Sorting algorithms

Insertion Sort, Bubble sort, Selection sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, and Bucket Sort, Radix sort, Inversions, External sorting.

Definition

Sorting is the process of:

- Taking a list of objects which could be stored in a linear order

$$(a_0, a_1, \ldots, a_{n-1})$$

e.*g*., numbers, and returning an reordering

$$(a'_0, a'_1, \ldots, a'_{n-1})$$

such that

$$a'_0 \le a'_1 \le \cdots \le a'_{n-1}$$

The conversion of an Abstract List into an Abstract Sorted List

Definition

Seldom will we sort isolated values

- Usually, we will sort a number of records containing a number of fields based on a *key*:

19991532	Stevenson	Monica	3 Glendridge Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19985832	Kilji	Islam	37 Masterson Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19981932	Carol	Ann	81 Oakridge Ave.
20003287	Redpath	David	5 Glendale Ave.

Numerically by ID Number

19981932	Carol	Ann	81 Oakridge Ave.
19985832	Khilji	Islam	37 Masterson Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.
20003287	Redpath	David	5 Glendale Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.

Lexicographically by surname, then given name

19981932	Carol	Ann	81 Oakridge Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19985832	Kilji	Islam	37 Masterson Ave.
20003287	Redpath	David	5 Glendale Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.

Assumption

In these topics, we will assume that:

- Arrays are to be used for both input and output,
- We will focus on sorting objects and leave the more general case of sorting records based on one or more fields as an implementation detail

In-place Sorting

Sorting algorithms may be performed *in-place*, that is, with the allocation of at most $\Theta(1)$ additional memory (*e*.*g*., fixed number of local variables)

Other sorting algorithms require the allocation of second array of equal size

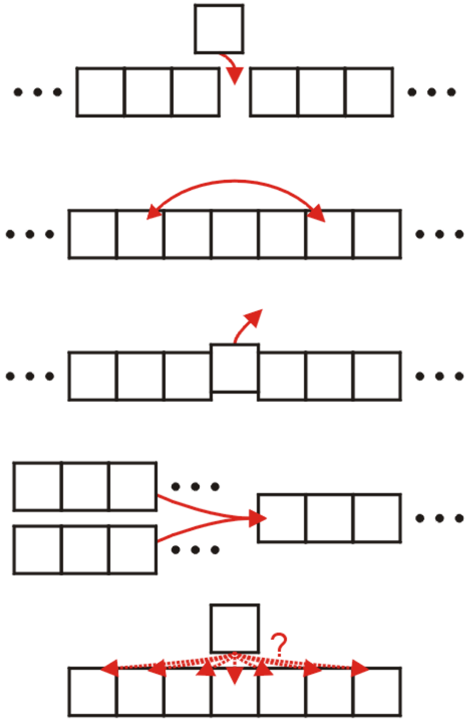
- Requires $\Theta(n)$ additional memory

We will prefer in-place sorting algorithms

Classifications

The operations of a sorting algorithm are based on the **actions** performed:

- Insertion
- Exchanging
- Selection
- Merging
- Distribution



Sorting by Comparison

Basic operation involved in this type of sorting technique is comparison. A data item is compared with other items in the list of items in order to find its place in the sorted list.

- Insertion
- Selection
- Exchange
- Enumeration

Sorting by Comparison

Sorting by comparison – Insertion:

- From a given list of items, one item is considered at a time. The item chosen is then inserted into an appropriate position relative to the previously sorted items. The item can be inserted into the same list or to a different list.

  e.g.: Insertion sort

Sorting by comparison – Selection:

- First the smallest (or largest) item is located and it is separated from the rest; then the next smallest (or next largest) is selected and so on until all item are separated.

  e.g.: Selection sort, Heap sort

Sorting by Comparison

Sorting by comparison – Exchange:

- If two items are found to be out of order, they are interchanged. The process is repeated until no more exchange is required.

  e.g.: Bubble sort, Shell Sort, Quick Sort

Sorting by comparison – Enumeration:

- Two or more input lists are merged into an output list and while merging the items, an input list is chosen following the required sorting order.

  e.g.: Merge sort

Sorting by Distribution

- No key comparison takes place
- All items under sorting are distributed over an auxiliary storage space based on the constituent element in each and then grouped them together to get the sorted list.
- Distributions of items based on the following choices

✓ **Radix -** An item is placed in a space decided by the bases (or radix) of its components with which it is composed of.

✓ **Counting -** Items are sorted based on their relative counts.

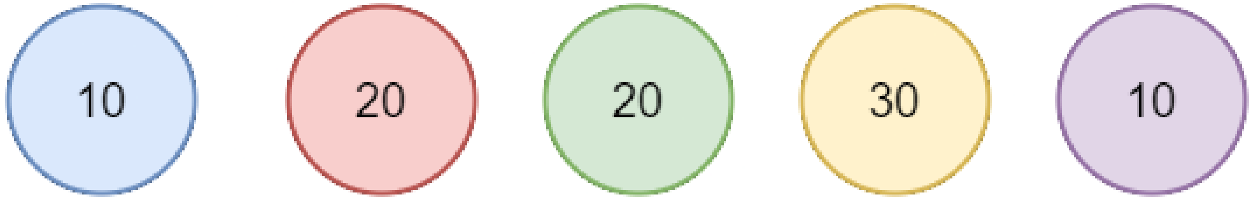
Note: This lecture concentrates only on sorting by comparison.

Optimal Sorting Algorithms

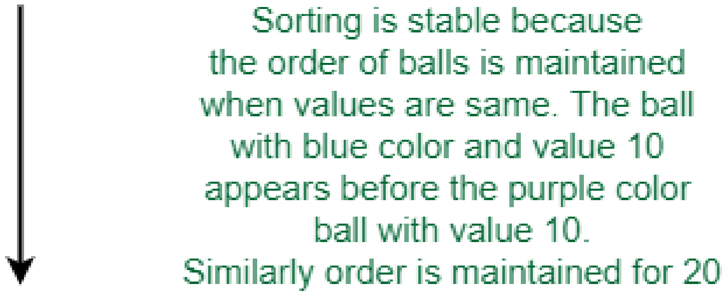
The next seven topics will cover common sorting algorithms

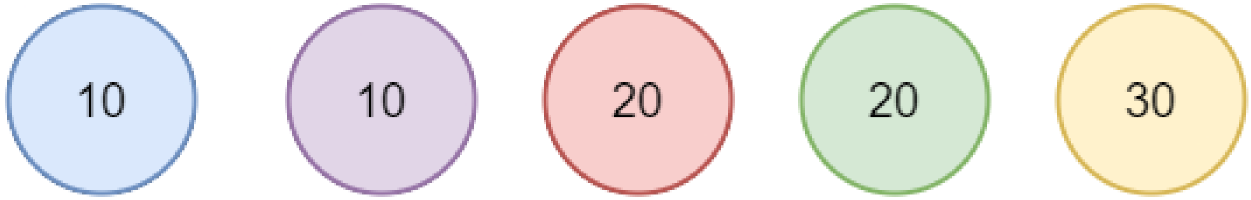
- There is no *optimal* sorting algorithm which can be used in all places
- Under various circumstances, different sorting algorithms will deliver optimal run-time and memory-allocation requirements

Stable sort



A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set.



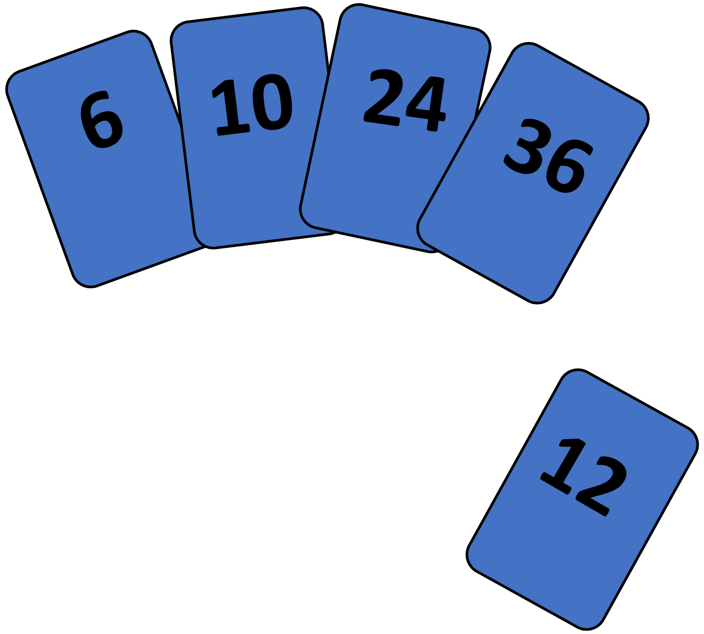


Insertion Sort

Insertion Sort

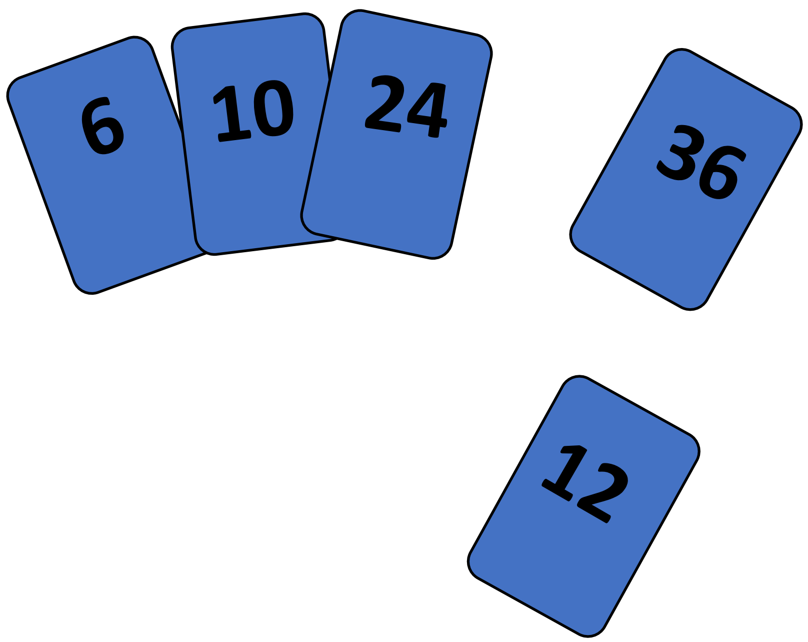
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

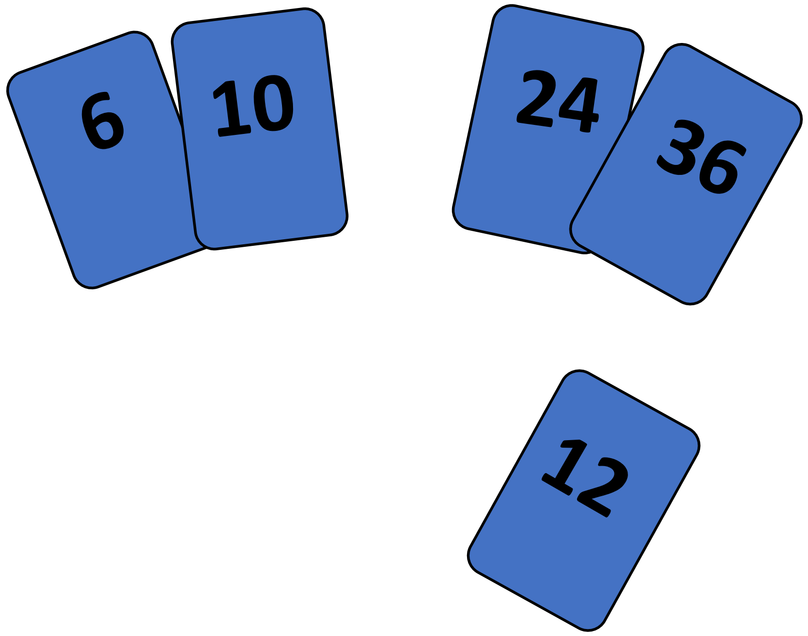


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort

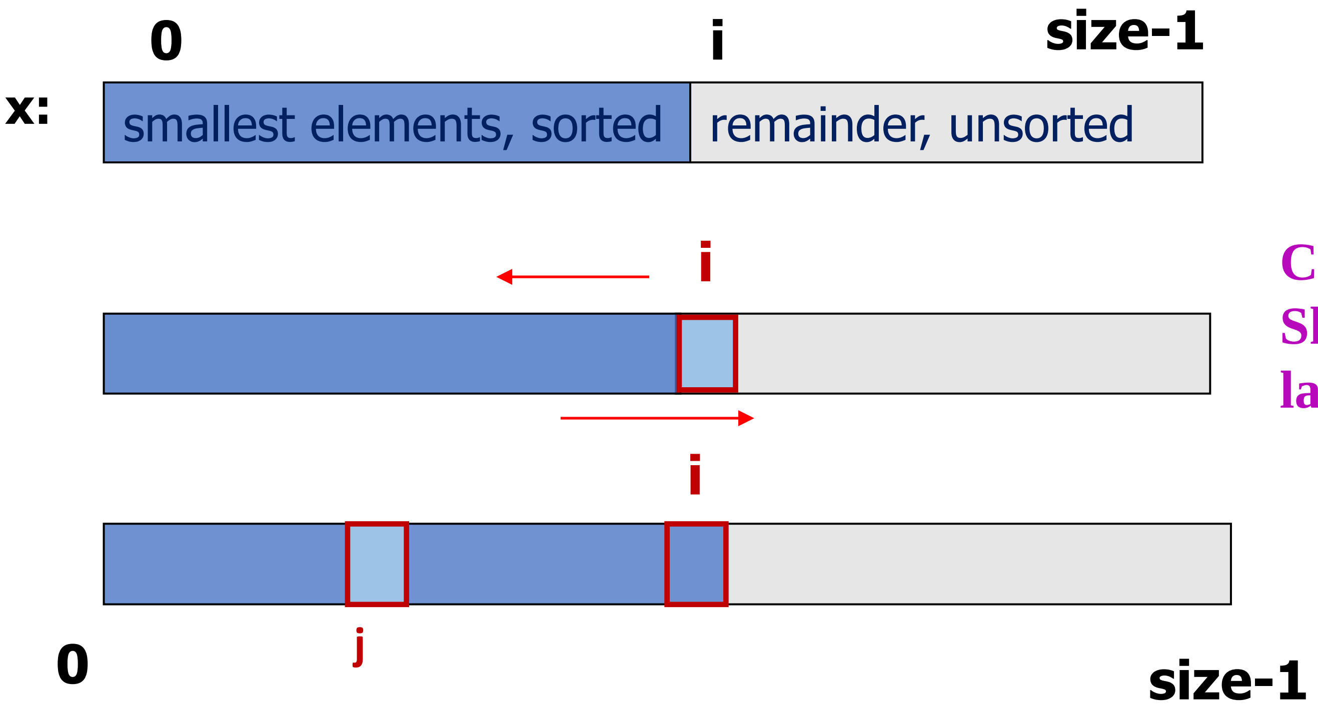


Insertion Sort



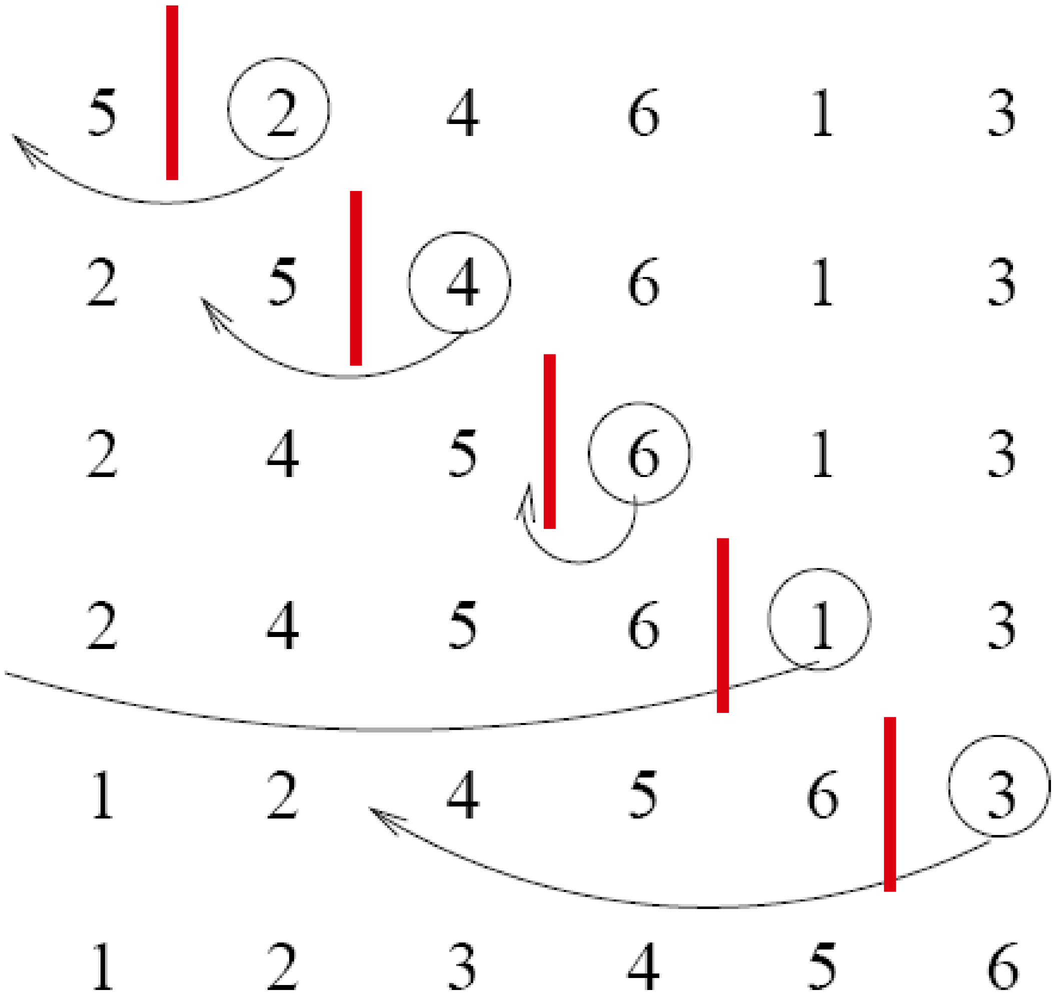
Insertion Sort

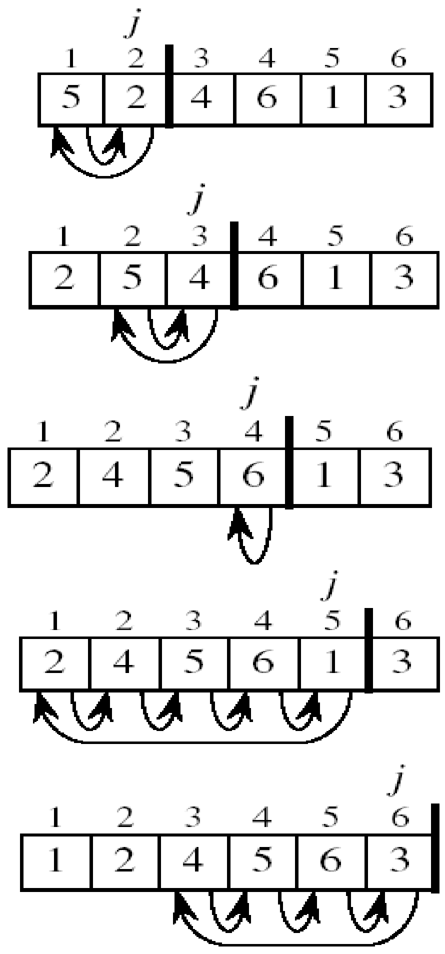
General situation :



Compare and Shift till x[i] is larger.

Insertion Sort





Algorithm/Steps/ Flowchart

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Insertion Sort

```
void insertionSort (int list[], int size)
{
        int i, j, item;

        for (i=1; i<size; i++)
        {
                item = list[i] ;

                /* Move elements of list[0..i-1], that are
                greater than item, to one position ahead of
                their current position */

                for (j=i-1; (j>=0)&& (list[j] > item); j--)
                        list[j+1] = list[j];
                list[j+1] = item ;
        }
}
```

Insertion Sort

```
int main()
{
    int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};

    int i;
    for(i=0;i<12;i++)
       printf("%d ",x[i]);
    printf("\n");

    insertionSort(x,12);

    for(i=0;i<12;i++)
       printf("%d ",x[i]);
    printf("\n");
}
```

OUTPUT

```
-45 89 -65 87 0 3 -23 19 56 21 76 -50

-65 -50 -45 -23 0 3 19 21 56 76 87 89
```

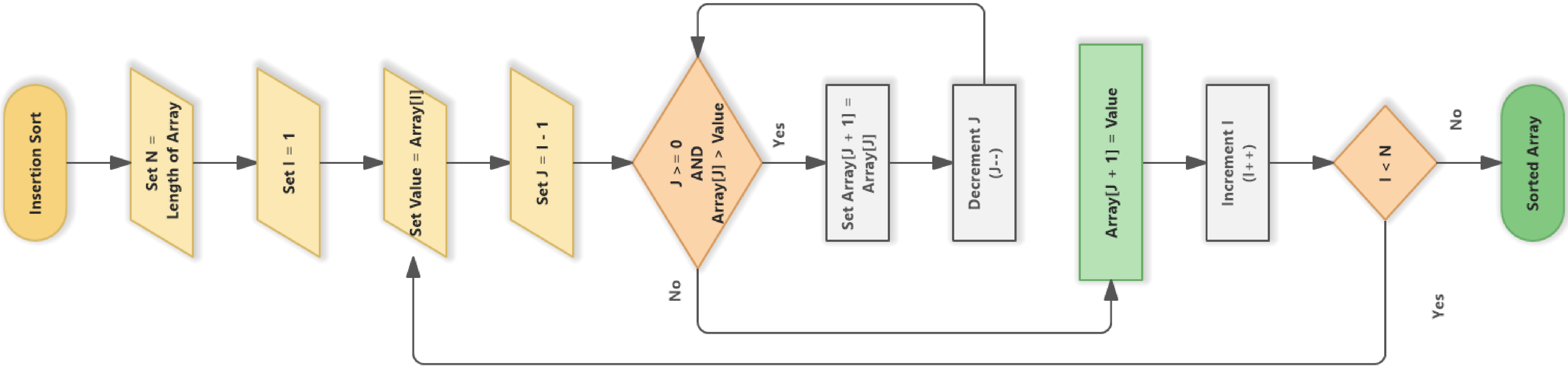
Insertion Sort

```
# Python program for implementation of Insertion Sort
# Function to do insertion sort
def insertionSort(arr):

    if (n := len(arr)) <= 1:
      return
    for i in range(1, n):

        key = arr[i]
        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
#sorting the array [12, 11, 13, 5, 6] using insertionSort
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print(arr)
```

Algorithm/Procedure/ Flowchart



Insertion Sort- Example

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 0: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 1: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	0.56	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	1.12	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	1.17	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	0.32	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	0.32	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	0.32	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	0.32	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 3.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 4.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 5.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Iteration 6: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Iteration 6: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	4.42	7.42	3.14	7.71

Iteration 7: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Iteration 7: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Iteration 7: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	3.14	7.42	7.71

Iteration 8: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	3.14	6.21	7.42	7.71

Iteration 8: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 3.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 9: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.
- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 10: DONE.

Insertion Sort – Another Example

54	26	93	17	77	31	44	55	20	Assume 54 is a sorted list of 1 item
26	54	93	17	77	31	44	55	20	inserted 26
26	54	93	17	77	31	44	55	20	inserted 93
17	26	54	93	77	31	44	55	20	inserted 17
17	26	54	77	93	31	44	55	20	inserted 77
17	26	31	54	77	93	44	55	20	inserted 31
17	26	31	44	54	77	93	55	20	inserted 44
17	26	31	44	54	55	77	93	20	inserted 55
17	20	26	31	44	54	55	77	93	inserted 20

Insertion Sort: Complexity Analysis

Case 1: If the input list is already in sorted order

Number of comparisons: Number of comparison in each iteration is 1.

$$C(n) = 1 + 1 + 1 + \cdots + 1 \quad \text{upto } (n\text{-}1)^{th} \text{ iteration.}$$

Number of movement: No data movement takes place in any iteration.

$$M(n) = 0$$

Insertion Sort: Complexity analysis

Case 2: If the input list is sorted but in reverse order

Number of comparisons: Number of comparison in each iteration is 1.

$$C(n) = 1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

Number of movement: Number of movements takes place in any i^{th} iteration is i.

$$M(n) = 1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

Insertion Sort: Complexity analysis

Case 3: If the input list is in random order

- Let p_j be the probability that the key will go to the j^{th} location $(1 \leq j \leq i+1)$. Then the number of comparisons will be $j \cdot p_j$.

- The average number of comparisons in the $(\mathrm{i}+1)^{th}$ iteration is

$$A_{i+1} = \sum_{j=1}^{i+1} j \cdot p_j$$

- Assume that all keys are distinct, and all permutations of keys are equally likely.

$$p_1 = p_2 = p_3 = \cdots = p_{i+1} = \frac{1}{i+1}$$

Insertion Sort: Complexity analysis

Case 3: Number of comparisons

- Therefore, the average number of comparisons in the $(i + 1)^{th}$ iteration

$$A_{i+1} = \frac{1}{i+1}\sum_{j=1}^{i+1} j = \frac{1}{i+1} \cdot \frac{(i+1)\cdot(i+2)}{2} = \frac{i+2}{2}$$

- Total number of comparisons for all $(n-1)$ iterations is

$$C(n) = \sum_{i=0}^{n-1} A_{i+1} = \frac{1}{2} \cdot \frac{n(n-1)}{2} + (n-1)$$

Insertion Sort: Complexity analysis

Case 3: Number of Movements

- On the average, number of movements in the i^{th} iteration

$$M_i = \frac{i + (i-1) + (i-2) + \cdots + 2 + 1}{i} = \frac{i+1}{2}$$

- Total number of movements

$$M(n) = \sum_{i=1}^{n-1} M_i = \frac{1}{2} \cdot \frac{n(n-1)}{2} + \frac{n-1}{2}$$

Alternate solution

The recurrence relation of the recursive insertion sort is:

$$\underline{\mathbf{T(n) = T(n-1) + n}}$$

```
def insertionSortRecursive(arr,n):
    # base case
    if n<=1:
        return

    # Sort first n-1 elements
    insertionSortRecursive(arr,n-1)
    '''Insert last element at its correct position
        in sorted array.'''
    last = arr[n-1]
    j = n-2

      # Move elements of arr[0..i-1], that are
      # greater than key, to one position ahead
      # of their current position
    while (j>=0 and arr[j]>last):
        arr[j+1] = arr[j]
        j = j-1

    arr[j+1]=last
```

The recurrence relation of the recursive insertion sort is:

<u>T(n) = T(n-1) + n</u>

```
T(n) = T(n-1) + n
     = T(n-2) + n-1 + n
     = T(n-3) + n-2 + n-1 + n
     // we can now generalize to k
     = T(n-k) + n-k+1 + n-k+2 + ... + n-1 + n
     // since n-k = 1 so T(1) = 1
     = 1 + 2 + ... + n    //Here
     = n(n-1)/2
     = n^2/2 - n/2
     // we take the dominating term which is n^2*1/2 therefor 1/2 = big O
     = big O(n^2)
```

Insertion Sort: Summary of Complexity Analysis

Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = c\,(n-1)$	$T(n) = O(n)$	Best case
Case 2	$T(n) = c\,n(n-1)$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = c\,\frac{(n-1)(n+3)}{2}$	$T(n) = O(n^2)$	Average case

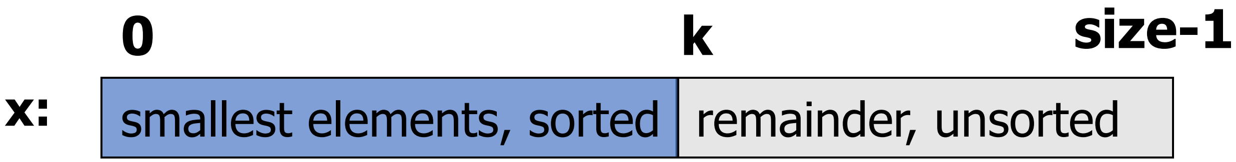
Is this Insertion sort stable ?

YES

Selection Sort

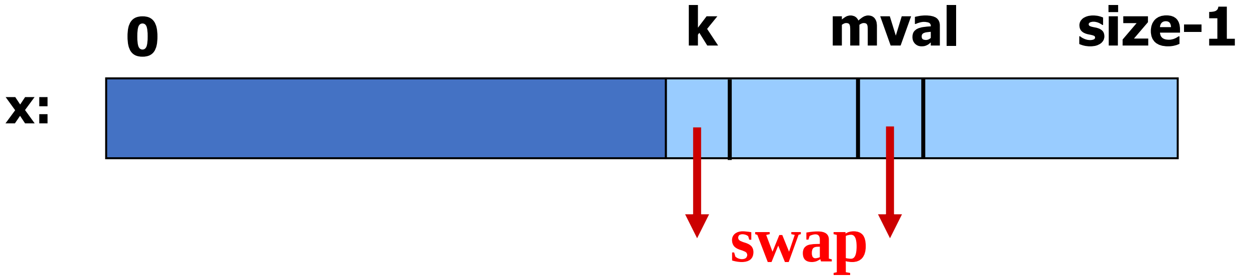
Selection Sort

General situation :



Steps :

- **Find smallest element, mval, in x[k…size-1]**
- **Swap smallest element with x[k], then increase k.**



Selection Sort - Example

x:	3	12	-5	6	142	21	-17	45
x:	-17	12	-5	6	142	21	3	45
x:	-17	-5	12	6	142	21	3	45
x:	-17	-5	3	6	142	21	12	45
x:	-17	-5	3	6	142	21	12	45
x:	-17	-5	3	6	12	21	142	45
x:	-17	-5	3	6	12	21	142	45
x:	-17	-5	3	6	12	21	45	142
x:	-17	-5	3	6	12	21	45	142

Algorithm/Procedure/ Flowchart

SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1

Step 2: CALL SMALLEST(arr, i, n, pos)

Step 3: SWAP arr[i] with arr[pos]

[END OF LOOP]

Step 4: EXIT

SMALLEST (arr, i, n, pos)

Step 1: [INITIALIZE] SET SMALL = arr[i]

Step 2: [INITIALIZE] SET pos = i

Step 3: Repeat for j = i+1 to n

if (SMALL > arr[j])

    SET SMALL = arr[j]

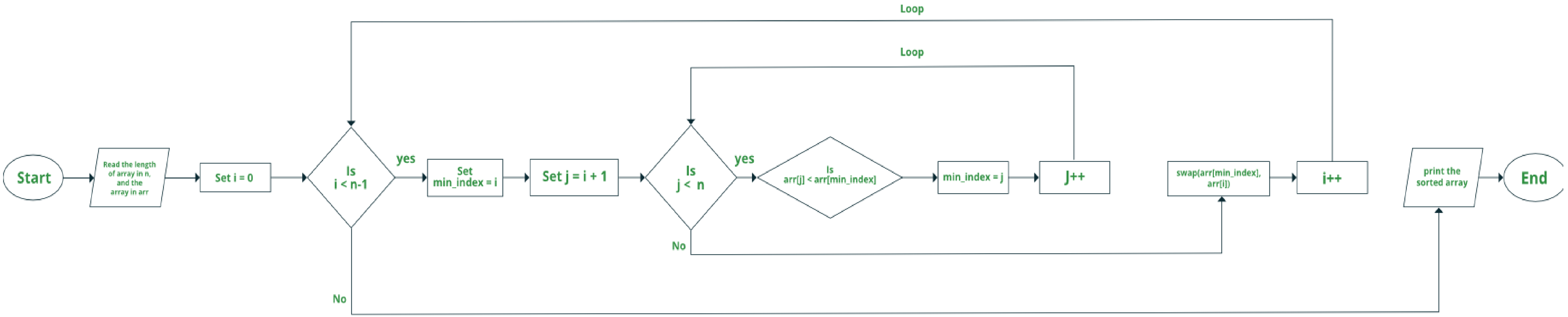
SET pos = j

[END OF if]

[END OF LOOP]

Step 4: RETURN pos

Algorithm/Procedure/ Flowchart



Flowchart for Selection Sort

Selection Sort

```
/* Yield location of smallest element in
x[k .. size-1];*/

int findMinLloc (int x[ ], int k, int size)
{
    int j, pos;       /* x[pos] is the smallest
element found so far */
      pos = k;
      for (j=k+1; j<size; j++)
            if (x[j] < x[pos])
                  pos = j;
      return pos;
}
```

Selection Sort

```
/* The main sorting function */
/* Sort x[0..size-1] in non-decreasing order */

int selectionSort (int x[], int size)
{   int k, m;
        for (k=0; k<size-1; k++)
      {
                m = findMinLoc(x, k, size);
                temp = a[k];
                a[k] = a[m];
                a[m] = temp;
        }
}
```

Selection Sort

```
# Selection sort in Python
# time complexity O(n*n)
#sorting by finding min_index
def selectionSort(array, size):

    for ind in range(size):
        min_index = ind


        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
         # swapping the elements to sort the array
        (array[ind], array[min_index]) = (array[min_index], array[ind])


arr = [-2, 45, 0, 11, -9,88,-97,-202,747]
size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort is:')
print(arr)
```

Selection Sort- Another example

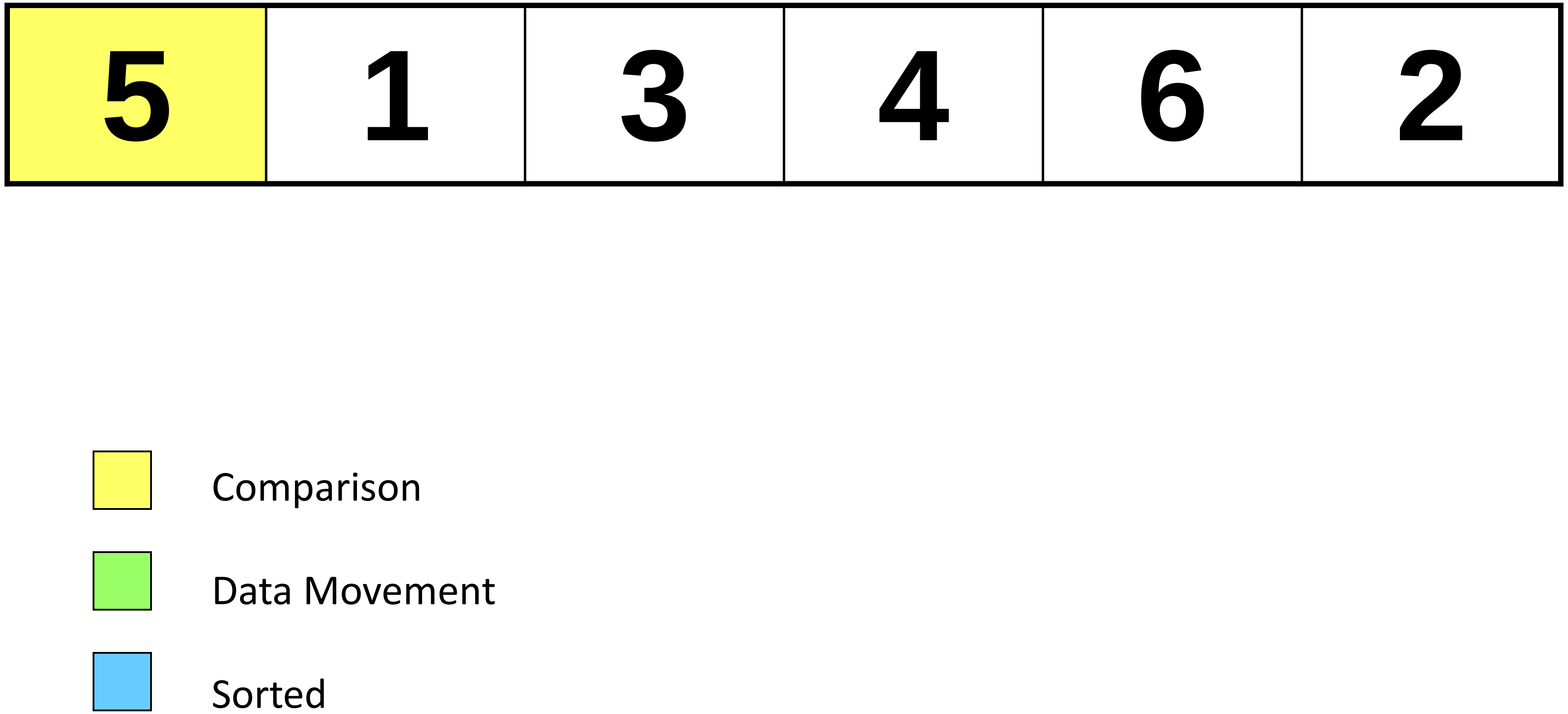
5	**1**	**3**	**4**	**6**	**2**

Comparison

Data Movement

Sorted

Selection Sort



Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	6	2

↑
Largest

Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

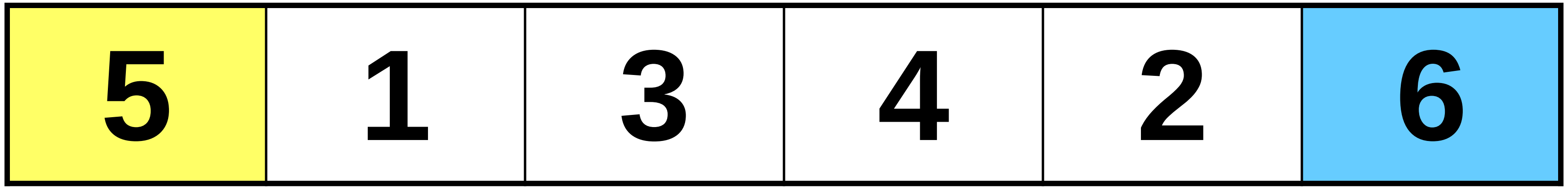
5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

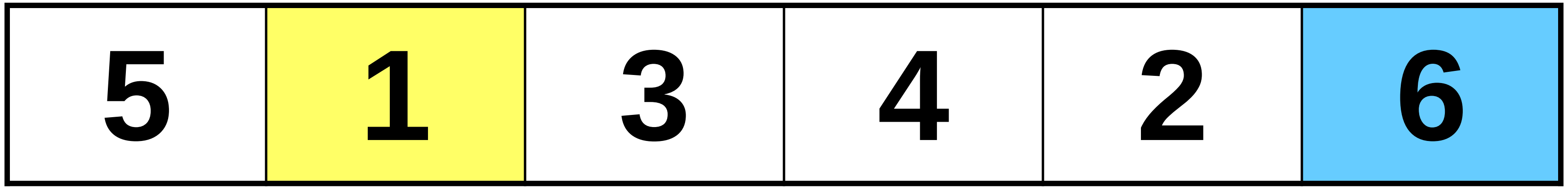


Comparison

Data Movement

Sorted

Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

5	1	3	4	2	6

Comparison

Data Movement

Sorted

Selection Sort

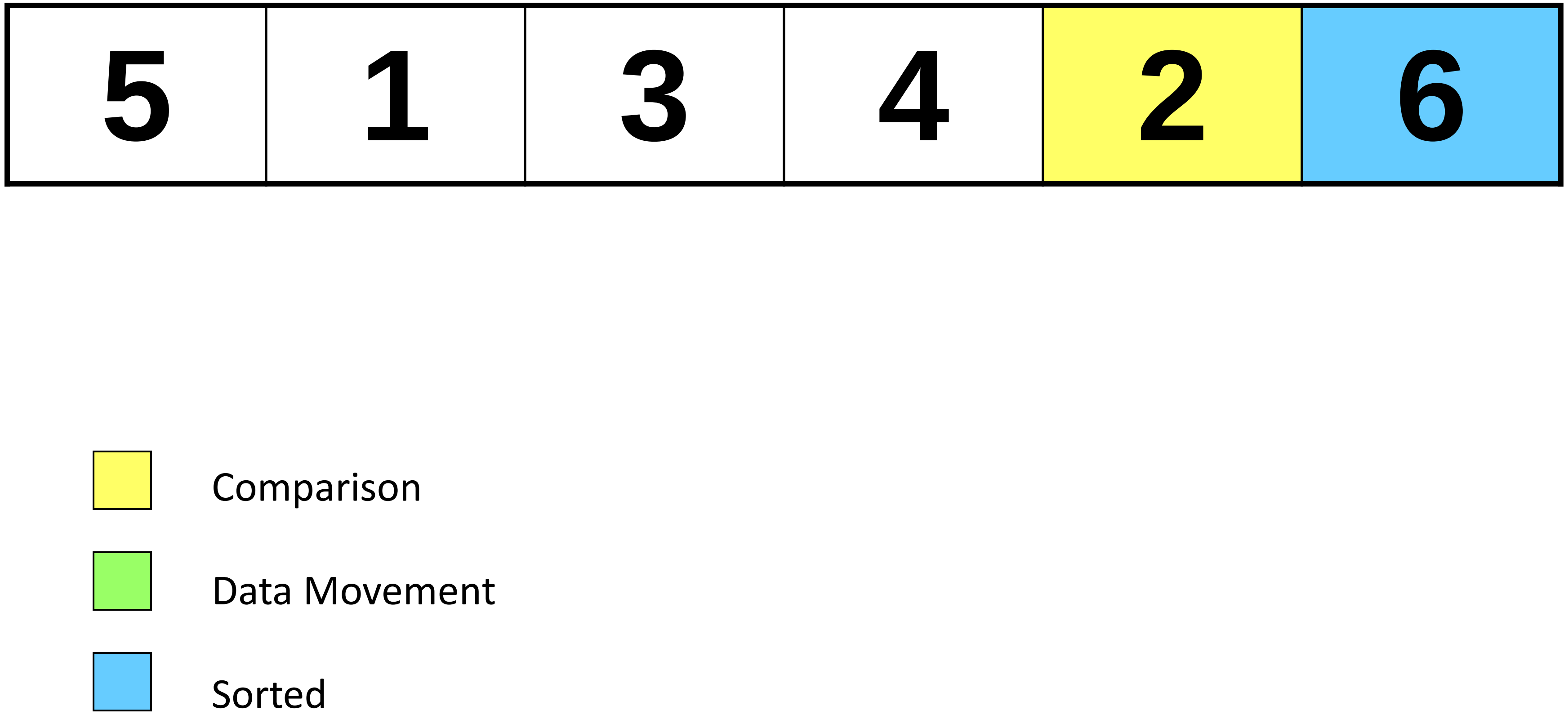
5	1	3	4	2	6

Comparison

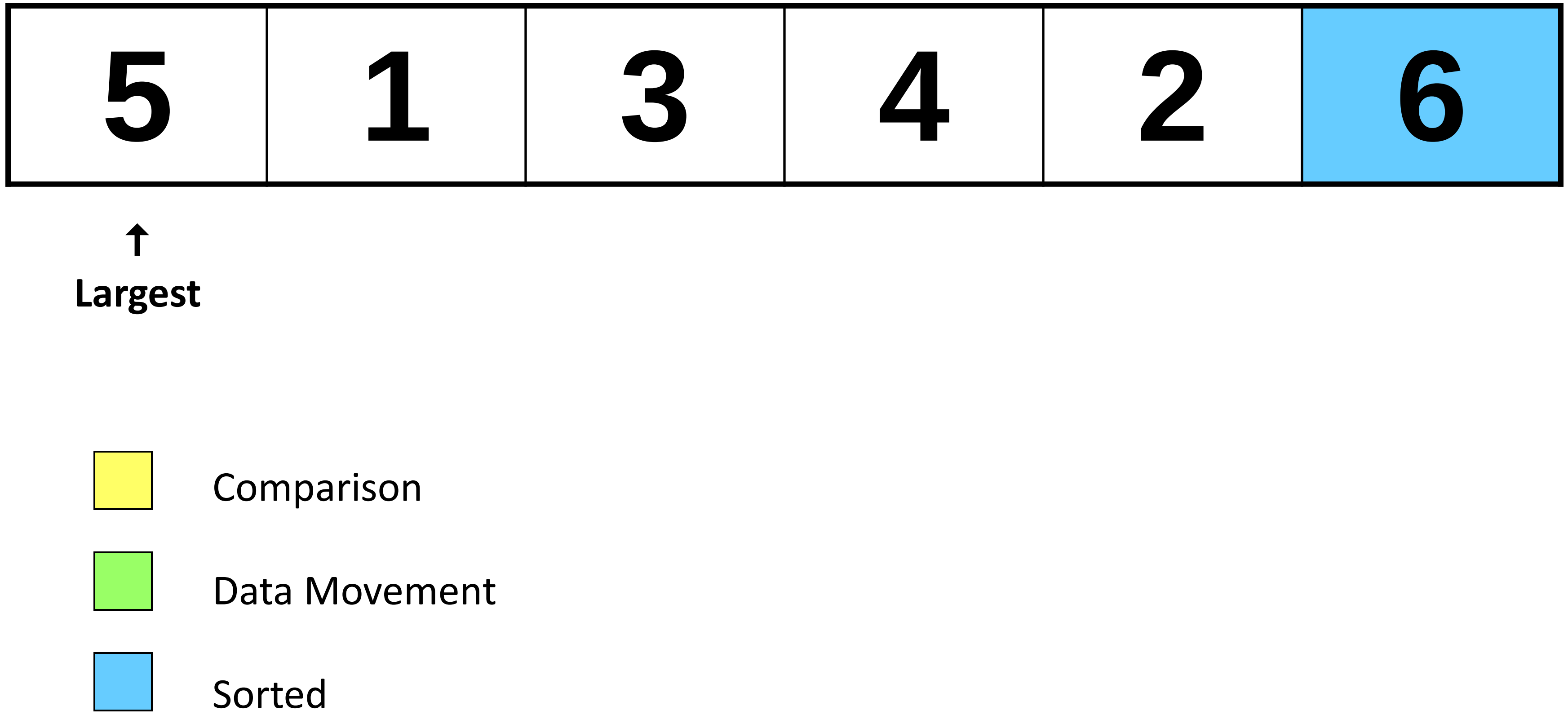
Data Movement

Sorted

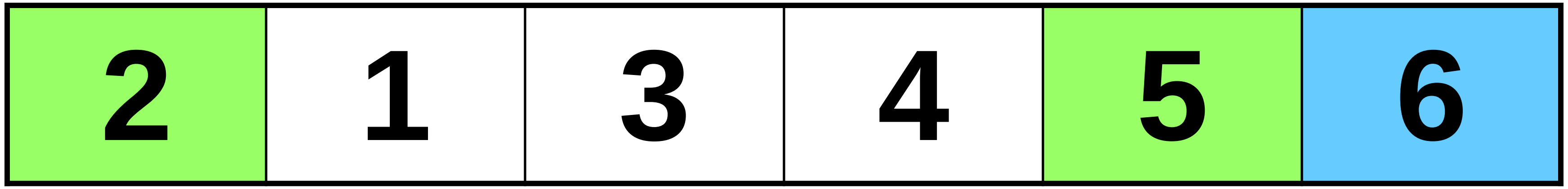
Selection Sort

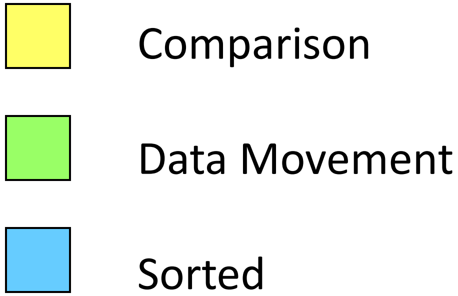


Selection Sort



Selection Sort





Selection Sort

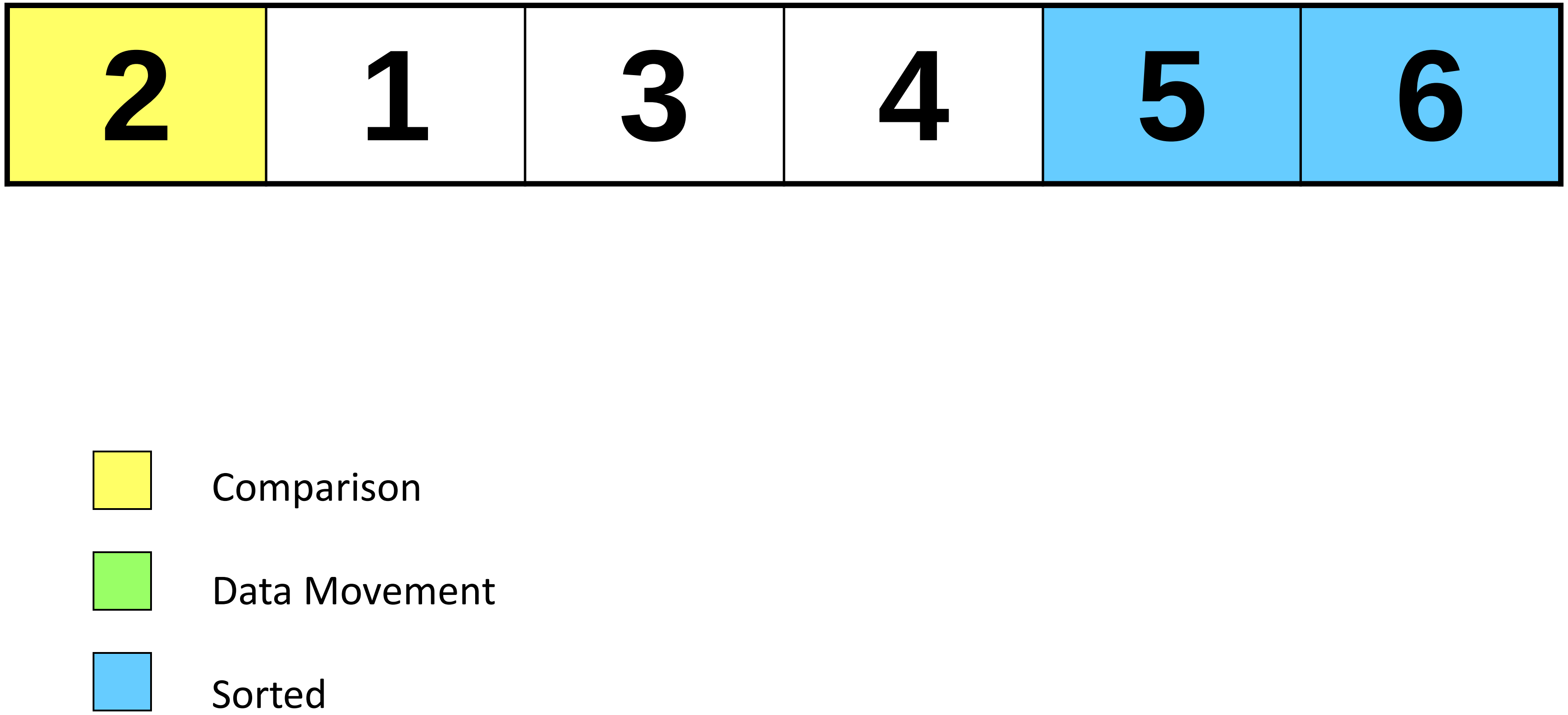
2	1	3	4	5	6

Comparison

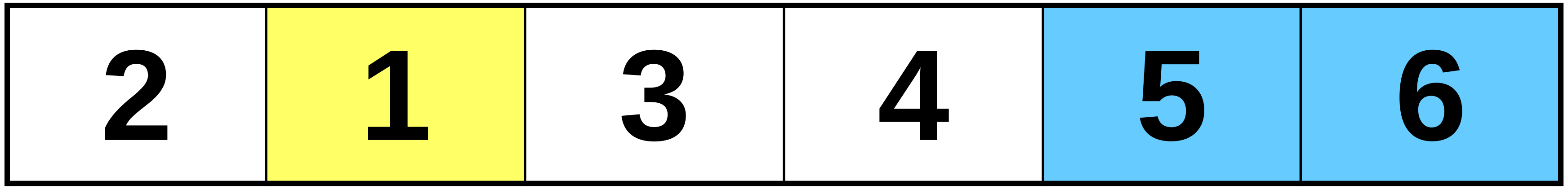
Data Movement

Sorted

Selection Sort



Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

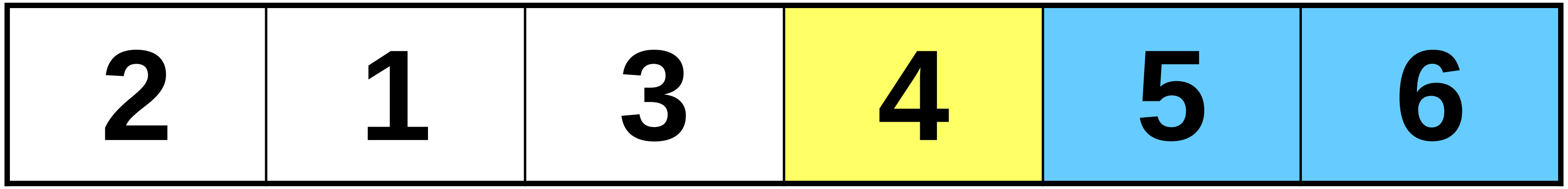
2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

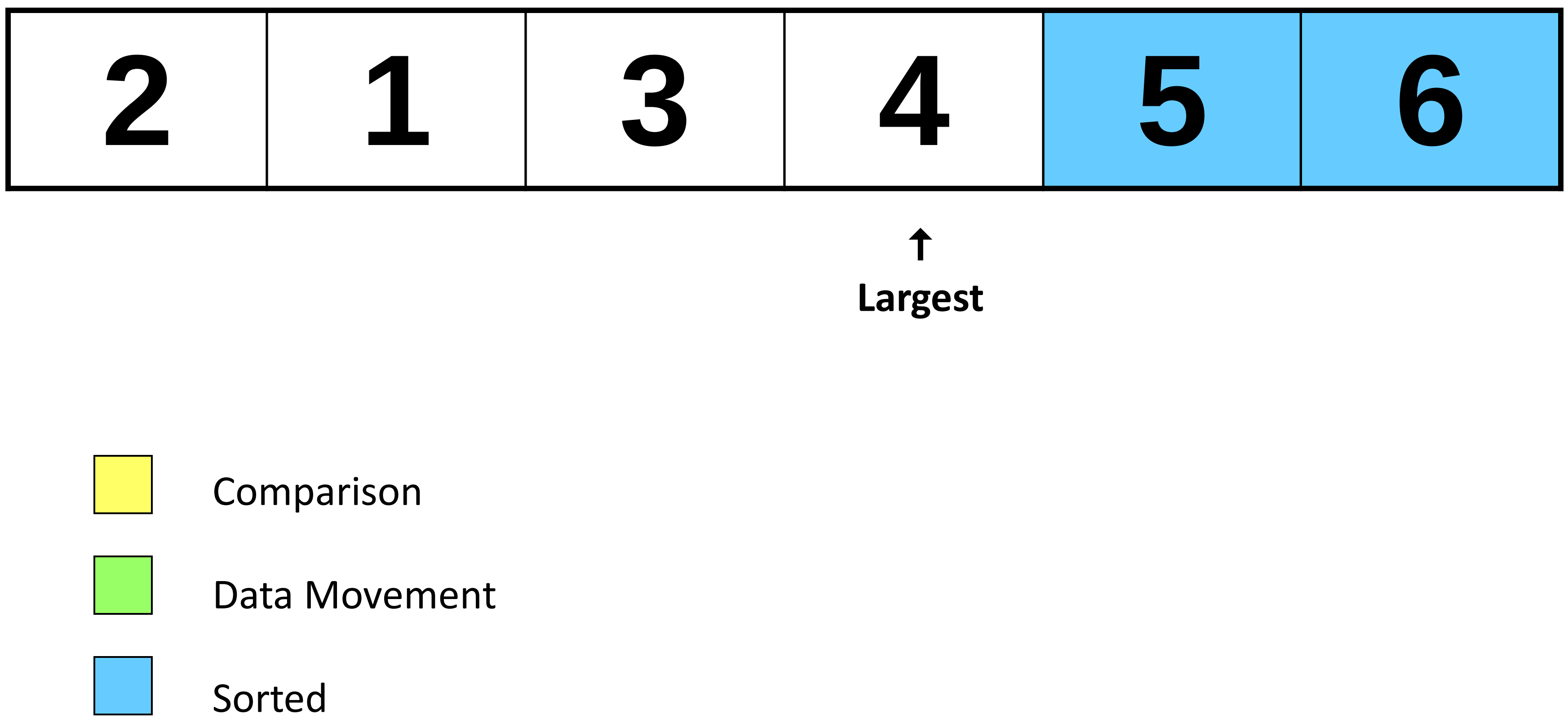


Comparison

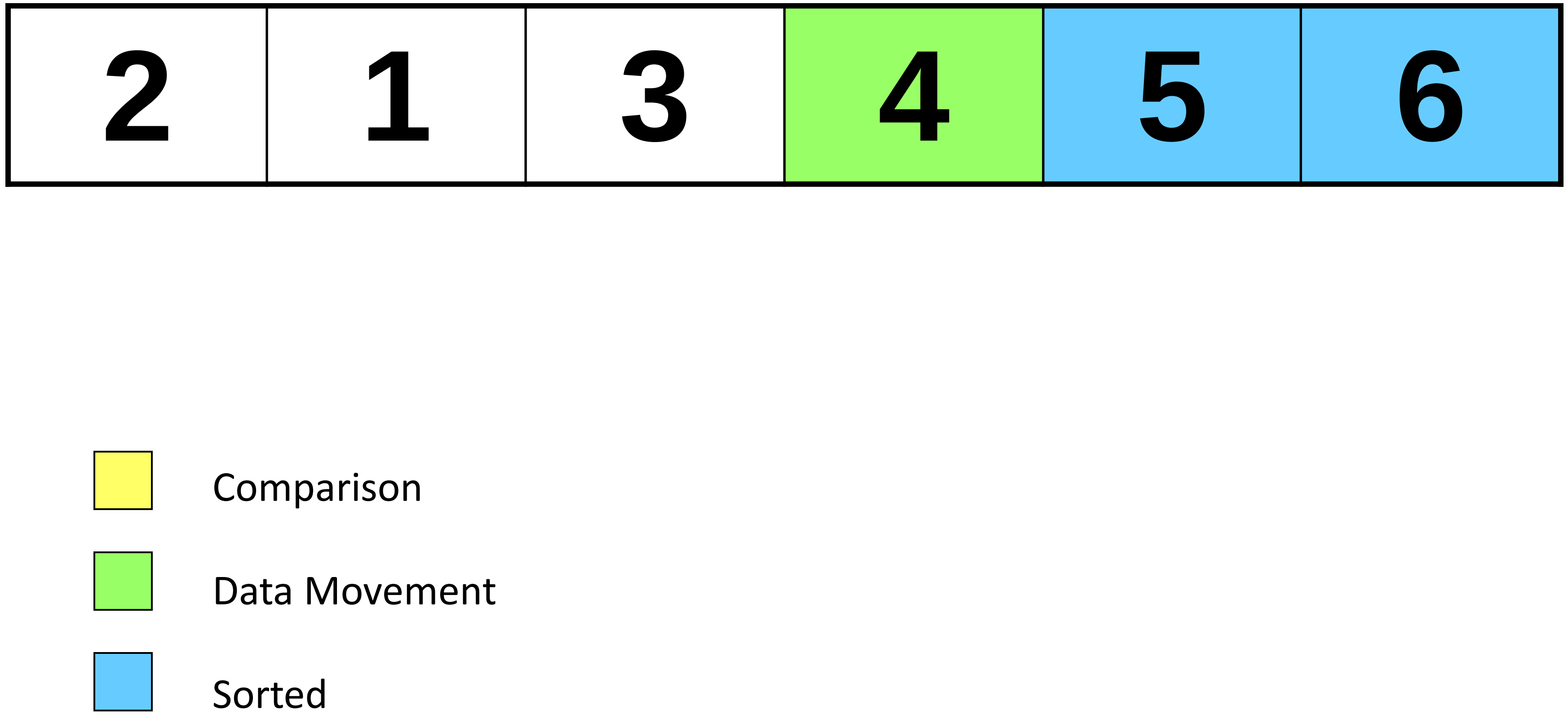
Data Movement

Sorted

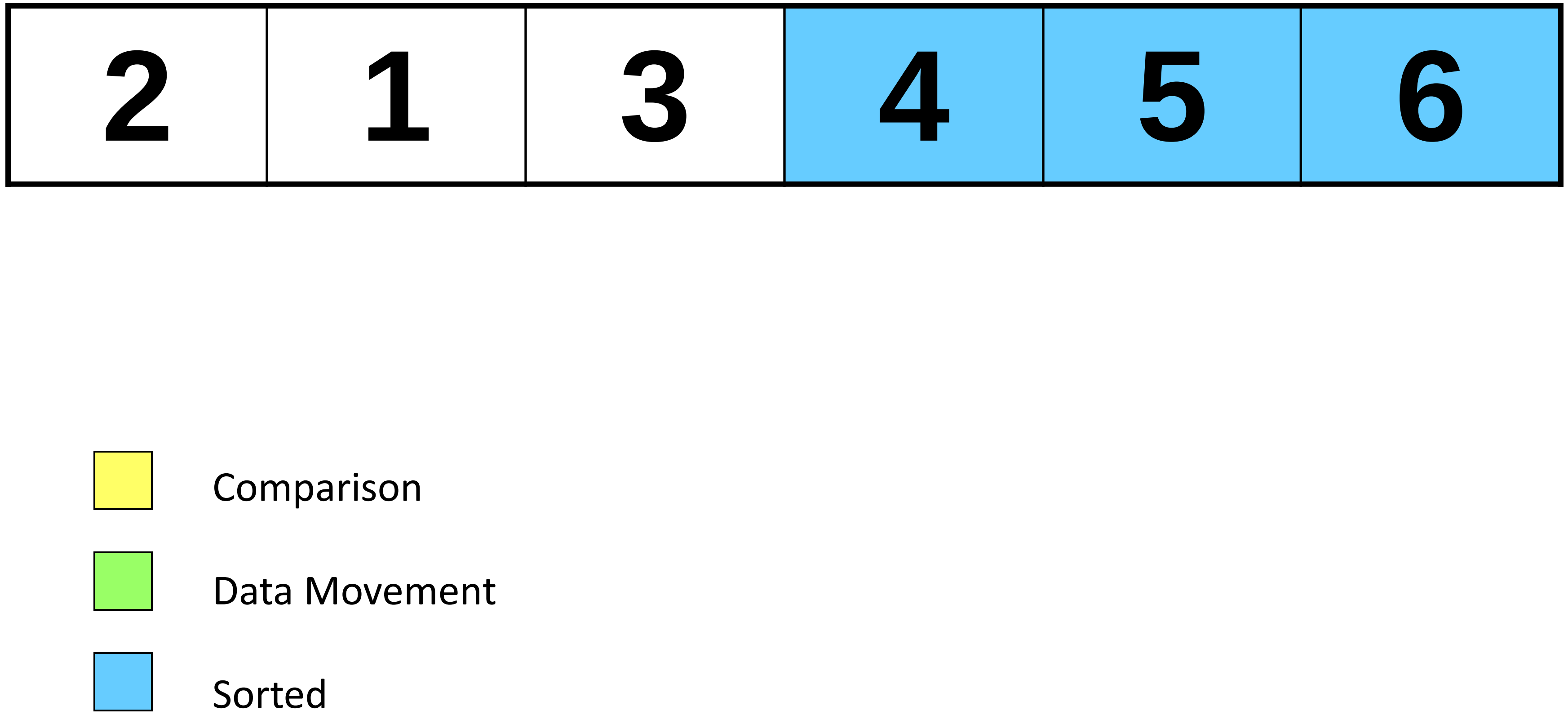
Selection Sort



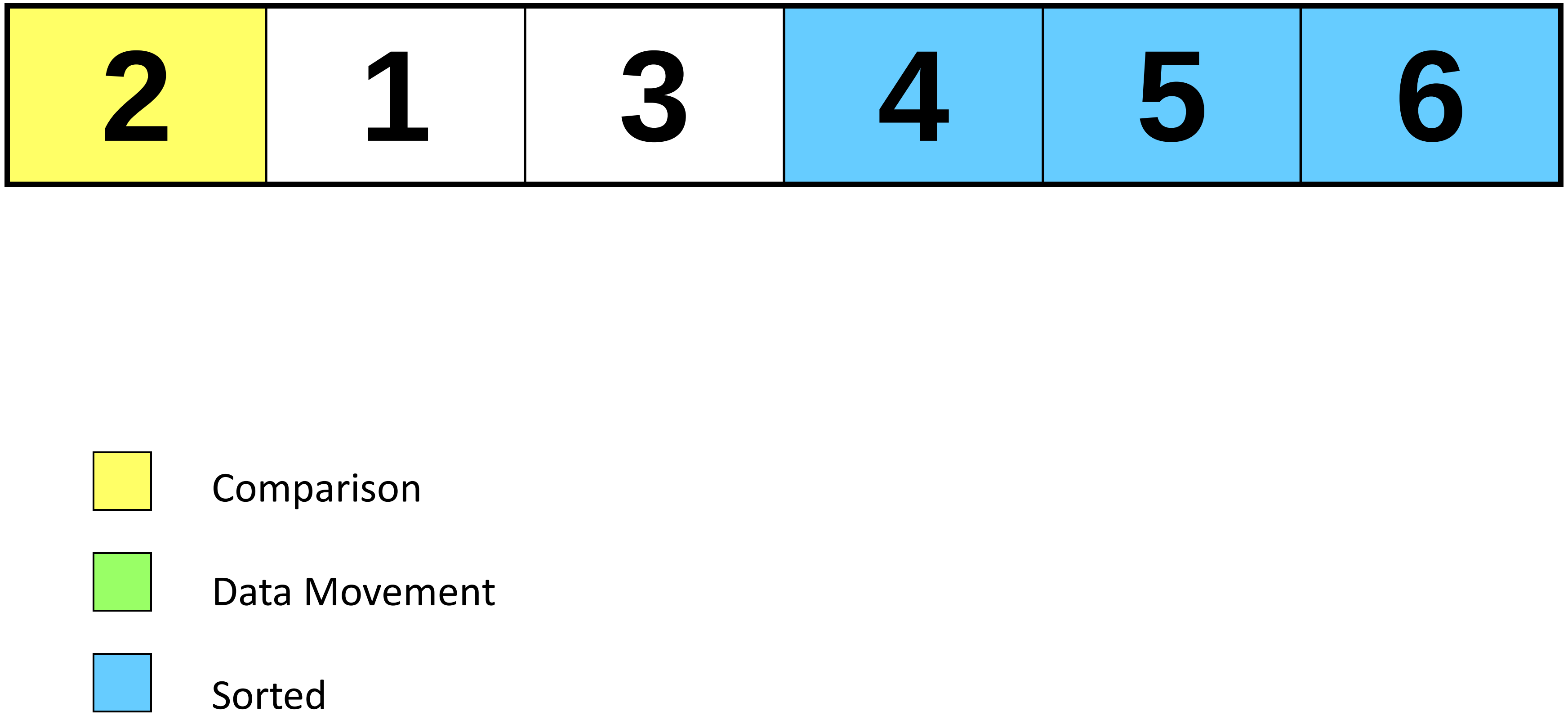
Selection Sort



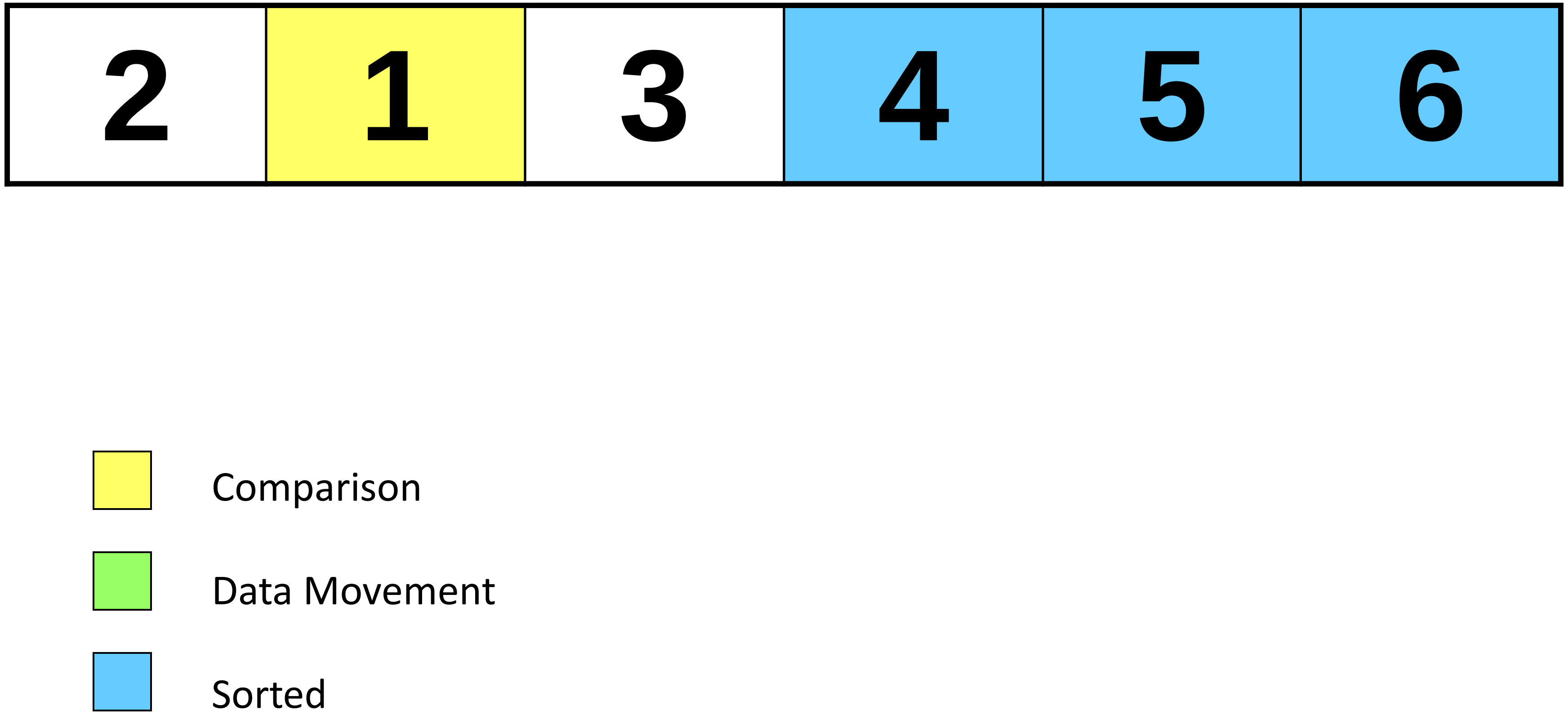
Selection Sort



Selection Sort



Selection Sort



Selection Sort

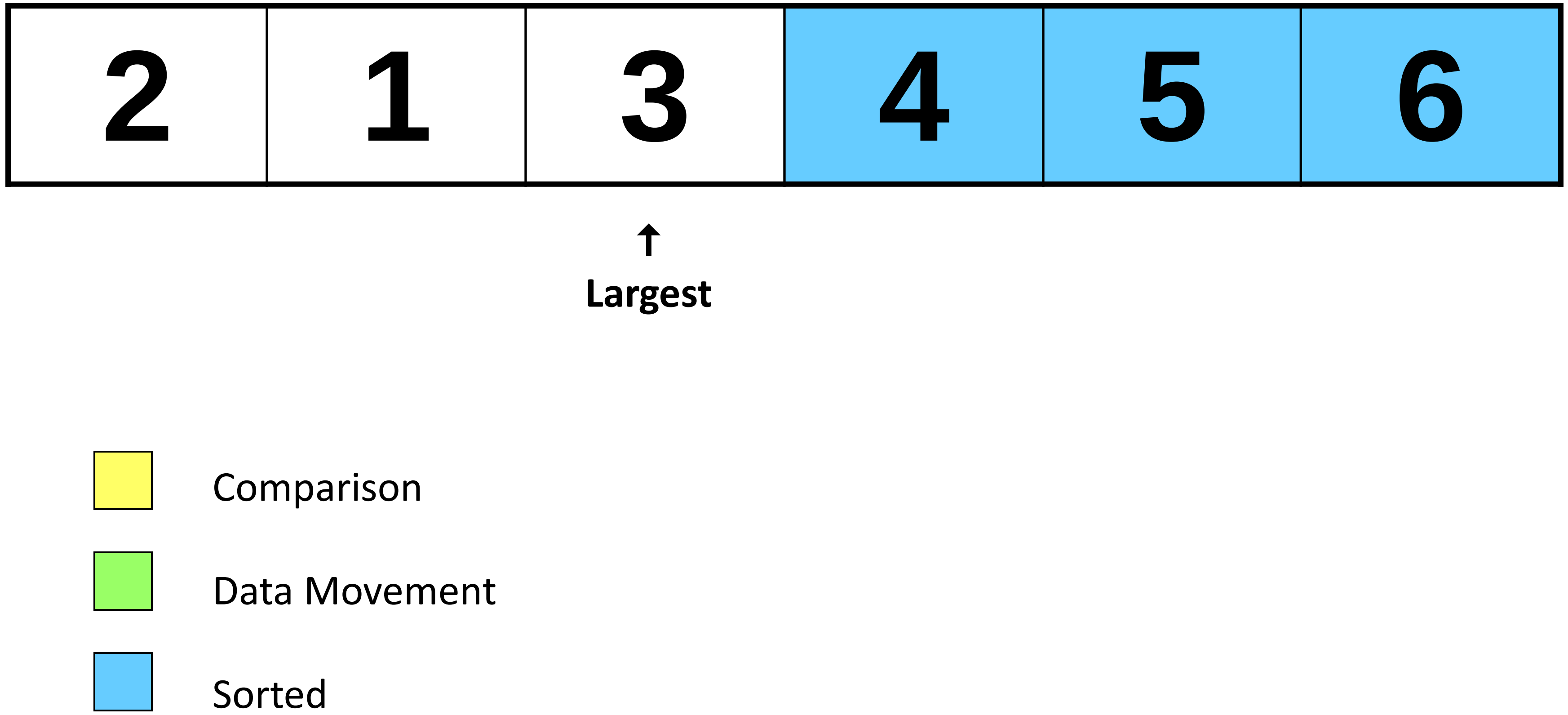
2	1	3	4	5	6

Comparison

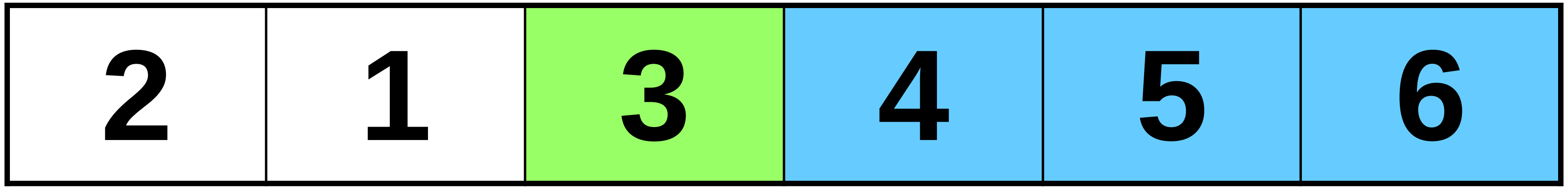
Data Movement

Sorted

Selection Sort



Selection Sort

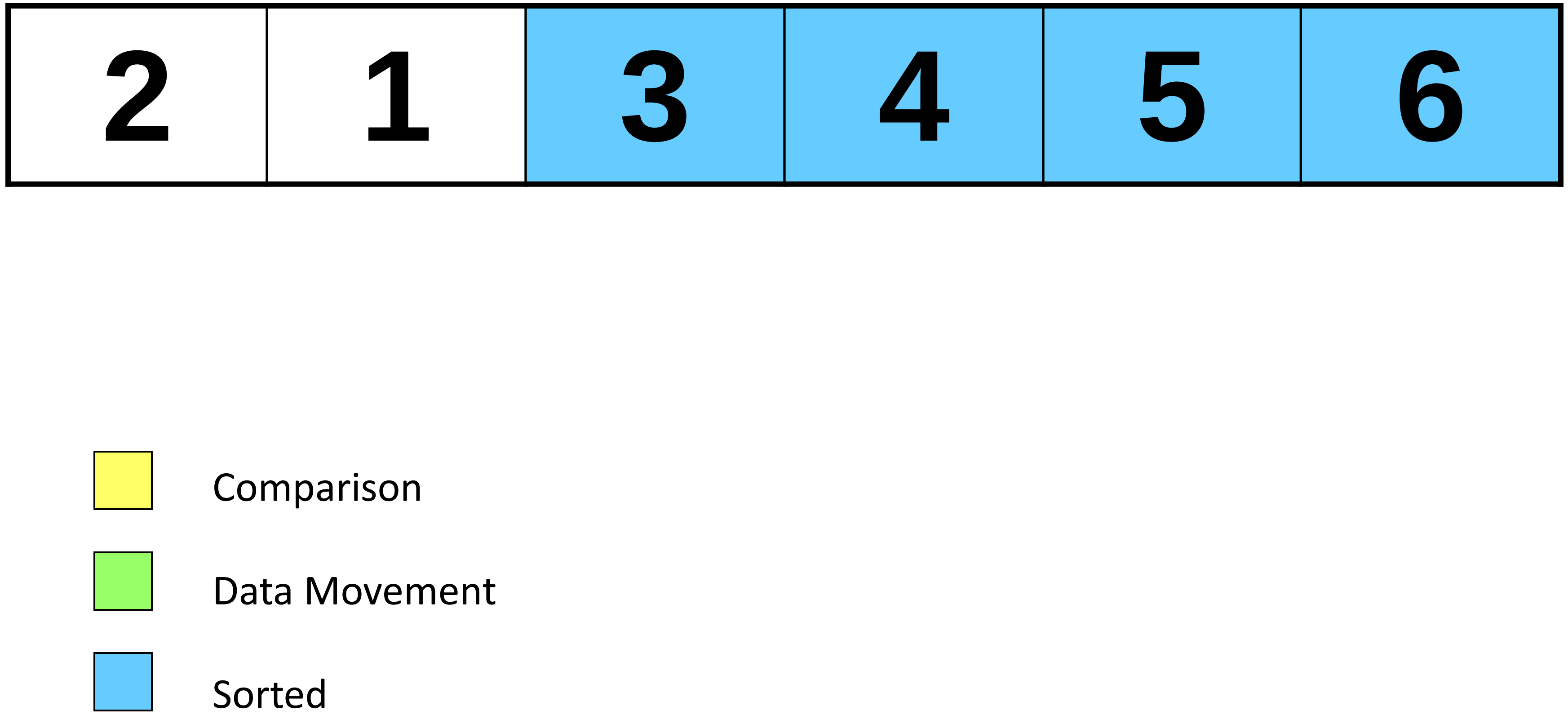


Comparison

Data Movement

Sorted

Selection Sort



Selection Sort

2	1	3	4	5	6

Comparison

Data Movement

Sorted

Selection Sort

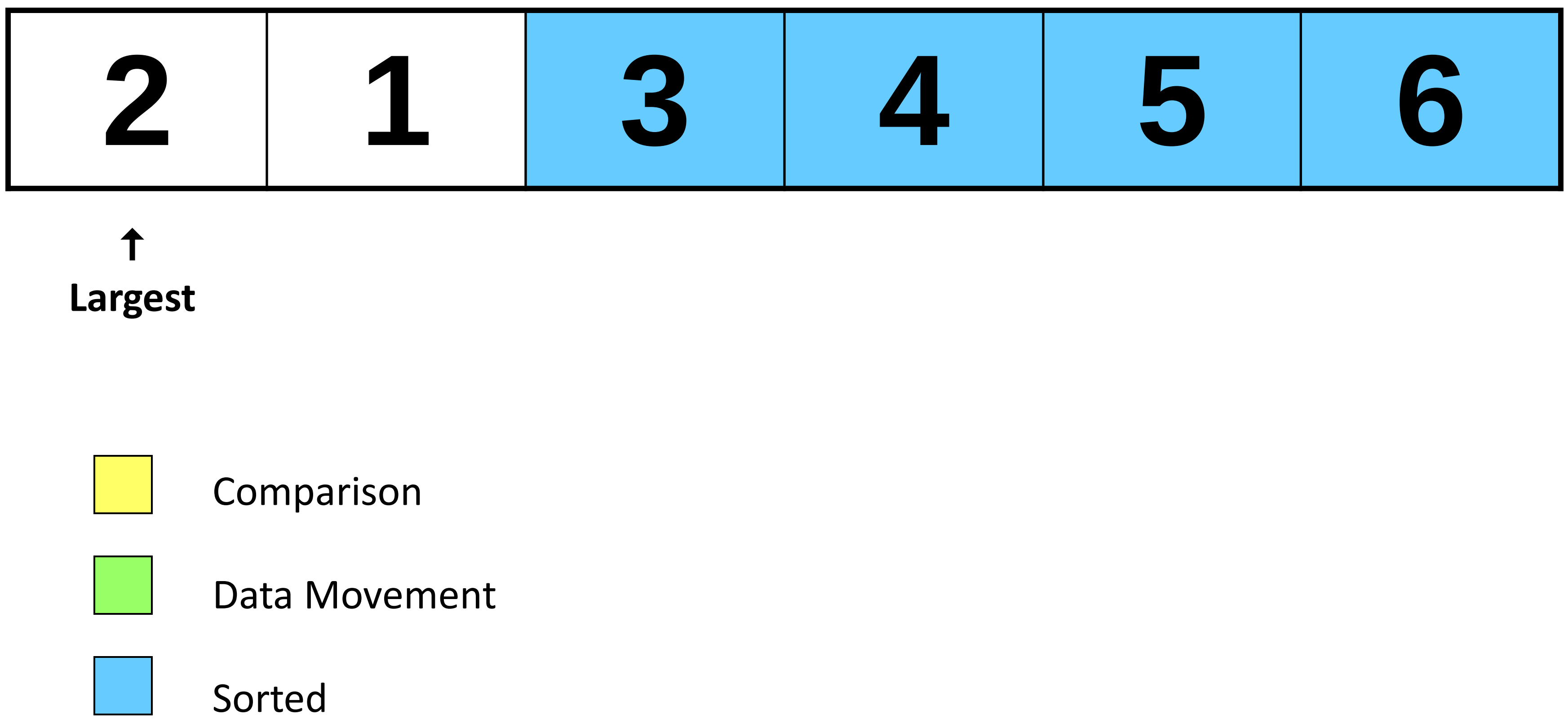
2	1	3	4	5	6

Comparison

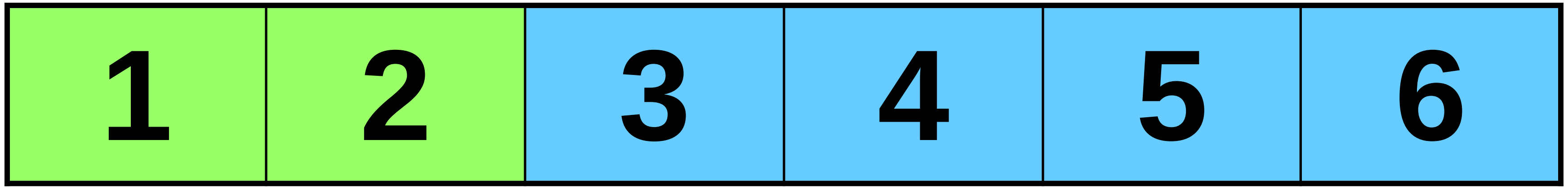
Data Movement

Sorted

Selection Sort



Selection Sort



Comparison

Data Movement

Sorted

Selection Sort

1	2	3	4	5	6

DONE!

Comparison

Data Movement

Sorted

Recurrence relation

$$T(n) = T(n-1) + n$$

```python
# Return minimum index
def minIndex( a , i , j ):
    if i == j:
        return i

    # Find minimum of remaining
elements
    k = minIndex(a, i + 1, j)

    # Return minimum of current
    # and remaining.
    return (i if a[i] < a[k] else k)
```

```python
def recurSelectionSort(a, n, index = 0):

    # Return when starting and
    # size are same
    if index == n:
        return -1

    # calling minimum index function
    # for minimum index
    k = minIndex(a, index, n-1)

    # Swapping when index and minimum
    # index are not same
    if k != index:
        a[k], a[index] = a[index], a[k]

    # Recursively calling selection
    # sort function
    recurSelectionSort(a, n, index + 1)
```

Selection Sort: Complexity Analysis

Case 1: If the input list is already in sorted order

Number of comparisons:

$$C(n) = \sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}$$

Selection Sort: Complexity Analysis

Case 2: If the input list is sorted but in reverse order

Number of comparisons:

$$c(n) = \sum_{i=1}^{n-1} (n - i) = \frac{n(n-1)}{2}$$

Selection Sort: Complexity Analysis

Case 3: If the input list is in random order

Number of comparisons:

$$C(n) = \frac{n(n-1)}{2}$$

- Let p_i be the probability that the i^{th} smallest element is in the i^{th} position. Number of total swap operations $= (1 - p_i) \times (n - 1)$

where $p_1 = p_2 = p_3 = \cdots = p_n = \frac{1}{n}$

Selection Sort: Summary of Complexity analysis

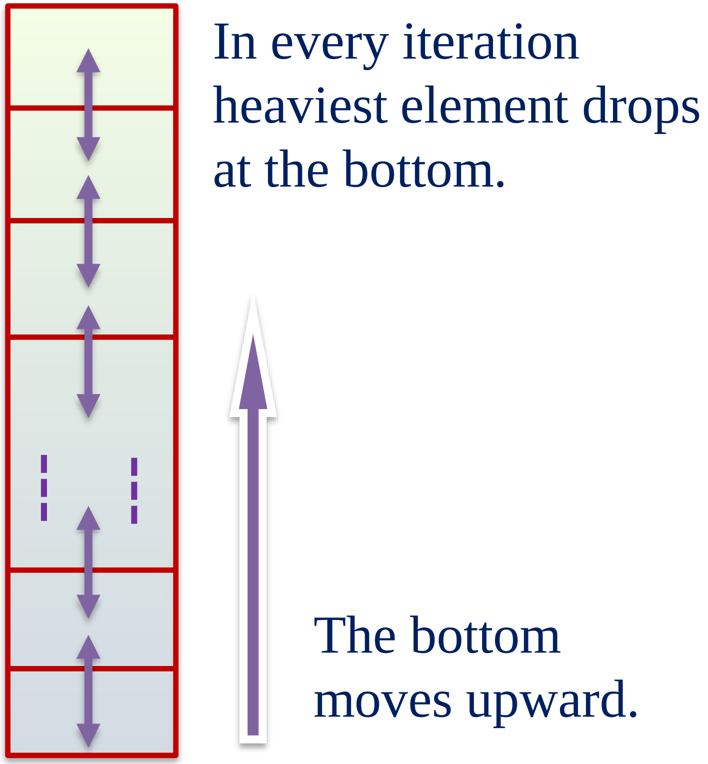
Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = \frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Best case
Case 2	$T(n) = \frac{(n-1)(n+3)}{2}$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) \approx \frac{(n-1)(2n+3)}{2}$ (Taking $n-1 \approx n$)	$T(n) = O(n^2)$	Average case

Is this a Stable sort ?

NO

Bubble Sort

Bubble Sort



The sorting process proceeds in several passes.

- In every pass we go on comparing neighbouring pairs and swap them if out of order.
- In every pass, the largest of the elements under considering will *bubble* to the top (i.e., the right).

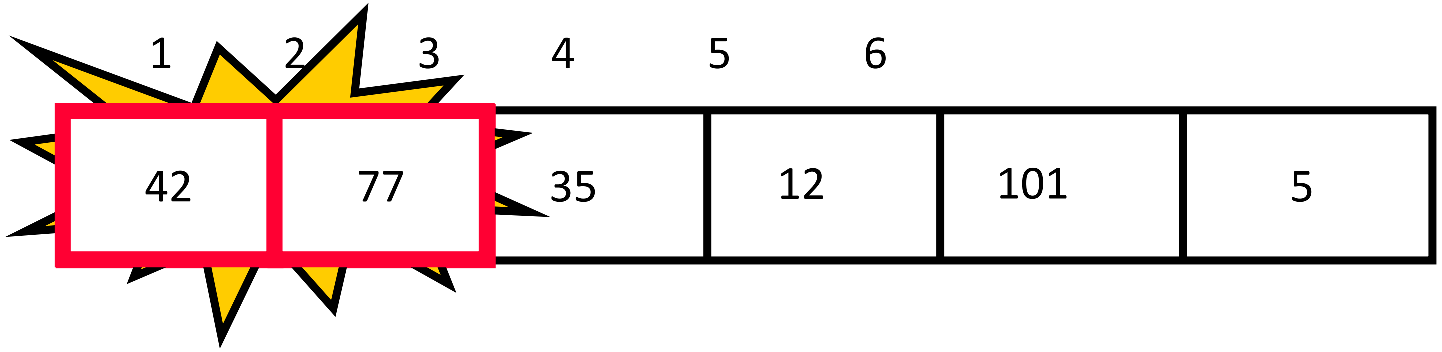
"Bubbling Up" the Largest Element

- **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

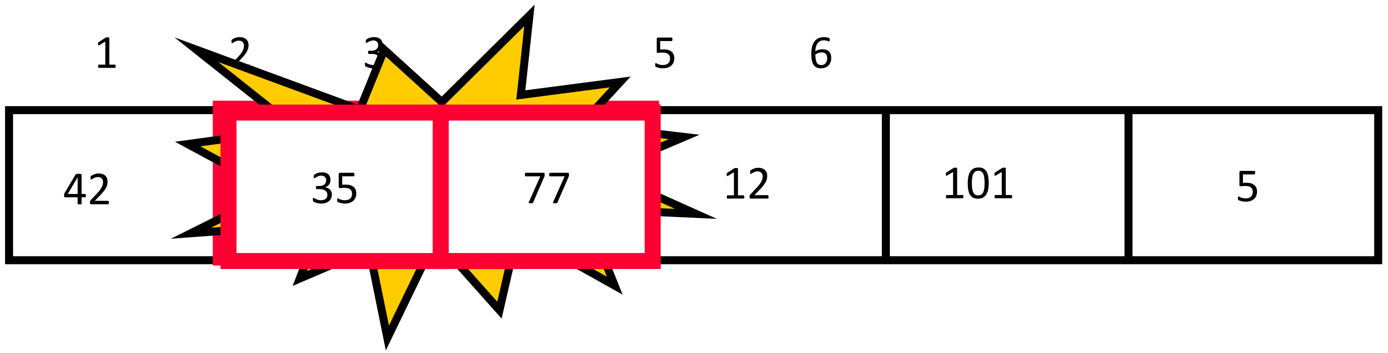
"Bubbling Up" the Largest Element

- **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**



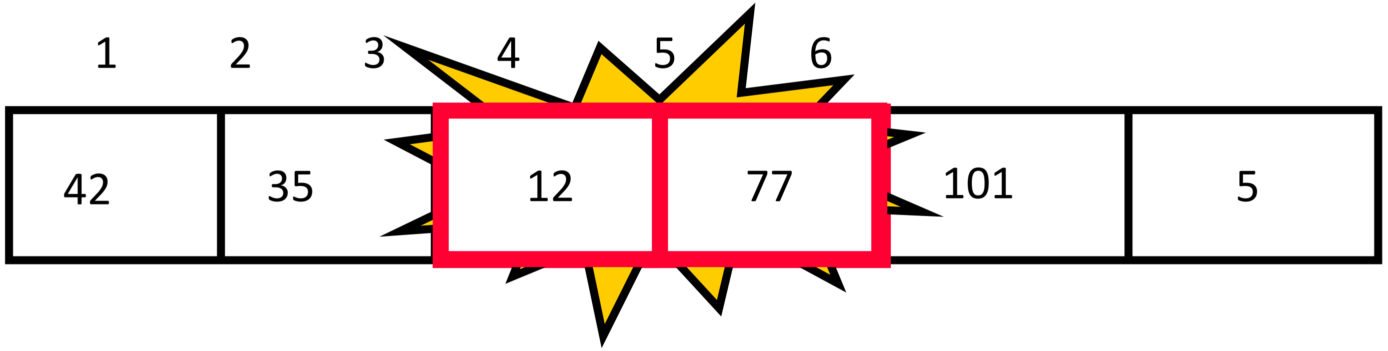
"Bubbling Up" the Largest Element

- **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

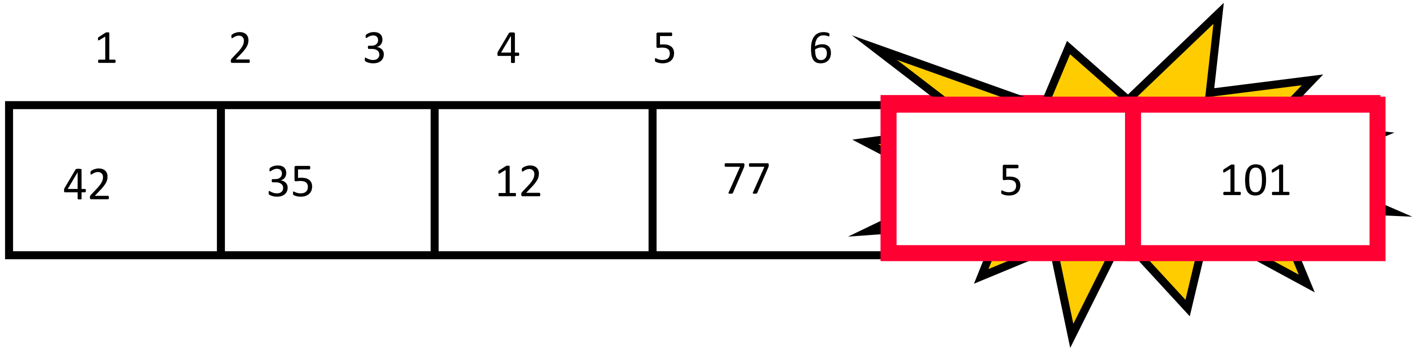
- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

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

No need to swap

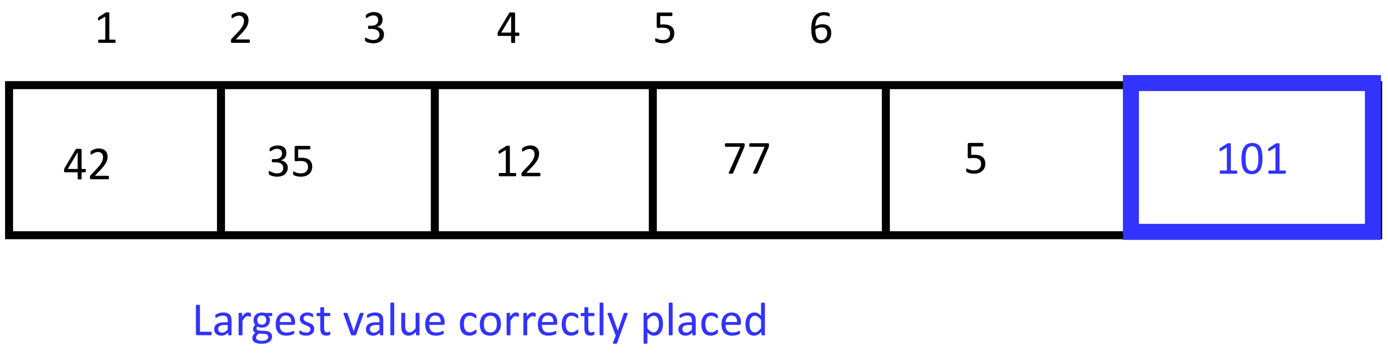
"Bubbling Up" the Largest Element

- **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**



"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**



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**

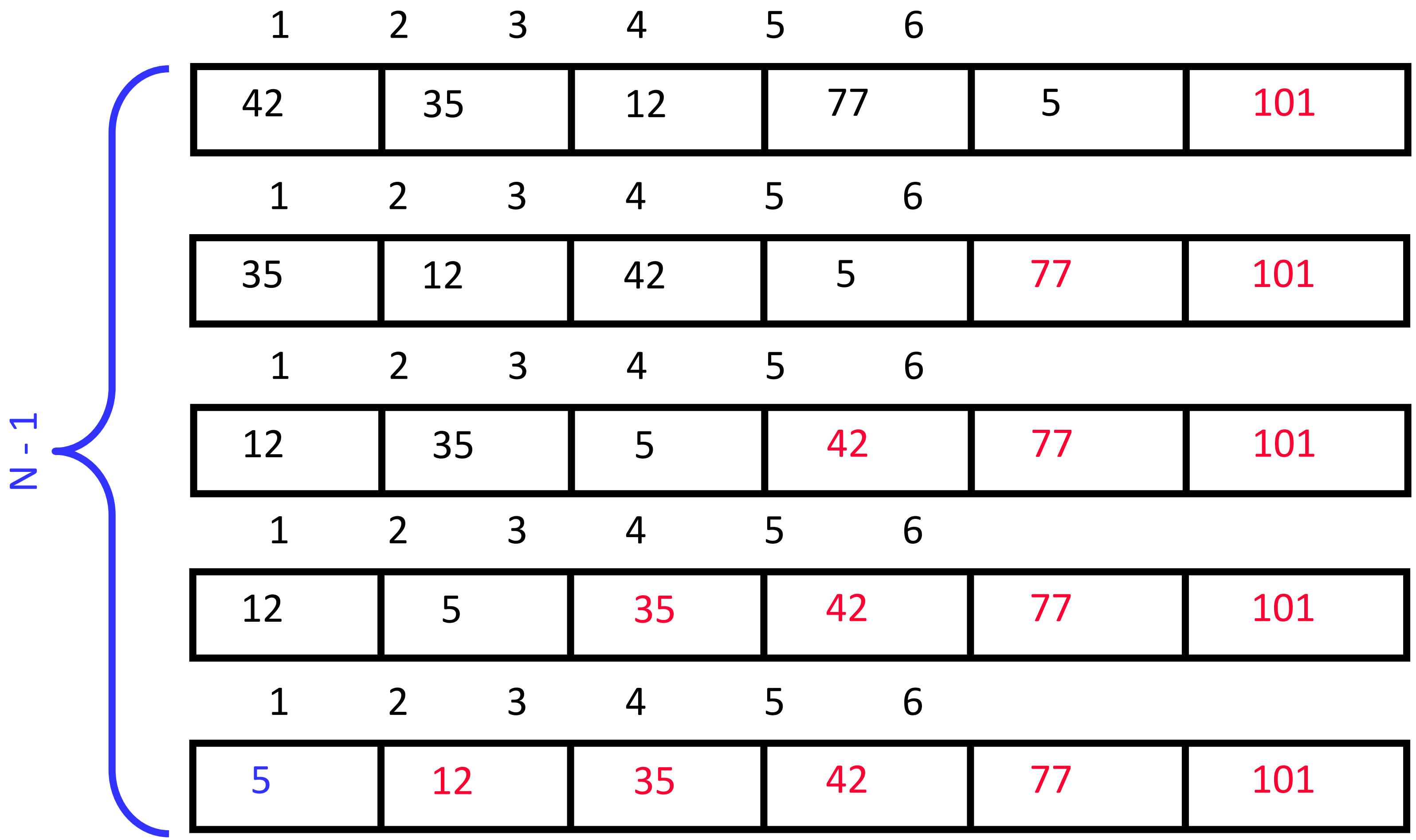
1	2	3	4	5	6
42	35	12	77	5	$\boxed{101}$

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 place all N elements.**

“Bubbling” All the Elements

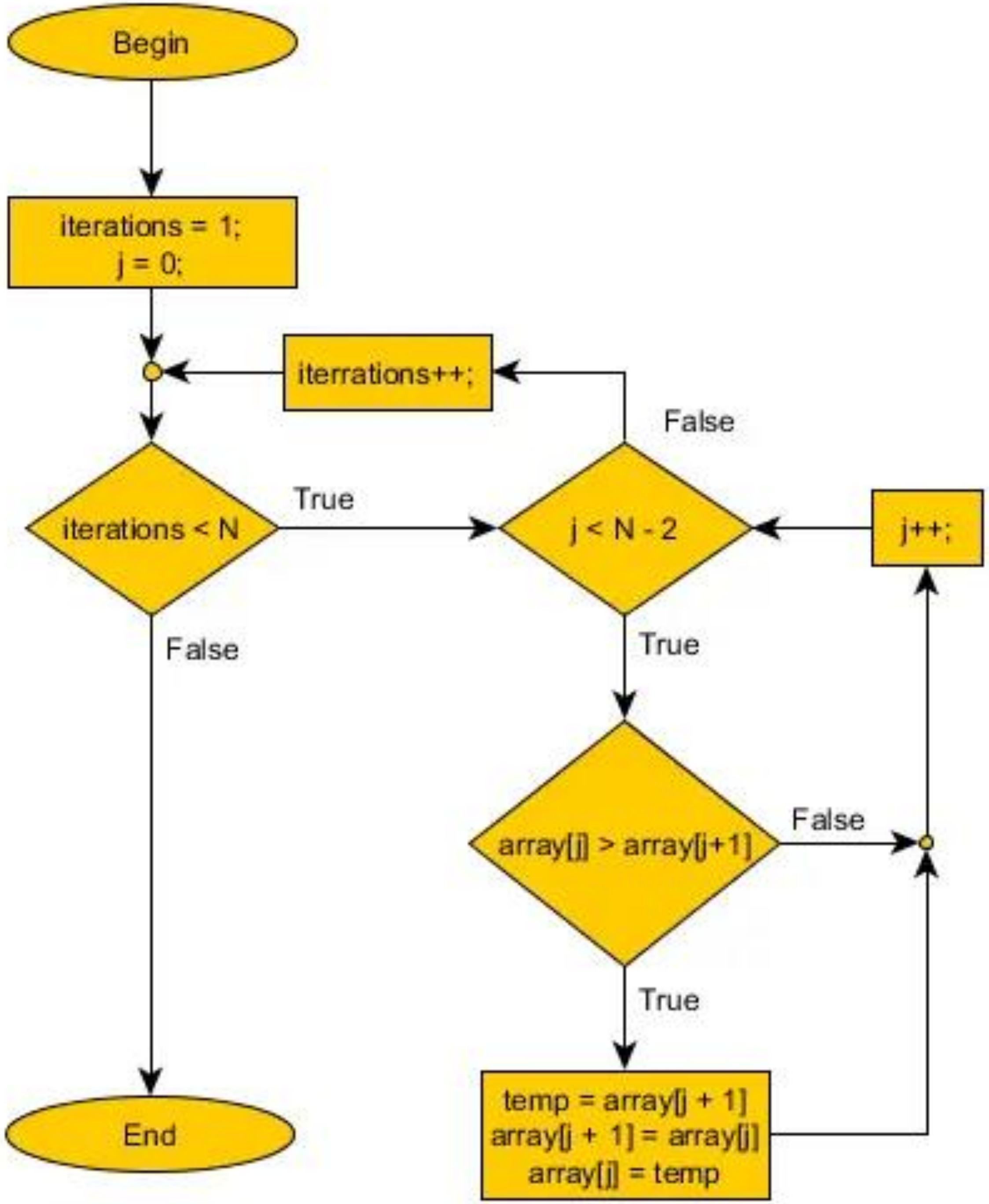


Bubble Sort

How the passes proceed?

- In **pass 1**, we consider index **0 to n-1.**
- In **pass 2**, we consider index **0 to n-2.**
- In **pass 3**, we consider index **0 to n-3.**
- ……
- ……
- In **pass n-1**, we consider index **0 to 1.**

Algorithm/Procedure/ Flowchart



Algorithm/Procedure/ Flowchart

Pseudo code: Bubble Sort(Array a[ ])

```
1.  begin
2.      for i = 0 to n – 1
3.          for j = 0 to n – i-1
4.              if (a[j] > a[j + 1]) then
5.                  Swap (a[j], a[j + 1])
6.  end
```

improvised pseudocode

```
procedure bubbleSort( list : array of items )

 loop = list.count;

 for i = 0 to loop-1 do:
   swapped = false

   for j = 0 to loop-1 do:

     /* compare the adjacent elements */
     if list[j] > list[j+1] then
       /* swap them */
       swap( list[j], list[j+1] )
       swapped = true
     end if

   end for

   /*if no number was swapped that means
   array is sorted now, break the loop.*/

   if(not swapped) then
     break
   end if

 end for

end procedure return list
```

Bubble Sort - Example

Pass: 1

x: 3 | 12 | -5 | 6 | 72 | 21 | -7 | 45

x: 3 | 12 | -5 | 6 | 72 | 21 | -7 | 45

x: 3 | -5 | 12 | 6 | 72 | 21 | -7 | 45

x: 3 | -5 | 6 | 12 | 72 | 21 | -7 | 45

x: 3 | -5 | 6 | 12 | 72 | 21 | -7 | 45

x: 3 | -5 | 6 | 12 | 21 | 72 | -7 | 45

x: 3 | -5 | 6 | 12 | 21 | -7 | 72 | 45

x: 3 | -5 | 6 | 12 | 21 | -7 | 45 | 72

Bubble Sort - Example

Pass: 2

x: | 3 | -5 | 6 | 12 | 21 | -7 | 45 | 72 |

x: | -5 | 3 | 6 | 12 | 21 | -7 | 45 | 72 |

x: | -5 | 3 | 6 | 12 | 21 | -7 | 45 | 72 |

x: | -5 | 3 | 6 | 12 | 21 | -7 | 45 | 72 |

x: | -5 | 3 | 6 | 12 | 21 | -7 | 45 | 72 |

x: | -5 | 3 | 6 | 12 | -7 | 21 | 45 | 72 |

x: | -5 | 3 | 6 | 12 | -7 | 21 | 45 | 72 |

Comparison will be reduce by 1 in each pass
Pass will continue upto n-1 number

Bubble Sort

```
void swap(int *x, int *y)
{
      int tmp = *x;
      *x = *y;
      *y = tmp;
}

void bubble_sort(int x[], int n)
{
      int i,j;
      for (i=n-1; i>0; i--)
         for (j=0; j<i; j++)
            if (x[j] > x[j+1])
               swap(&x[j],&x[j+1]);
}
```

Bubble Sort

```
int main()
{
   int x[ ]={-45,89,-65,87,0,3,-23,19,56,21,76,-50};
   int i;
   for(i=0;i<12;i++)
      printf("%d ",x[i]);
   printf("\n");
   bubble_sort(x,12);
   for(i=0;i<12;i++)
      printf("%d ",x[i]);
   printf("\n");
}
```

OUTPUT

-45 89 -65 87 0 3 -23 19 56 21 76 -50

-65 -50 -45 -23 0 3 19 21 56 76 87 89

Pseudocode

```
# Bubble sort in Python
def bubbleSort(array):

    # loop to access each array element
    for i in range(len(array)):

        # loop to compare array elements
        for j in range(0, len(array) - i - 1):

            # compare two adjacent elements
            # change > to < to sort in descending order
            if array[j] > array[j + 1]:

                # swapping elements if elements are not in the intended order
                temp = array[j]
                array[j] = array[j+1]
                array[j+1] = temp

data = [-2, 45, 0, 11, -9]

bubbleSort(data)

print('Sorted Array in Ascending Order:')
print(data)
```

More example

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”!**

1	2	3	4	5	6
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.”**

An Animated Example

N 8

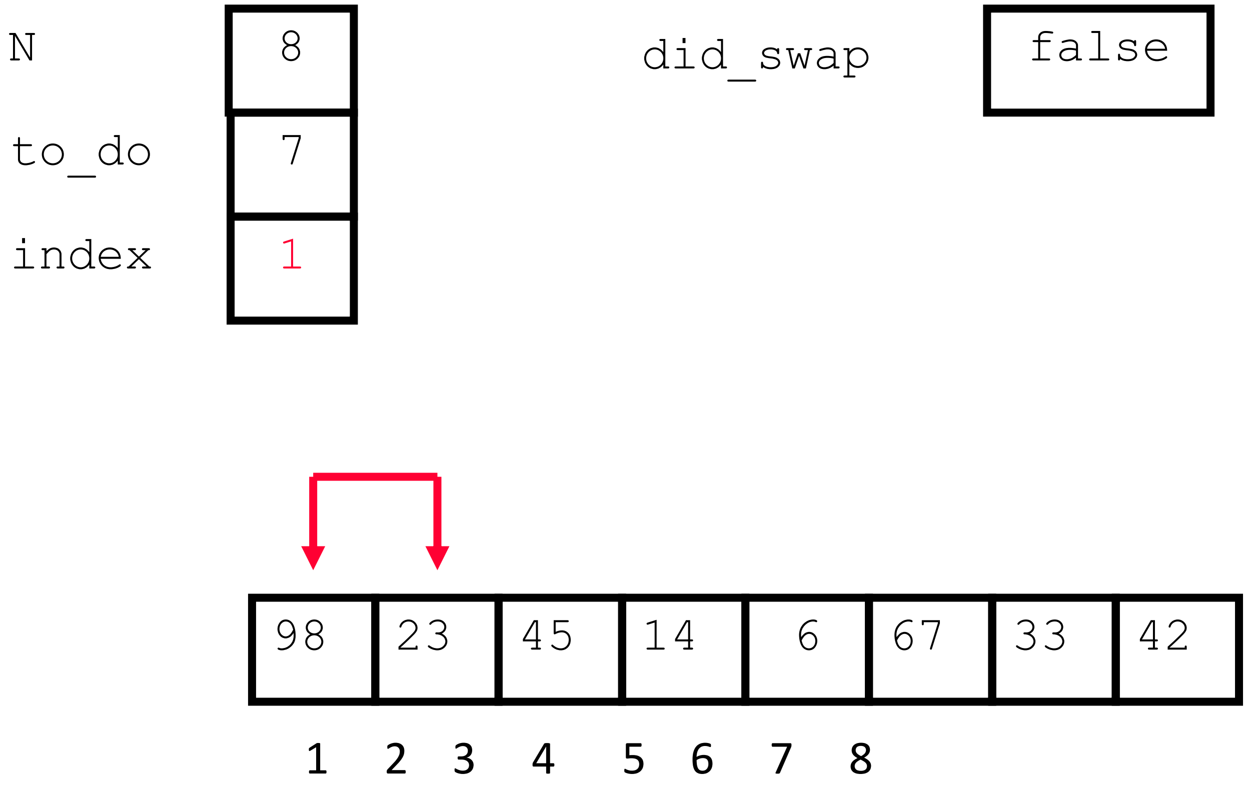
to_do 7

index

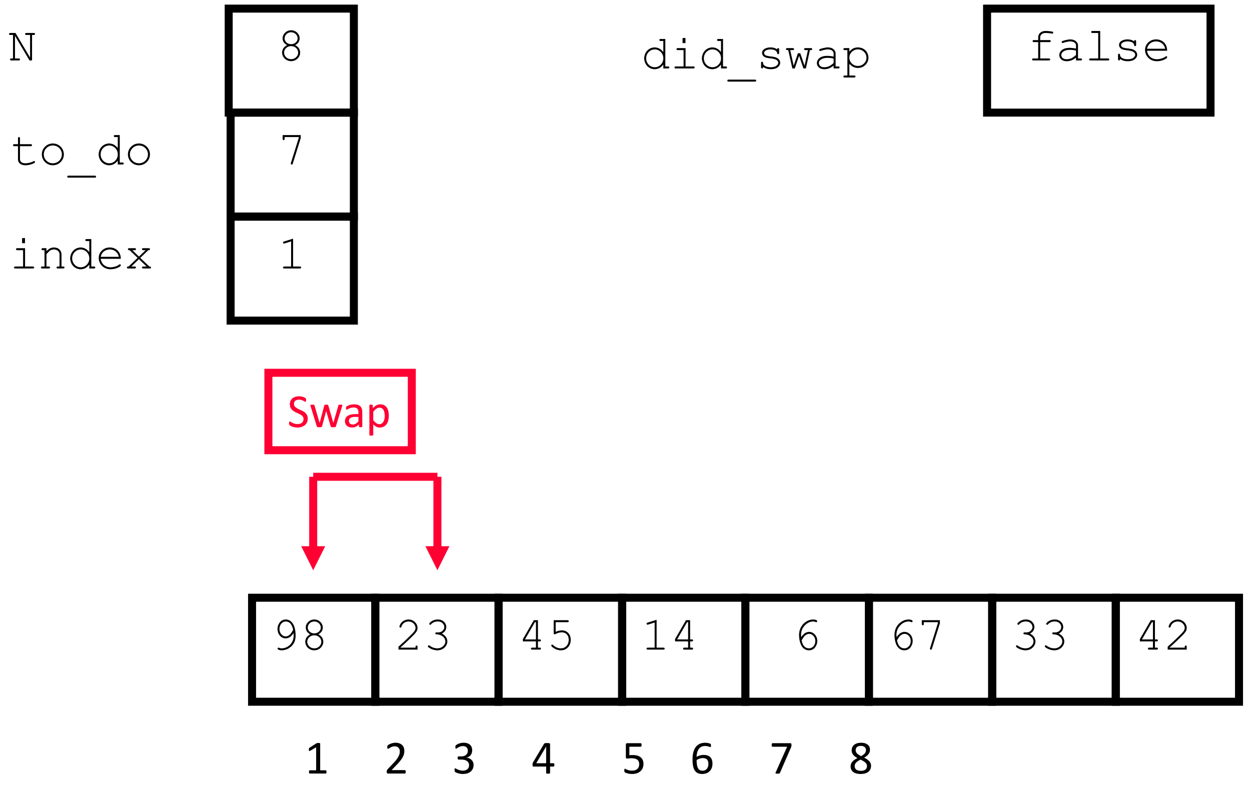
did_swap true

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

An Animated Example



An Animated Example



An Animated Example

N	8
to_do	7
index	1

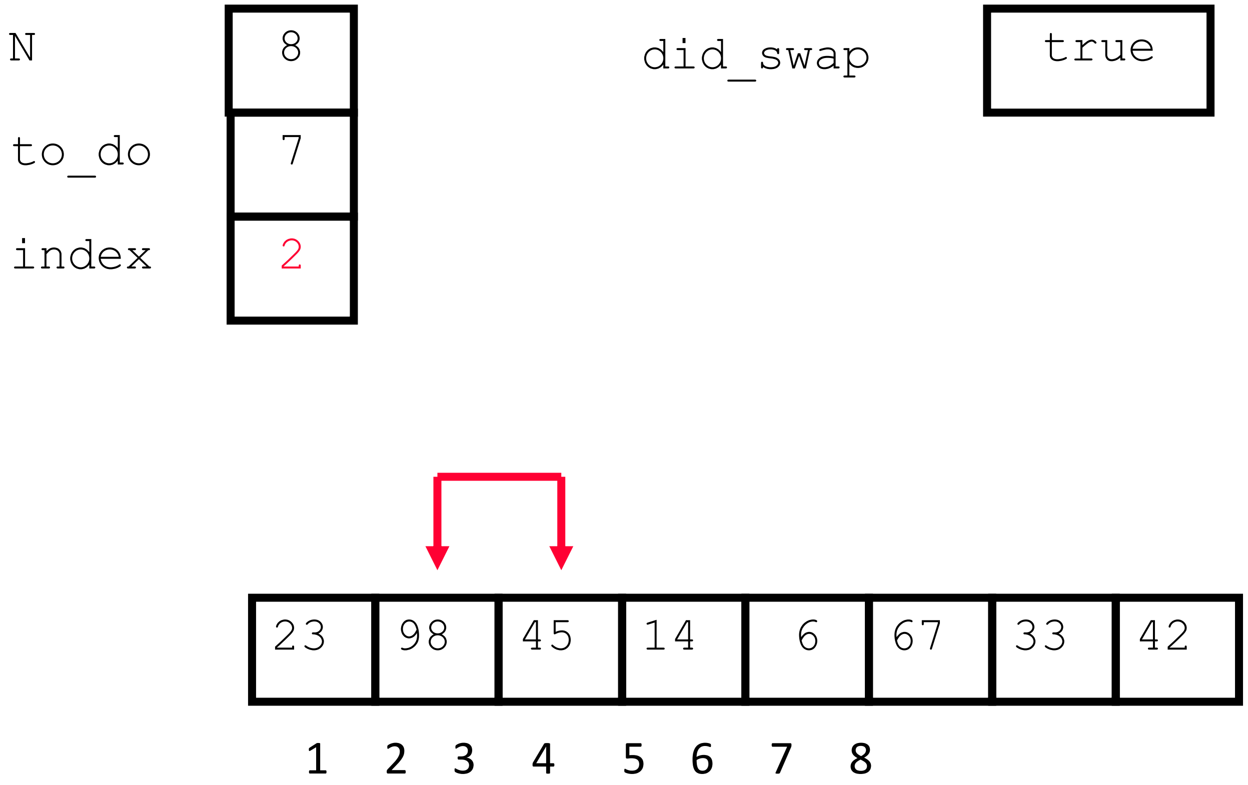
did_swap: true

Swap

23	98	45	14	6	67	33	42

1 2 3 4 5 6 7 8

An Animated Example



An Animated Example

N	8
to_do	7
index	2

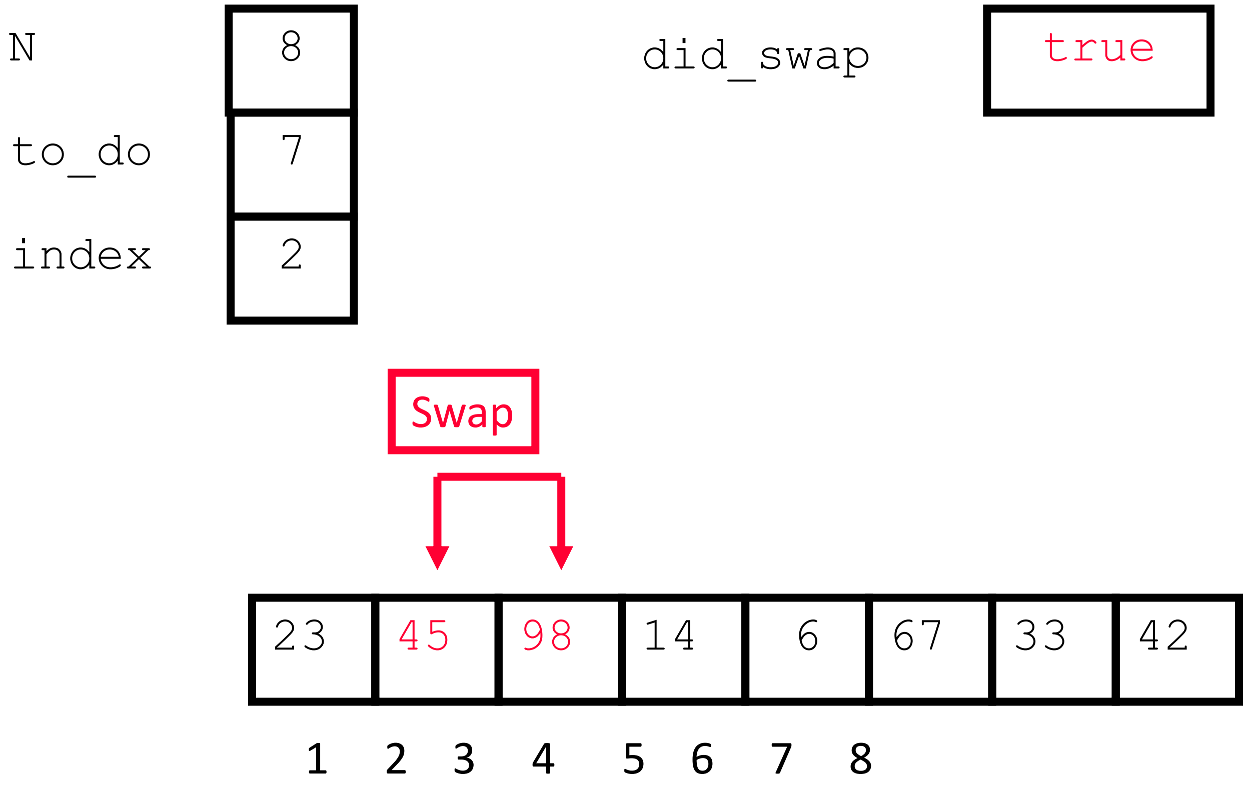
did_swap	true

Swap

23	98	45	14	6	67	33	42

1 2 3 4 5 6 7 8

An Animated Example



An Animated Example

N	8
to_do	7
index	3

did_swap	true

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

An Animated Example

N	8
to_do	7
index	3

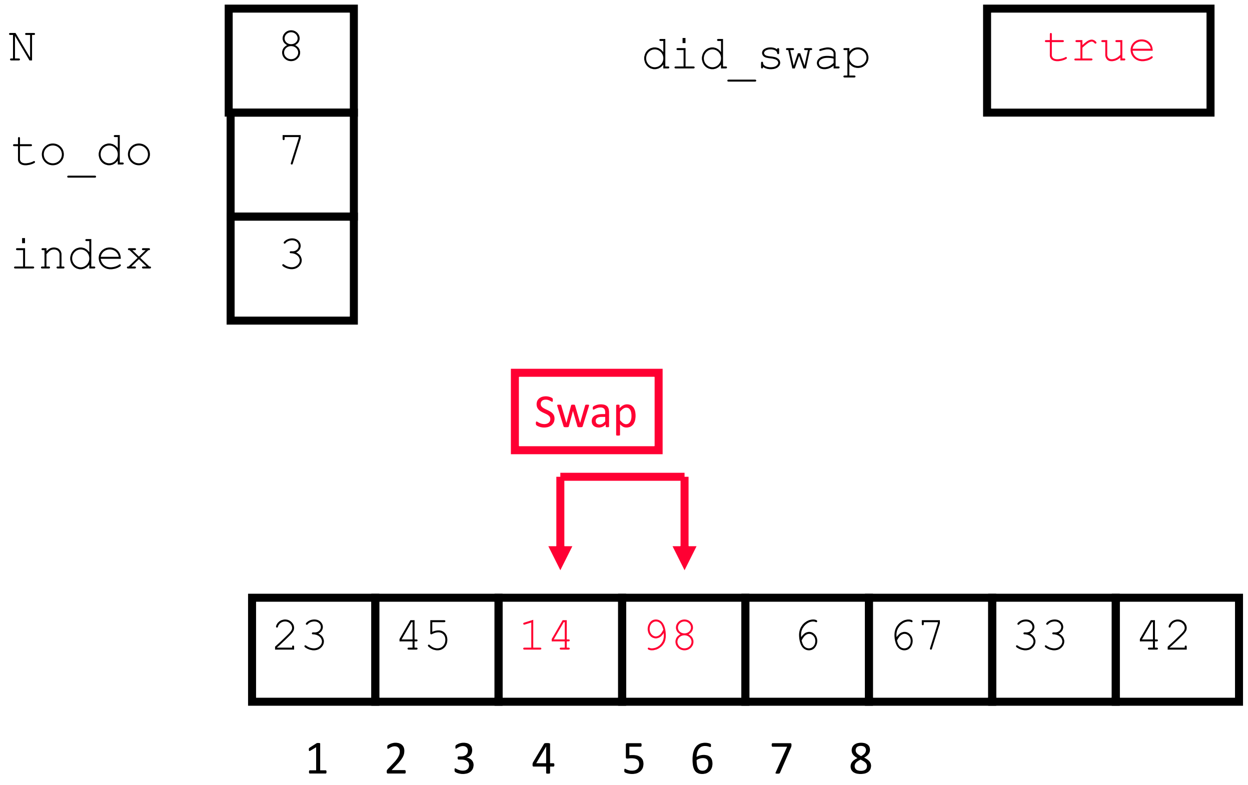
did_swap	true

Swap

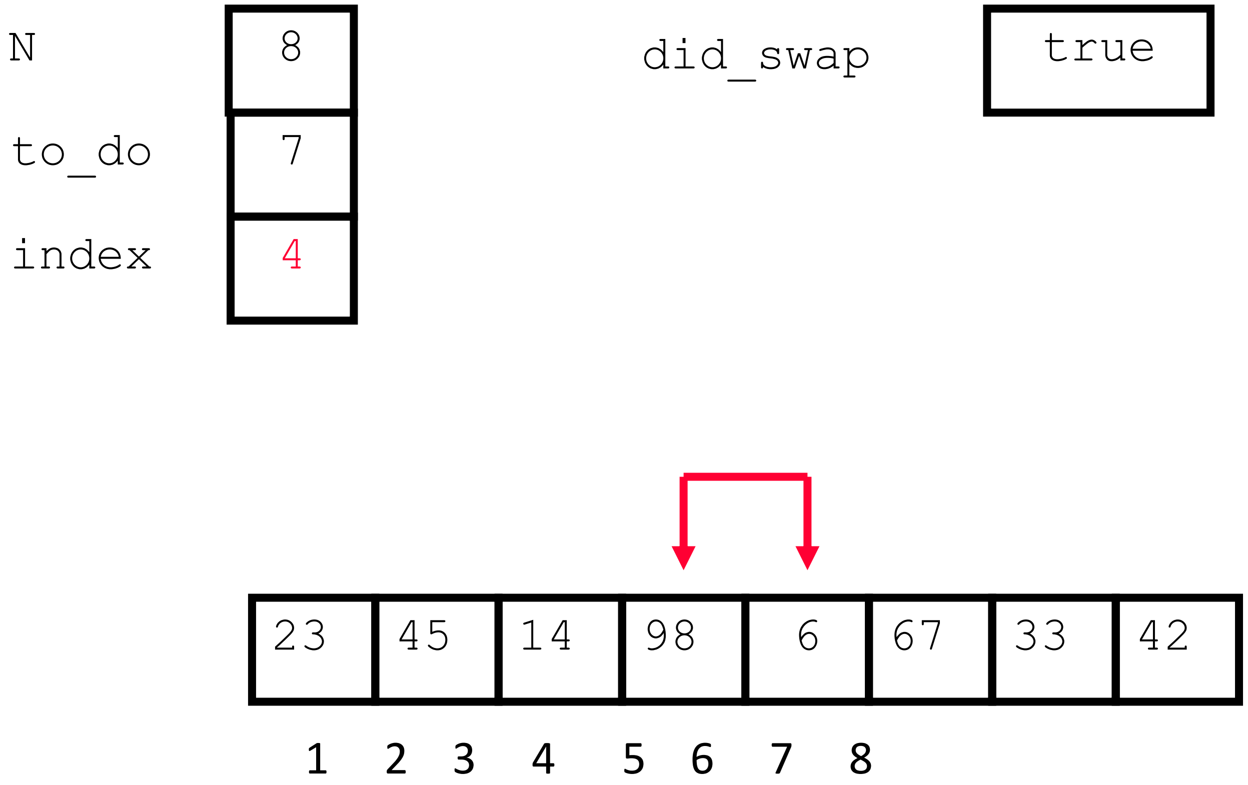
23	45	98	14	6	67	33	42

1 2 3 4 5 6 7 8

An Animated Example



An Animated Example



An Animated Example

N 8

to_do 7

index 4

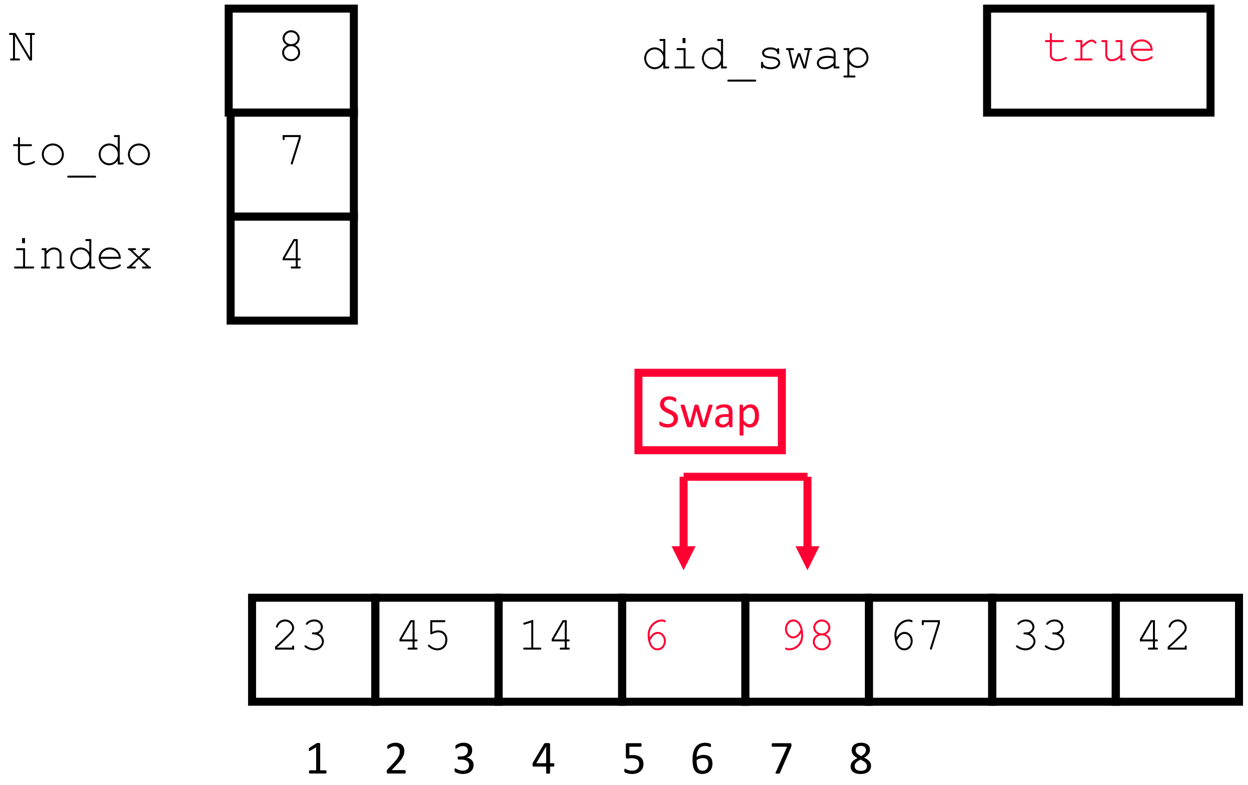
did_swap true

Swap

23	45	14	98	6	67	33	42

1 2 3 4 5 6 7 8

An Animated Example



An Animated Example

N	8
to_do	7
index	5

did_swap	true

23	45	14	6	98	67	33	42

1 2 3 4 5 6 7 8

An Animated Example

N 8

to_do 7

index 5

did_swap true

Swap

23	45	14	6	98	67	33	42

1 2 3 4 5 6 7 8

An Animated Example

N	8
to_do	7
index	5

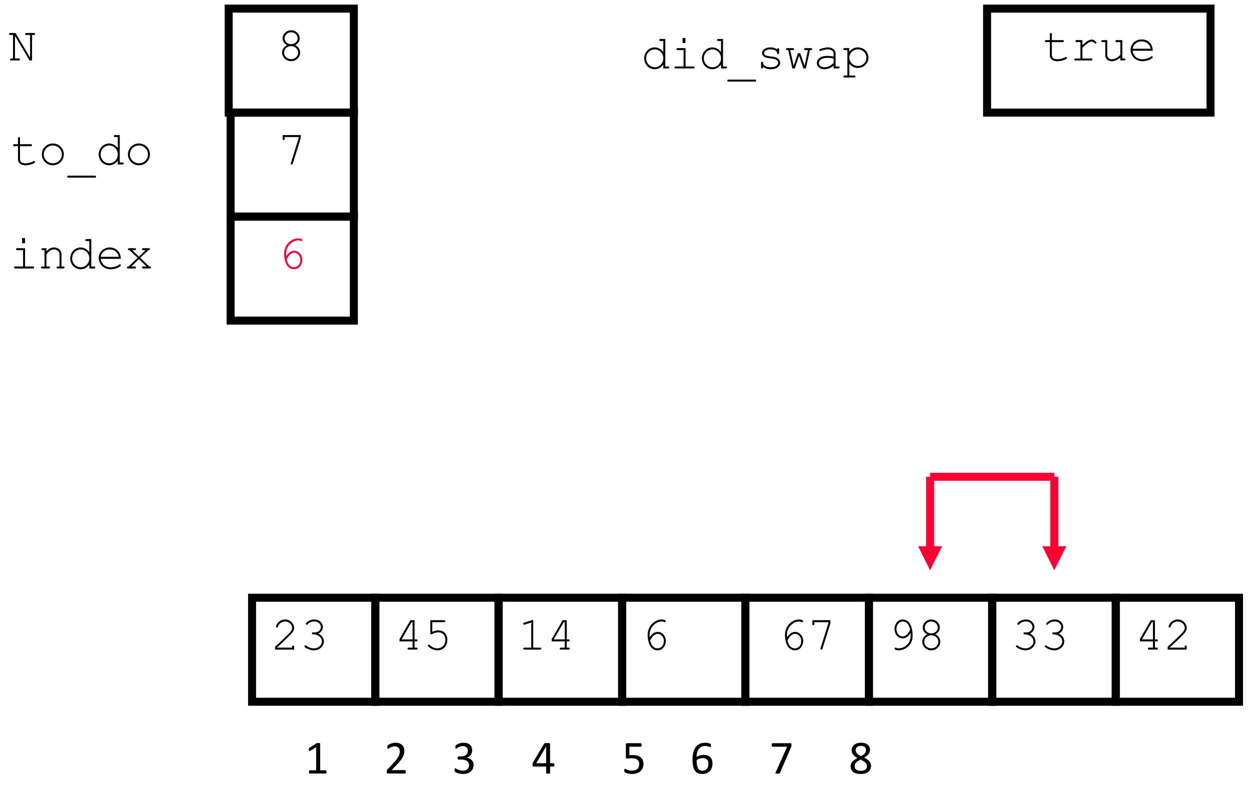
did_swap: true

Swap

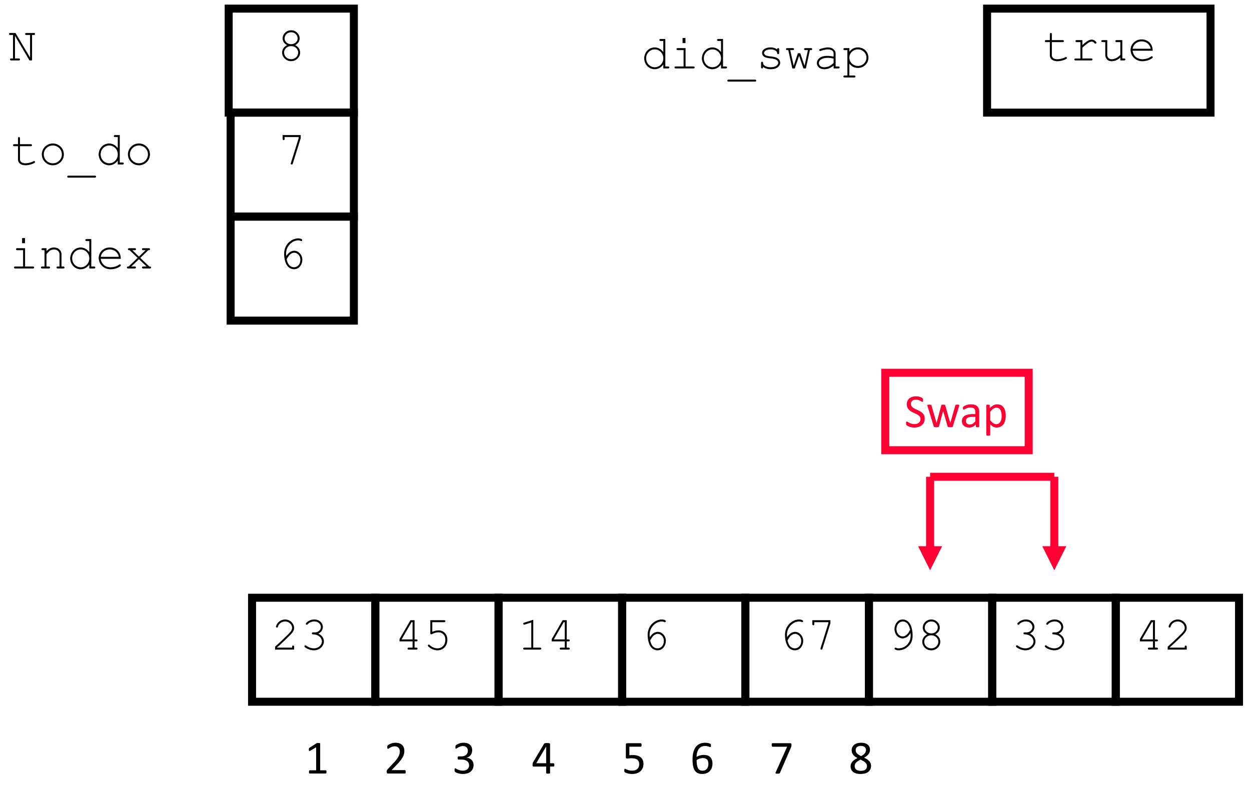
23	45	14	6	67	98	33	42

1 2 3 4 5 6 7 8

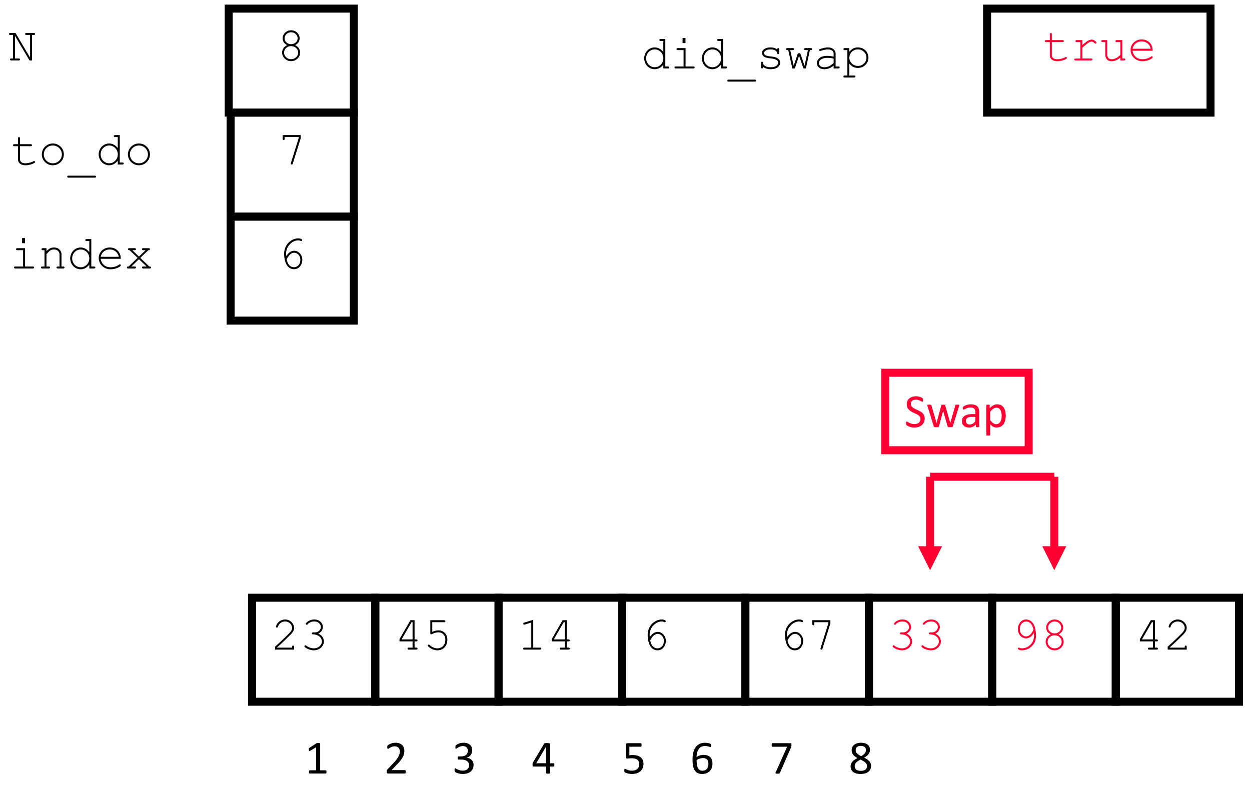
An Animated Example



An Animated Example



An Animated Example



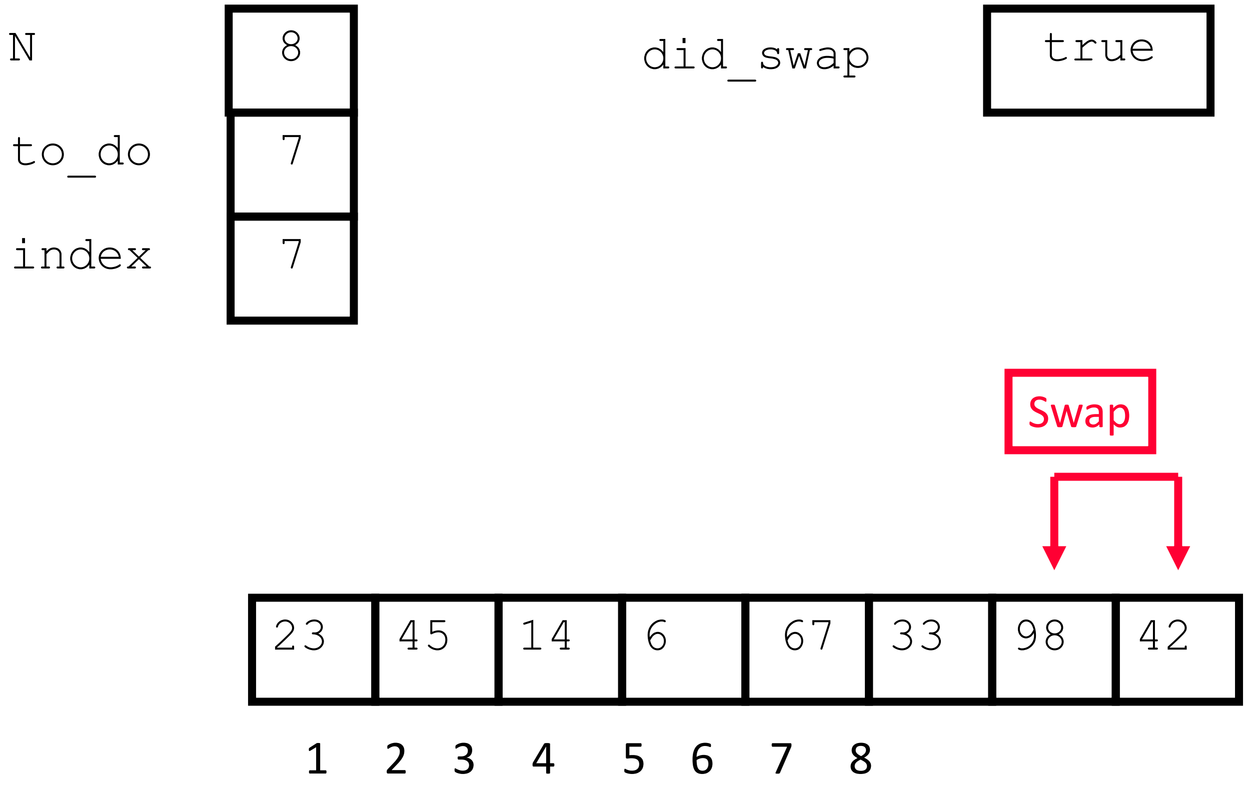
An Animated Example

N	8
to_do	7
index	7

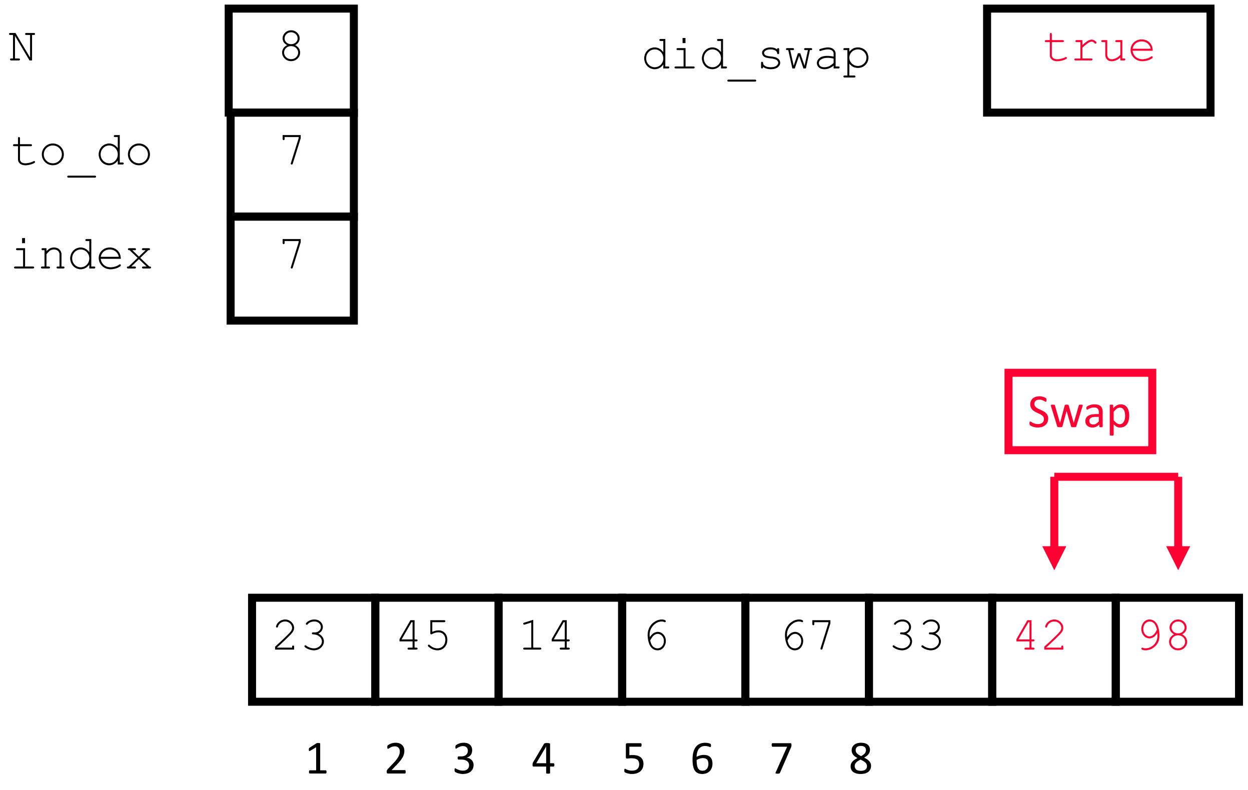
did_swap	true

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

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”

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

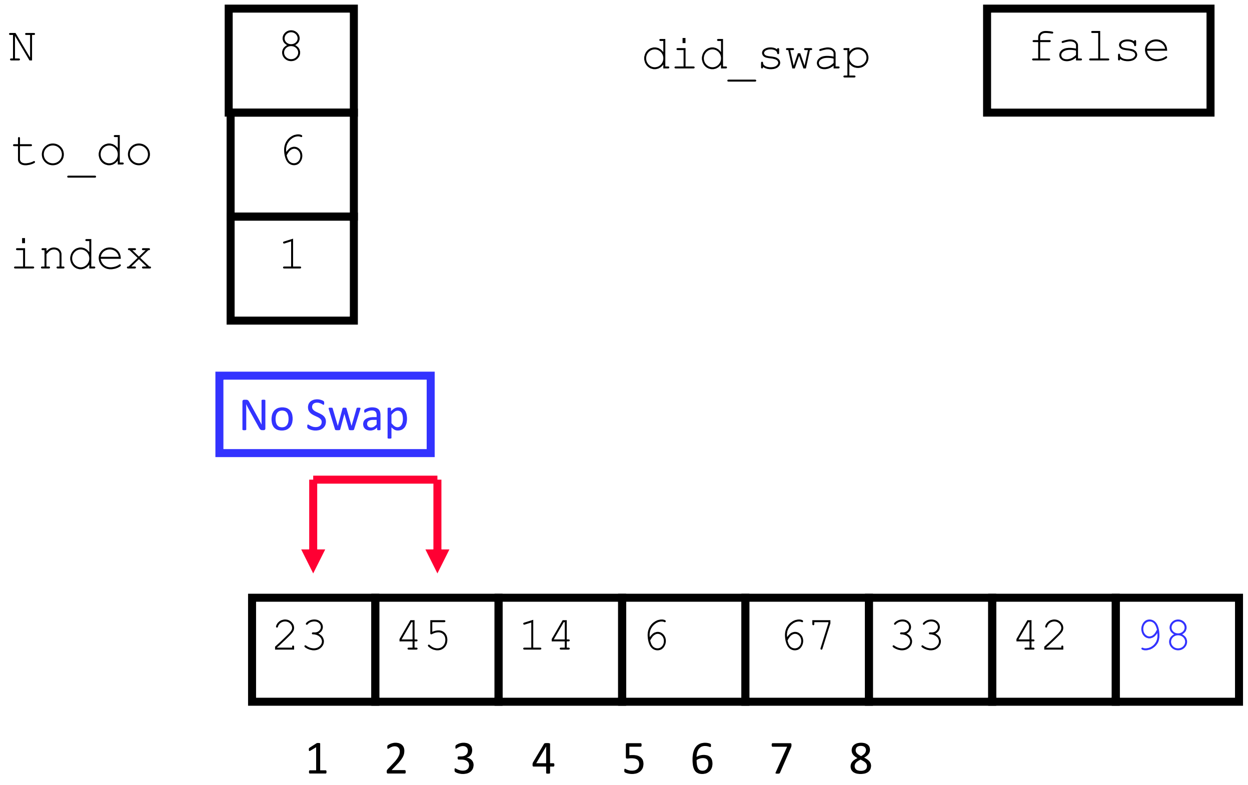
index 1

did_swap false

23	45	14	6	67	33	42	98

1 2 3 4 5 6 7 8

The Second “Bubble Up”



The Second "Bubble Up"

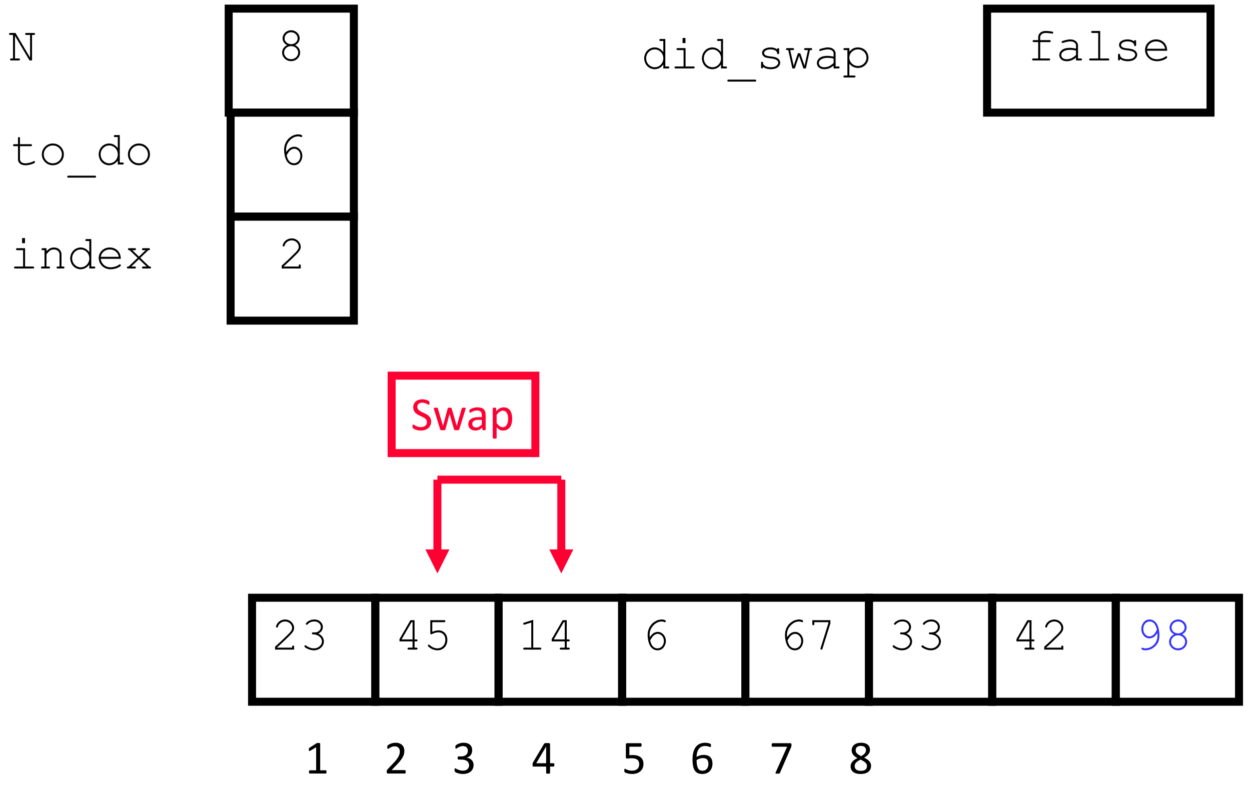
N	8
to_do	6
index	2

did_swap false

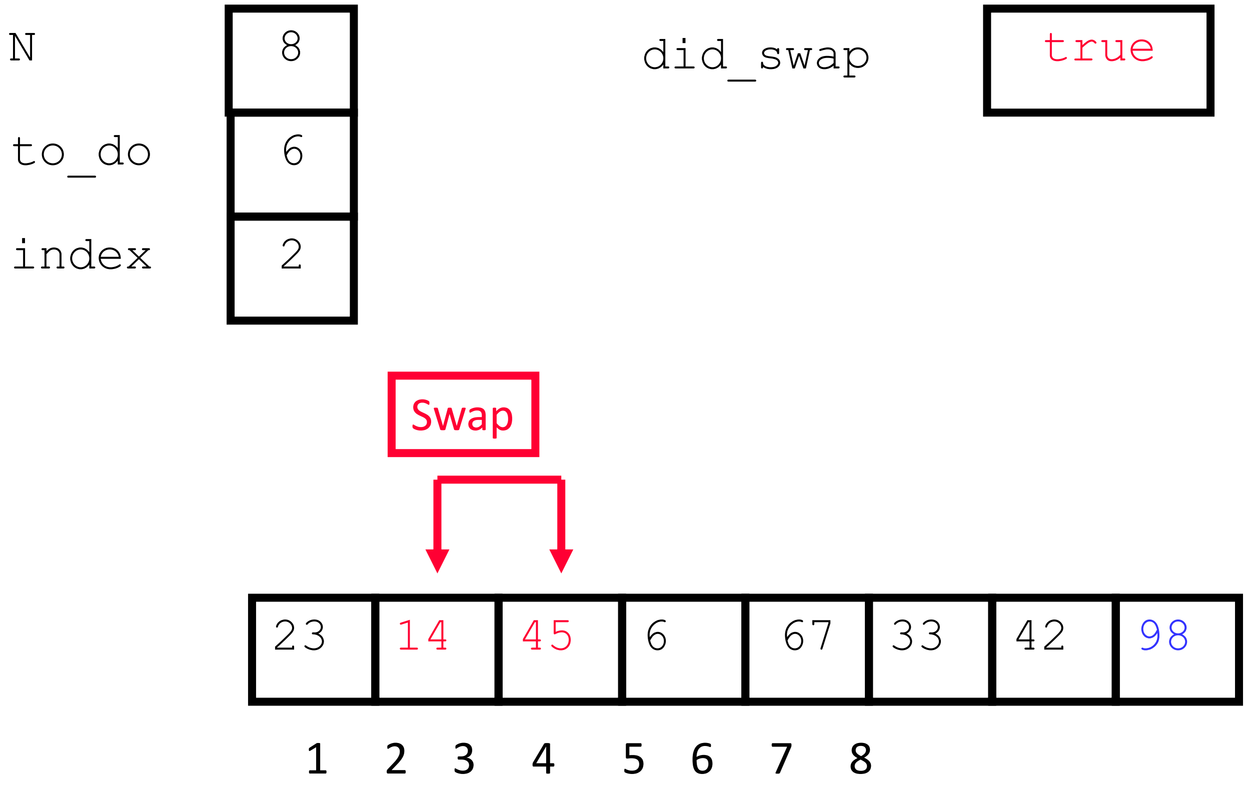
23	45	14	6	67	33	42	98

1 2 3 4 5 6 7 8

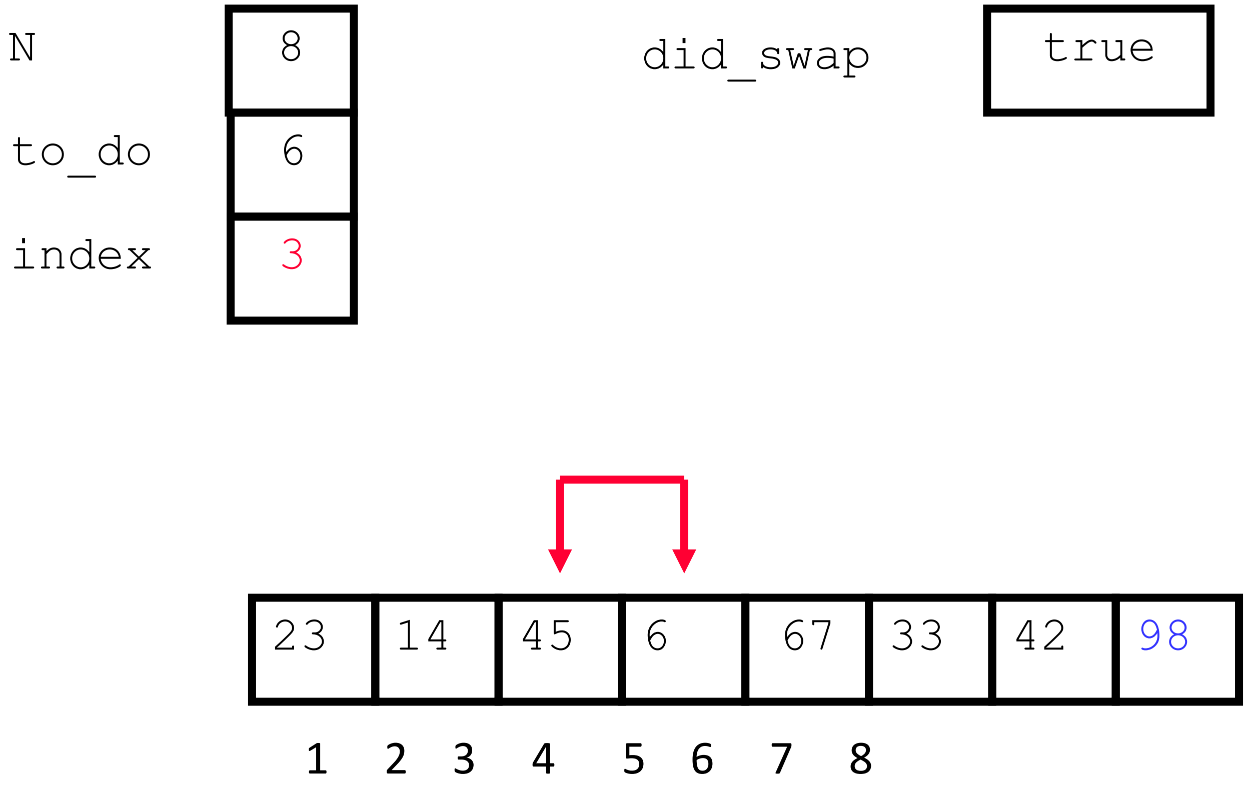
The Second "Bubble Up"



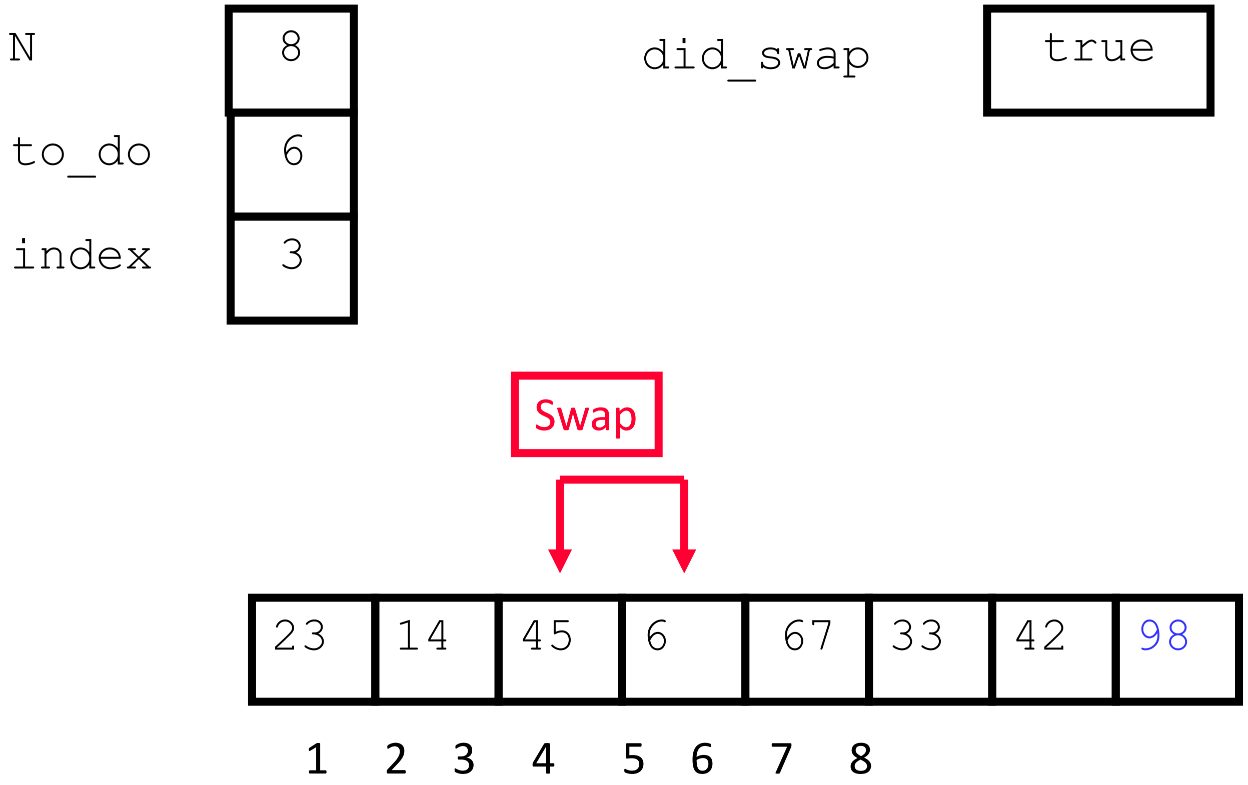
The Second “Bubble Up”



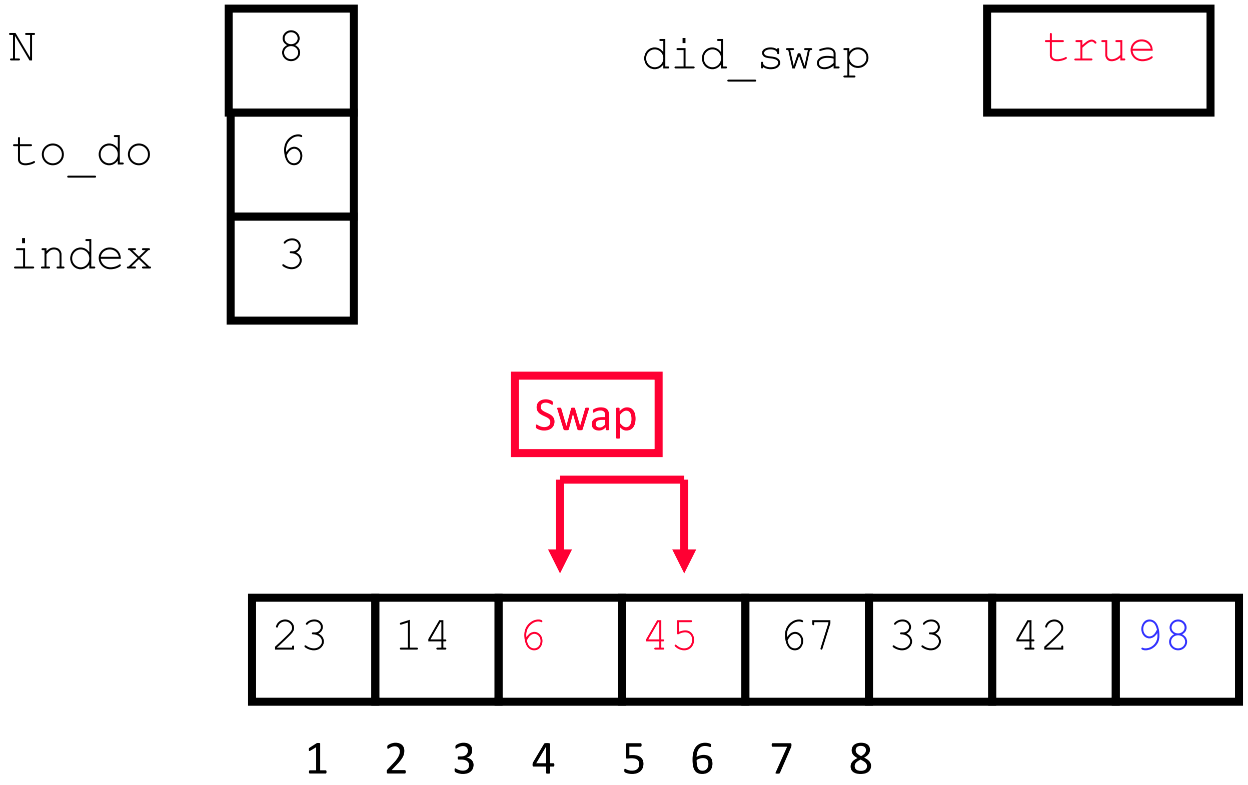
The Second "Bubble Up"



The Second "Bubble Up"



The Second "Bubble Up"



The Second “Bubble Up”

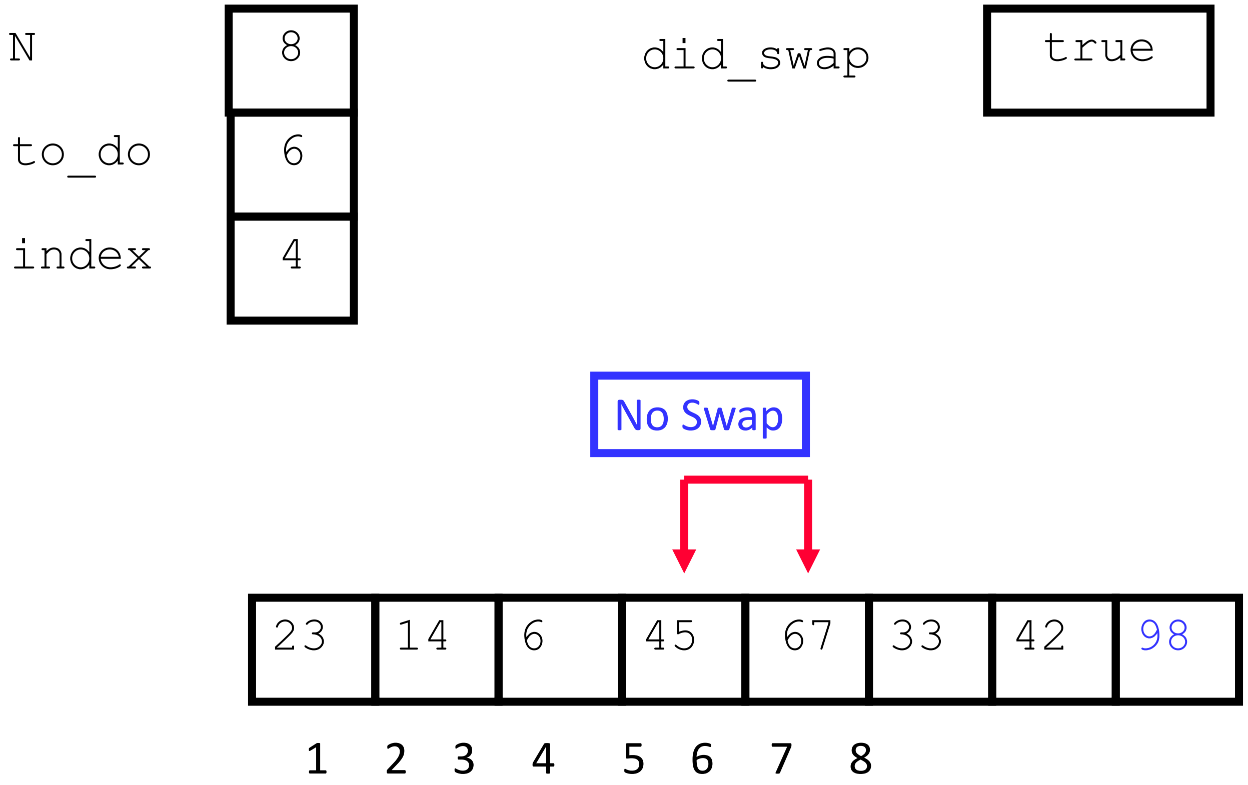
N	8
to_do	6
index	4

did_swap	true

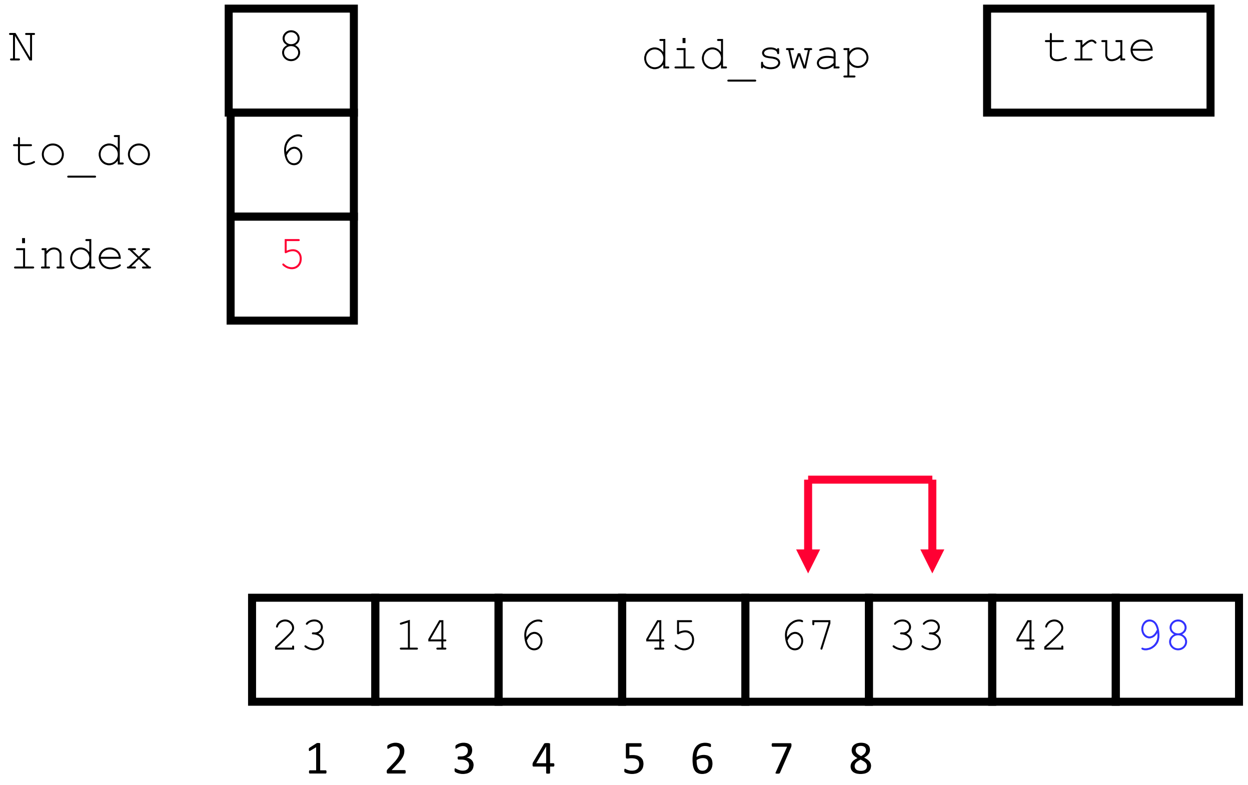
23	14	6	45	67	33	42	98

1 2 3 4 5 6 7 8

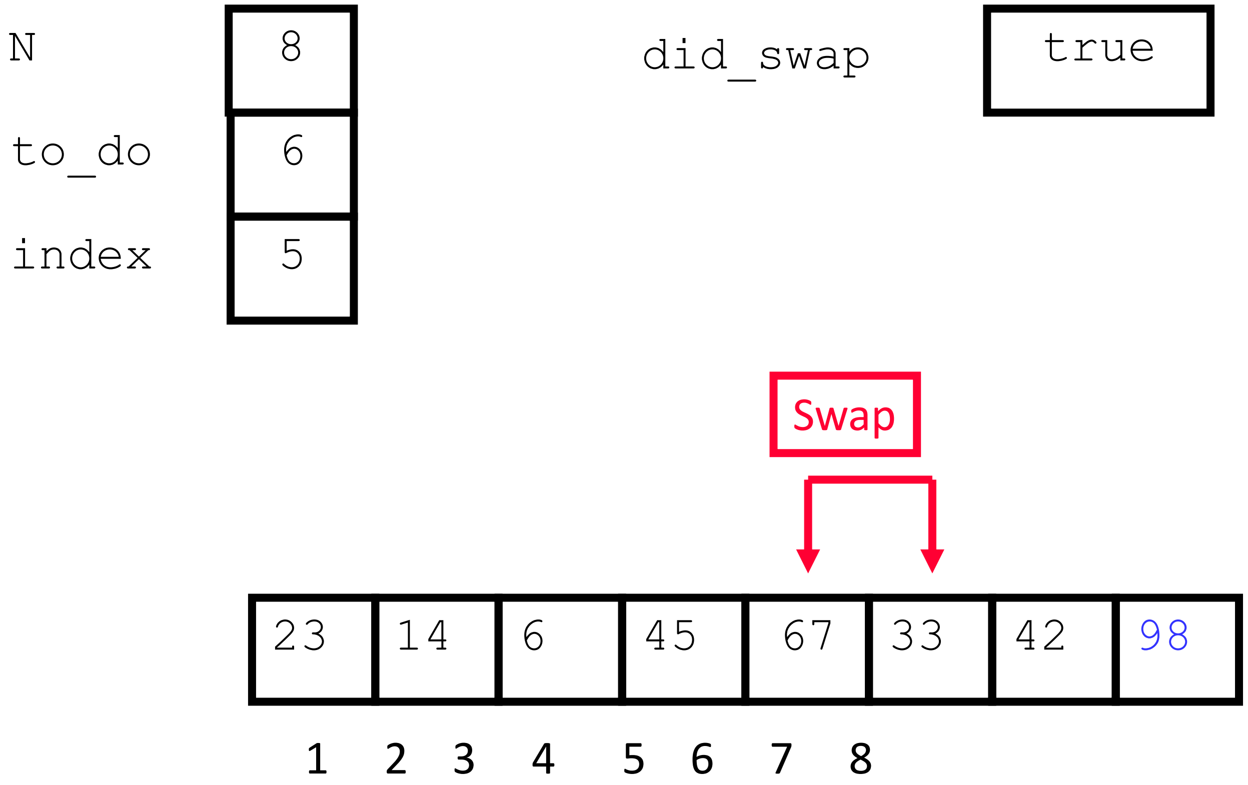
The Second "Bubble Up"



The Second "Bubble Up"



The Second “Bubble Up”



The Second “Bubble Up”

N 8

to_do 6

index 5

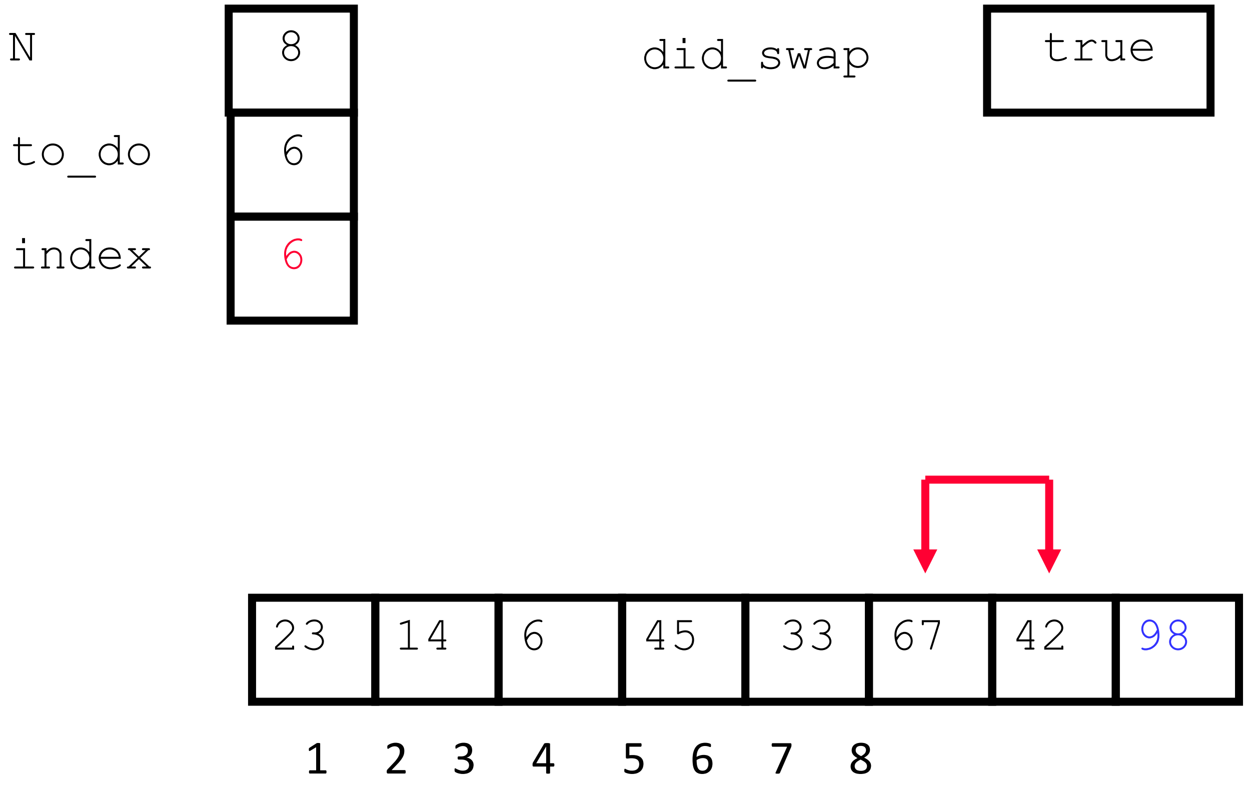
did_swap true

Swap

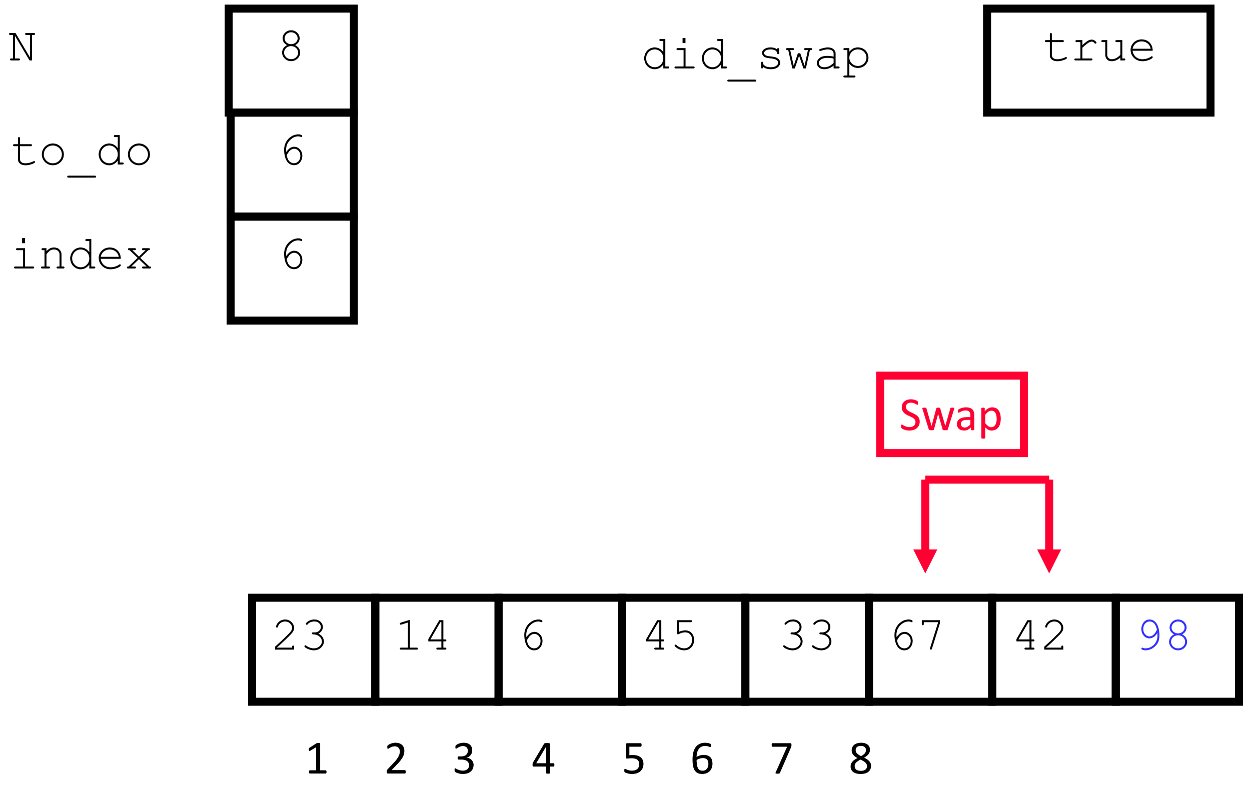
23	14	6	45	33	67	42	98

1 2 3 4 5 6 7 8

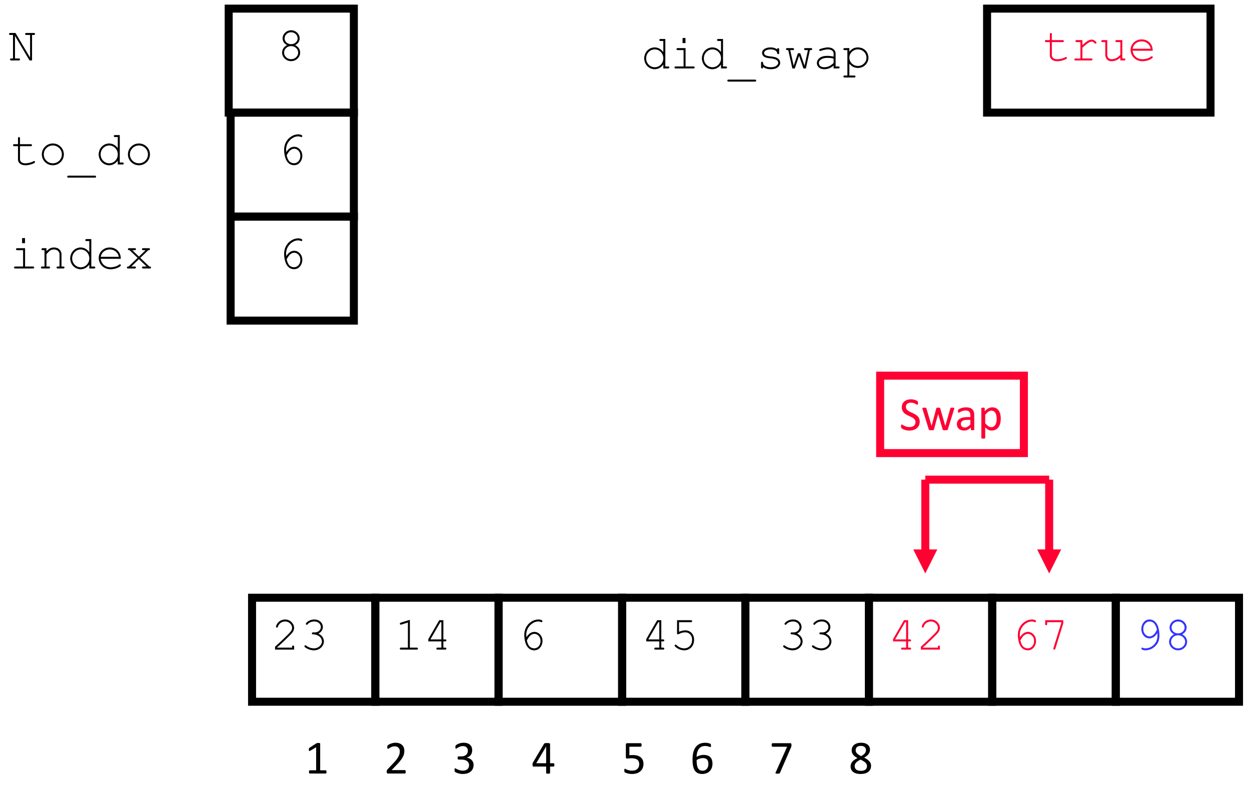
The Second "Bubble Up"



The Second "Bubble Up"



The Second "Bubble Up"



After Second Pass of Outer Loop

`N`	`8`
`to_do`	`6`
`index`	`7`

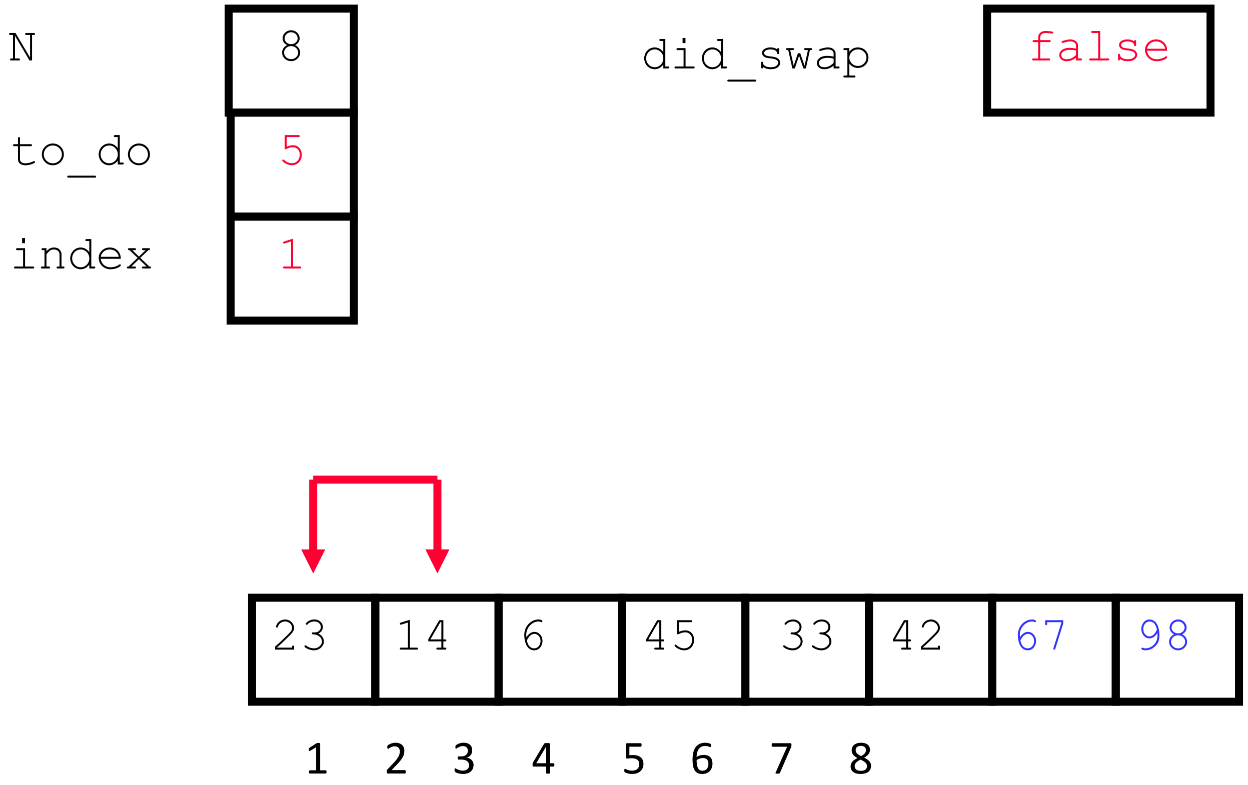
`did_swap`	`true`

Finished second “Bubble Up”

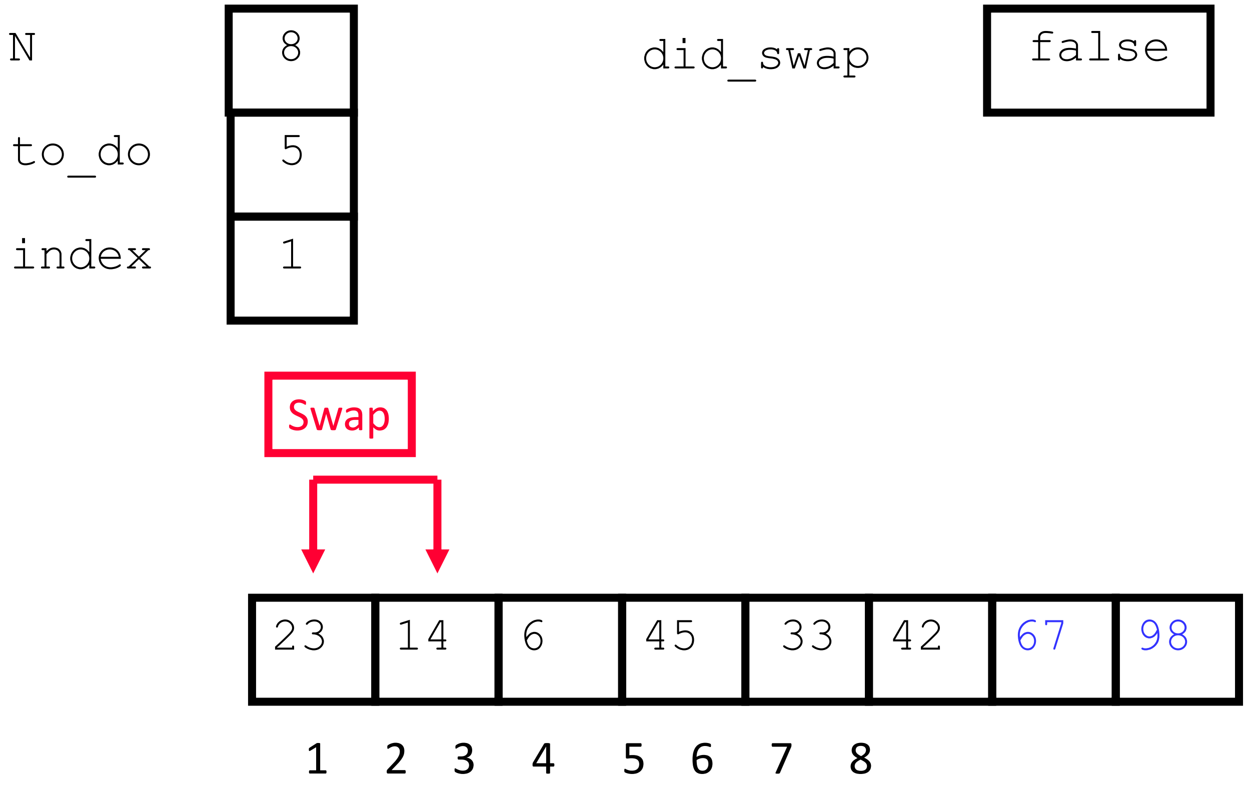
23	14	6	45	33	42	67	98

1 2 3 4 5 6 7 8

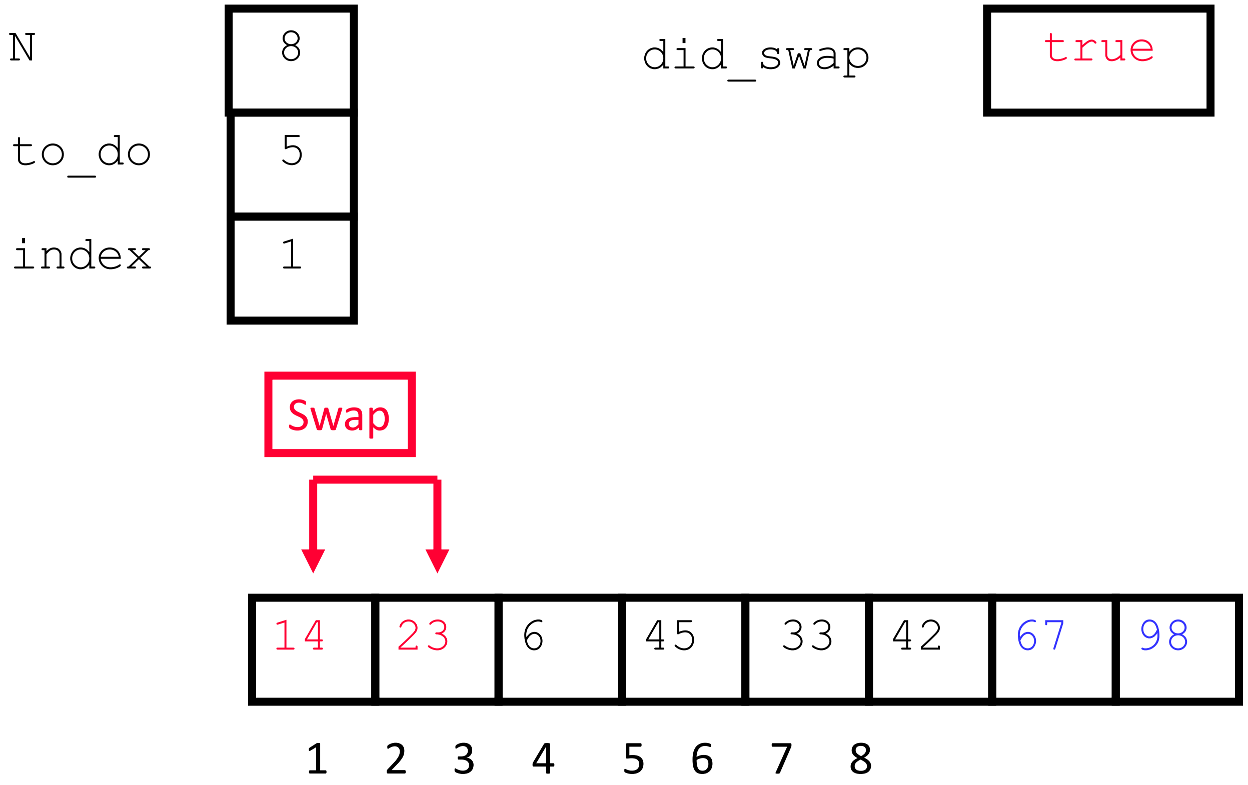
The Third “Bubble Up”



The Third “Bubble Up”



The Third "Bubble Up"



The Third “Bubble Up”

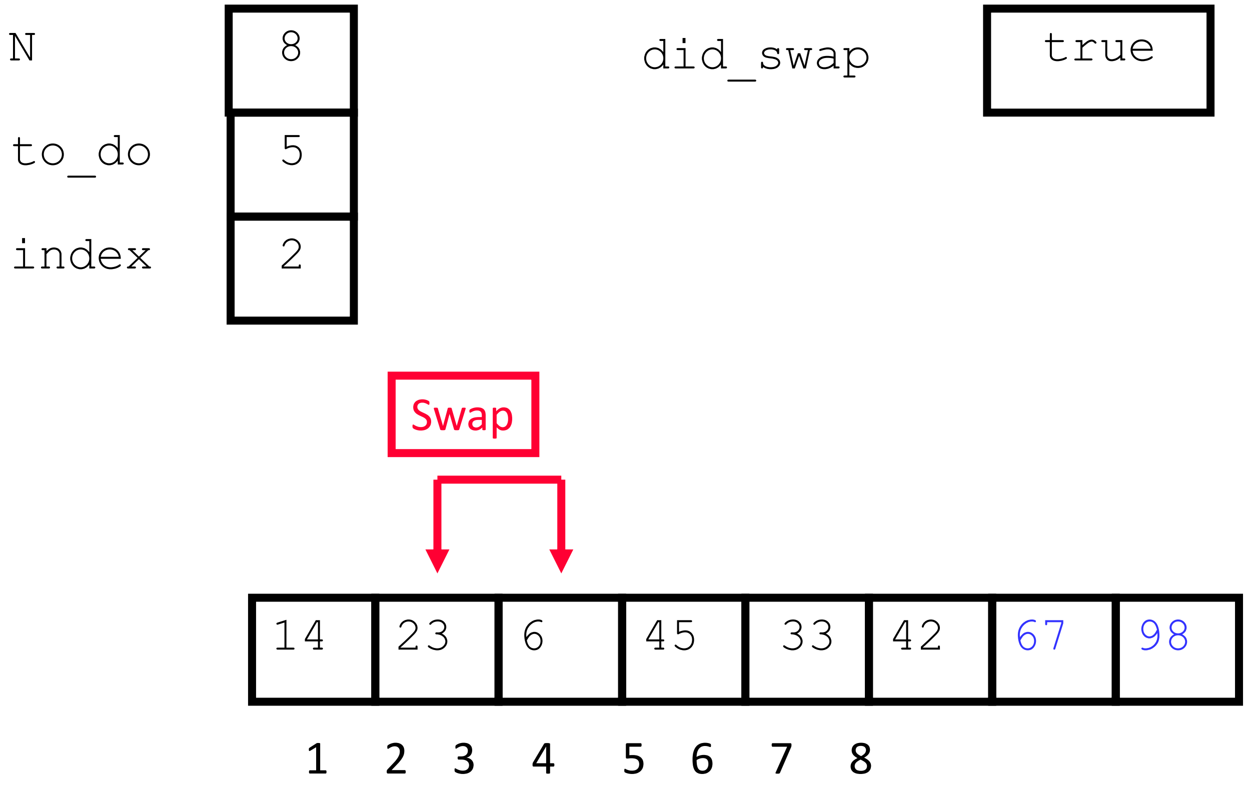
N	8
to_do	5
index	2

did_swap	true

14	23	6	45	33	42	67	98

1 2 3 4 5 6 7 8

The Third “Bubble Up”



The Third “Bubble Up”

N	8
to_do	5
index	2

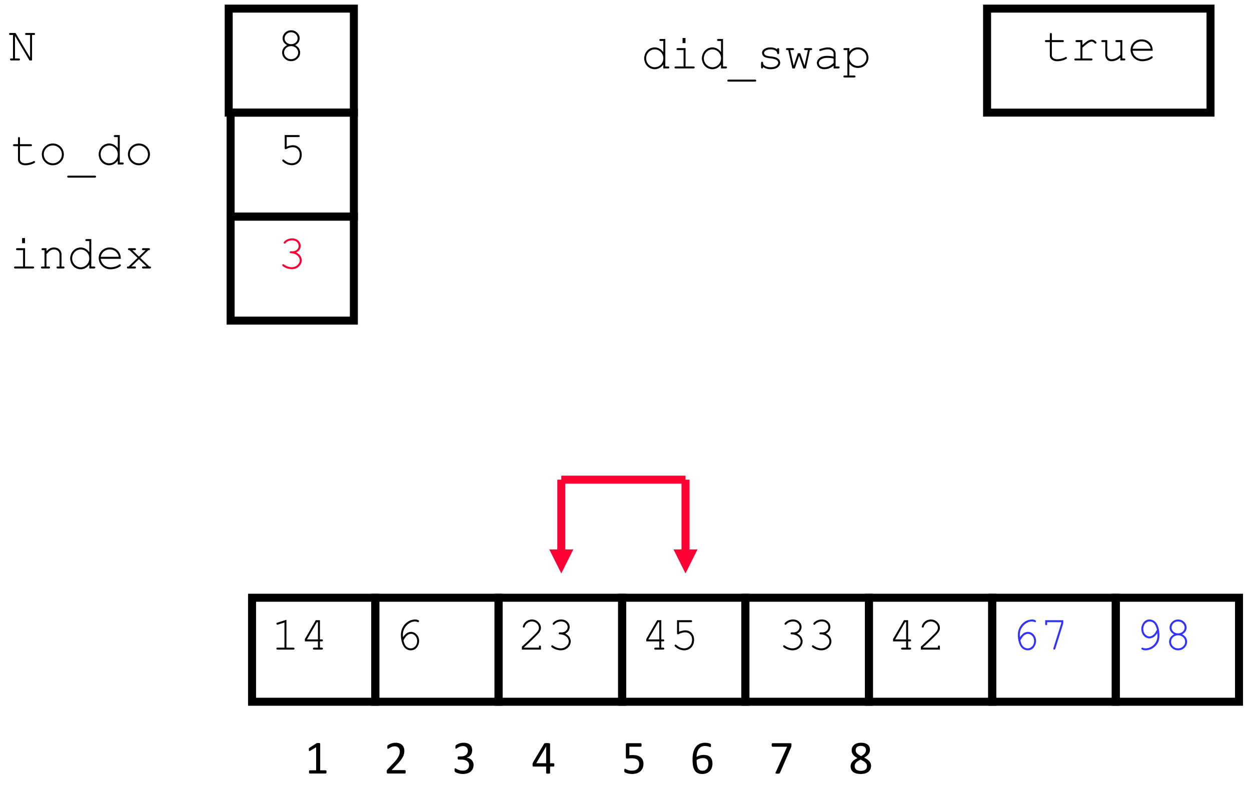
did_swap	true

Swap

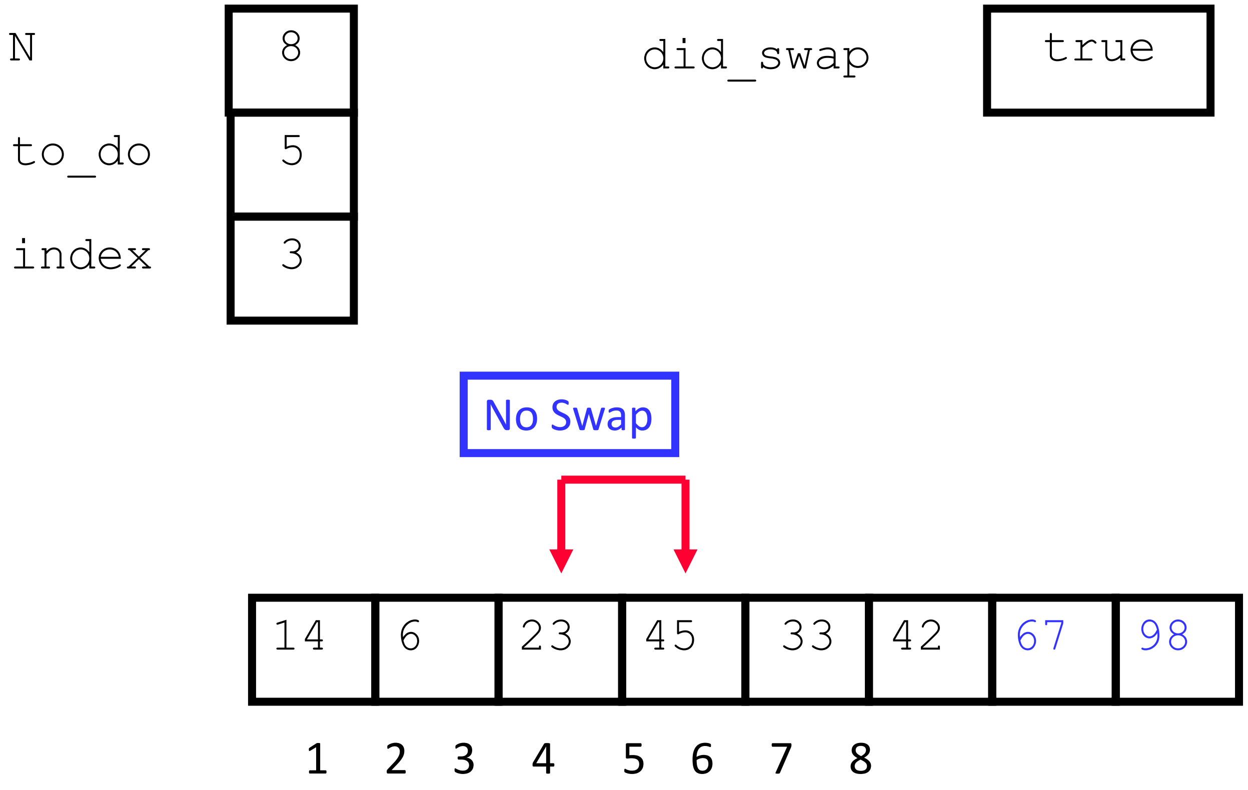
14	6	23	45	33	42	67	98

1 2 3 4 5 6 7 8

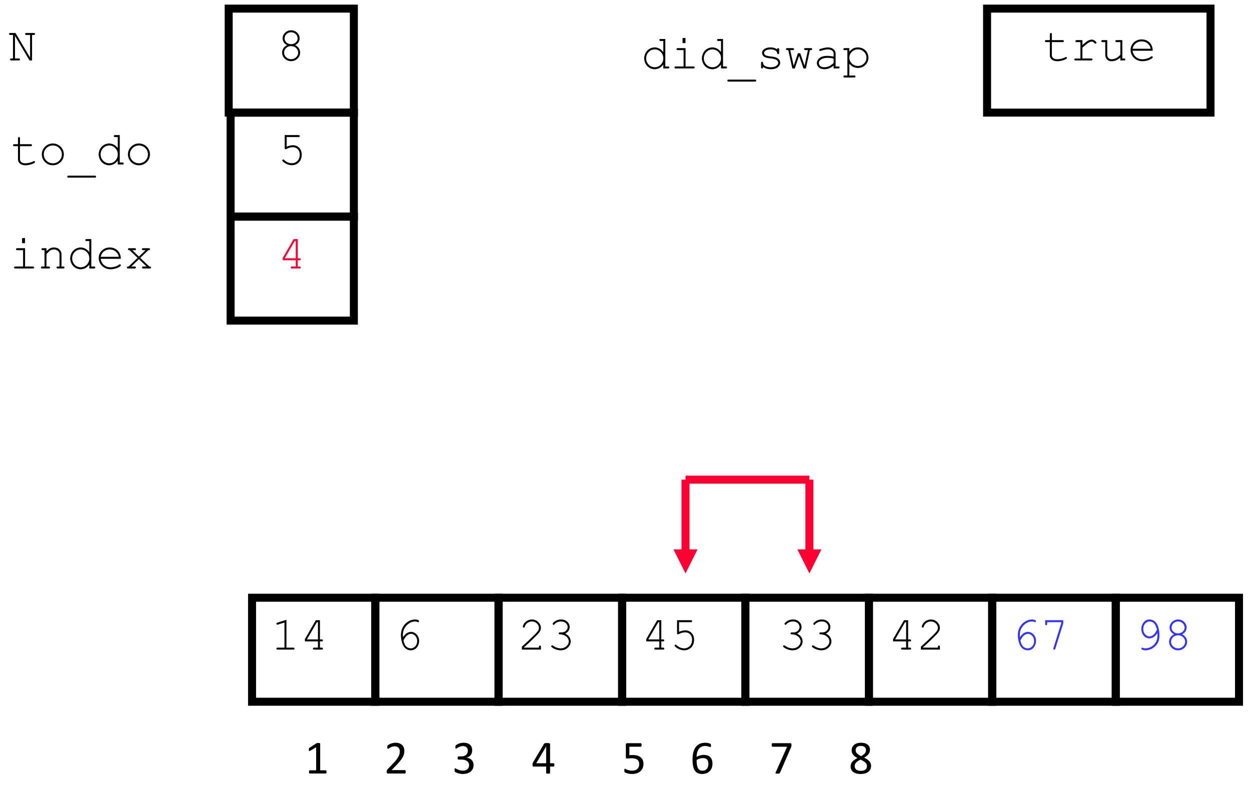
The Third "Bubble Up"



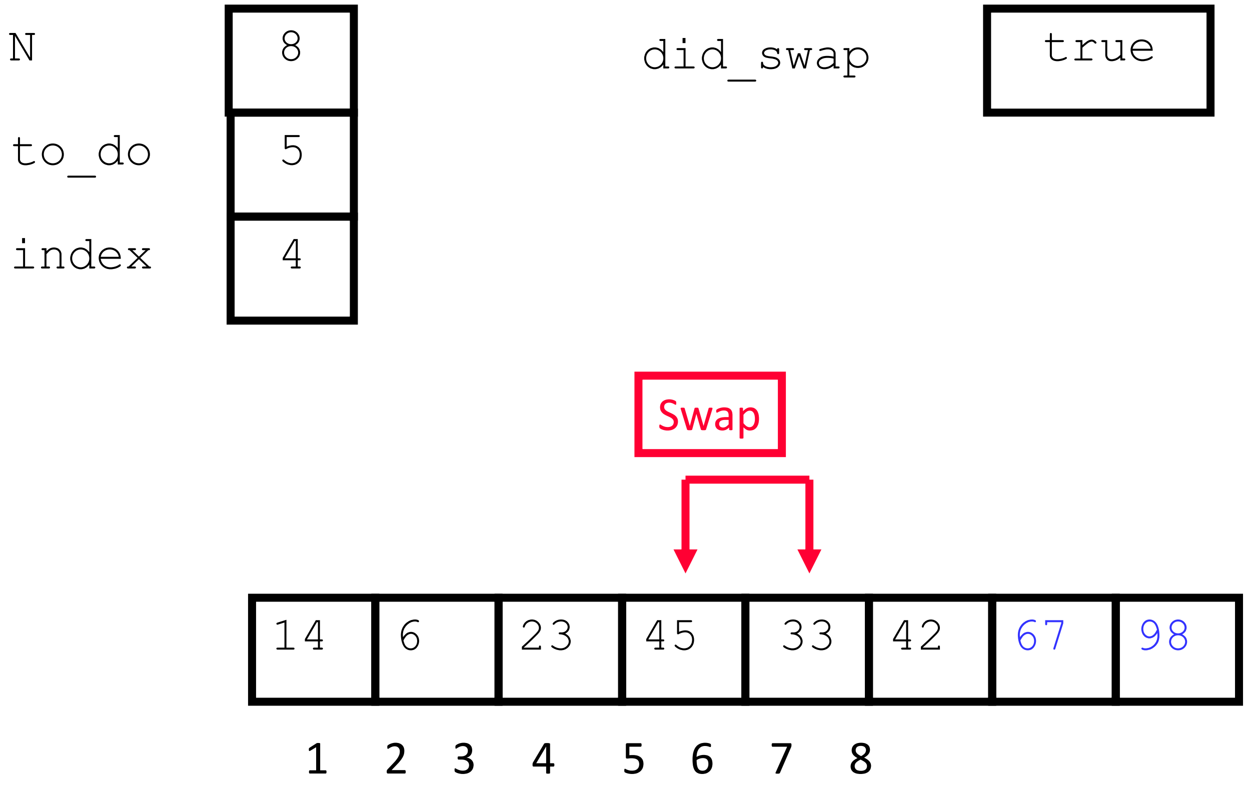
The Third "Bubble Up"



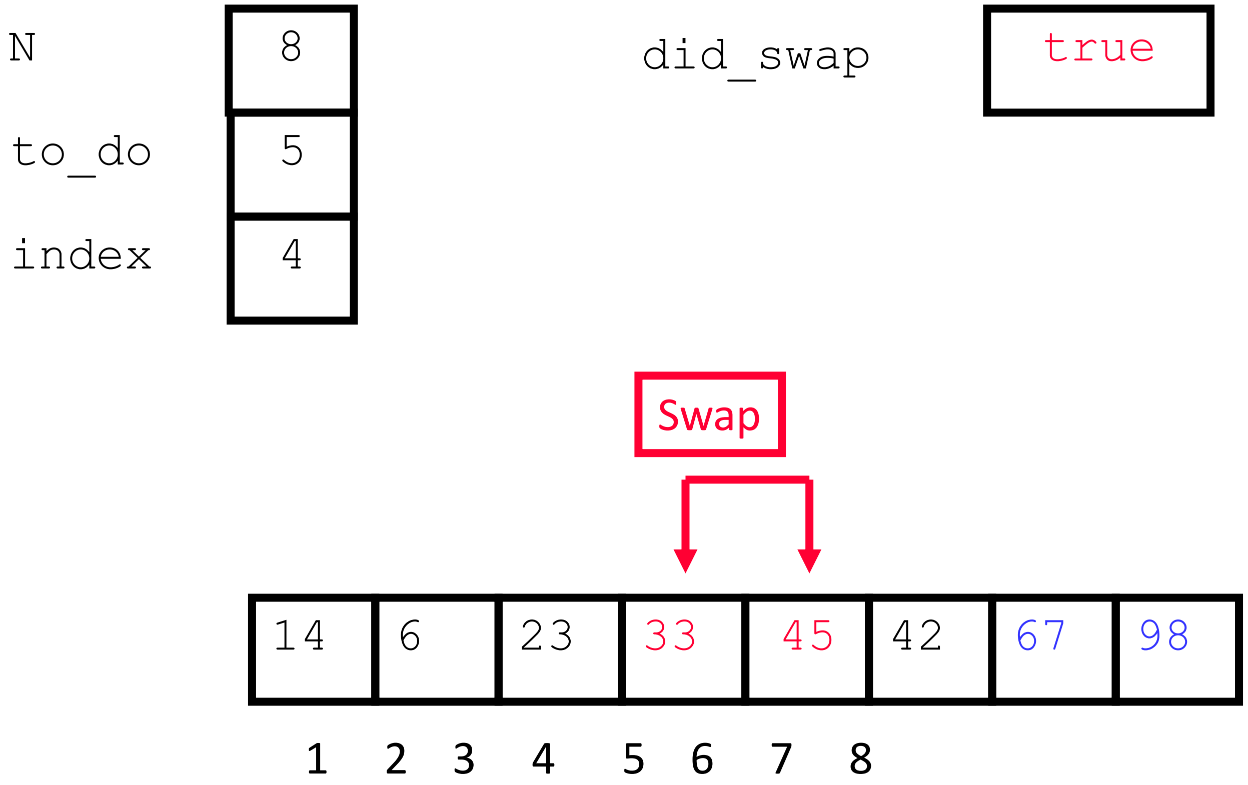
The Third "Bubble Up"



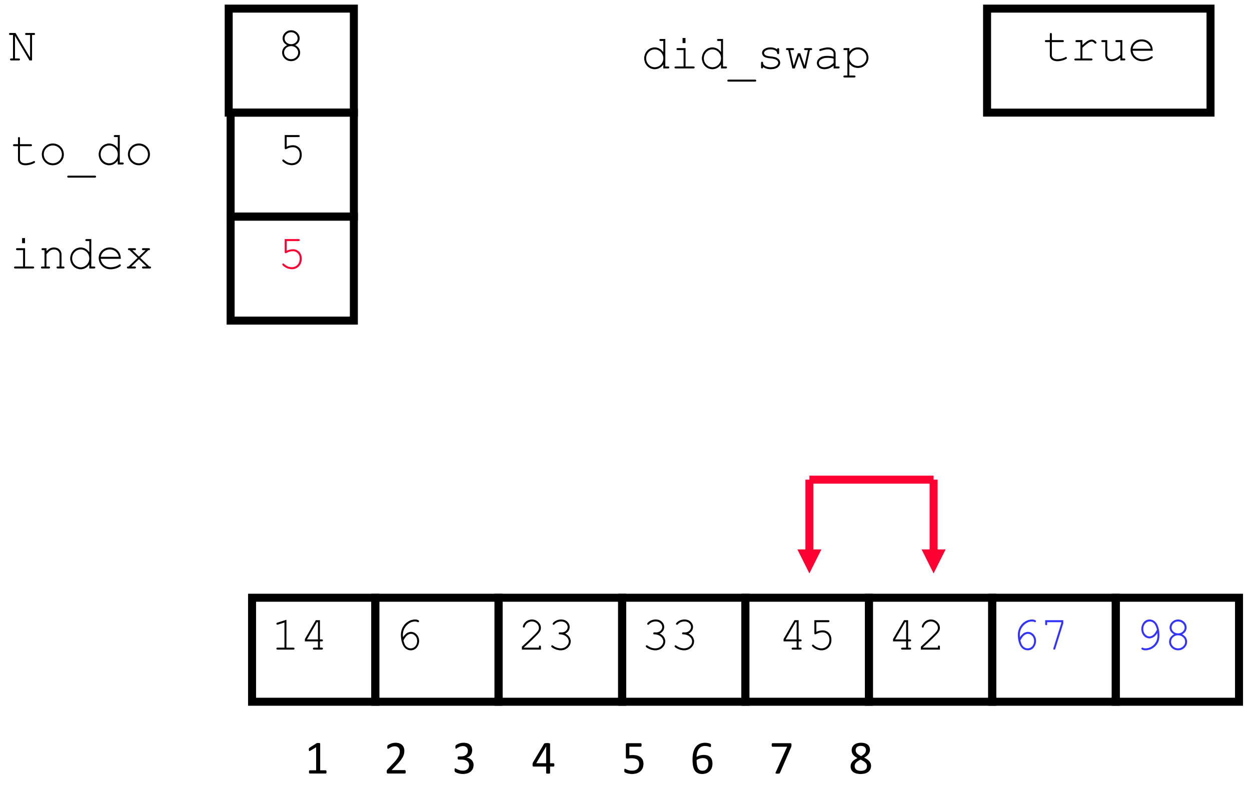
The Third “Bubble Up”



The Third “Bubble Up”



The Third “Bubble Up”



The Third “Bubble Up”

N 8

to_do 5

index 5

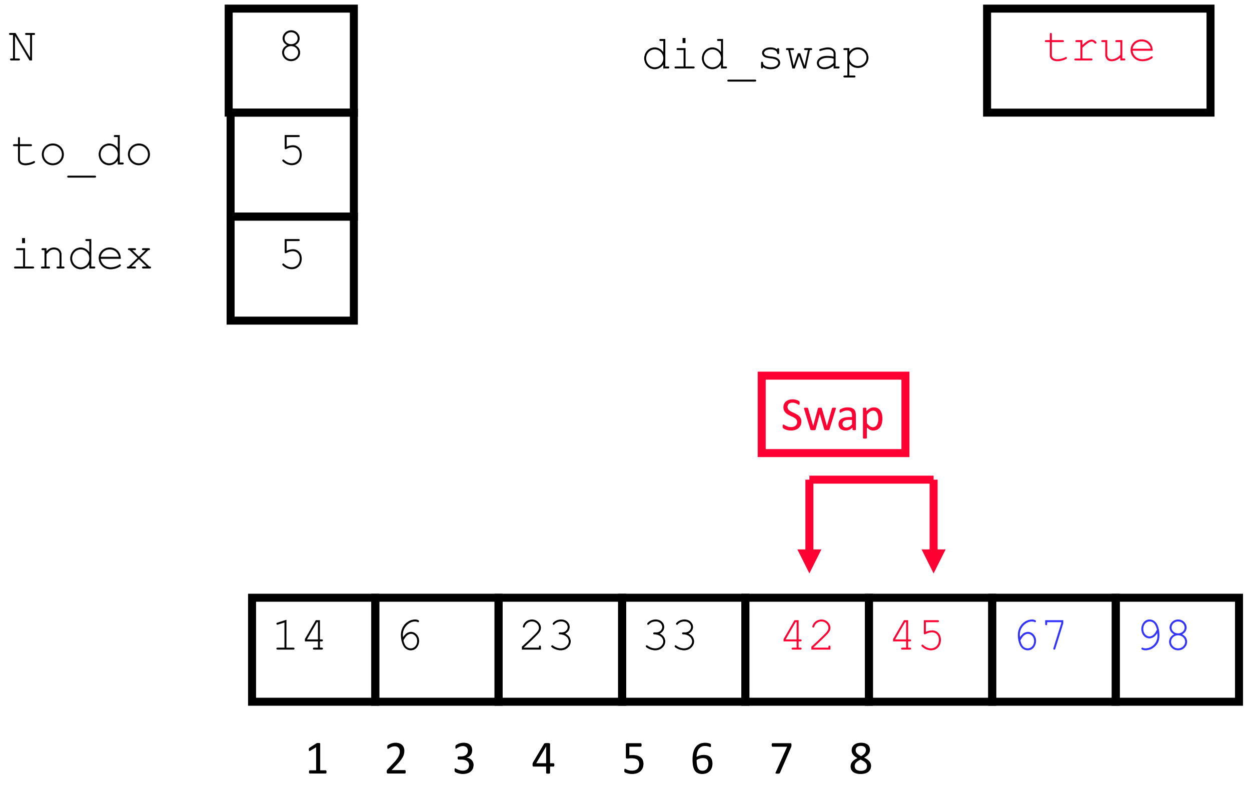
did_swap true

Swap

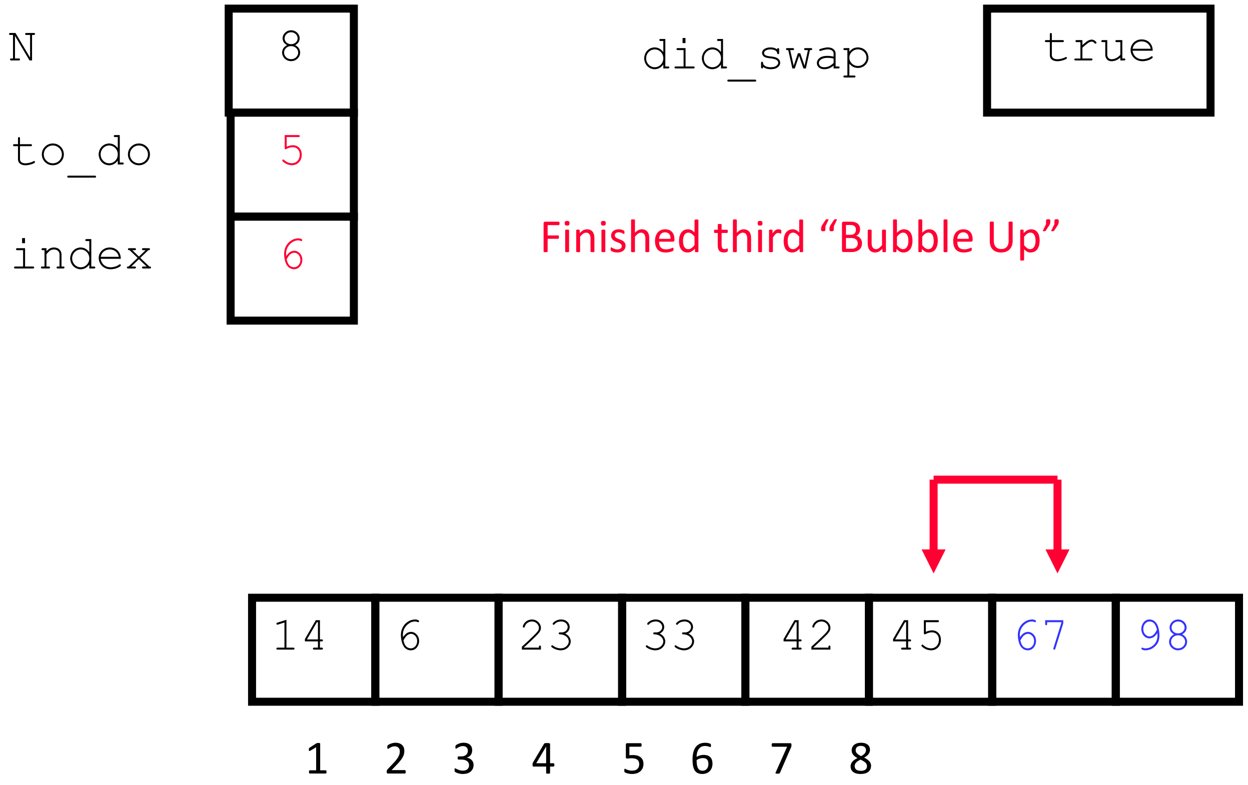
14	6	23	33	45	42	67	98

1 2 3 4 5 6 7 8

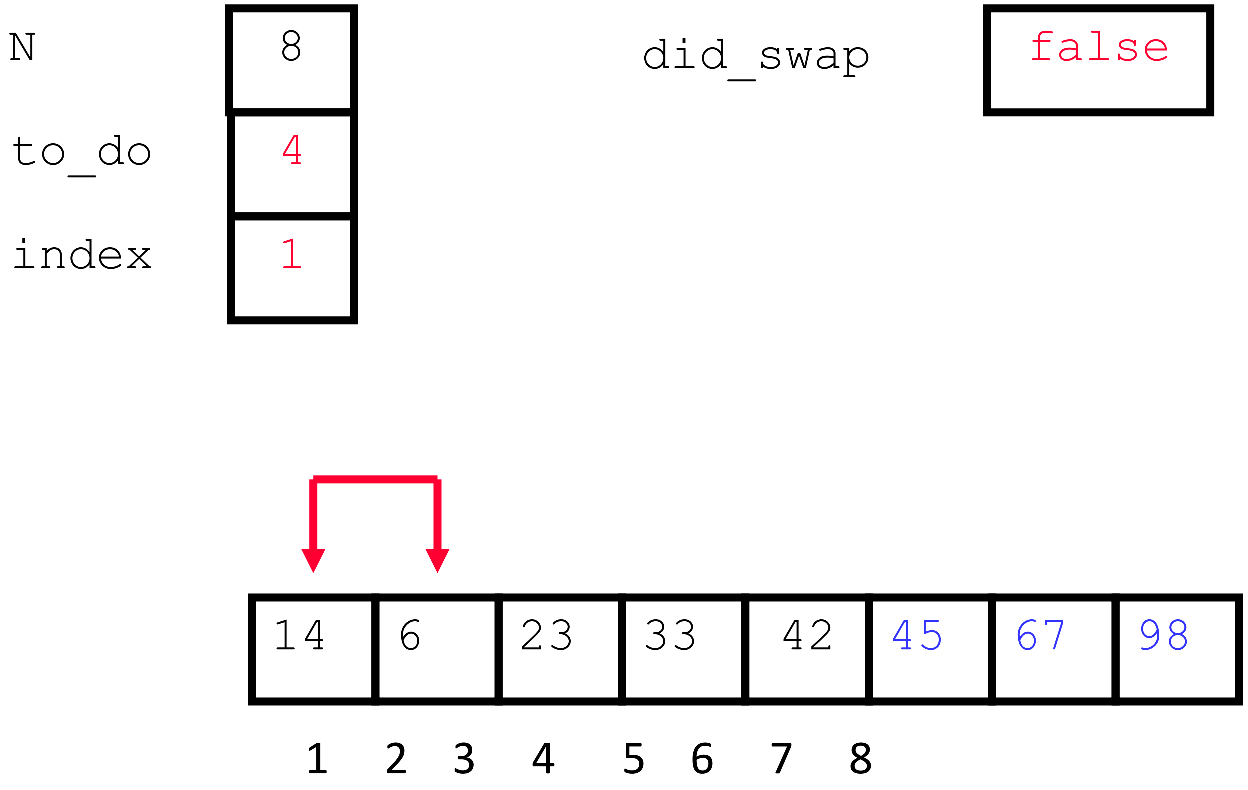
The Third "Bubble Up"



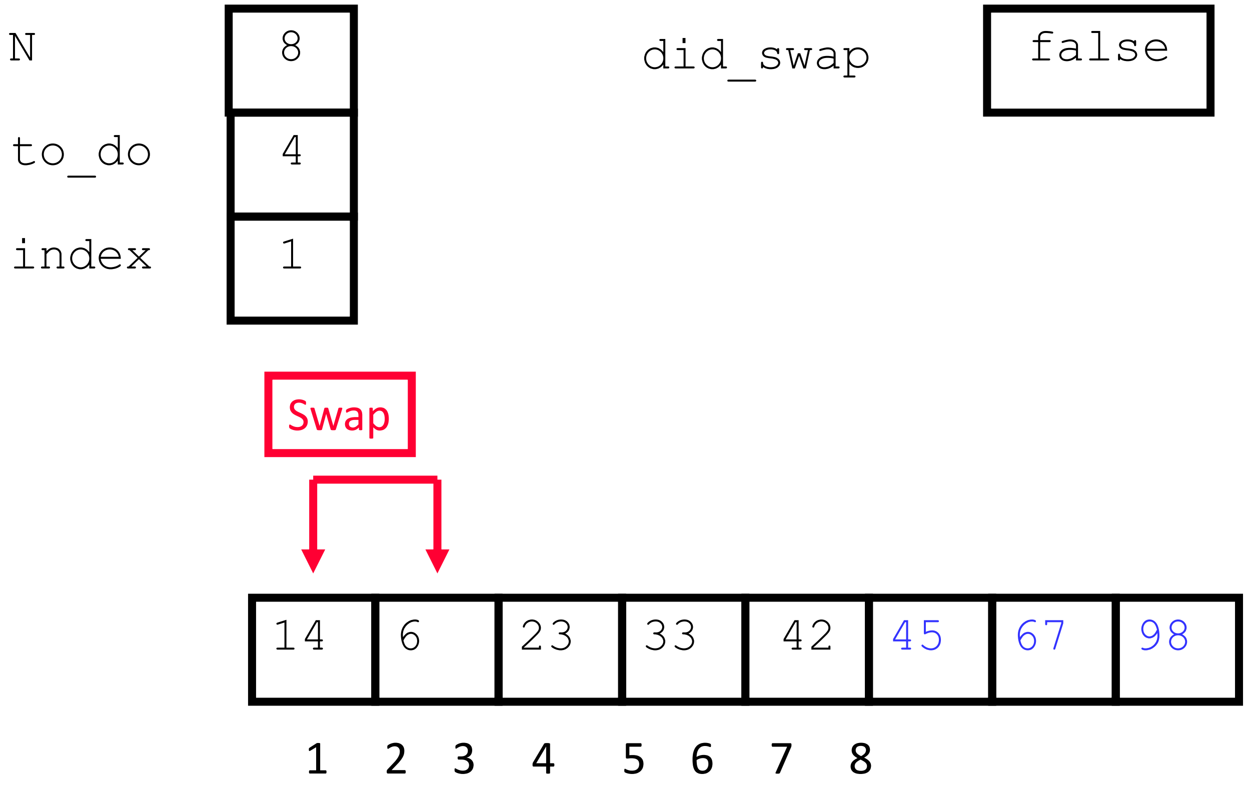
After Third Pass of Outer Loop



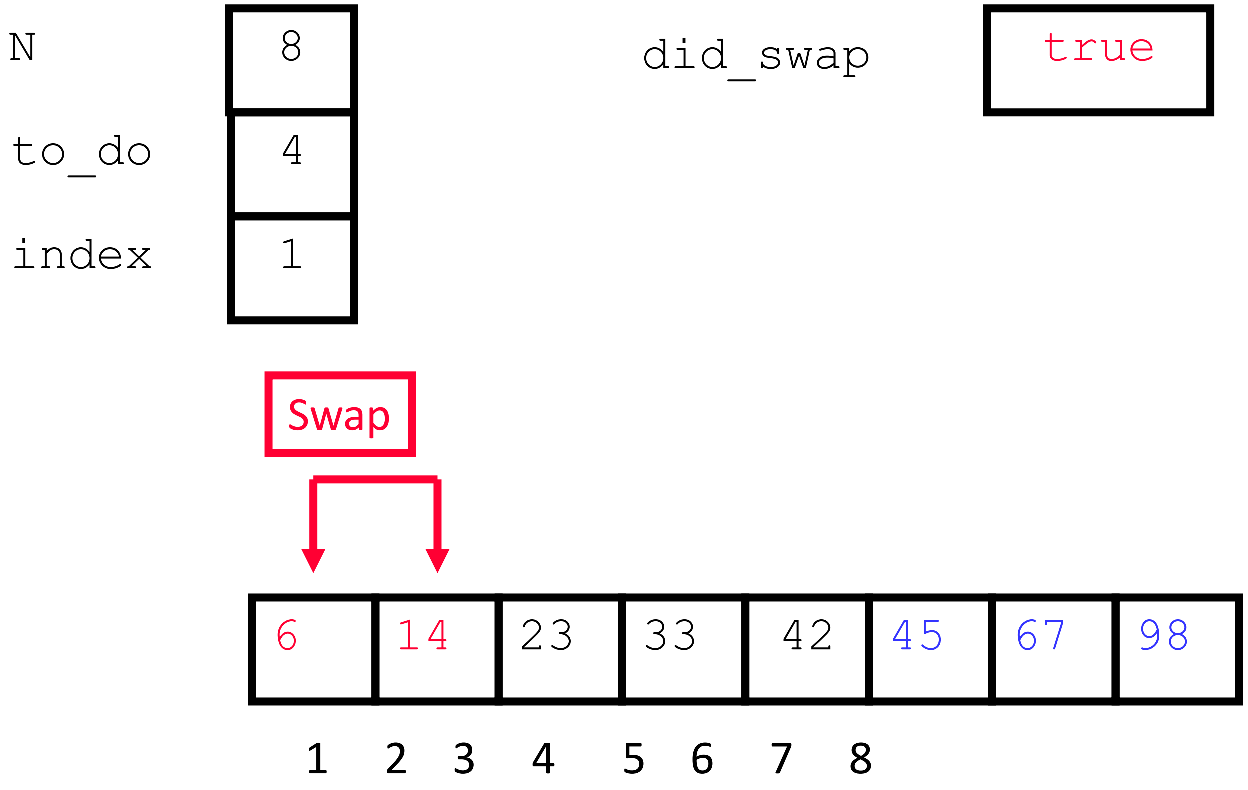
The Fourth "Bubble Up"



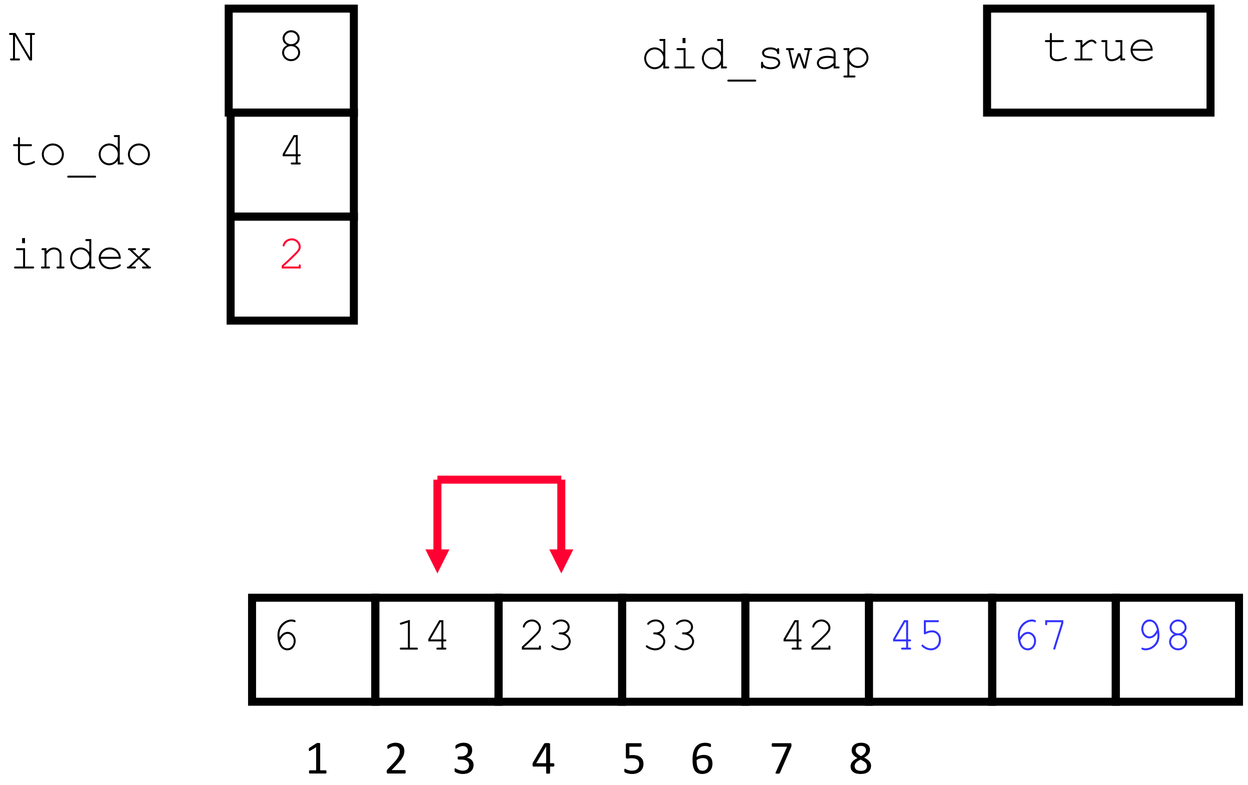
The Fourth “Bubble Up”



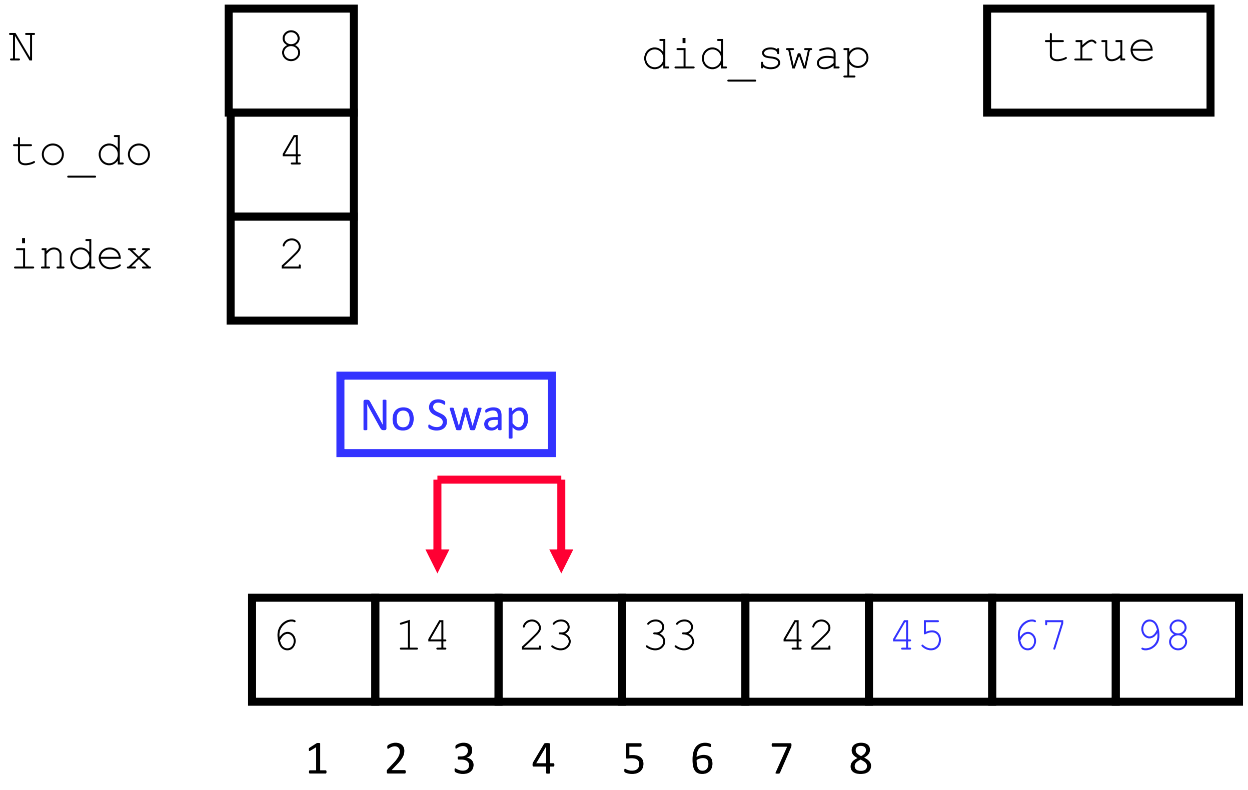
The Fourth "Bubble Up"



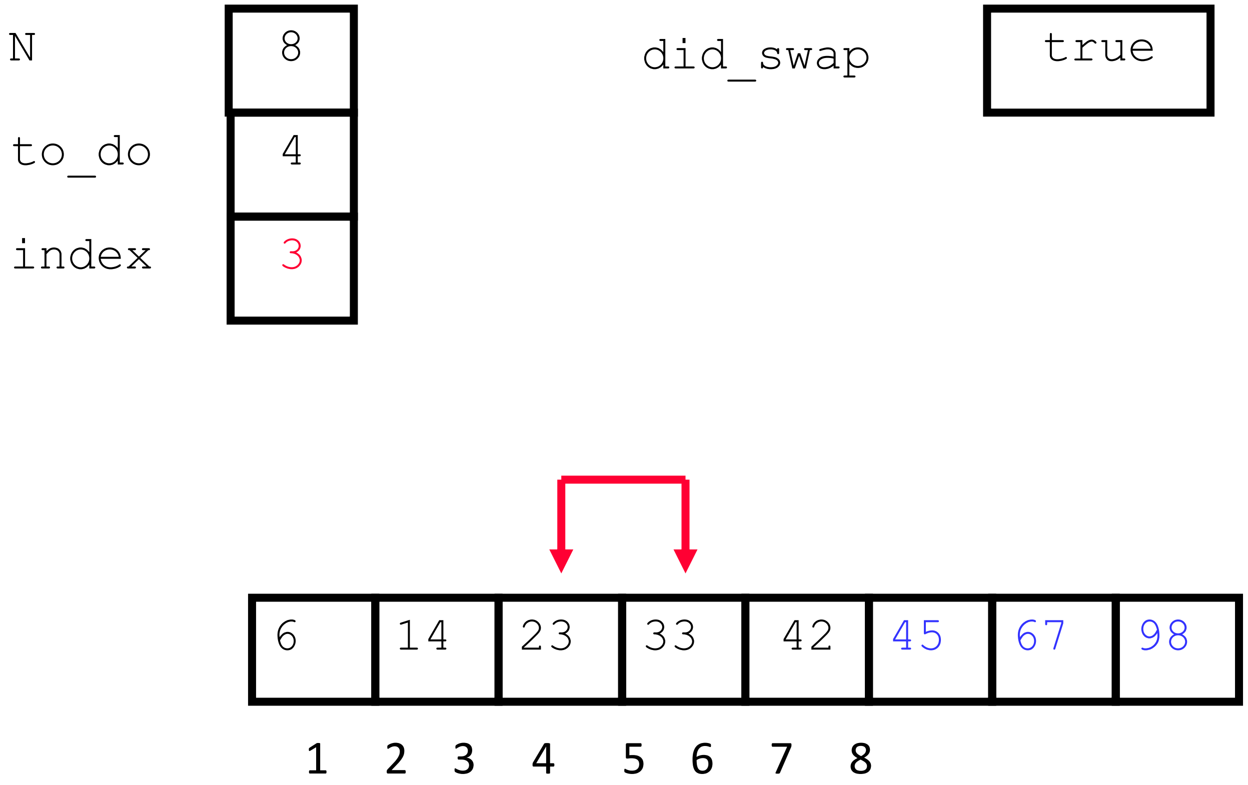
The Fourth "Bubble Up"



The Fourth “Bubble Up”



The Fourth “Bubble Up”



The Fourth "Bubble Up"

N	8
to_do	4
index	3

did_swap	true

No Swap

6	14	23	33	42	45	67	98

1 2 3 4 5 6 7 8

The Fourth “Bubble Up”

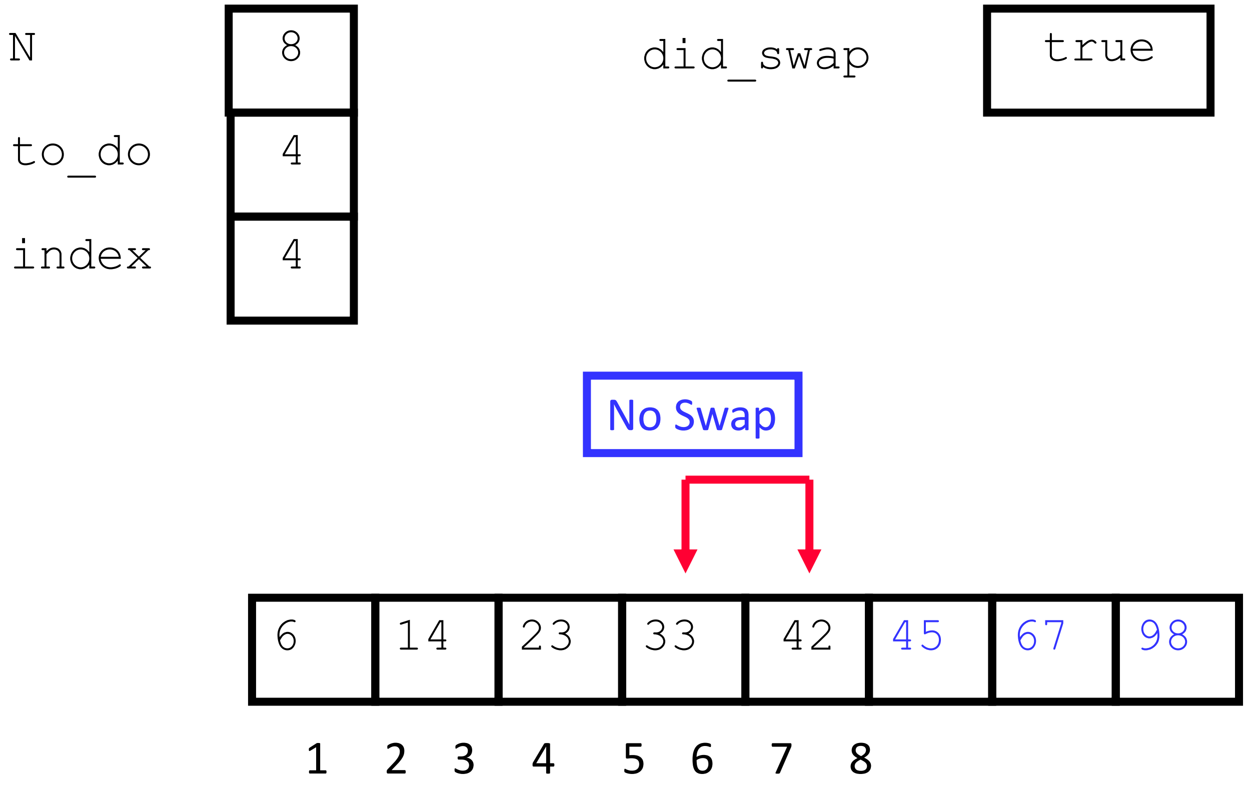
N	8
to_do	4
index	4

did_swap	true

6	14	23	33	42	45	67	98

1 2 3 4 5 6 7 8

The Fourth "Bubble Up"



After Fourth Pass of Outer Loop

N	8
to_do	4
index	5

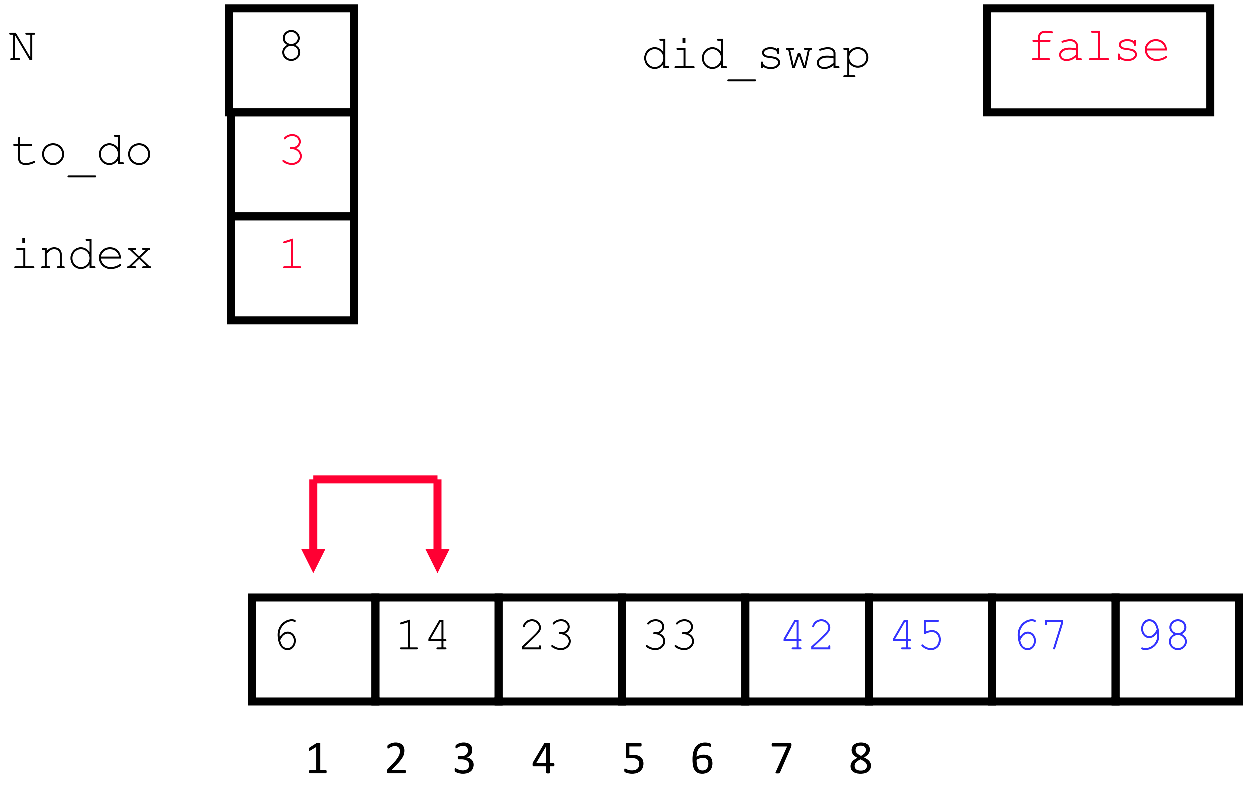
did_swap	true

Finished fourth "Bubble Up"

6	14	23	33	42	45	67	98

1 2 3 4 5 6 7 8

The Fifth “Bubble Up”



The Fifth "Bubble Up"

N	8
to_do	3
index	1

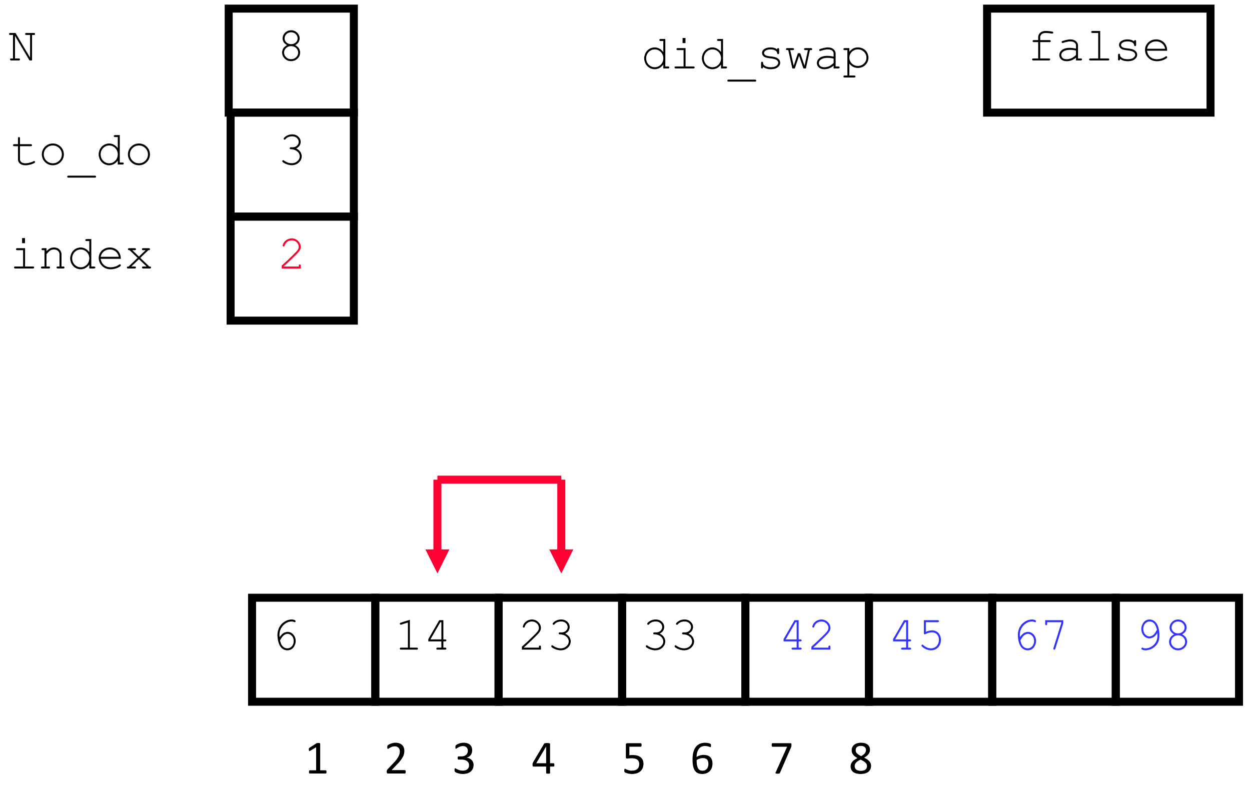
did_swap	false

No Swap

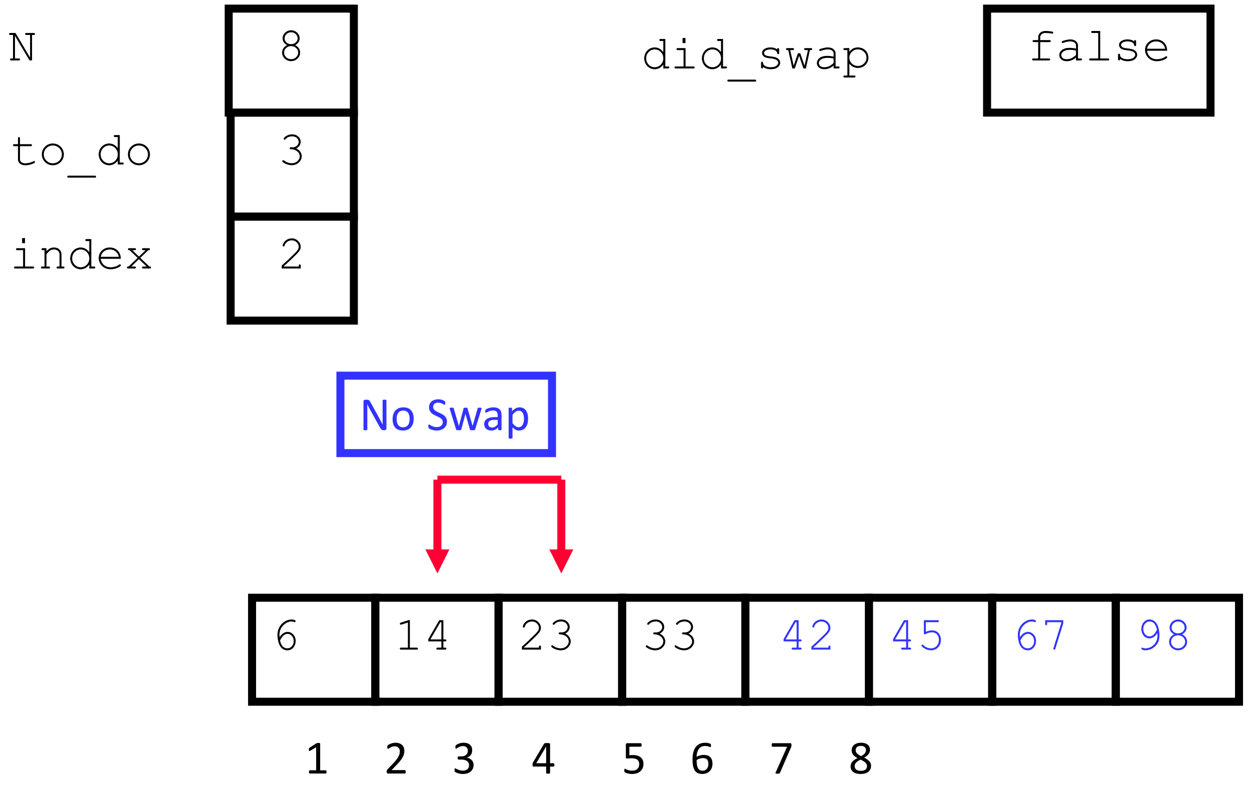
6	14	23	33	42	45	67	98

1 2 3 4 5 6 7 8

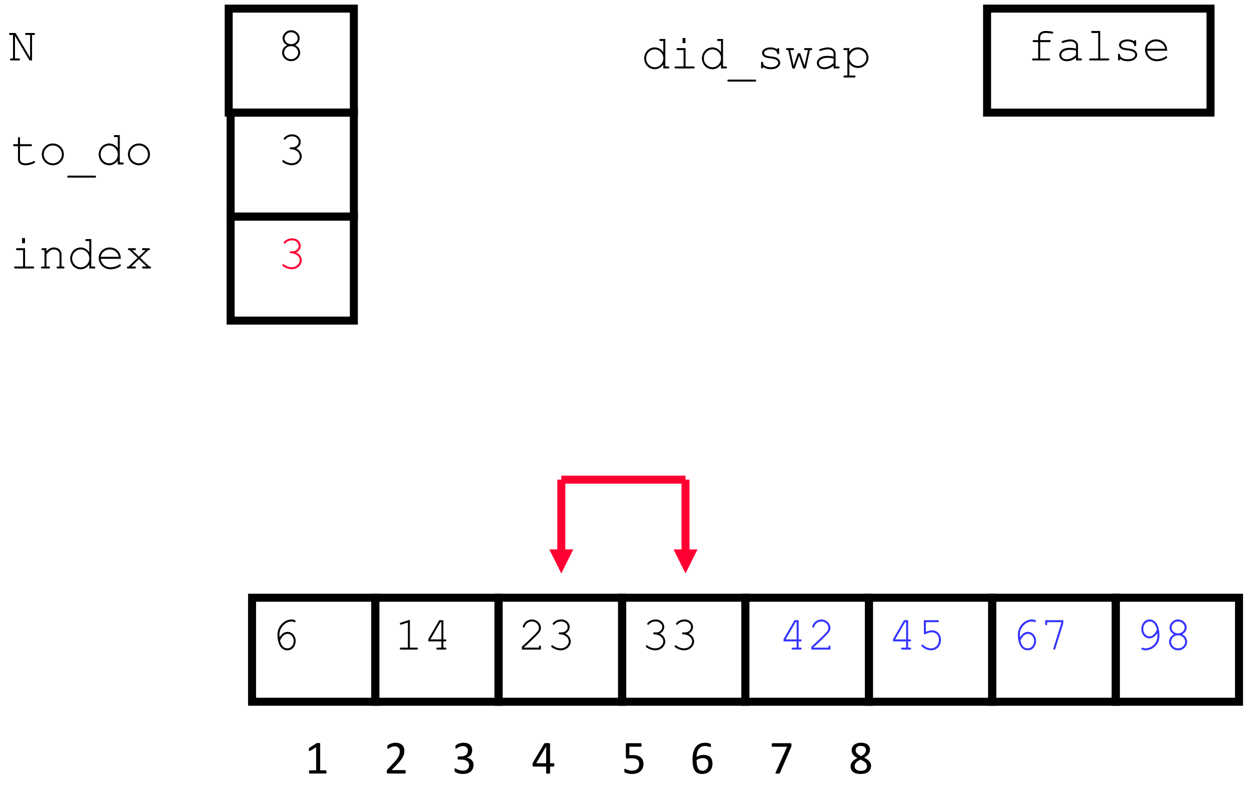
The Fifth "Bubble Up"



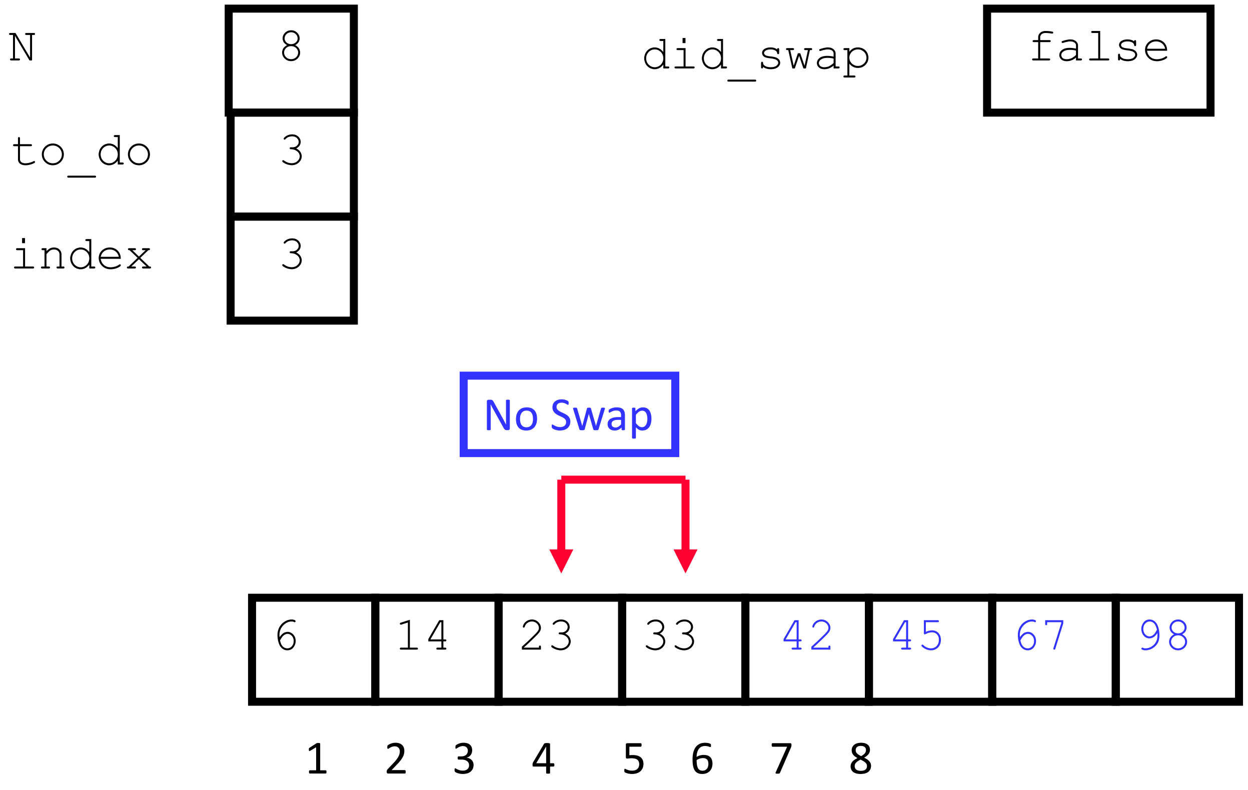
The Fifth “Bubble Up”



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"

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

Pseudocode

improvised

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)
    # optimize code, so if the array is already sorted, it doesn't
need
    # to go through the entire process
    swapped = False
    # Traverse through all array elements
    for i in range(n-1):
        # range(n) also work but outer loop will
        # repeat one time more than needed.
        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            # if we haven't needed to make a single swap, we
            # can just exit the main loop.
            return
# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)
print("Sorted array is:")
for i in range(len(arr)):
    print("% d" % arr[i], end=" ")
```

Credit : geeksforgeeks.org

Bubble Sort: Complexity analysis

Case 1: If the input list is already in sorted order

Number of comparisons:

$$C(n) = \frac{n(n-1)}{2}$$

Number of movements:

$$M(n) = 0$$

Bubble Sort: Complexity analysis

Case 2: If the input list is sorted but in reverse order

Number of comparisons:

$$c(n) = \frac{n(n-1)}{2}$$

Number of movements:

$$M(n) = \frac{n(n-1)}{2}$$

Bubble Sort: Complexity analysis

Case 3: If the input list is in random order

Number of comparisons:

$$C(n) = \frac{n(n-1)}{2}$$

Number of movements:

- Let p_j be the probability that the largest element is in the unsorted part is in j^{th} $(1 \le j \le n - i + 1)$ location.
- The average number of swaps in the i^{th} pass is

$$= \sum_{j=1}^{n-i+1} (\overline{n-i+1} - j) \cdot p_j$$

Bubble Sort: Complexity analysis

Case 3: If the input list is in random order

Number of movements:

- $p_1 = p_2 = p_{n-i+1} = \frac{1}{n-i+1}$
- Therefore, the average number of swaps in the i^{th} pass is

$$= \sum_{j=1}^{n-i+1} \frac{1}{n-i+1} \cdot (\overline{n-i+1} - j) = \frac{n-i}{2}$$

- The average number of movements

$$M(n) = \sum_{i=1}^{n-1} \frac{n-i}{2} = \frac{n(n-1)}{4}$$

Bubble Sort: Summary of Complexity analysis

Case	Run time, $T(n)$	Complexity	Remarks
Case 1	$T(n) = c\frac{n(n-1)}{2}$	$T(n) = O(n^2)$	Best case
Case 2	$T(n) = cn(n-1)$	$T(n) = O(n^2)$	Worst case
Case 3	$T(n) = \frac{3}{4}n(n-1)$	$T(n) = O(n^2)$	Average case

The best explanation: The recurrence relation of the code of recursive bubble sort is $T(n) = T(n-1) + n$.

Obama on bubble sort

When asked the most efficient way to sort a million 32-bit integers, Senator Obama had an answer:

http://www.youtube.com/watch?v=k4RRi_ntQc8

Bubble Sort

How do you make best case with (n-1) comparisons only?

- By maintaining a variable flag, to check if there has been any swaps in a given pass.
- If not, the array is already sorted.

Bubble Sort

```
void bubble_sort(int x[], int n)
{
  int i,j;
  int flag = 0;
   for (i=n-1; i>0; i--)
   {
     for (j=0; j<i; j++)
     if (x[j] > x[j+1])
     {
       swap(&x[j],&x[j+1]);
       flag = 1;
     }
     if (flag == 0) return;
   }
}
```

Efficient Sorting algorithms

Two of the most popular sorting algorithms are based on divide-and-conquer approach.

- Quick sort
- Merge sort

Basic concept of divide-and-conquer method:

```
sort (list)
{
    if the list has length greater than 1
    {
        Partition the list into lowlist and highlist;
        sort (lowlist);
        sort (highlist);
        combine (lowlist, highlist);
    }
}
```

Is this Stable ?

YES

If you try to solve problems yourself, then you will learn many things automatically.

Spend few minutes and then enjoy the study.