COMP2054-ADE

ADE Lec07

Simple Sorting Algorithms

Lecturer: Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

Simple sorting algorithms and their complexity

Consider an array of integers, and the goal is to sort into non-decreasing order (put the largest on the right)

1. Bubble sort
2. Selection sort
3. Insertion sort

Bubble Sort: Basic Idea

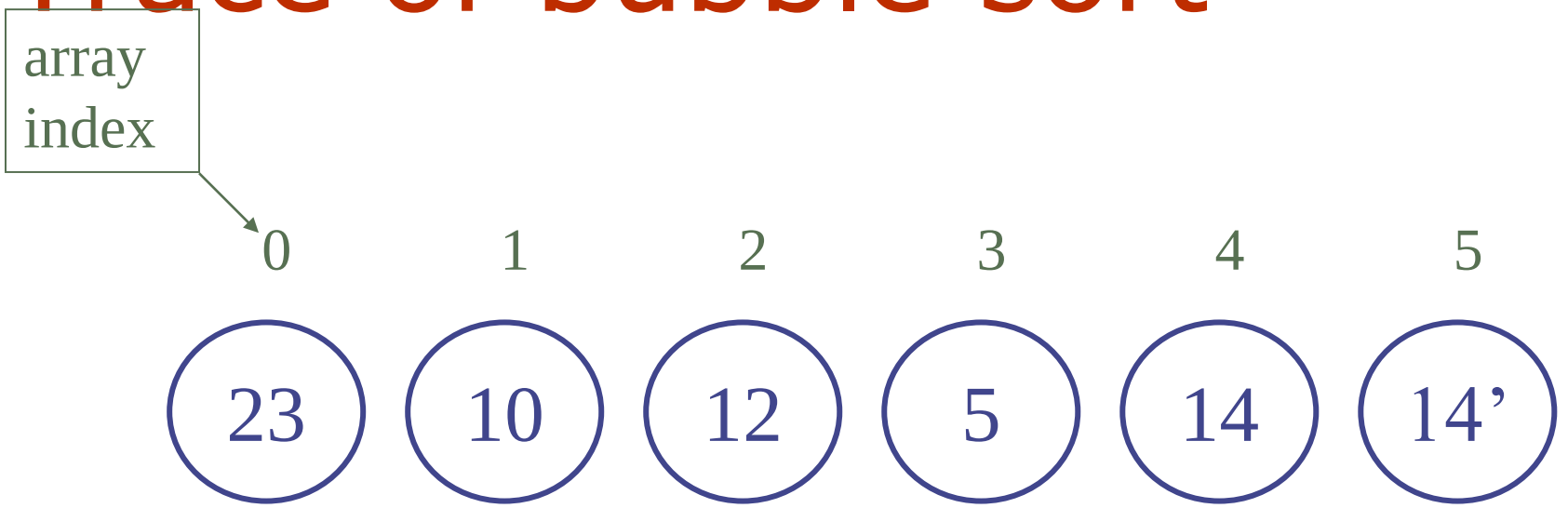
- Outer loop:
  Repeated scans through array
- Inner loop: on each scan do comparison with immediate neighbour
  - think of air bubbles rising in water
  - do swaps to make sure that the largest number “bubbles up” to the end of the array

Bubble sort

```
void  bubbleSort(int arr[]){
    int i;
    int j;
    int temp;
    for(i = arr.length-1; i > 0; i--){
       for(j = 0; j < i; j++){
          if(arr[j] > arr[j+1]){
             temp = arr[j];
             arr[j] = arr[j+1];
             arr[j+1] = temp;
          }//
       }// end inner loop
    }//end outer loop}// end bubble sort
```

swap adjacent elements, if in the wrong order

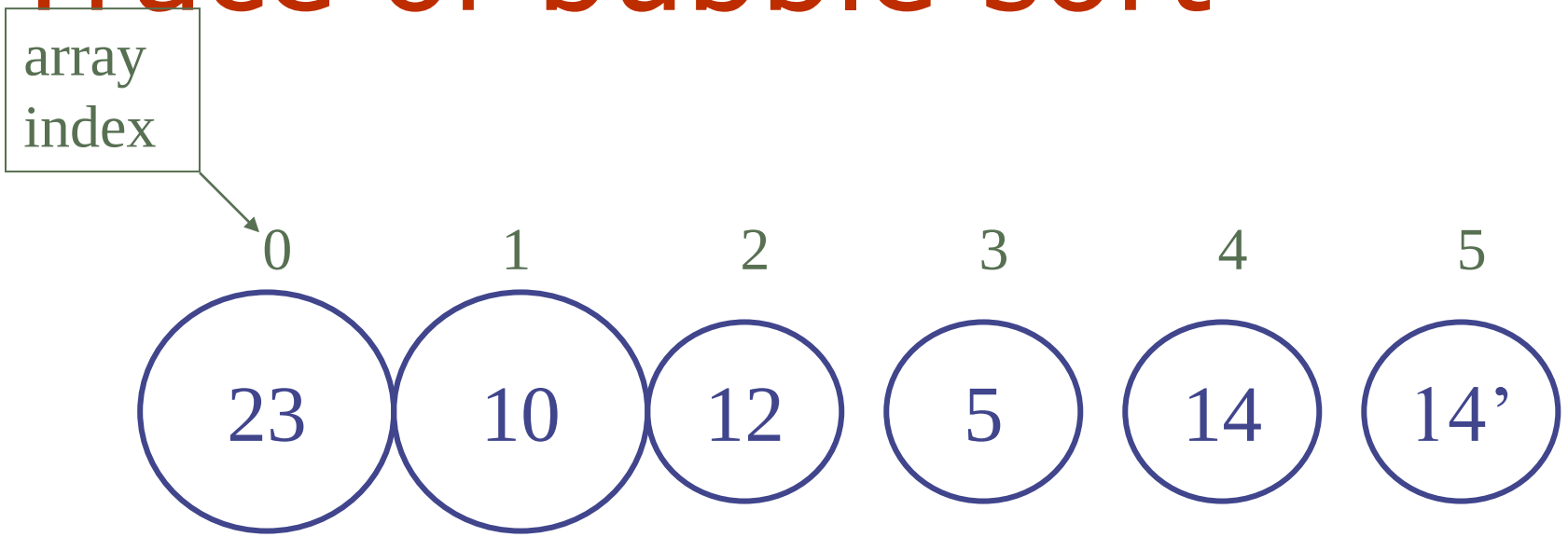
Trace of bubble sort



i = 5, first iteration of the outer loop

“Stability”: 14 and 14’ are two copies of the same number but we keep track of which copy ends up where. (Explained why later)

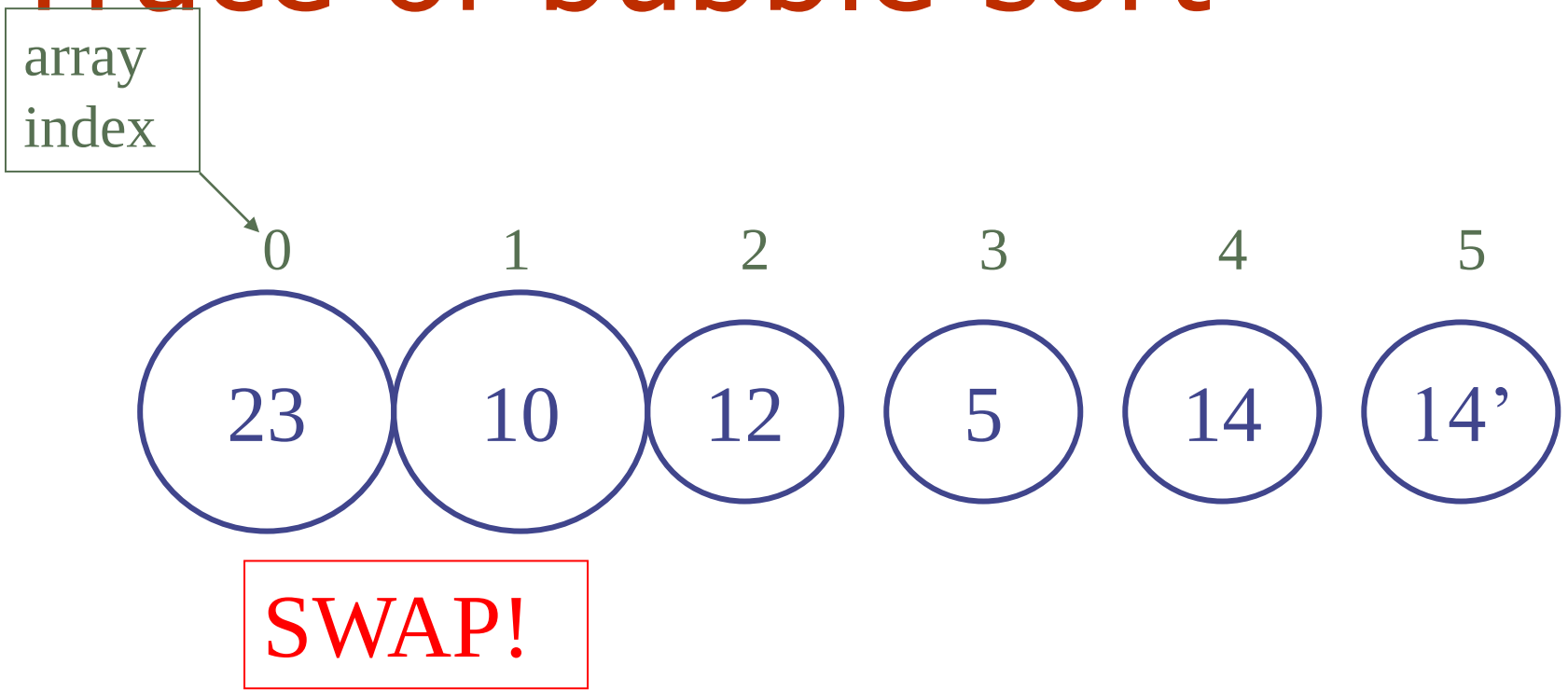
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 0, comparing arr[0] and arr[1]

Trace of bubble sort

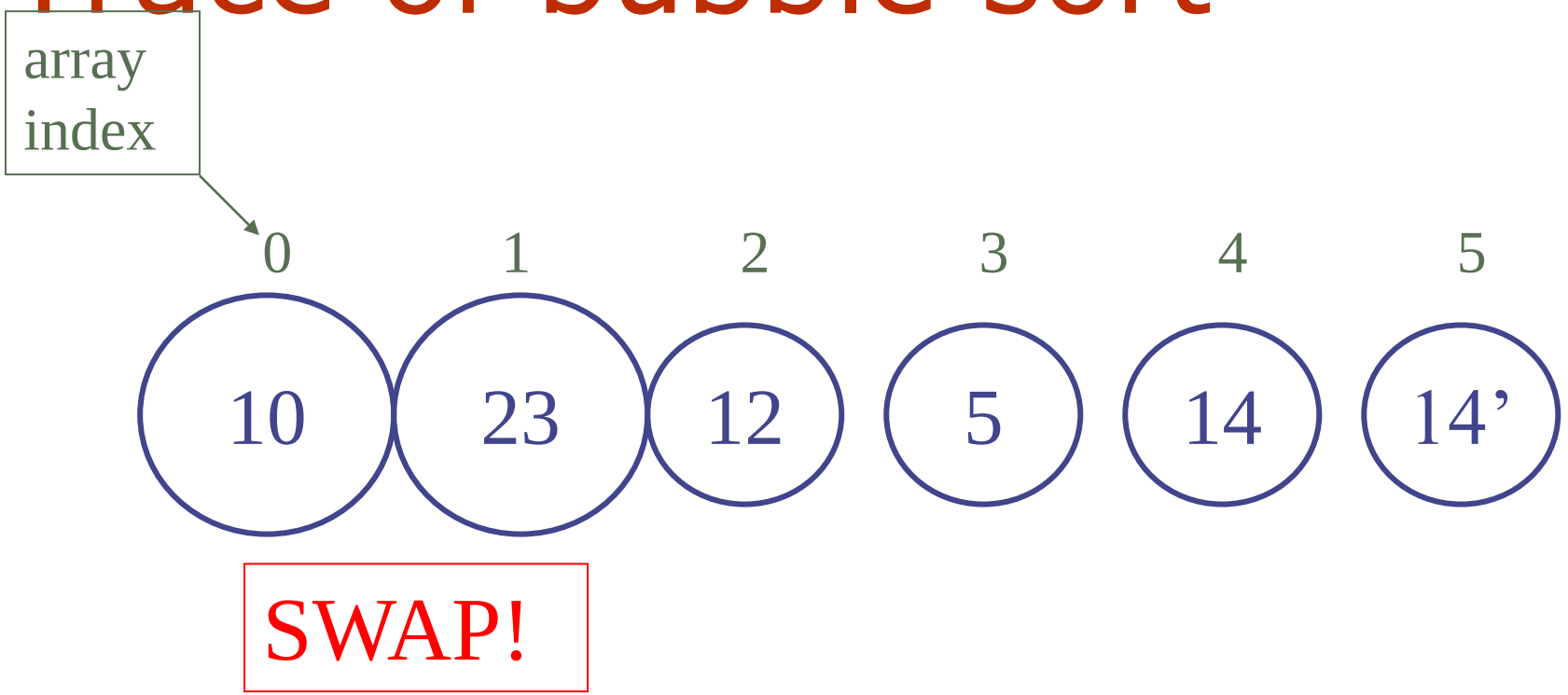


SWAP!

i = 5, first iteration of the outer loop

j = 0, comparing arr[0] and arr[1]

Trace of bubble sort

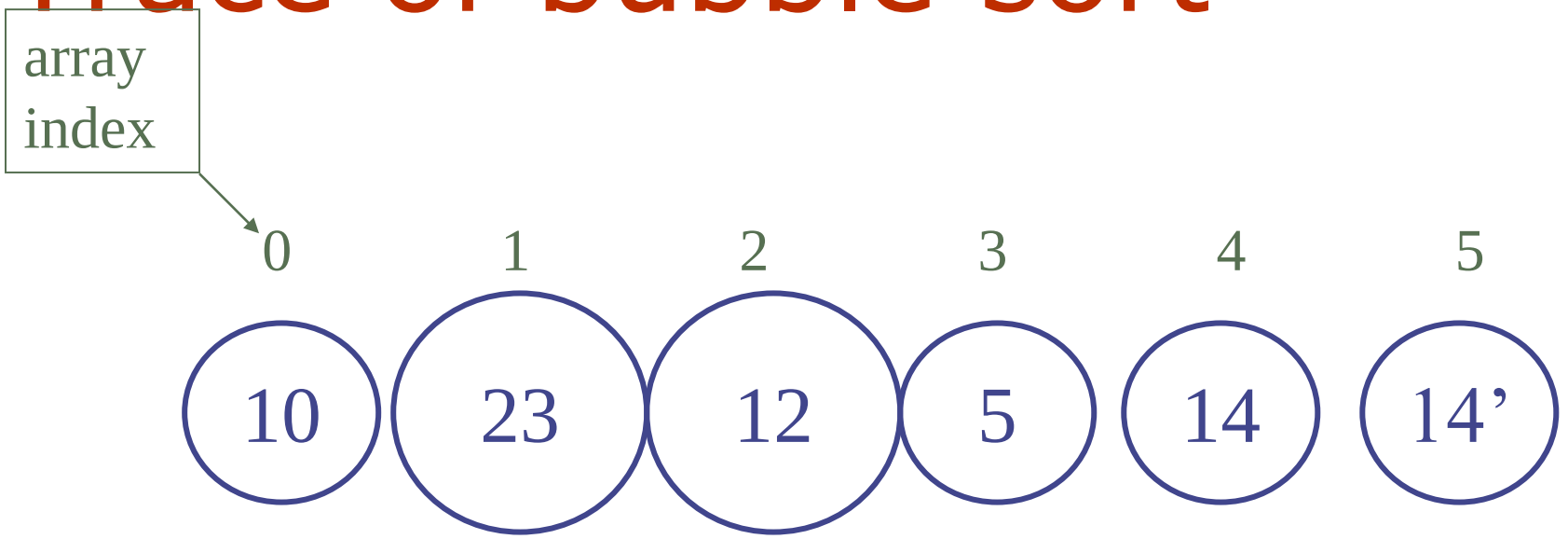


SWAP!

i = 5, first iteration of the outer loop

j = 0, comparing arr[0] and arr[1]

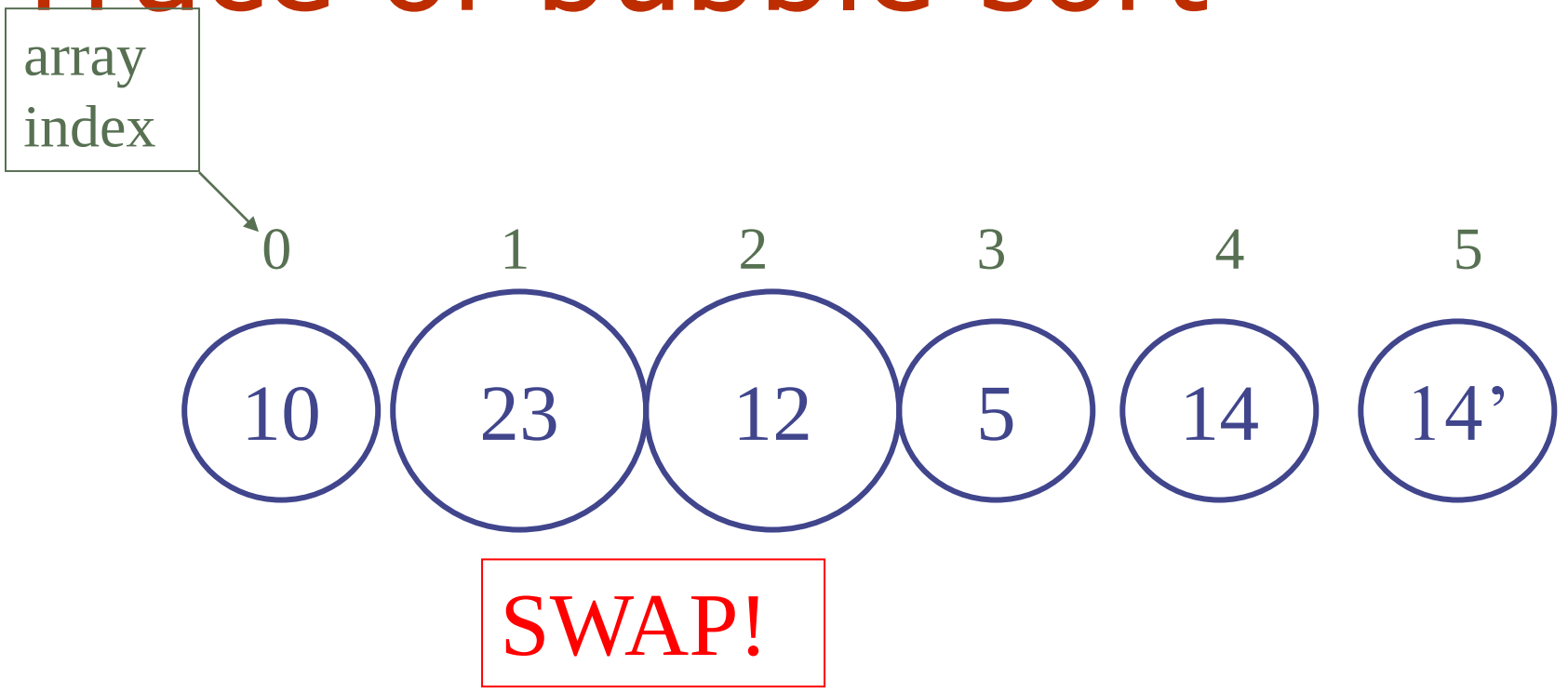
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 1, comparing arr[1] and arr[2]

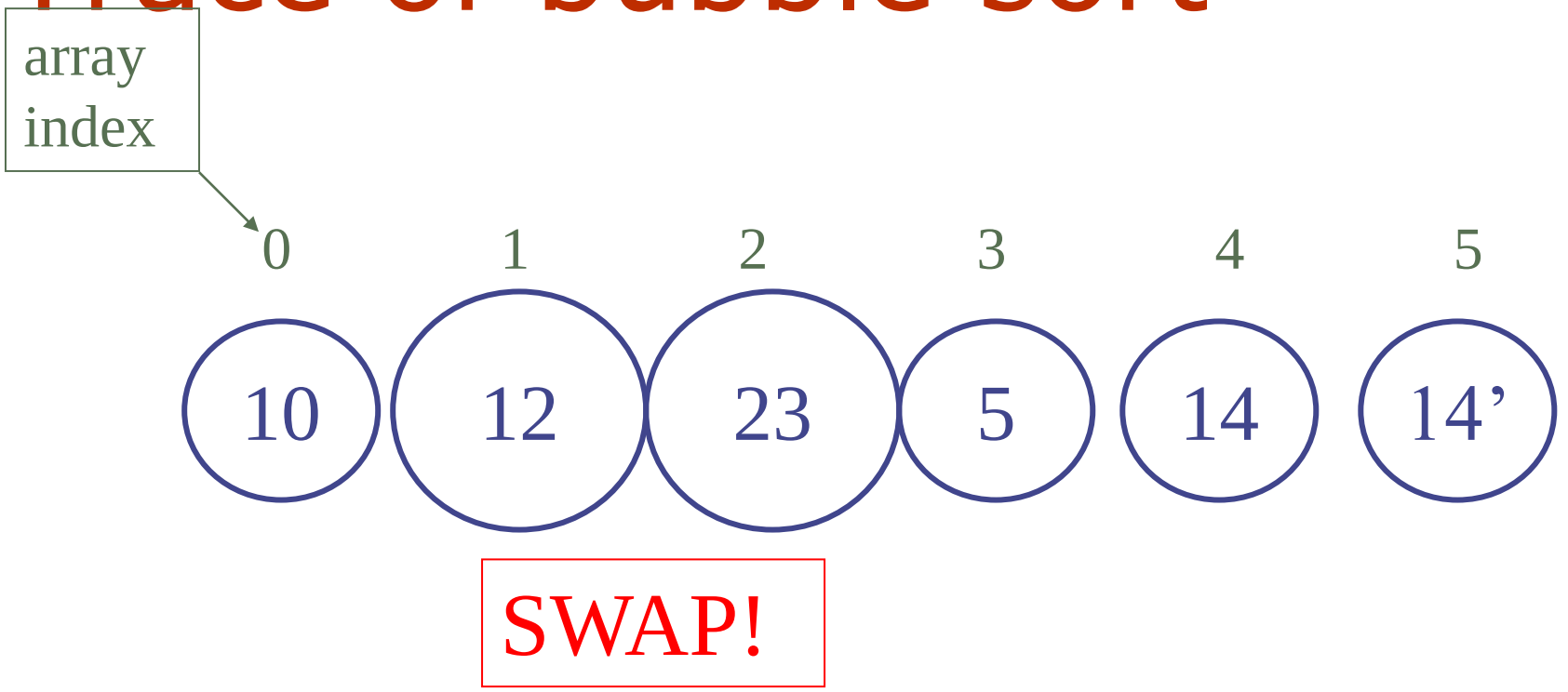
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 1, comparing arr[1] and arr[2]

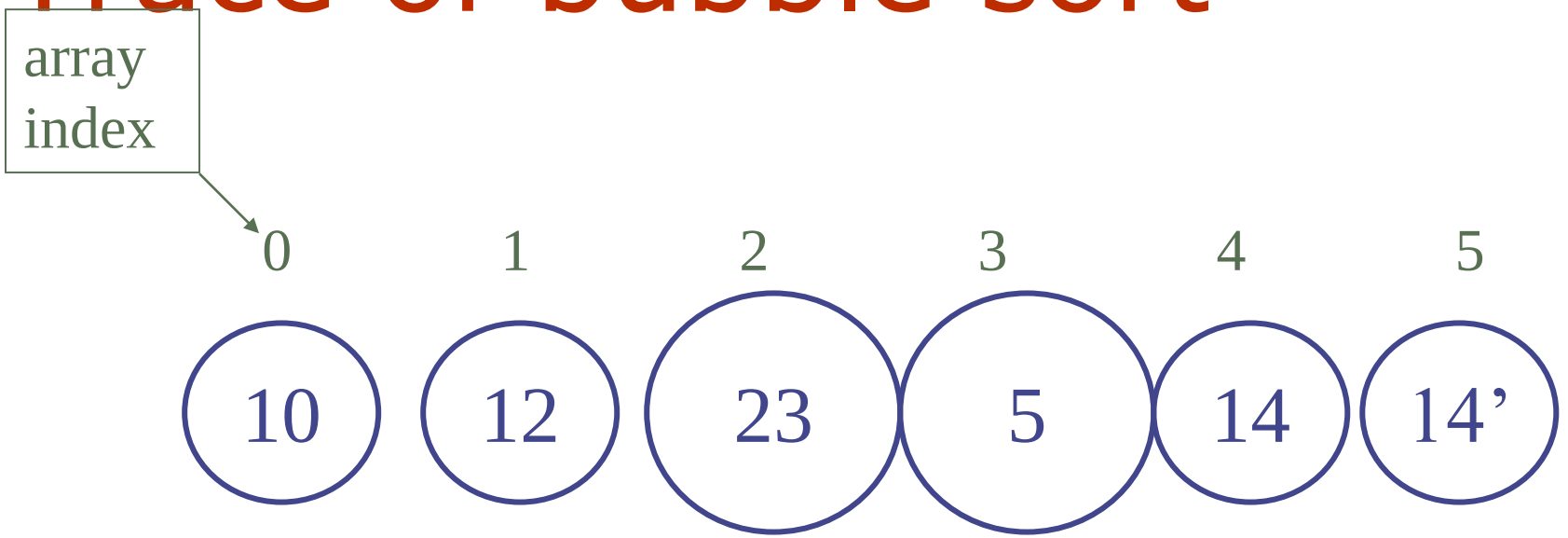
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 1, comparing arr[1] and arr[2]

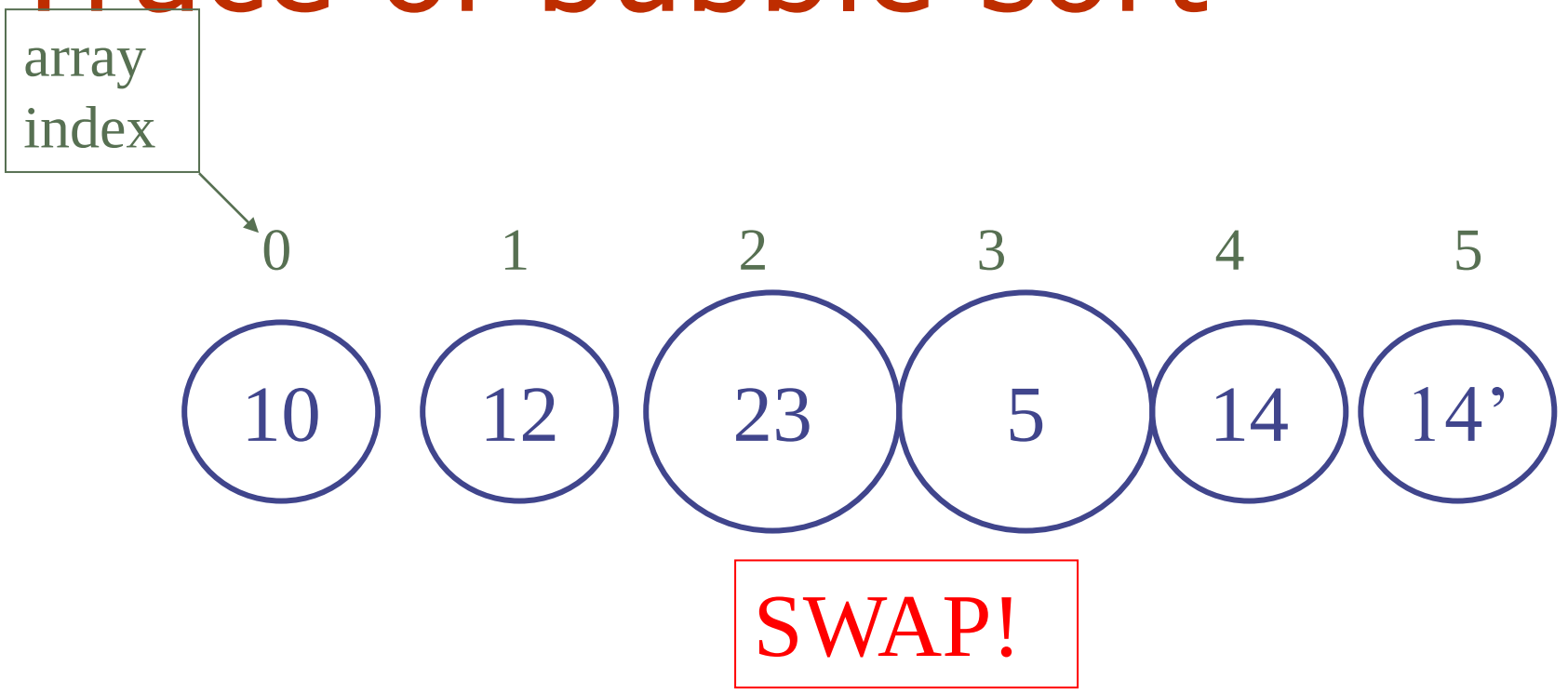
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 2, comparing arr[2] and arr[3]

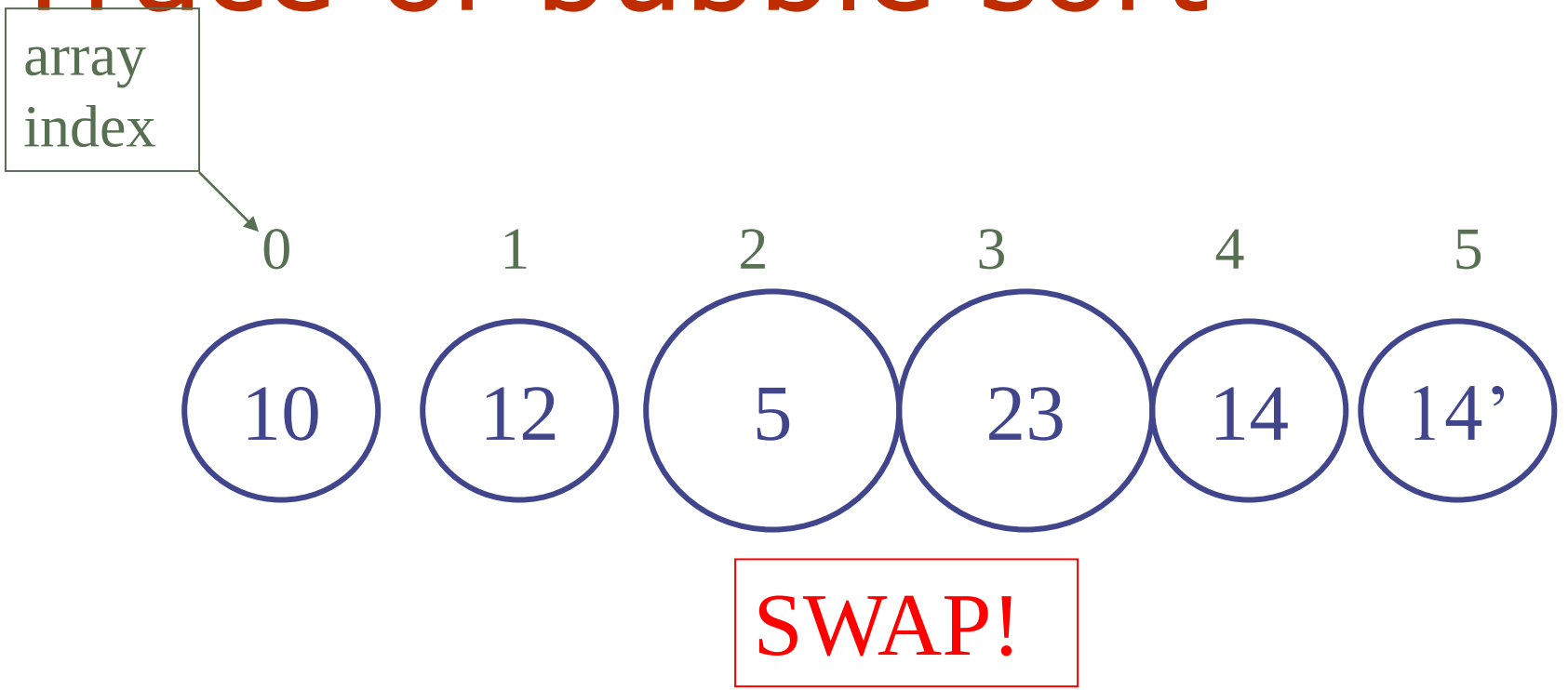
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 2, comparing arr[2] and arr[3]

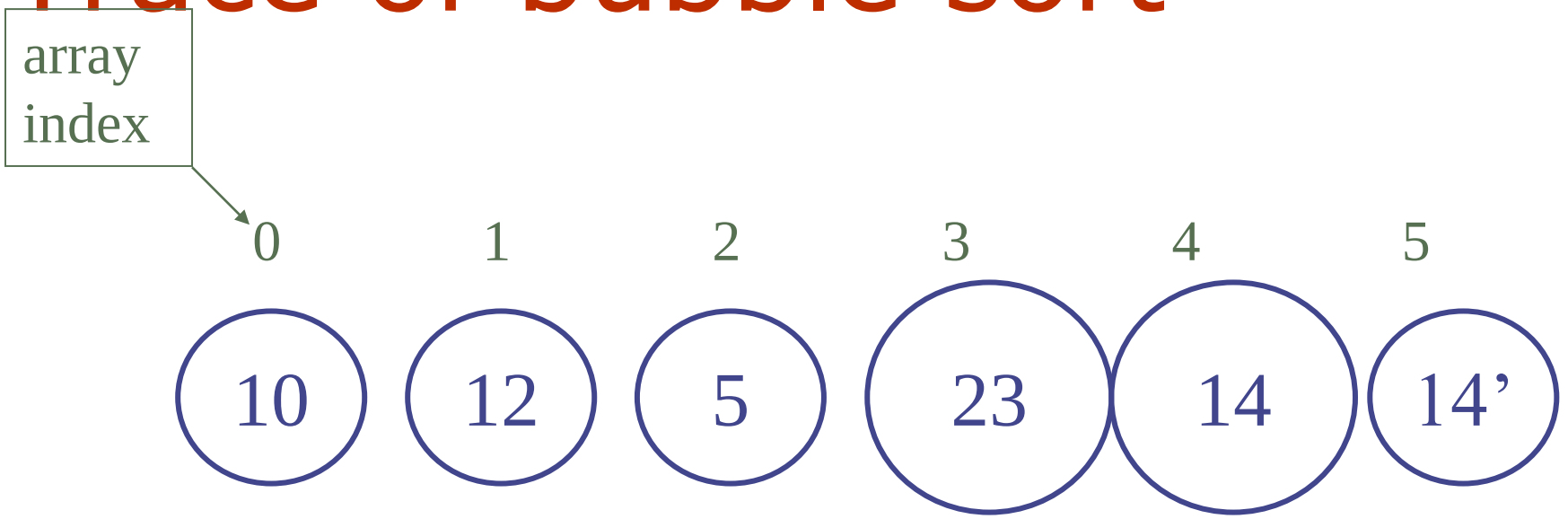
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 2, comparing arr[2] and arr[3]

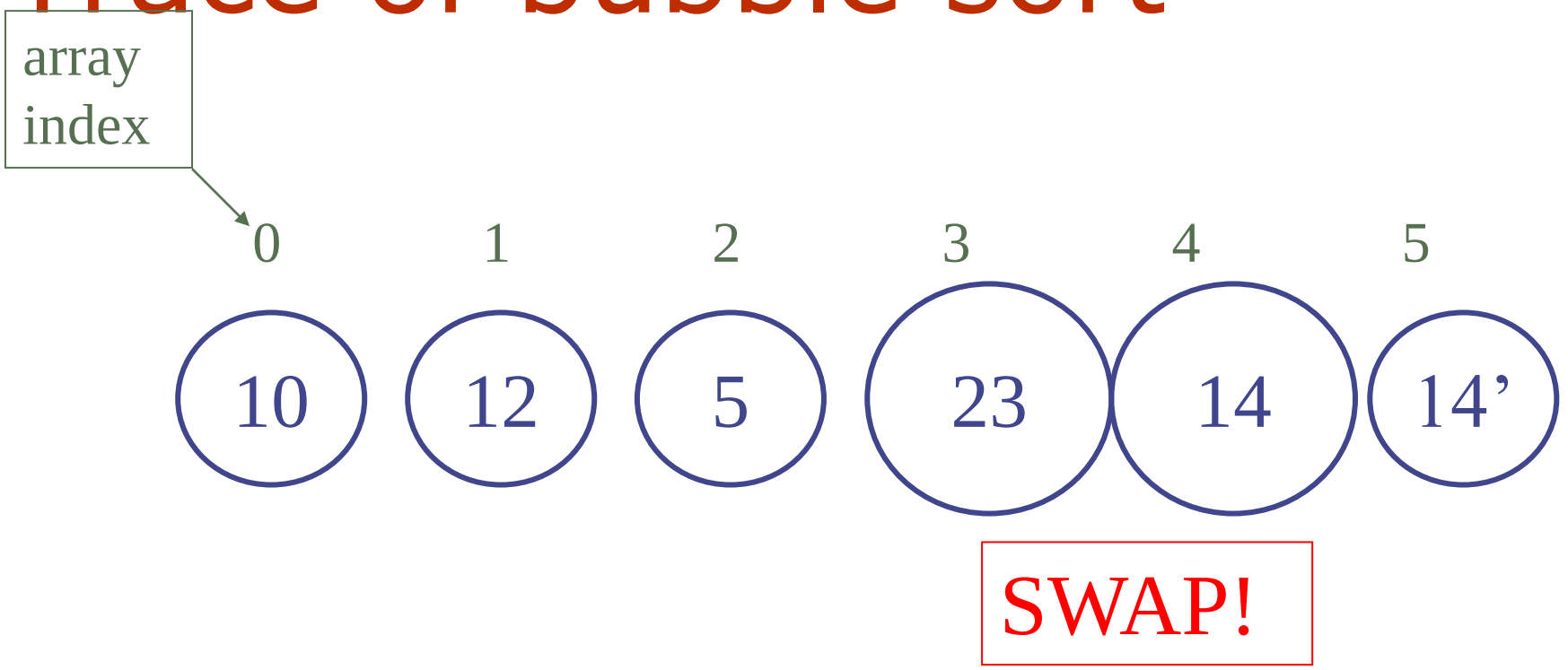
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 3, comparing arr[3] and arr[4]

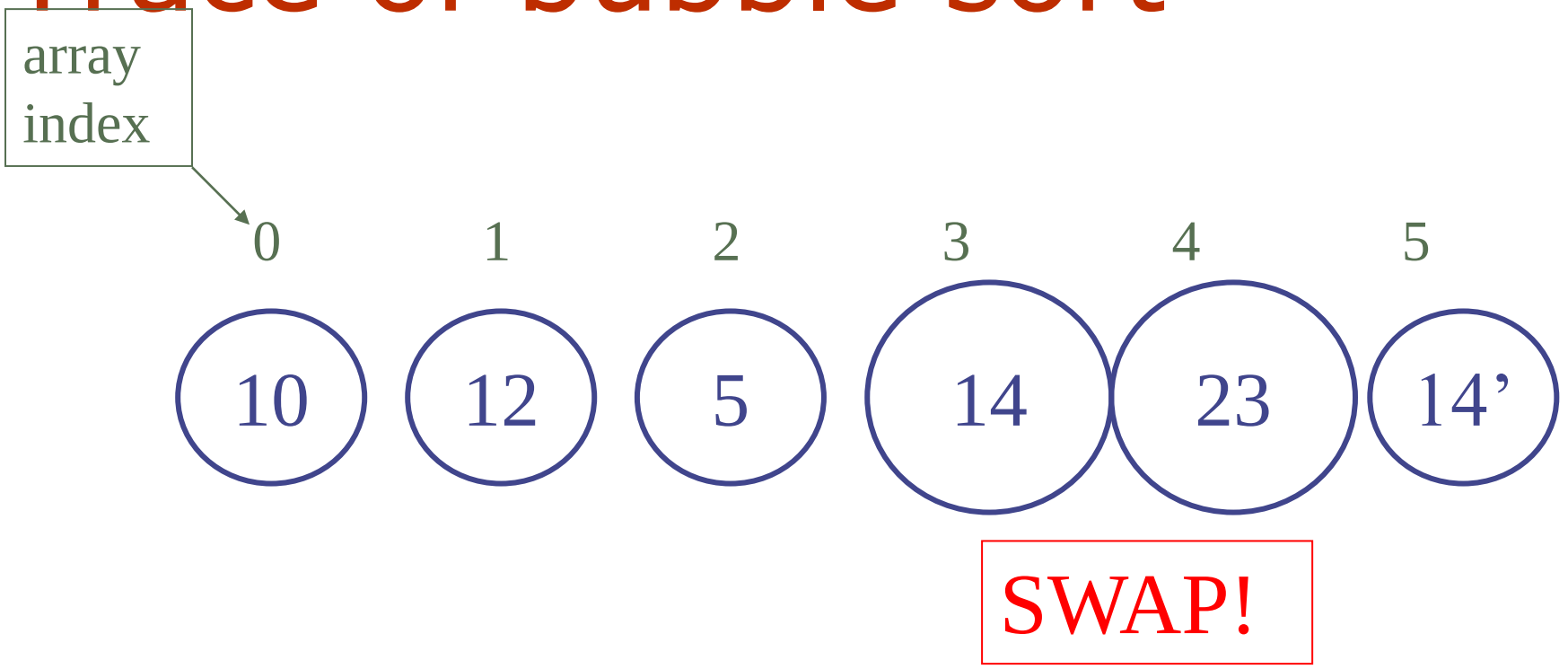
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 3, comparing arr[3] and arr[4]

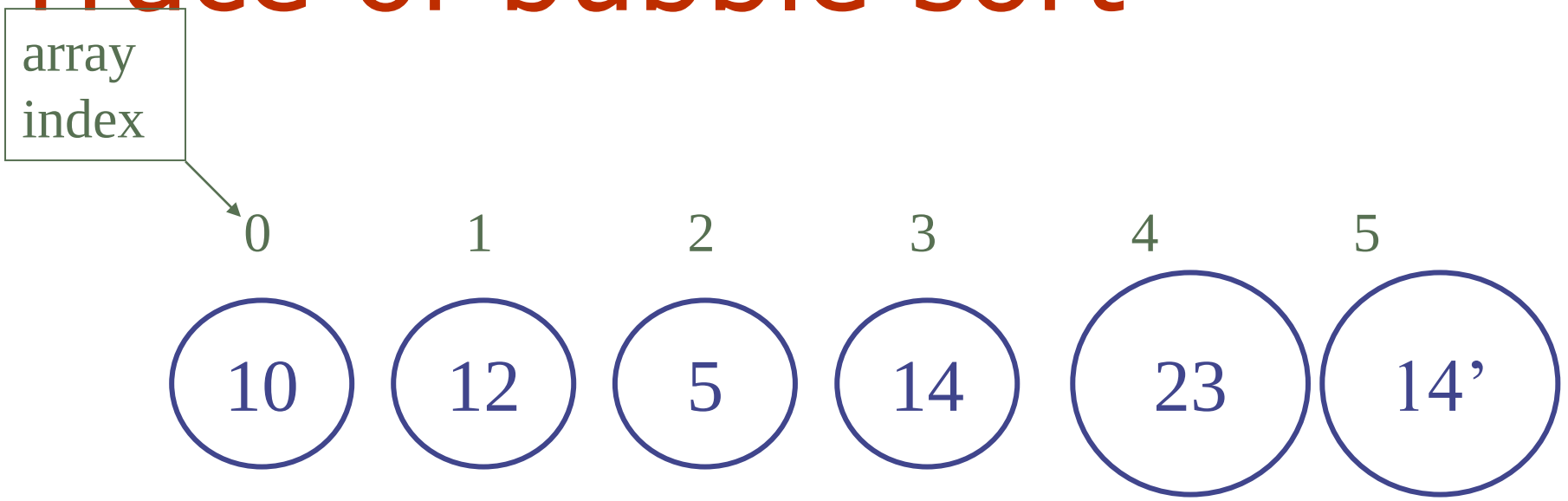
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 3, comparing arr[3] and arr[4]

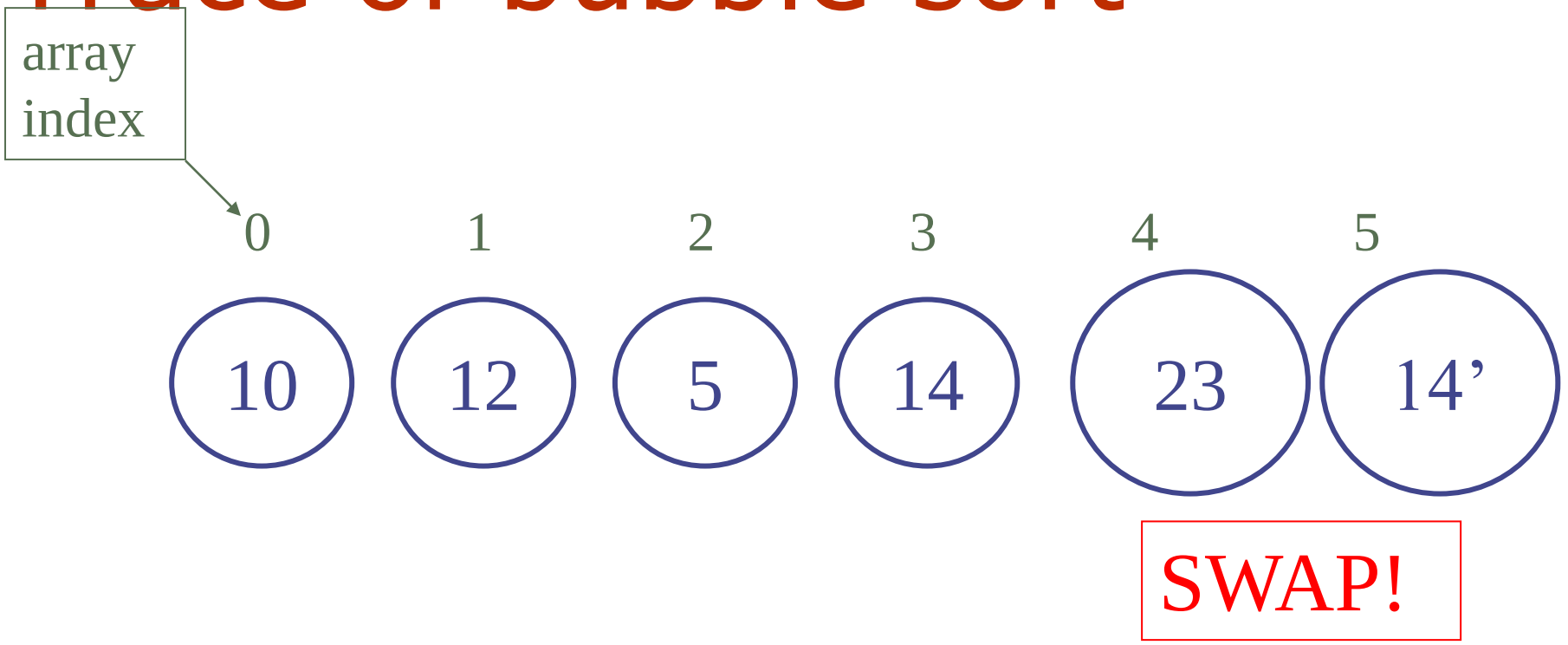
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 4, comparing arr[4] and arr[5]

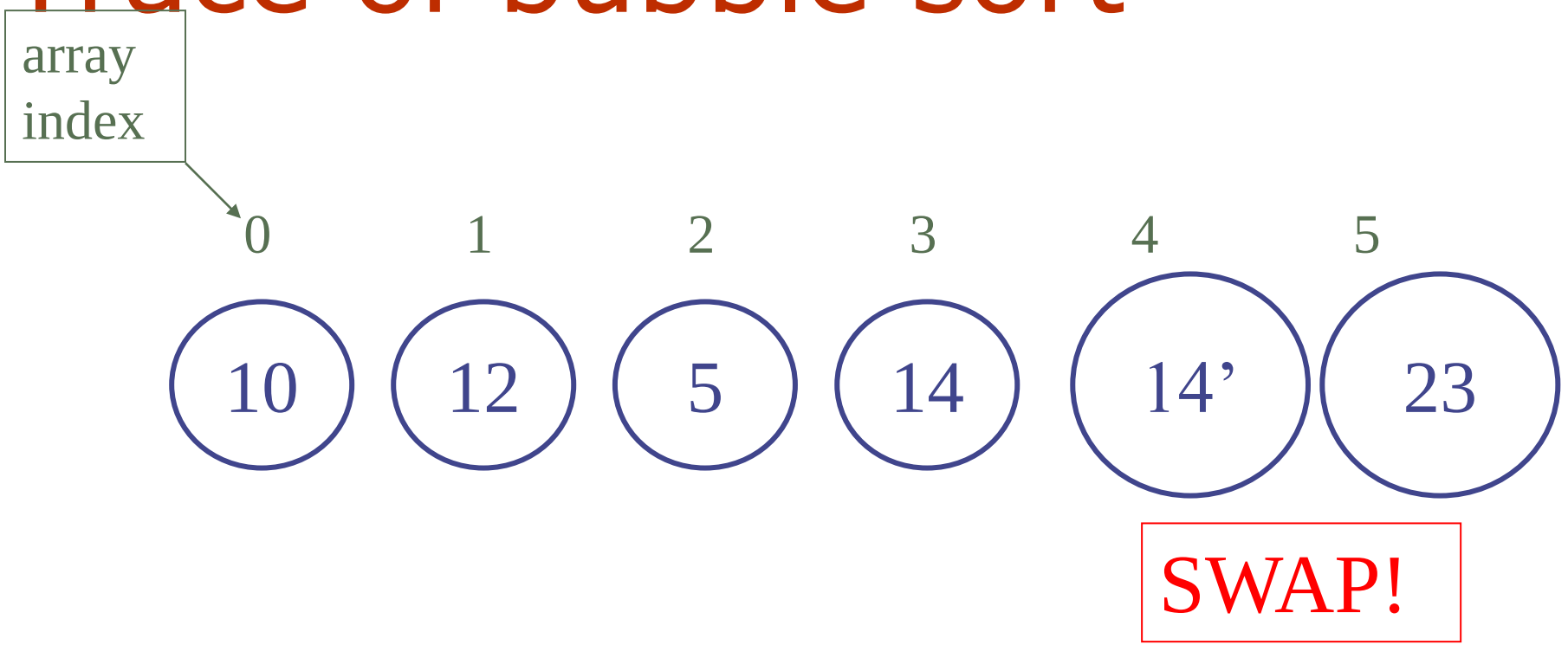
Trace of bubble sort



i = 5, first iteration of the outer loop

j = 4, comparing arr[4] and arr[5]

Trace of bubble sort



i = 5, first iteration of the outer loop

j = 4, comparing arr[4] and arr[5]

Trace of bubble sort

i = 5, first iteration of the outer loop

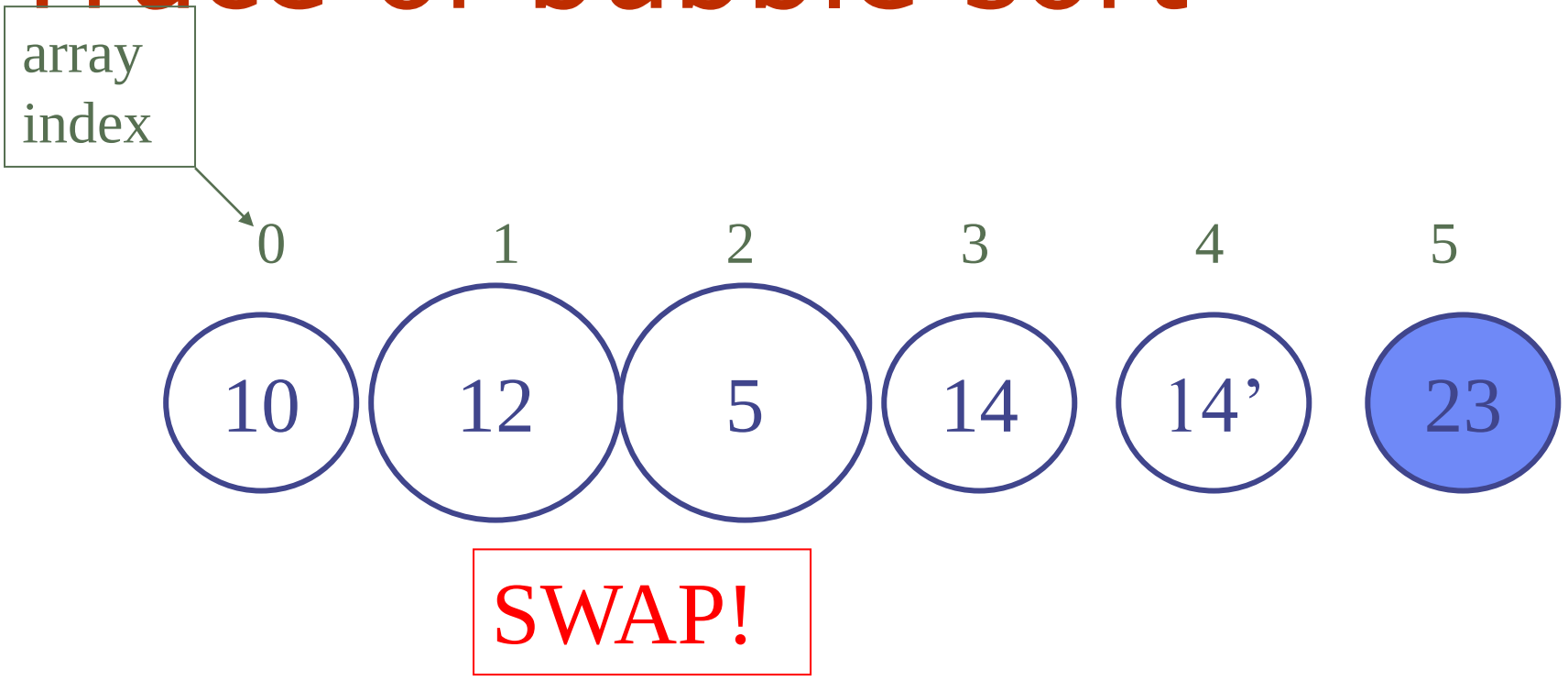
inner loop finished; largest element
in position 5, positions 0-4 unsorted

Trace of bubble sort

i = 4, second iteration of the outer loop

j = 0, comparing arr[0] with arr[1]

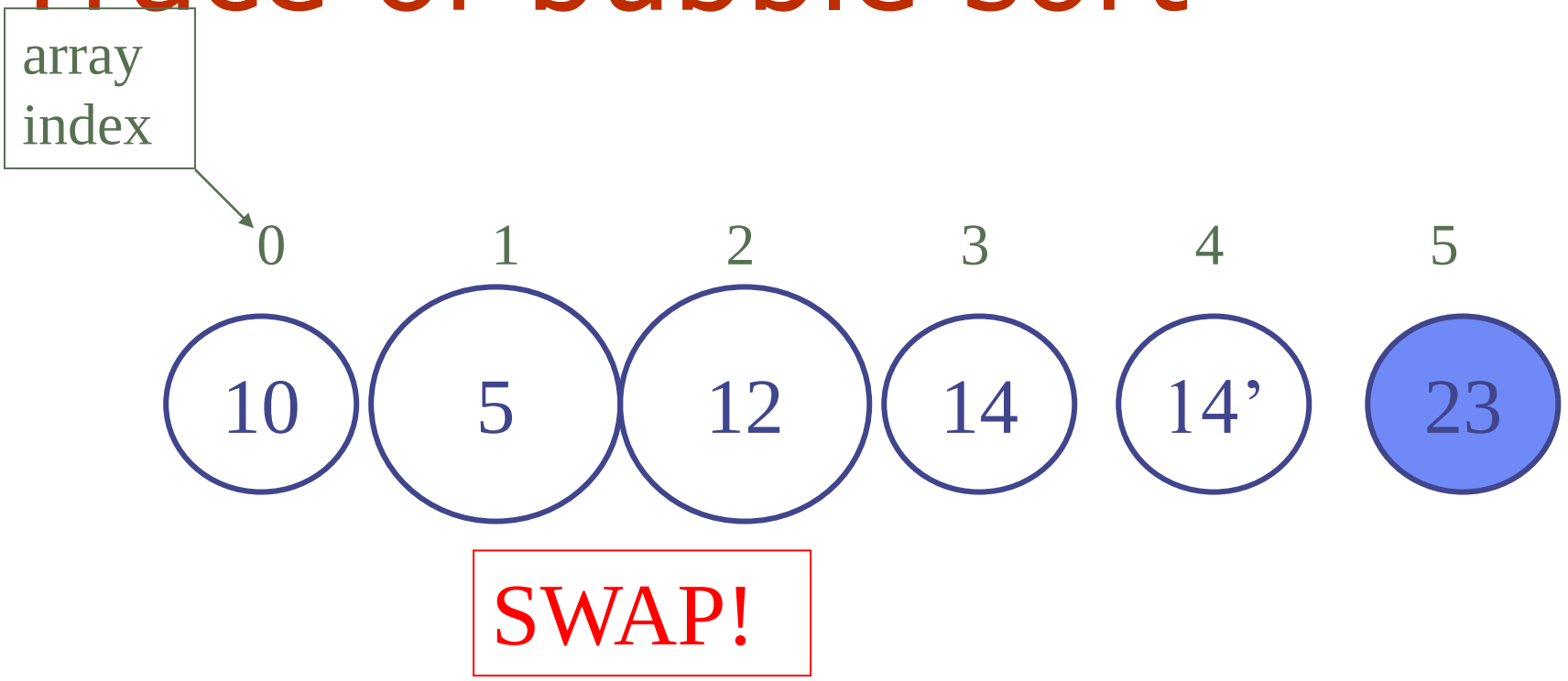
Trace of bubble sort



i = 4, second iteration of the outer loop

j = 1, comparing arr[1] with arr[2]

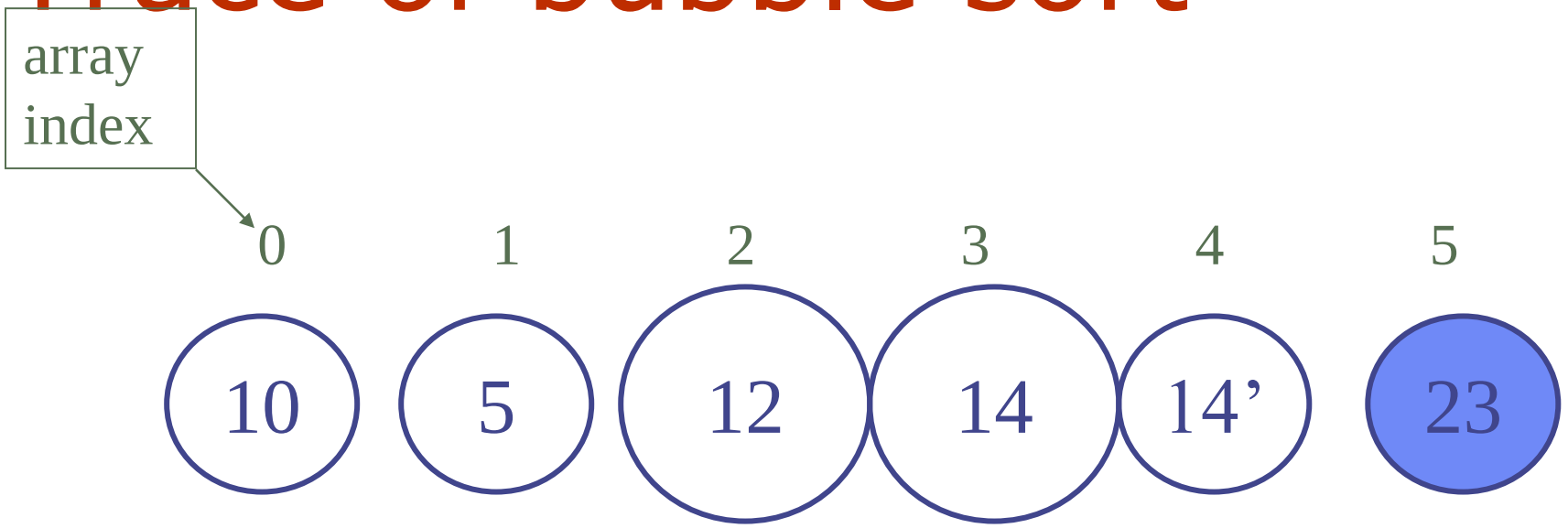
Trace of bubble sort



i = 4, second iteration of the outer loop

j = 1, comparing arr[1] with arr[2]

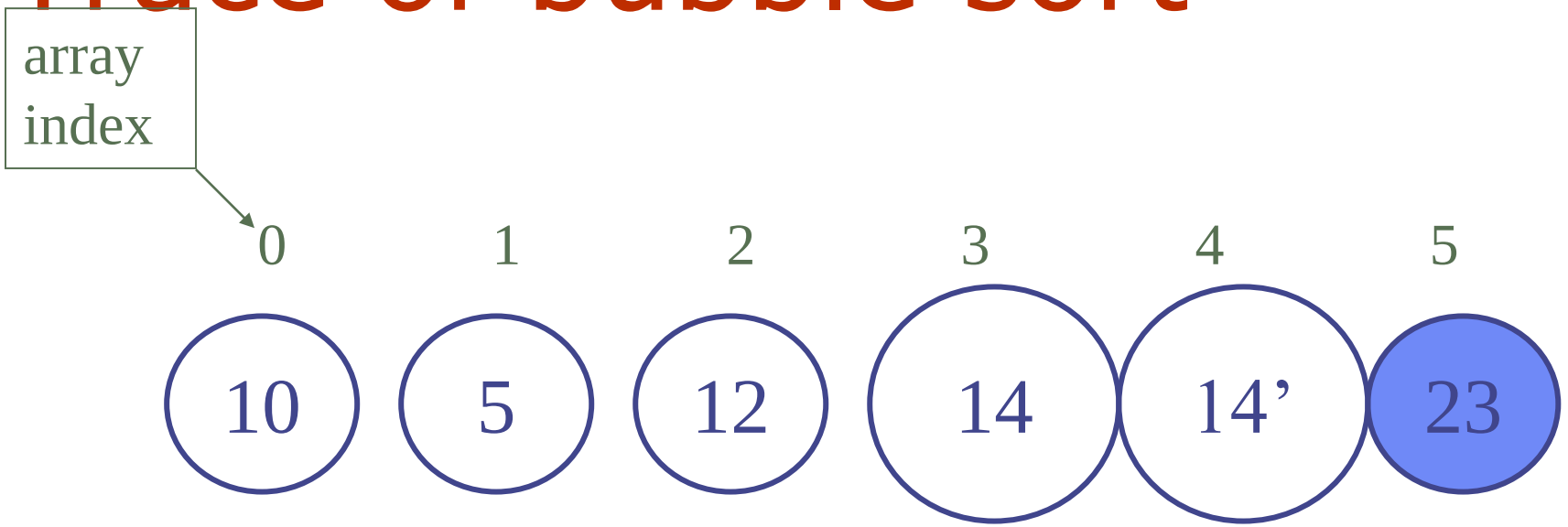
Trace of bubble sort



i = 4, second iteration of the outer loop

j = 2, comparing arr[2] with arr[3]

Trace of bubble sort



i = 4, second iteration of the outer loop

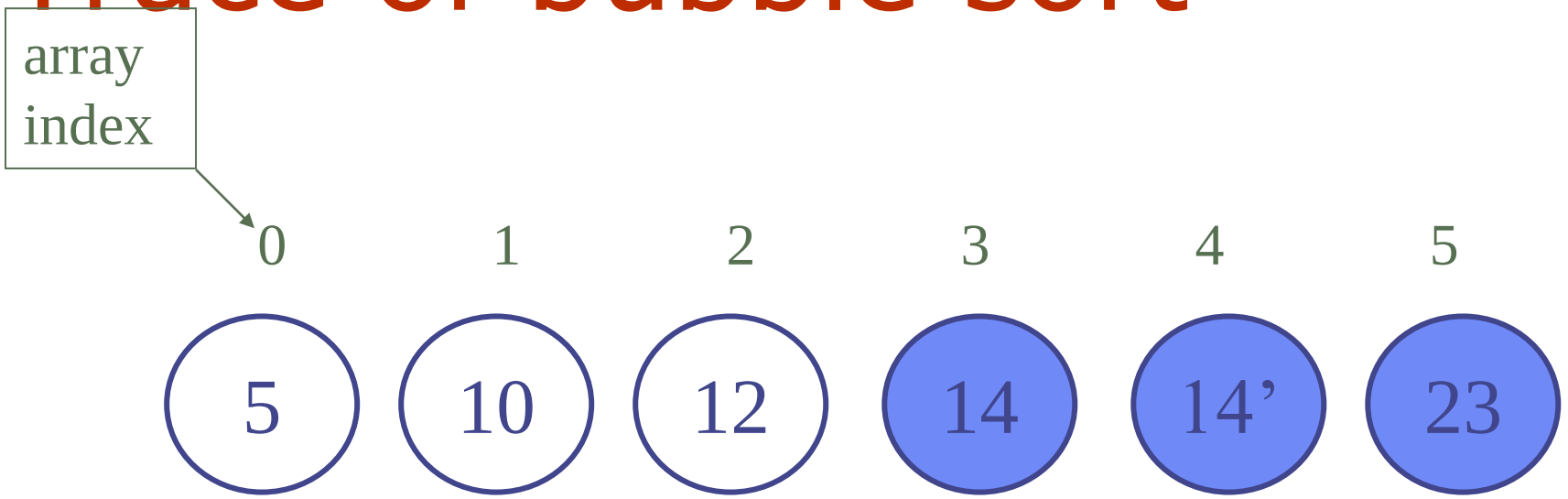
j = 3, comparing arr[3] with arr[4]

Trace of bubble sort

i = 4, second iteration of the outer loop

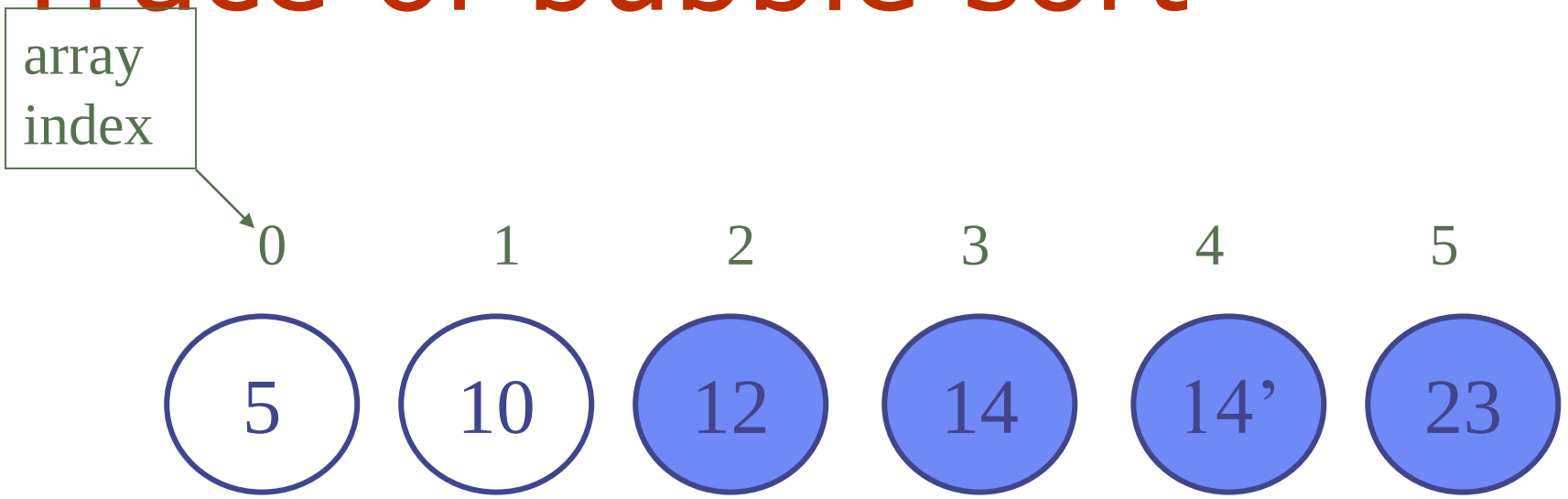
inner loop finished, second largest element in position 4, positions 0-3 unsorted

Trace of bubble sort



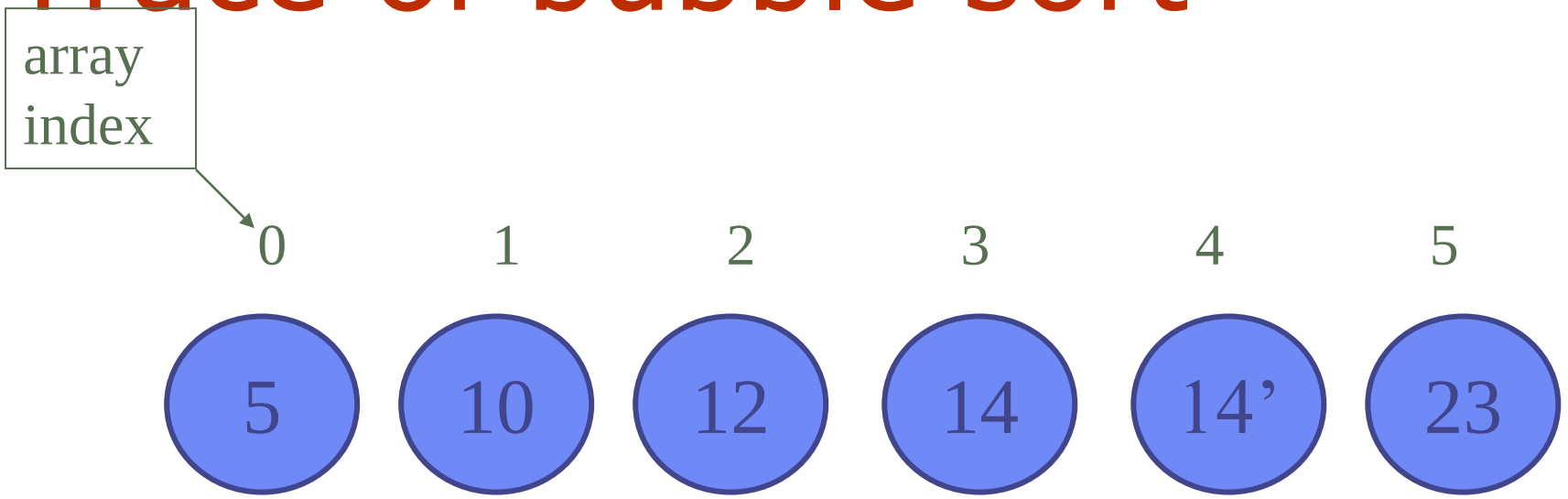
After third iteration…

Trace of bubble sort



After fourth iteration…

Trace of bubble sort



After fifth iteration…

Note: 14 and 14’ are in same relative order as they started, hence:

bubble sort is “stable”

Sorting “Stability”

- If sorting int[] then it does not really matter if the entries are swapped
- So why care about stability at all?

- NOTE: “stable” does NOT mean “the code does not break” !!
  - This is a common error!

Sorting “Stability”

- Often we sorting objects according to some comparison function: compare(o1, o2) returns negative, zero, or positive for “less than”, “equal”, or “greater than”
- compare(o1,o2)==0 means objects o1 and o2 are
  - equal with respect to the desired ordering
  - but not necessarily that they have the same contents
    - i.e. are not identical !

- E.g. object is a row of a spreadsheet and compare uses just one specified column - many different rows can be equal, but not identical

Sorting "Stability"

- If sorting a spreadsheet, then might sort by one column then another.
- Do not want the sorting to unnecessarily change the order of the rows, as this can be annoying and confusing.
- *"Sort by column A, followed by a stable sort on column B"*
  means that still will have a secondary sort on column A

- **EXERCISE (offline): Experiment with Excel to see if it does stable sorts or not.**

Complexity of bubble sort

- For an array of size n, in the worst case:
  1st passage through the inner loop: n-1 comparisons and n-1 swaps
- ...
- (n-1)st passage through the inner loop: one comparison and one swap.
- All together: t ((n-1) + (n-2) + ... + 1), where t is the time required to do one comparison, one swap, check the inner loop condition and increment j.
- We also spend constant time k declaring i,j,temp and initialising i. Outer loop is executed n-1 times, suppose the cost of checking the loop condition and decrementing i is t_1.

Complexity of bubble sort

$t\ ((n-1) + (n-2) + \ldots + 1) + k + t_1(n-1)$

$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$

["arithmetic sum" see self-study material]

so our function equals

$$t\ n*(n-1)/2 + k + t_1(n-1) =$$
$$\tfrac{1}{2}\ t\ (n^2-n) + t_1\ (n-1) + k$$

(worst-case) complexity $O(n^2)$.

Proof: Complexity of bubble sort (study offline)

Need to find n_0 and c, such that for all $n \geq n_0$, $\frac{1}{2}\, t\, (n^2-n) + t_1(n-1) + k \leq c * n^2$

$\frac{1}{2}\, t\, n^2 - \frac{1}{2}\, t\, n + t_1 n - t_1 + k \leq$

$\frac{1}{2}\, t\, n^2 + t_1 n + k \leq$

$t\, n^2 + t_1\, n^2 + k\, n^2$ (if $n \geq 1$)

Take $c = t + t_1 + k$ and $n_0 = 1$.

Selection Sort: Basic Idea

- Similar to bubble sort;
- On each scan:
  - instead of always try to move the “greatest element so far” immediately, we just remember its location and move it at end of scan

Why one want to do this!?

Exercise: Why delay swaps?

Answer:

- Suppose that the entries are large then a swap operation might be quite expensive
- So might want to reduce the number of swaps by directly moving entries to "the right place"

Selection sort

```
void  selectionSort(int arr[]){
  int i, j, temp, pos_greatest;
  for( i = arr.length-1; i > 0; i--){
     pos_greatest = 0;
     for(j = 0; j <= i; j++){
       if( arr[j] >= arr[pos_greatest])
          pos_greatest =  j;
     }//end inner for loop
     if ( i != pos_greatest ) {
       temp = arr[i];
       arr[i] = arr[pos_greatest];
        arr[pos_greatest] = temp; }
  }//end outer for loop
}//end selection sort
```

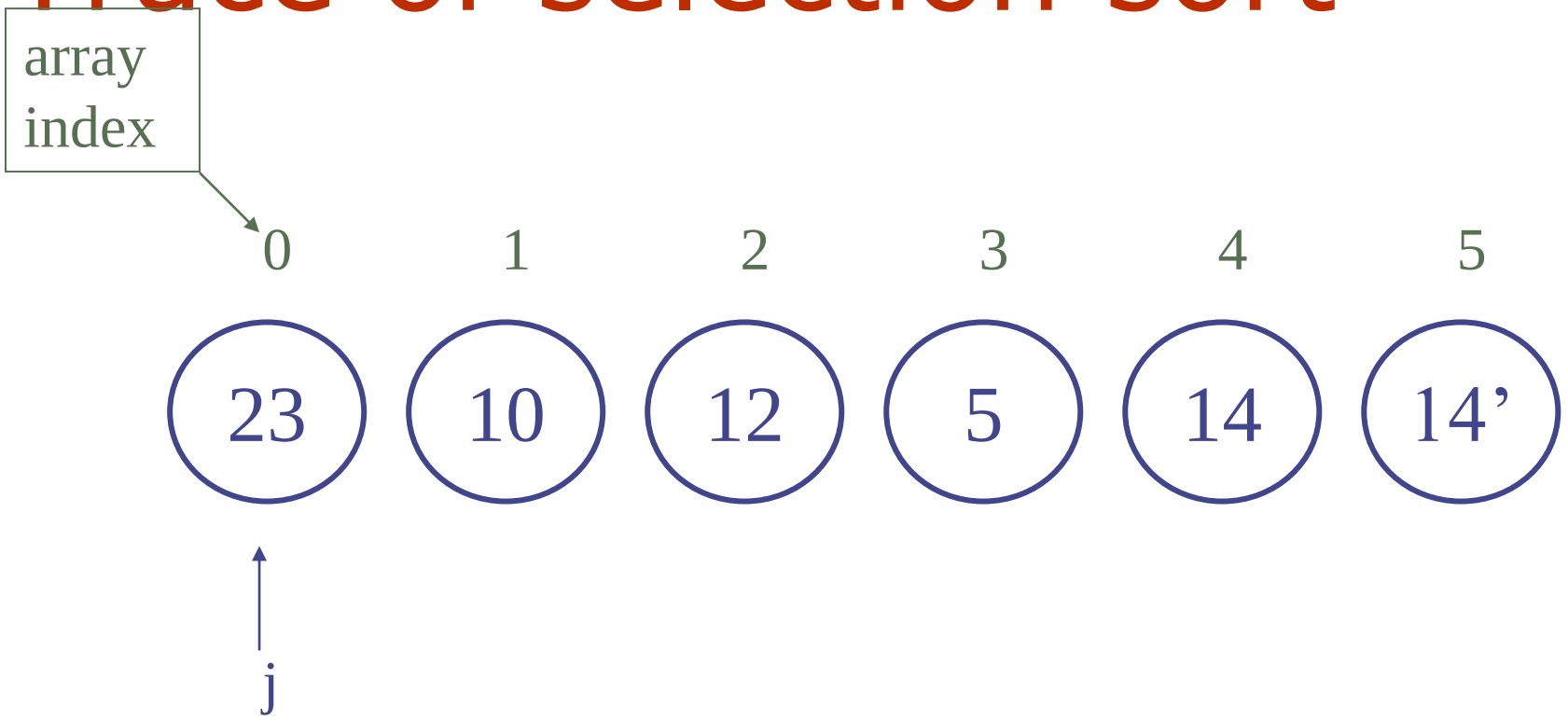
compare the current element to the largest seen so far; if it is larger, remember its index

swap the largest element to the end of range, if not already there

Trace of selection sort

i = 5, first iteration of the outer loop

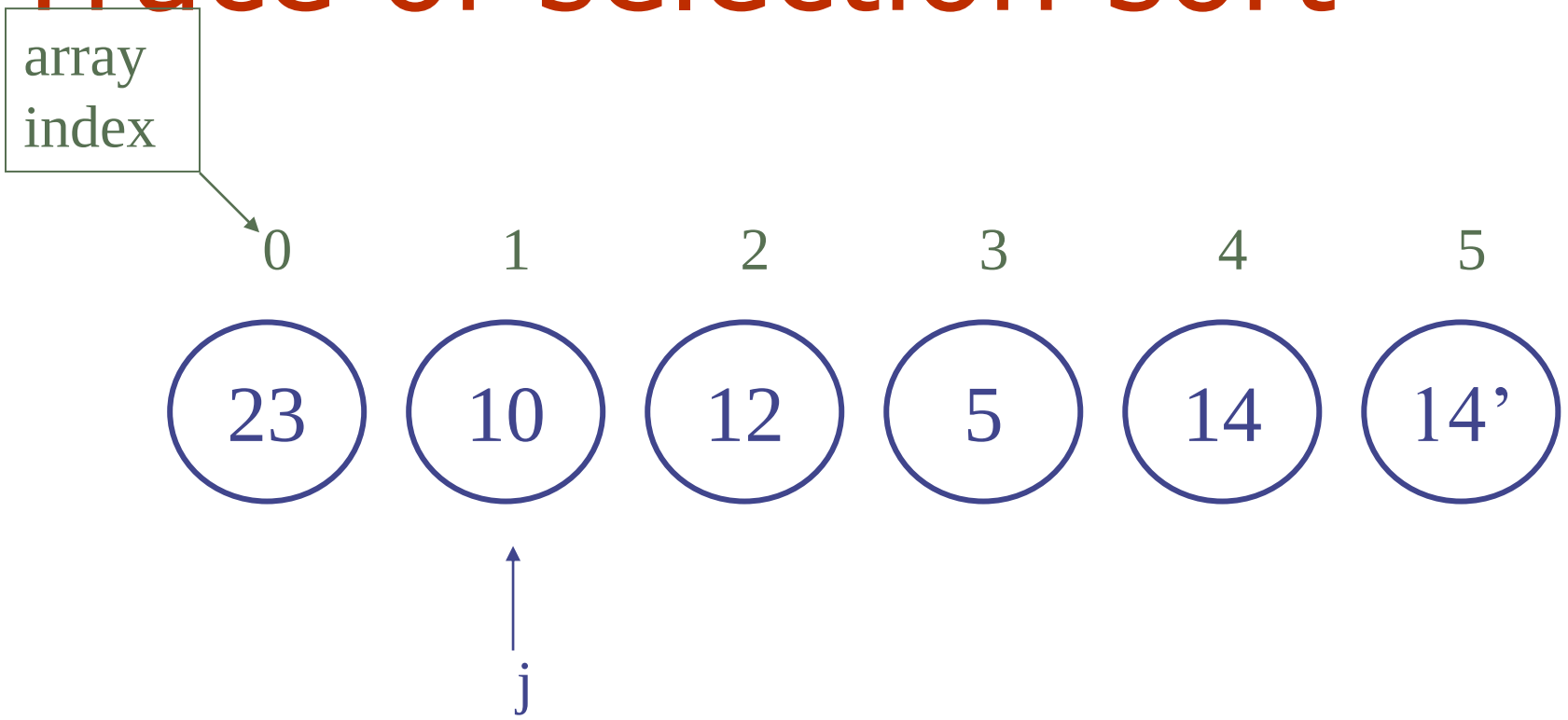
Trace of selection sort



i = 5, first iteration of the outer loop

j = 0, pos_greatest = 0

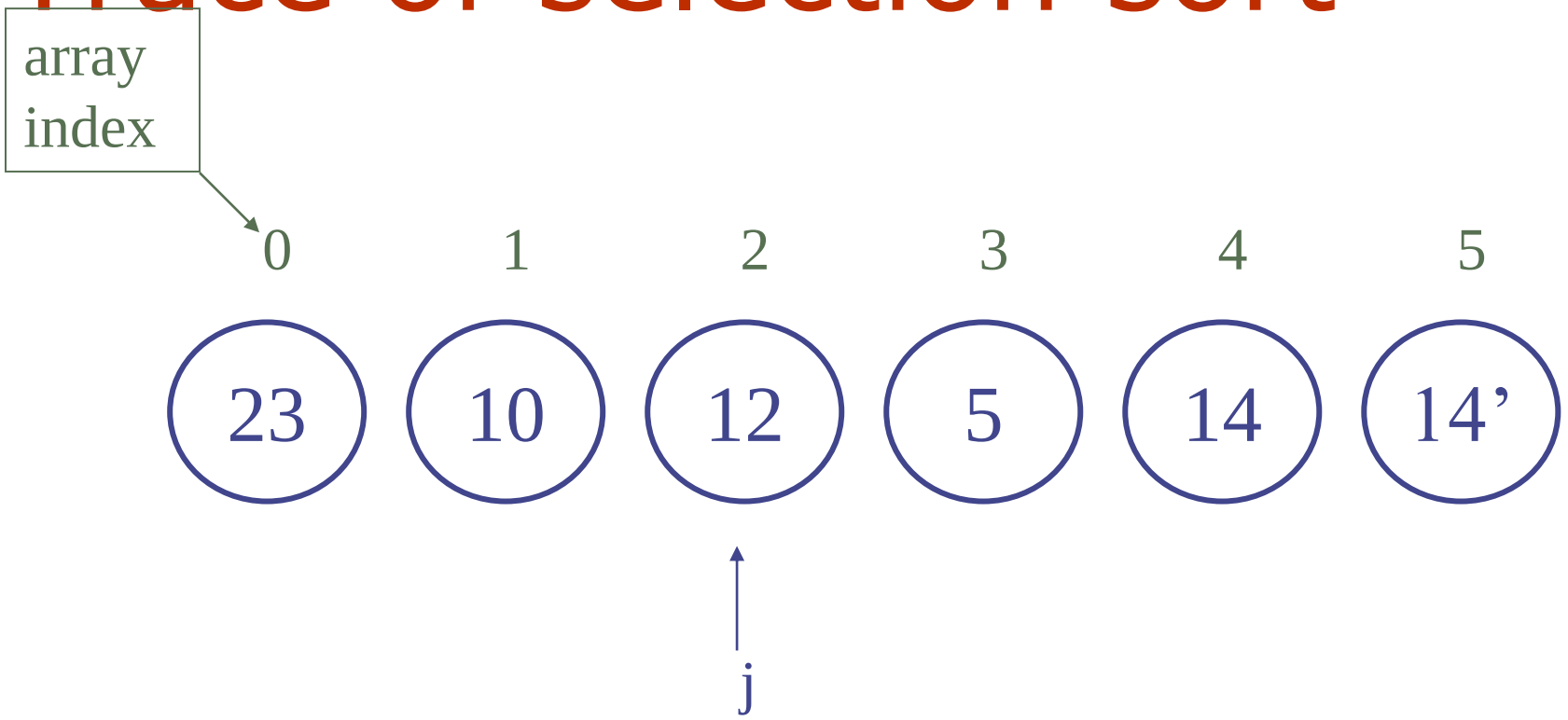
Trace of selection sort



i = 5, first iteration of the outer loop

j = 1, pos_greatest = 0

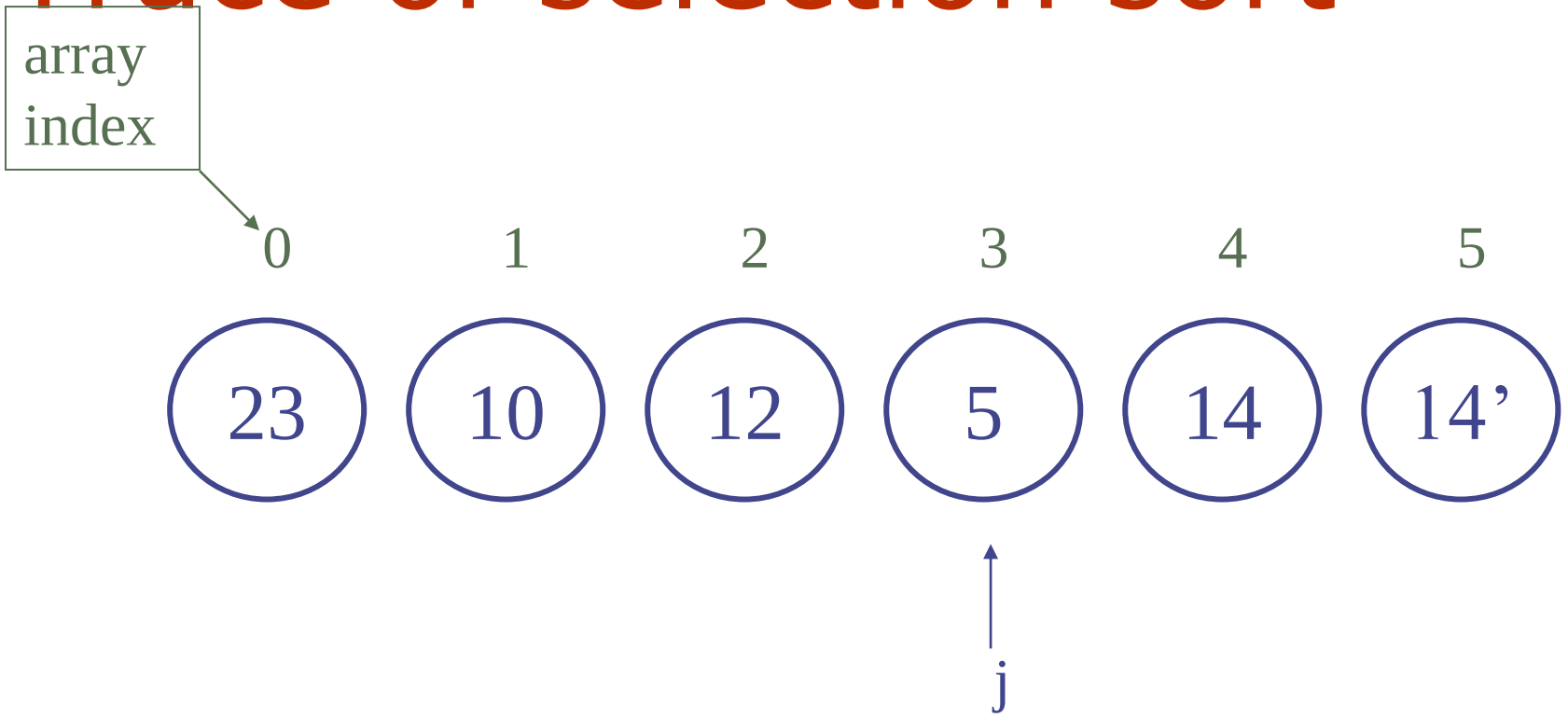
Trace of selection sort



i = 5, first iteration of the outer loop

j = 2, pos_greatest = 0

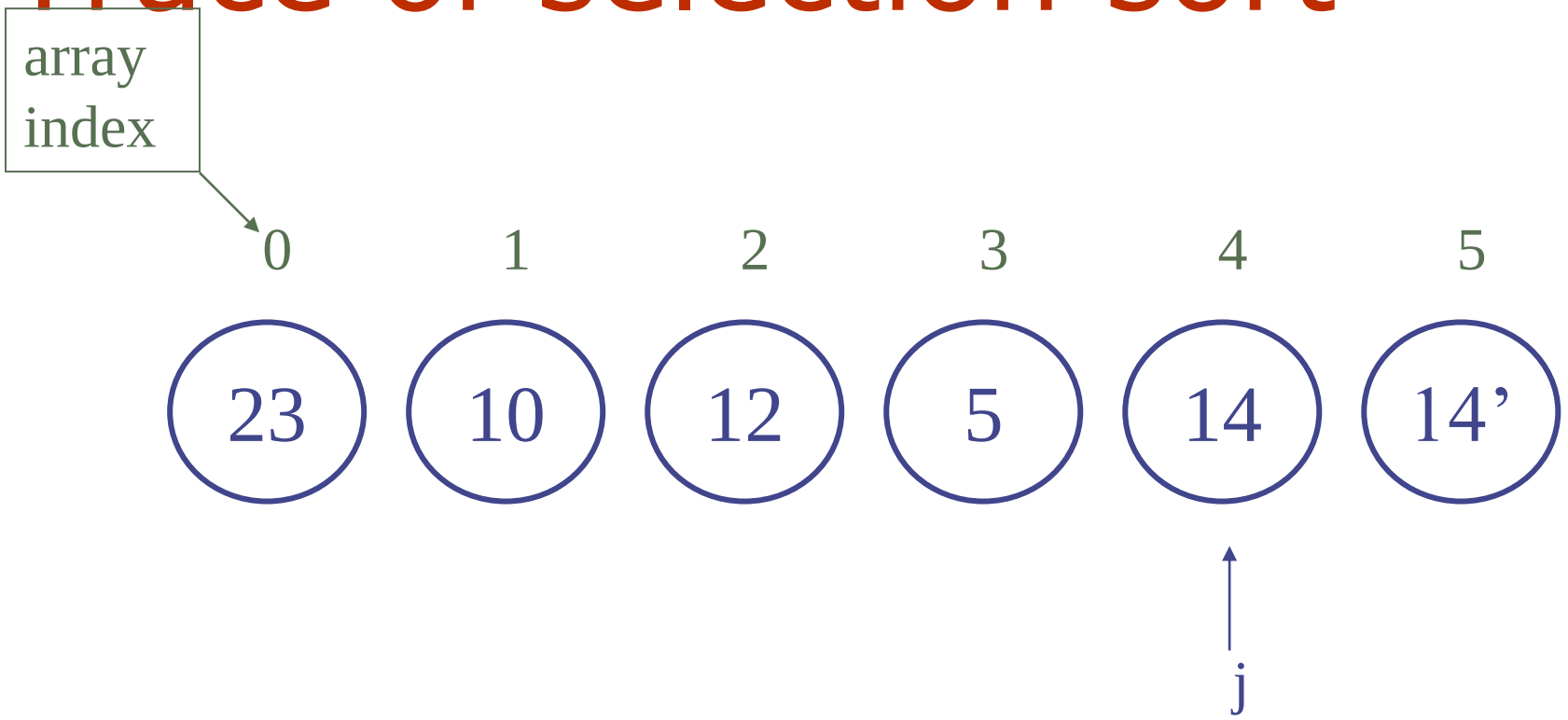
Trace of selection sort



i = 5, first iteration of the outer loop

j = 3, pos_greatest = 0

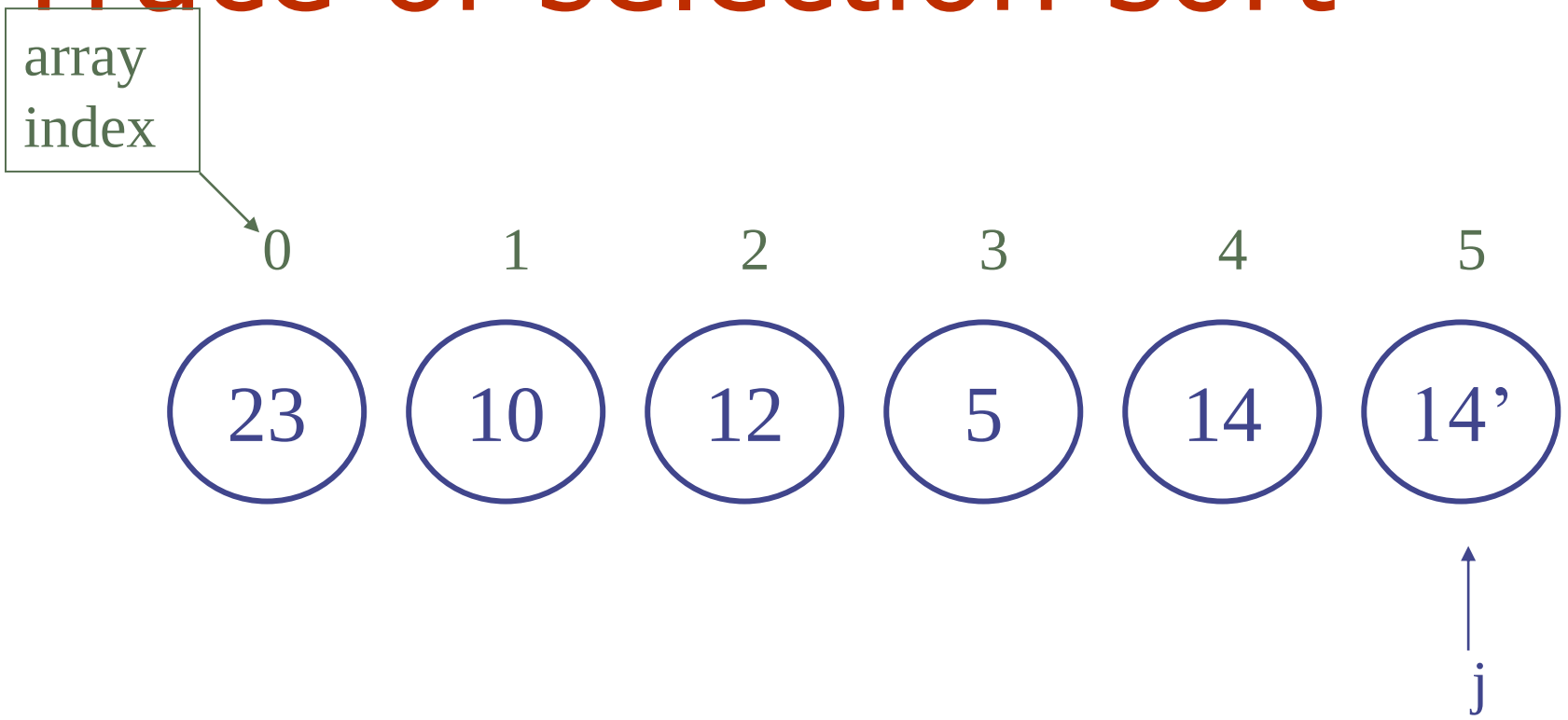
Trace of selection sort



i = 5, first iteration of the outer loop

j = 4, pos_greatest = 0

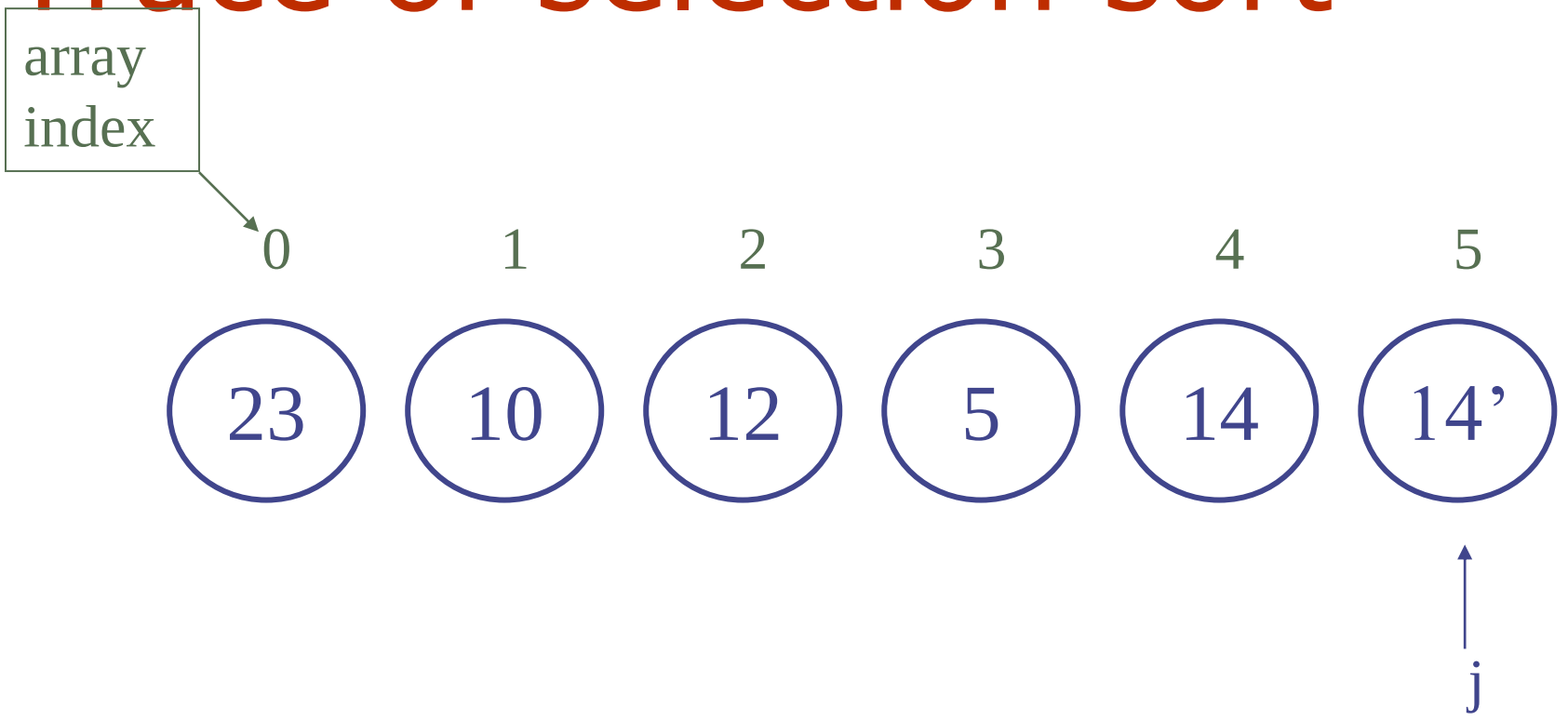
Trace of selection sort



i = 5, first iteration of the outer loop

j = 5, pos_greatest = 0

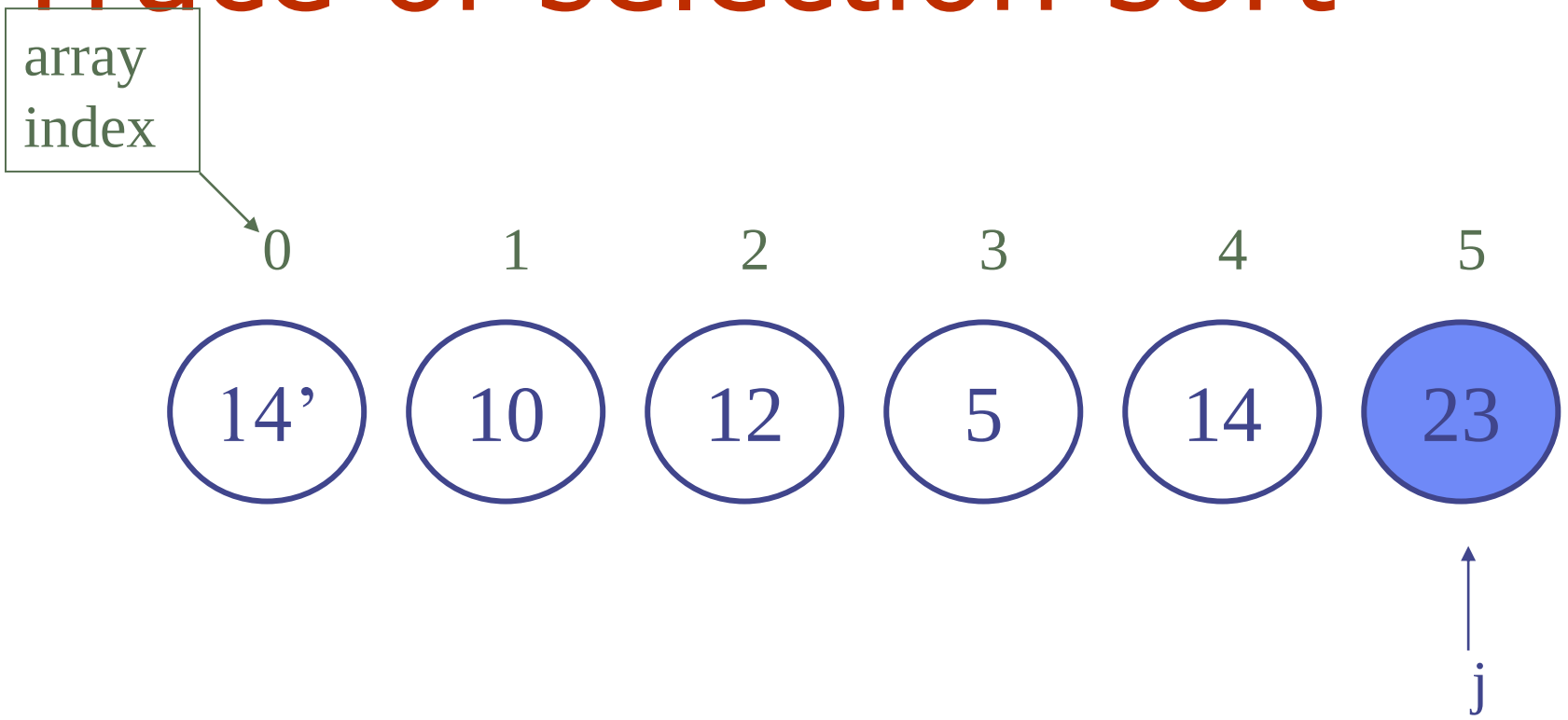
Trace of selection sort



i = 5, first iteration of the outer loop

swap element at pos_greatest=0 to 5

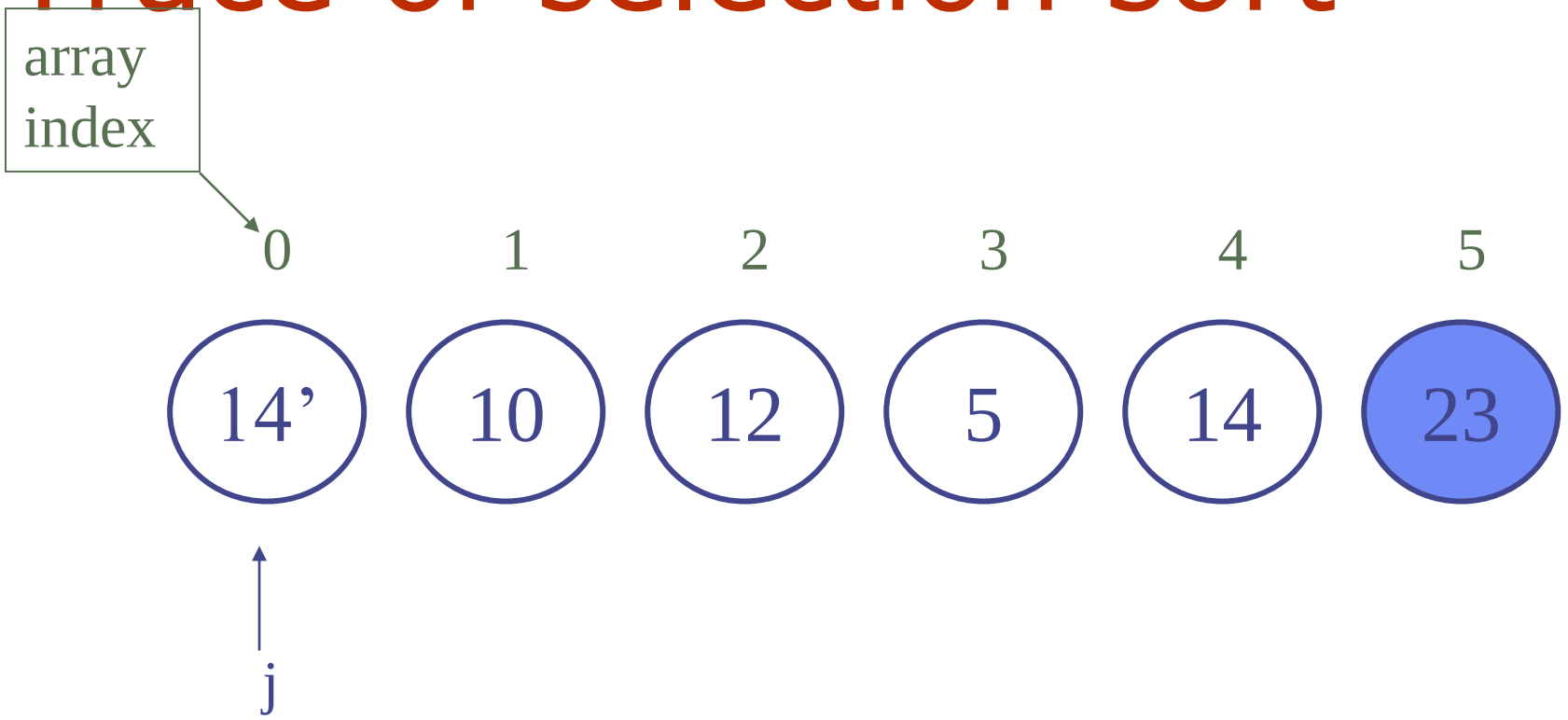
Trace of selection sort



i = 5, first iteration of the outer loop

swap element at pos_greatest=0 to 5

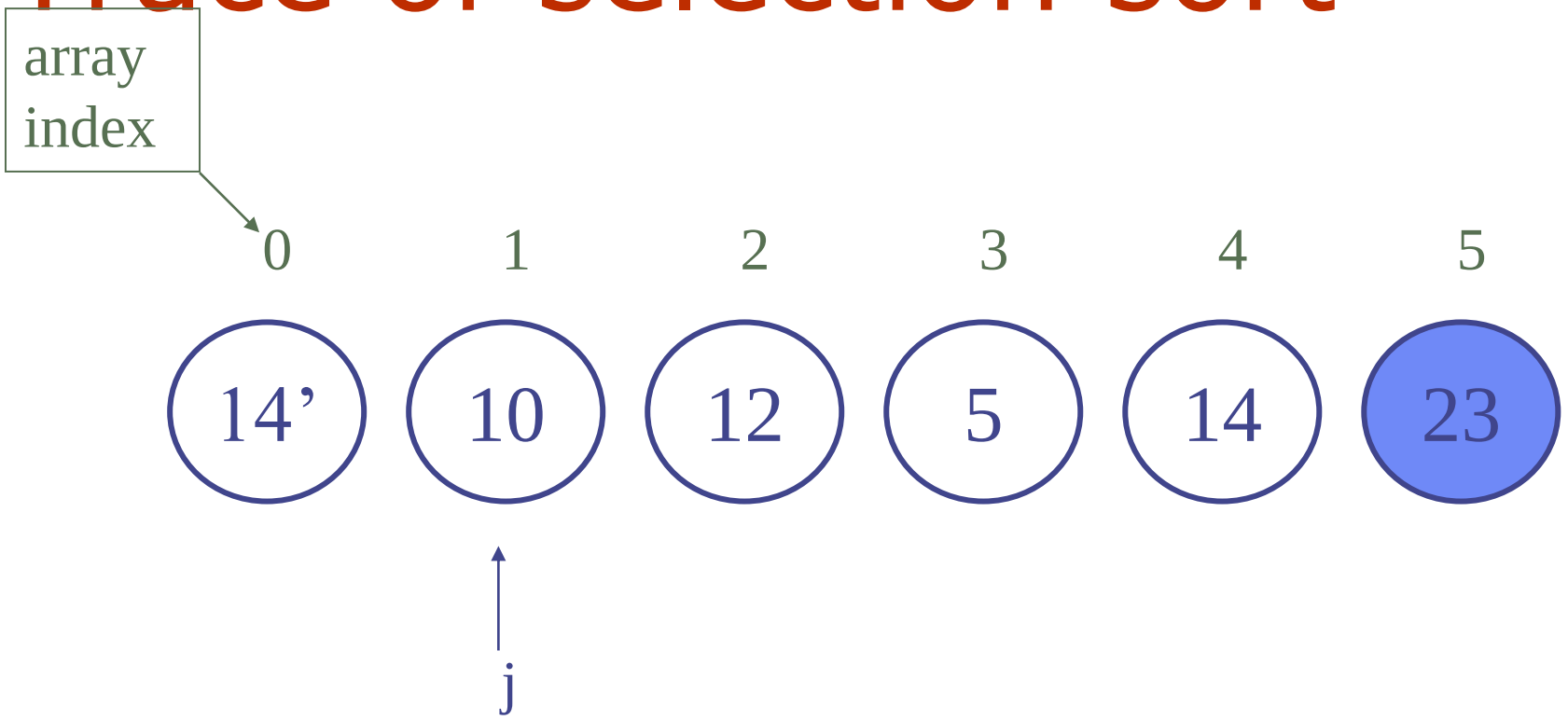
Trace of selection sort



i = 4, second iteration of the outer loop

j = 0, pos_greatest = 0

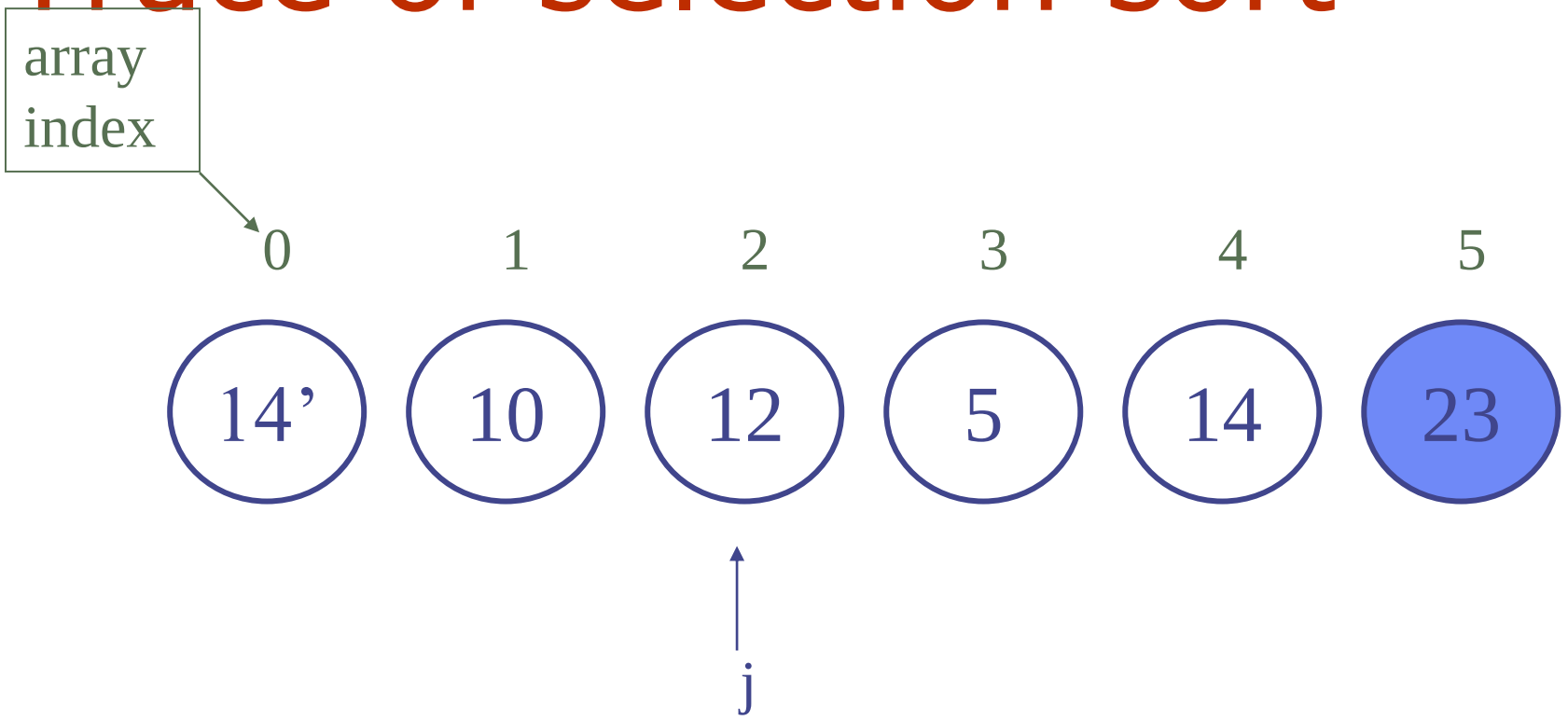
Trace of selection sort



i = 4, second iteration of the outer loop

j = 1, pos_greatest = 0

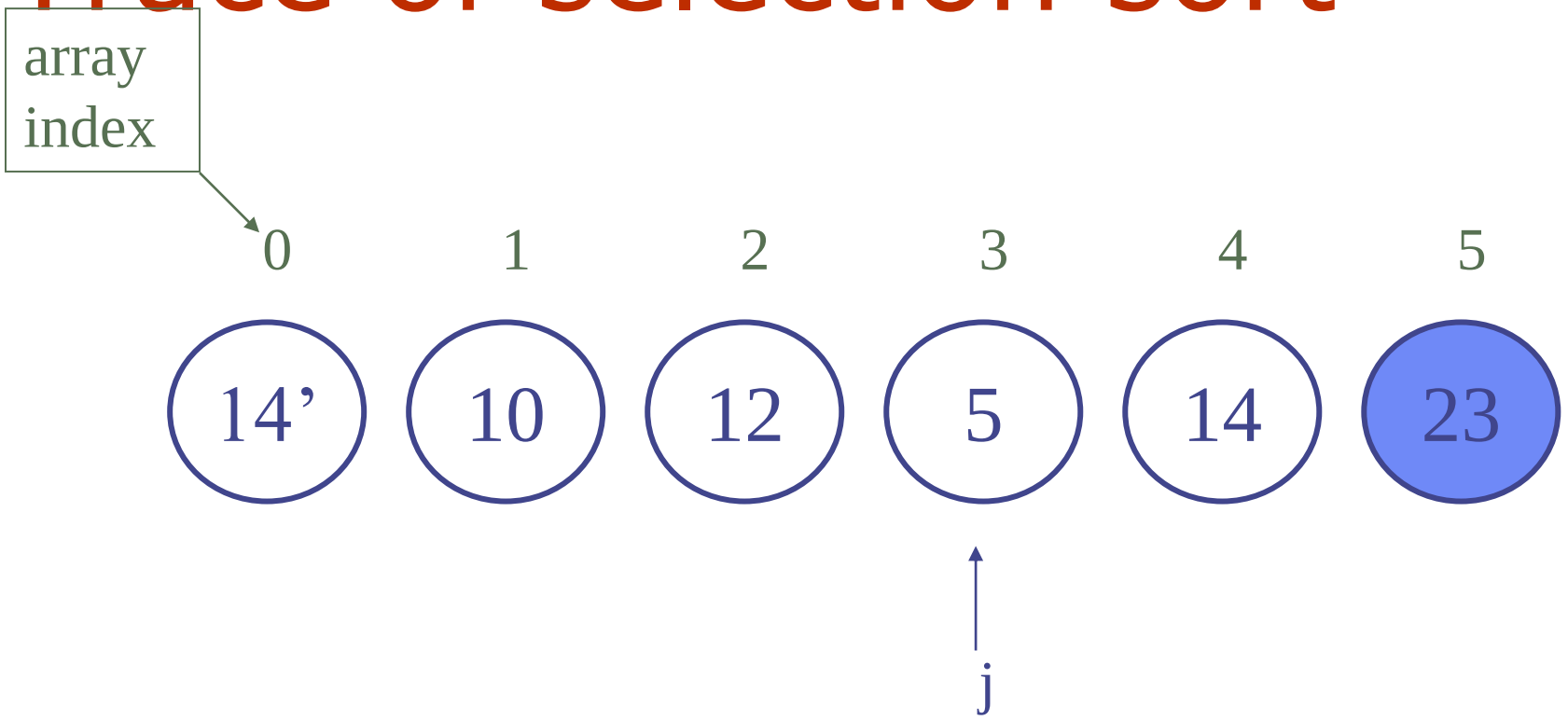
Trace of selection sort



i = 4, second iteration of the outer loop

j = 2, pos_greatest = 0

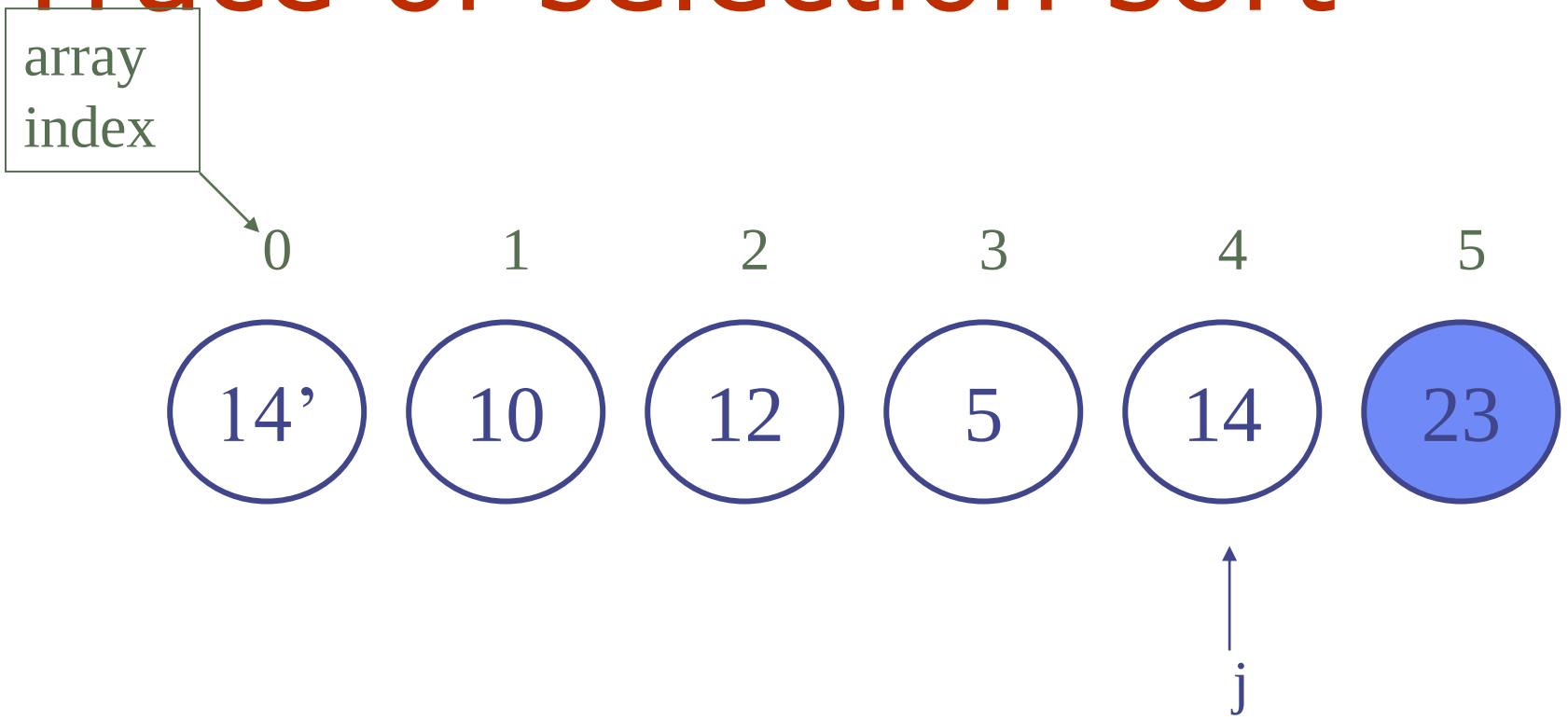
Trace of selection sort



i = 4, second iteration of the outer loop

j = 3, pos_greatest = 0

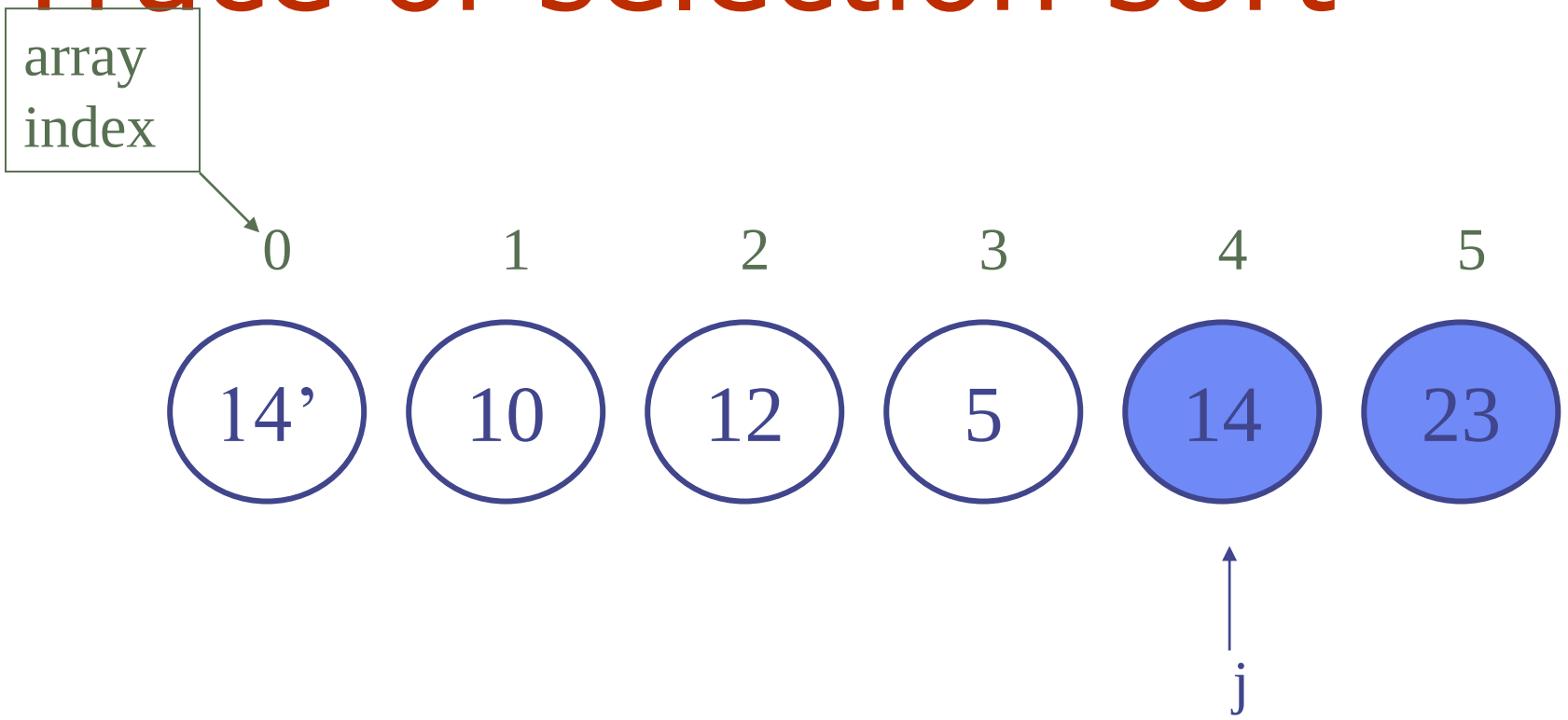
Trace of selection sort



i = 4, second iteration of the outer loop

j = 4, pos_greatest = 4 (because the “>=” in the comparison)

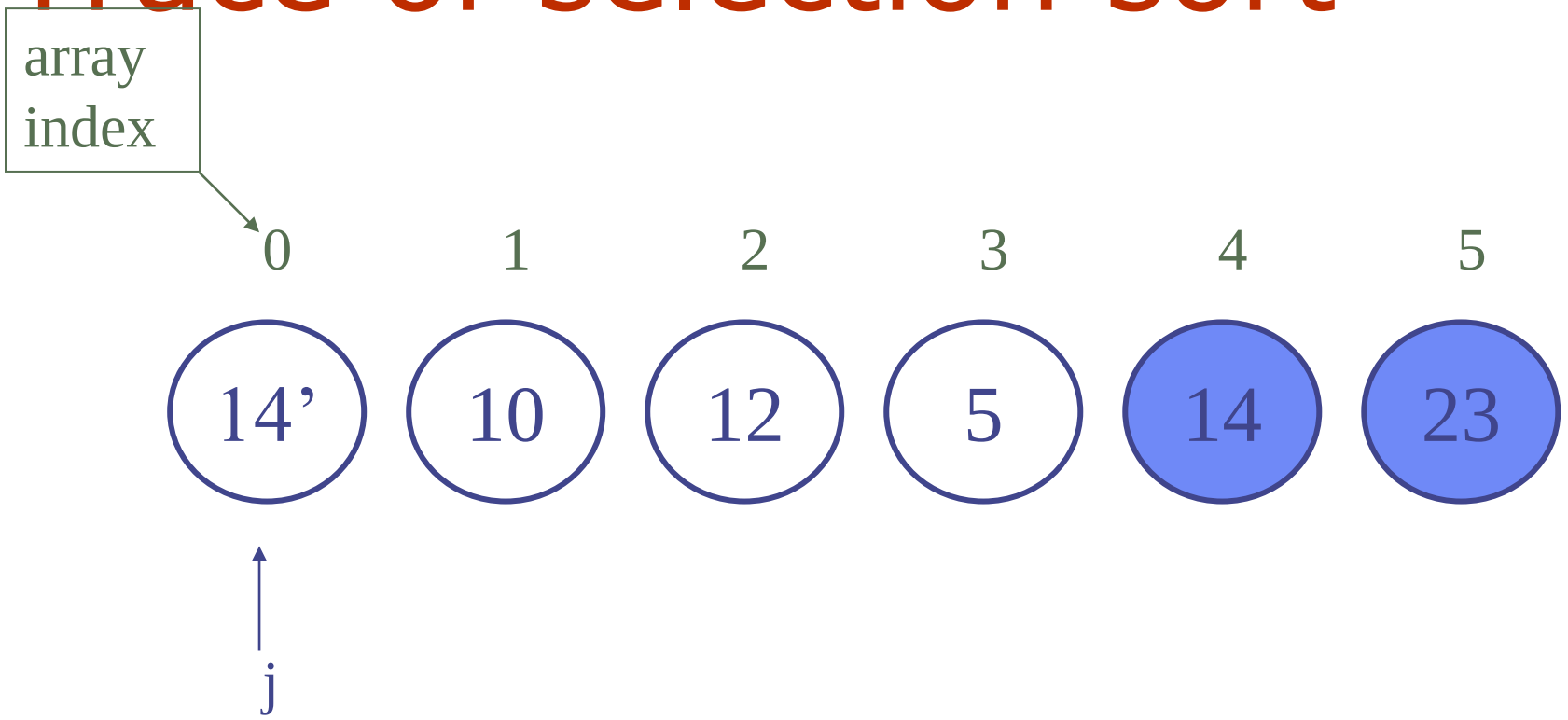
Trace of selection sort



i = 4, second iteration of the outer loop

No need to swap elements

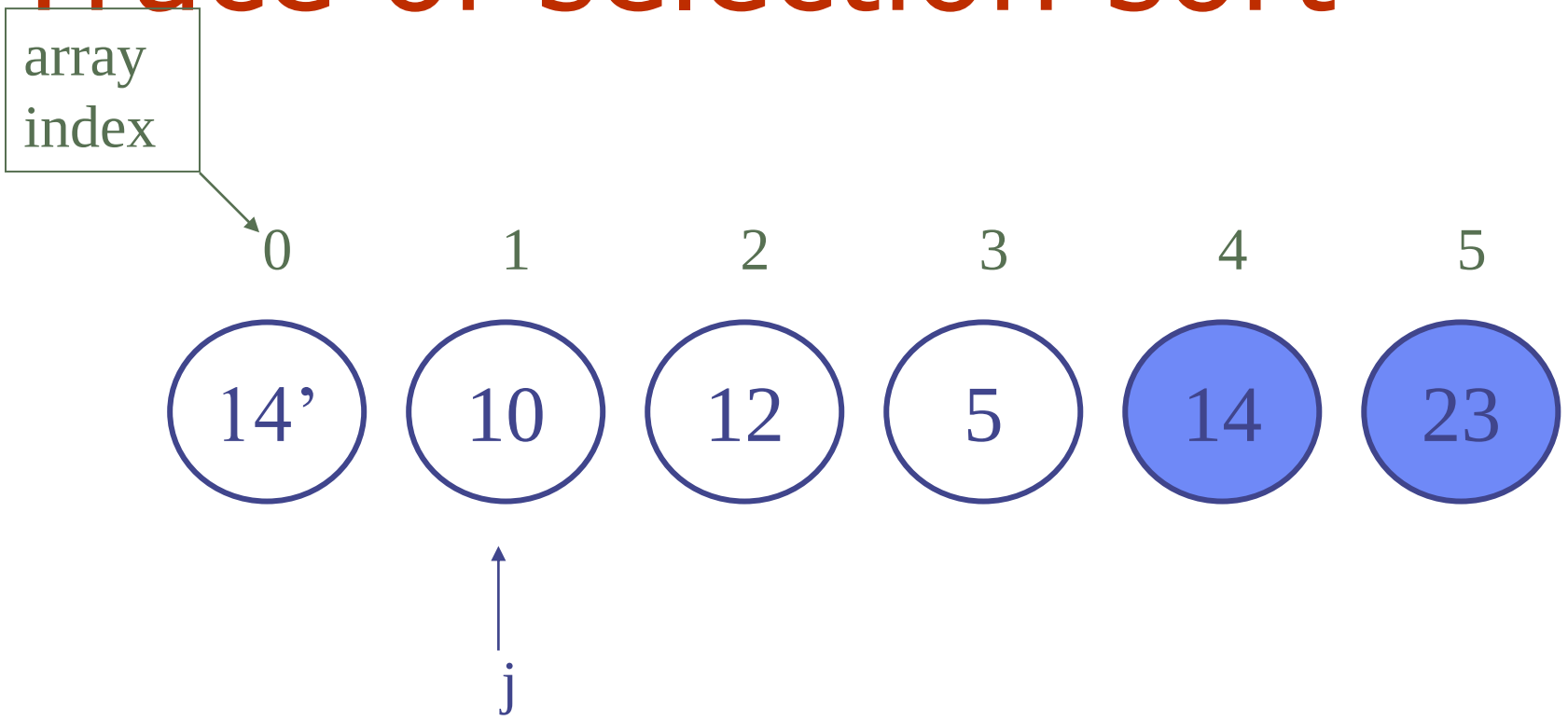
Trace of selection sort



i = 3, third iteration of the outer loop

j = 0, pos_greatest = 0

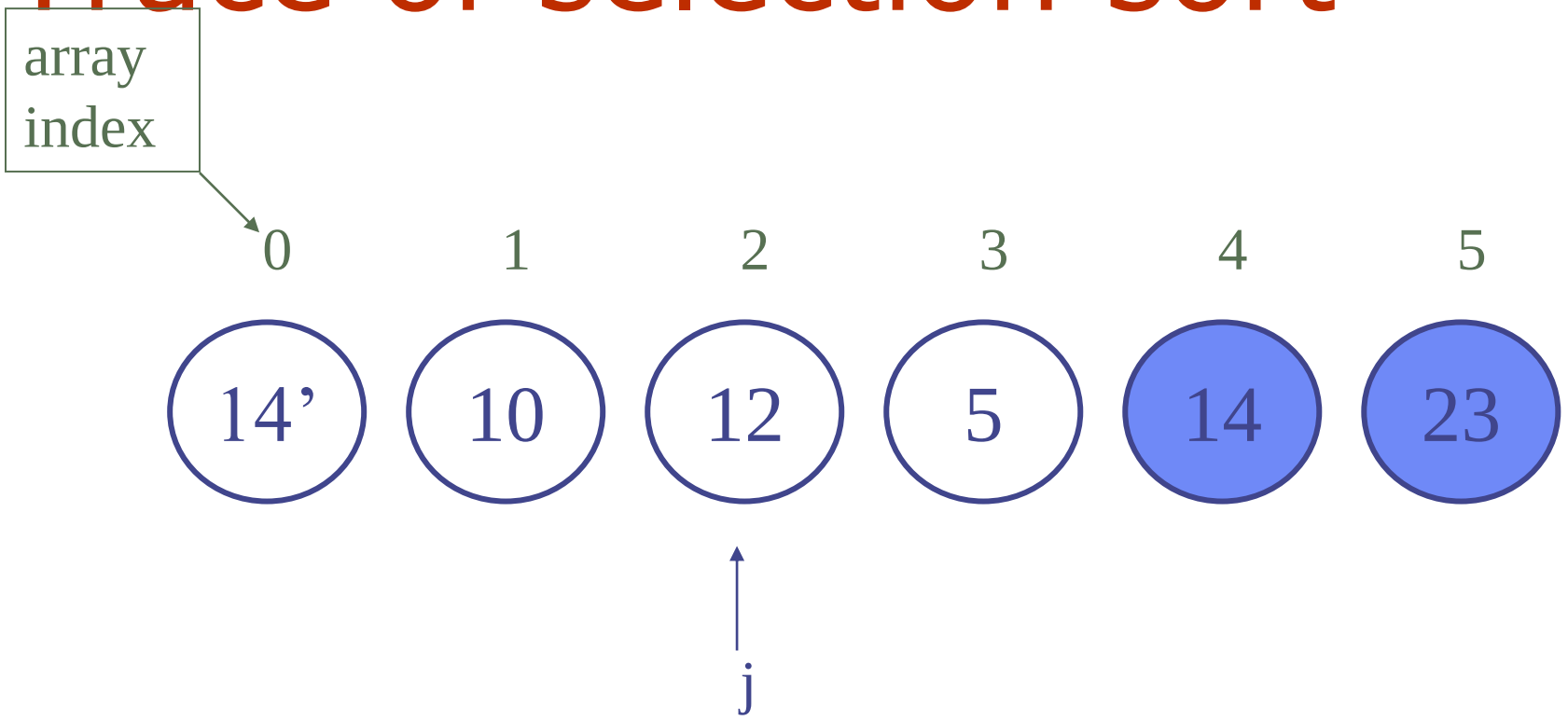
Trace of selection sort



i = 3, third iteration of the outer loop

j = 1, pos_greatest = 0

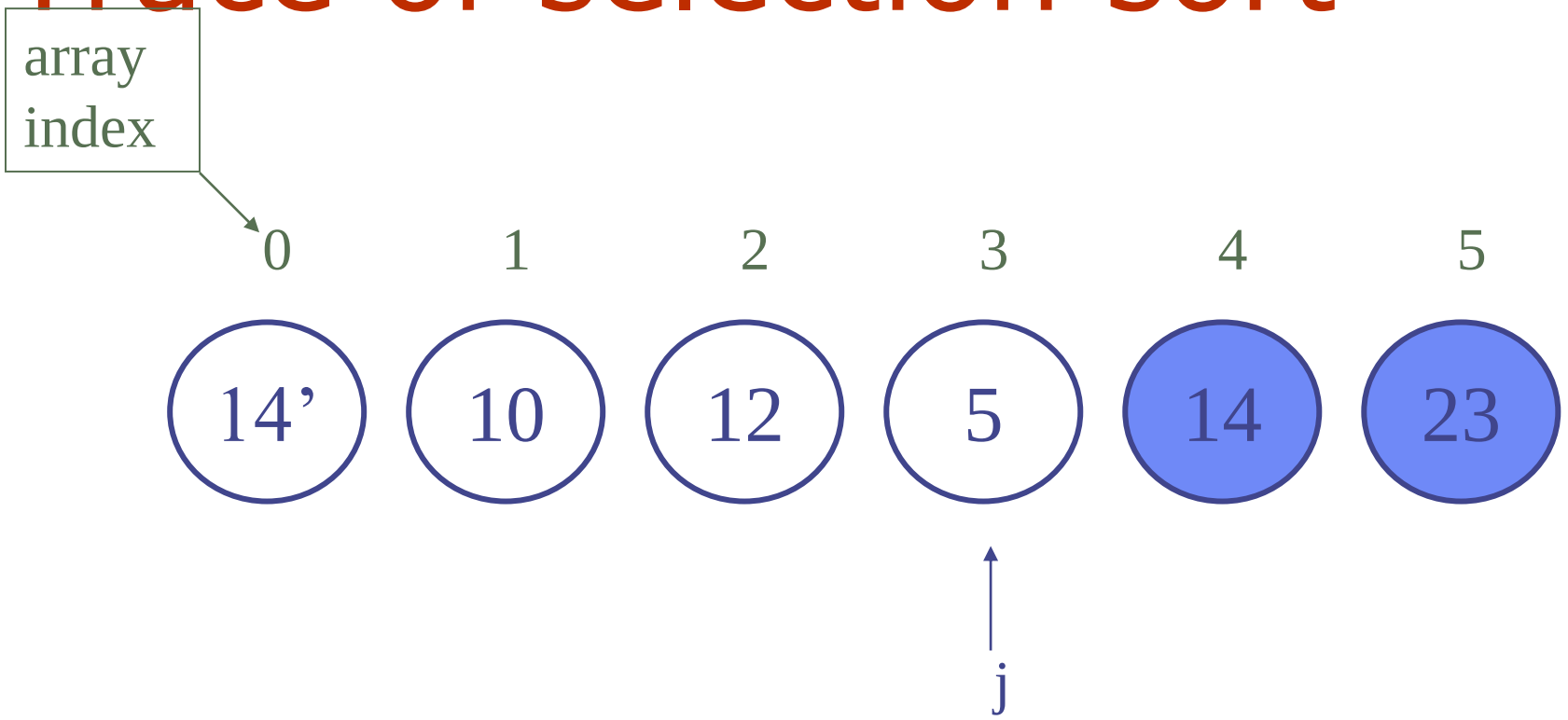
Trace of selection sort



i = 3, third iteration of the outer loop

j = 2, pos_greatest = 0

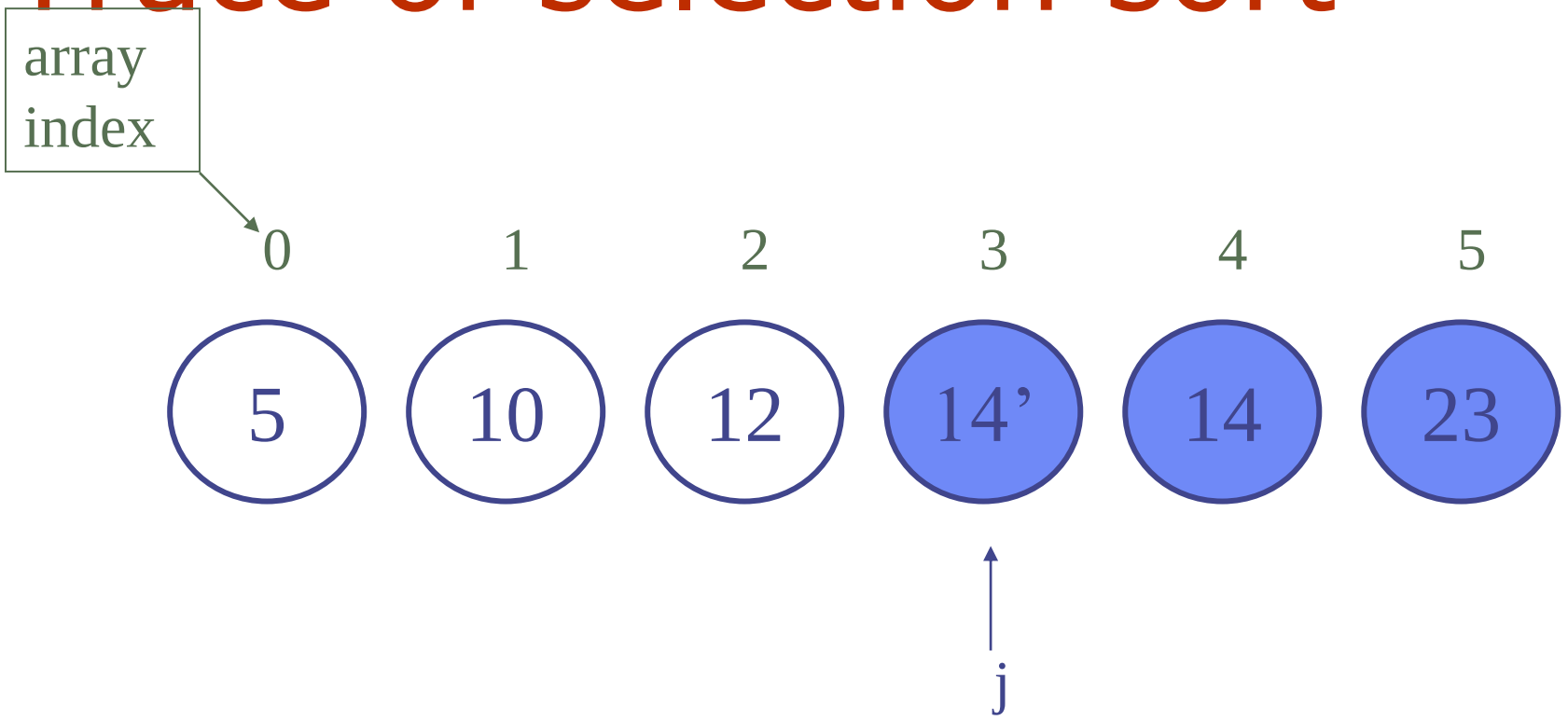
Trace of selection sort



i = 3, third iteration of the outer loop

j = 3, pos_greatest = 0

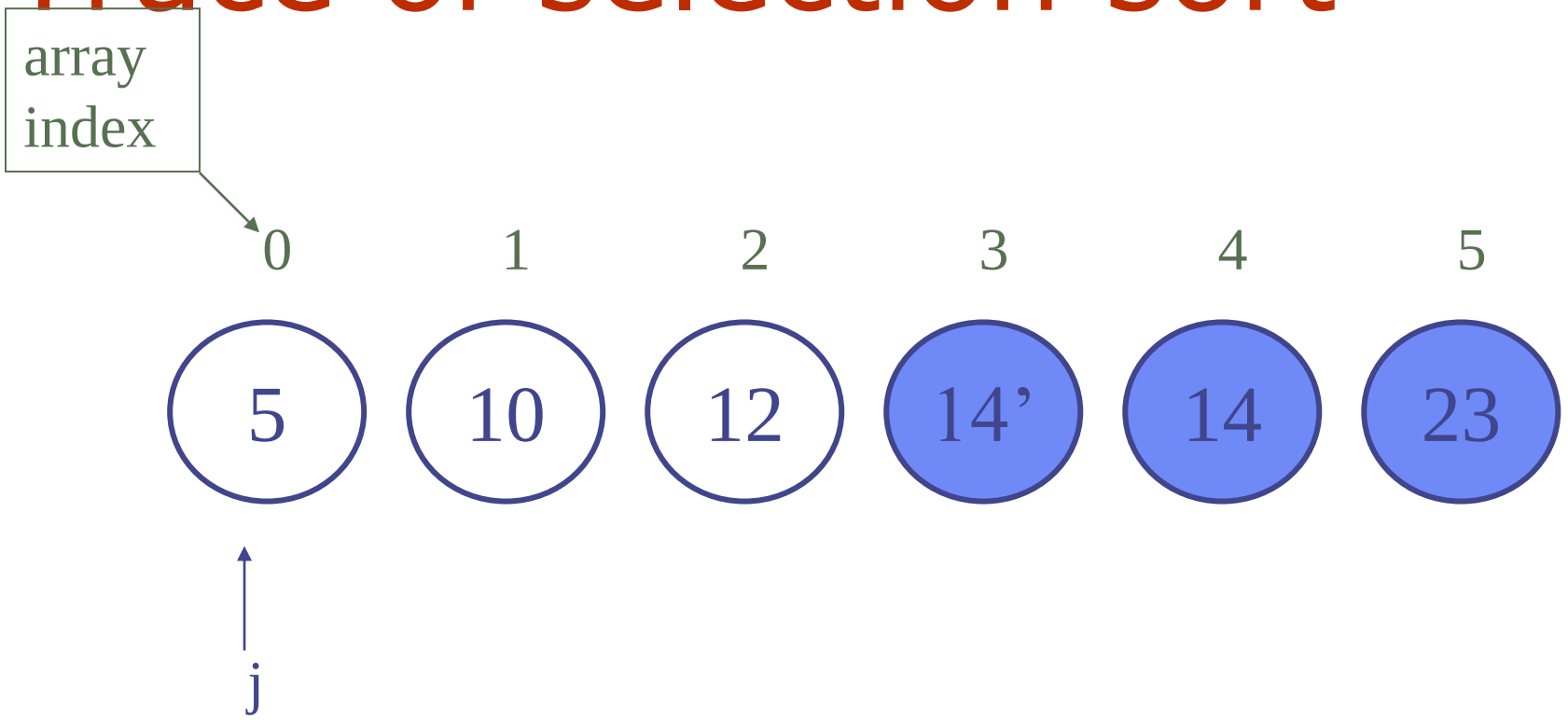
Trace of selection sort



i = 3, third iteration of the outer loop

swap elements at pos_greatest and 3

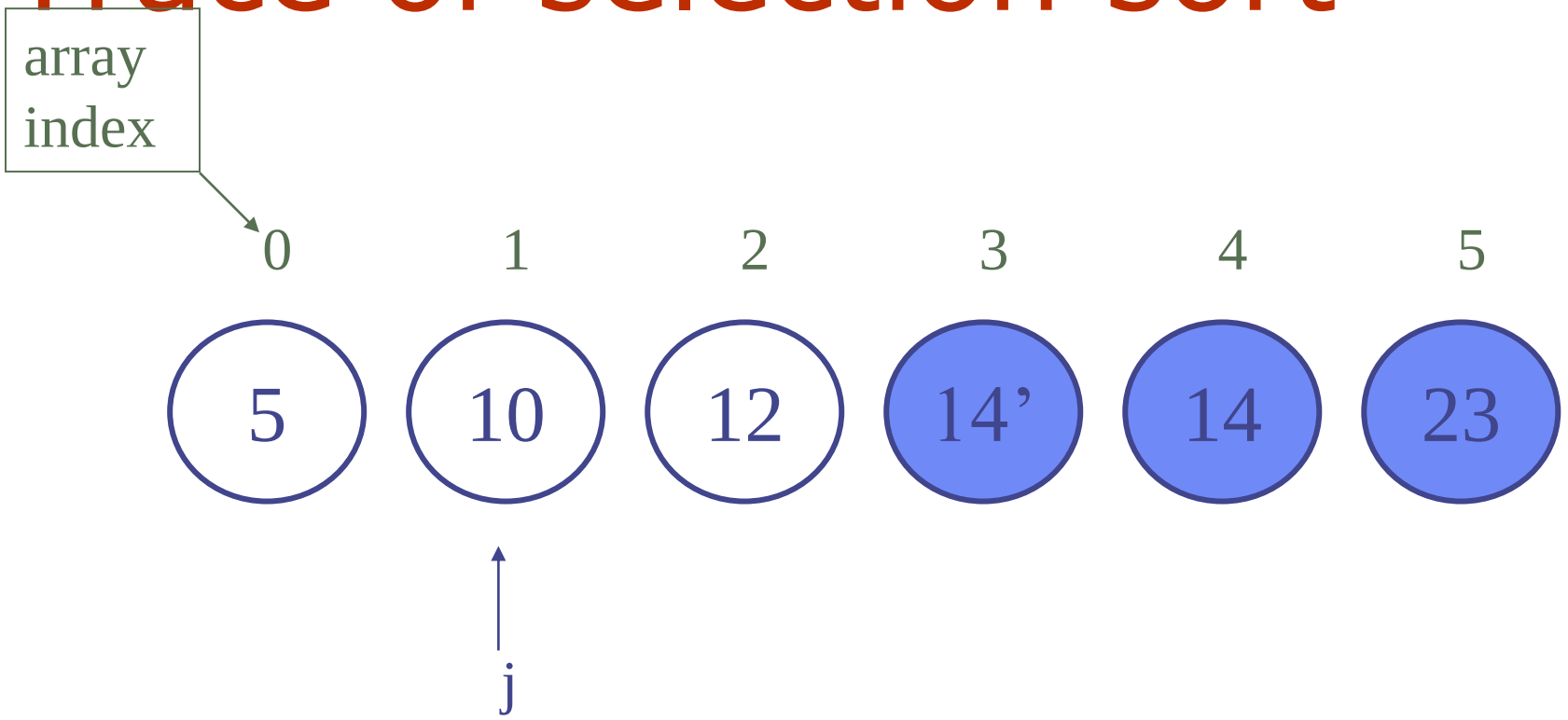
Trace of selection sort



i = 2, fourth iteration of the outer loop

j = 0, pos_greatest = 0

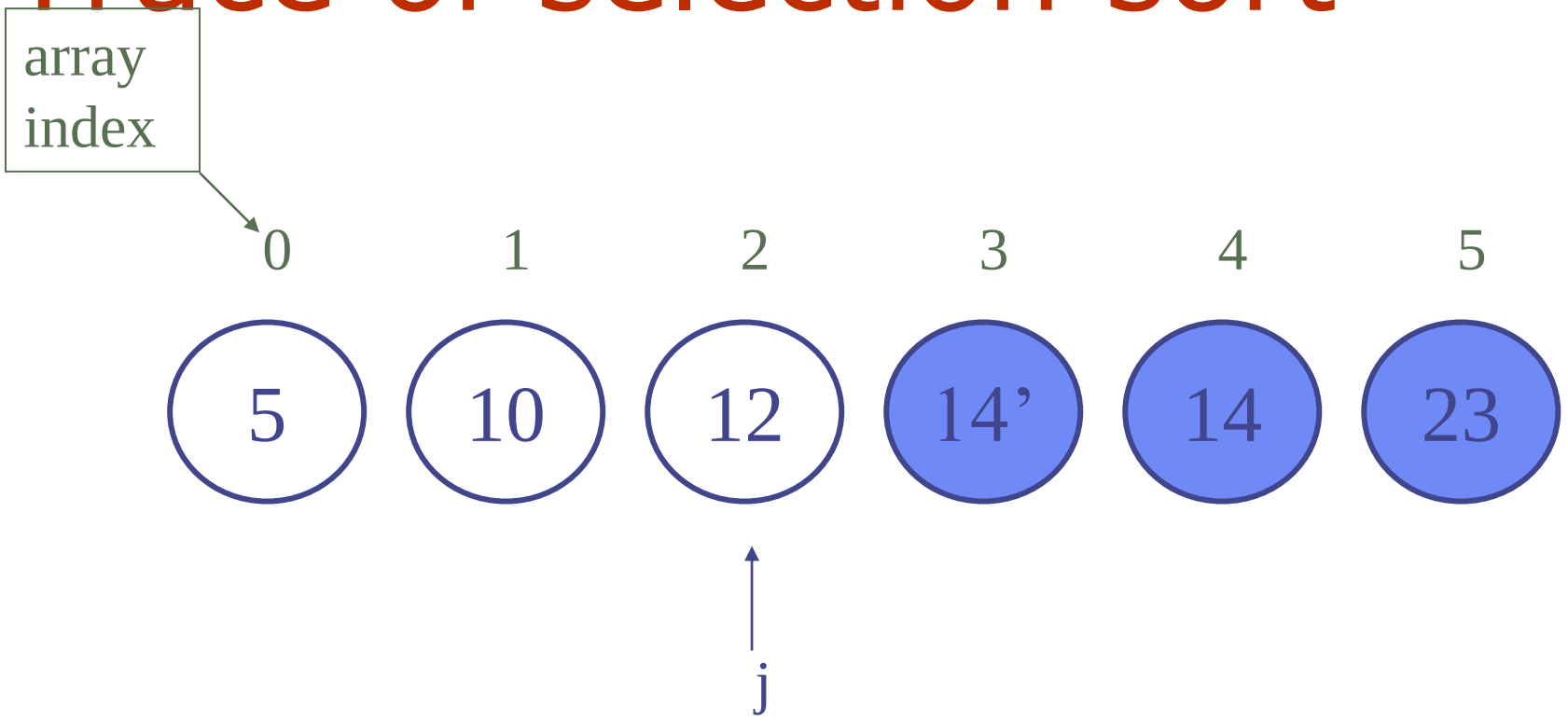
Trace of selection sort



i = 2, fourth iteration of the outer loop

j = 1, pos_greatest = 1 (changed!)

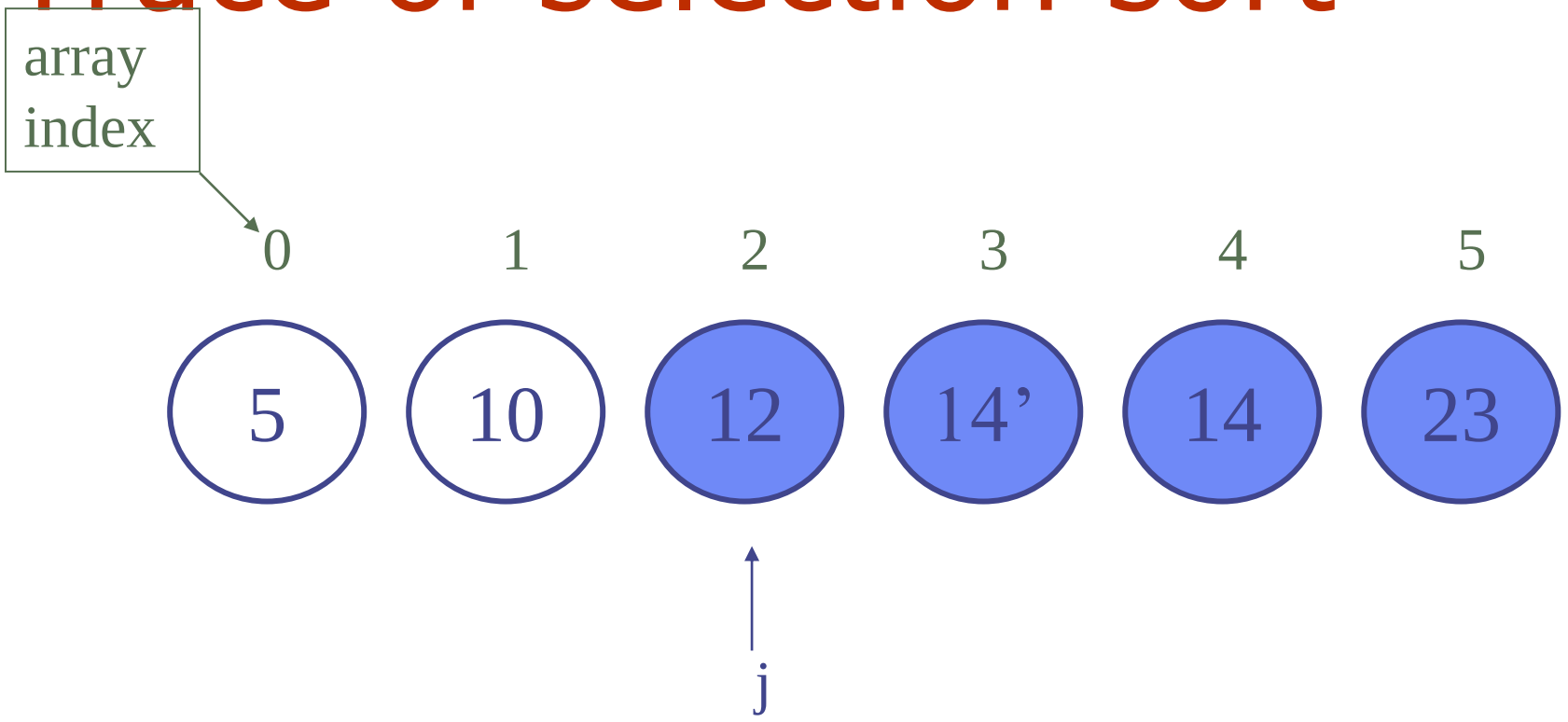
Trace of selection sort



i = 2, fourth iteration of the outer loop

j = 2, pos_greatest = 2 (changed again!)

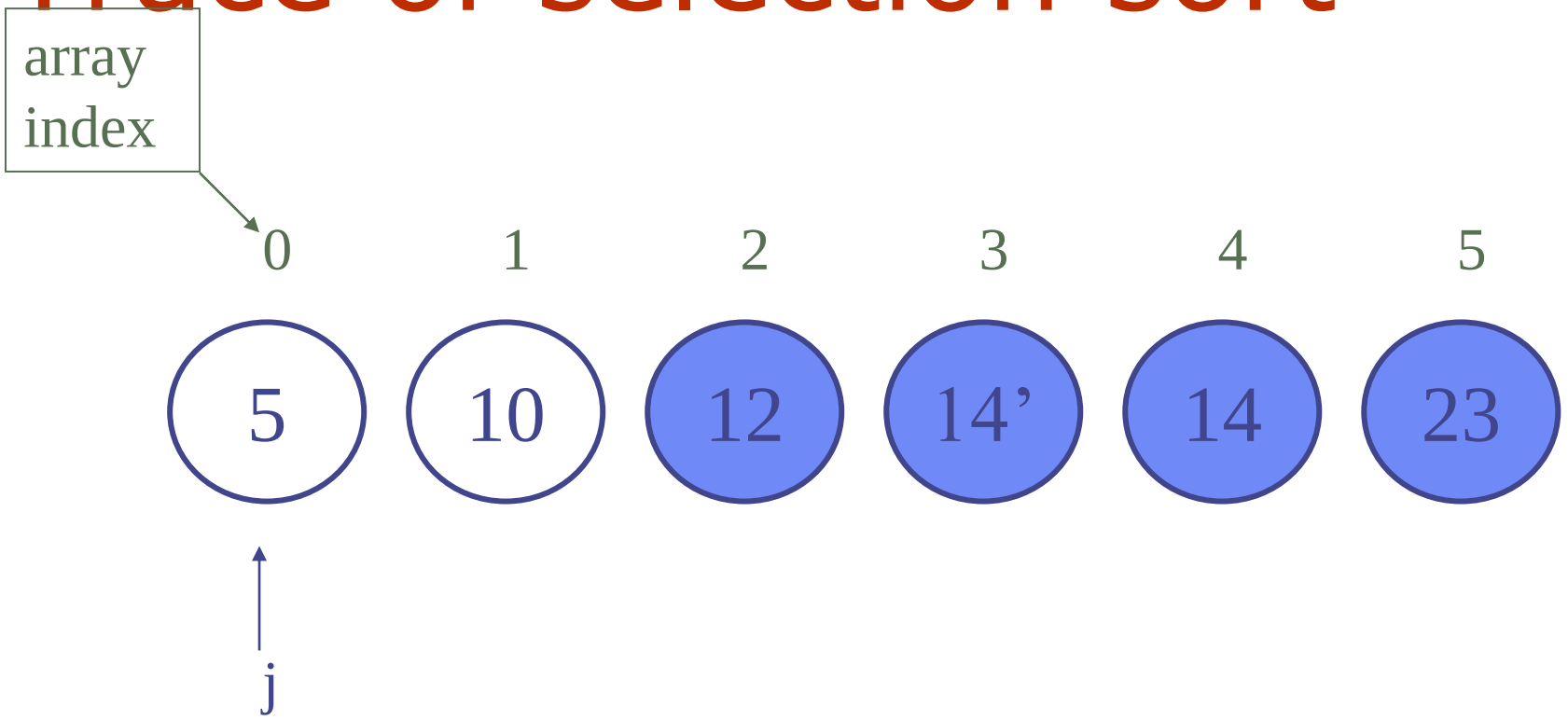
Trace of selection sort



i = 2, fourth iteration of the outer loop

swap elements at pos_greatest and 2 (element 12 with itself…)

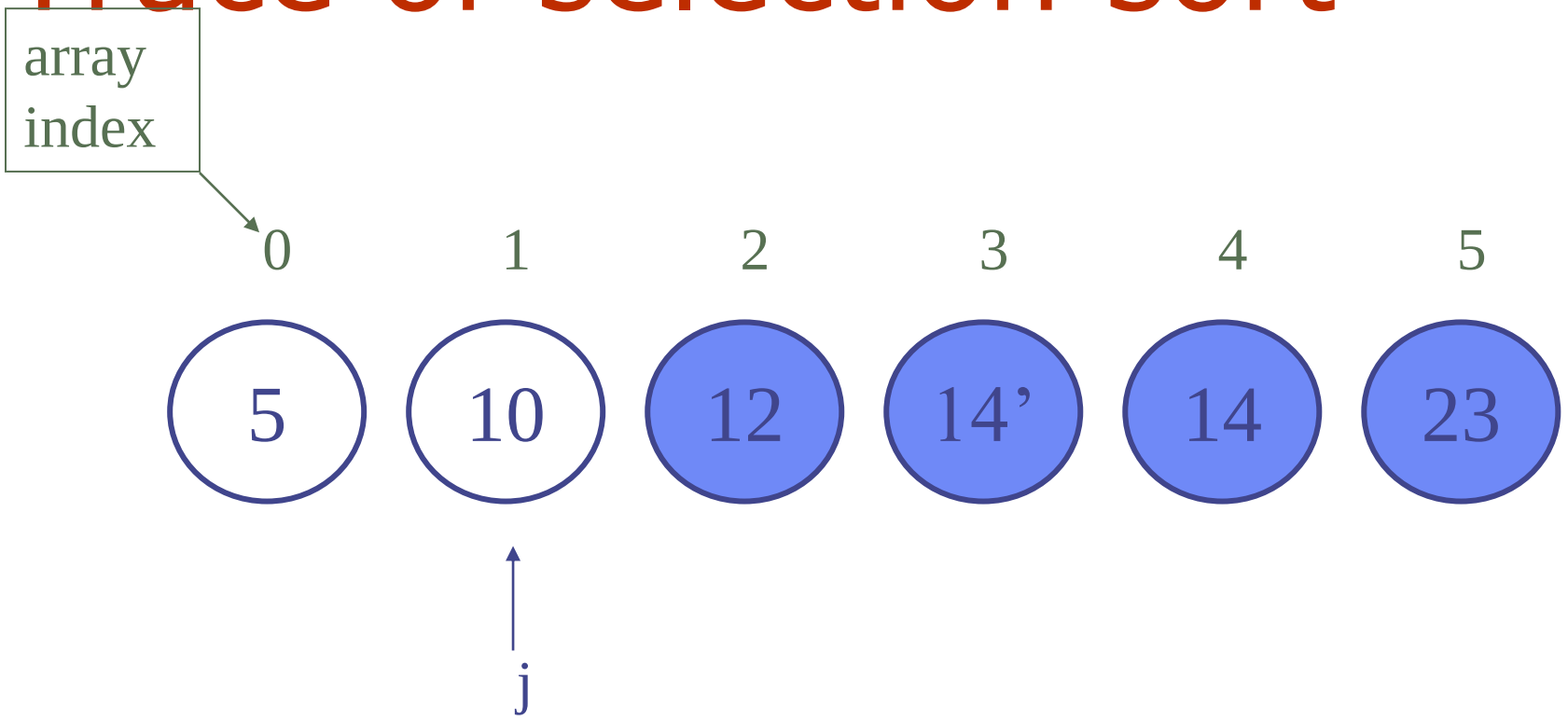
Trace of selection sort



i = 1, fifth iteration of the outer loop

j = 0, pos_greatest = 0

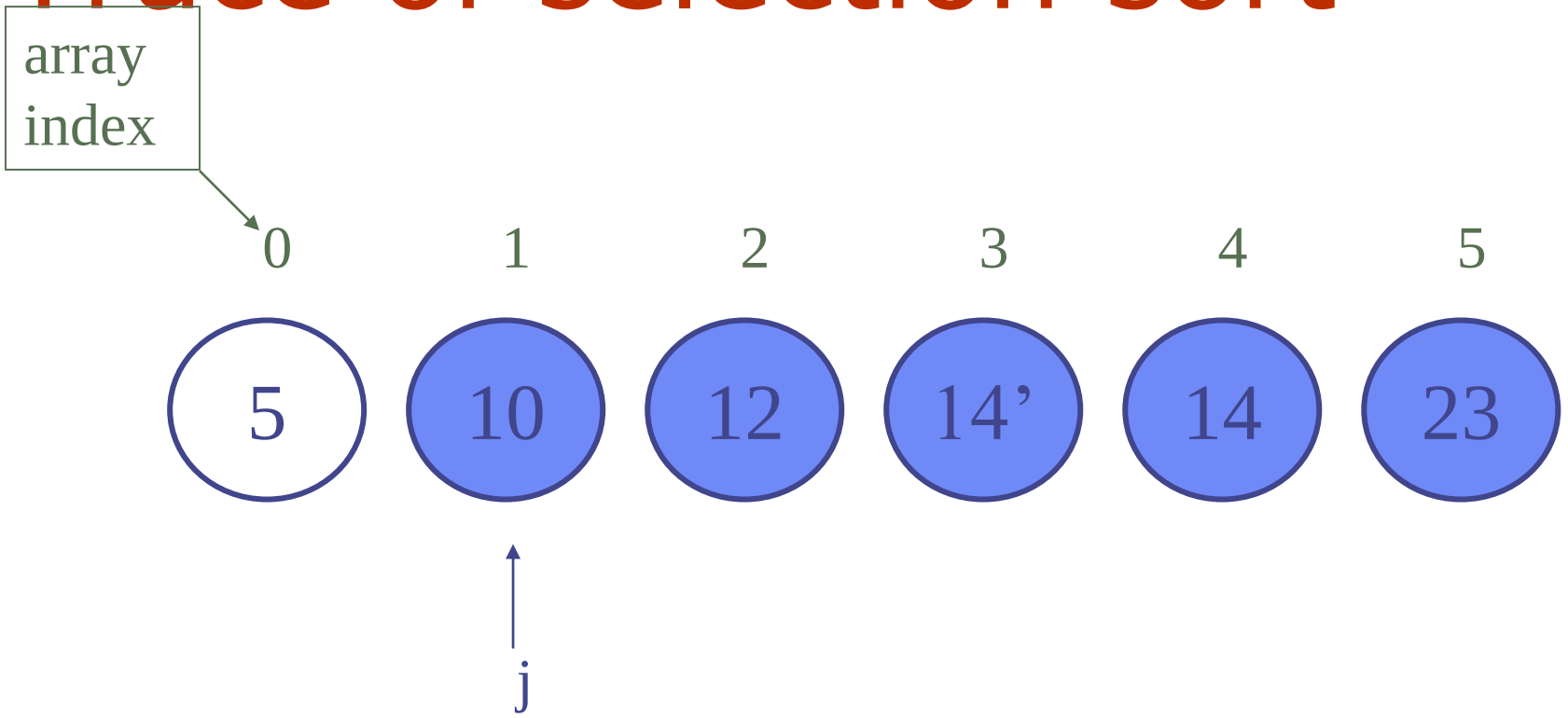
Trace of selection sort



i = 1, fifth iteration of the outer loop

j = 1, pos_greatest = 1 (changed)

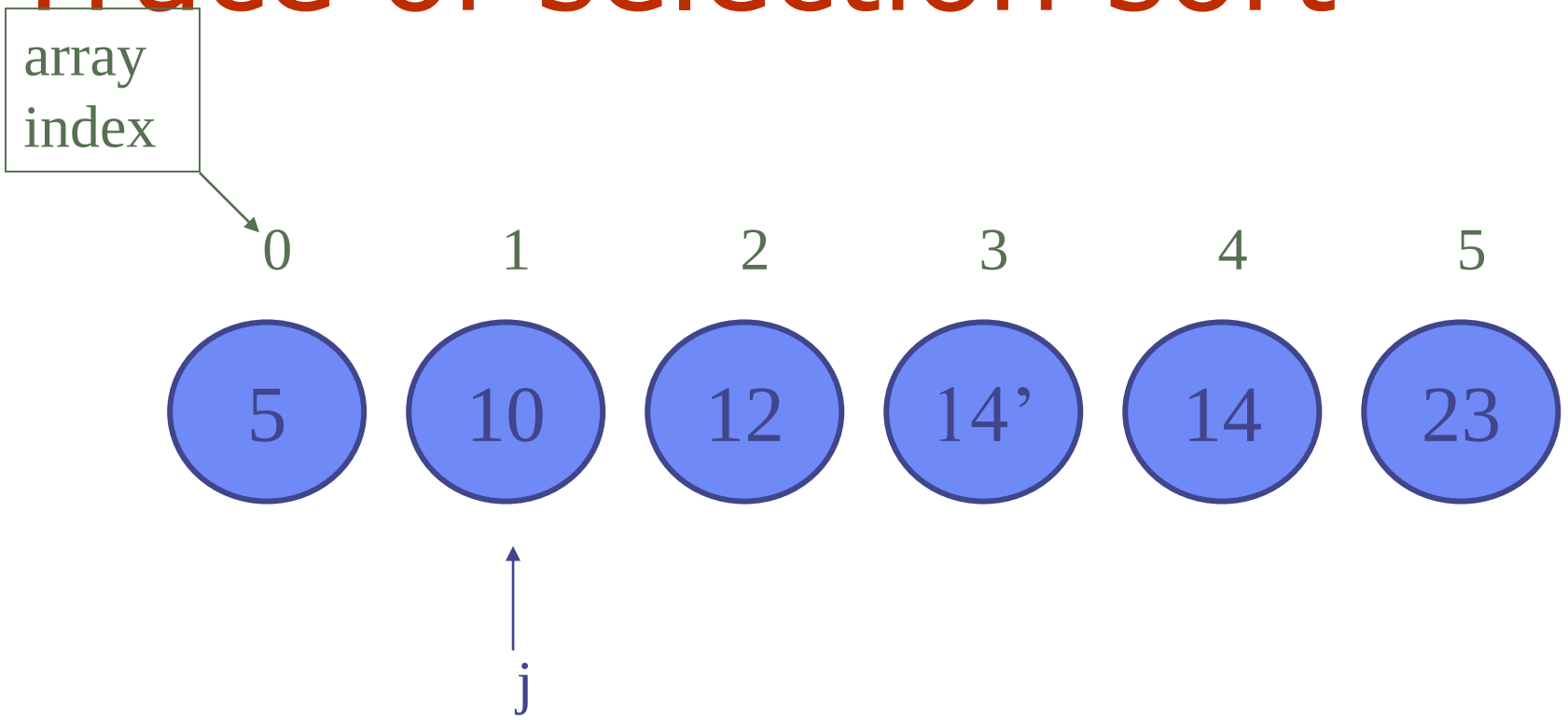
Trace of selection sort



i = 1, fifth iteration of the outer loop

swap element at pos_greatest with element at position 1 (10 with itself)

Trace of selection sort



i = 1, fifth iteration of the outer loop

NOTE: 14 and 14’ swapped ordering

This implementation of selection sort is not stable

Exercise (offline): can you make it stable or not?

Complexity of selection sort

Compared to bubble sort:

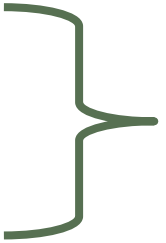
- Same number of iterations
- Same number of comparisons in the worst case
- fewer swaps
  (one for each outer loop = n-1)
- Hence, also $O(n^2)$

Insertion Sort: Basic Idea

- **Keep the front of the list sorted**, and as we move through the back, elements we insert them into the correct place in the front

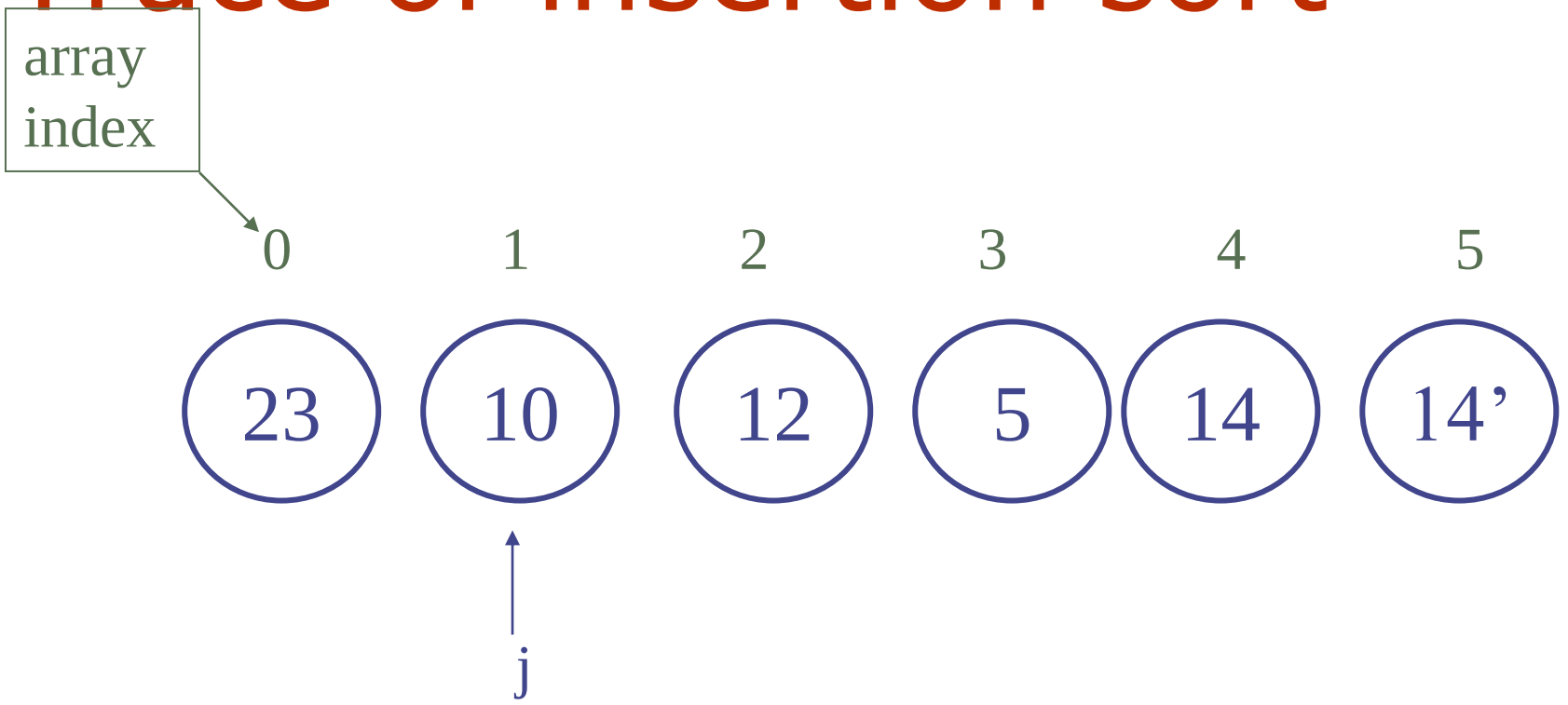
Insertion sort

```
void insertionSort(int arr[]){
  for(int j=1; j < arr.length; j++){
    int temp = arr[j];
    int i = j; // range 0 to j-1 is sorted
    while(i > 0 && arr[i-1] > temp){
      arr[i] = arr[i-1];
      i--;
    }
    arr[i] = temp;
  } // end outer for loop
}  // end insertion sort
```



Find a place to insert temp in the sorted range; as you are looking, shift elements in the sorted range to the right

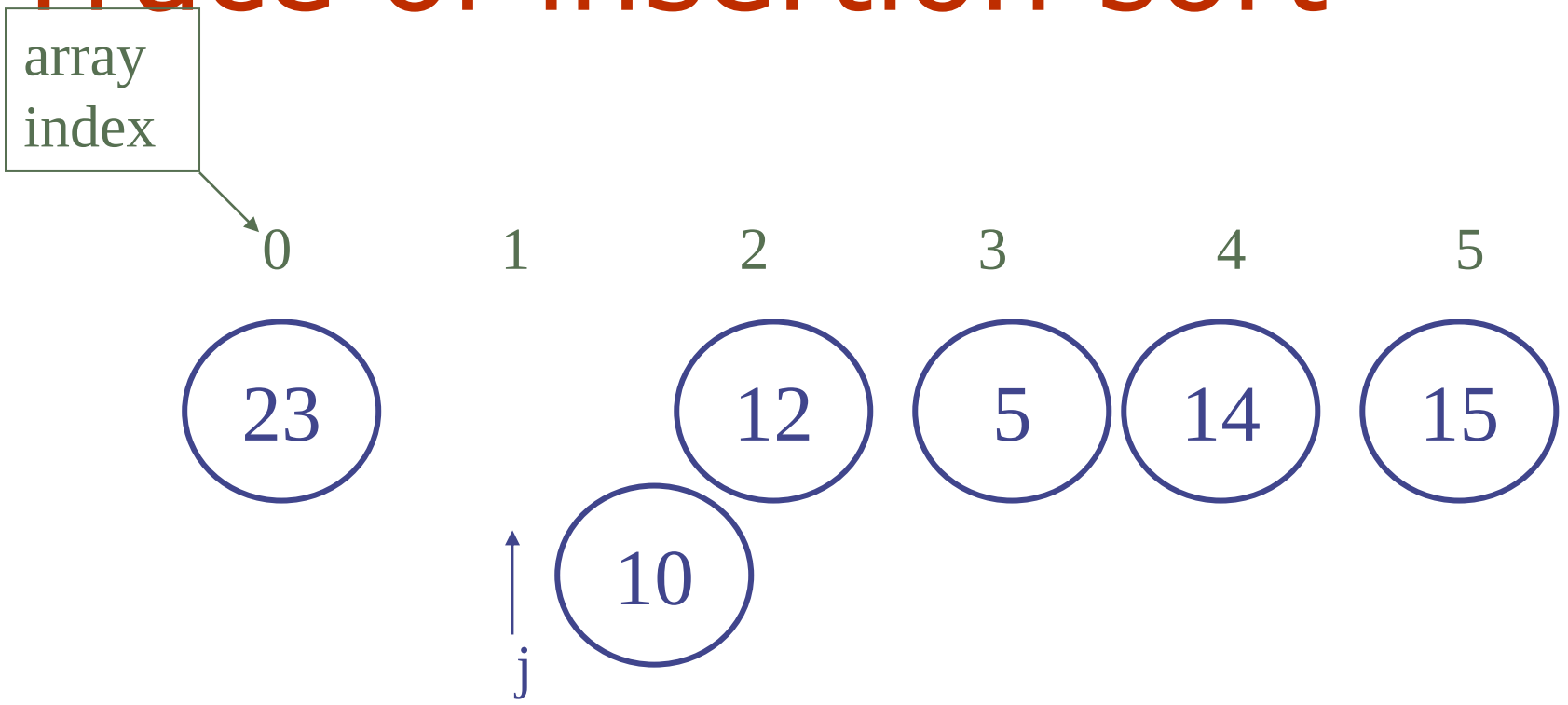
Trace of insertion sort



j = 1, first iteration of the outer loop

temp = 10; i = 1; arr[i-1] >= 10

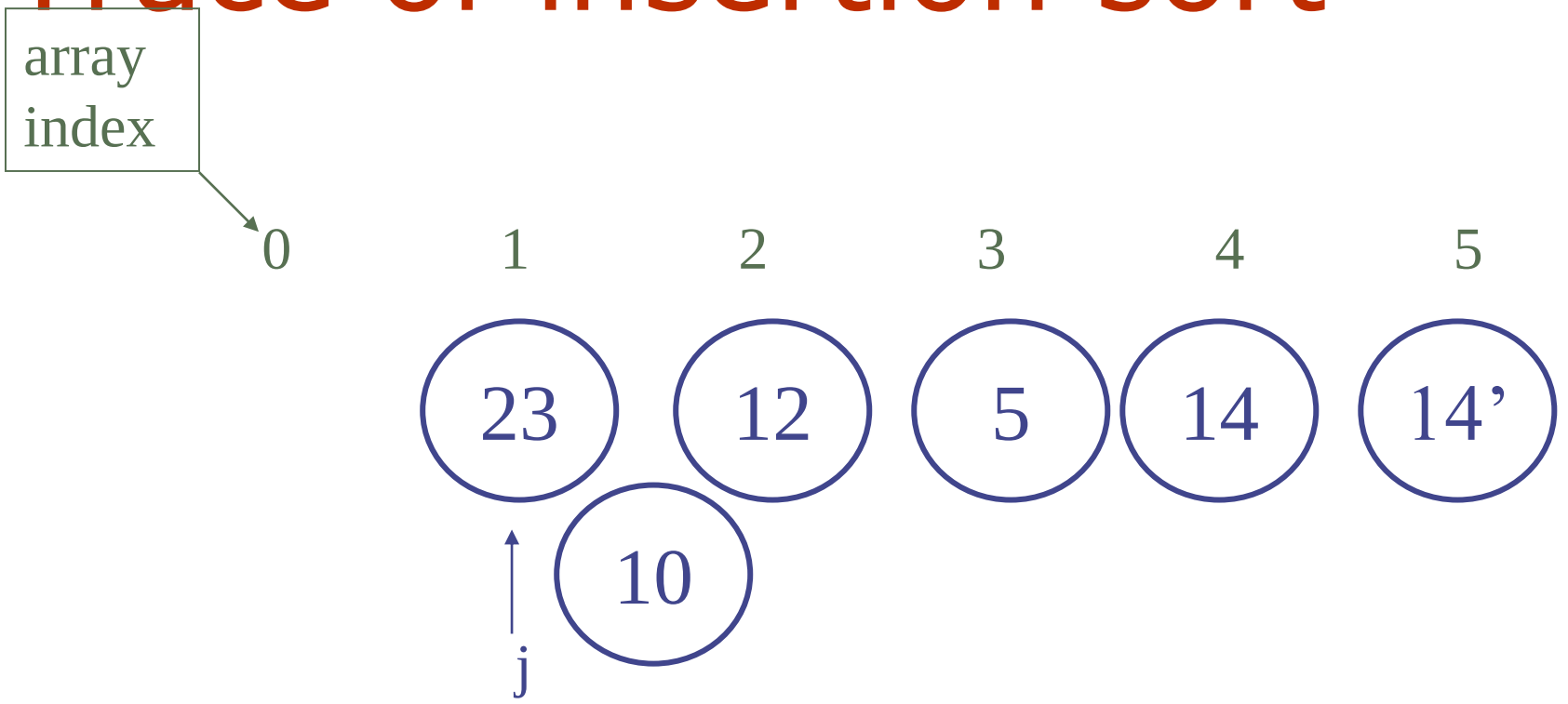
Trace of insertion sort



j = 1, first iteration of the outer loop

temp = 10; i = 1; arr[i-1] >= 10

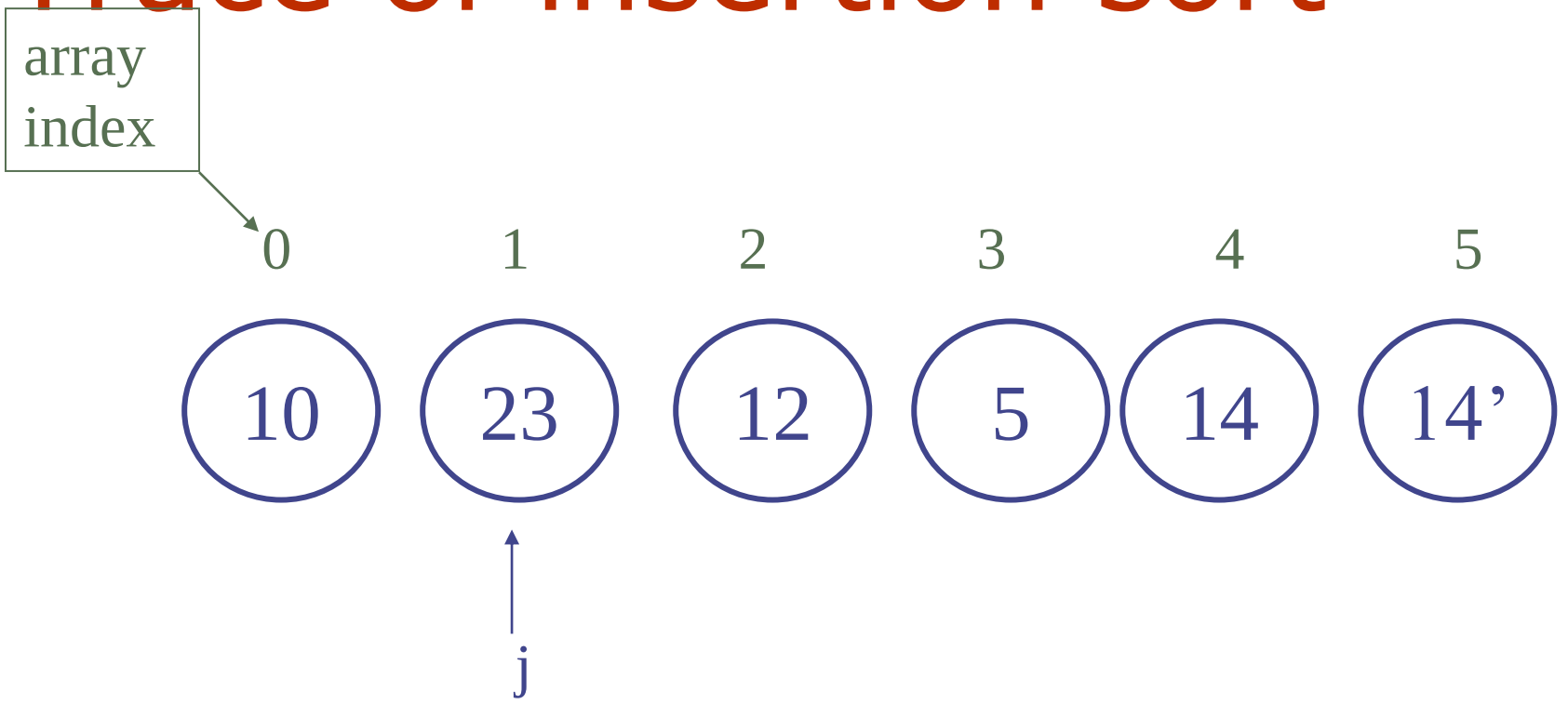
Trace of insertion sort



j = 1, first iteration of the outer loop

arr[i] = arr[i-1]

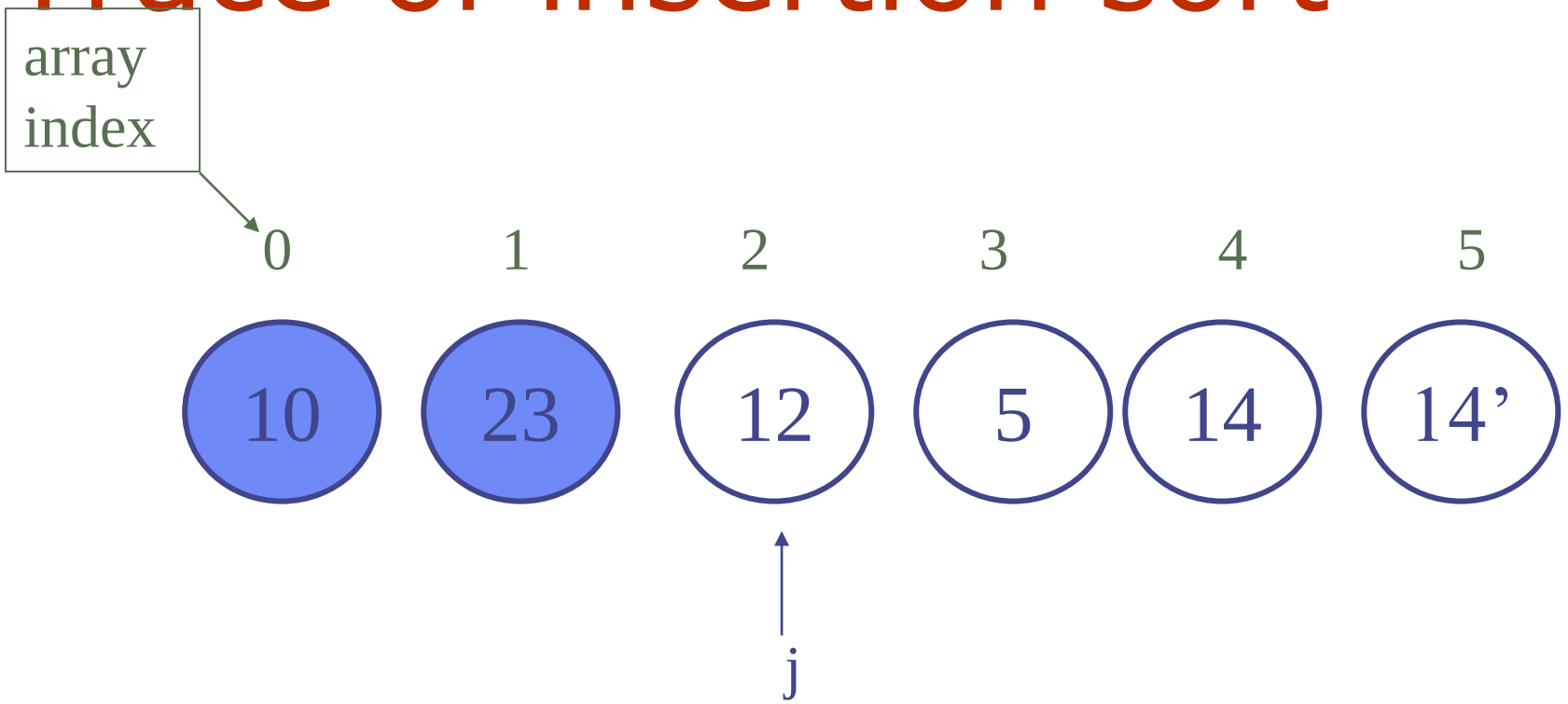
Trace of insertion sort



j = 1, first iteration of the outer loop

arr[i] = temp

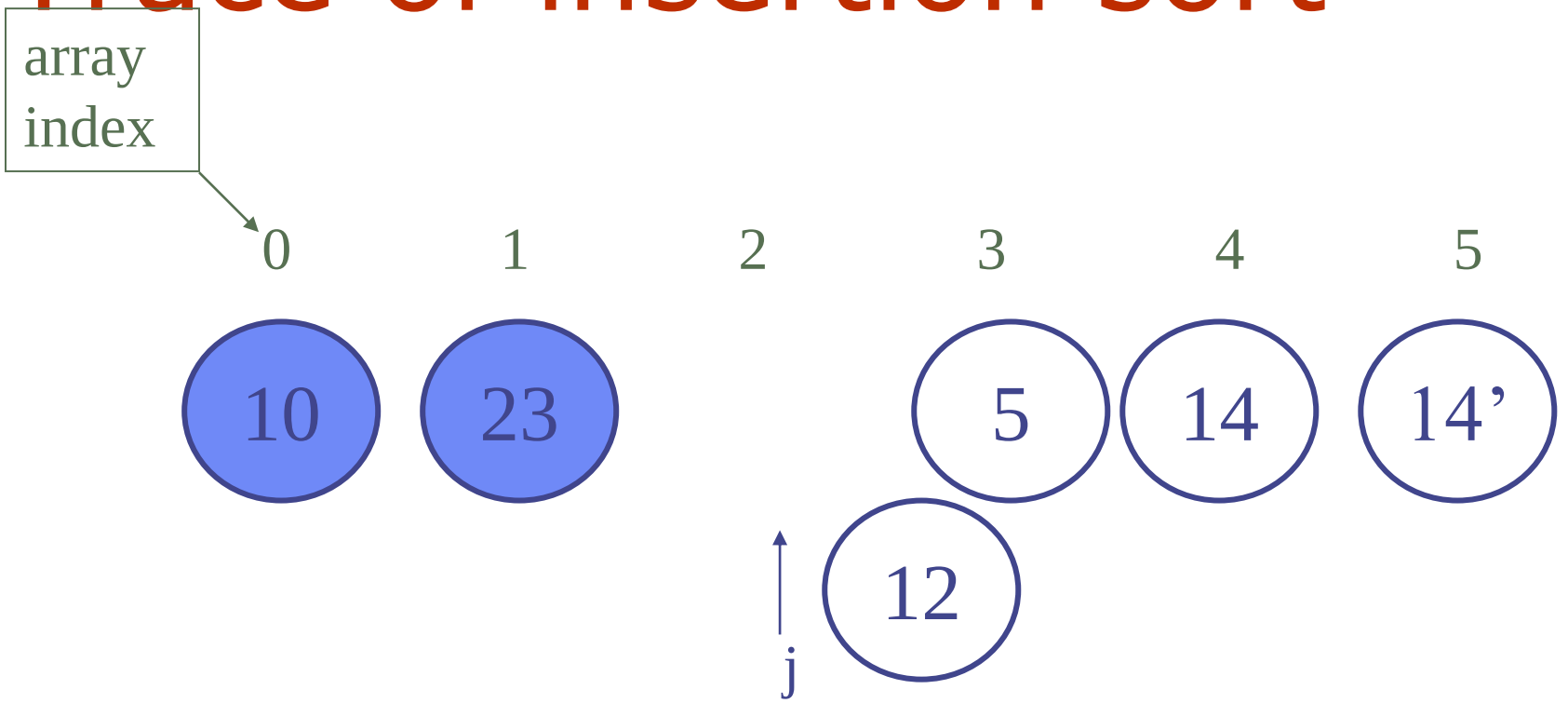
Trace of insertion sort



j = 2, second iteration of the outer loop

temp = 12; arr[i-1] >= temp

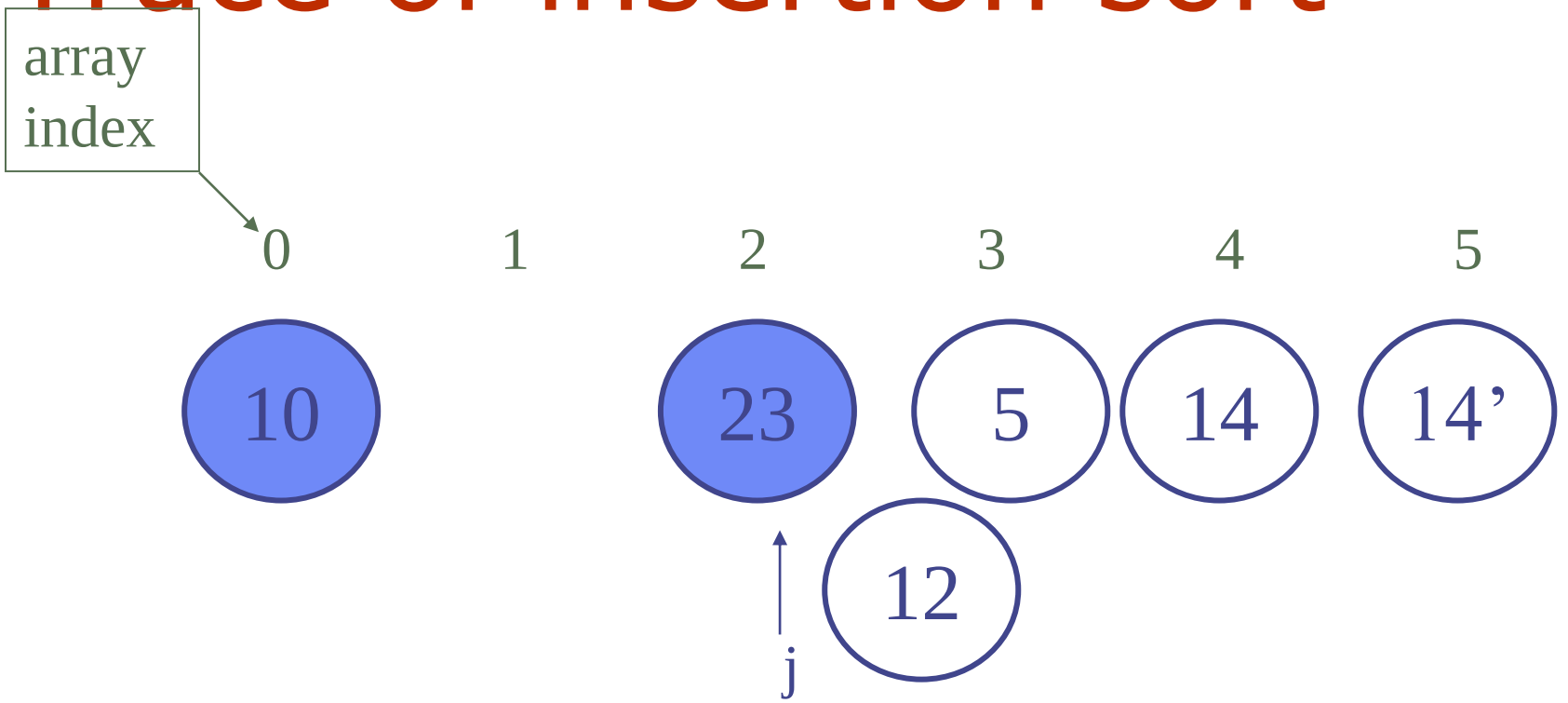
Trace of insertion sort



j = 2, second iteration of the outer loop

temp = 12; arr[i-1] >= temp

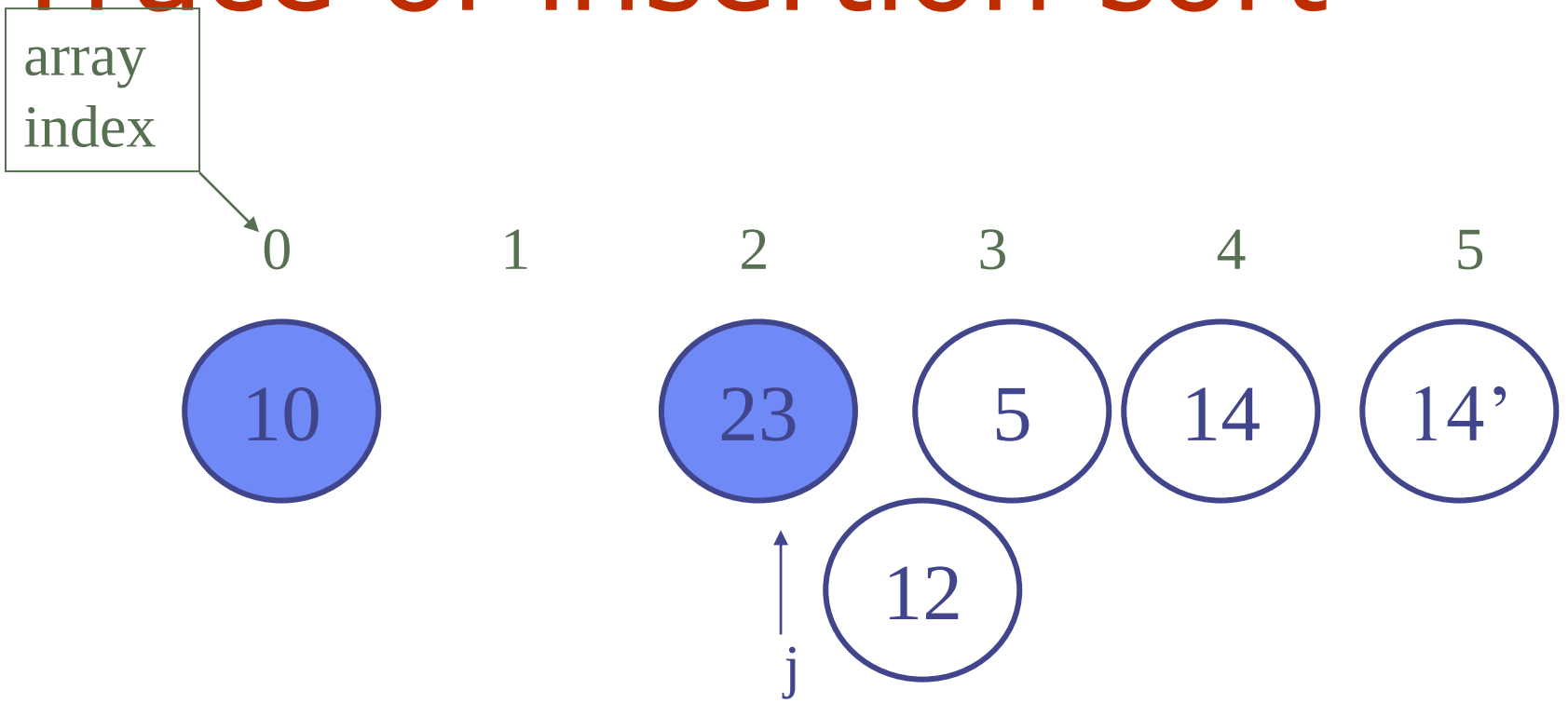
Trace of insertion sort



j = 2, second iteration of the outer loop

arr[i-1] = arr[i]

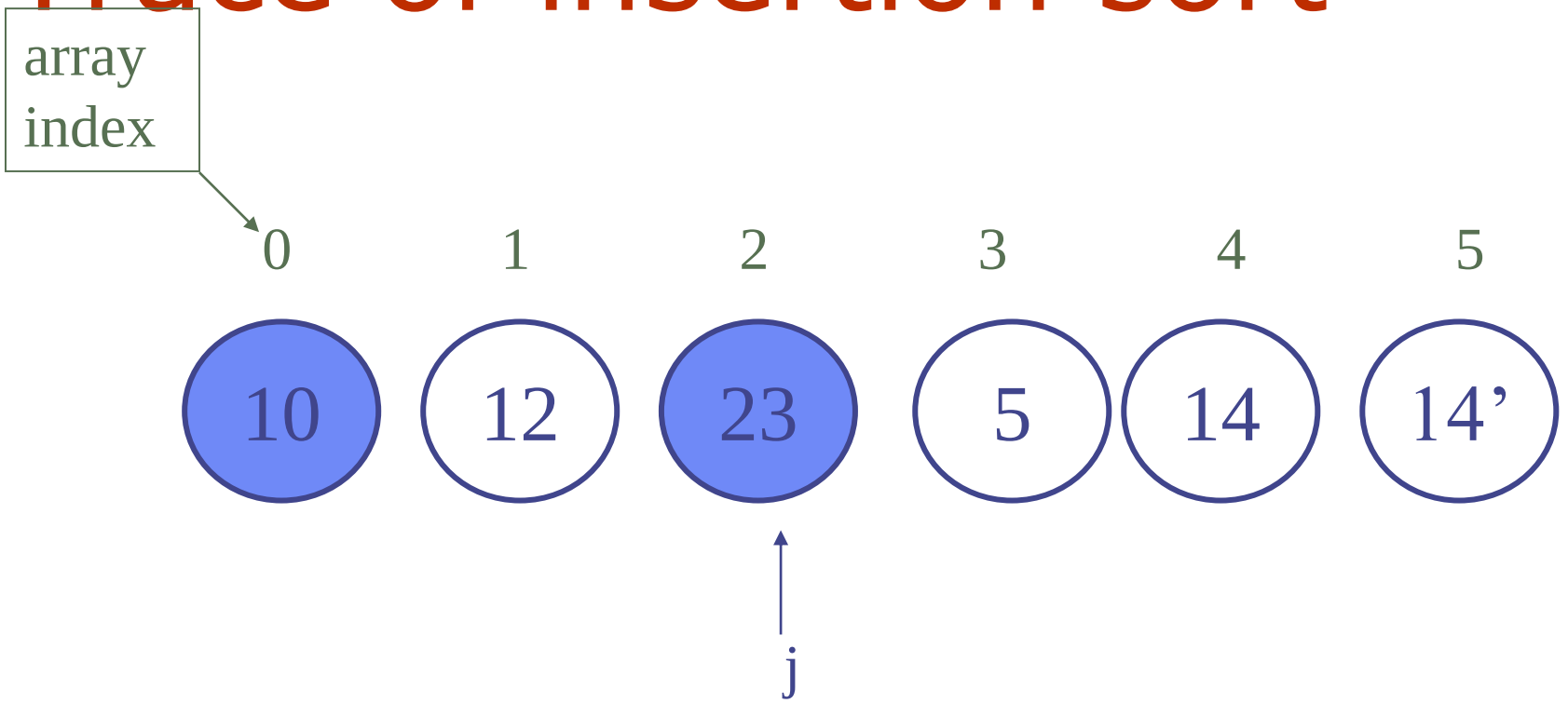
Trace of insertion sort



j = 2, second iteration of the outer loop

arr[i-1] < temp

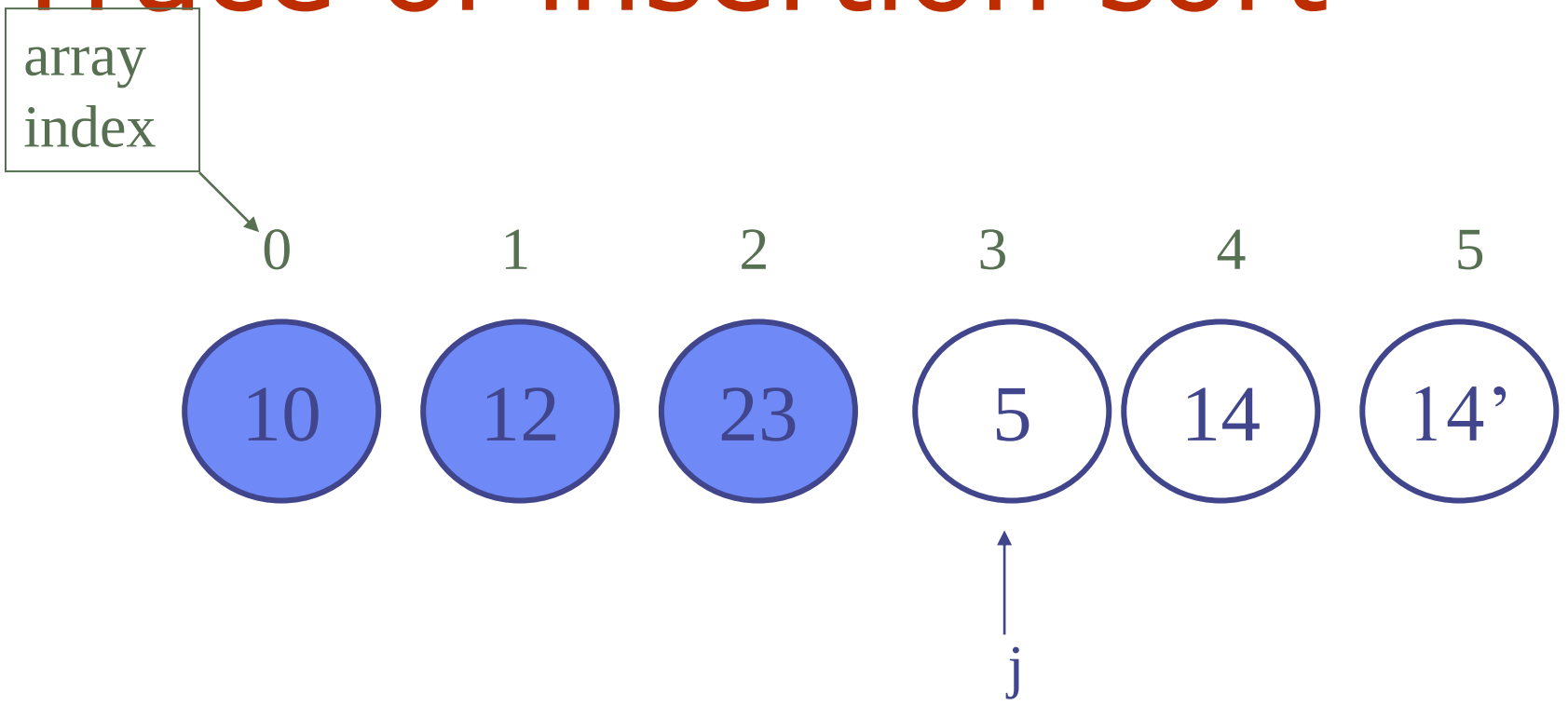
Trace of insertion sort



j = 2, second iteration of the outer loop

arr[i-1] = temp

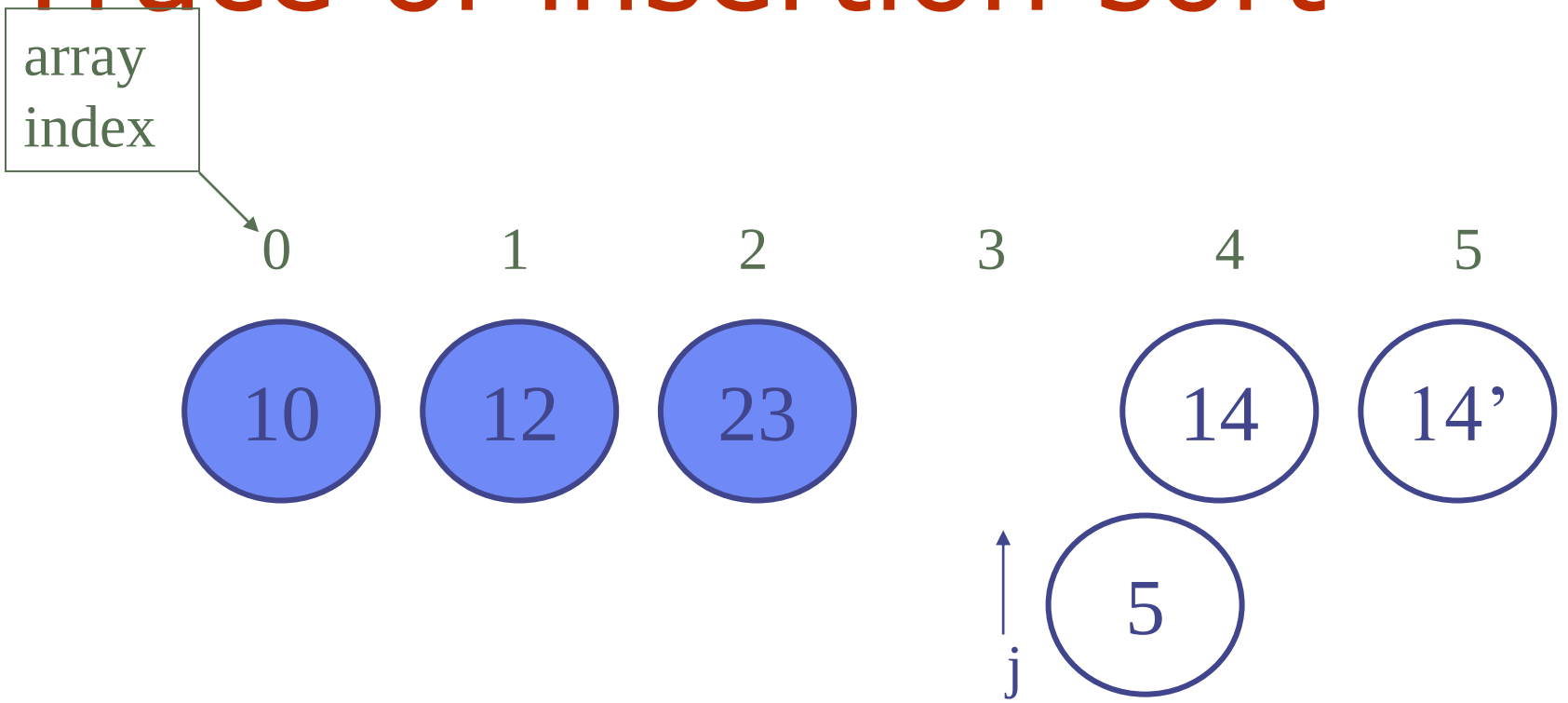
Trace of insertion sort



j = 3, third iteration of the outer loop

temp = 5

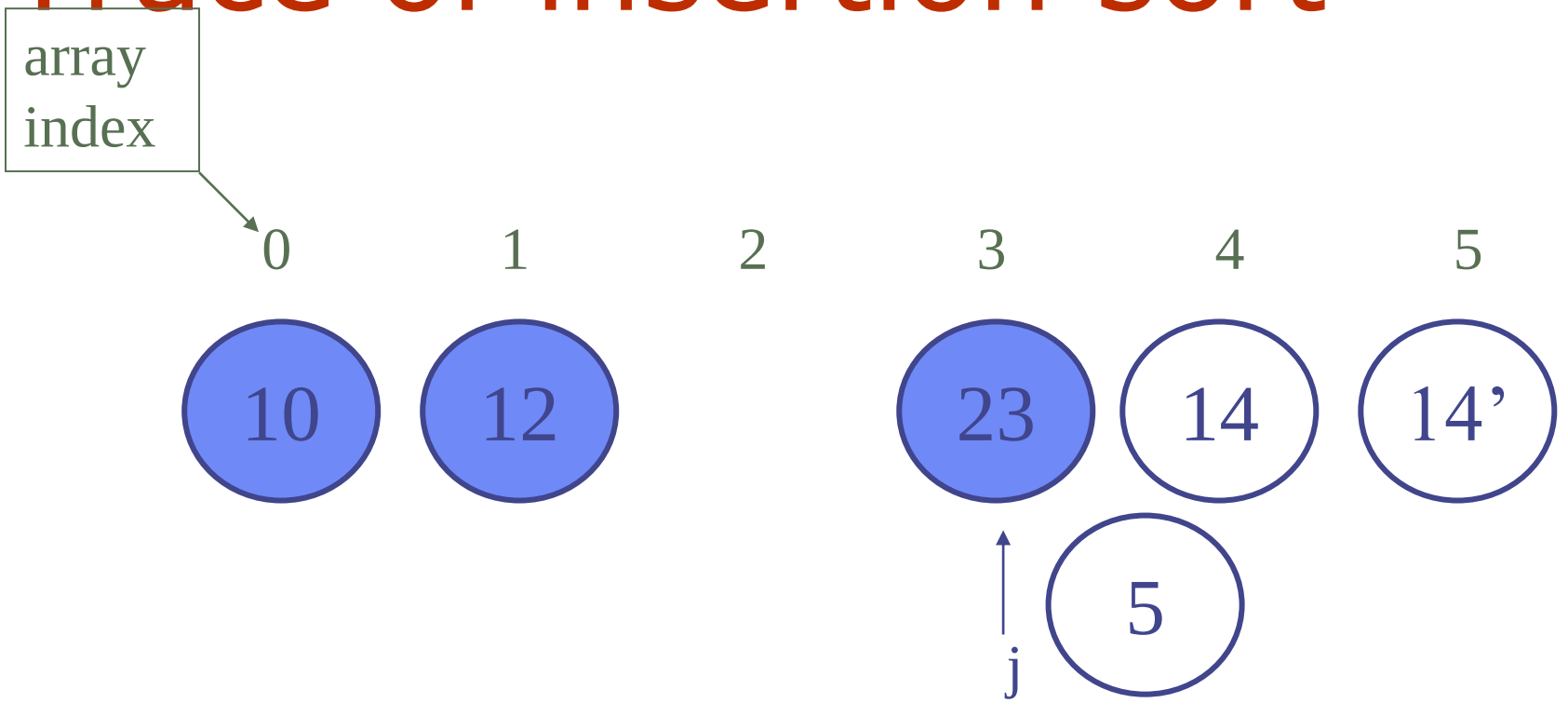
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

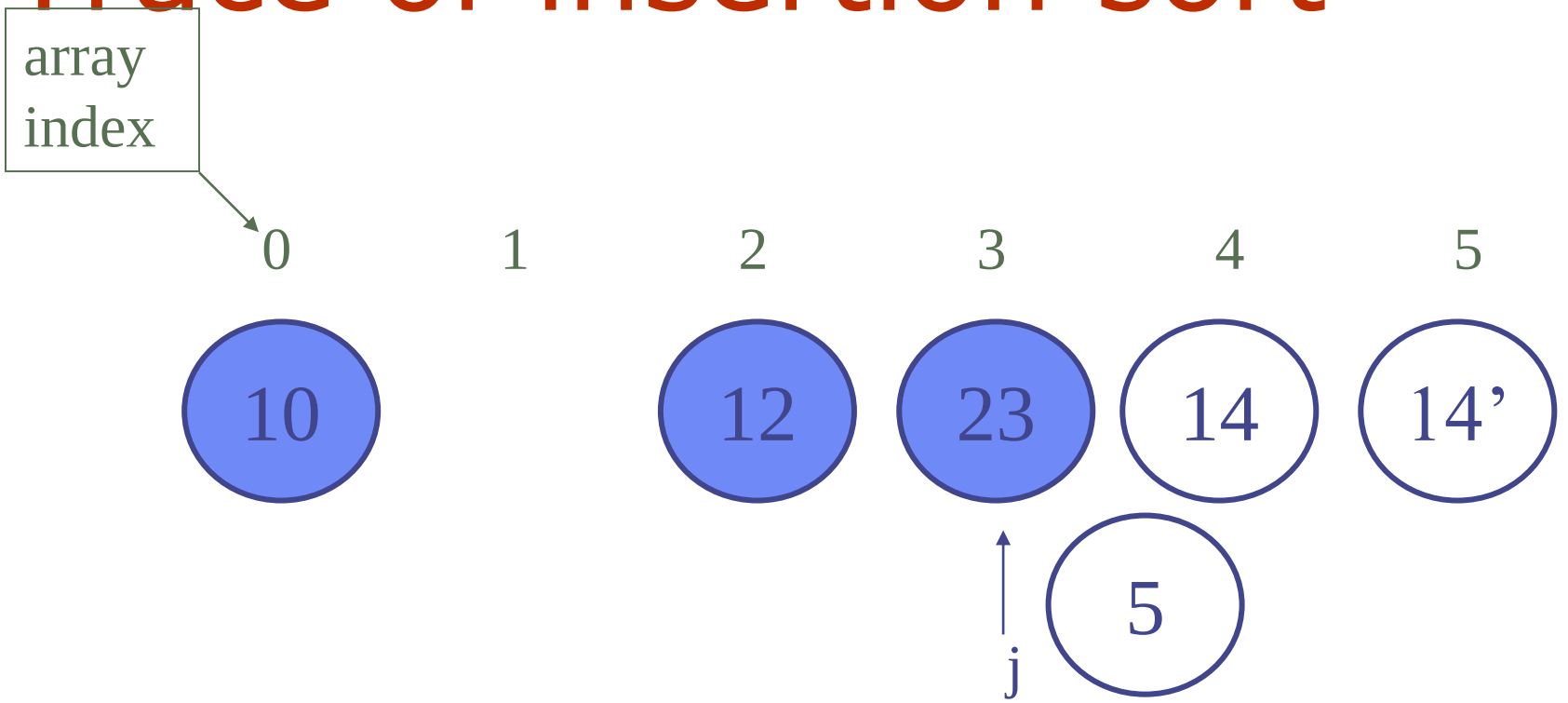
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

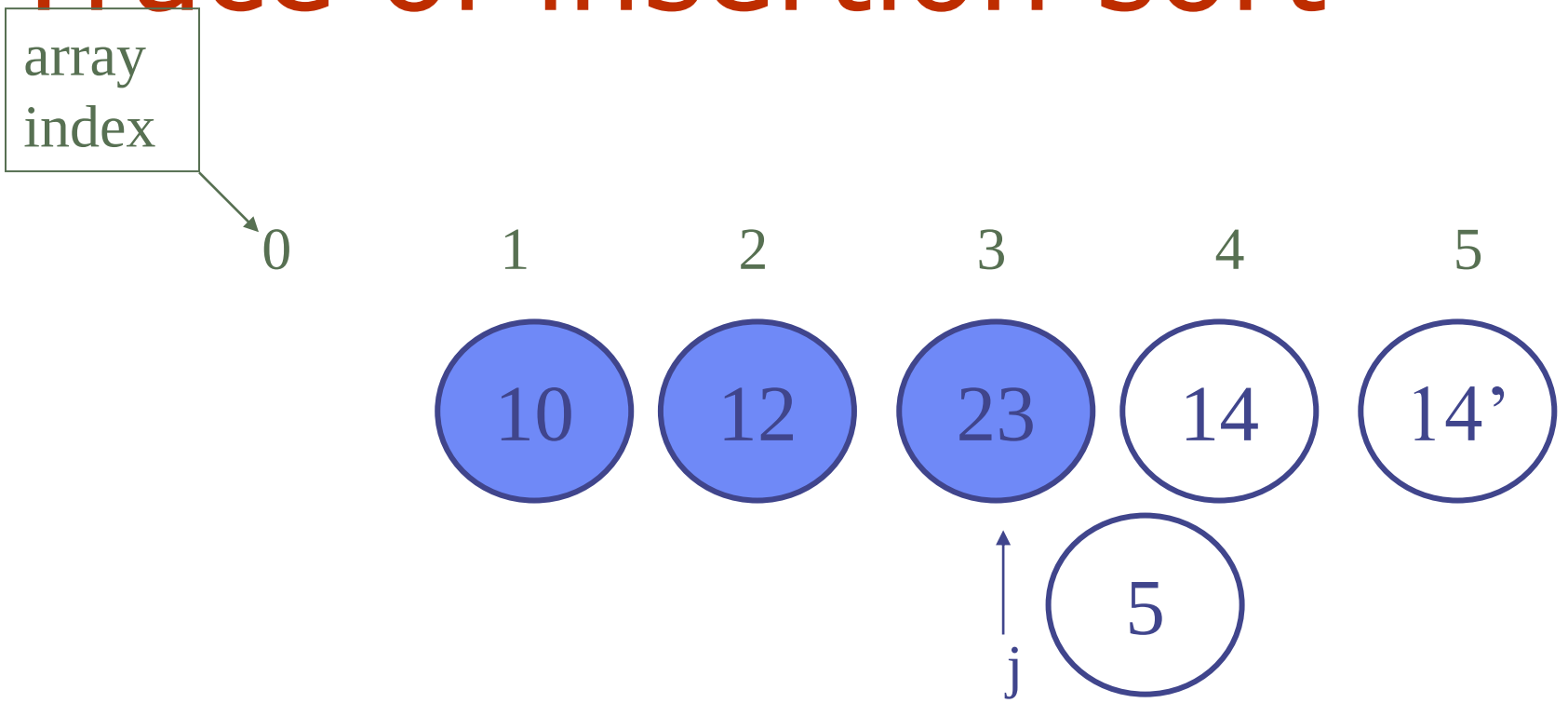
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

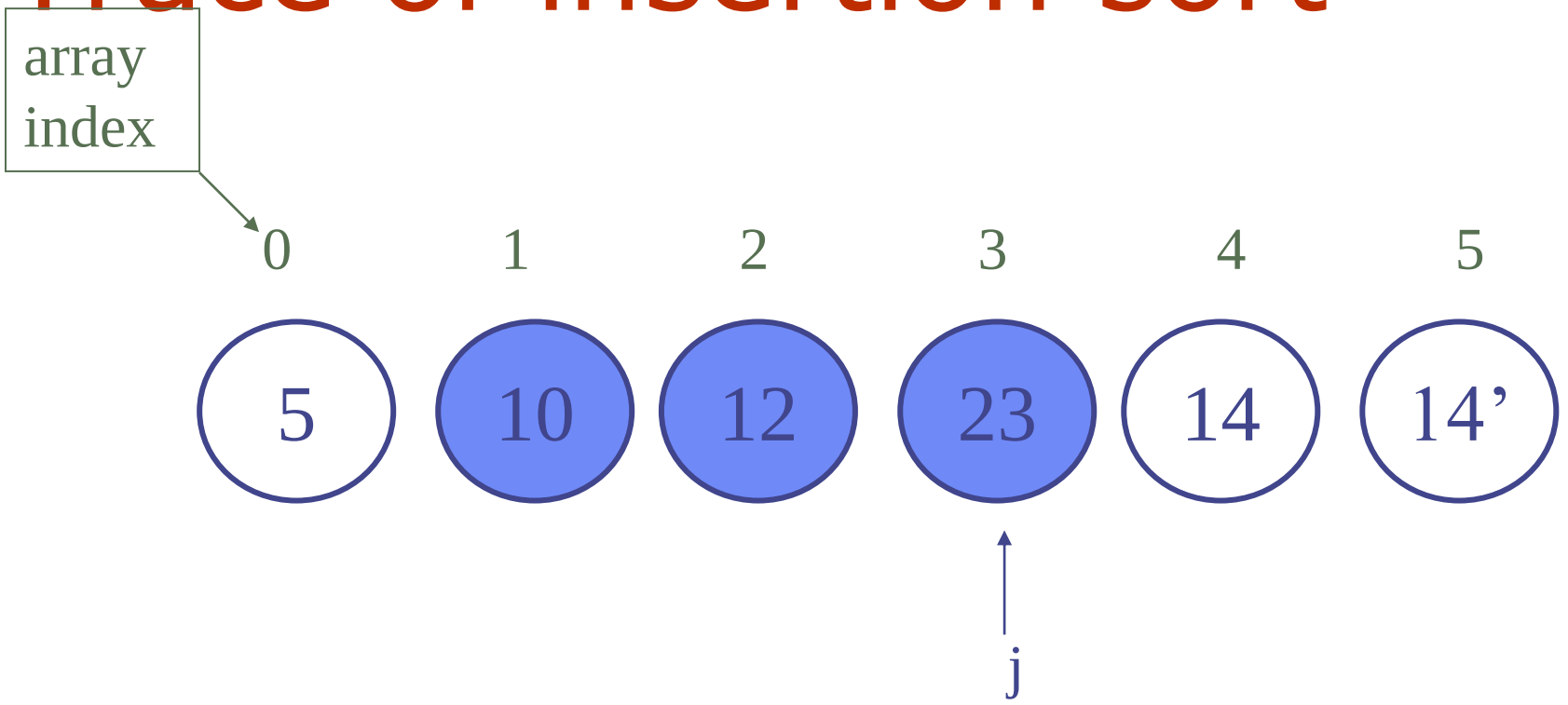
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] >= temp

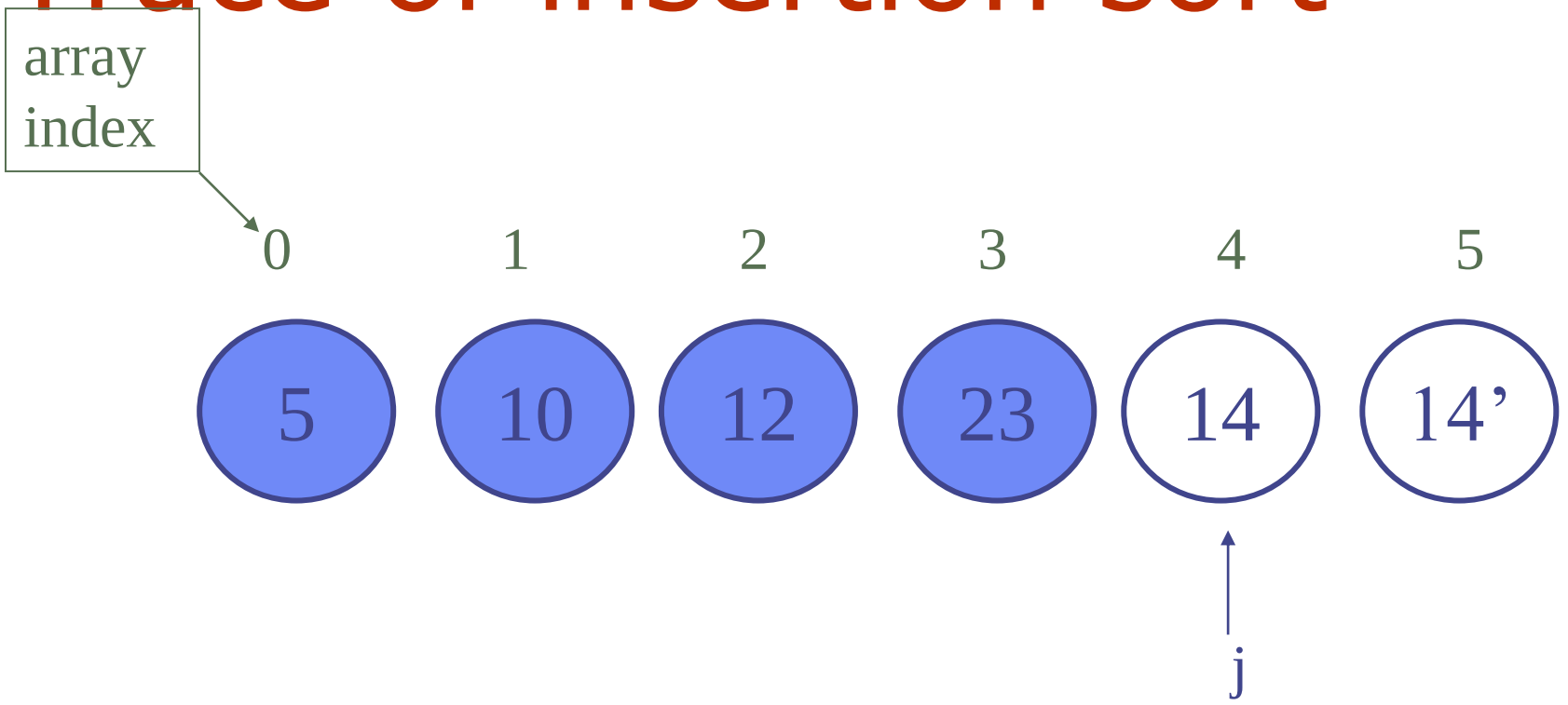
Trace of insertion sort



j = 3, third iteration of the outer loop

arr[i-1] = temp

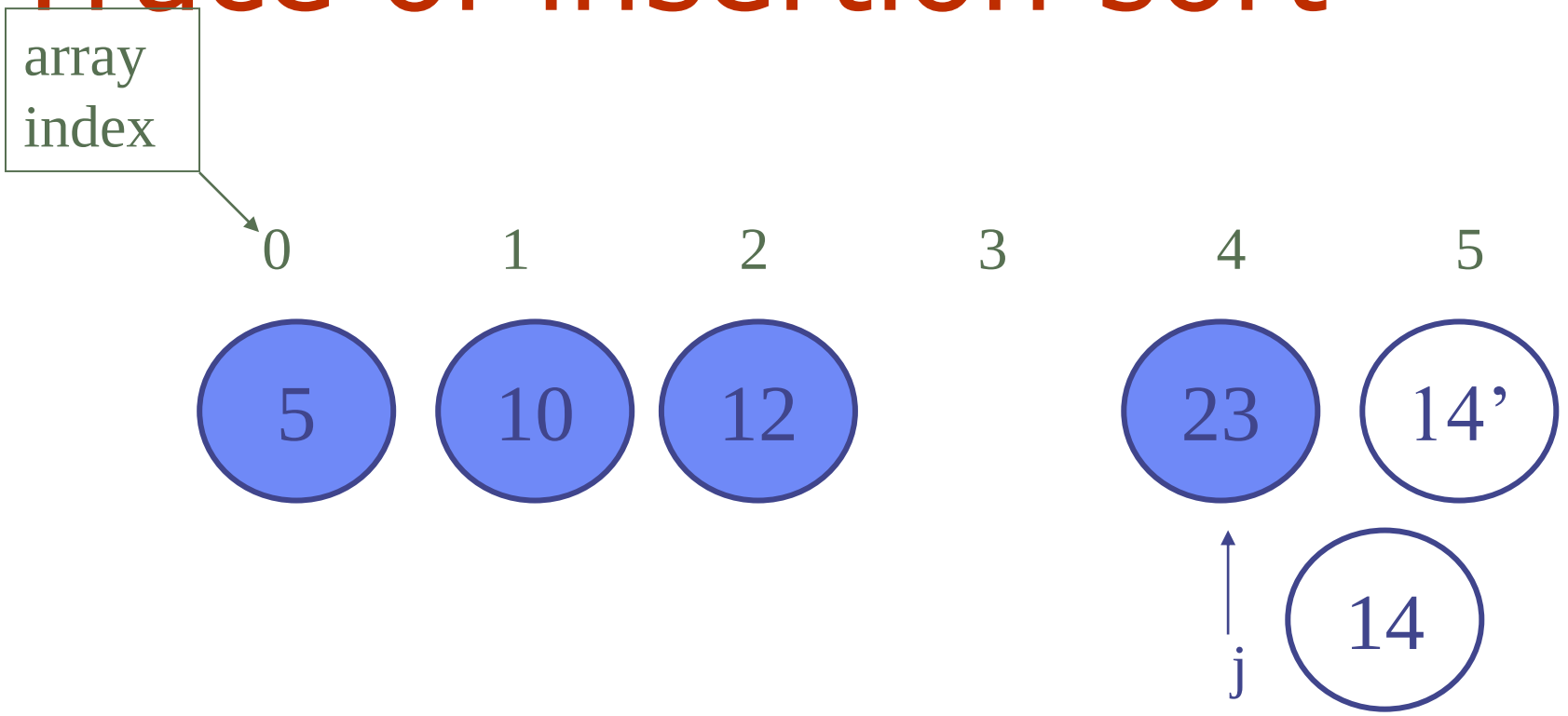
Trace of insertion sort



j = 4, fourth iteration of the outer loop

temp = 14

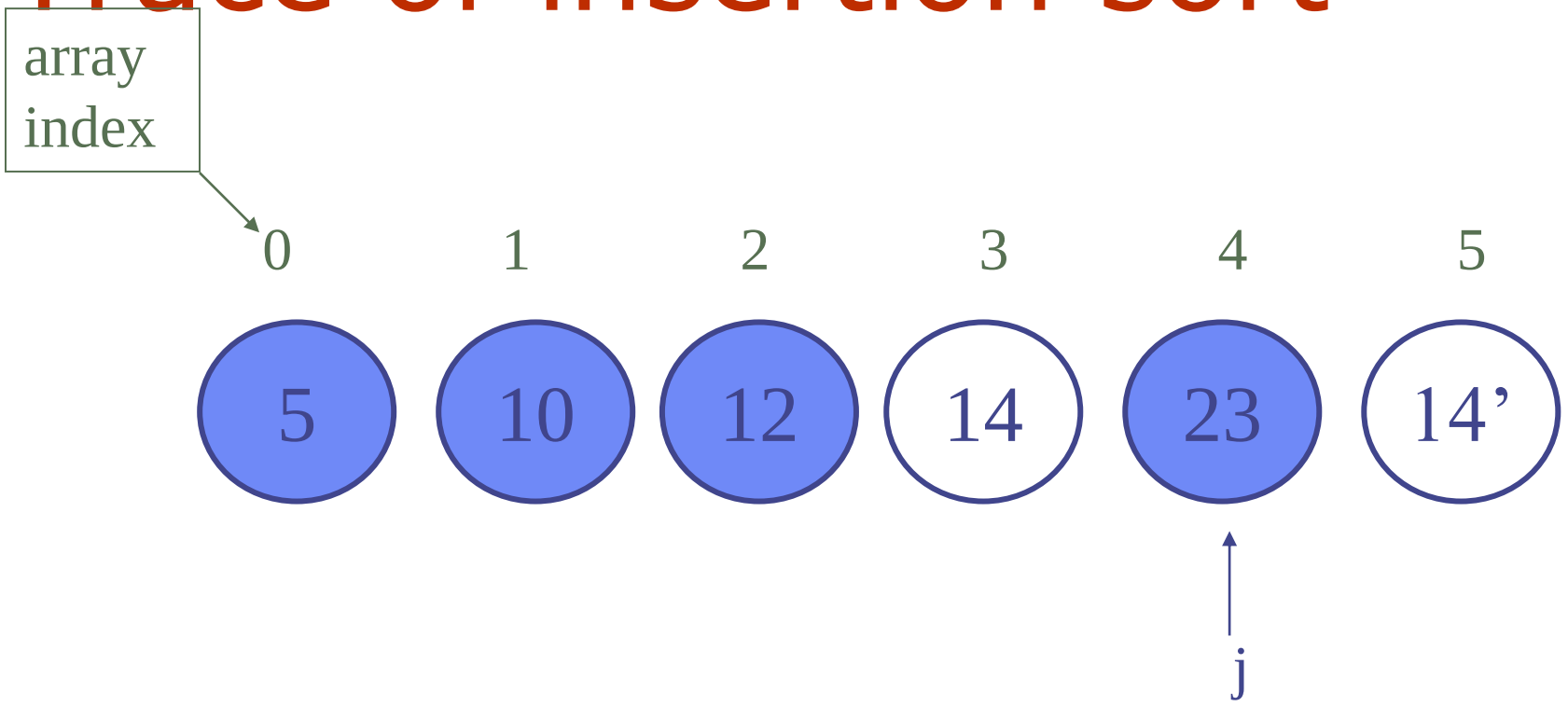
Trace of insertion sort



j = 4, fourth iteration of the outer loop

arr[i-1] >= temp

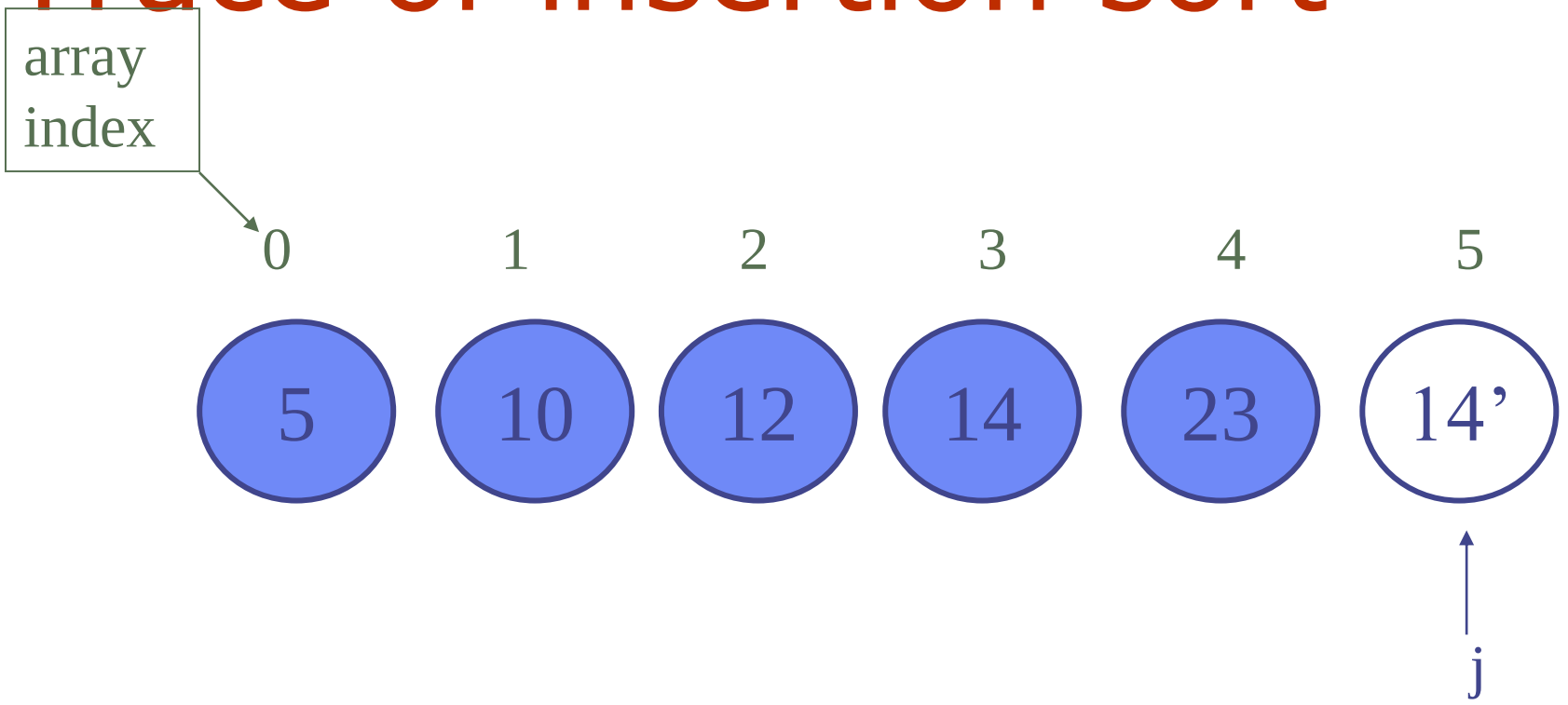
Trace of insertion sort



j = 4, fourth iteration of the outer loop

arr[i-1] = temp

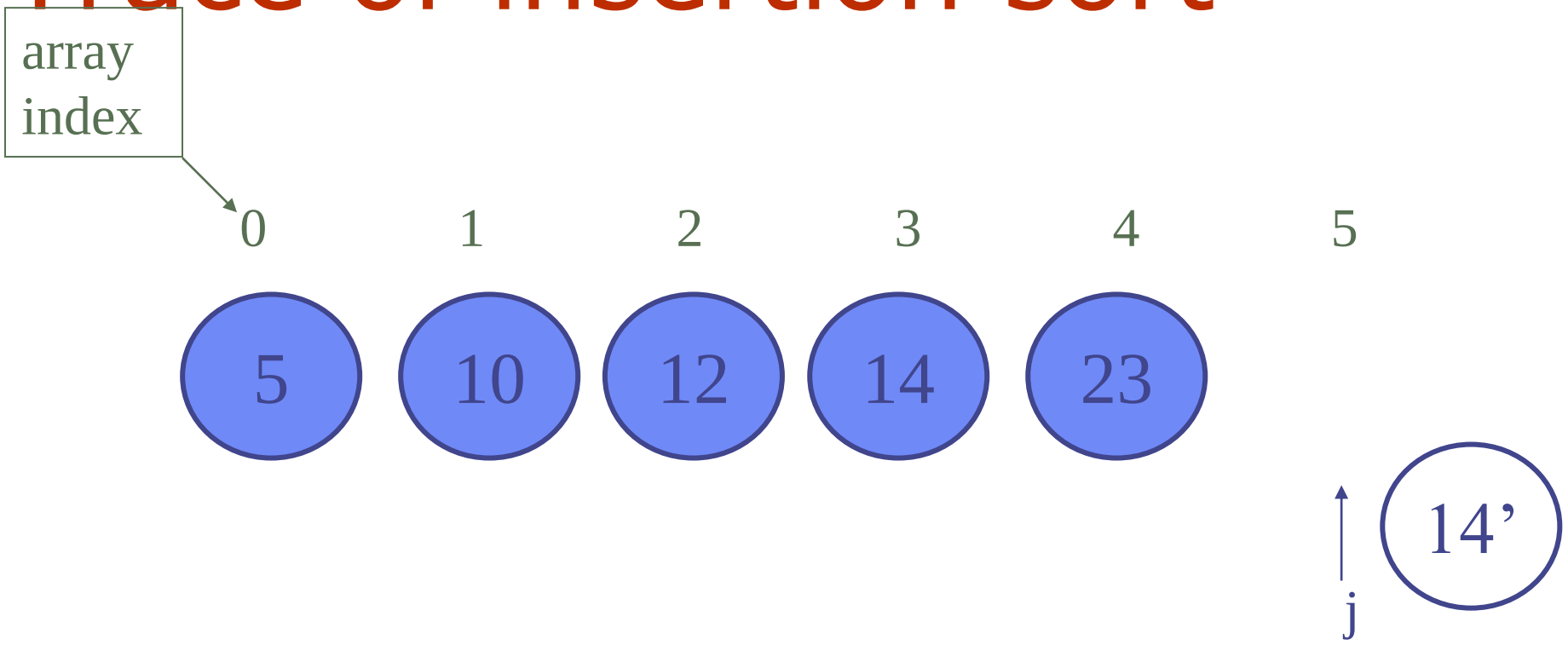
Trace of insertion sort



j = 5, fifth iteration of the outer loop

temp = 15

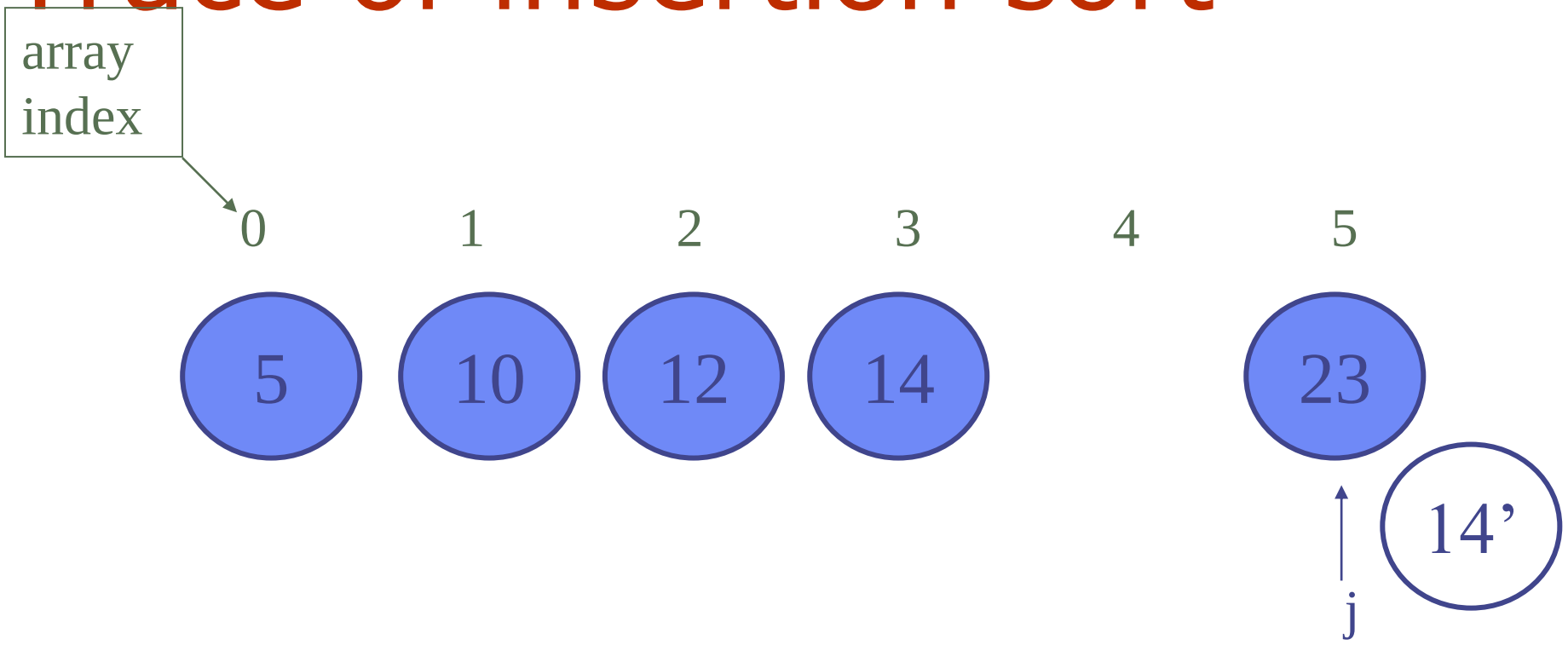
Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] >= temp

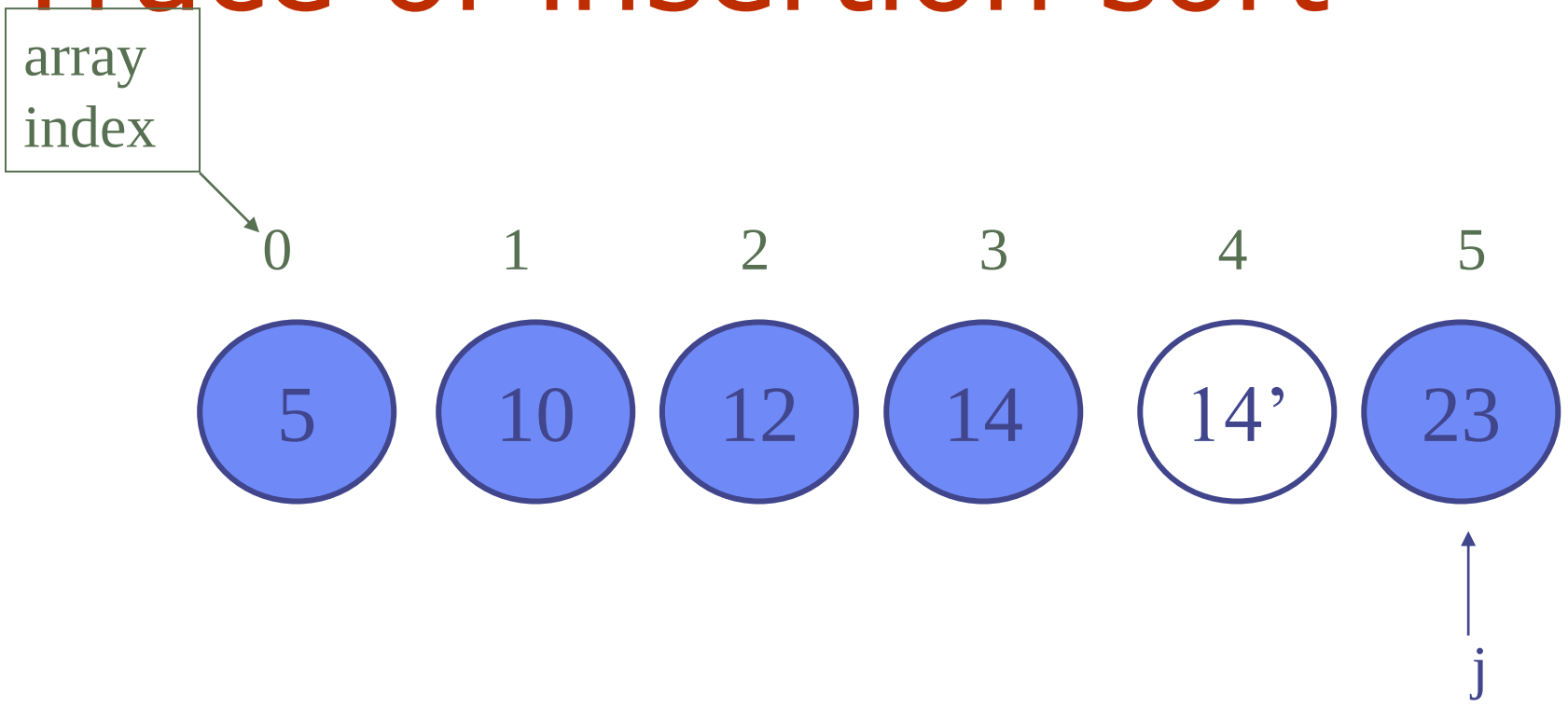
Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] >= temp

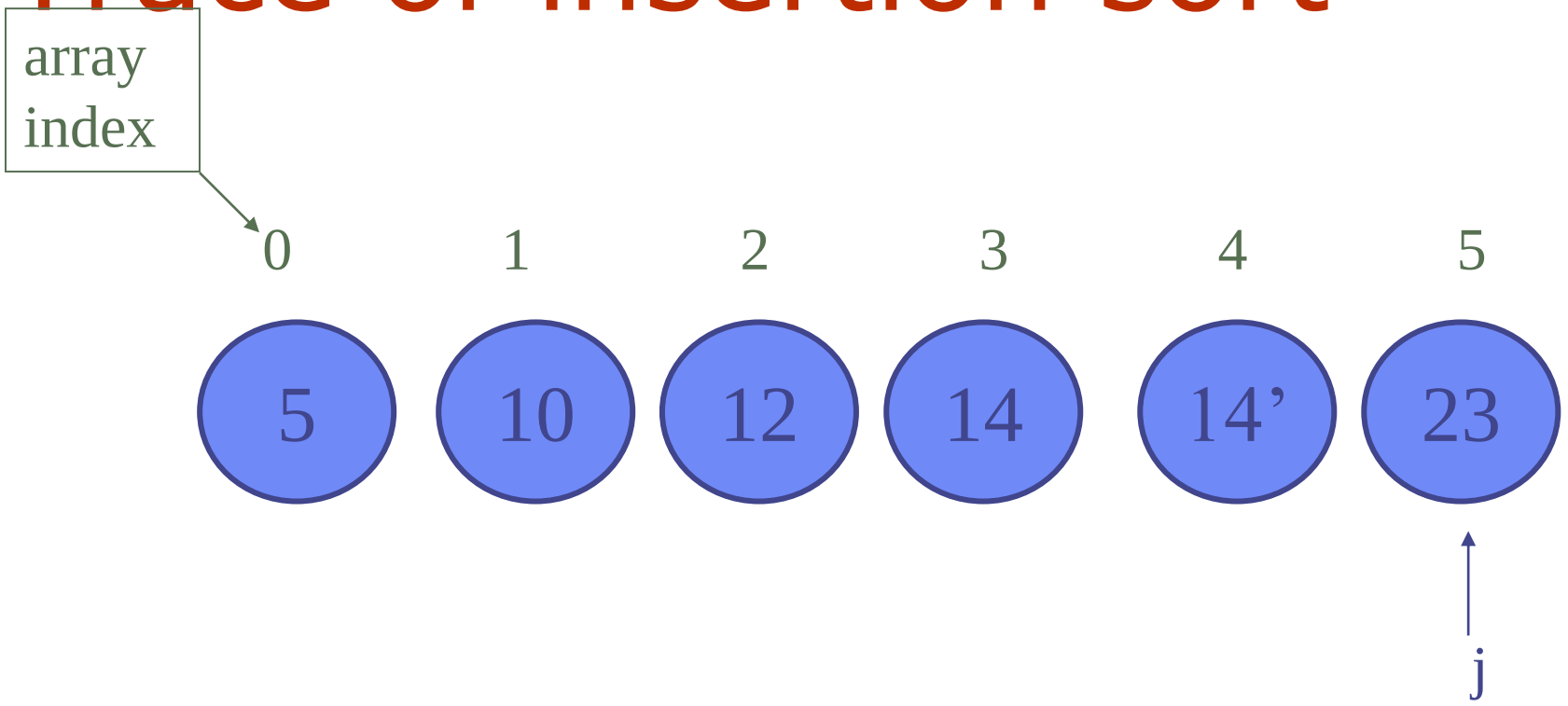
Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] = temp

Trace of insertion sort



j = 5, fifth iteration of the outer loop

arr[i-1] = temp

Note: the sort was stable

Complexity of insertion sort

- In the worst case, has to make n(n-1)/2 comparisons and shifts to the right
- also $O(n^2)$ worst case complexity
- Best case: array already sorted
  - Backwards walk of inner loop stops immediately; no shifts.
  - Becomes O(n)

“Adaptive Sort”

- This is a (bizarre) name for what asking what happens to the complexity when the lists are already “nearly sorted”.
- In many applications lists might already be close to being sorted.
  - E.g. maybe they were from a list that was sorted and then some corrections were made.
  - It is then natural to ask, for each algorithm, whether the efficiency improves.
  - As a start one might consider what happens when the input list is actually already sorted?

"Adaptive Sort"

- Exercise (offline)
- For each of the 3 simple sorting methods
  - Consider how will they will perform if the data is already sorted, or nearly sorted
  - Consider how they might be modified to take advantage.
    - E.g. Suppose that each entry is guaranteed to be no more then one place from its final proper position – then can the algorithms take account of this?

Sorting on Lists

- So far have worked on arrays.

- Will these algorithms also work well on linked lists?
  - What about singly- vs. doubly-linked ?

Bubble sort of lists?

- Is bubble sort also workable for linked lists?
- Bubble sort is just as efficient (or rather inefficient) on linked lists.
  - We can easily bubble sort even for a singly linked list.

Bubble sort of singly-linked lists

- Assume we have a class Node with fields: element of type E and next of type Node.
  - (Strictly speaking getter/setter methods would be better, but this is just for the sake of brevity…)
- The List class just has head field.
- Which way are we going to traverse?
  - There is only one way we can traverse! From the head.
- What should we keep track of?
  - A "border" between the unsorted beginning part of the list, and the sorted end of the list

Bubble sort of a linked list

```
Node border = null; // first node in the sorted part
while (border != head) {
   Node current = head; // start from the first node
   while (current.next != border) {
      if (current.element > current.next.element) {
         E element v = current.element;
         current.element = current.next.element;
         current.next.element = v;
      }
      current = current.next;
   }
   border = current; // the sorted part increases by one
}
```

swap with the next node if elements out of order

Bubble sort of a linked list

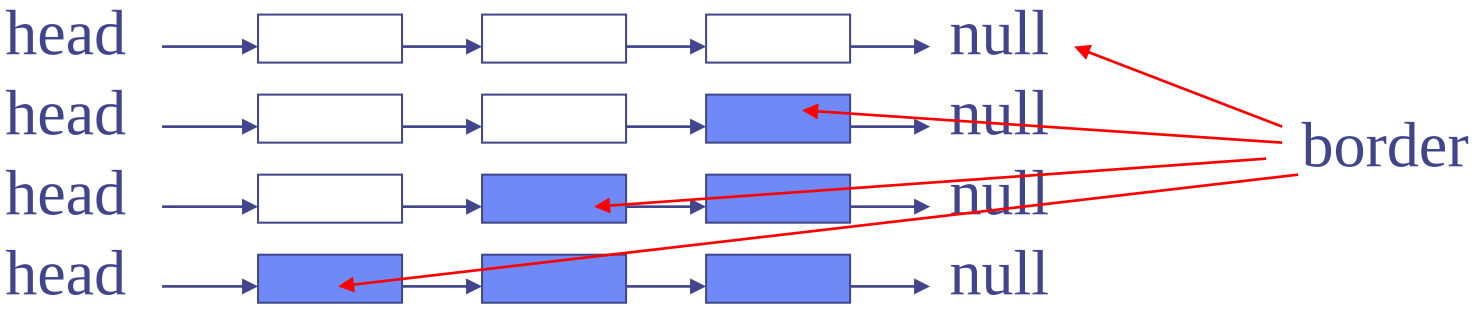
In the previous slide

```
E element v = current.element;
current.element = current.next.element;
current.next.element = v;
```

- Does a direct swap of the contents. This is fine if the element is small.
- This is usually the case as it will be a primitive data type or an object reference.
- One very rarely swaps the actual contents of objects, but usually swap the references to them.
- If necessary, then one could instead the references/pointers that give the structure of the list – that is, swap the way that the nodes are built into the list by rearranging the ‘next’ values.

Complexity of bubble sort on lists

- Same complexity as for arrays $O(n^2)$:
  - First time we iterate until we see a null (swapping elements)
  - Second time we iterate until we see the last node;
  - … each time the border is one link closer to the head of the list
  - until border == head.



Selection sort on linked lists

- Implementation similar to bubble sort; also $O(n^2)$
- Instead of pos_greatest, have a variable Node largest which keeps the reference of the node with the largest element we have seen so far
- Swap elements once in every iteration through the unsorted part of the list:

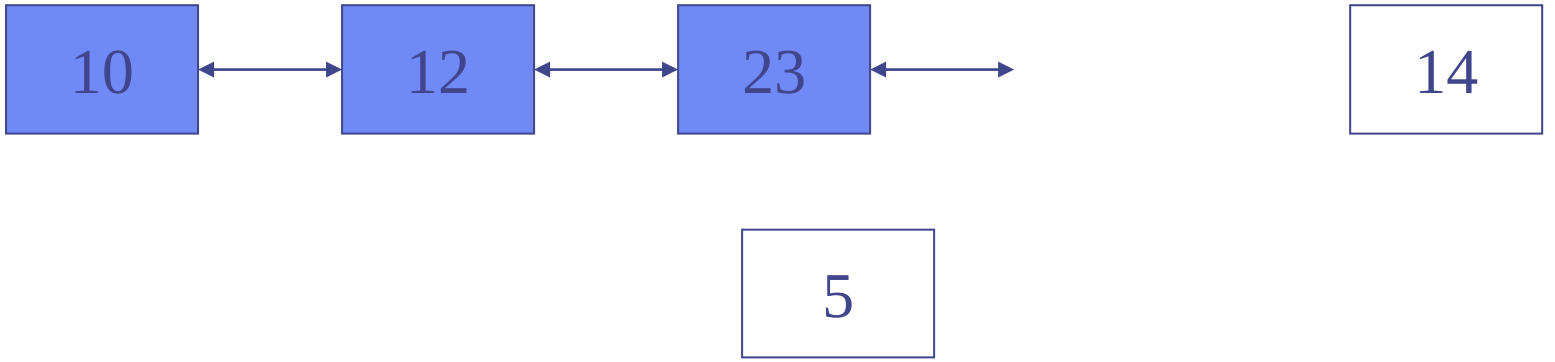
E element v = current.element; current.element =largest.element; largest.element= v;

Insertion sort on linked lists

- This is a suitable sorting method (only) for **doubly** linked lists – as need to walk backwards
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):

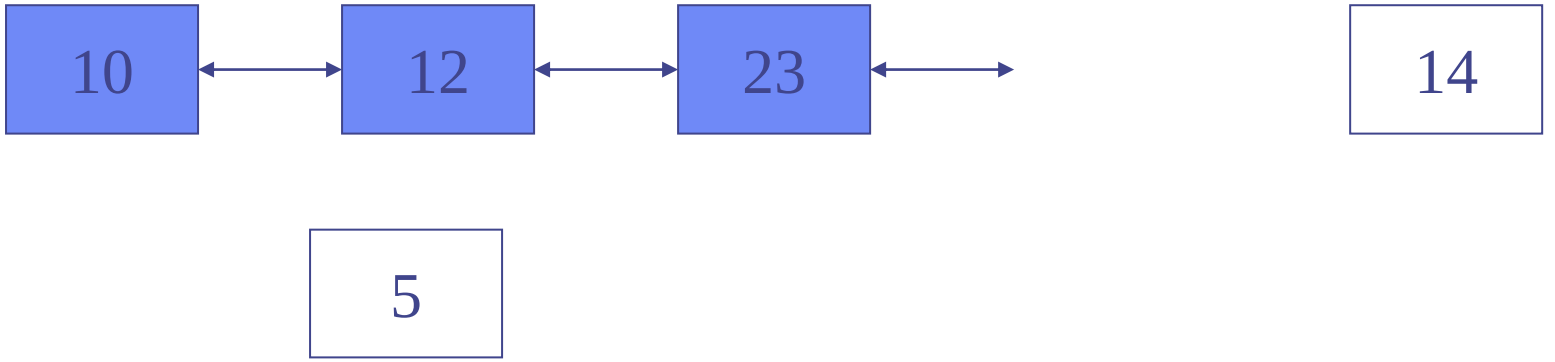
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don’t need to shift other nodes to make space for it (but need to find the place…):



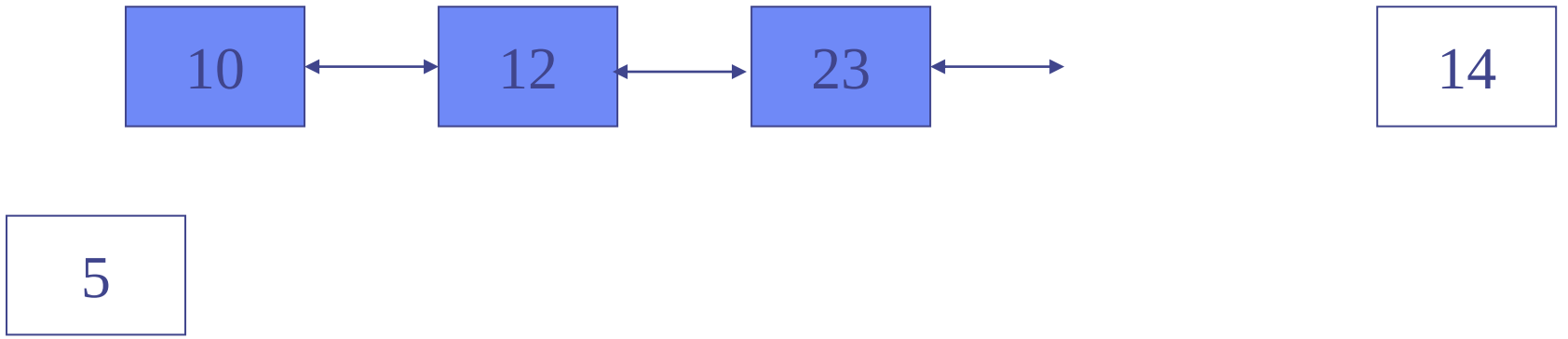
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don’t need to shift other nodes to make space for it (but need to find the place…):



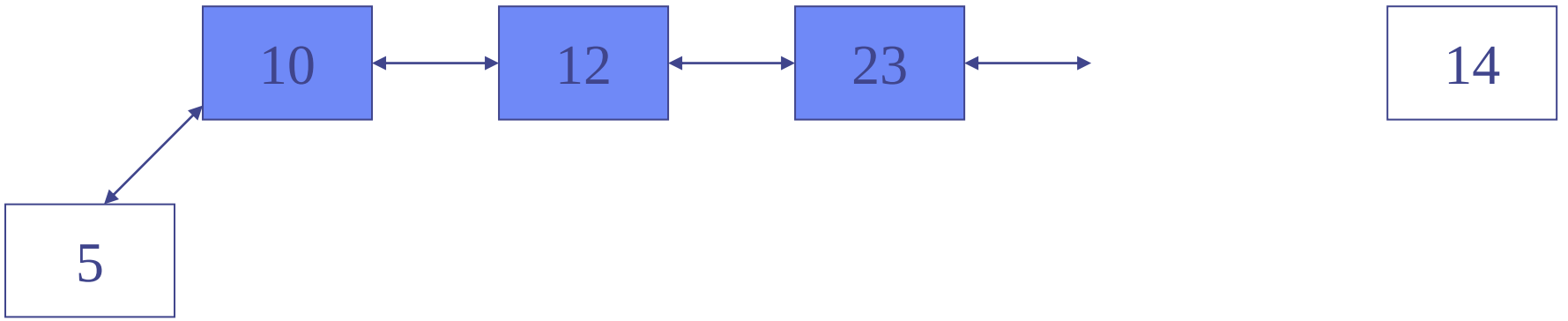
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place…):



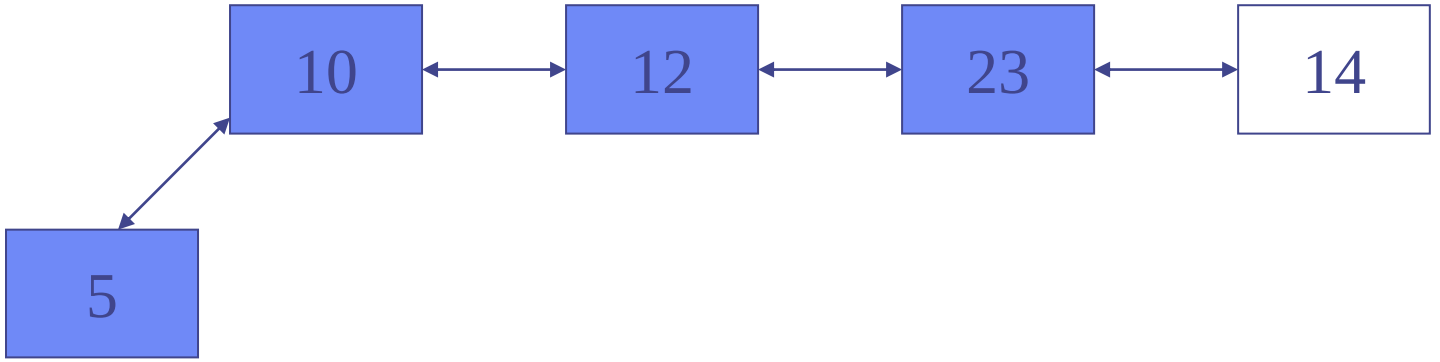
Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don’t need to shift other nodes to make space for it (but need to find the place…):



Insertion sort on linked lists

- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don’t need to shift other nodes to make space for it (but need to find the place…):



Exercise (offline)

- Implement all the methods.
- Good practice at coding ☺

Expectations

- Know these simple sorting algorithms
  - be able to implement them
  - recognise them
  - be able to analyse their complexity
    - And roughly what happens to the complexity if the input is already (nearly) sorted
- Understand the meaning of “stable” in the context of sorting
- Understand the idea of “adaptive sort” and performance on lists that are already partially sorted.