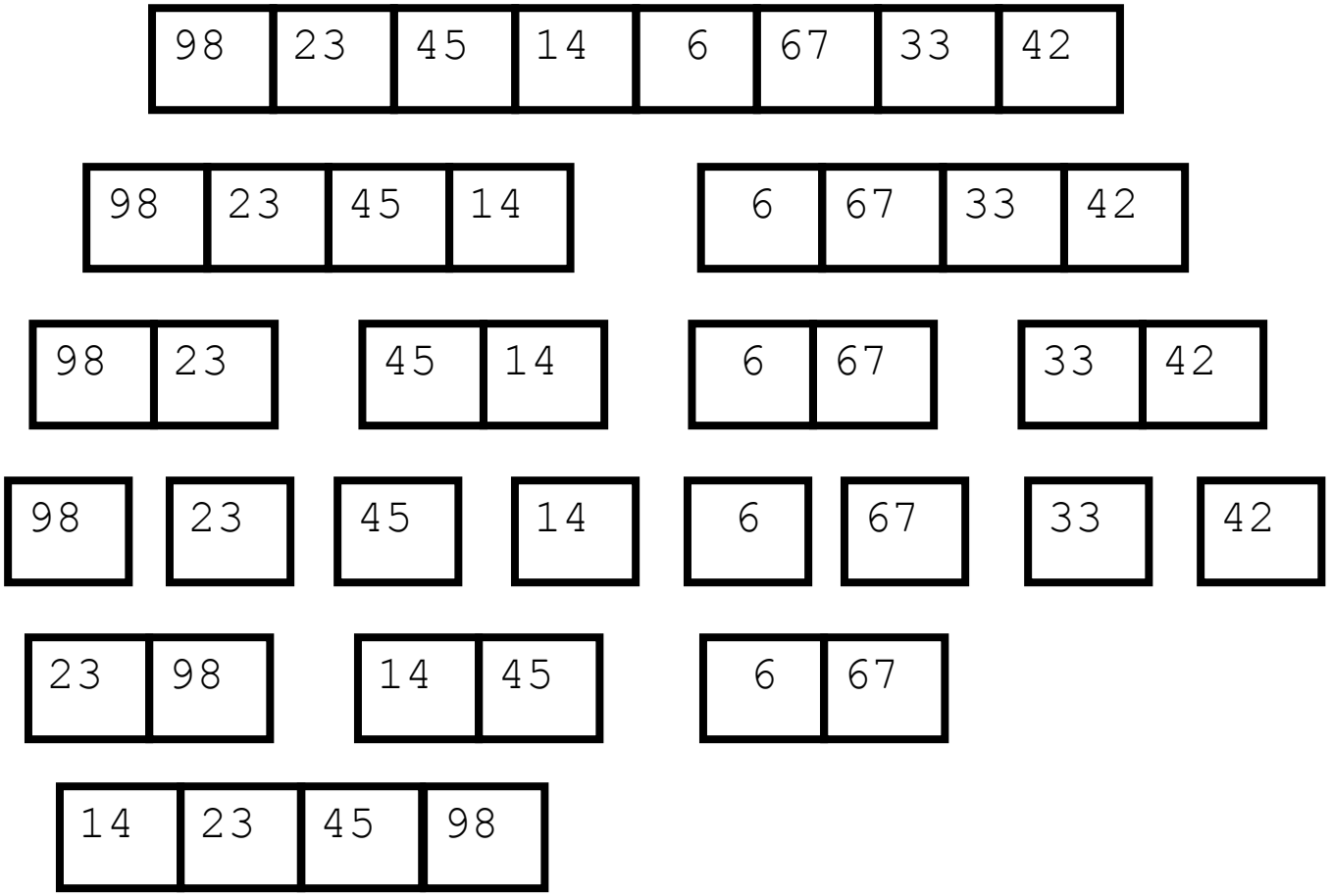


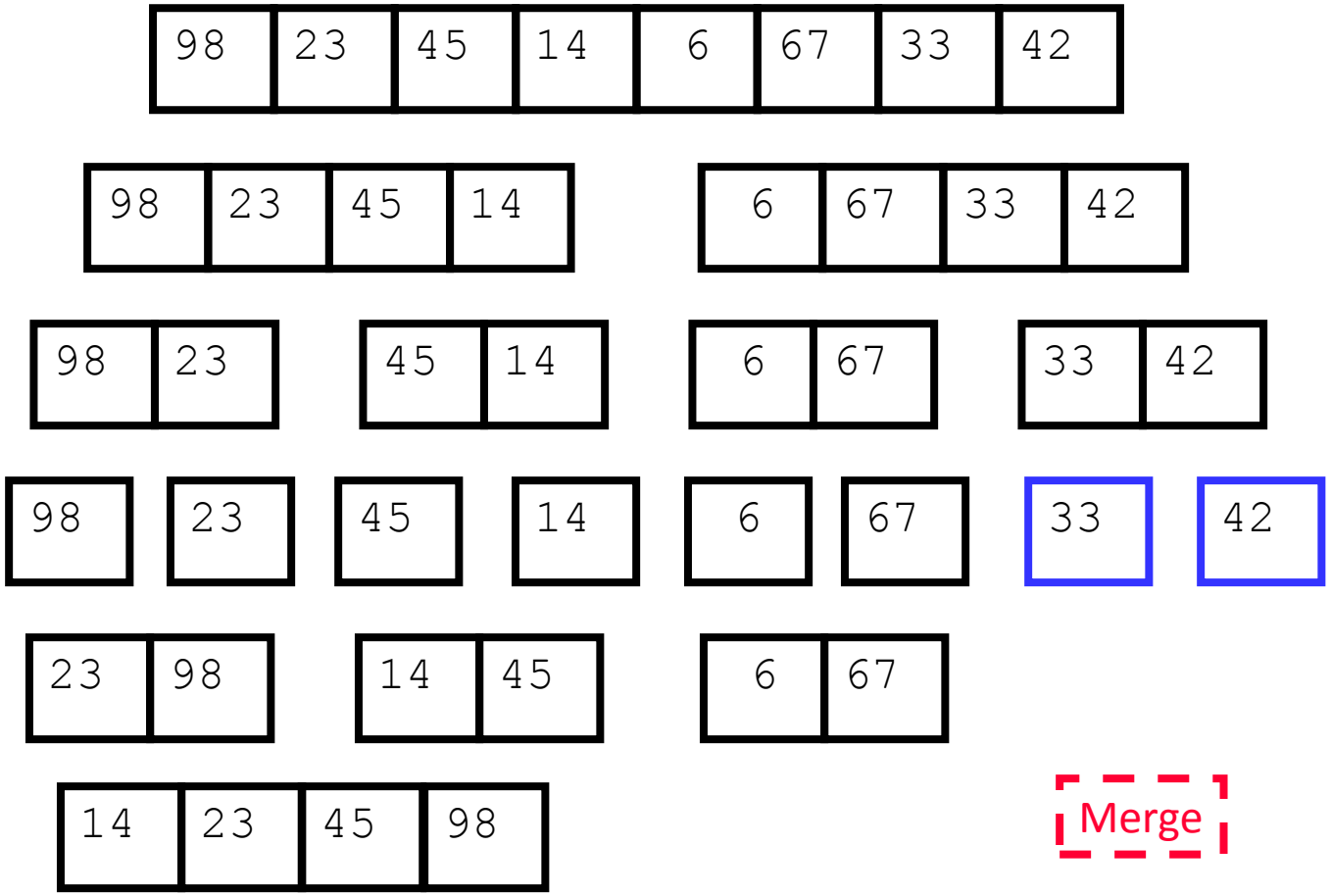


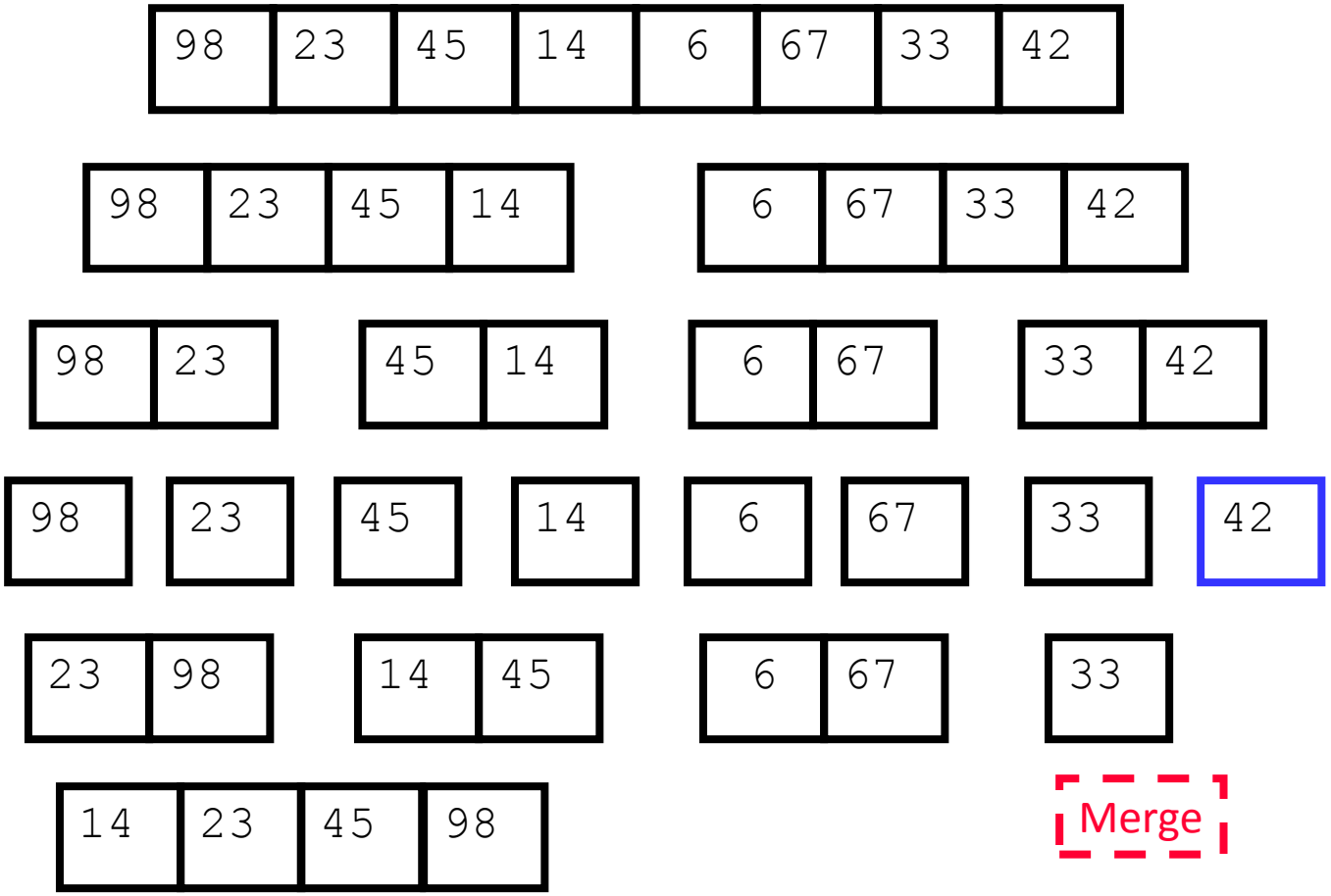


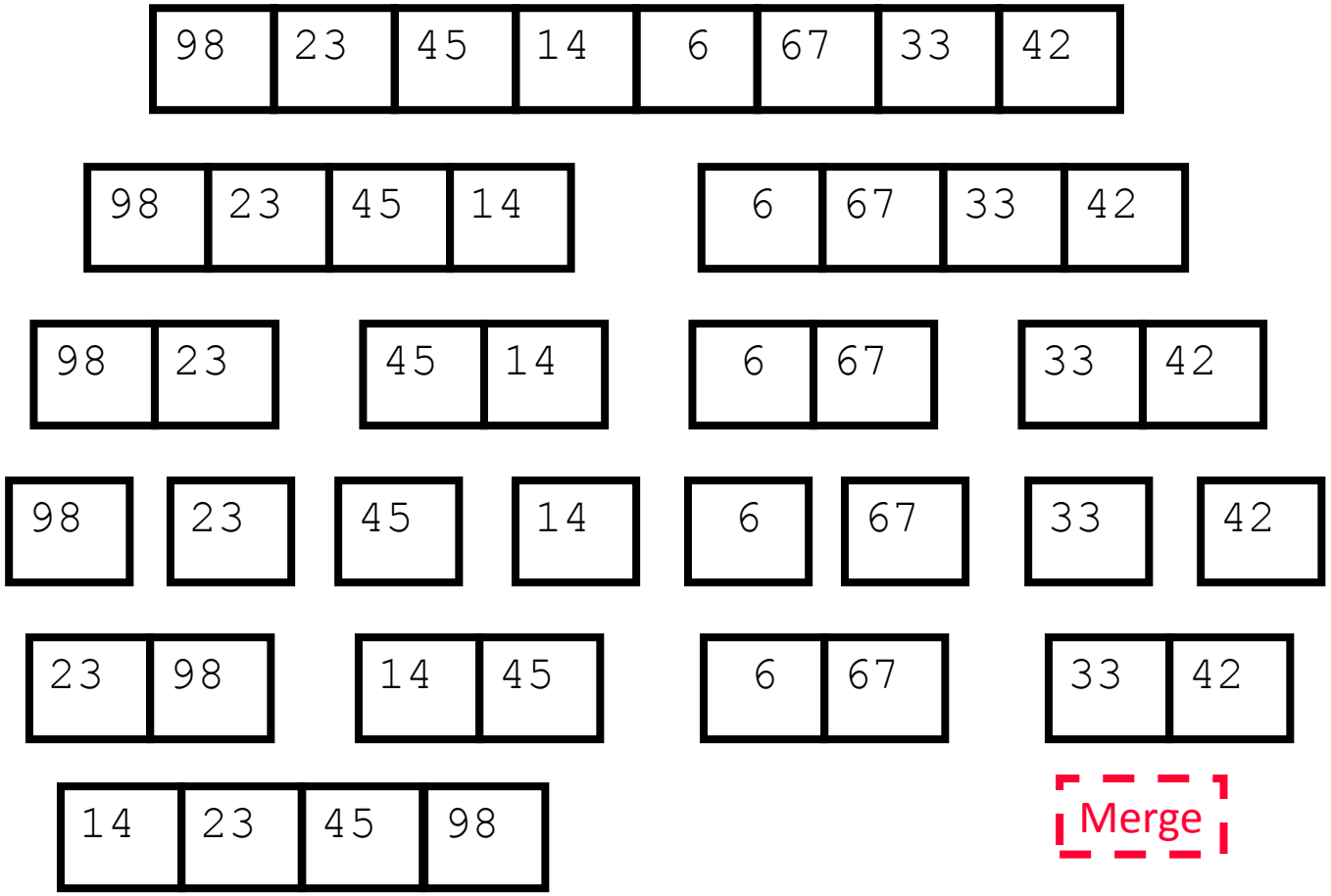


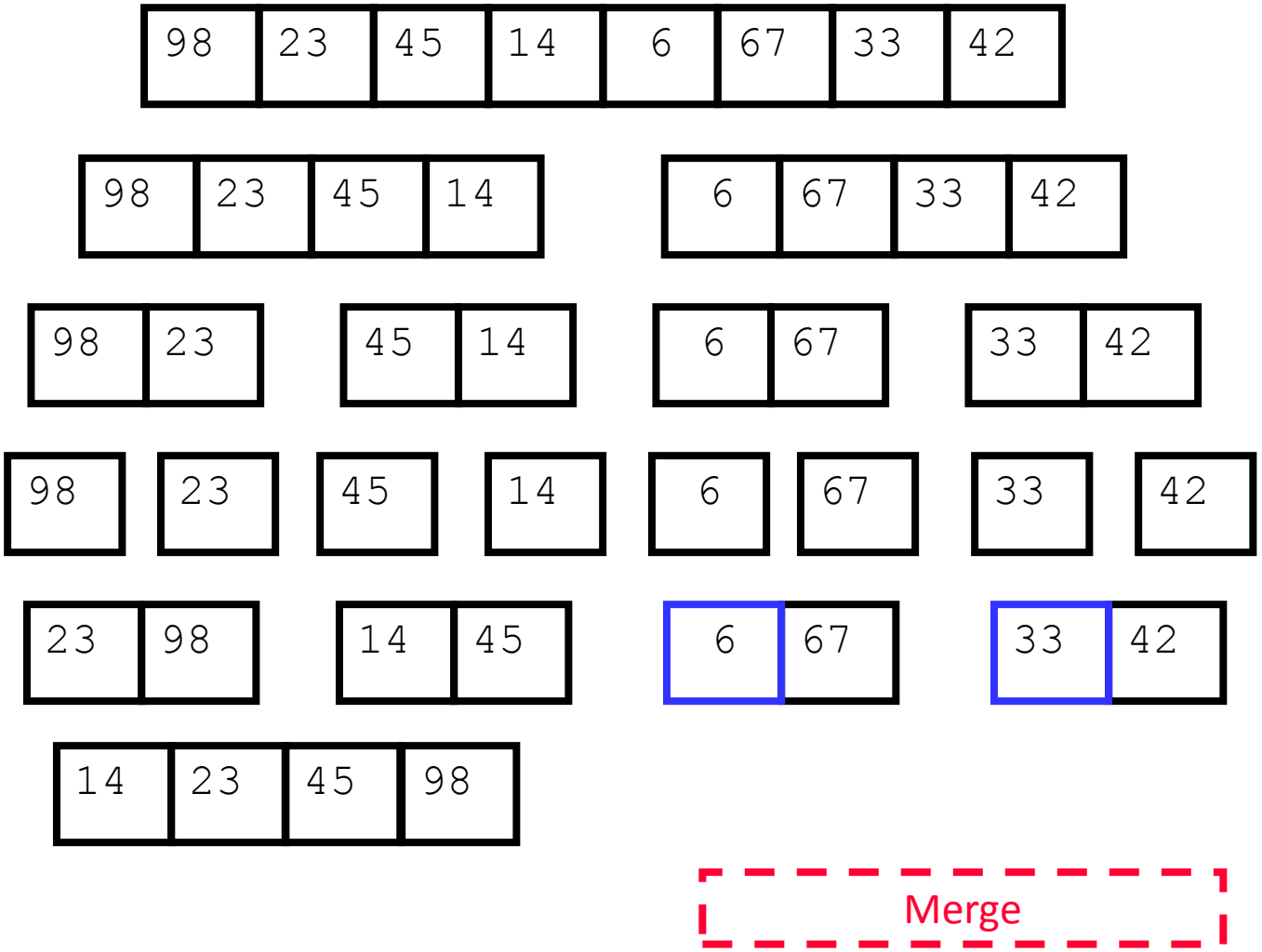


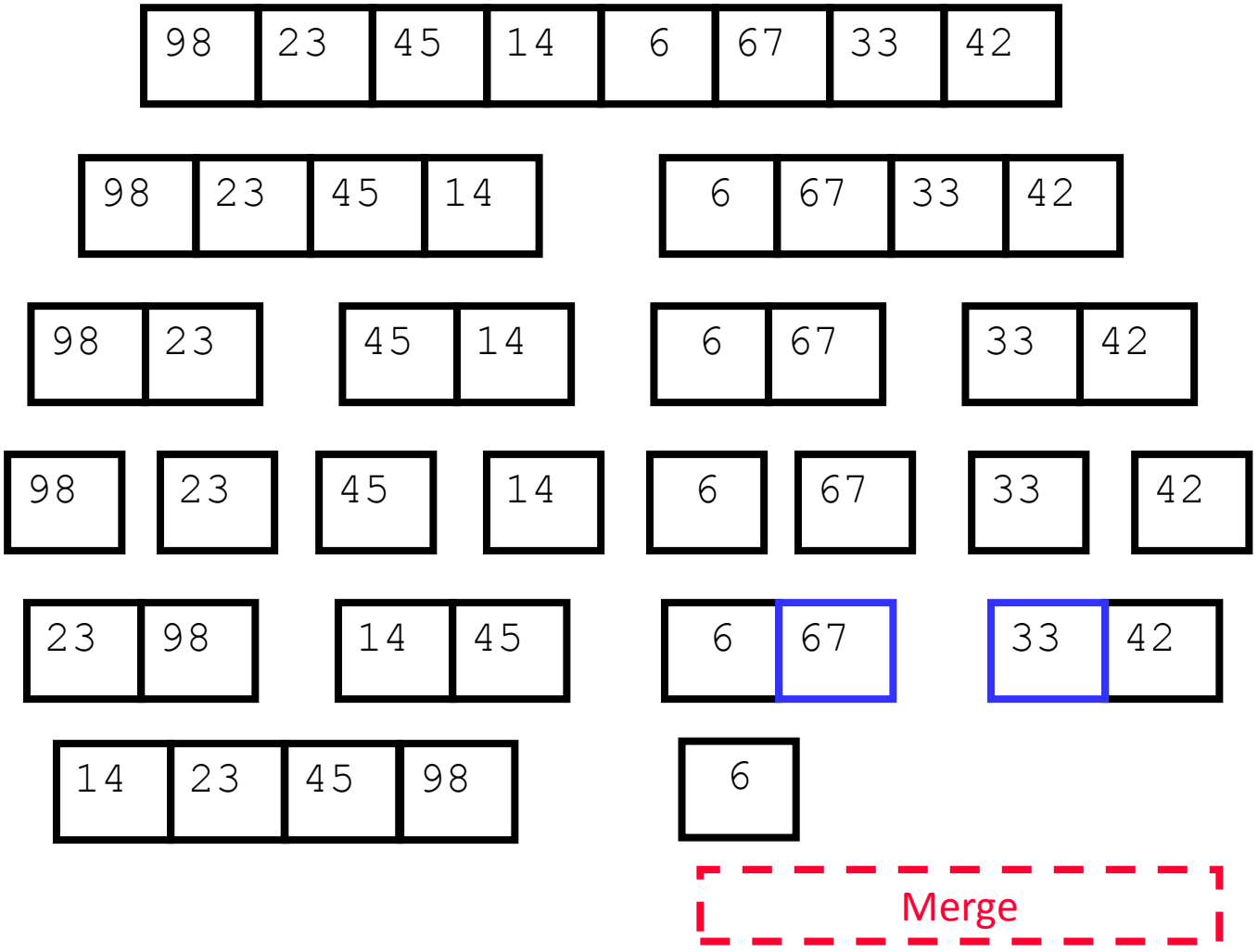


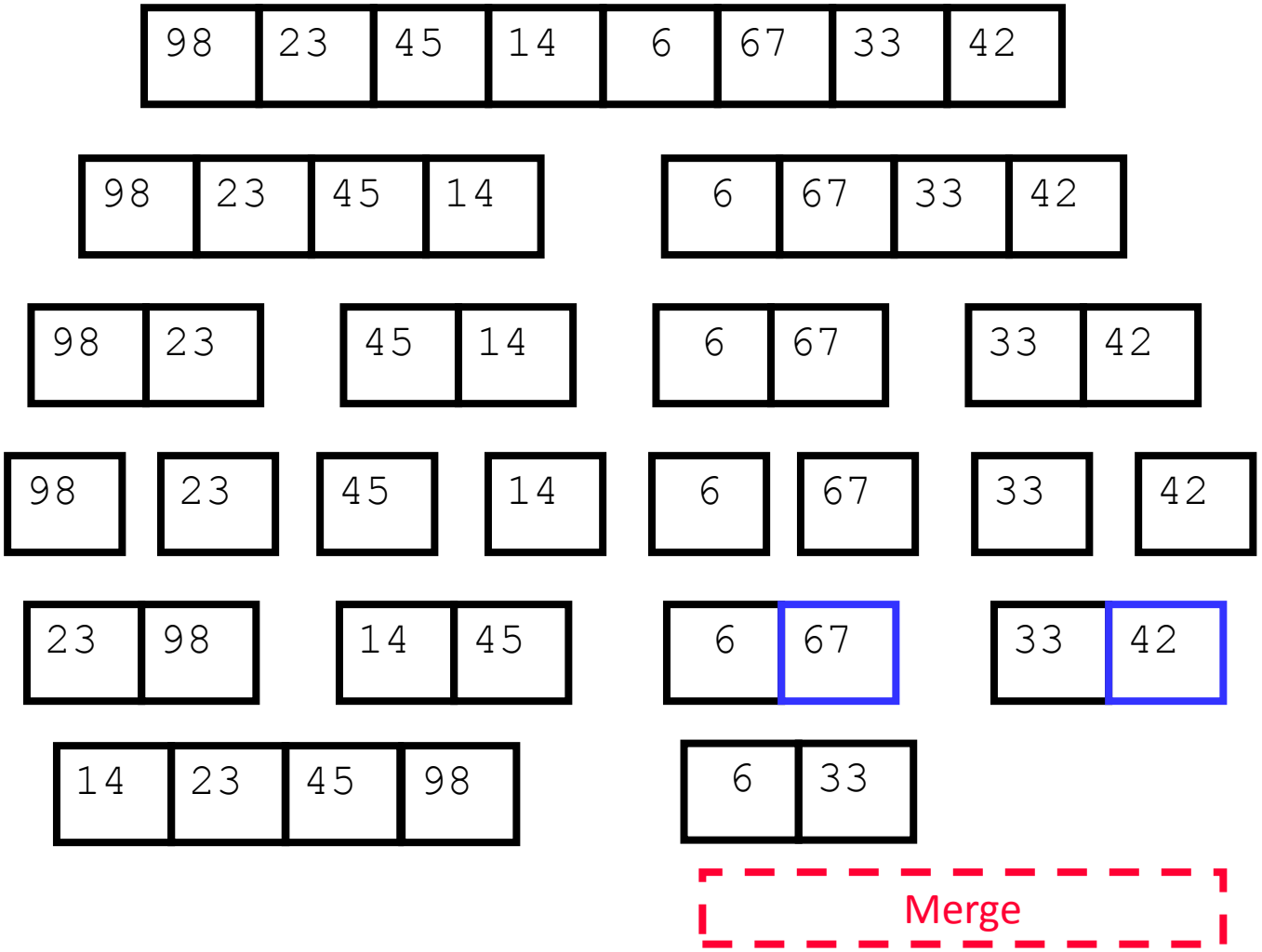


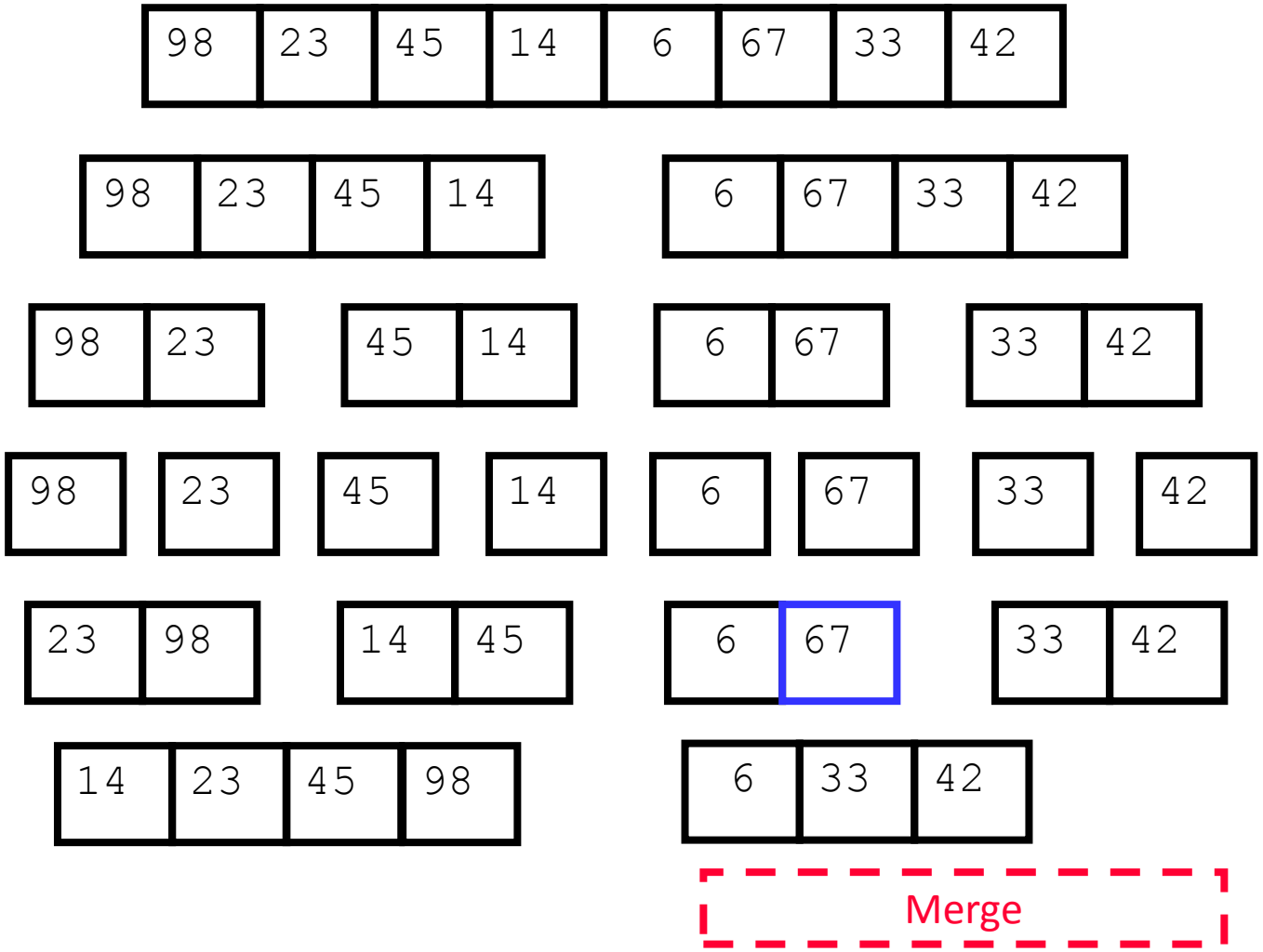


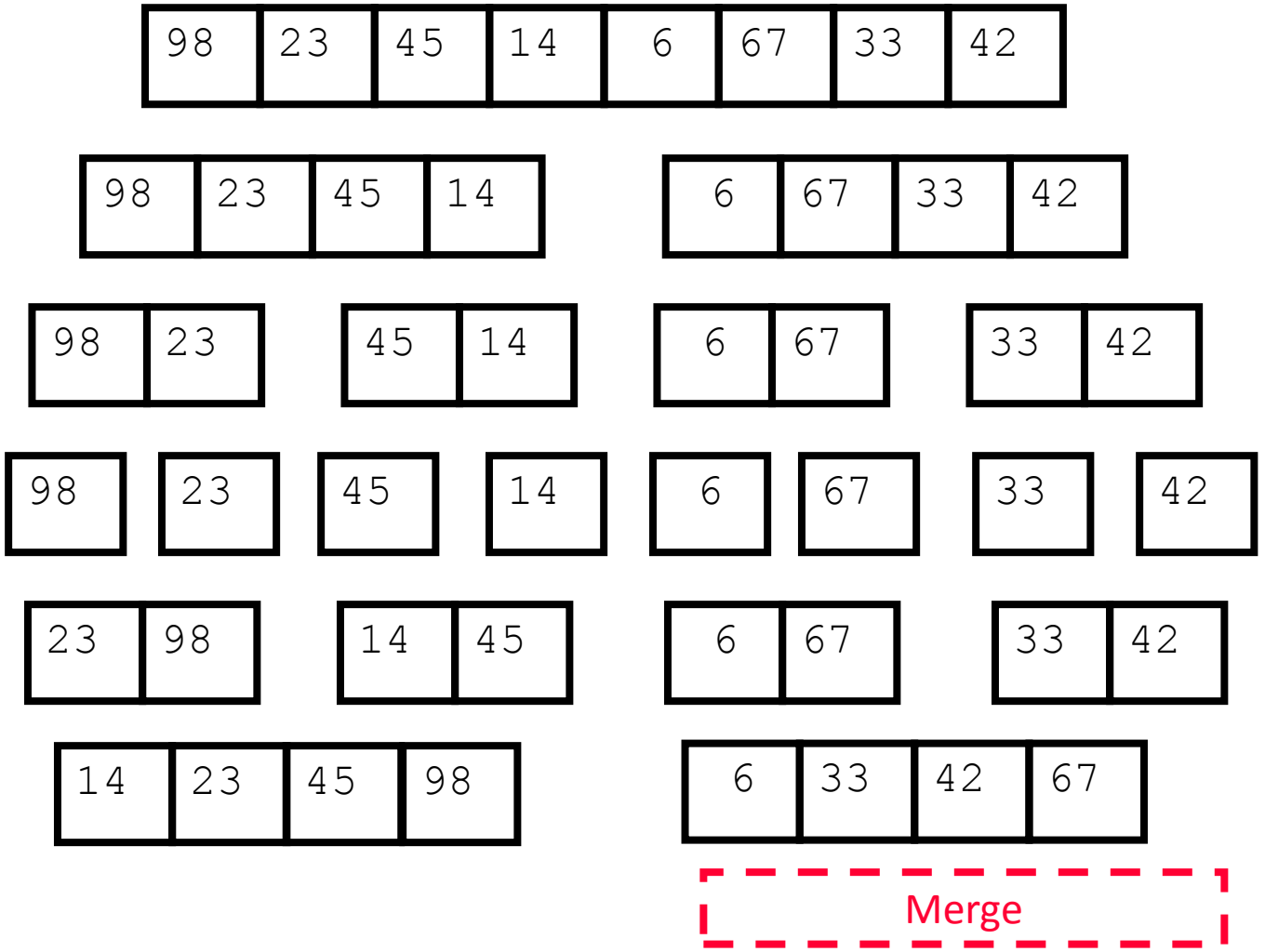


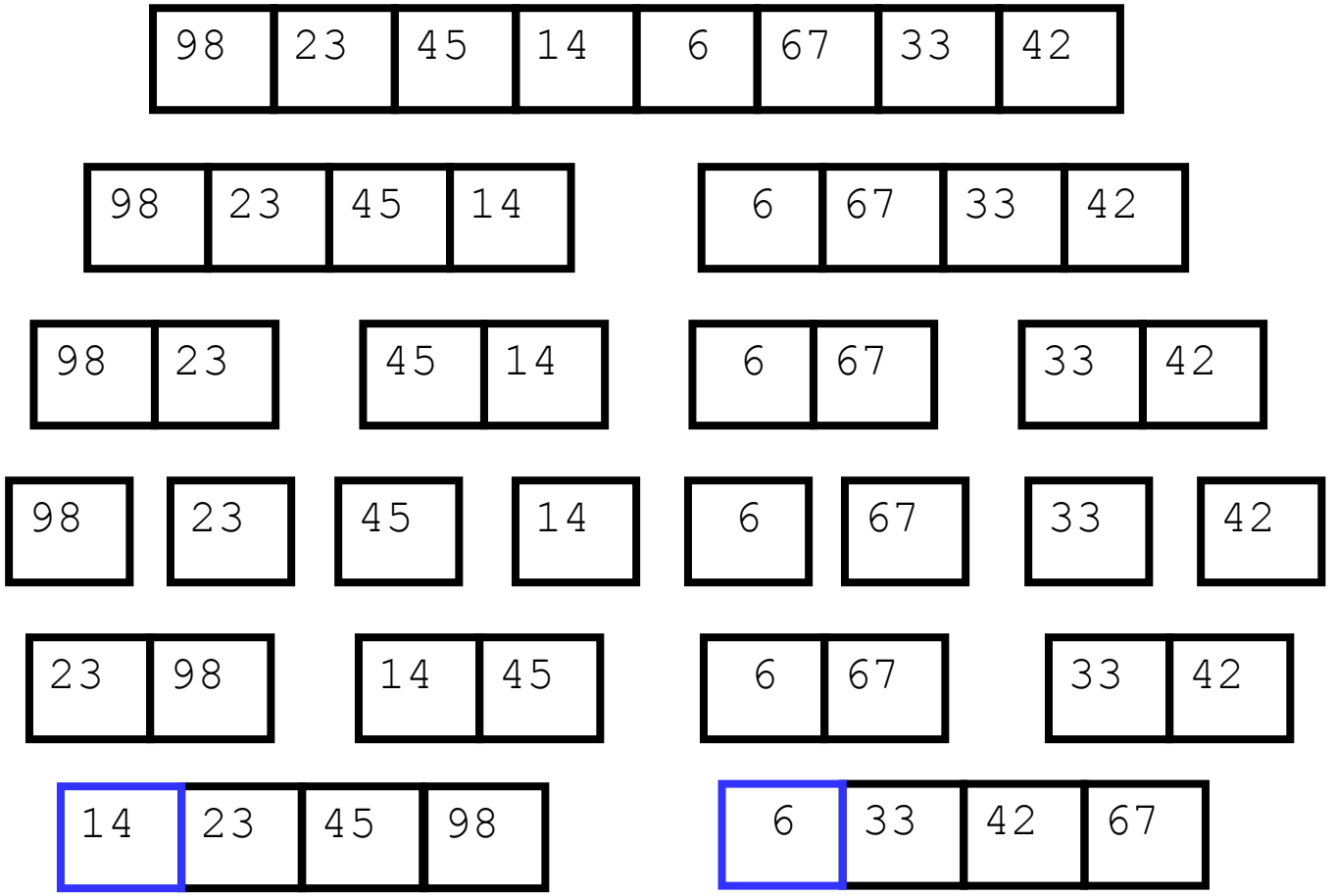


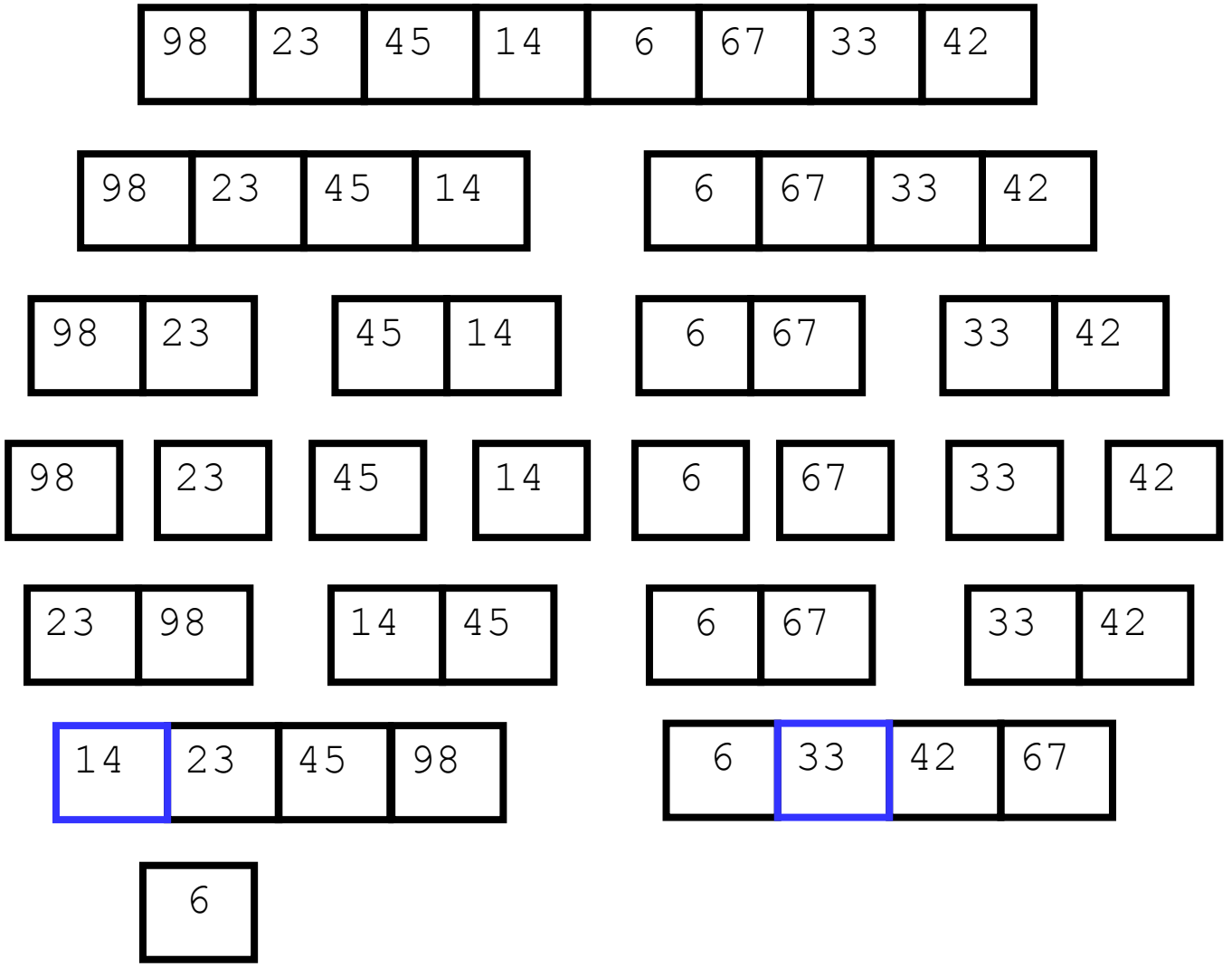


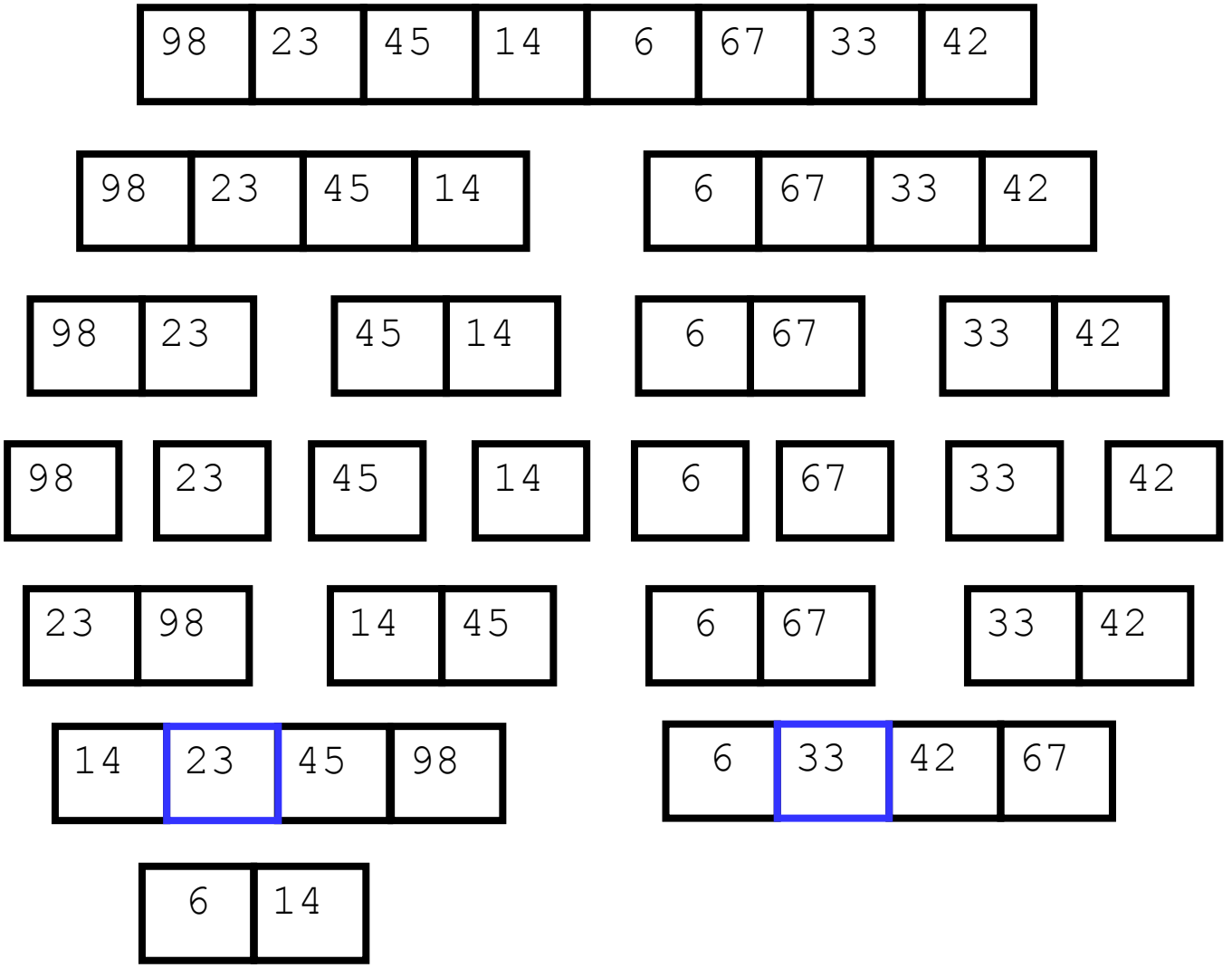


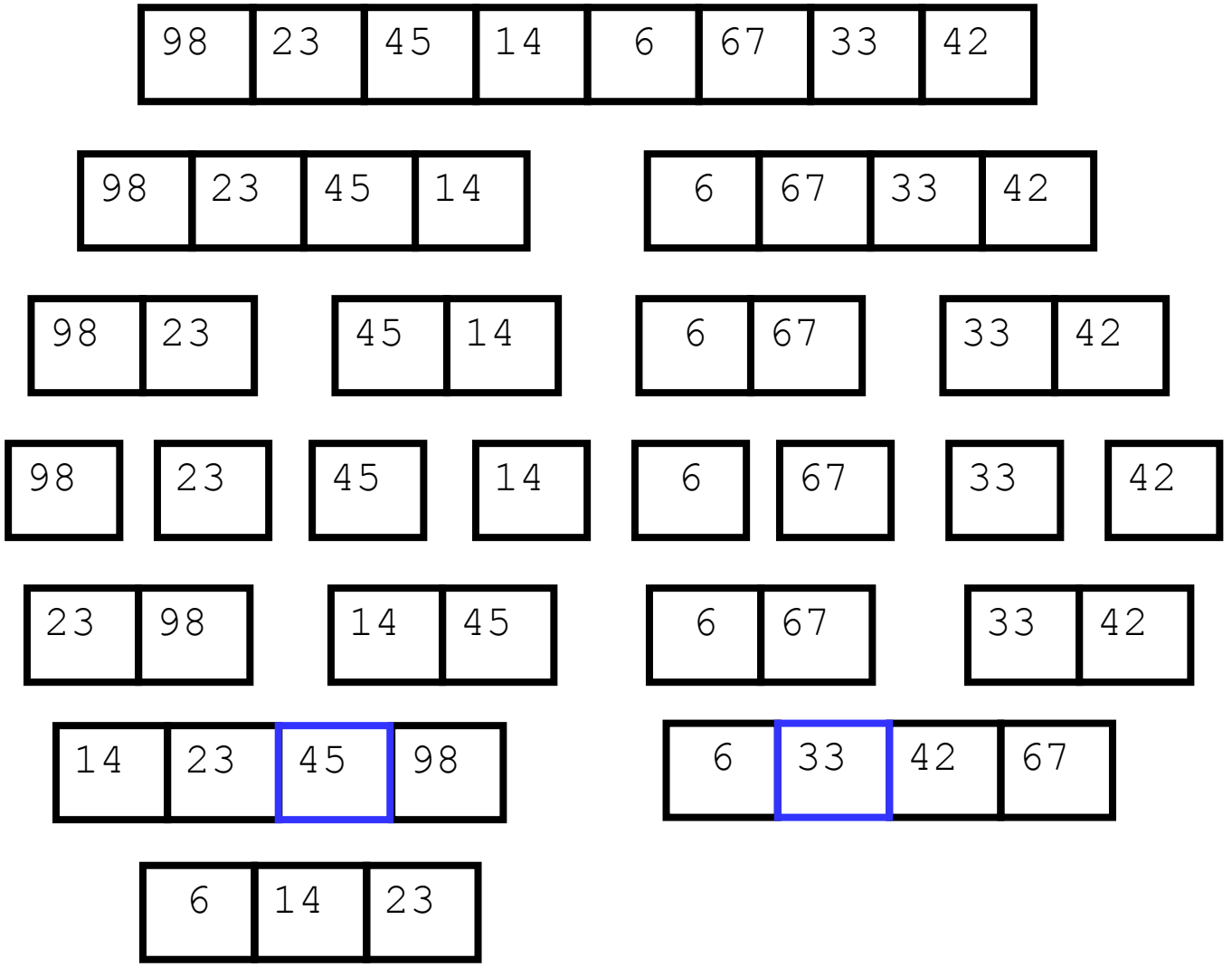


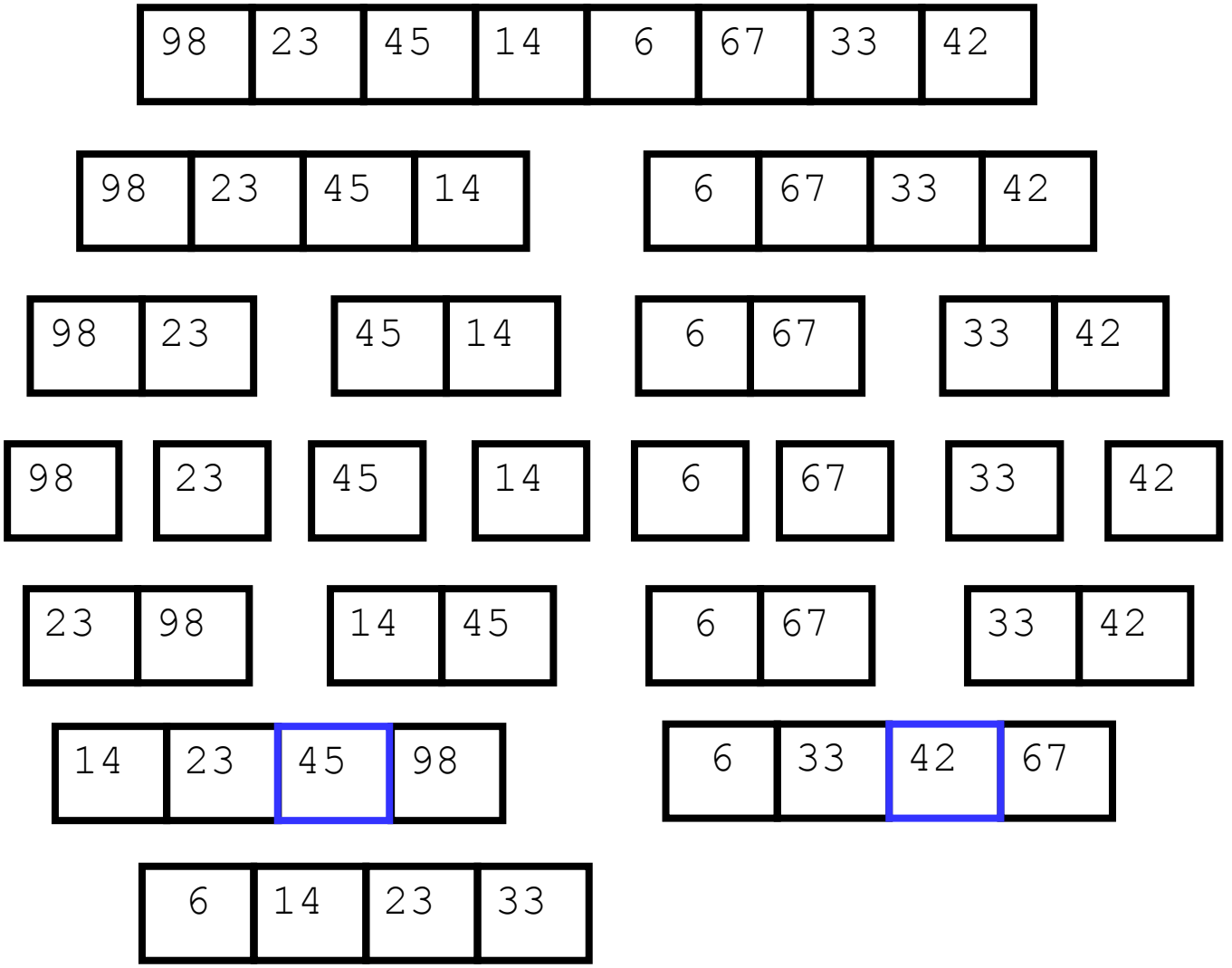


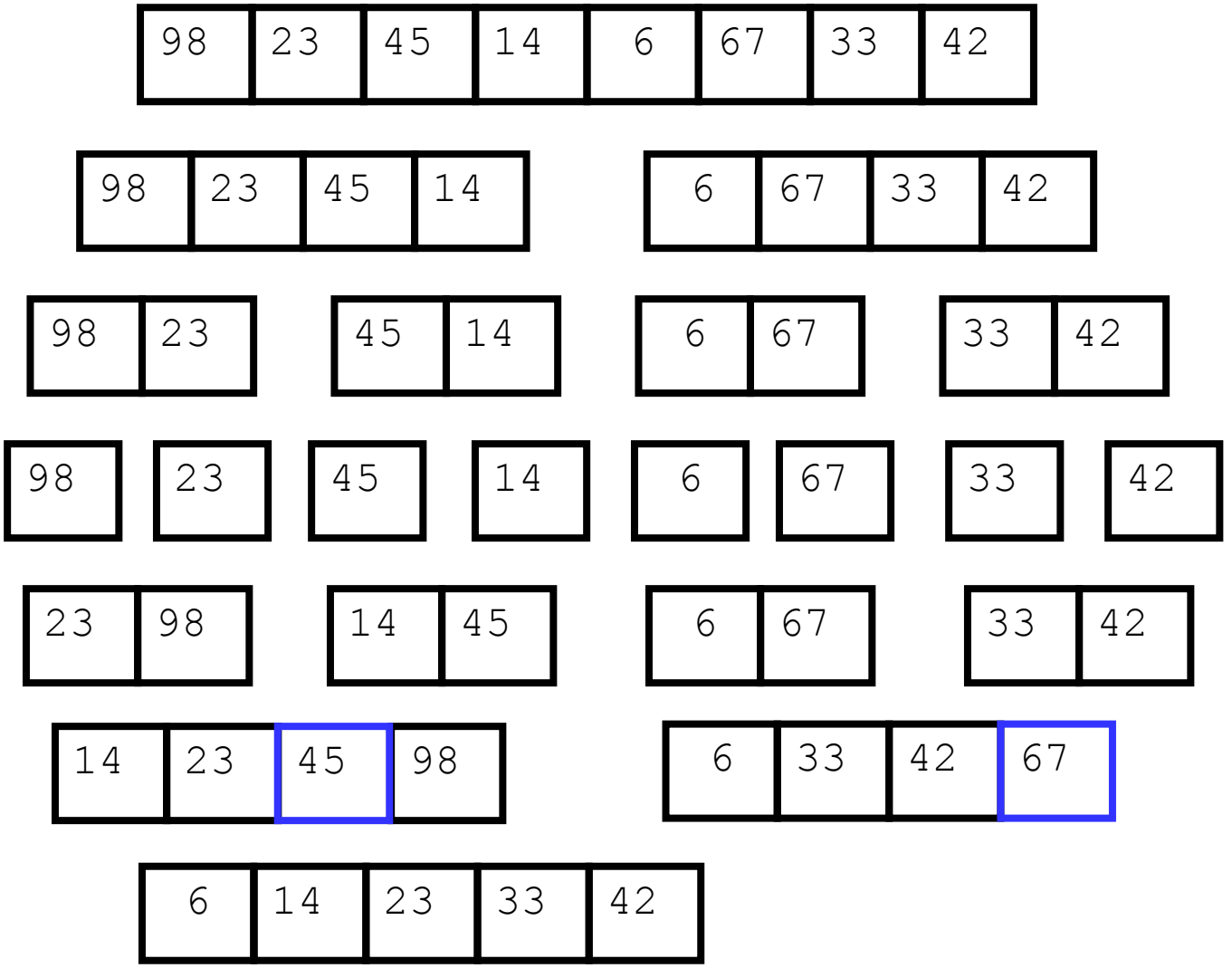


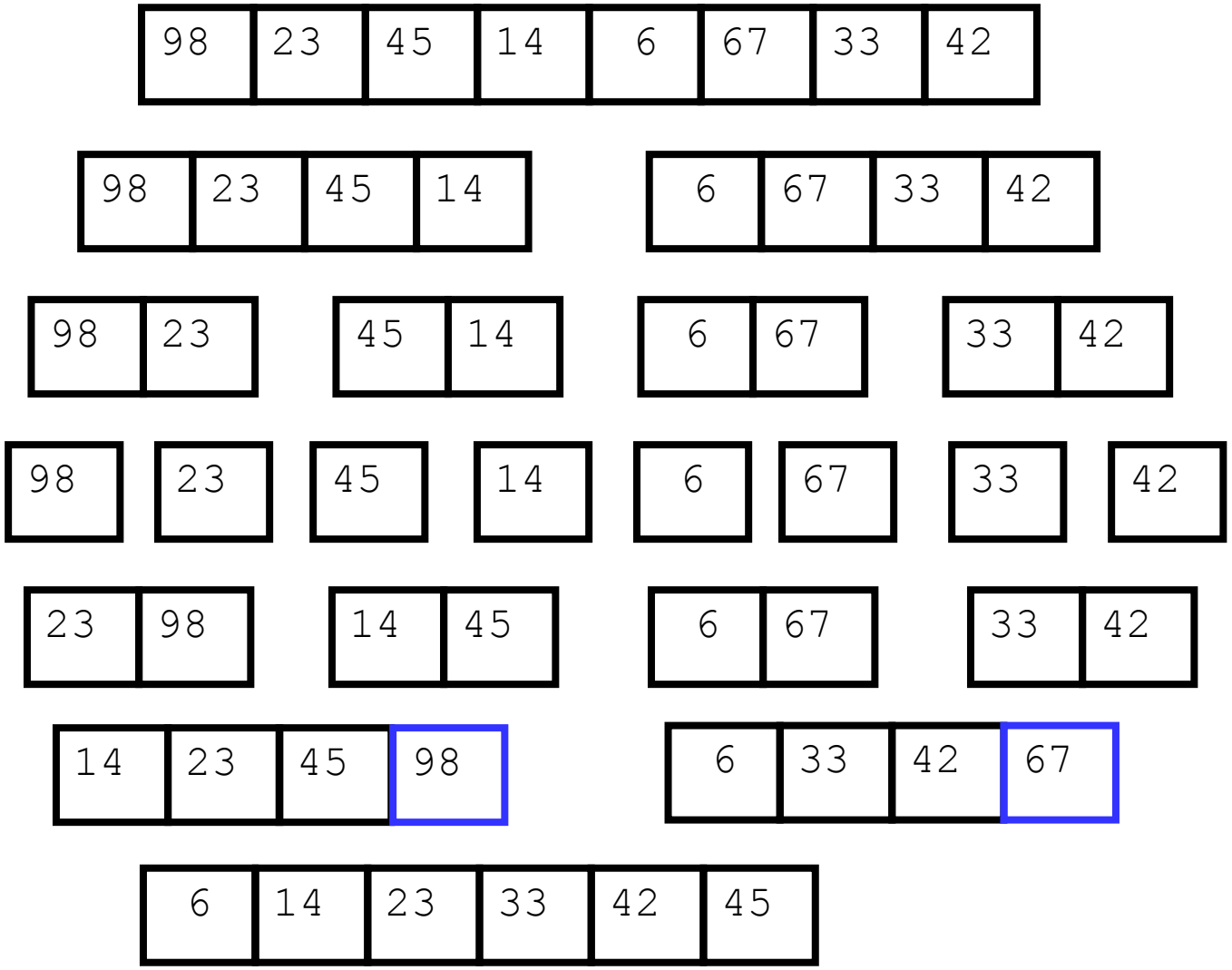


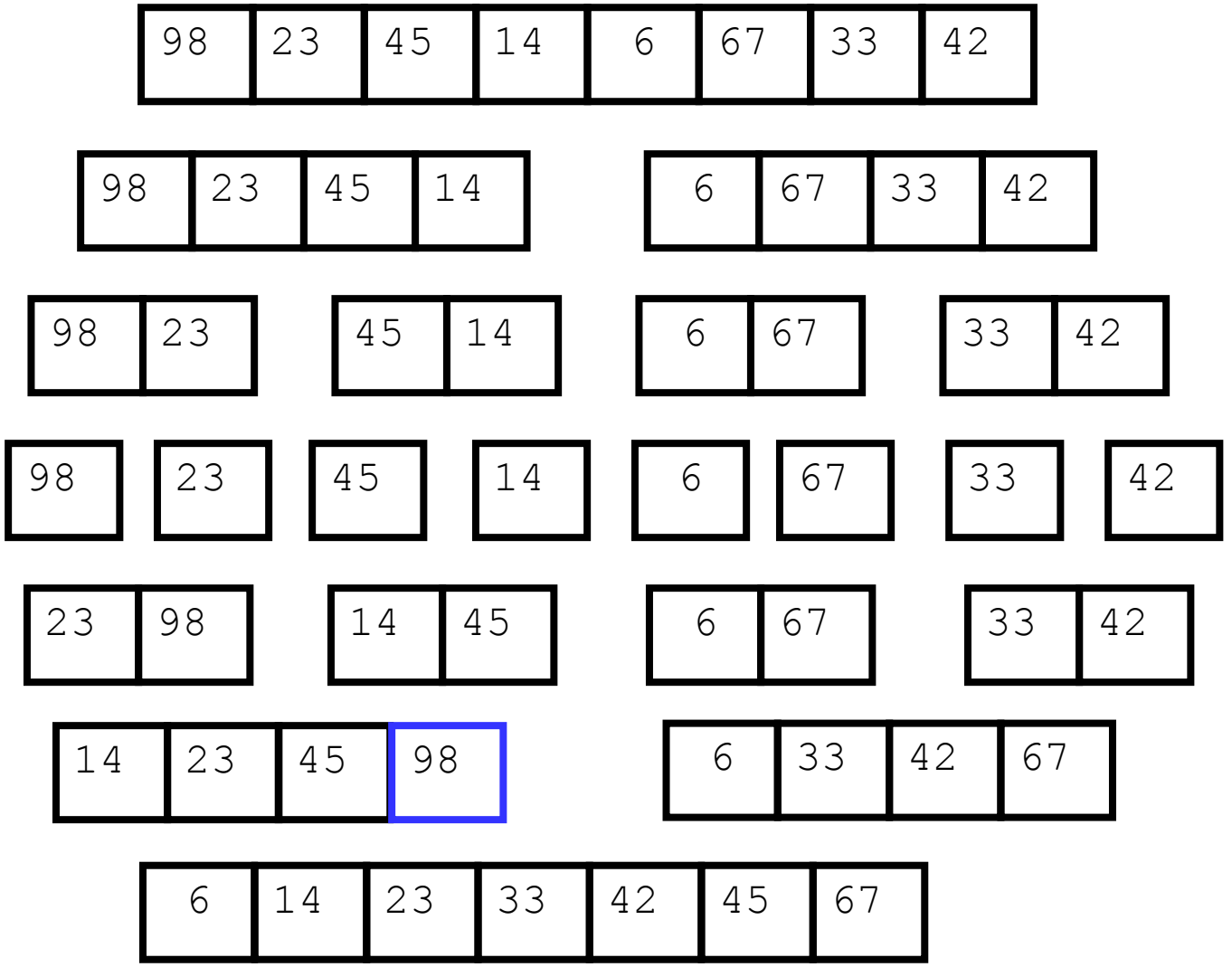


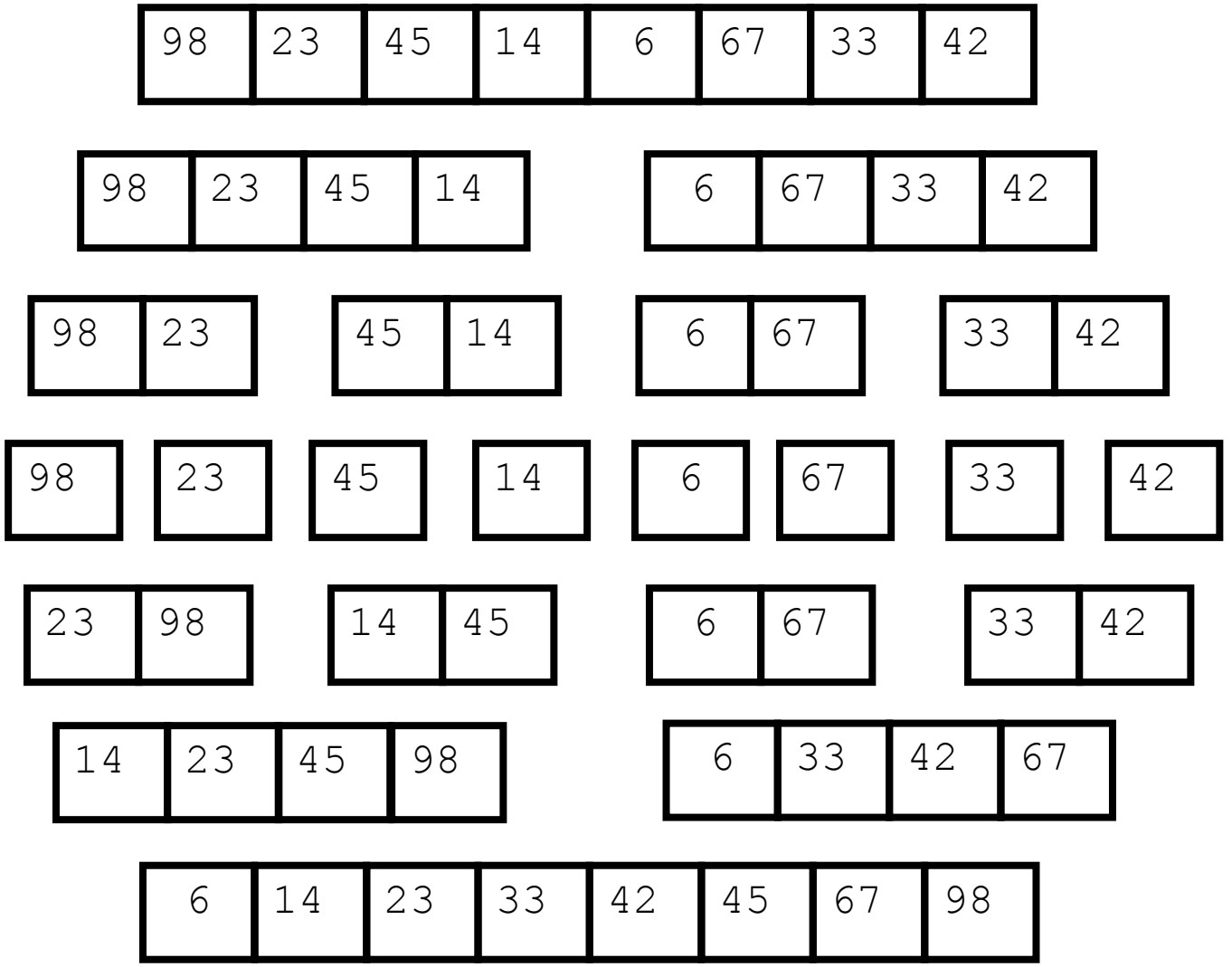


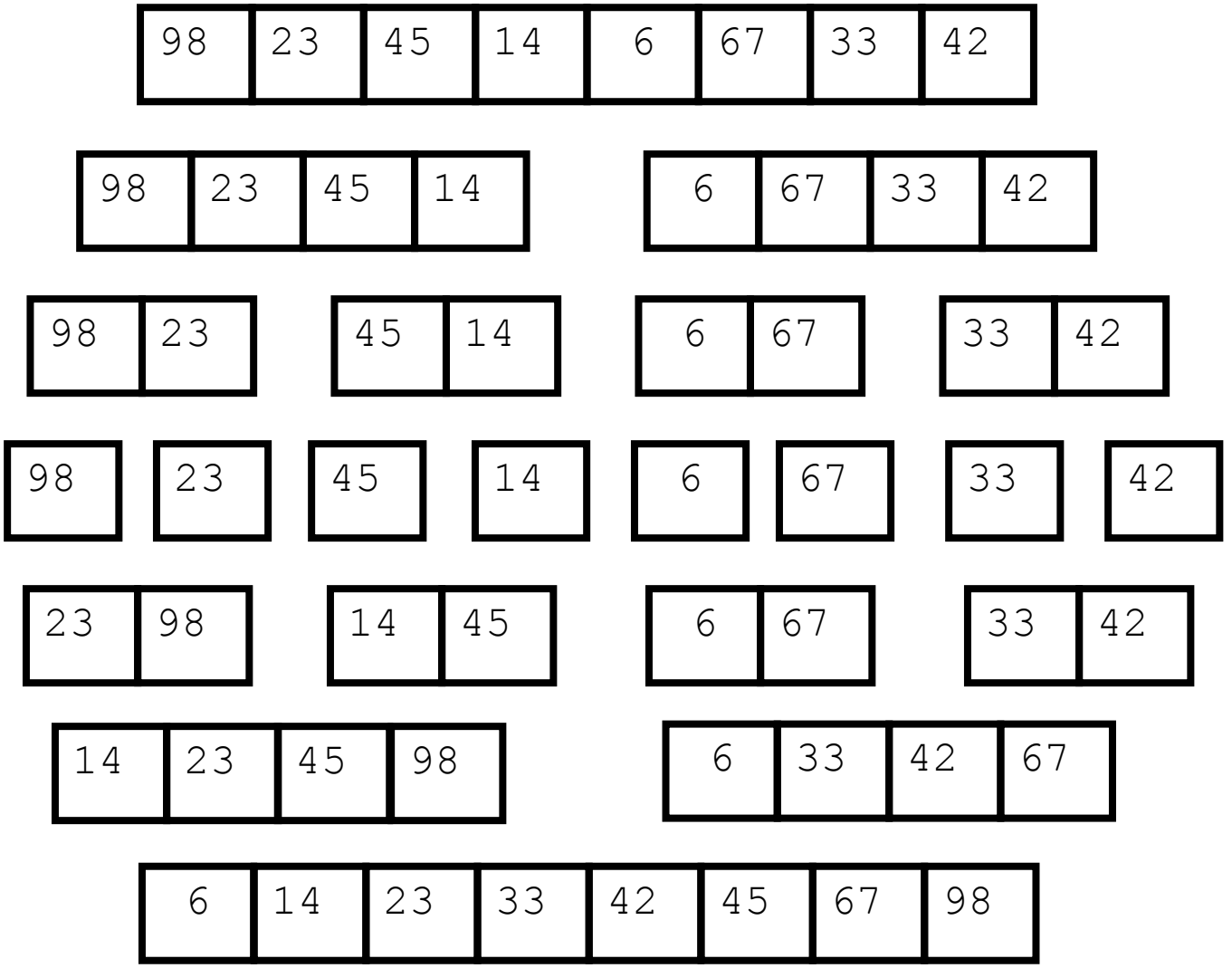


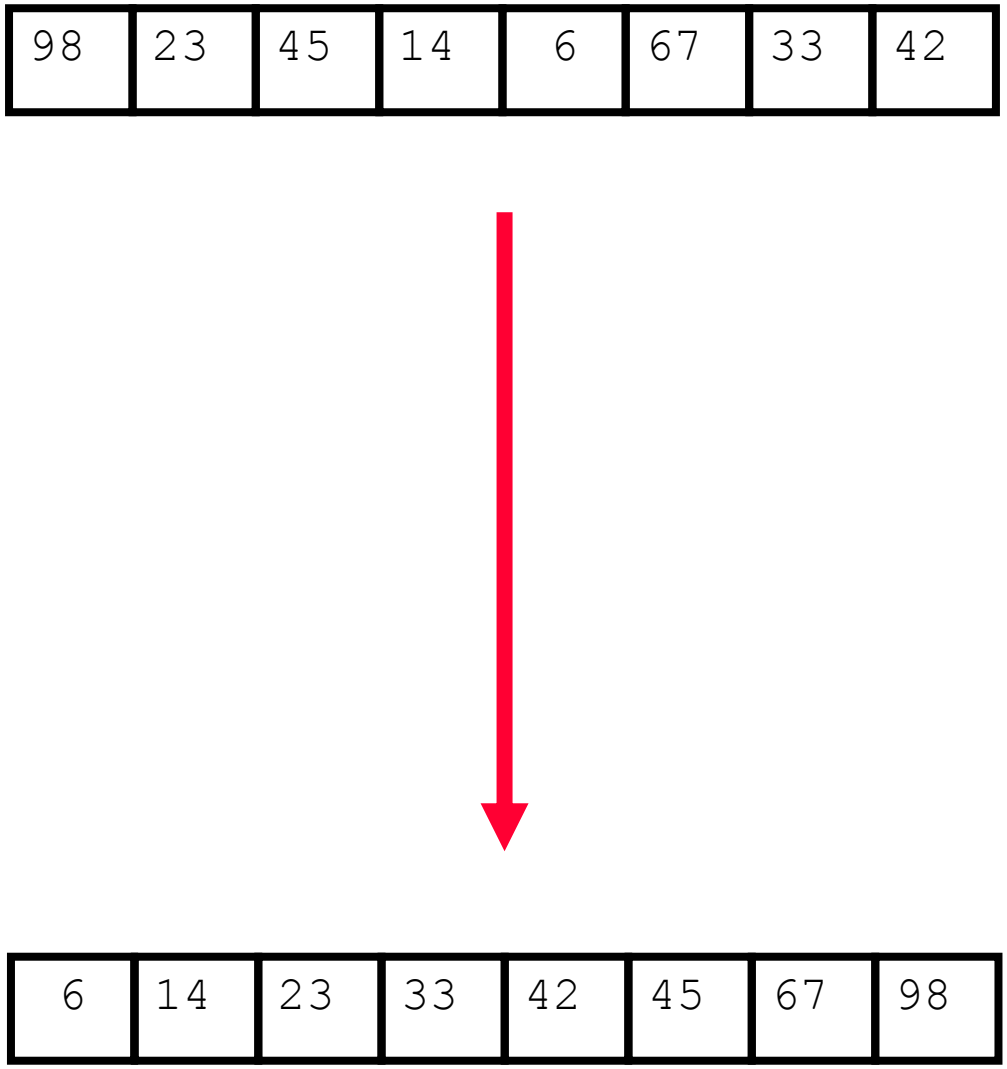












Summary

- Divide the unsorted collection into two
- Until the sub-arrays only contain one element
- Then merge the sub-problem solutions together

Exercise

- Given a randomly ordered set of n numbers, design and develop an algorithm to sort them into non-descending order using bubble sort and merge sort compare their efficiency. Tabulate the output for various inputs and verify against expected values. Analyse the efficiency of both the algorithms. Describe your learning along with the limitations of both, if any. Suggest how these can be overcome.

Key factors for discussion and analysis

- Implement bubble sort and sort random integers
- Implement merge sort and sort random integers
- Analyse the performance of both
- List out the advantages and disadvantages of both

Results and Presentations

- Calculations/Computations/Algorithms

  The calculations/computations/algorithms involved in each program has to be presented

- Presentation of Results

  The results for all the valid and invalid cases have to be presented

- Analysis and Discussions

  how the data is manipulated or transformed, what are the key operations involved. Errors encounters and how they are resolved.

- Conclusions

  Summary

Comments

- Limitations of Experiments

  Outline the loopholes in the program, data structures or solution approach.

- Limitations of Results

  Present the test cases; justify if the program is tested correctly considering all the outcomes. Mention what is not tested, if any.

- Learning happened

  What is the overall learning happened

- Conclusions

  Summary

References

- Gilberg, R. F., and Forouzan, B. A. (2007): A Pseudocode Approach With C, 2nd edn. Cengage Learning