Sorting Techniques

Unit-5 Lecture-18

Dr. Dillip Rout, Assistant Professor, Dept. of Computer Science and Engineering

Outline

- Fundamentals of Sorting
- Comparison vs Non-comparison Sorting
- Bubble Sort
- Selection Sort
- Insertion Sort
- Heap Sort
- Counting Sort
- Homework
- Conclusions

Fundamental of Sorting

Introduction

- Sorting is a technique to arrange/order the given list of elements in an ascending or non-decreasing (if duplicates present). If the elements have values as strings then a lexicography order is followed.
- Examples of Sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Counting Sort, etc.

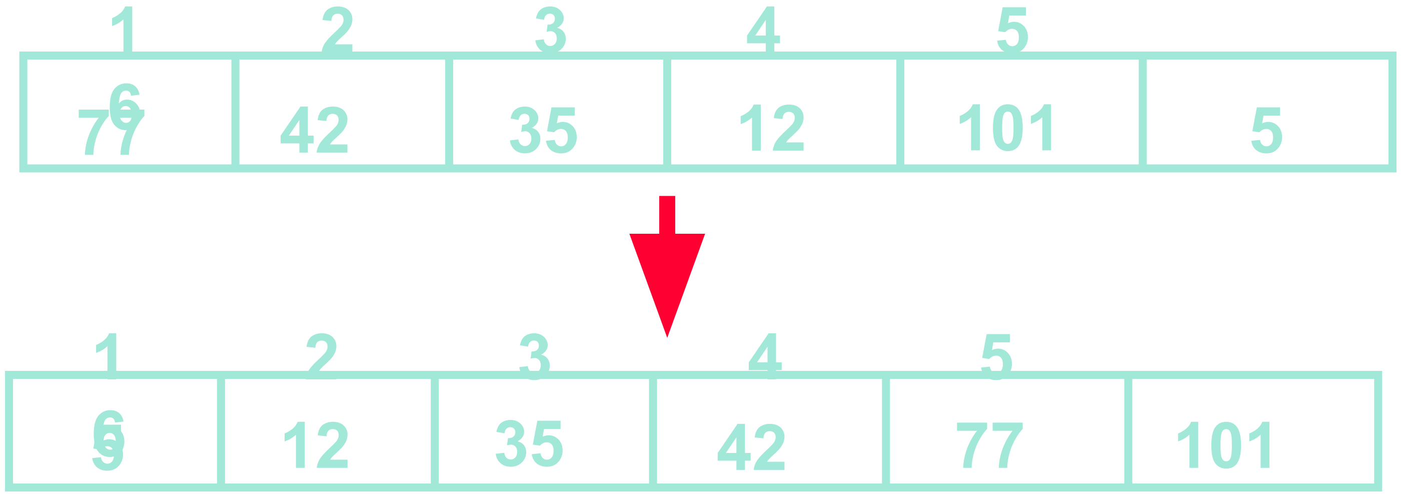
Types of Sorting Algorithm

- Comparison Based:
  - Bubble Sort, Selection Sort, Insertion Sort, Heap Sort
- Non-Comparison Based:
  - Counting Sort (frequency), Radix Sort (weight on position)
- Divide-and-Conquer:
  - Heap Sort, Insertion Sort
- Data Structure:
  - Array: Bubble Sort, Selection Sort, Insertion Sort, Counting Sort
  - Tree: Heap Sort

Bubble Sort

Sorting

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



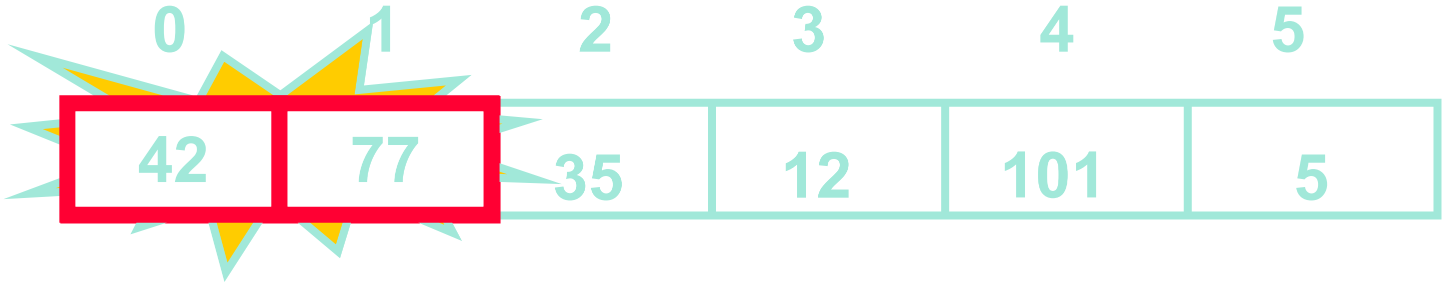
"Bubbling Up" the Largest Element

- 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**

0	1	2	3	4	5
77	42	35	12	101	5

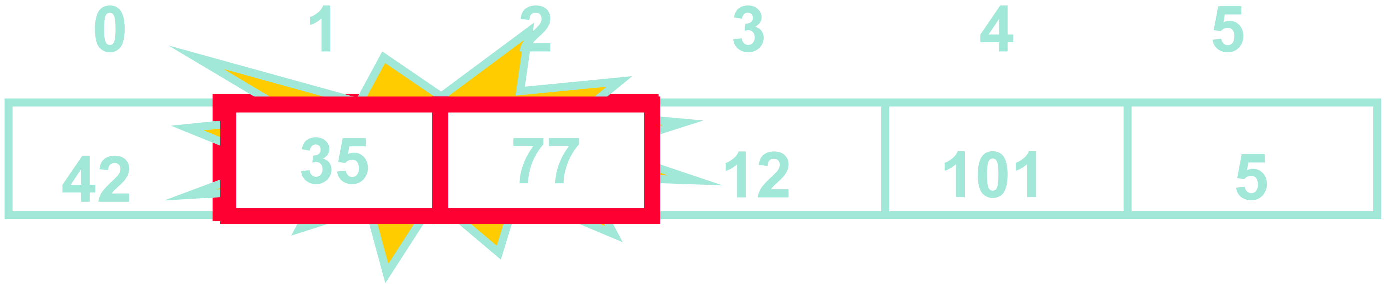
"Bubbling Up" the Largest Element

- 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



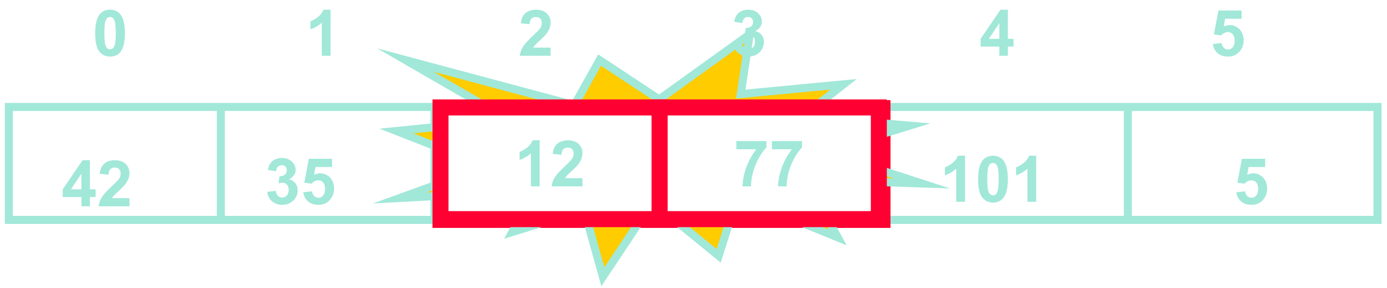
"Bubbling Up" the Largest Element

- 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



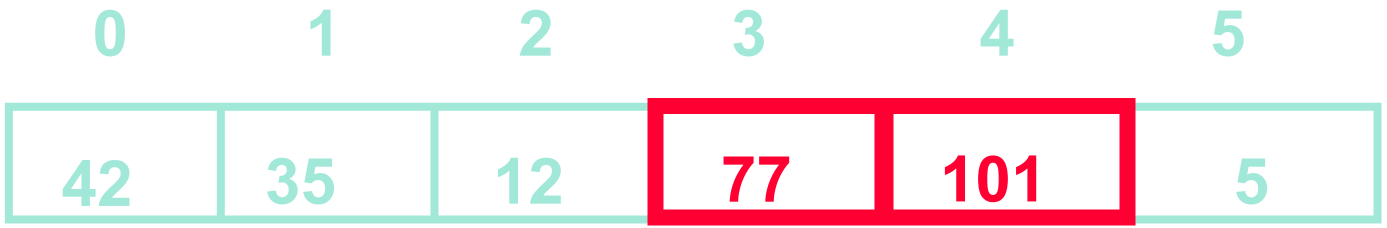
"Bubbling Up" the Largest Element

- 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

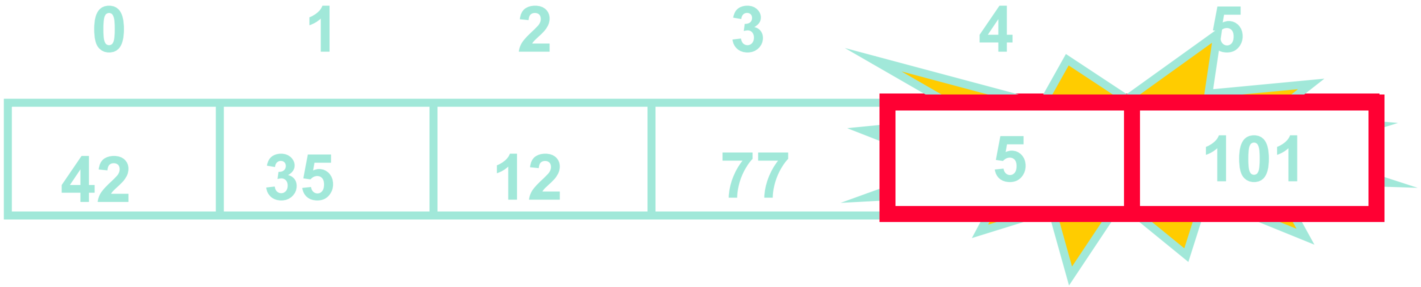
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



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



"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

0	1	2	3	4	5
42	35	12	77	5	**101**

Largest value correctly placed

The “Bubble Up” Algorithm

```
Algorithm BubbleUp(A, n)
1.  for i= 0 to n-2 do
2.    if(A[i] > A[i + 1]) then
3.      Swap(A[i], A[i + 1])
```

Time Complexity = O(n), Space Complexity = O(1)

What is the outcome of BubbleUp algorithm?

```
Procedure Swap(a, b)
1.  t = a
2.  a = b
3.  b = t
```

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**

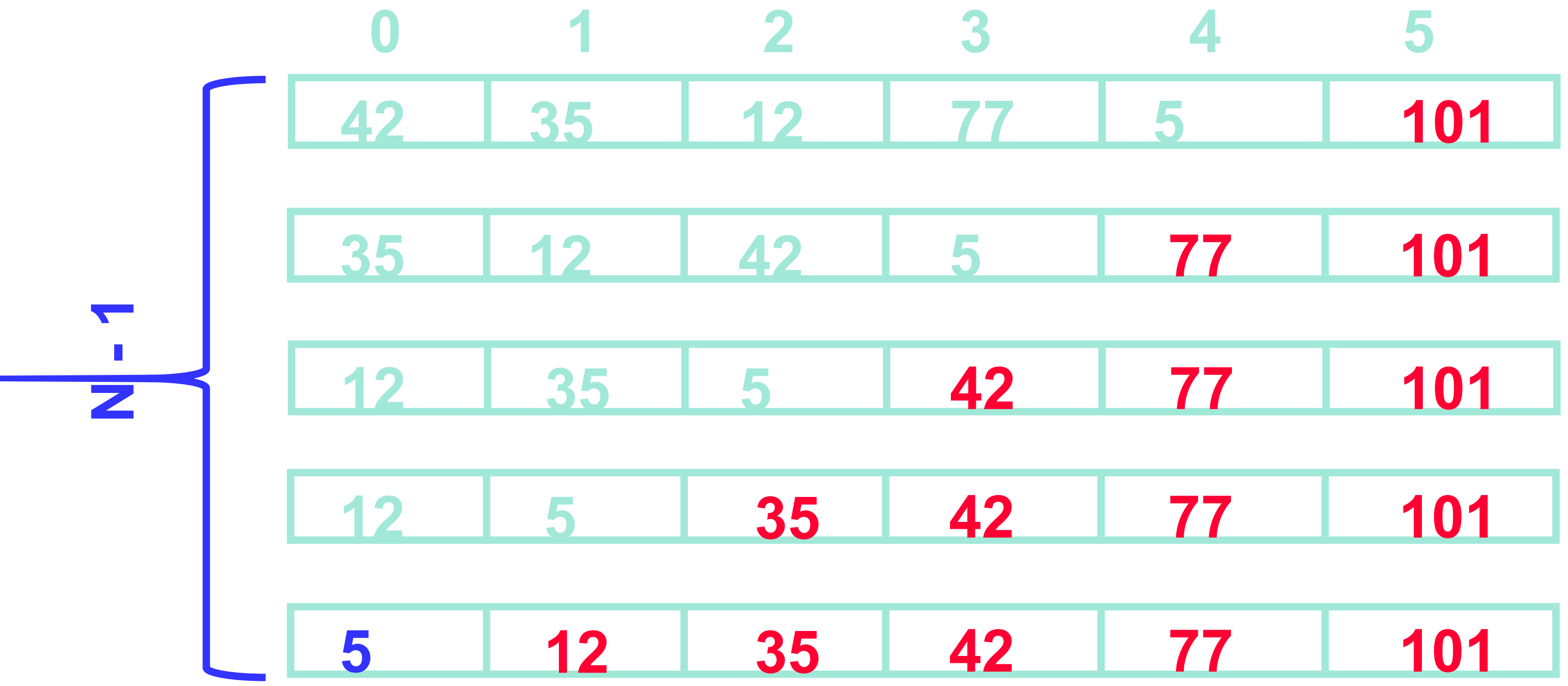
0	1	2	3	4	5
42	35	12	77	5	**101**

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**

"Bubbling" All the Elements



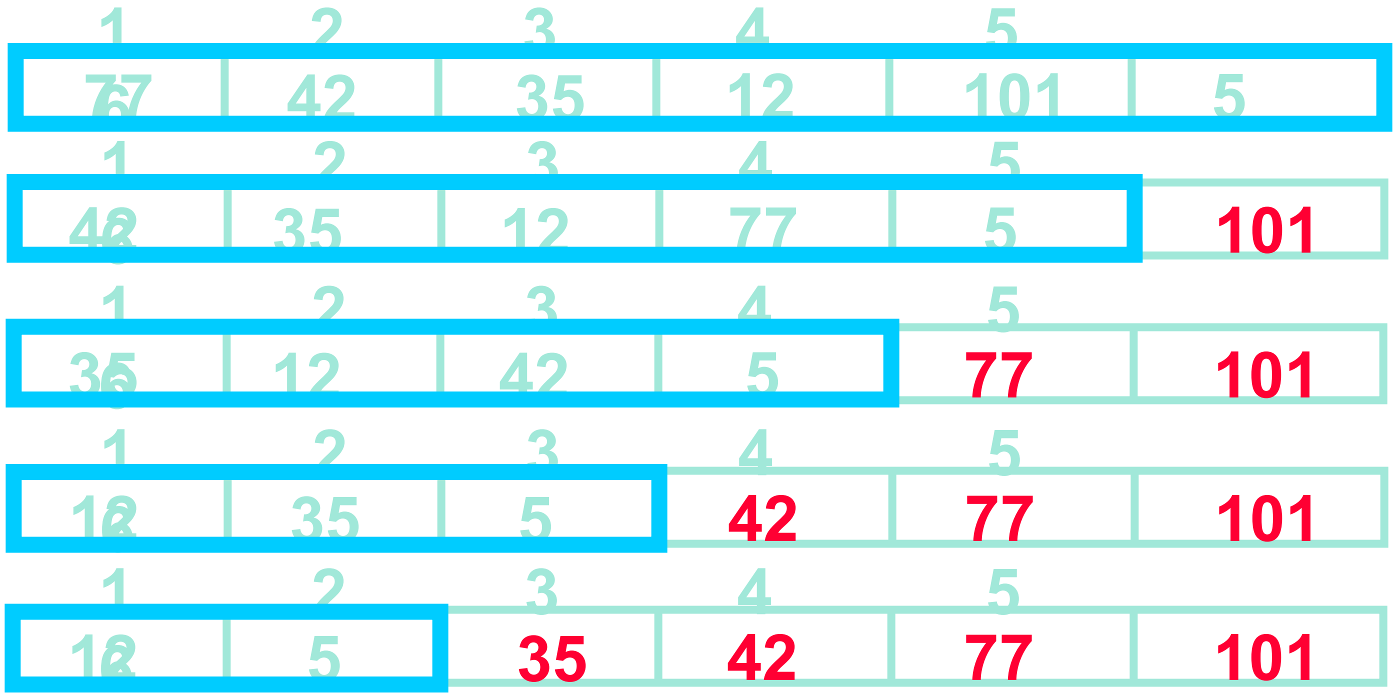
0	1	2	3	4	5
42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101
5	12	35	42	77	101

Bubble Sort Algorithm

```
Algorithm BubbleSort(A, n)
1.  for j = 1 to n-1 do
2.    for i= 0 to n-2 do
3.      if(A[i] > A[i + 1]) then
4.        Swap(A[i], A[i + 1])
```

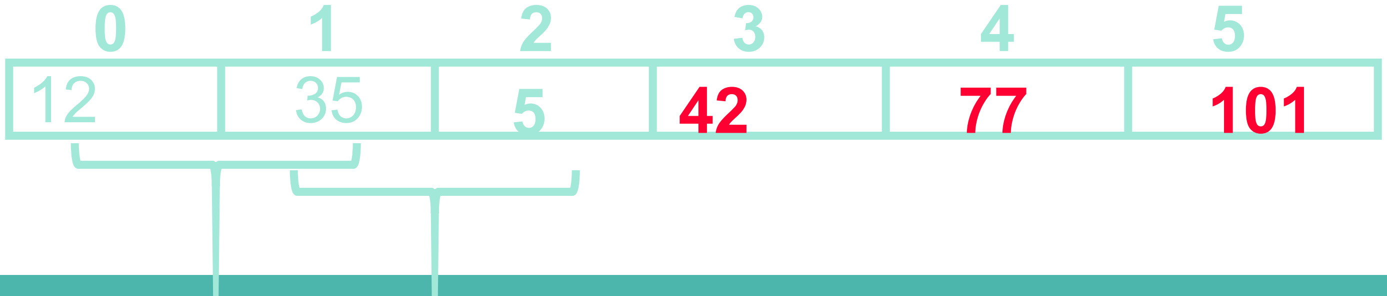
Time Complexity = O(n^2), Space Complexity = O(1)

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 3rd "bubble up"
  - MAX is 5
  - Thus we have **2 comparisons** to do



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"!**

0	1	2	3	4	5
5	12	35	42	77	101

Using a Boolean “Flag”

- We can use a boolean variable to determine if any swapping occurred during the “bubble up.”

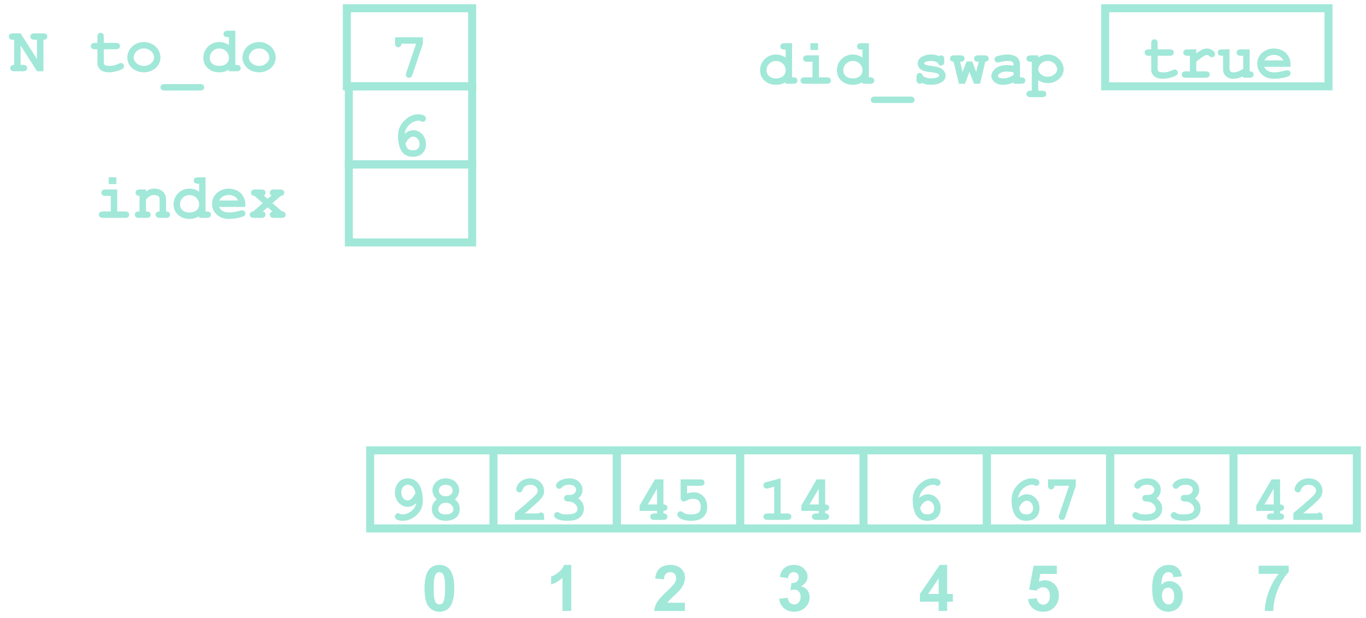
- If no swapping occurred, then we know that the collection is already sorted!

- This boolean “flag” needs to be reset after each “bubble up.”

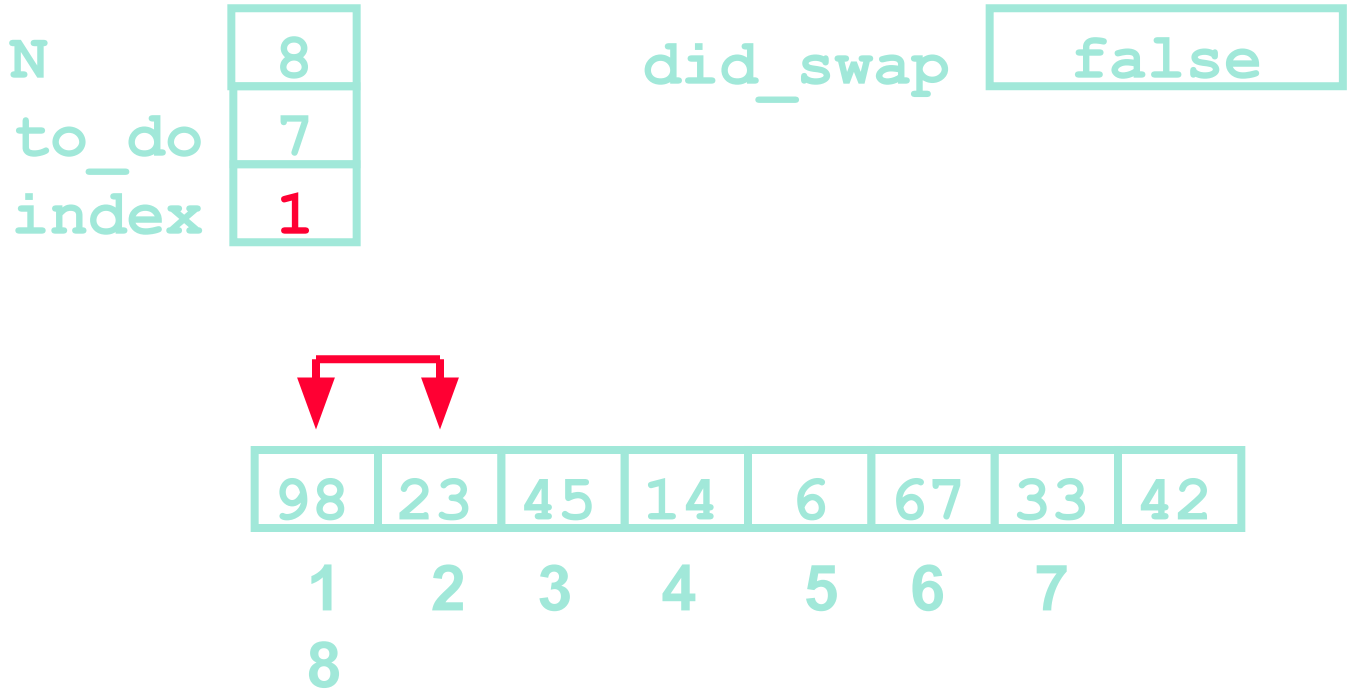
Bubble Sort Improved

```
Algorithm BubbleSort(A, n)
1.  for j = 1 to n-1 do
2.    flag = FALSE
3.    for i= 0 to n-2 do
4.      if(A[i] > A[i + 1]) then
5.        flag = TRUE
6.        Swap(A[i], A[i + 1])
7.    if flag == FALSE then
8.      ???
```

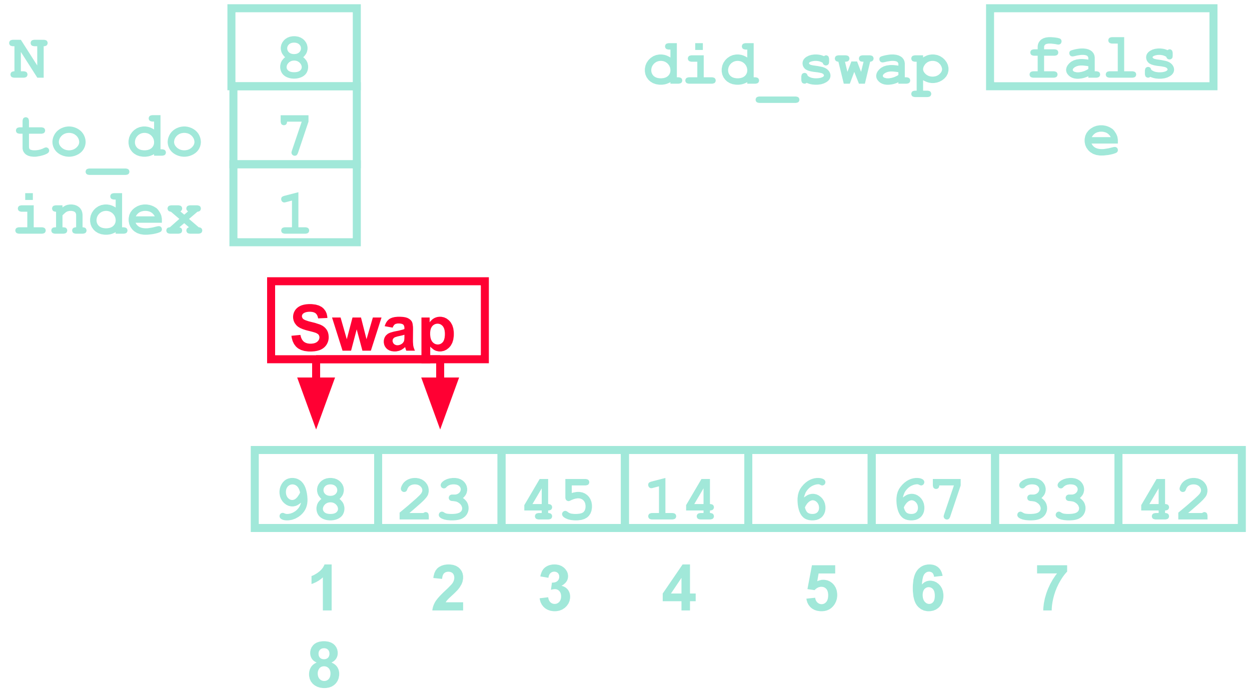
An Animated Example



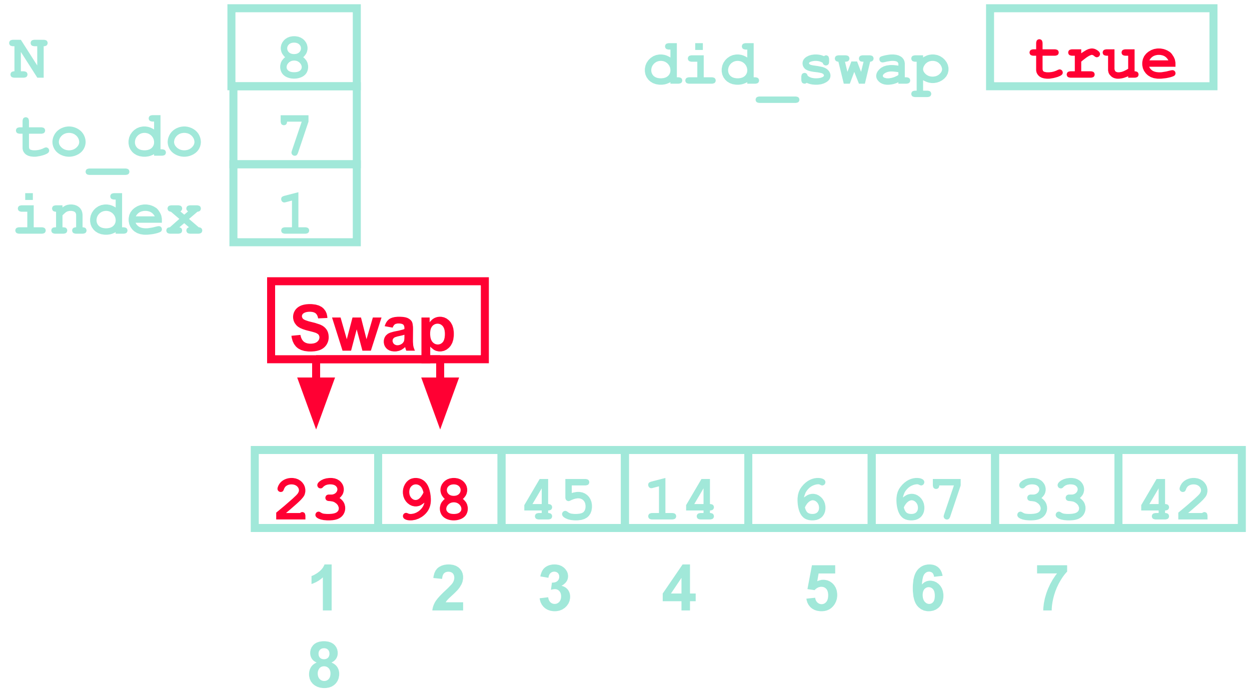
An Animated Example



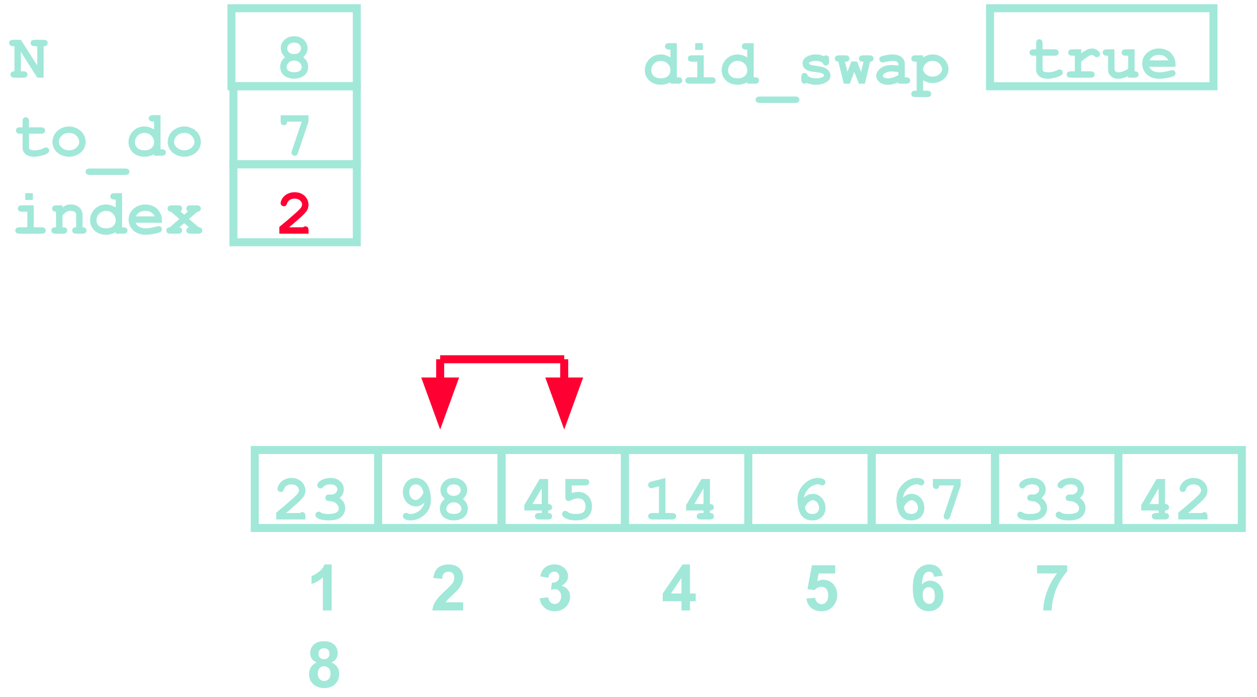
An Animated Example



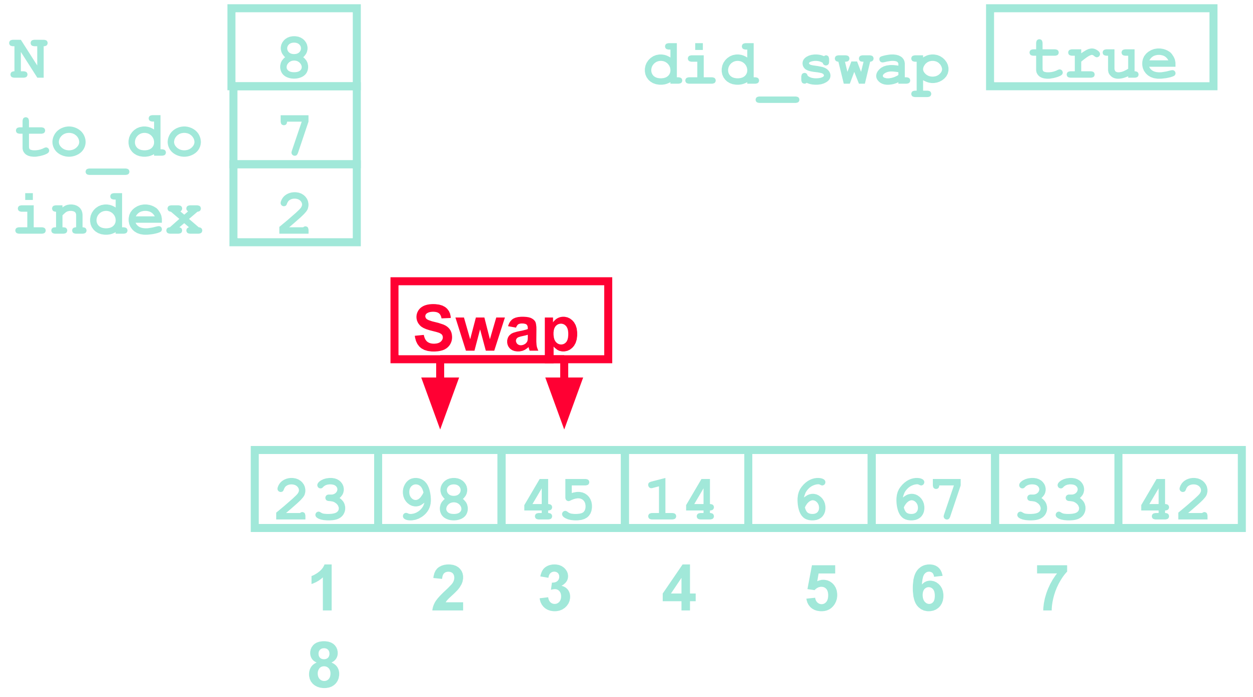
An Animated Example



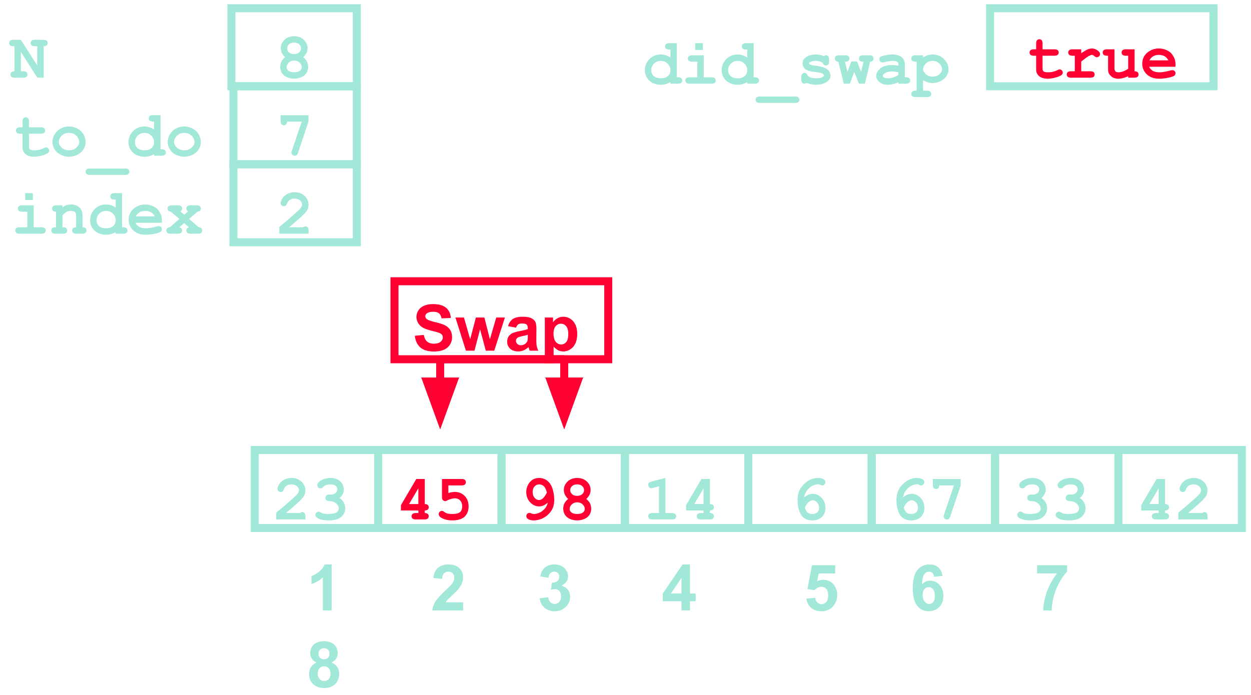
An Animated Example



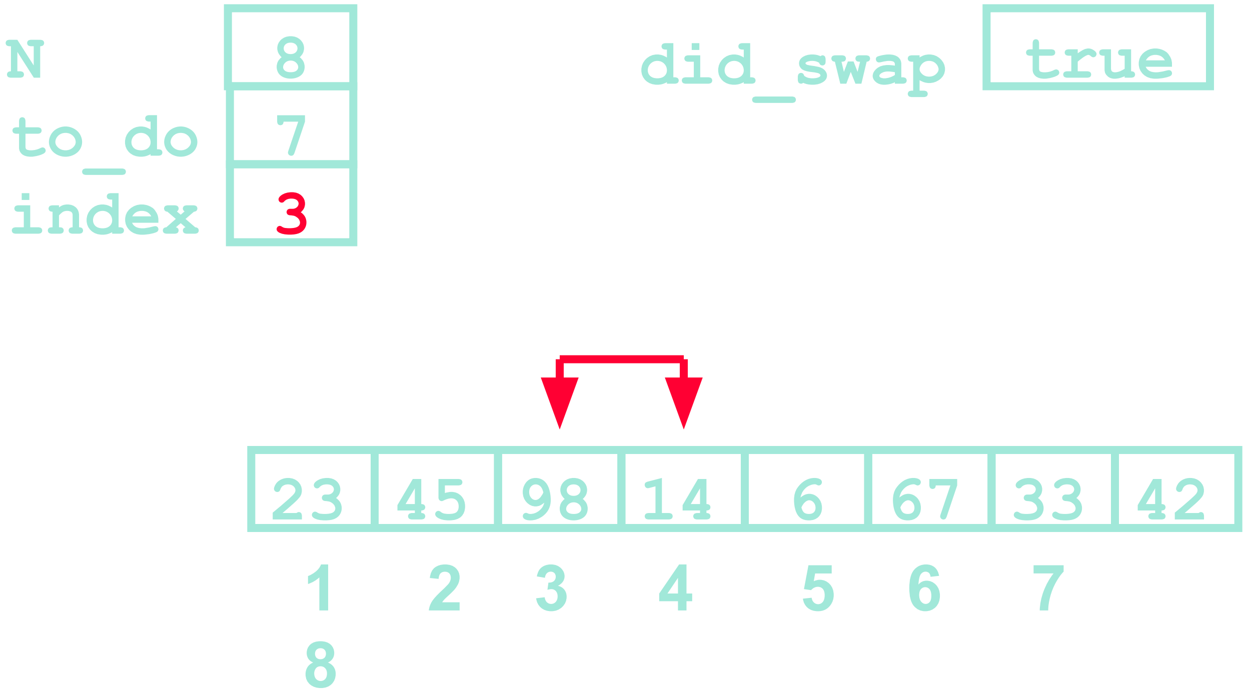
An Animated Example



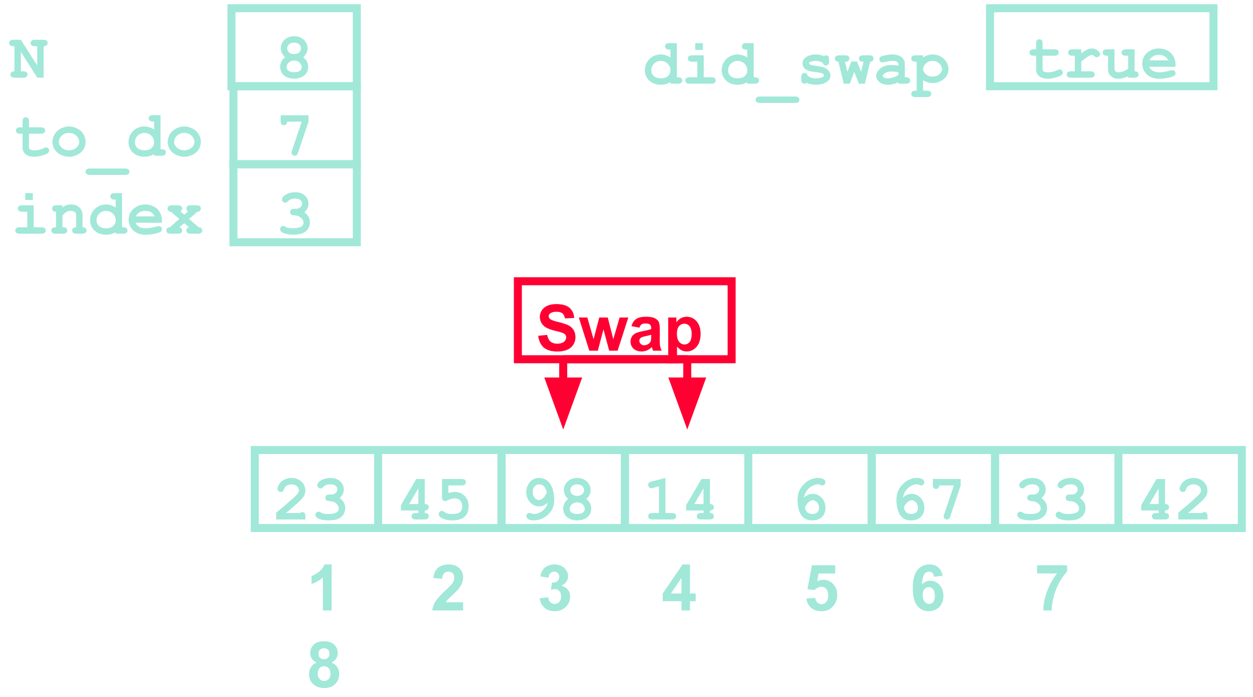
An Animated Example



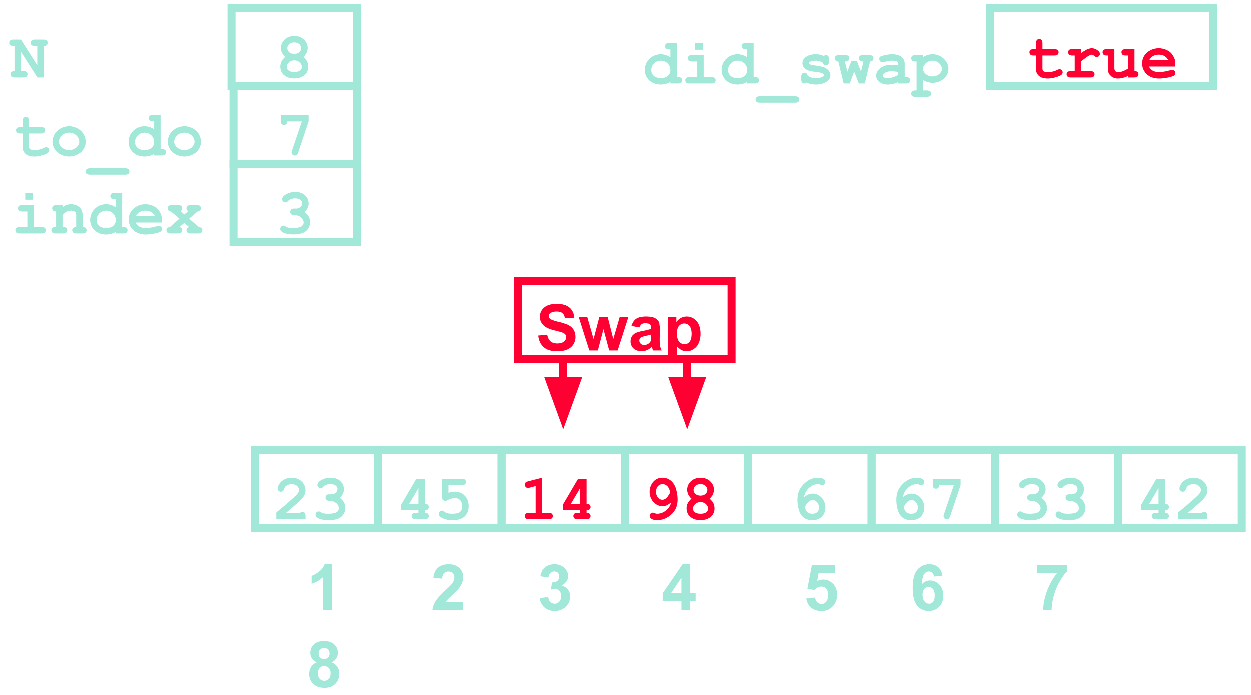
An Animated Example



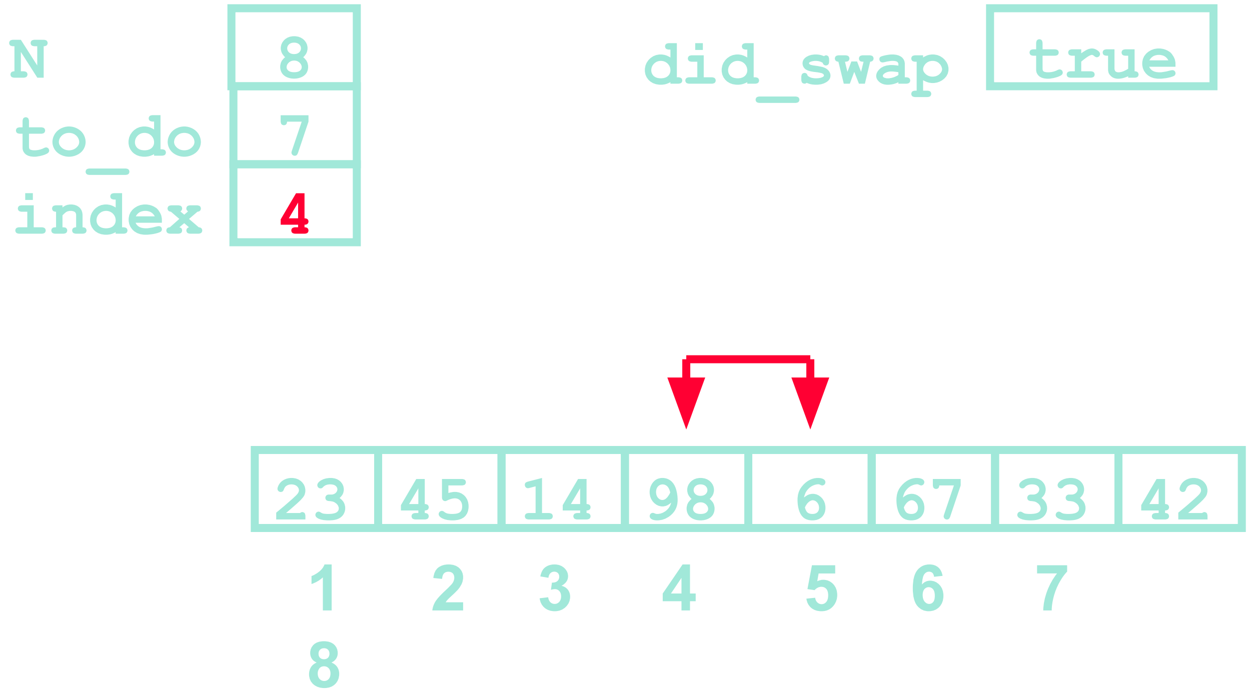
An Animated Example



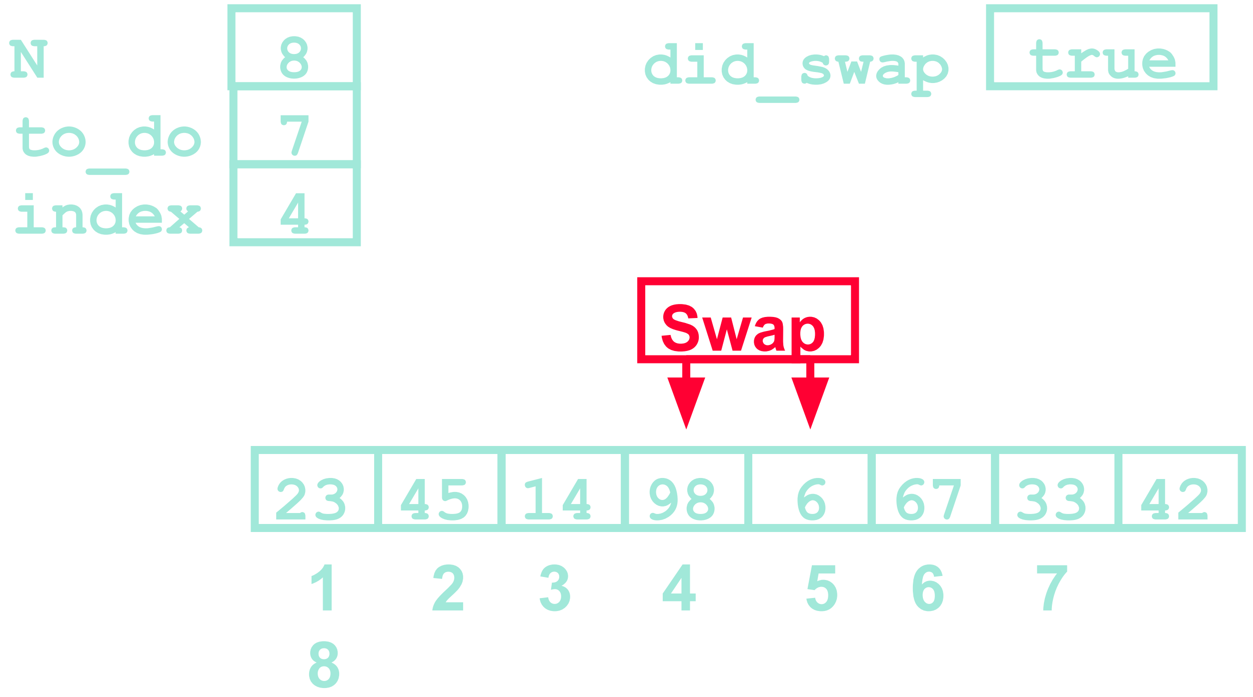
An Animated Example



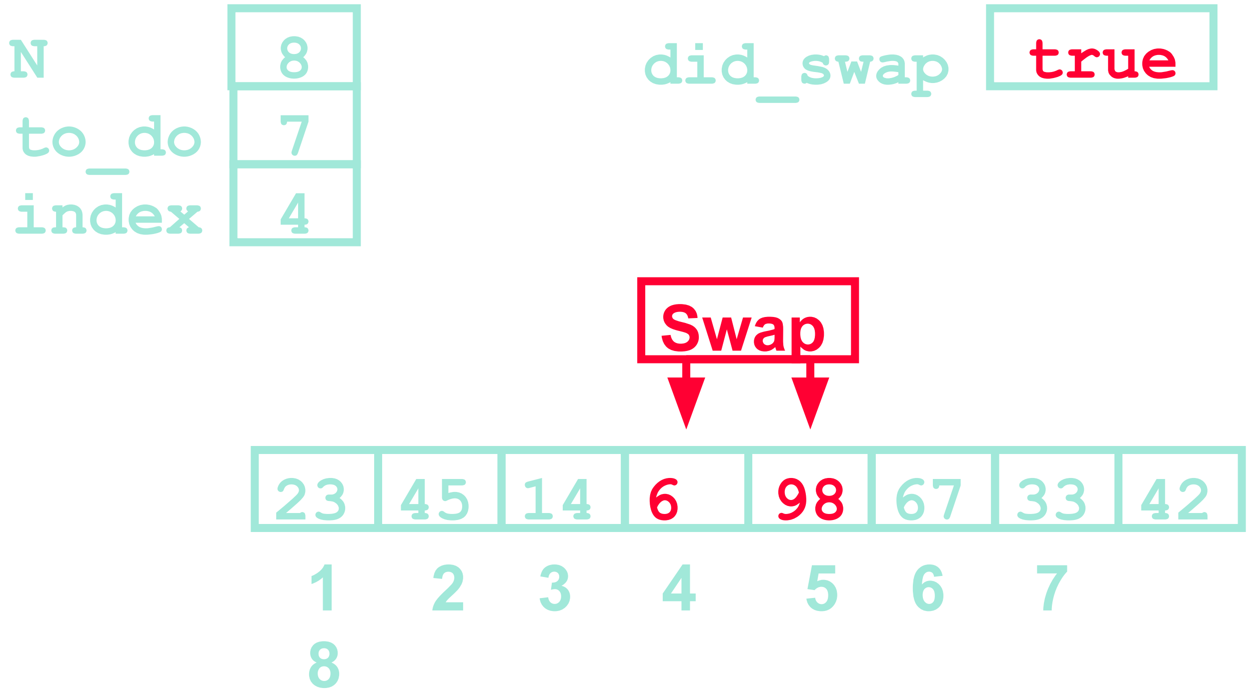
An Animated Example



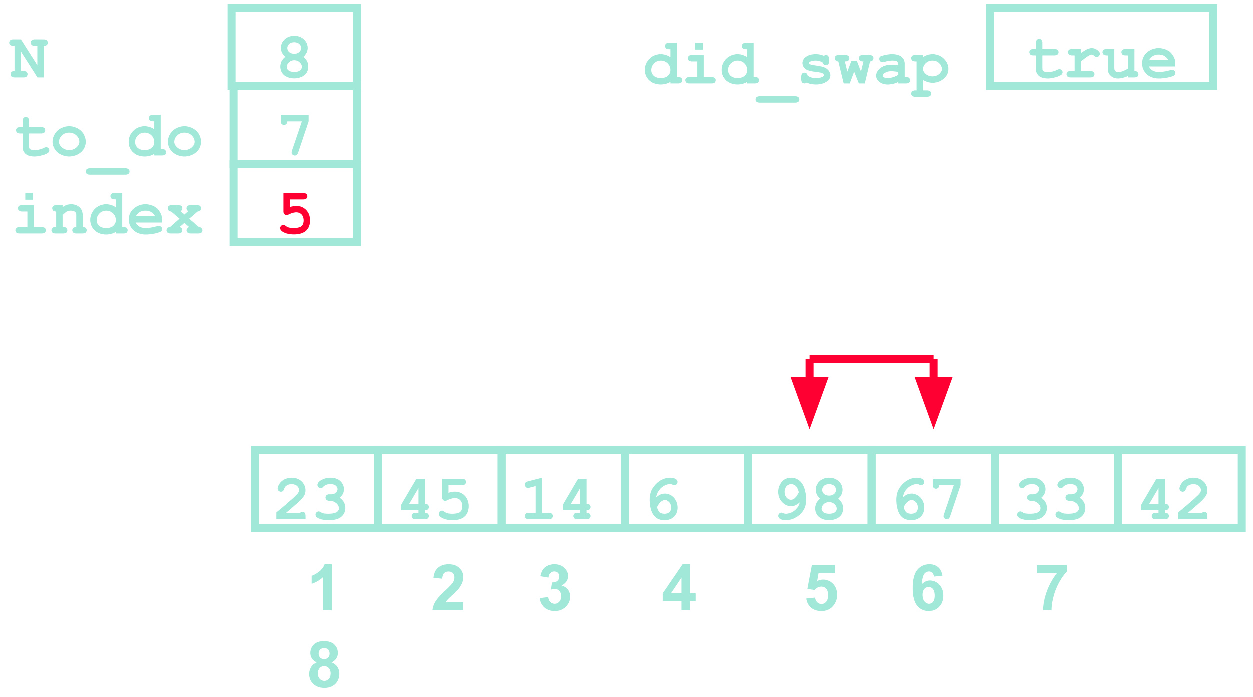
An Animated Example



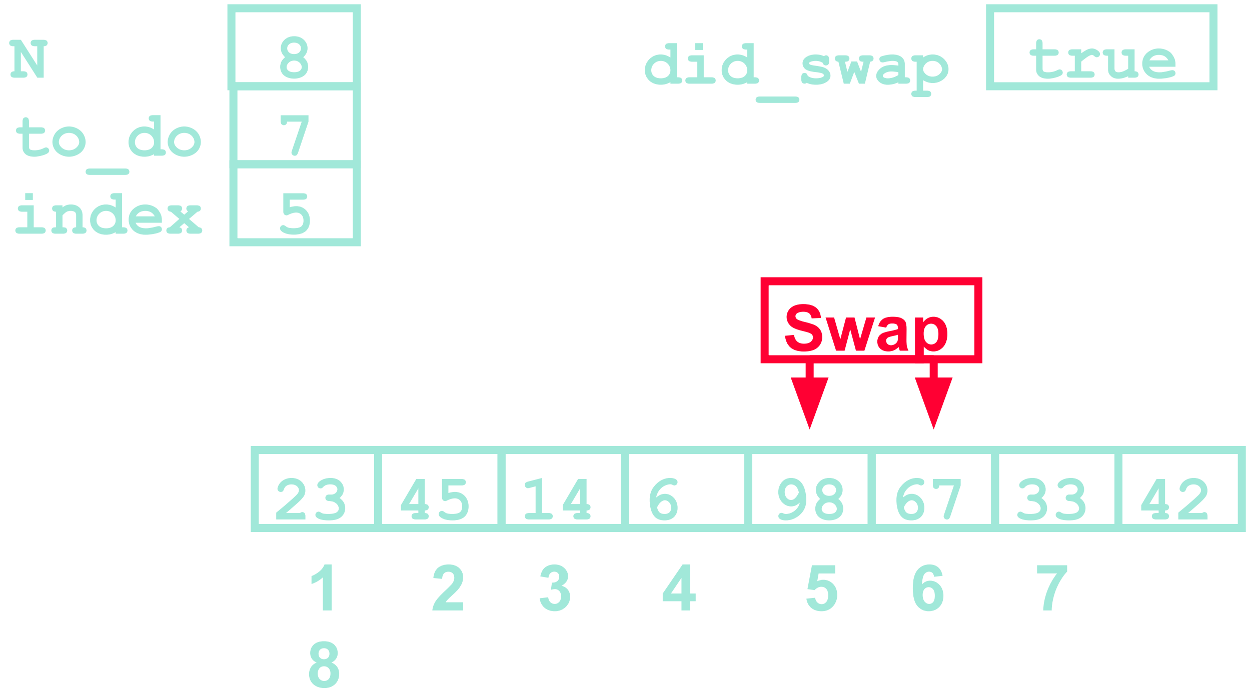
An Animated Example



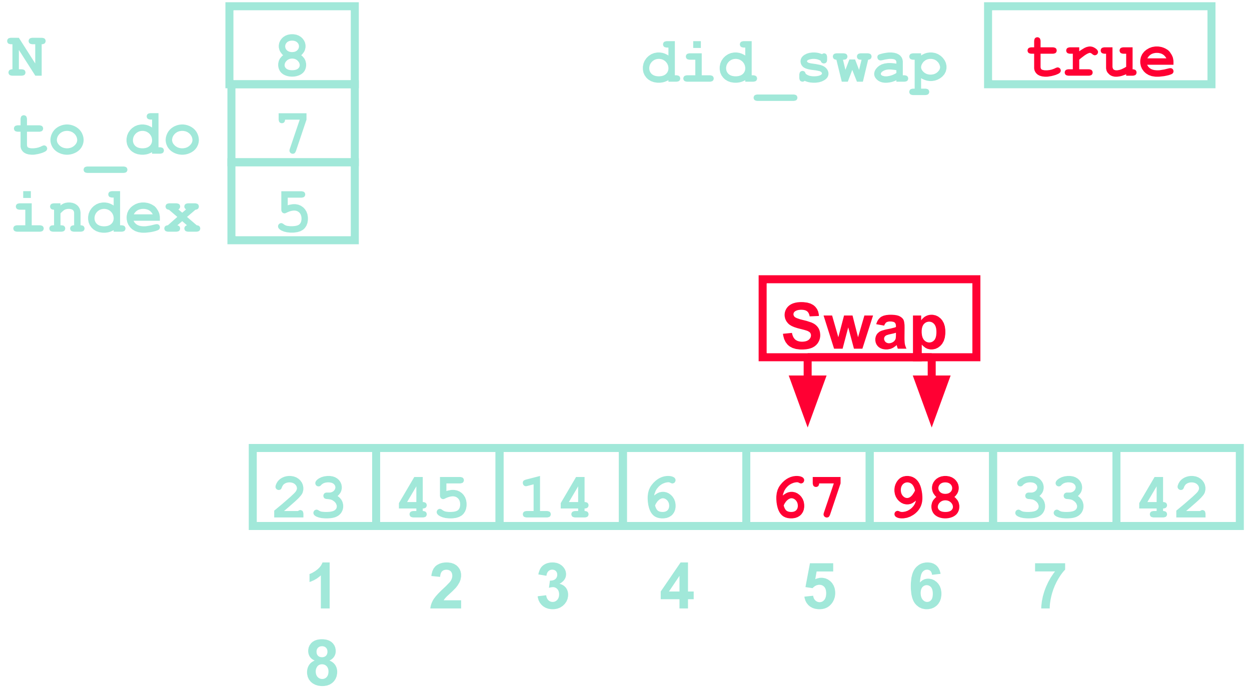
An Animated Example



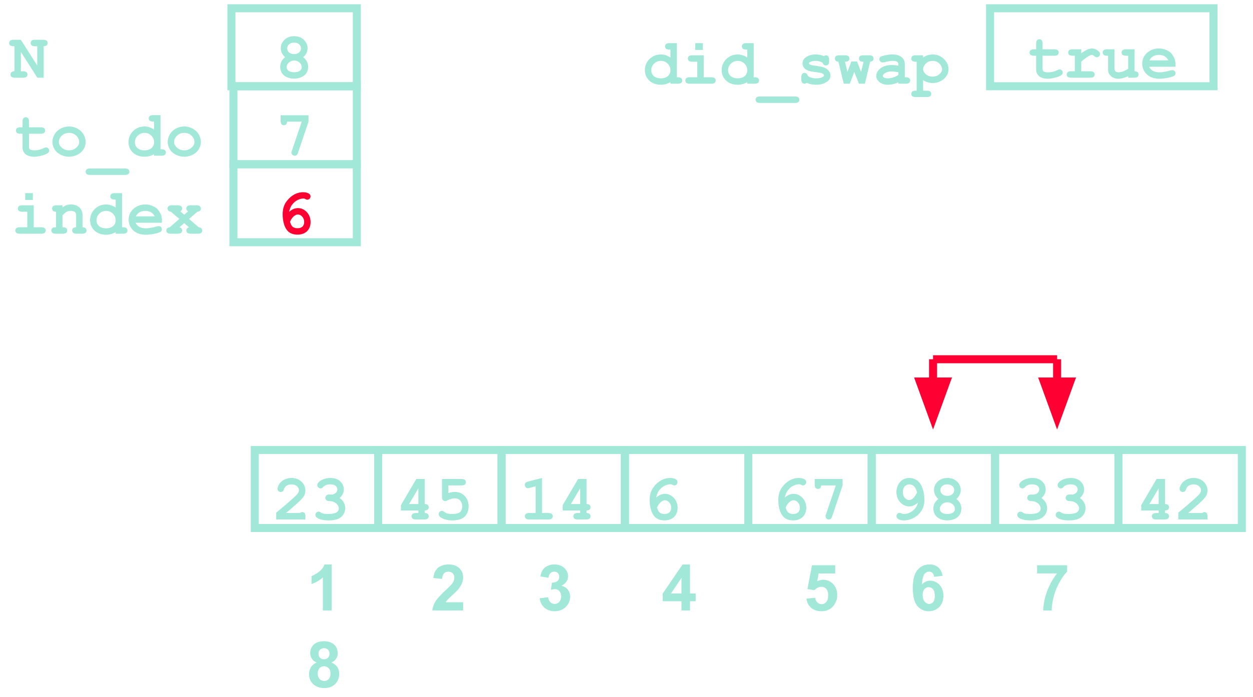
An Animated Example



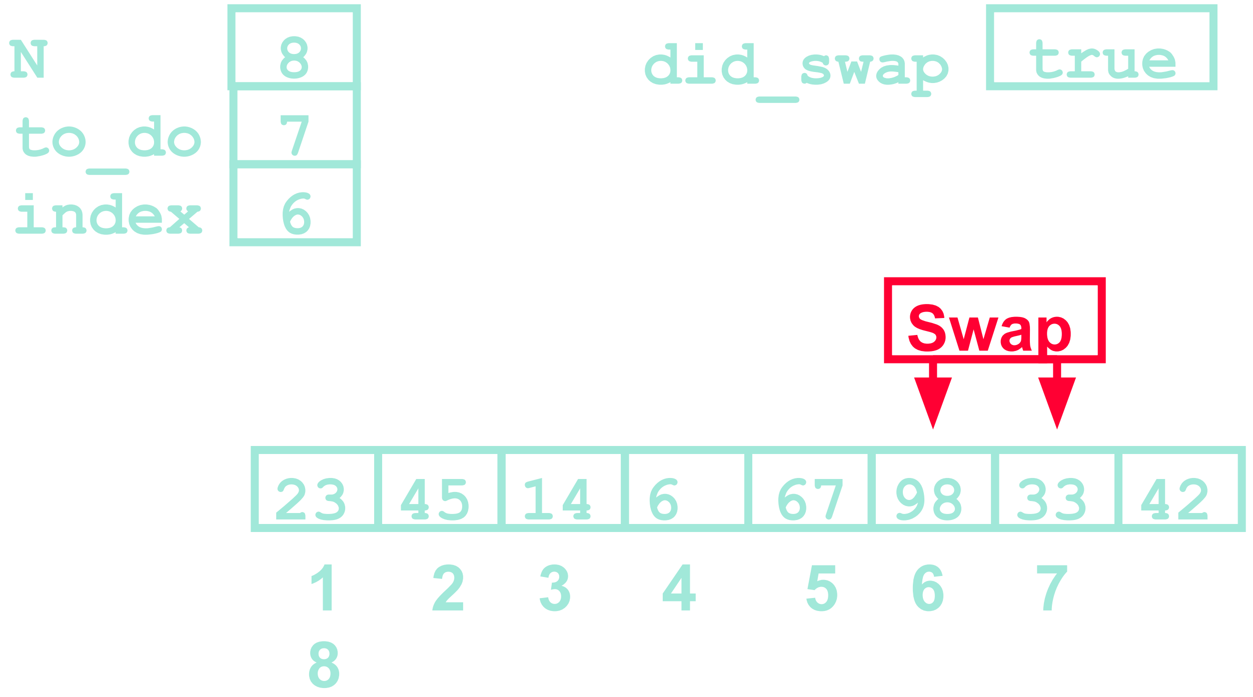
An Animated Example



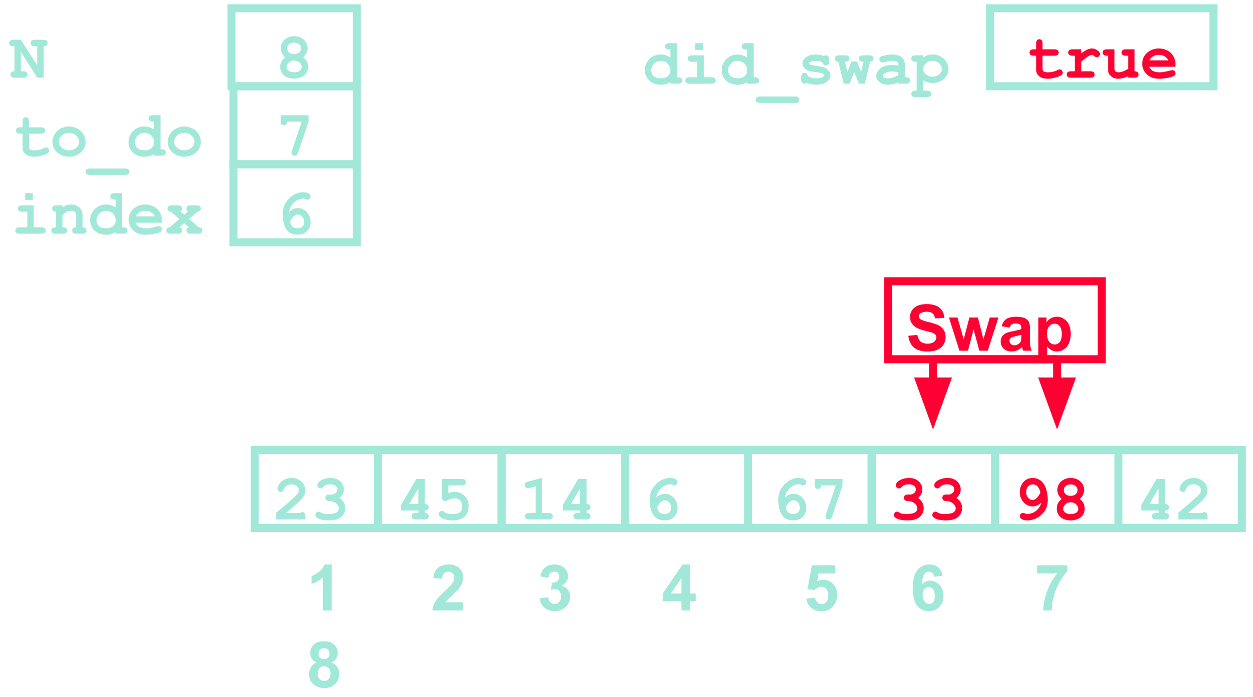
An Animated Example



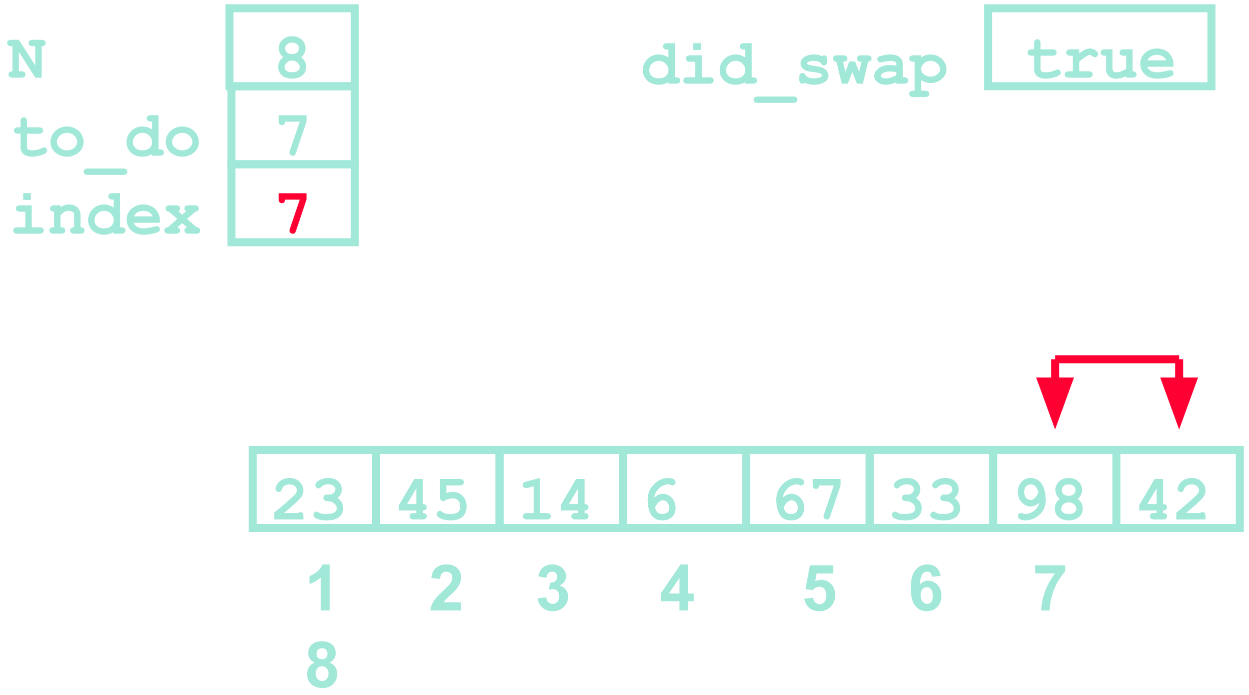
An Animated Example



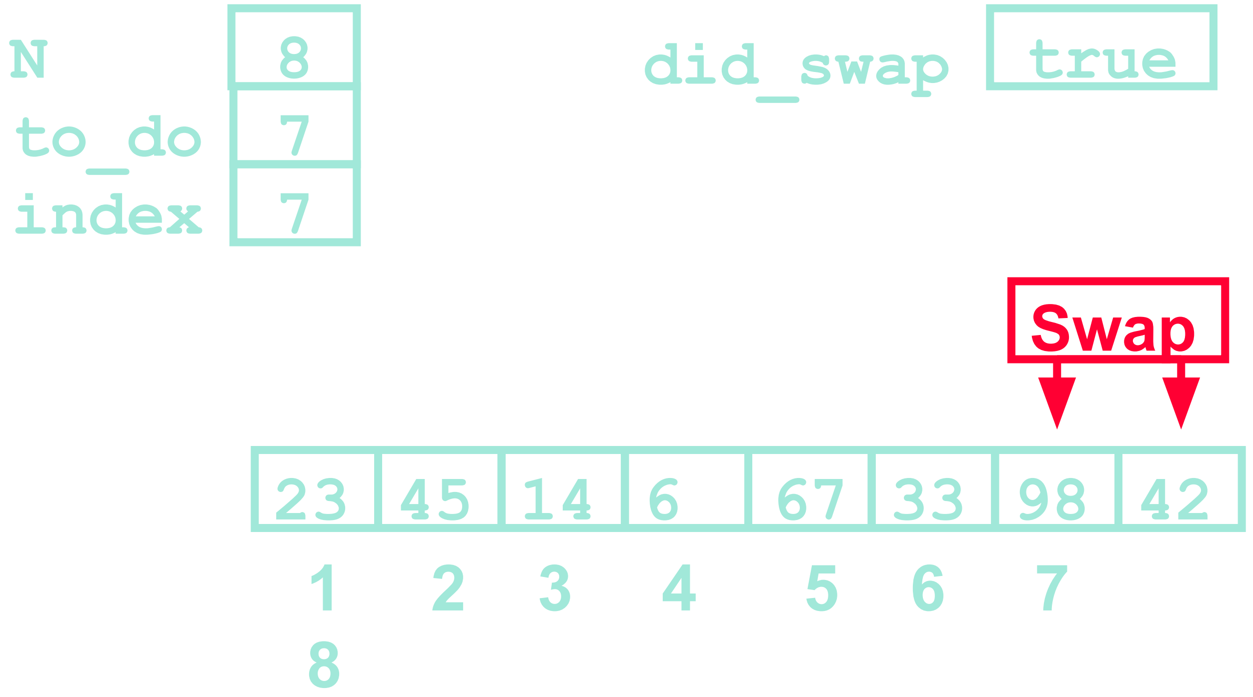
An Animated Example



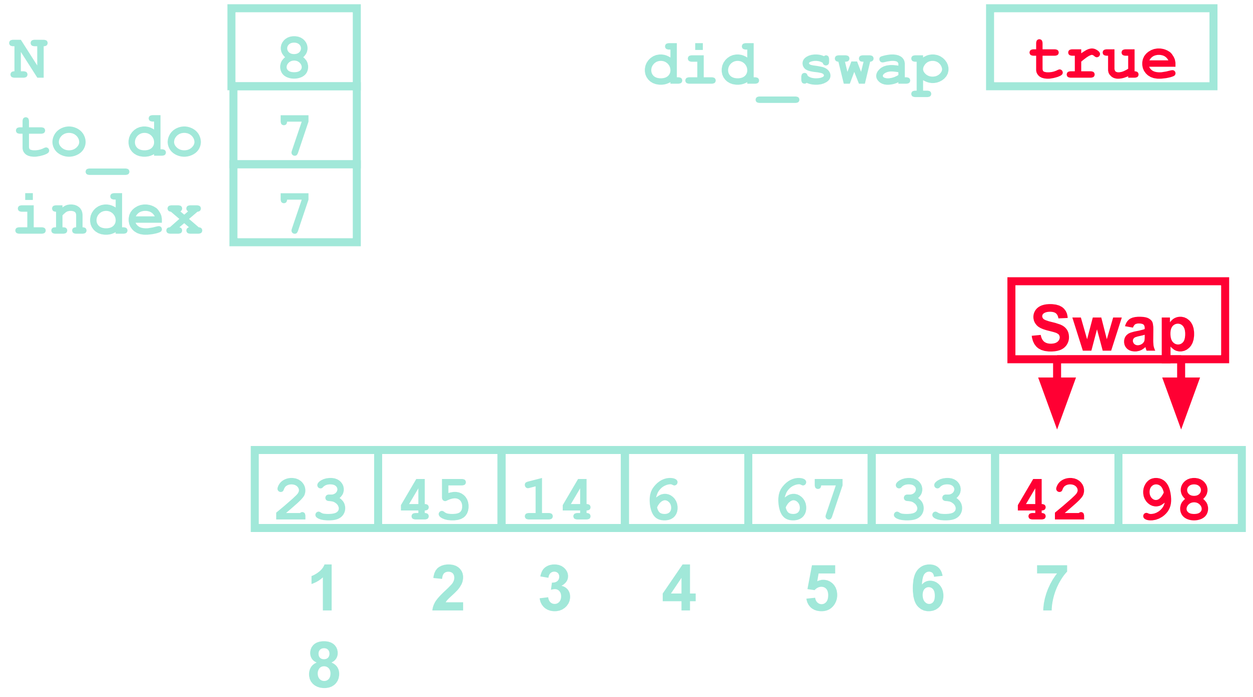
An Animated Example



An Animated Example



An Animated Example

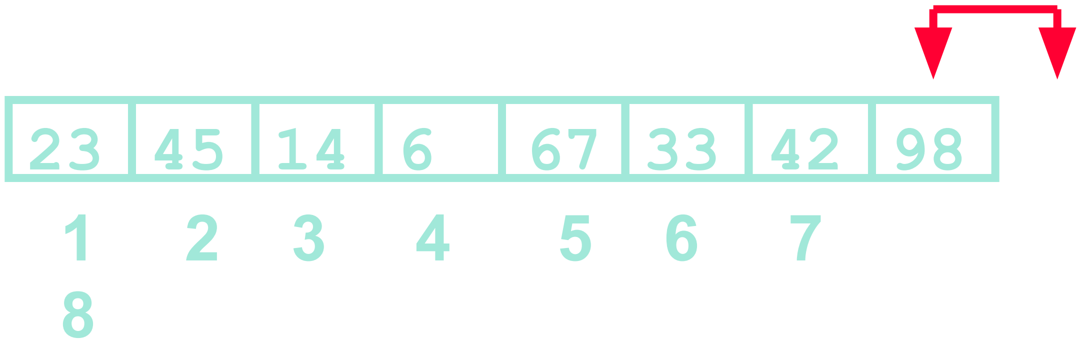


After First Pass of Outer Loop

N	8
to_do	7
index	8

did_swap `true`

Finished first “Bubble Up”

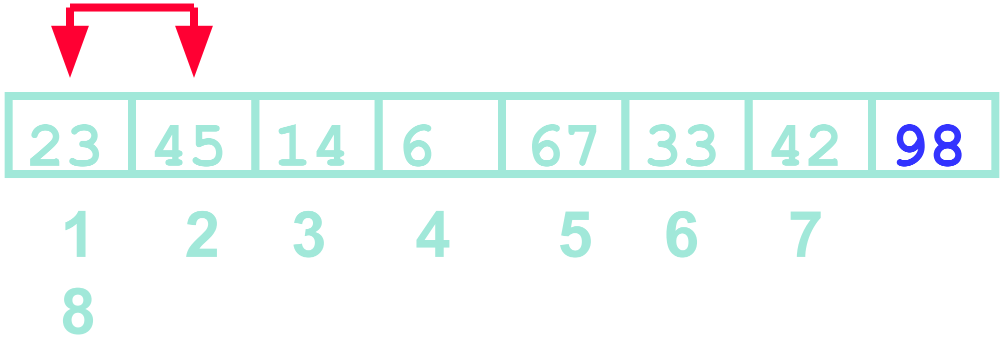


1	2	3	4	5	6	7	8
23	45	14	6	67	33	42	98

The Second “Bubble Up”

N	8
to_do	6
index	1

did_swap	fals e



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

The Second “Bubble Up”

N	8
to_do	6
index	1

did_swap false

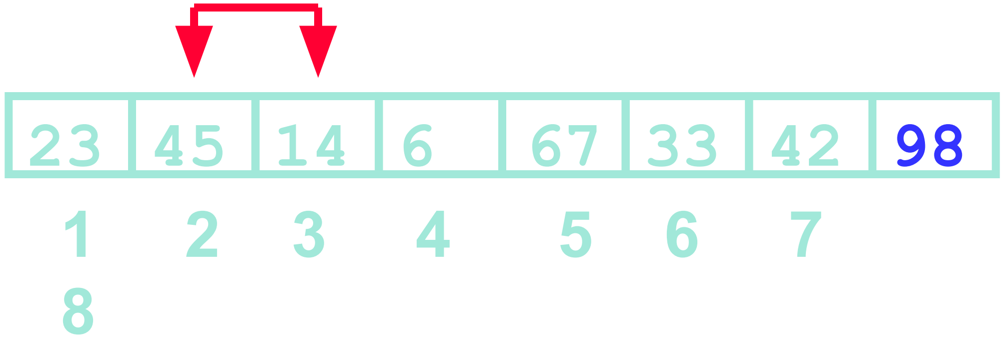
No Swap

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

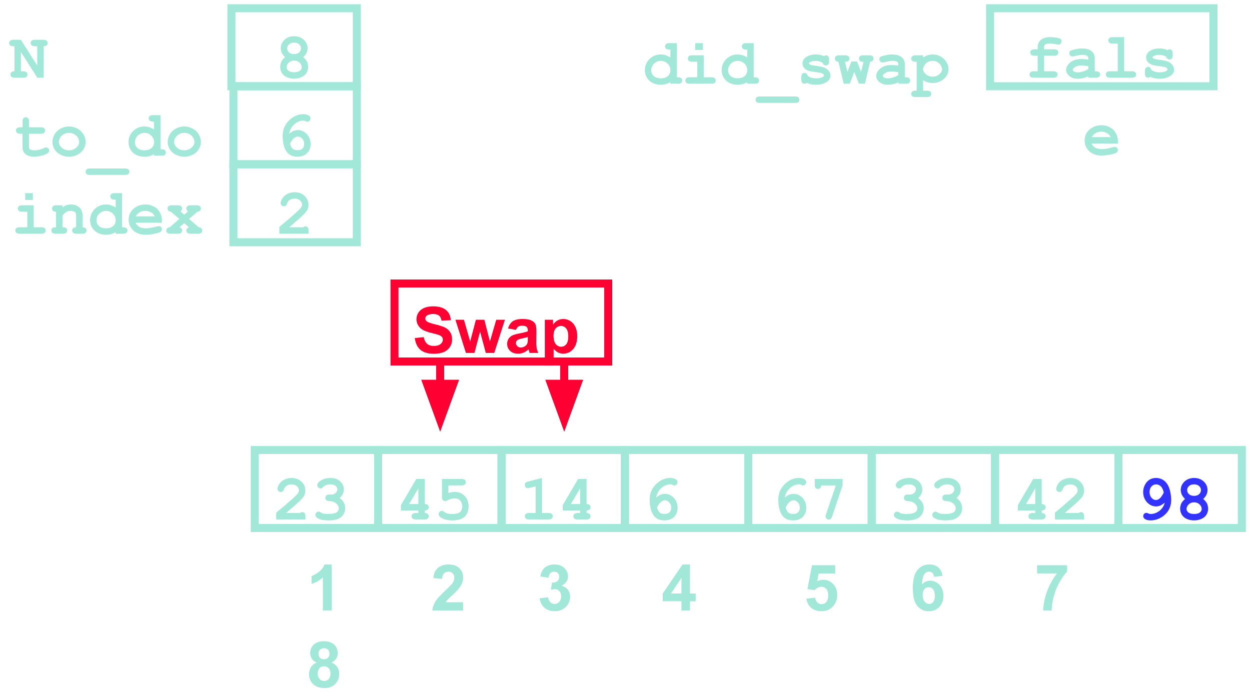
The Second “Bubble Up”

N	8
to_do	6
index	2

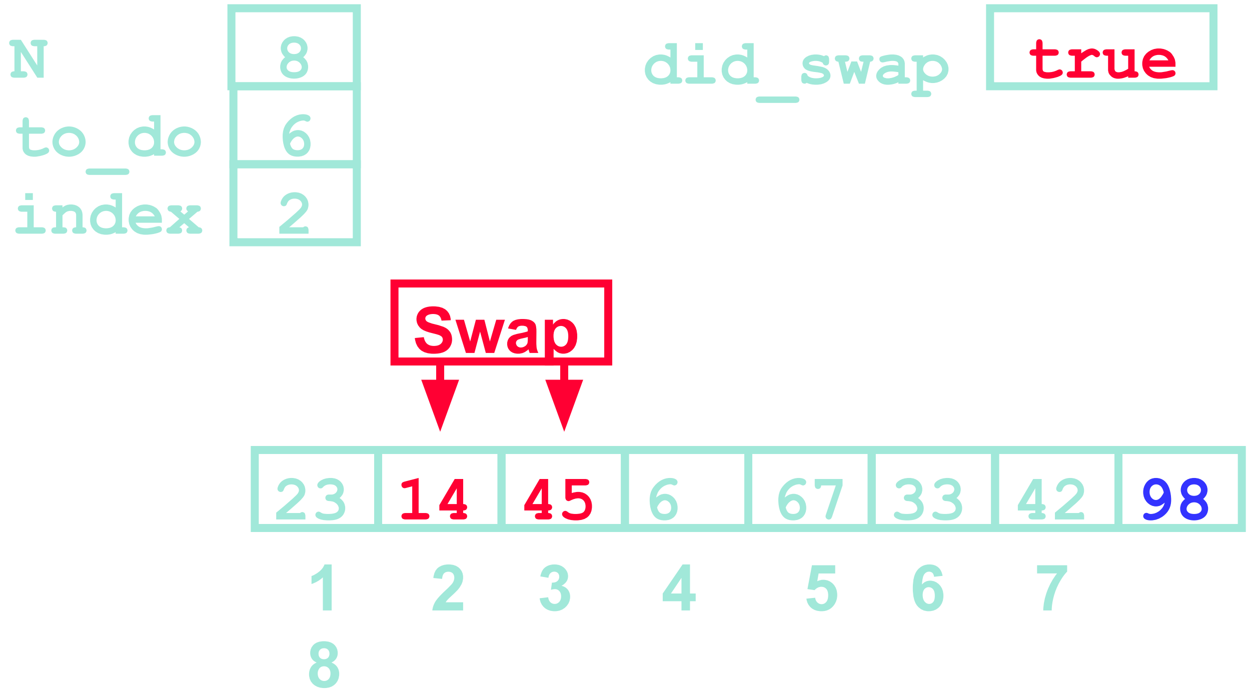
did_swap	false



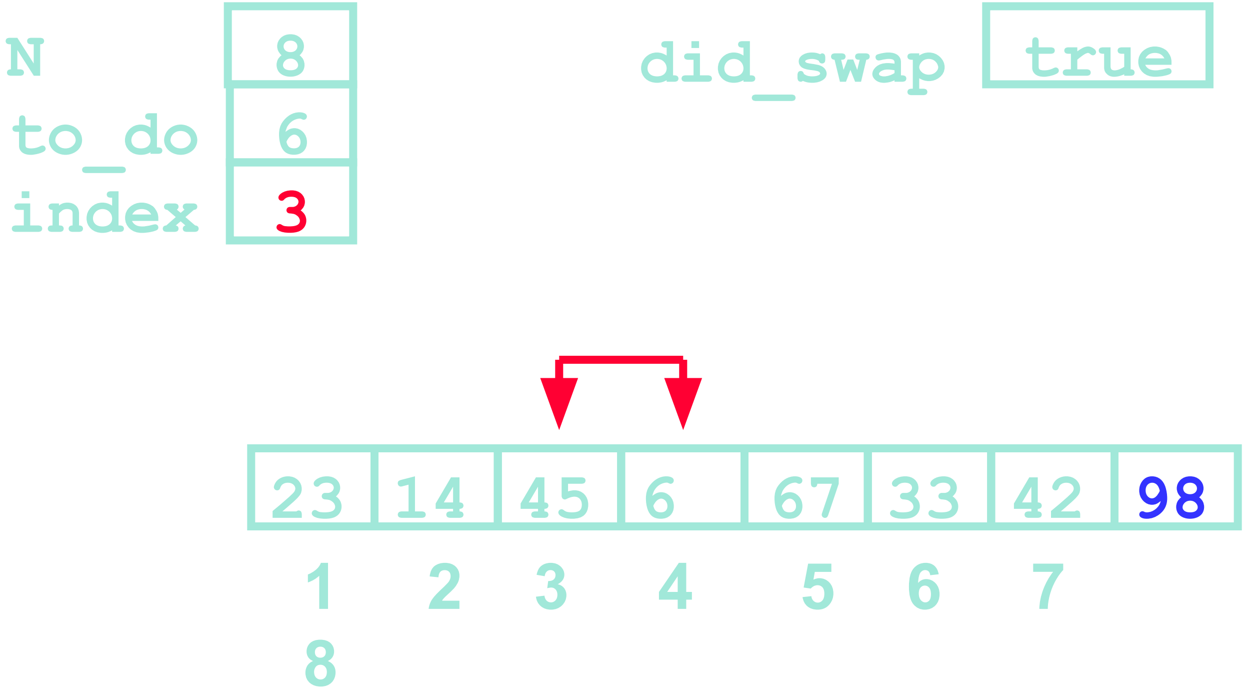
The Second “Bubble Up”



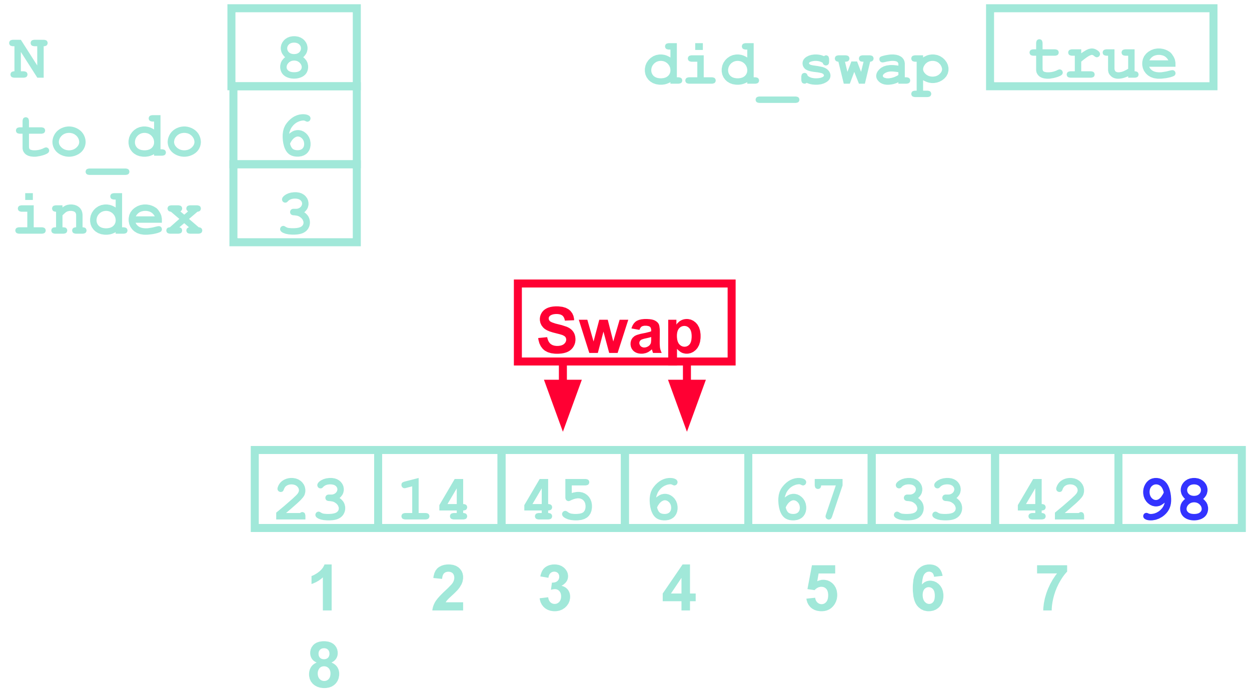
The Second “Bubble Up”



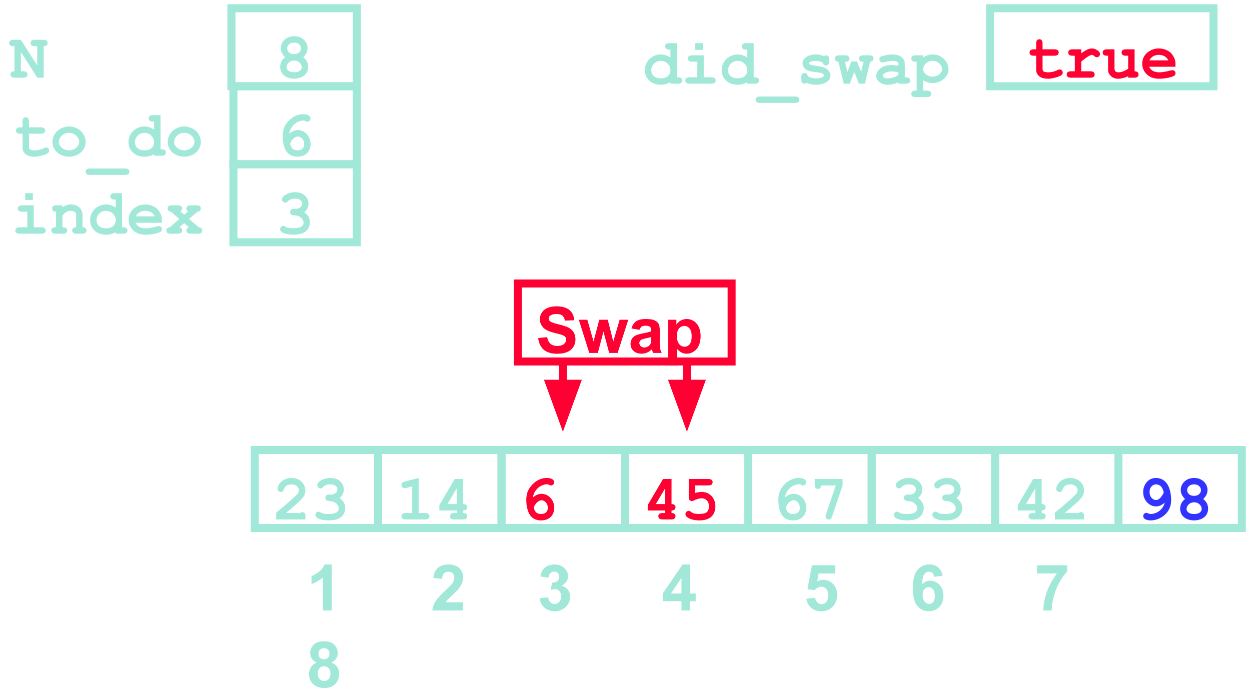
The Second “Bubble Up”



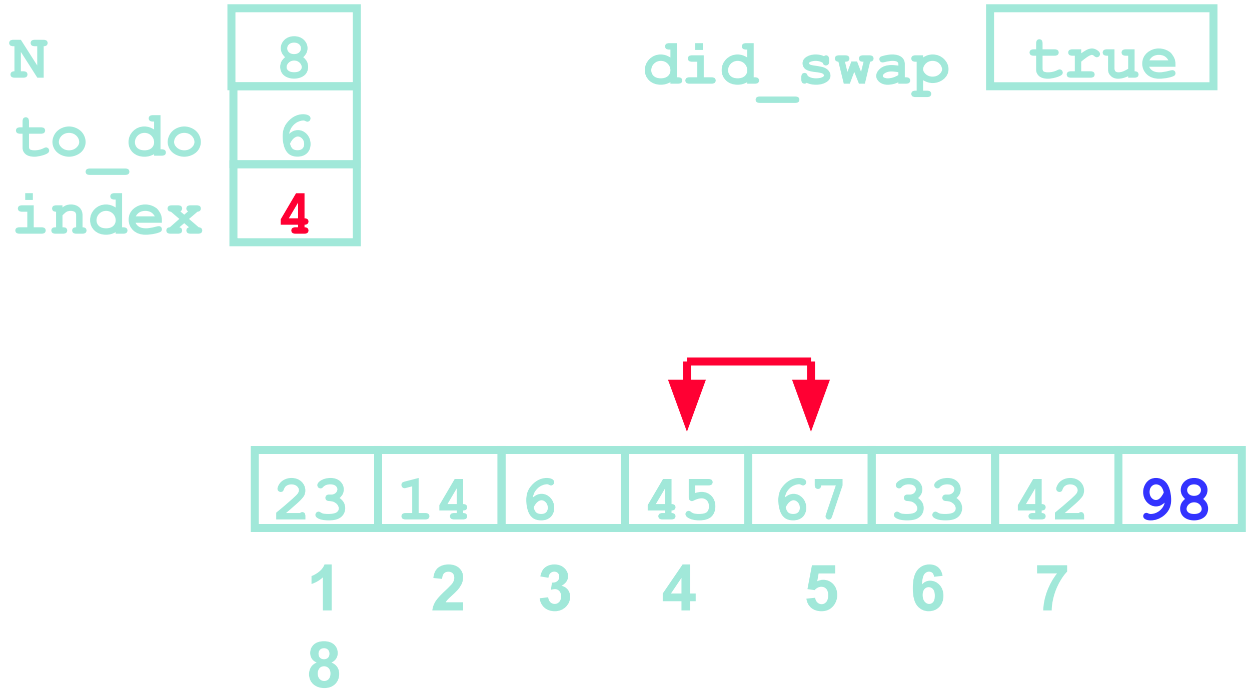
The Second “Bubble Up”



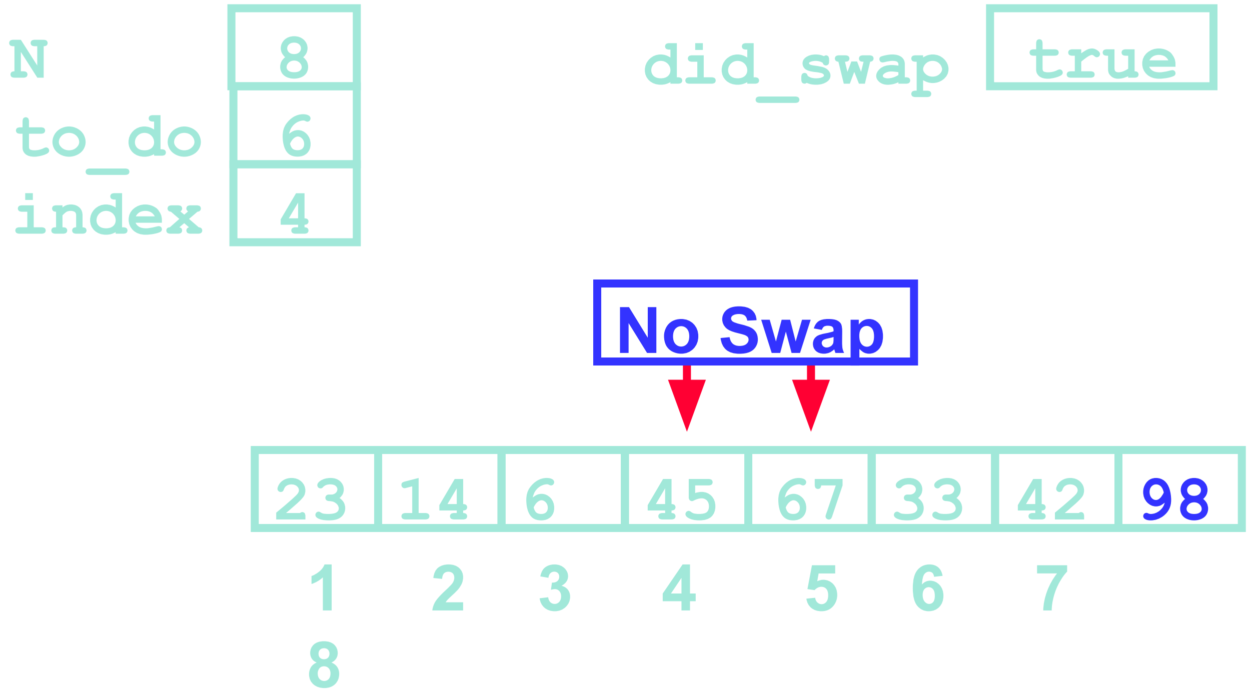
The Second "Bubble Up"



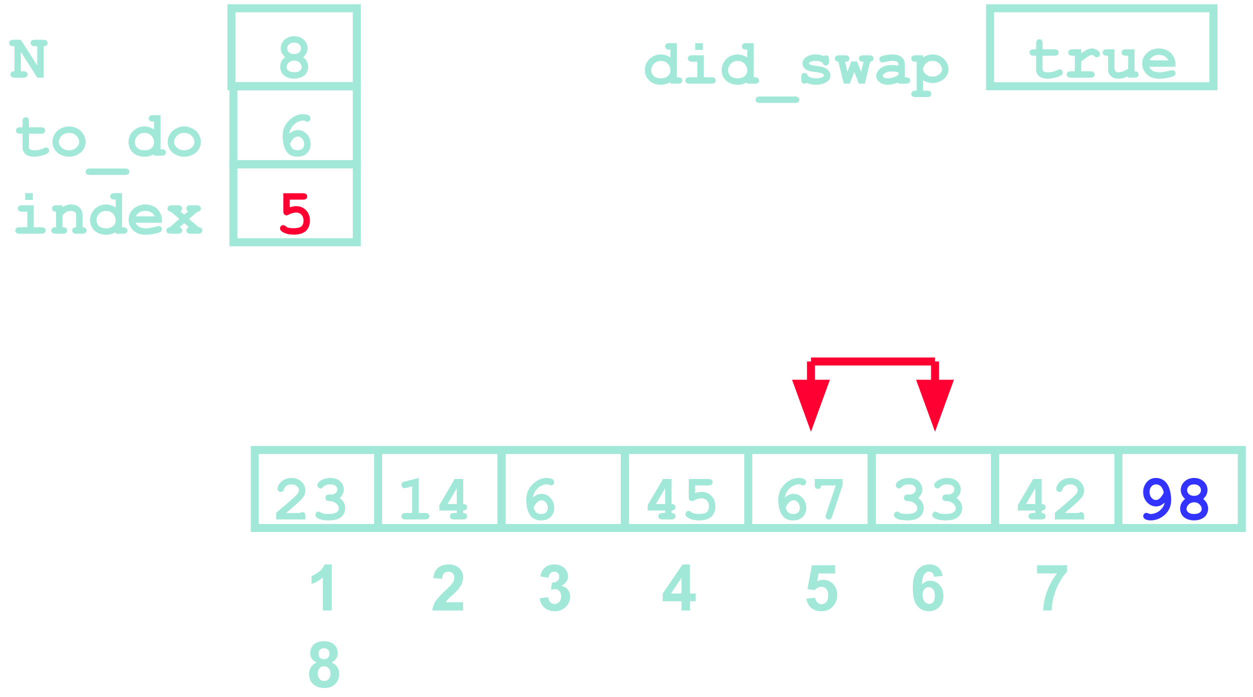
The Second "Bubble Up"



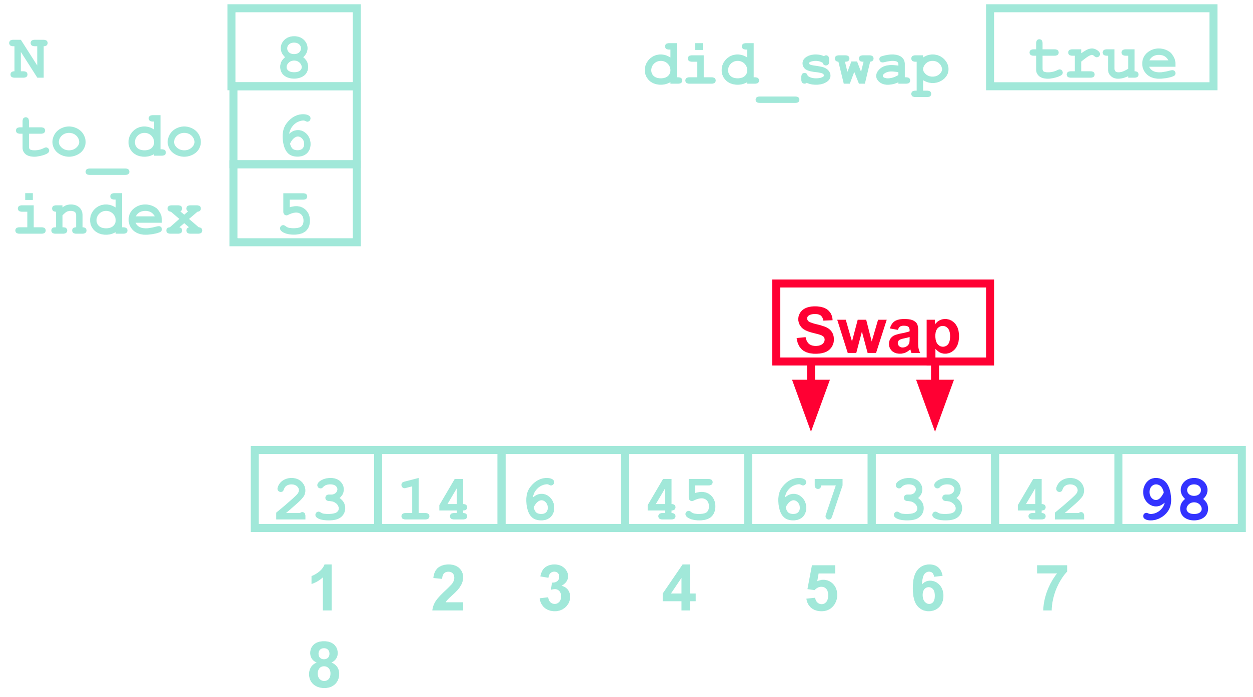
The Second "Bubble Up"



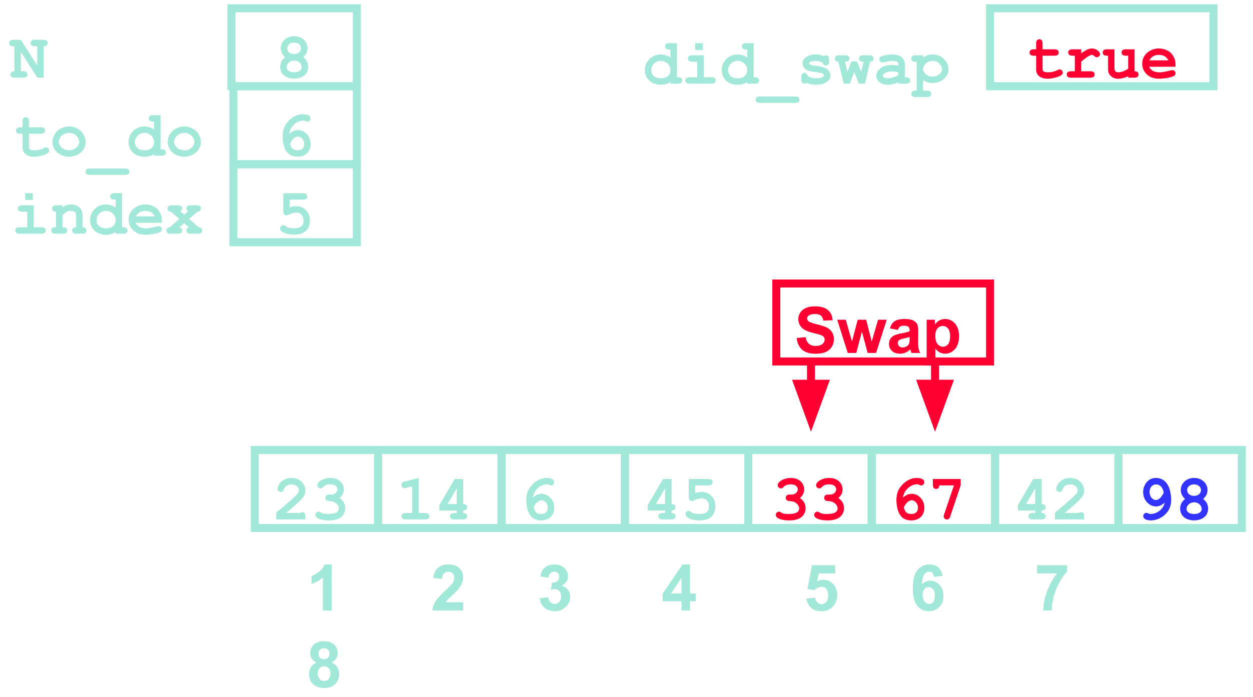
The Second "Bubble Up"



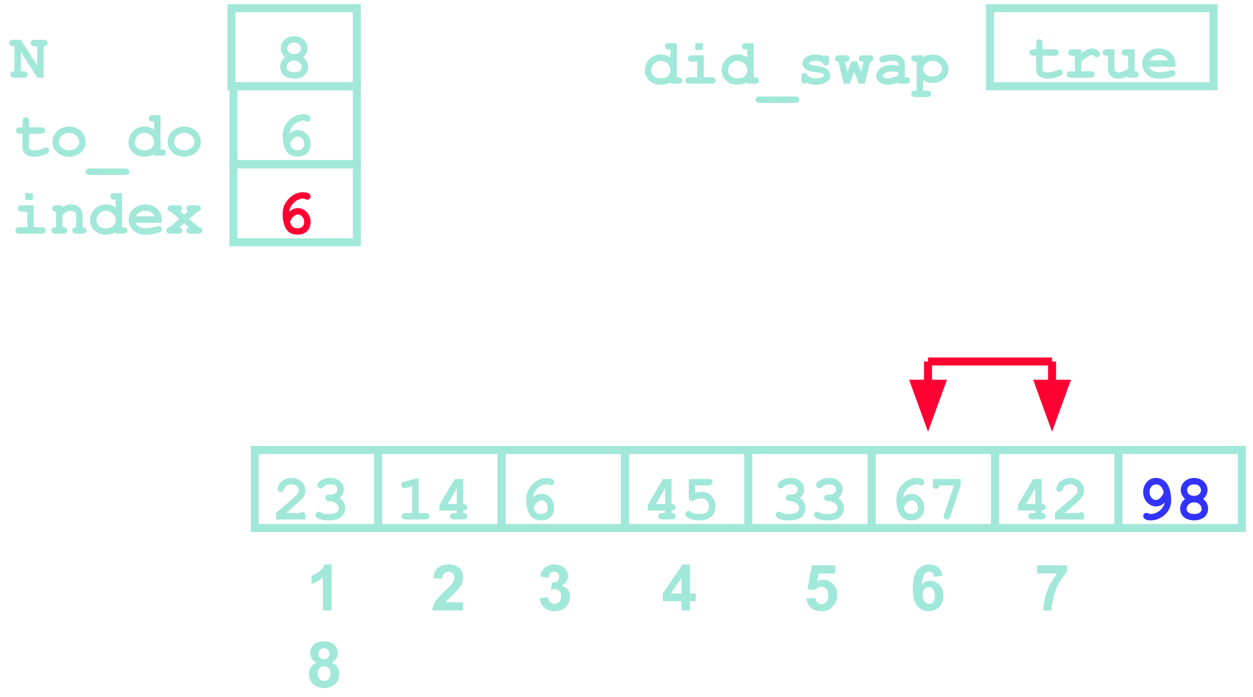
The Second "Bubble Up"



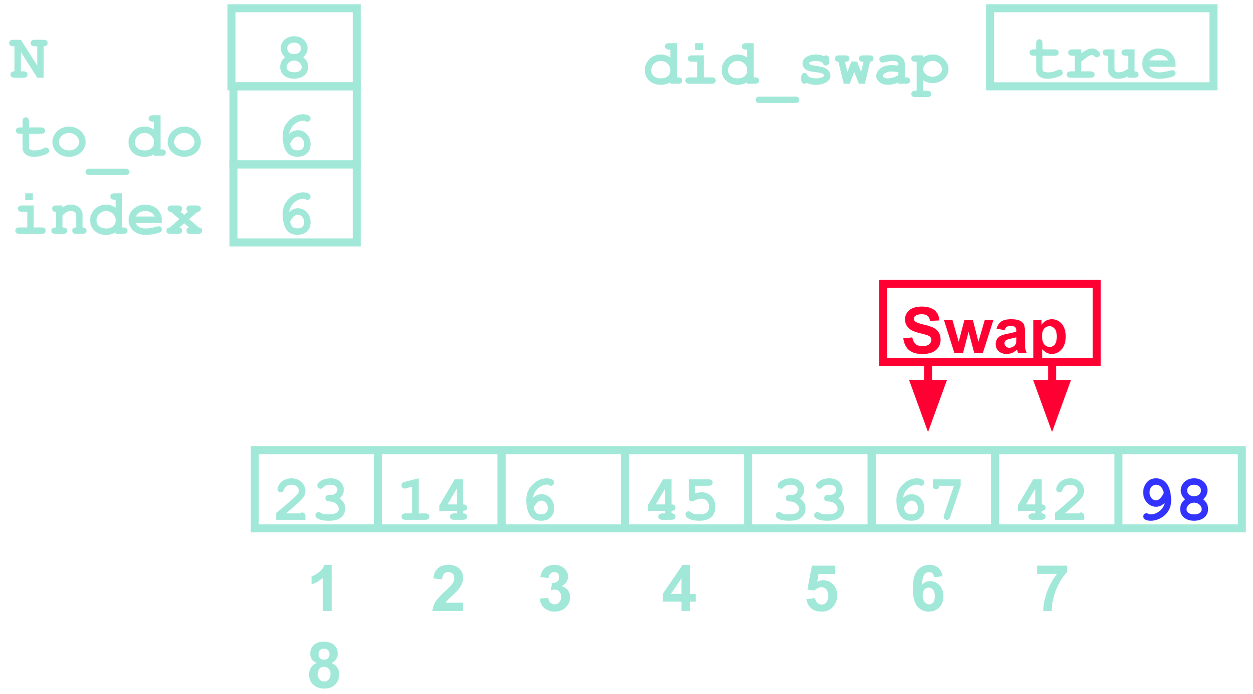
The Second "Bubble Up"



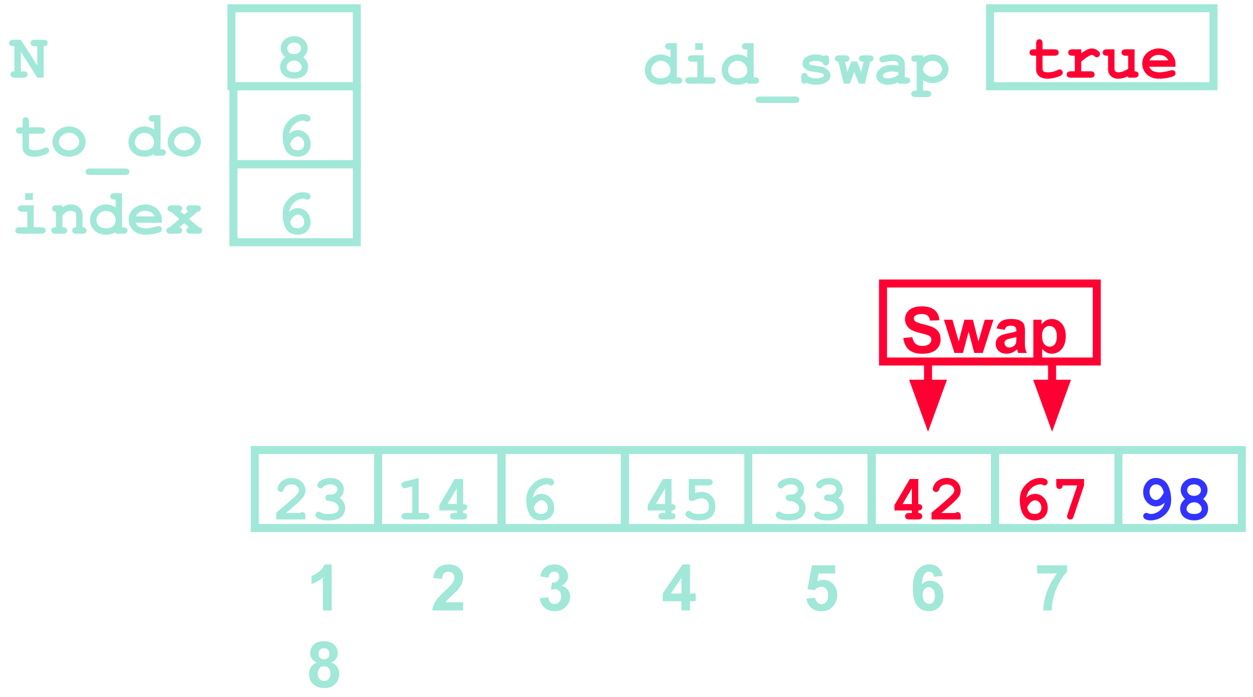
The Second "Bubble Up"



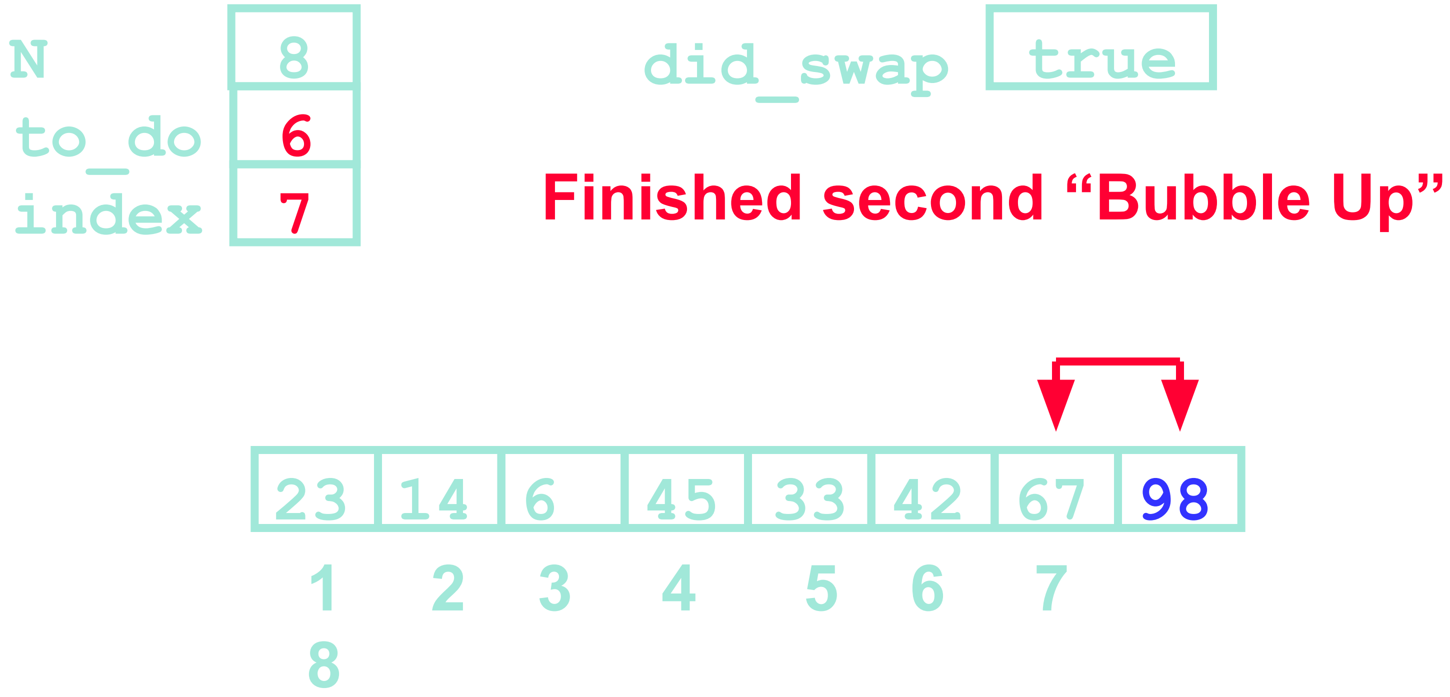
The Second “Bubble Up”



The Second “Bubble Up”



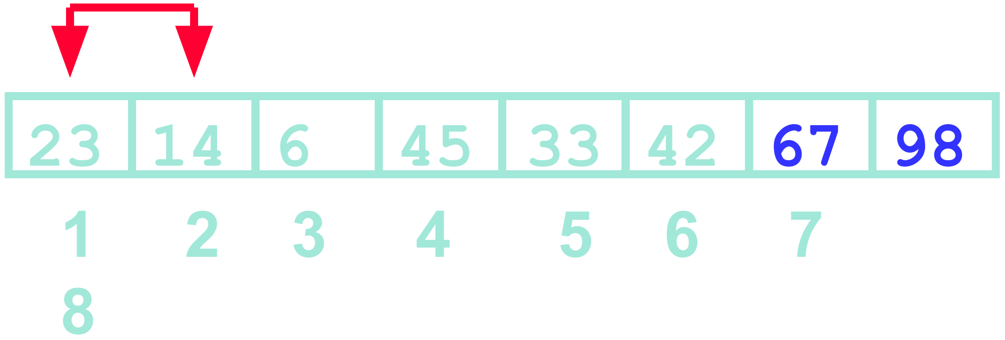
After Second Pass of Outer Loop



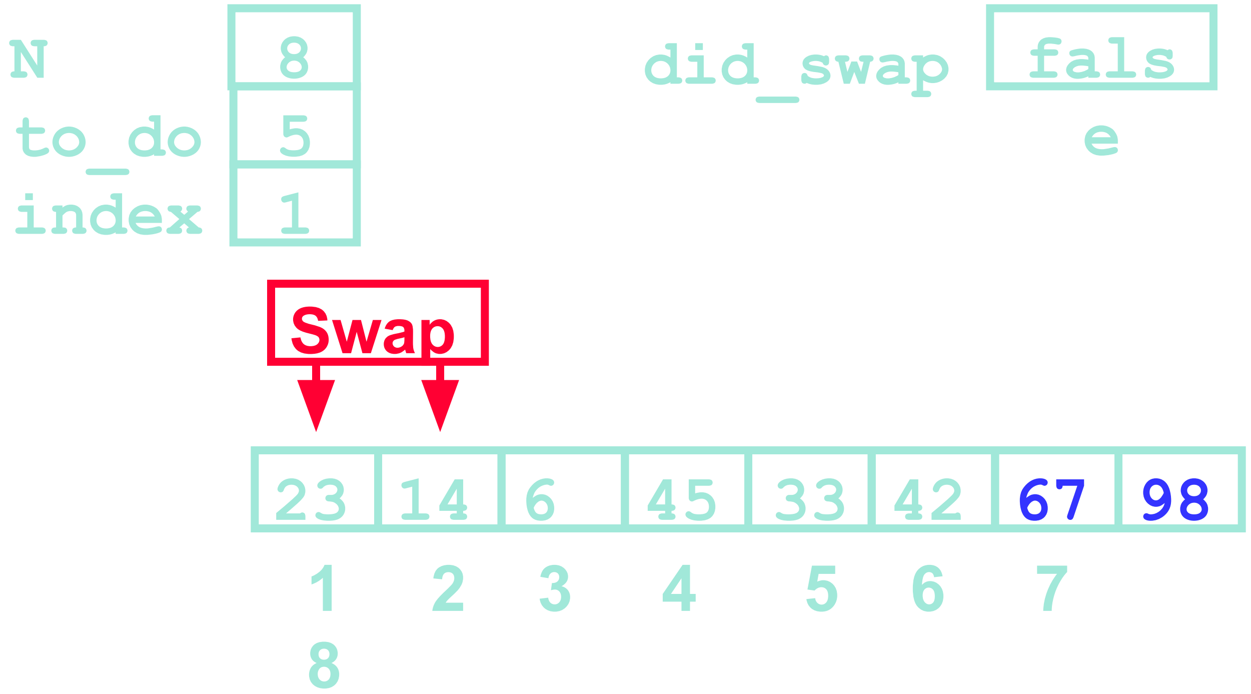
The Third "Bubble Up"

N	8
to_do	5
index	1

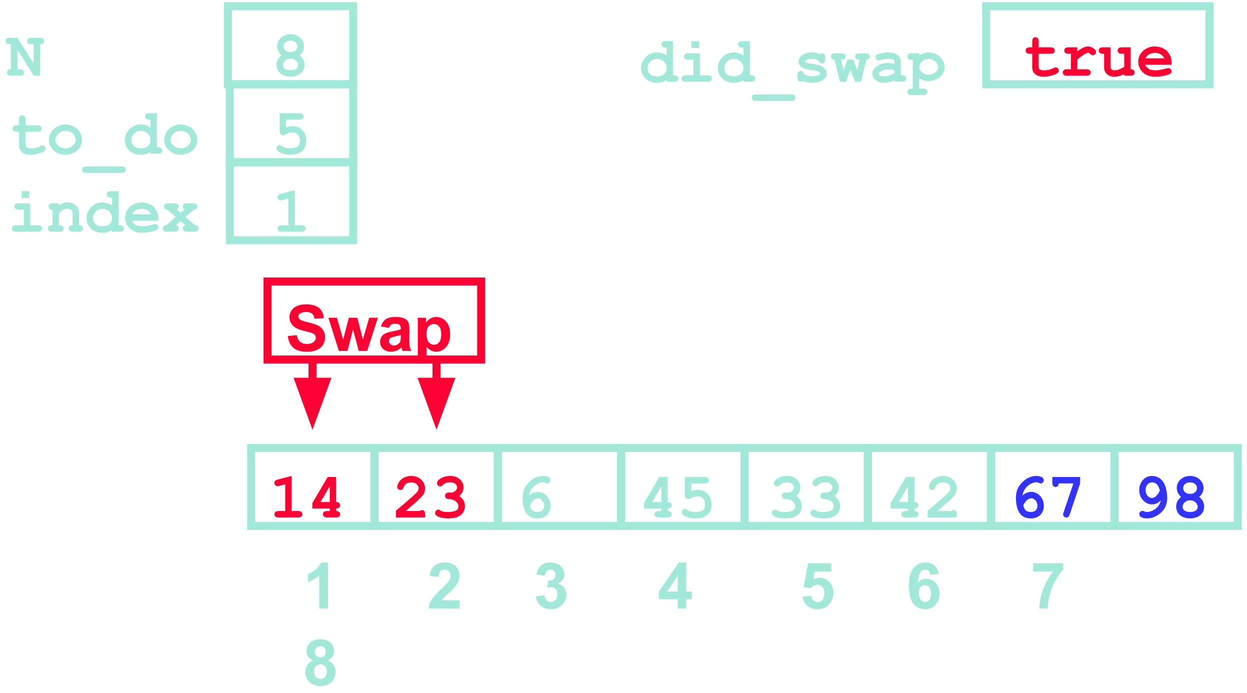
did_swap fals
e



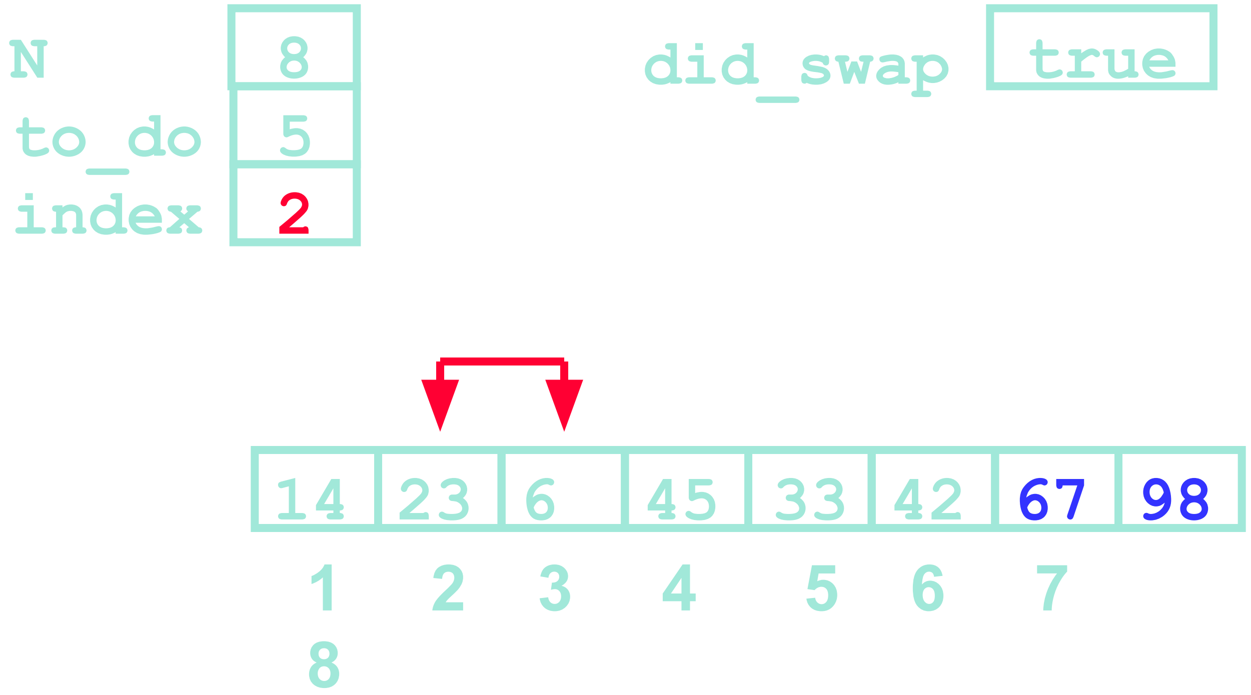
The Third “Bubble Up”



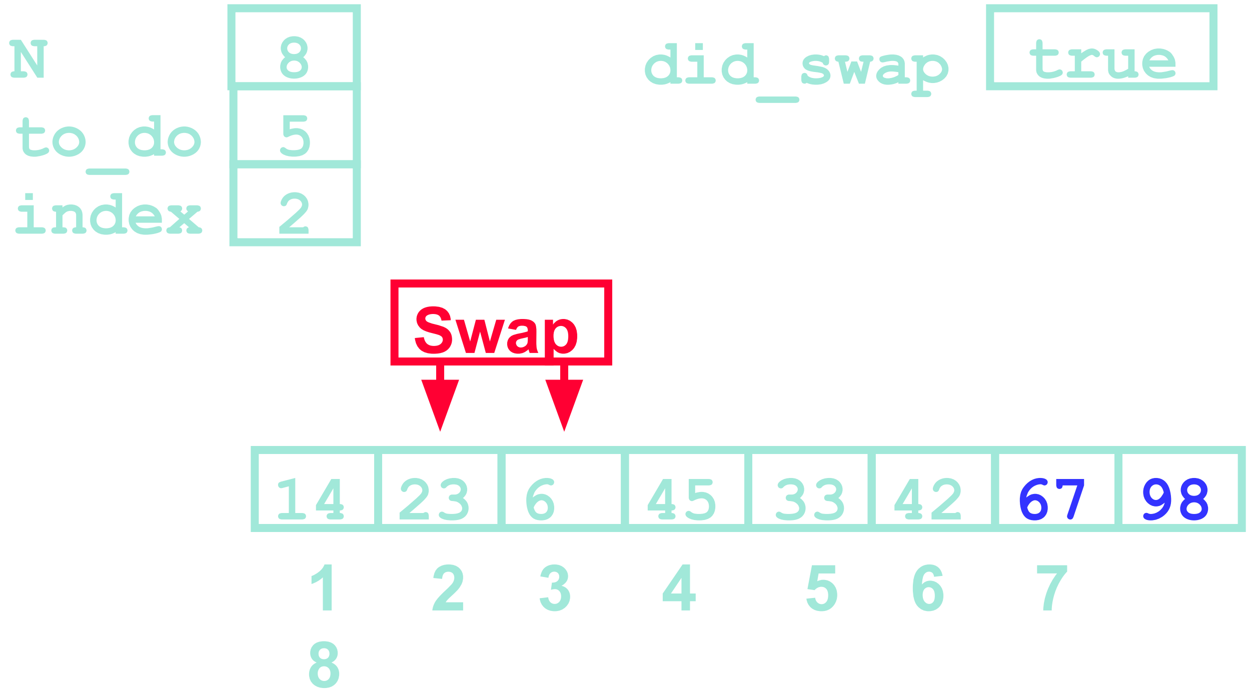
The Third "Bubble Up"



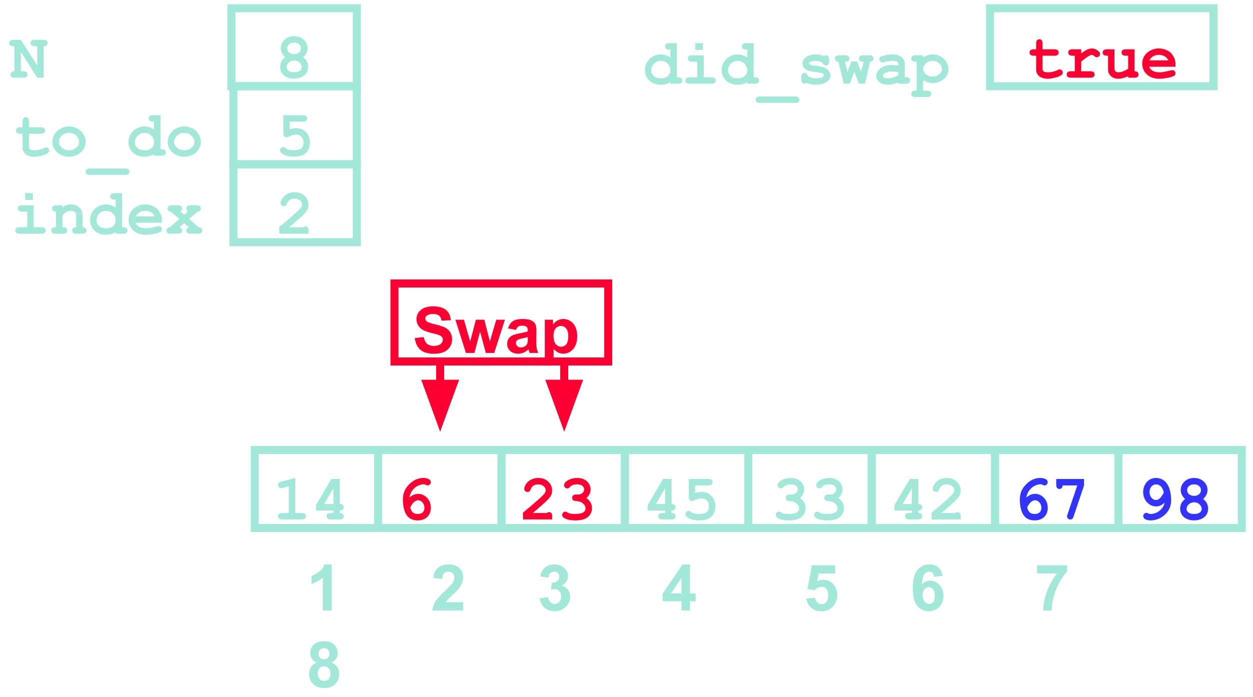
The Third “Bubble Up”



The Third "Bubble Up"



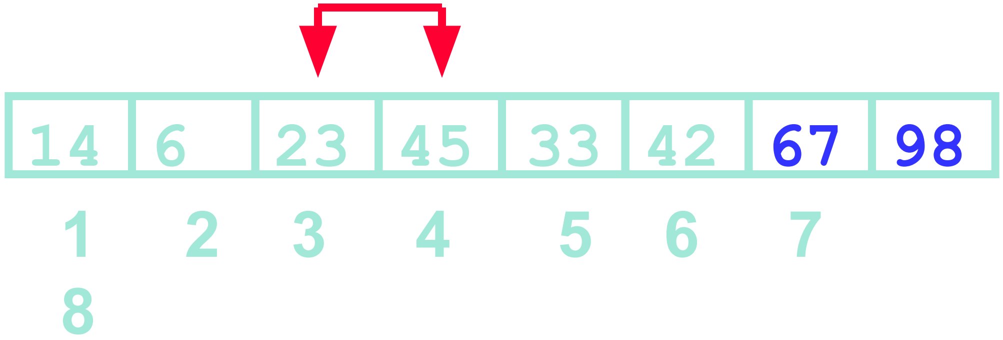
The Third "Bubble Up"



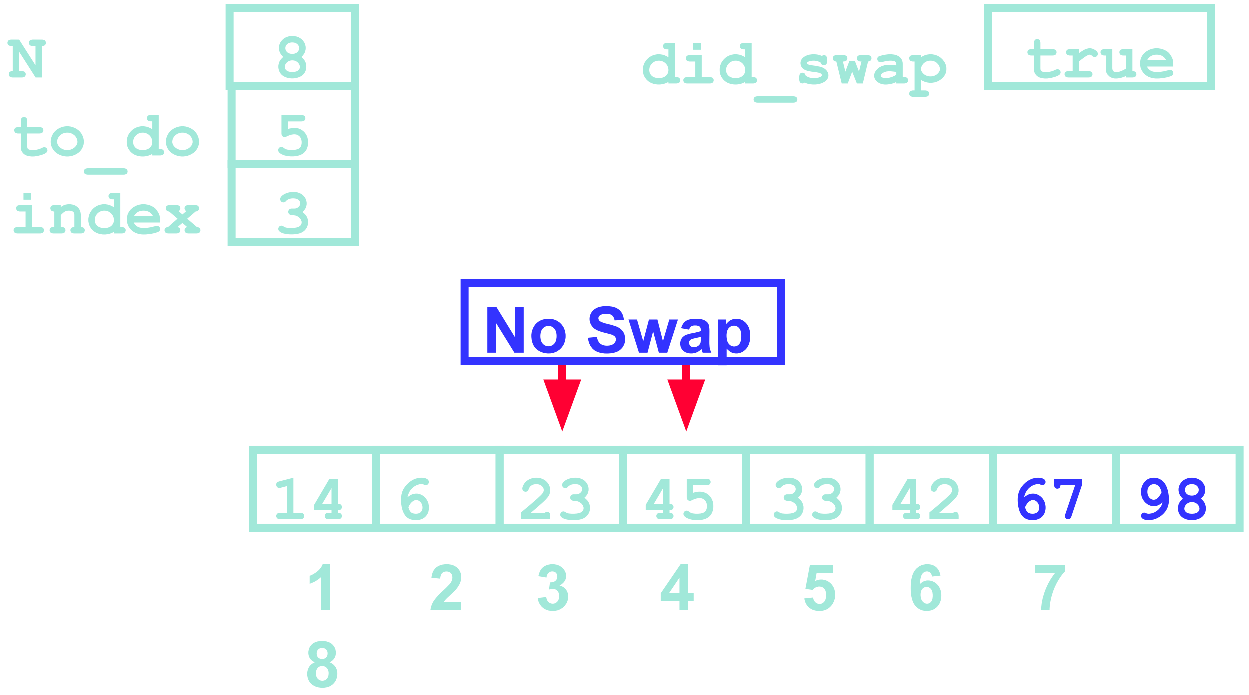
The Third "Bubble Up"

N	8
to_do	5
index	3

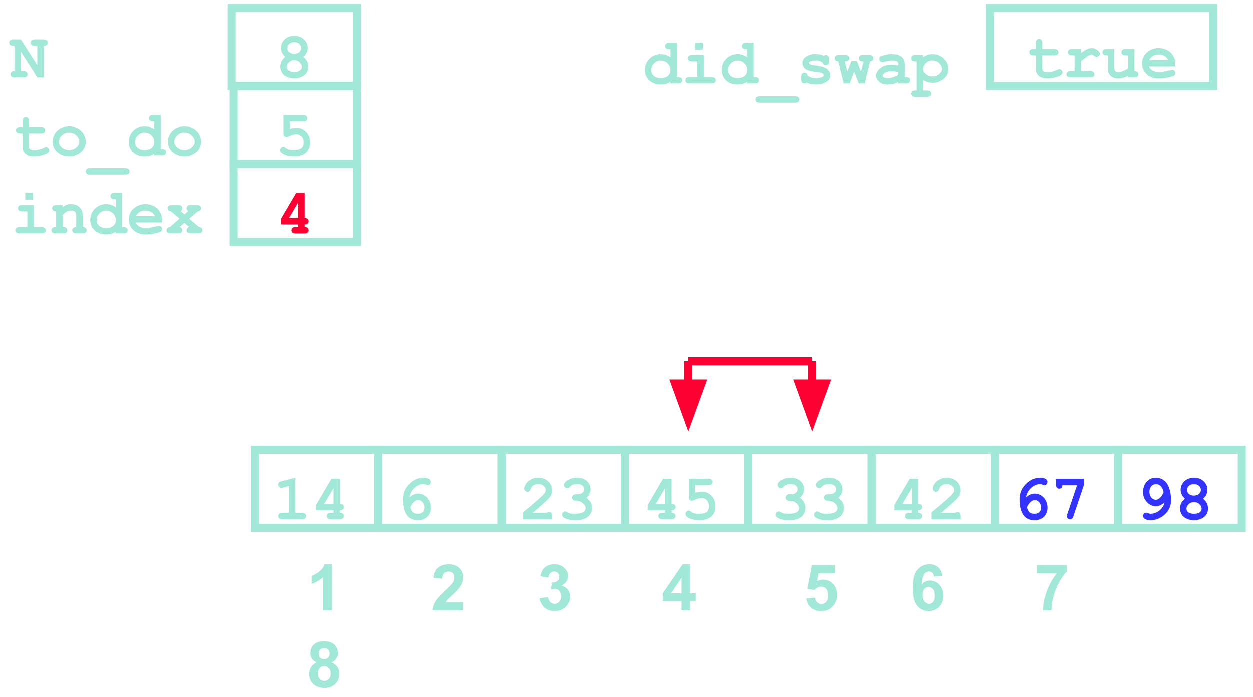
did_swap true



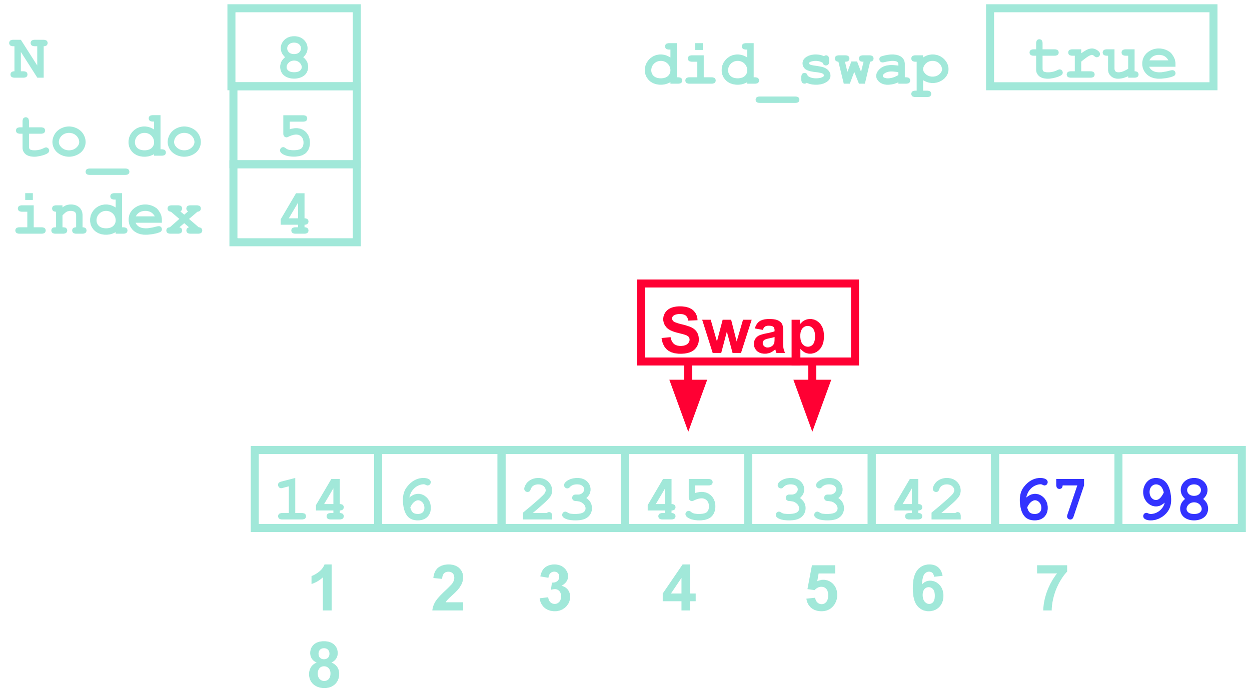
The Third “Bubble Up”



The Third “Bubble Up”



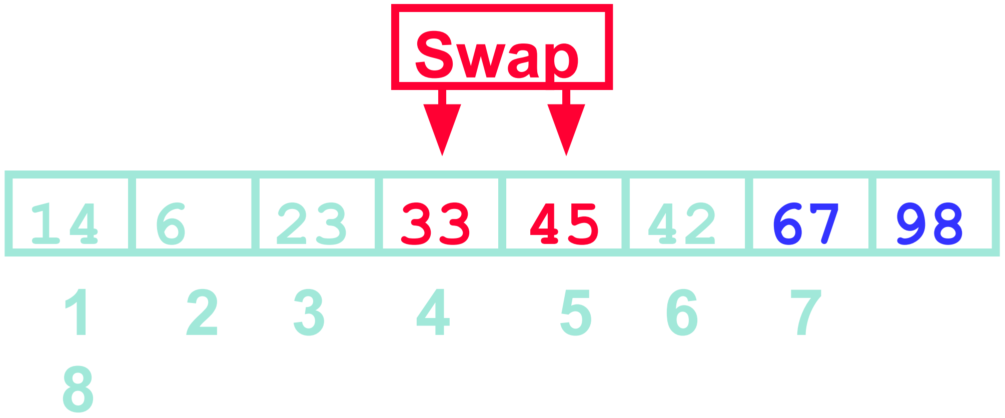
The Third “Bubble Up”



The Third “Bubble Up”

N	8
to_do	5
index	4

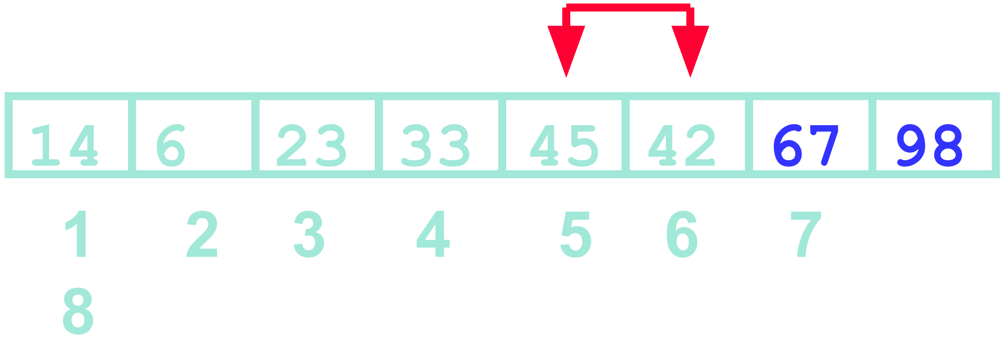
did_swap	true



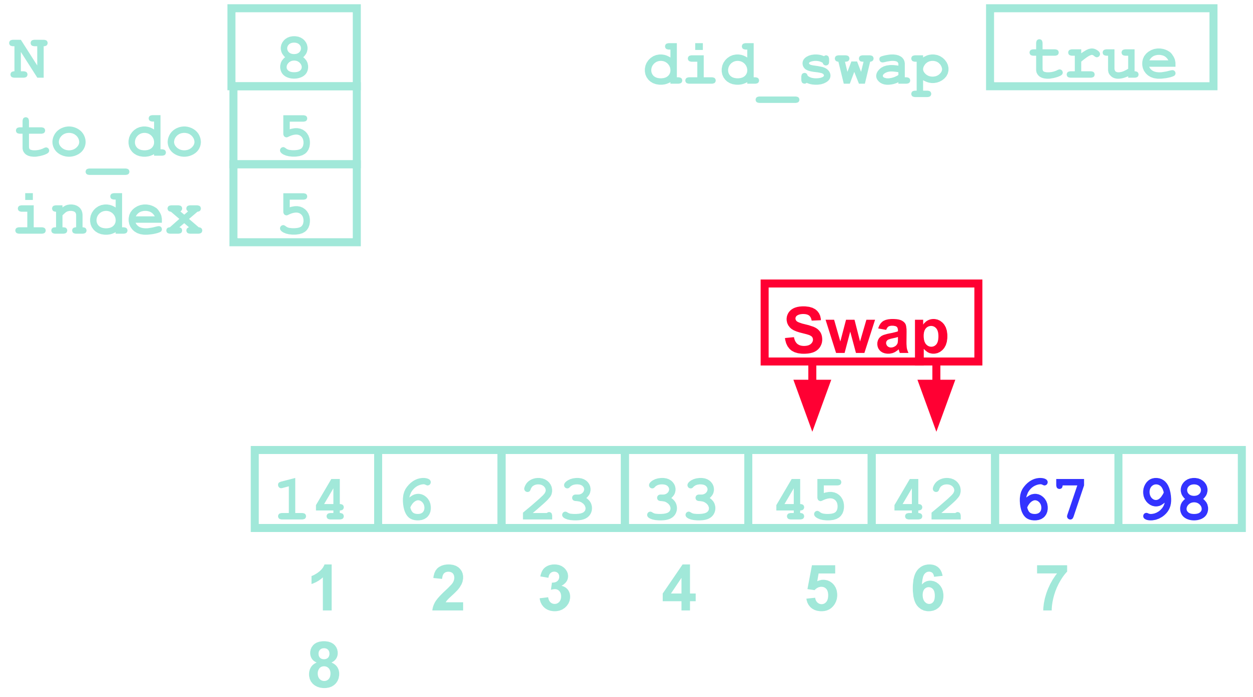
The Third “Bubble Up”

N	8
to_do	5
index	5

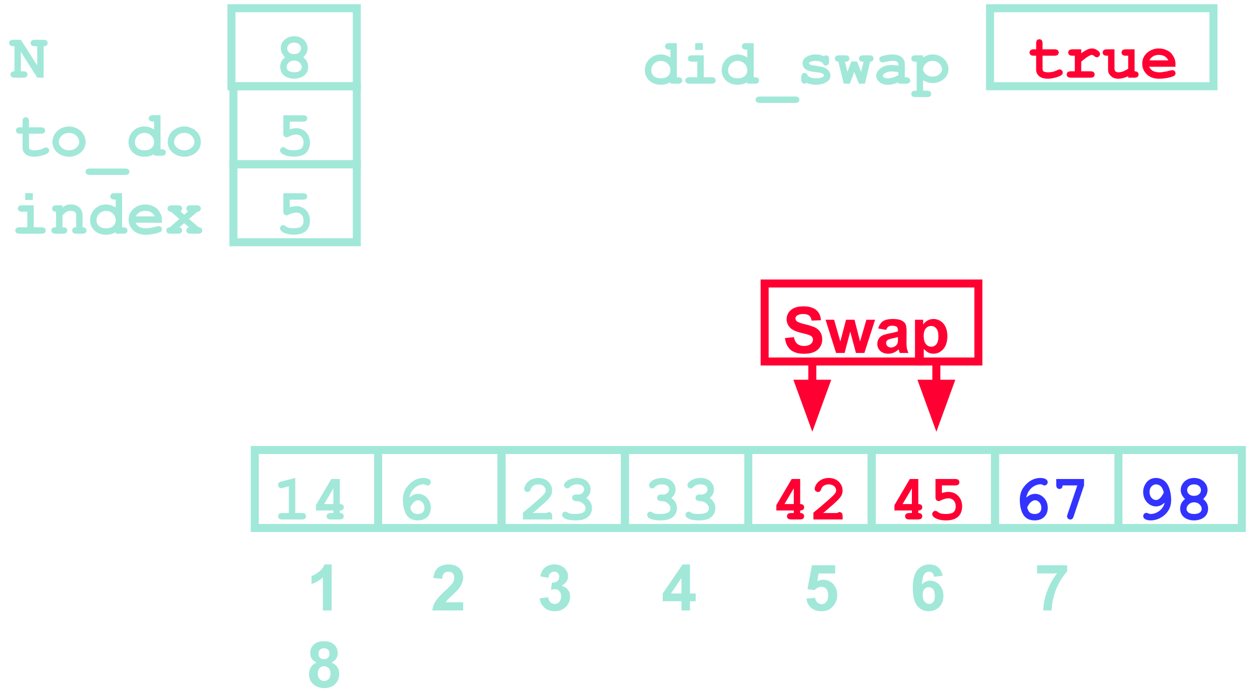
did_swap	true



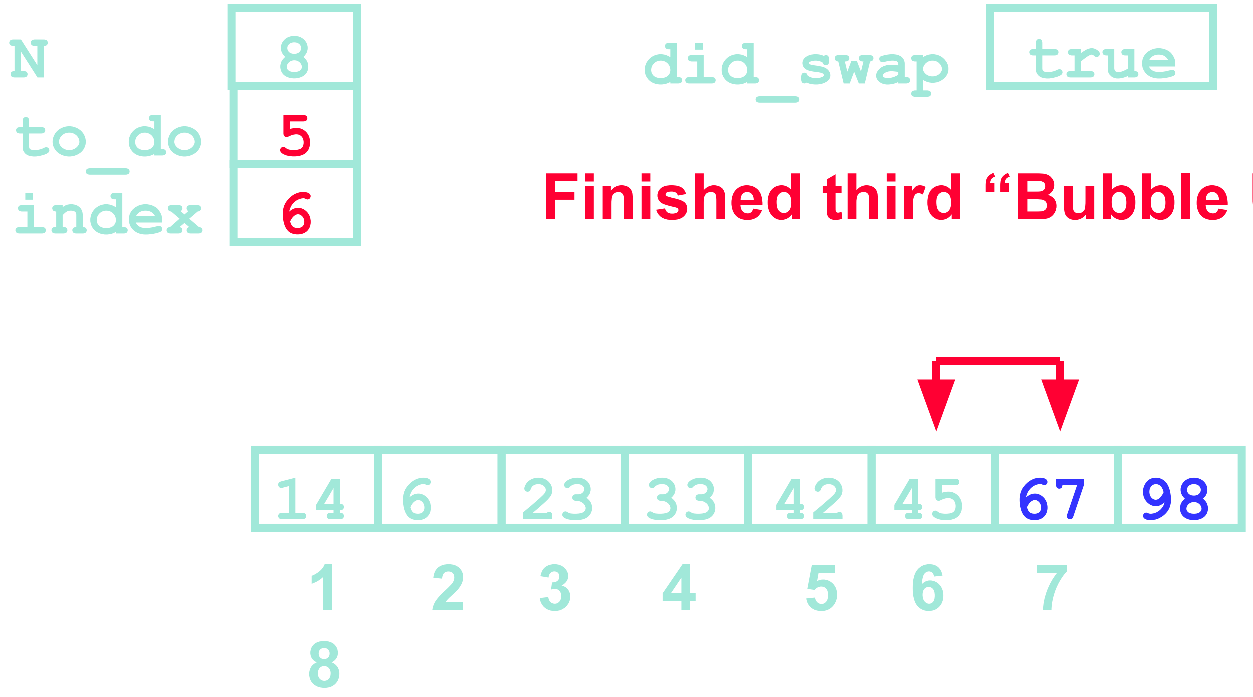
The Third "Bubble Up"



The Third “Bubble Up”



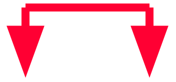
After Third Pass of Outer Loop



The Fourth “Bubble Up”

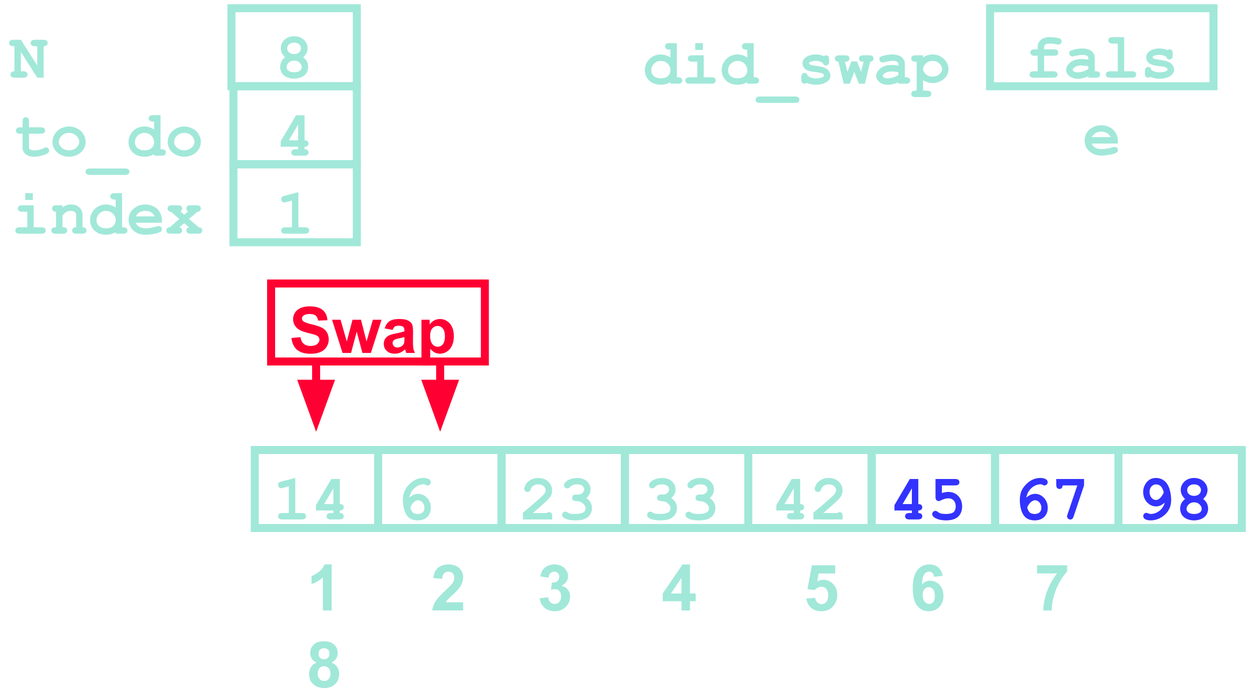
N	8
to_do	4
index	1

did_swap	fals e

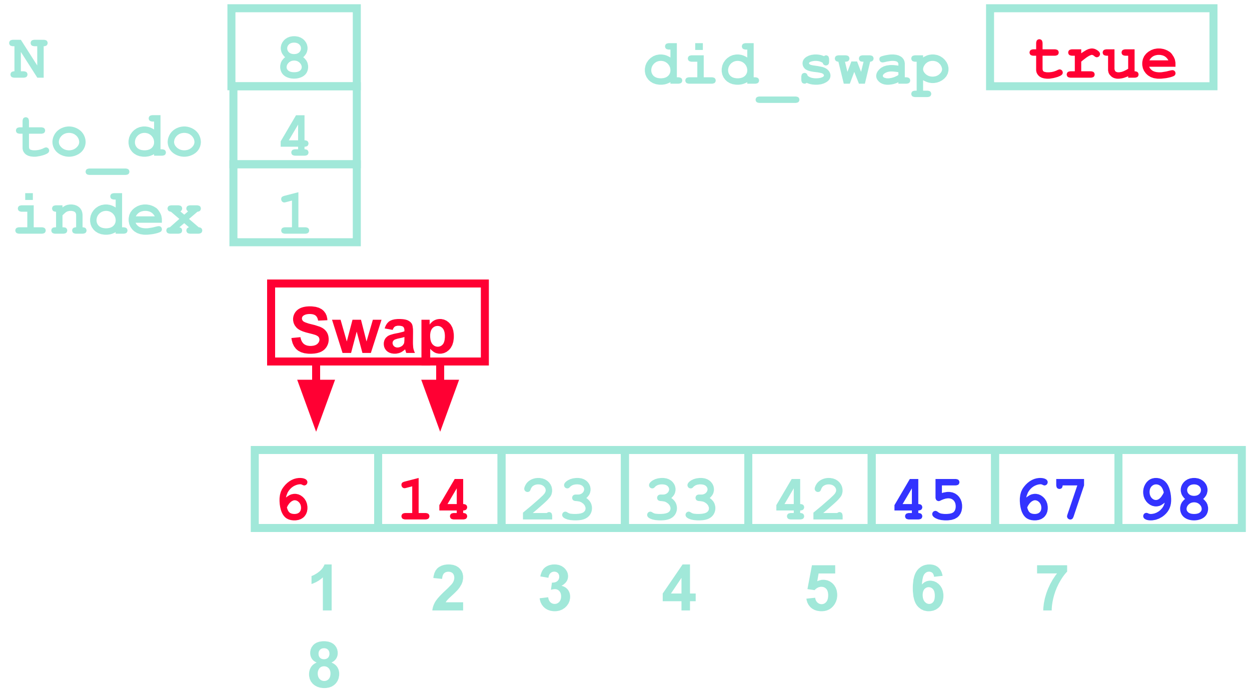


14	6	23	33	42	45	67	98
1	2	3	4	5	6	7	8

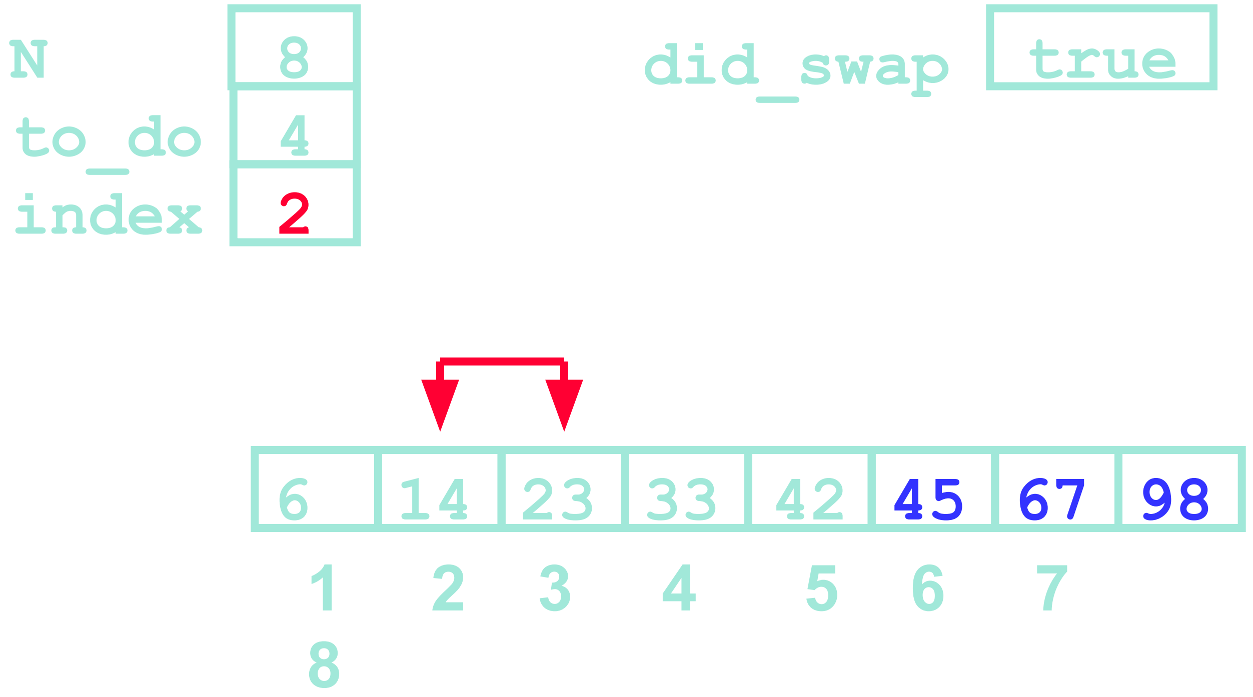
The Fourth “Bubble Up”



The Fourth “Bubble Up”



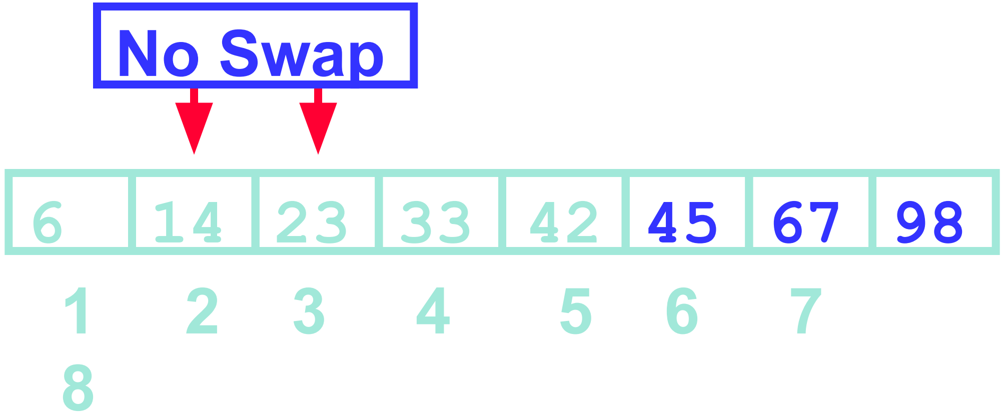
The Fourth "Bubble Up"



The Fourth "Bubble Up"

N	8
to_do	4
index	2

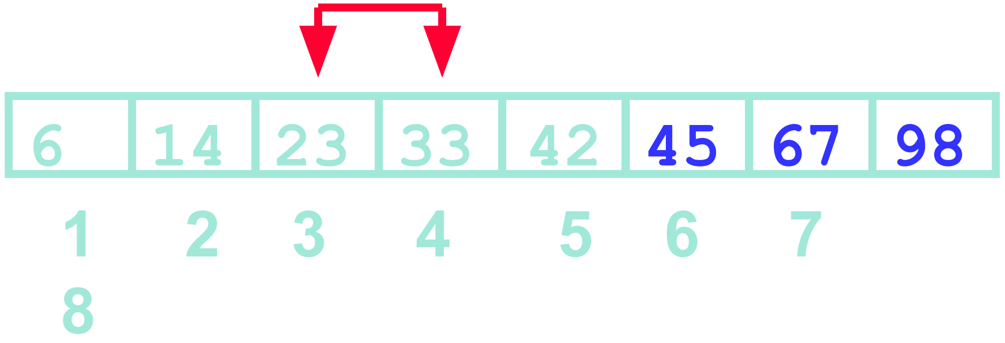
did_swap	true



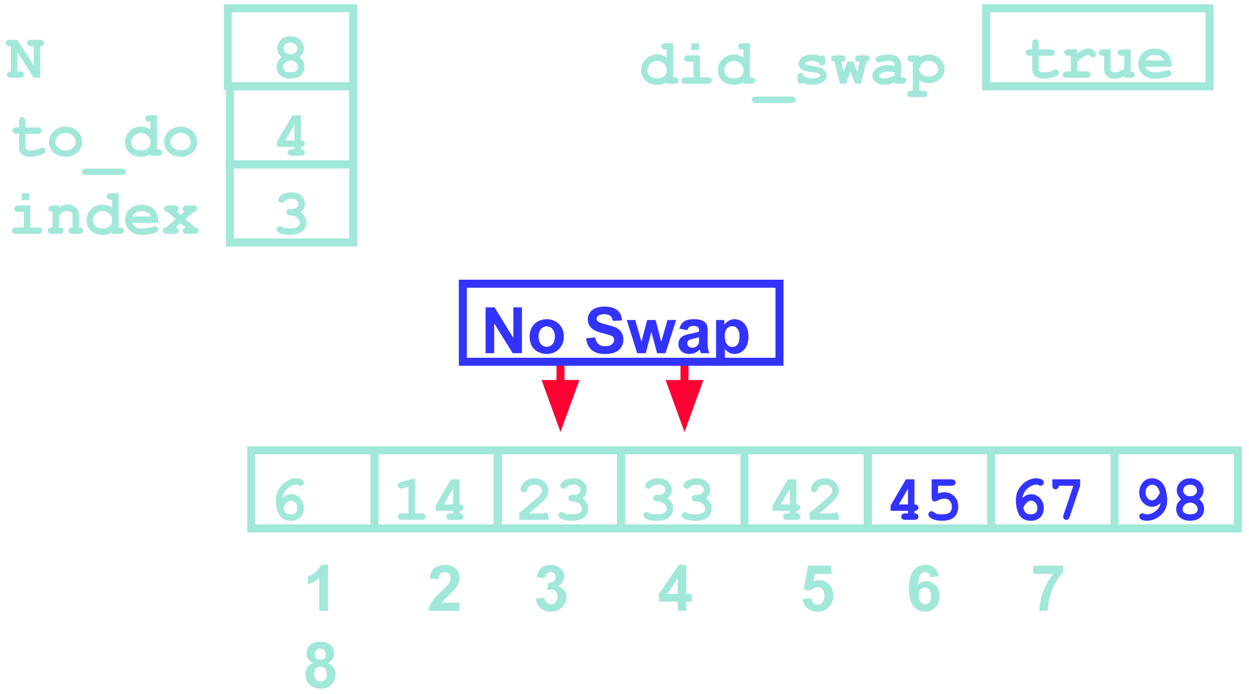
The Fourth “Bubble Up”

N	8
to_do	4
index	3

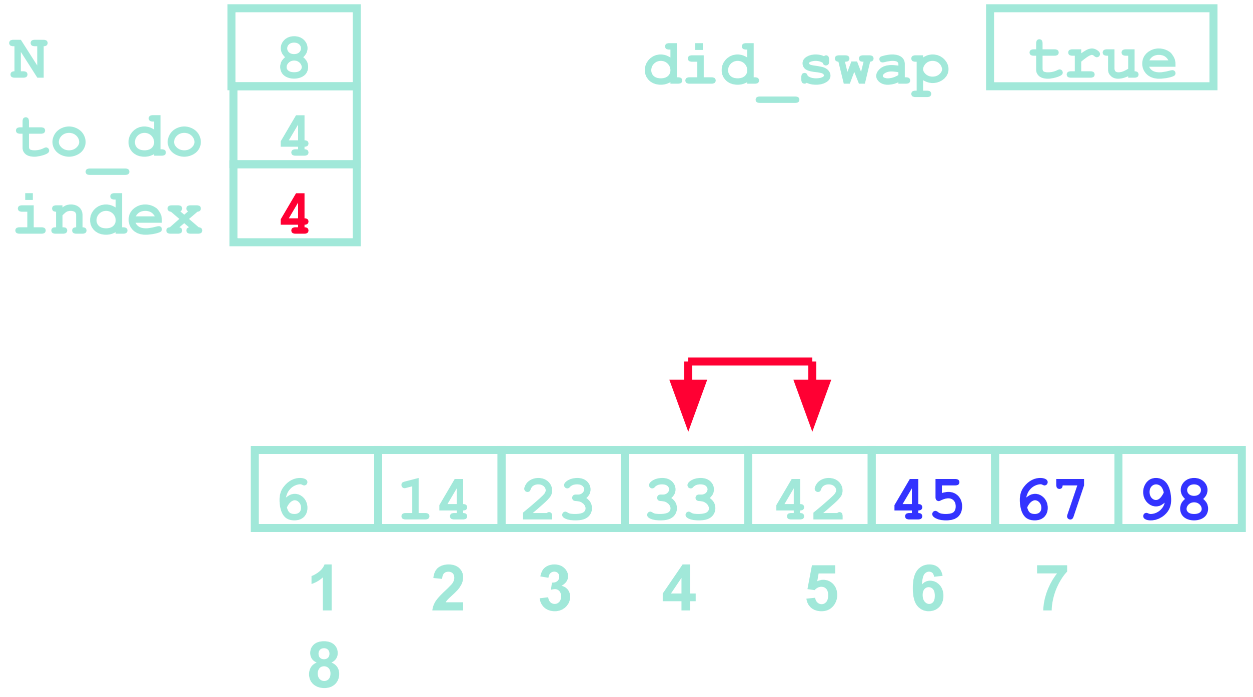
did_swap	true



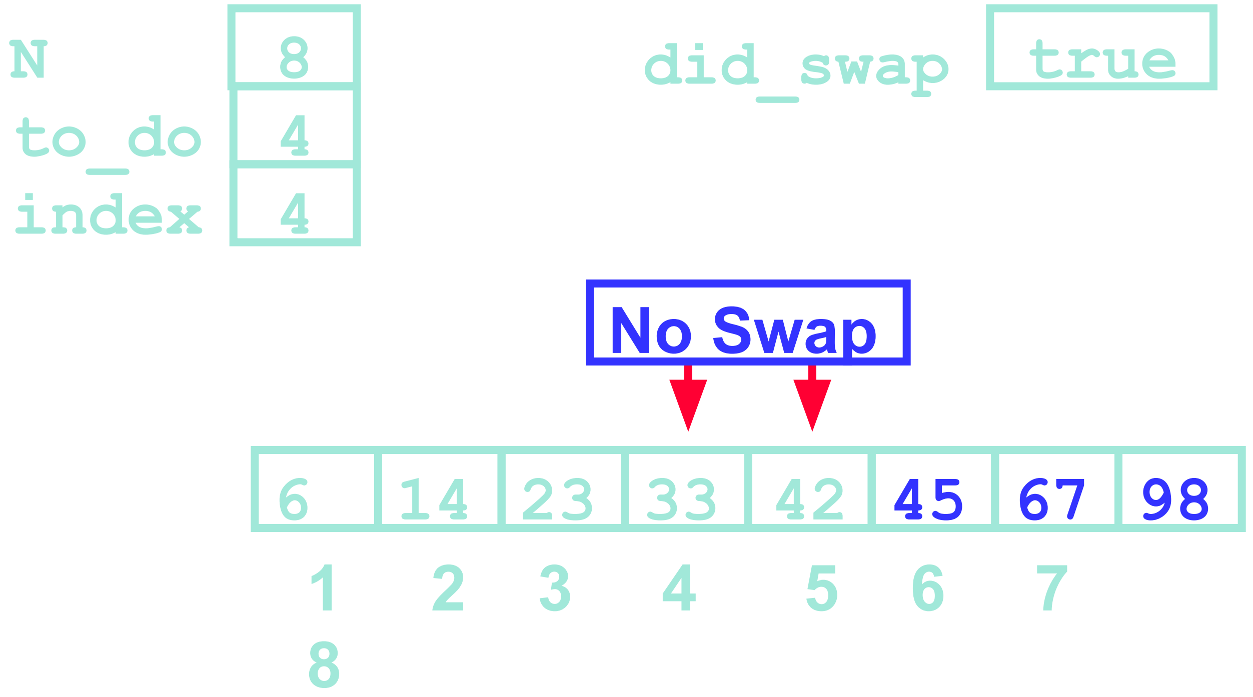
The Fourth "Bubble Up"



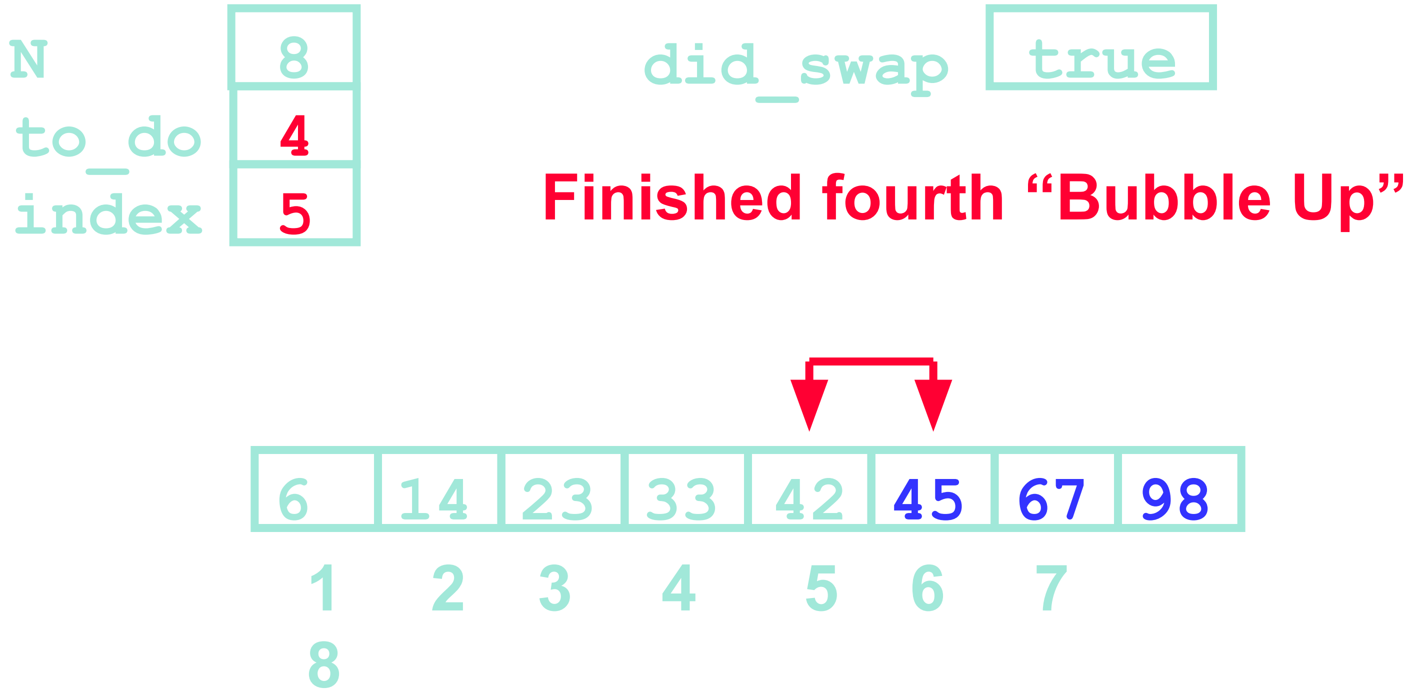
The Fourth “Bubble Up”



The Fourth "Bubble Up"



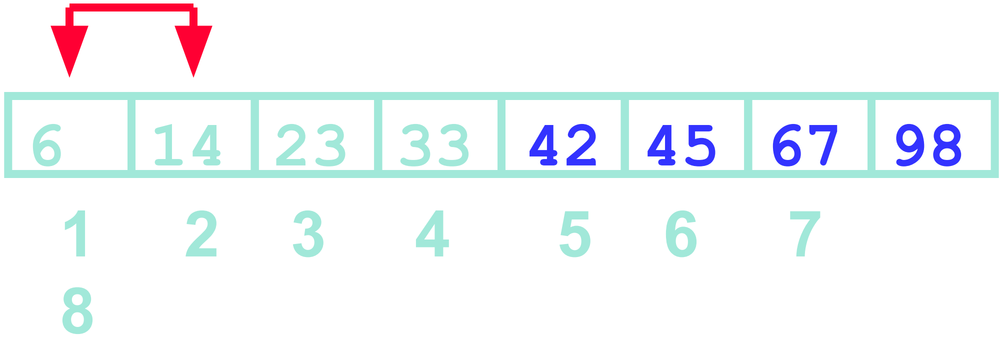
After Fourth Pass of Outer Loop



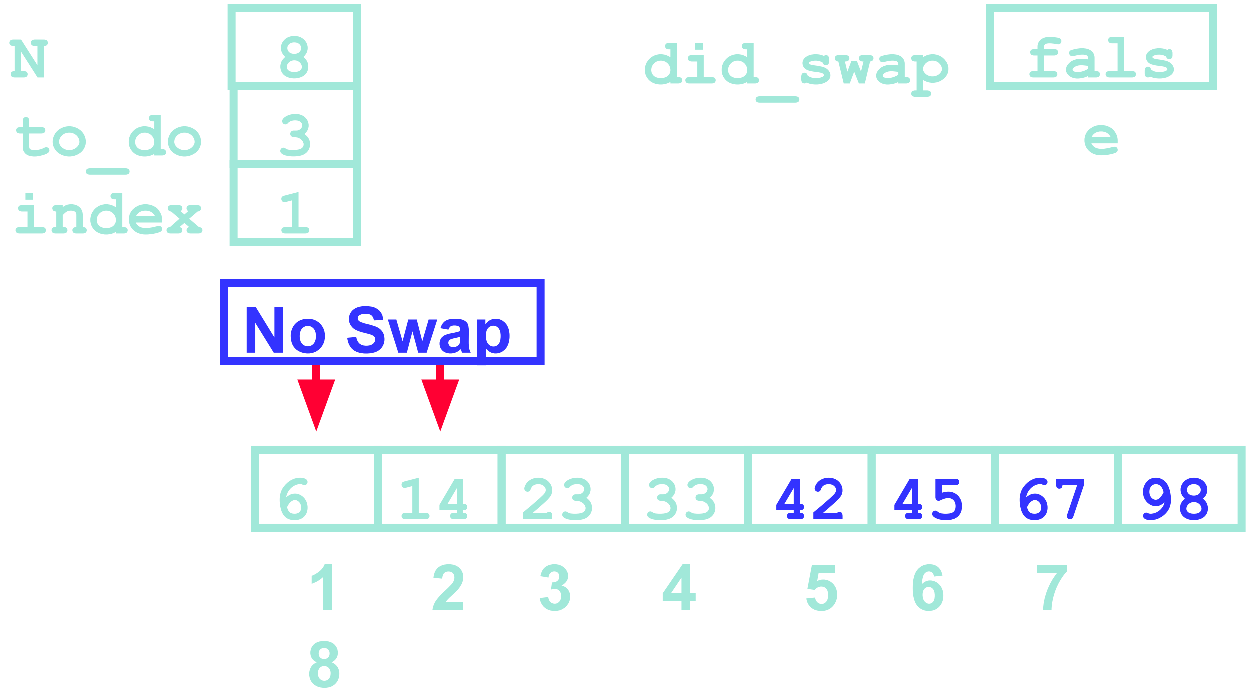
The Fifth “Bubble Up”

N	8
to_do	3
index	1

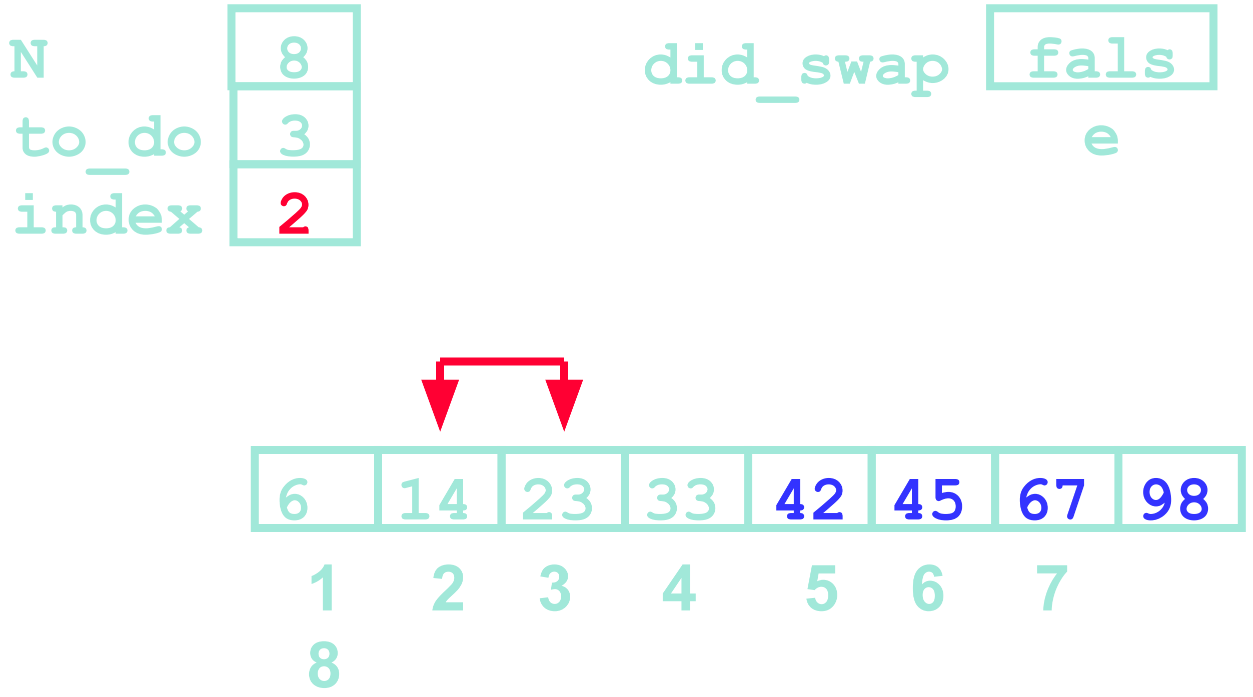
did_swap fals
e



The Fifth "Bubble Up"



The Fifth “Bubble Up”



The Fifth “Bubble Up”

N	8
to_do	3
index	2

did_swap fals e

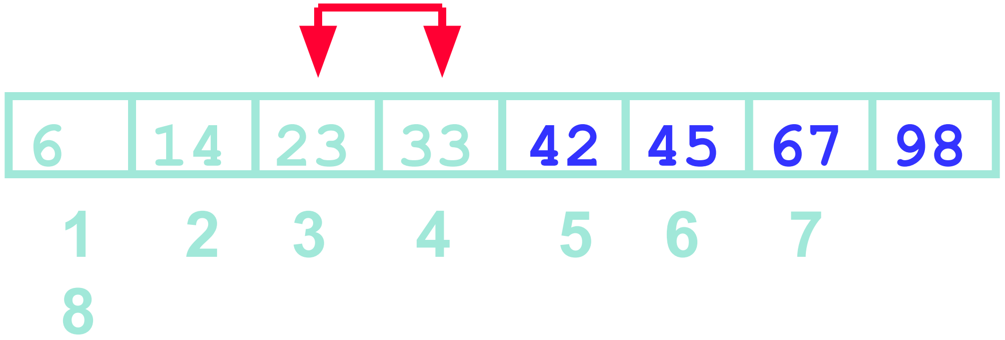
No Swap

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

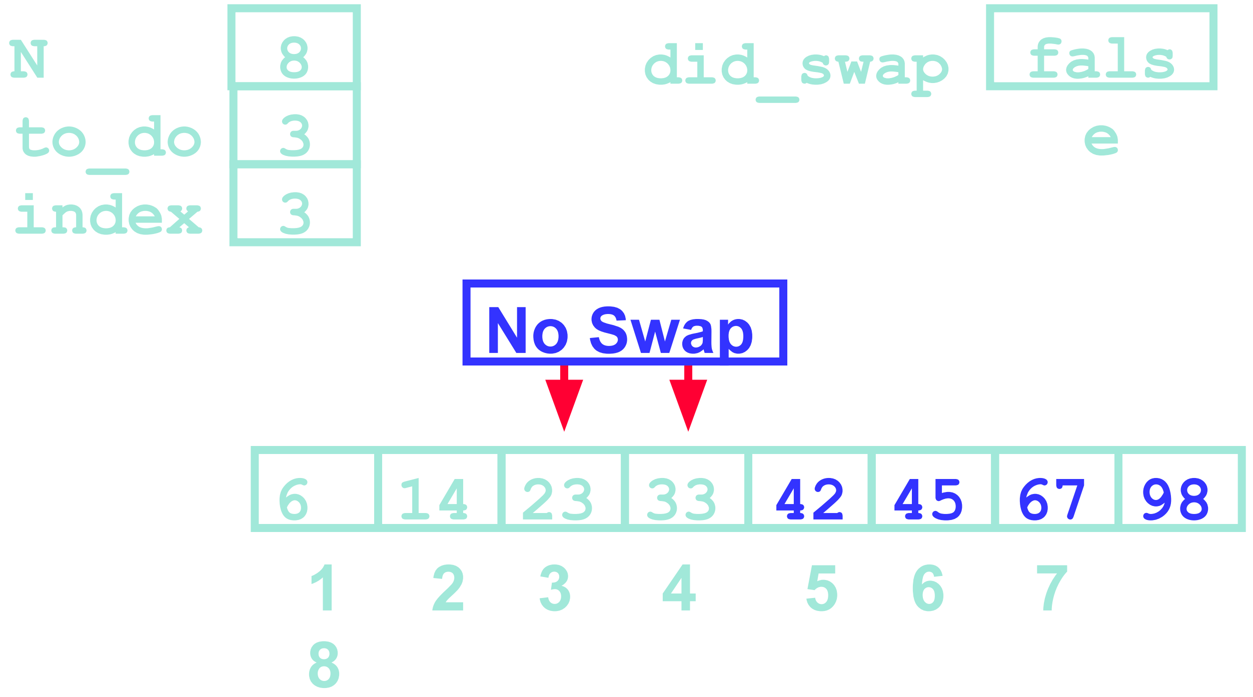
The Fifth "Bubble Up"

N	8
to_do	3
index	3

did_swap fals
e



The Fifth “Bubble Up”



After Fifth Pass of Outer Loop

N	8
to_do	3
index	4

did_swap fals
e

Finished fifth “Bubble Up”

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

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.

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

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

Truth in CS Act

- NOBODY EVER USES BUBBLE SORT
- NOBODY
- NOT EVER
- BECAUSE IT IS EXTREMELY INEFFICIENT

Selection Sort

Insertion Sort

Insertion Sort Summary

Pro’s:

- Easy to code
- Fast on small inputs (less than ~50 elements)
- Fast on nearly-sorted inputs
- In-place sorting

Con’s:

- O(n^2) worst case
- O(n^2) average case
- O(n^2) reverse-sorted

Heap Sort

Heap Sort Summary

Uses the very useful heap data structure

Complete binary tree

Heap property: parent key > children’s keys

Pro’s:

- O(n lg n) worst case - asymptotically optimal for comparison sorts
- Sorts in place

Con’s:

- Fair amount of shuffling memory around

Counting Sort

Introduction

Many times we have restrictions on our keys

- Deck of cards: Ace->King and four suites
- Social Security Numbers
- Employee ID's

We will examine three algorithms which under certain conditions can run in O(n) time.

- Counting sort
- Radix sort
- Bucket sort

Fundamentals of Counting Sort

Depends on assumption about the numbers being sorted

- Assume numbers are in the range 1.. k

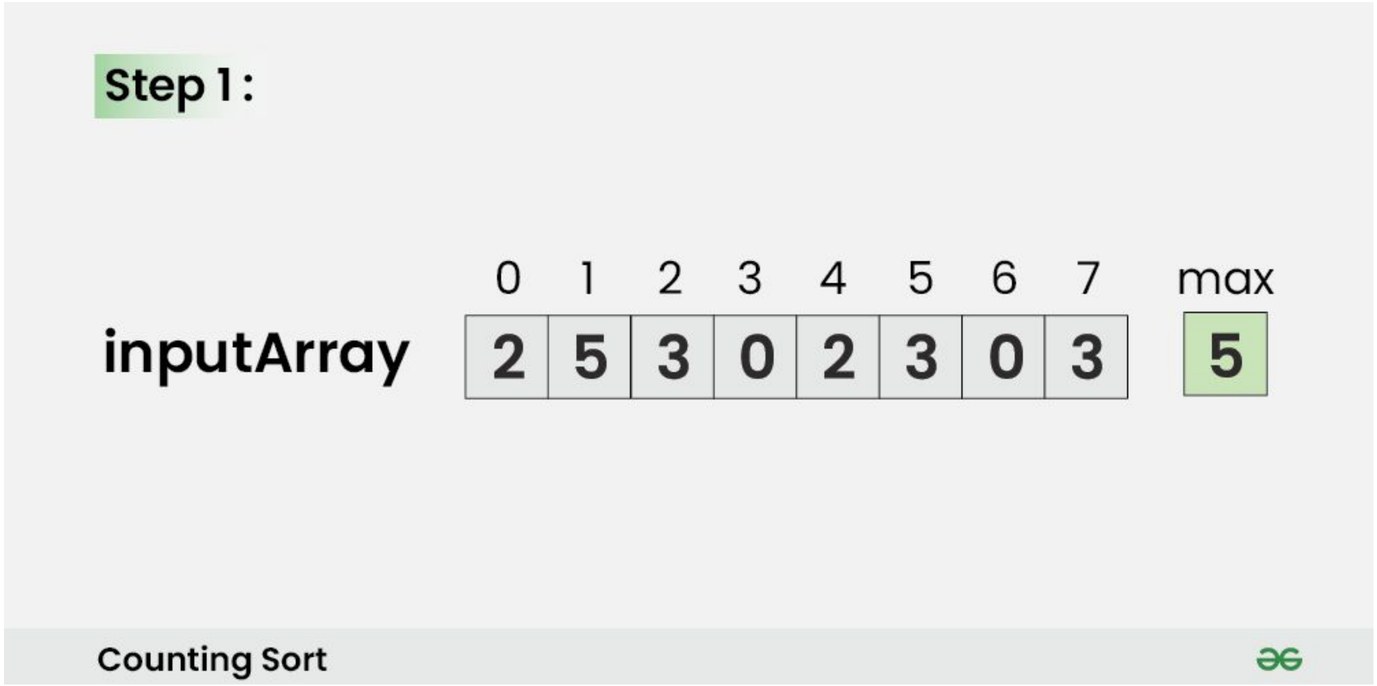
The algorithm:

- Input: A[1..n], where A[j] {1, 2, 3, …, k}
- Output: B[1..n], sorted (not sorted in place)
- Also: Array C[1..k] for auxiliary storage

Key Idea

1. Create a frequency array:
    - Find the largest element in the list
    - Declare an auxiliary array, countArray[] of size max(inputArray[])+1. This array will be used for storing the occurrences of the elements of the input array.
    - Initialize every element of it with 0.
2. Compute frequencies:
    - Traverse array inputArray[] and map each element of inputArray[] as an index of countArray[] array
    - Increase the count for the visited element, i.e., execute countArray[inputArray[i]]++ for 0 <= i < N.
3. Compute cumulative frequencies:
    - Calculate the prefix sum at every index of array inputArray[]
    - countArray[i] = countArray[i] + countArray[i-1]
4. Create an output array:
    - Create an array outputArray[] of size N.
    - Traverse array inputArray[] from end and update outputArray[countArray[inputArray[i]] – 1] = inputArray[i].
    - Update countArray[inputArray[i]] = countArray[inputArray[i]]–
5. Sort the elements:
    - The sorted elements are present in outputArray.

Example



Create frequency array

Find cumulative frequency

Sort the elements using additional array

Algorithm

Algorithm CountingSort(A, B, n)

```
1.    K = FindMax(A, n)             // Find the largest element in the list
2.    for i=0 to K do               // Initialize the frequency vector
3.        C[i]= 0
4.    for j=0 to n-1 do                   // Compute frequency of each element
5.        C[A[j]] = C[A[j]] + 1
6.    for i=1 to K                        // Compute cumulative frequency
7.        C[i] = C[i] + C[i-1]
8.    for j=n-1 downto 0 do               // Sort the elements and store to another array
9.        B[C[A[j]]] = A[j]
10.       C[A[j]] = C[A[j]] - 1
11.   Exit
```

Example

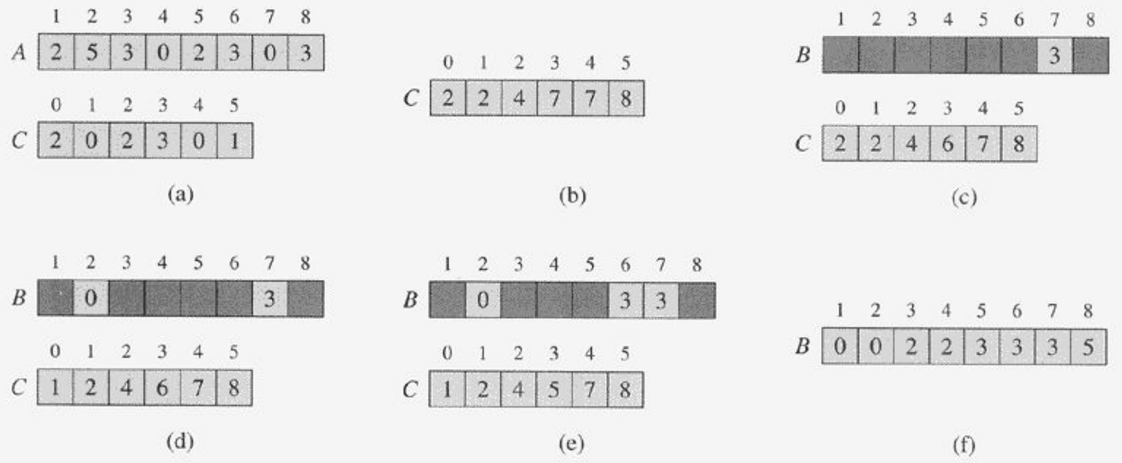


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Complexity

Time Complexity:

- Total time: $O(n + K) = O(\max\{n,K\})$
- Works well if $K = O(n)$ or $K = O(1)$; otherwise if K is really big or the data is skewed then it will take long time.

Space Complexity:

- Total Space: $O(n + K) = O(\max\{n,K\})$

Why don’t we always use counting sort?

- **Depends on range k of elements.**

Could we use counting sort to sort 32 bit integers? Why or why not?

Analysis of Counting Sort

- Count sort is stable:
  - A sorting algorithm is stable when numbers with the same values appear in the output array in the same order as they do in the input array.
  - It means that the same values need not be shuffled.
- When the length of the input list is not substantially smaller than the largest key value, K, in the input array, the counting sort has a running time of O(n). In contrast, any comparison-based sorting algorithm takes O(n (log n)) comparisons.
- In circumstances where the range of input elements is comparable to the number of input elements, counting sort is particularly efficient since it accomplishes sorting in linear time, which might be an advantage over other sorting algorithms like quicksort. When rapid sort takes O(n^2) time in the worst scenario, counting sort only takes O(n) time if the range of elements is not too vast.

Applications of Counting Sort

- If the range of input data is not much bigger than the number of objects to be sorted, counting sort is efficient. Consider the following scenario: the data is 10, 5, 10K, 5K, and the input sequence is 1 to 10K.
- It isn't a sorting system based on comparisons. It has an O(n) running time complexity, with space proportional to the data range.
- It's frequently used as a subroutine in other sorting algorithms, such as radix sort.
- Counting sort counts the occurrences of the data object in O using partial hashing (1).
- The counting sort can also be used with negative inputs.

Homework

- Theory
- Practice

Homeworks

- Write a program to sort the given numbers using stack (s). Consider that elements are known on the go. How many stacks are required for a smooth operation? Which sorting have you implemented?
- Write an algorithm to design Insertion sort using linked list.
- Compare on the go sorting bubble sort, selection sort and insertion sort and heap sort.
- Modify the algorithm for Bubble Sort to find the number of comparisons it has saved. Also, report how many execution iterations are saved if it is stopping earlier. You need to compare it with O(n^2).
- Modify Counting Sort so that it is able to sort negative numbers.
- Can we sort fractional numbers using Counting Sort? Justify your answer.
- Make a table for the sorting algorithms. Each row should state about one sorting algorithm. There are two columns, Time Complexity and Space Complexity, fill the table appropriately.
- In the case of Counting Sort, what if we do not calculate the cumulative frequencies. Will the algorithm behave correctly? Justify your answer with suitable examples.
- Write a modified Counting Sort algorithm which will not calculate the cumulative frequencies, rather it will sort based on frequency count only. Find out the time and space complexity of the modified algorithm. Is it a stable sorting?
- Write an algorithm to count the frequencies of the numbers with using an extra array with the same size of the input array. [Hint- You may sort the elements before counting the frequencies]