

COMP2054-ADE

ADE Lec07

Simple Sorting Algorithms

Lecturer: Andrew Parkes

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

Simple Sorting Algorithms

1

Today is simple sorting algorithms.

Again these will be probably have been seen before, but now the emphasis is on the efficiency and a couple of other properties that are likely to be new.

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

Will focus on just sorting and will just do 3 algorithms,
For simplicity we just do integers and we assume we want larger elements on the right.
But can of course do more general cases.

Note there are a lot of slides, but only because we do “animations” one frame at a time.

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

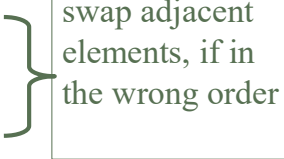
We will do bubble sort very quickly as most people will already have seen it.

The basic idea is to do repeated scans and swapping everything to the right whenever any two entries are found out of order.

Hence larger numbers move to the right..

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;
            }
        }
    }
}
```



swap adjacent elements, if in the wrong order

The code is straightforward.

There are two nested for loops.

The inner loop does a scan and bubbles larger elements to the right.

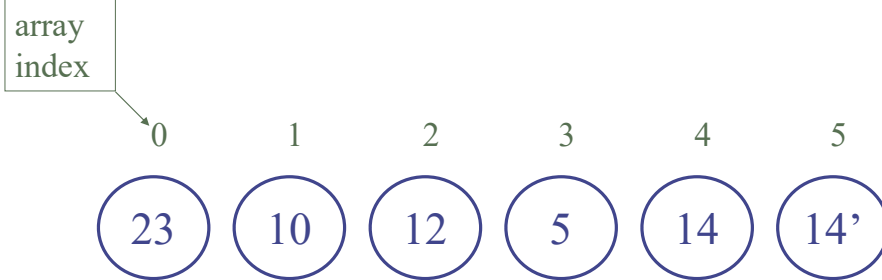
The range of the inner loop is reduced at each step, starting as the entire array and finishing with just two elements.

As ever the recommendation is to code it yourself and put in enough print statements to be able to do it.

The best way to understand is to just do a trace and animation.

Which we do now.

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)

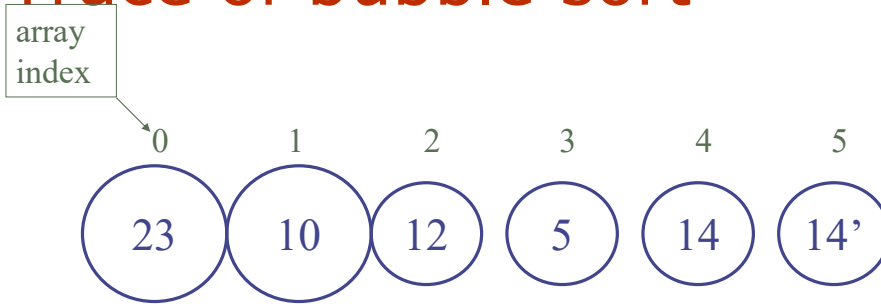
First iteration of outer loop, scans across and swaps.

First thing to notice is that we have two 14s.

They both have value 14, but are different – the prime is just to enable keeping track.

We will keep track of what happens to them.

Trace of bubble sort

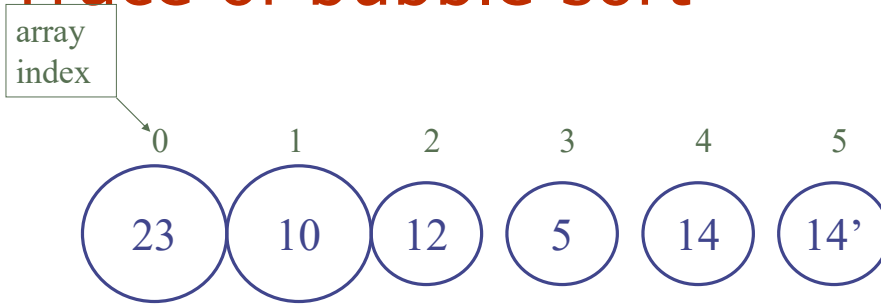


$i = 5$, first iteration of the outer loop

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

Compare the first two elements.
They need to swap, so do so.

Trace of bubble sort

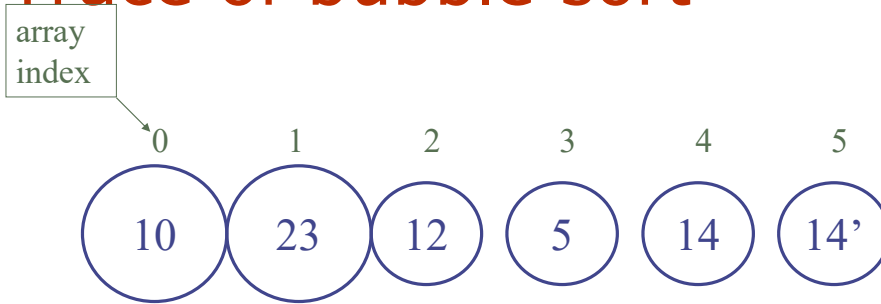


SWAP!

$i = 5$, first iteration of the outer loop

$j = 0$, comparing $\text{arr}[0]$ and $\text{arr}[1]$

Trace of bubble sort



SWAP!

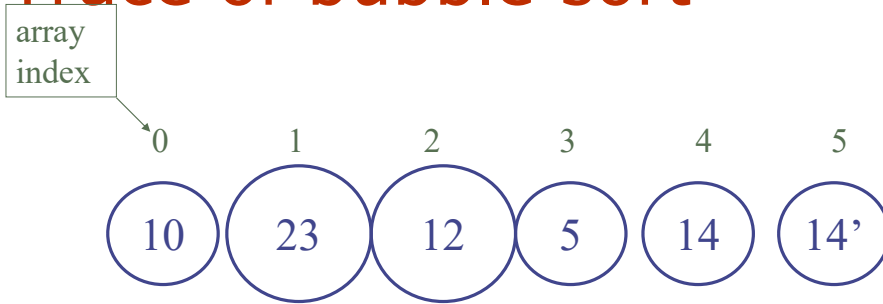
$i = 5$, first iteration of the outer loop

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

Swap done. Move to the next.

And we carry on.

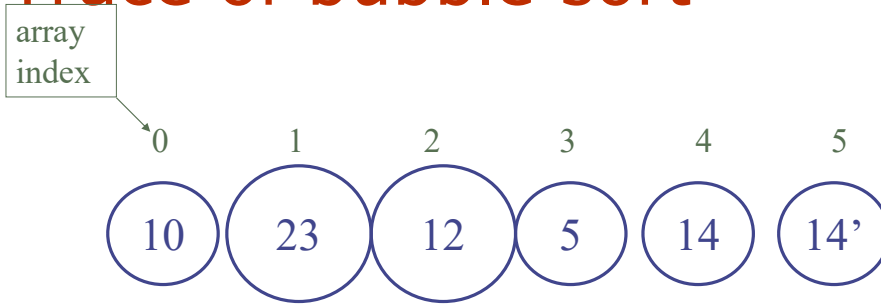
Trace of bubble sort



$i = 5$, first iteration of the outer loop

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

Trace of bubble sort

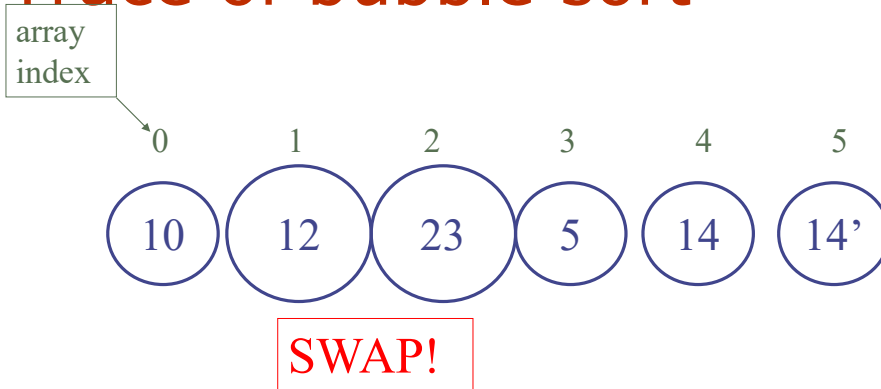


SWAP!

$i = 5$, first iteration of the outer loop

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

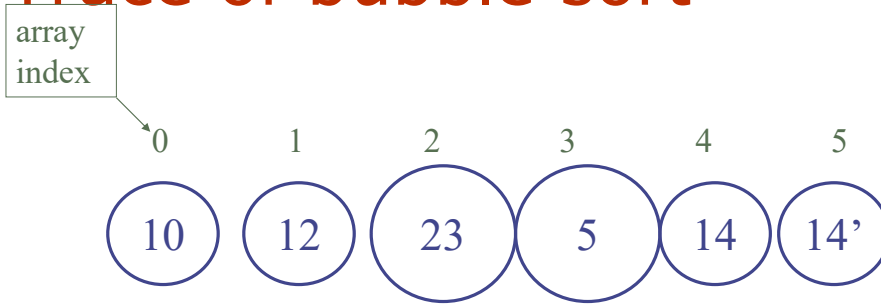
Trace of bubble sort



$i = 5$, first iteration of the outer loop

$j = 1$, comparing $\text{arr}[1]$ and $\text{arr}[2]$

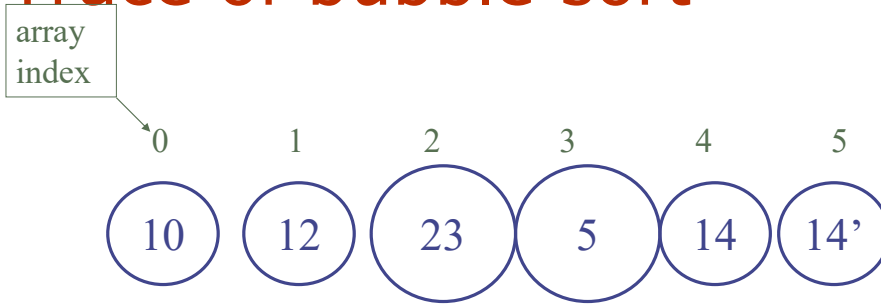
Trace of bubble sort



$i = 5$, first iteration of the outer loop

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

Trace of bubble sort

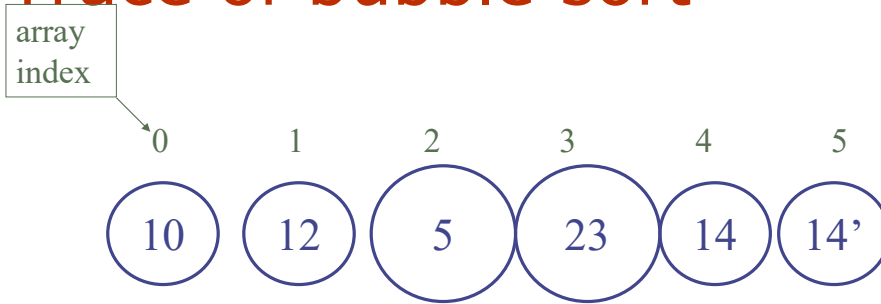


SWAP!

$i = 5$, first iteration of the outer loop

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

Trace of bubble sort

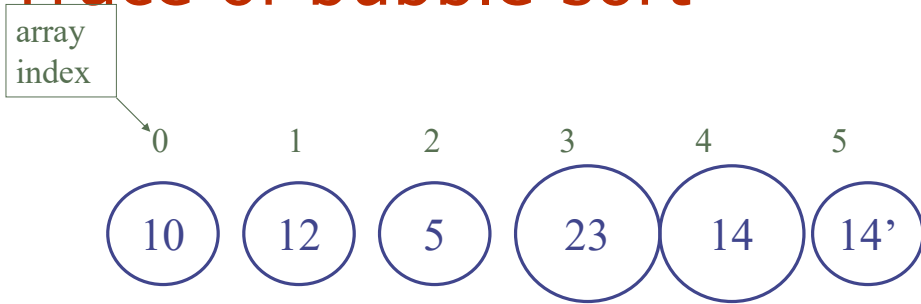


SWAP!

$i = 5$, first iteration of the outer loop

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

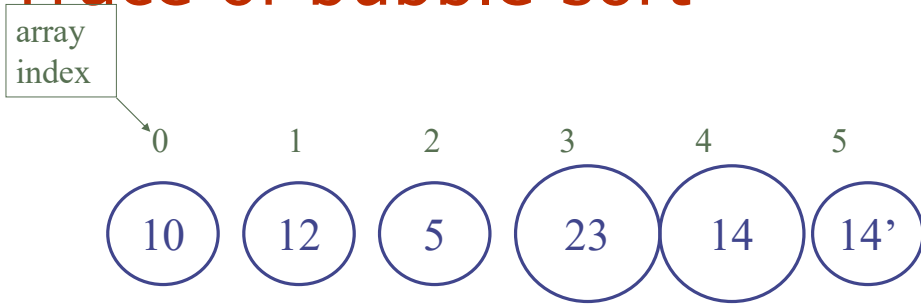
Trace of bubble sort



$i = 5$, first iteration of the outer loop

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

Trace of bubble sort

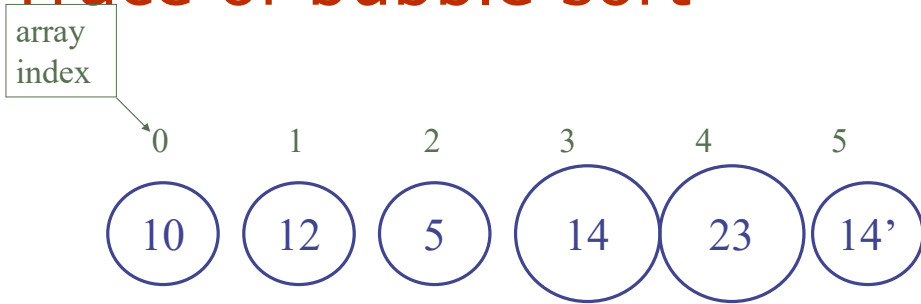


SWAP!

$i = 5$, first iteration of the outer loop

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

Trace of bubble sort

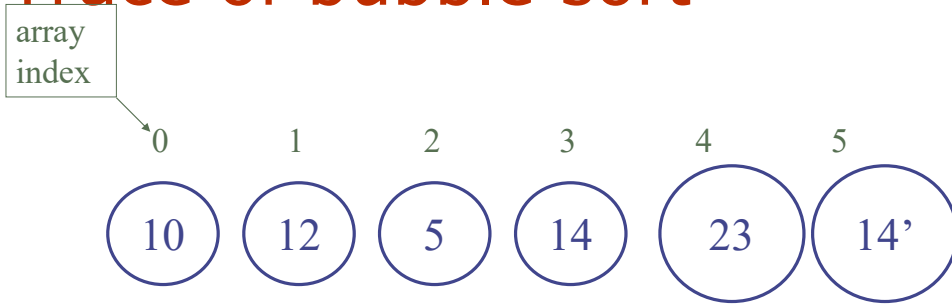


SWAP!

$i = 5$, first iteration of the outer loop

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

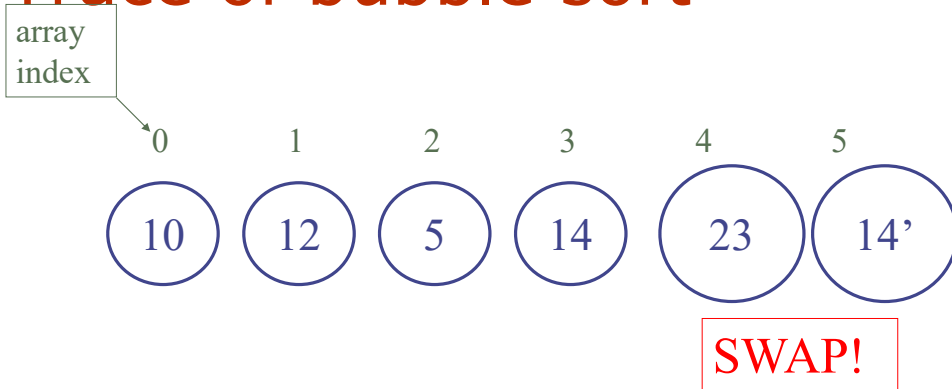
Trace of bubble sort



$i = 5$, first iteration of the outer loop

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

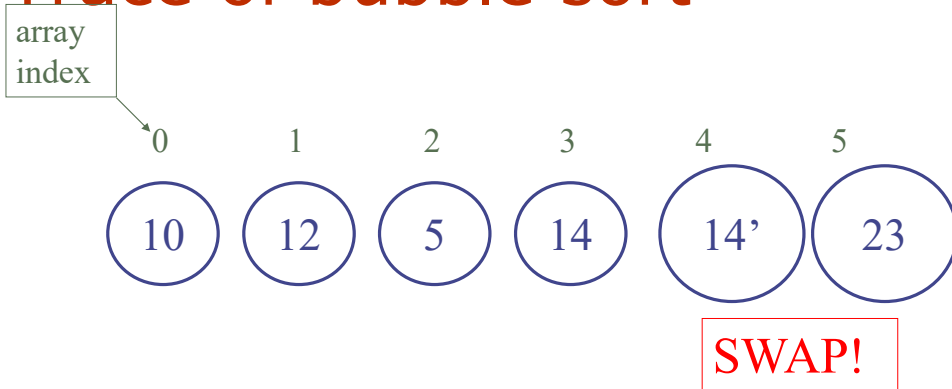
Trace of bubble sort



$i = 5$, first iteration of the outer loop

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

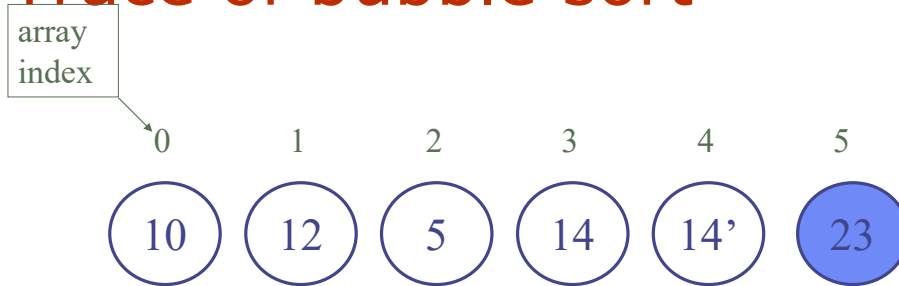
Trace of bubble sort



$i = 5$, first iteration of the outer loop

$j = 4$, comparing $\text{arr}[4]$ and $\text{arr}[5]$

Trace of bubble sort



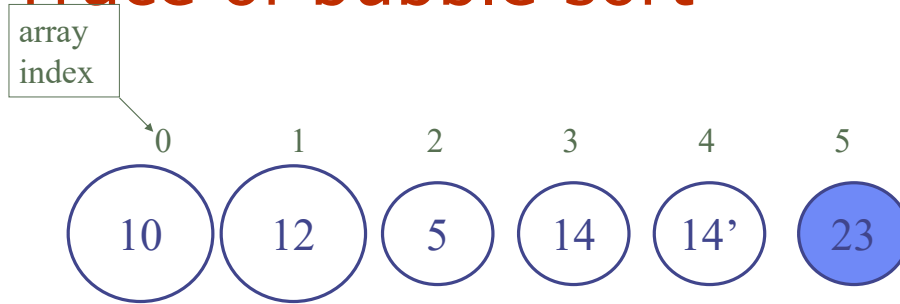
$i = 5$, first iteration of the outer loop
inner loop finished; largest element
in position 5, positions 0-4 unsorted

At this point the largest element will have been swapped to the rightmost.

So the 23 will not move again, so the next outer loop does not need to scan it, hence the range is decreased.

So start scanning again.

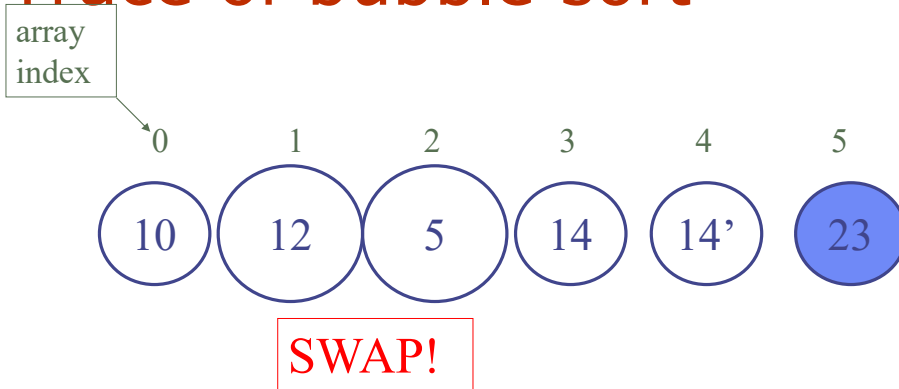
Trace of bubble sort



$i = 4$, second iteration of the outer loop

$j = 0$, comparing $\text{arr}[0]$ with $\text{arr}[1]$

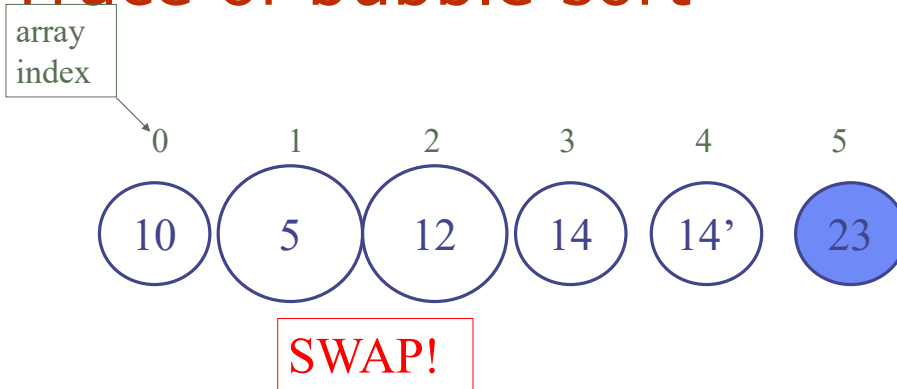
Trace of bubble sort



$i = 4$, second iteration of the outer loop

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

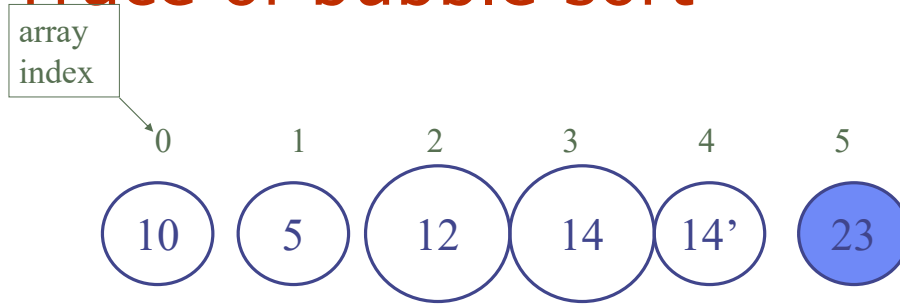
Trace of bubble sort



$i = 4$, second iteration of the outer loop

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

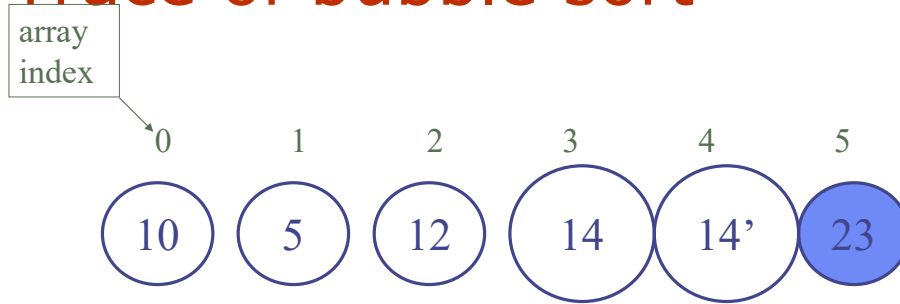
Trace of bubble sort



$i = 4$, second iteration of the outer loop

$j = 2$, comparing $\text{arr}[2]$ with $\text{arr}[3]$

Trace of bubble sort



$i = 4$, second iteration of the outer loop

$j = 3$, comparing $\text{arr}[3]$ with $\text{arr}[4]$

Note that the 14 and 14' are not swapped.

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

Now note that the 14' and 23 are in final positions.

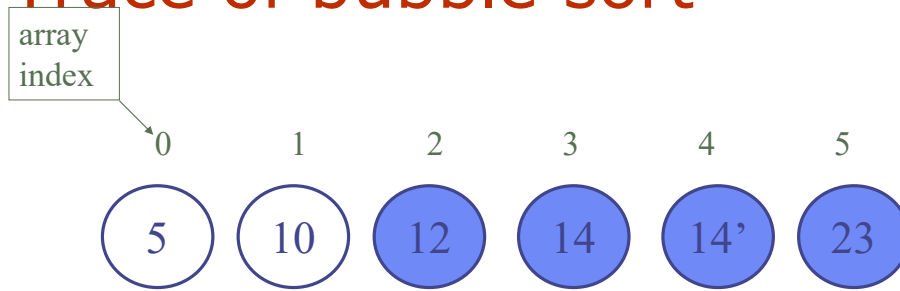
Trace of bubble sort



After third iteration...

Next iterations are the same.

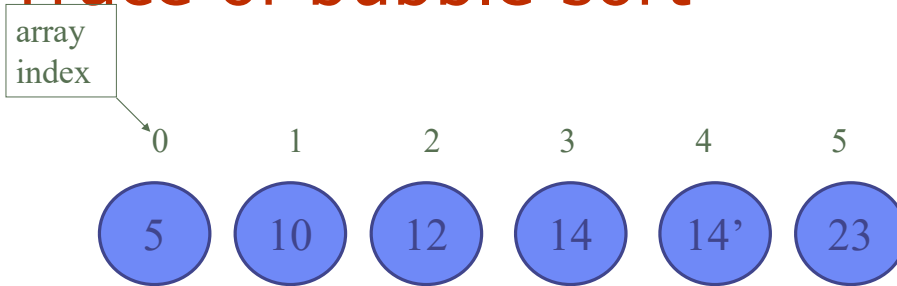
Trace of bubble sort



After fourth iteration...

And we just finish off.

Trace of bubble sort



After fifth iteration...

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

bubble sort is “stable”

We end up sorted. And the order of the 14 and 14' has not changed.
This is stated as bubble sort being 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!

But 14 and 4 are both 14, so why care!?

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

The reason is that we compare with respect to a property but not the entire object. We can have different objects, and they might compare as equal. E.g. consider a spreadsheet we might sort a sheet by one column,

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.**

It would be very annoying if sorting on a column then entries that we equal in that column were swapped.

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: $t((n-1) + (n-2) + \dots + 1)$, 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 .

We can now analyse the complexity, and again this is just a matter of counting.

We tend to just count the number of comparisons.

The largest loop will do $n-1$ and each iteration of the outer loop it drops by one.

So we get the sum as given.

There is also other stuff to count.

Complexity of bubble sort

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

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

[“arithmetic sum” see self-study material]

so our function equals

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

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

Final answer come to the expression, and there is the “arithmetic sum”.
It comes to an expression that is quadratic.
Hence by the work we already did then this is just $O(n^2)$.

Proof: Complexity of bubble sort (study offline)

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

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

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

$$t n^2 + t_1 n^2 + k n^2 \text{ (if } n \geq 1)$$

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

This is just to do the formal proof.

The reasoning just uses

$A - B \leq A$ when $B \geq 0$. for the first line, i.e. can ignore negative terms

Then also uses (when $n \geq 1$)

$$\frac{1}{2} t n^2 \leq t n^2$$

and

$$n \leq n^2$$

and

$$1 \leq n^2$$

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!?

37

So we now move to selection sort.

This basically similar to bubble sort, but delays the move to the end.

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”

38

We might want to do this if the swap is expensive, and so might make it a bit faster.

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

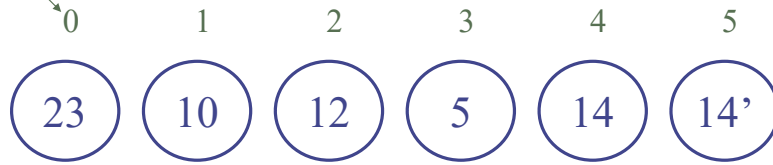
39

The code is basically the same as bubble sort, except the inner loop does not swap, just identifies.

Then the actual swap, if it is needed, is done after the inner loop.
As usual best way to see this is just an animation.

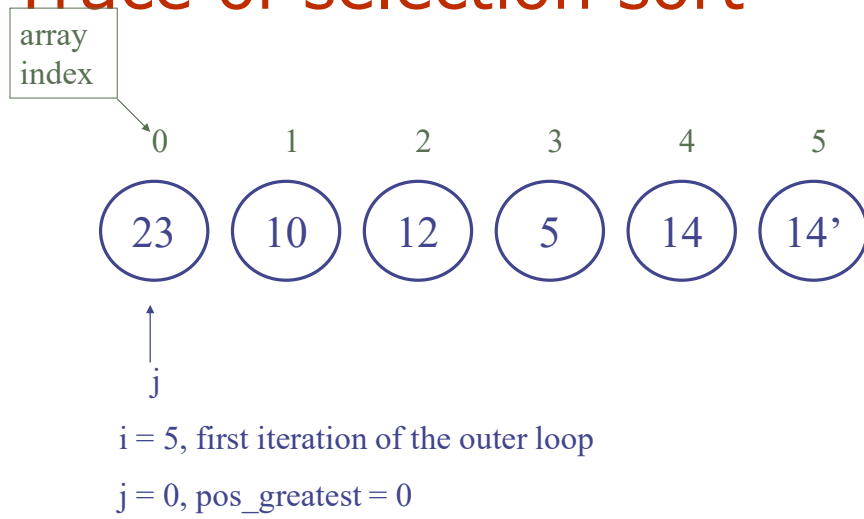
Trace of selection sort

array
index

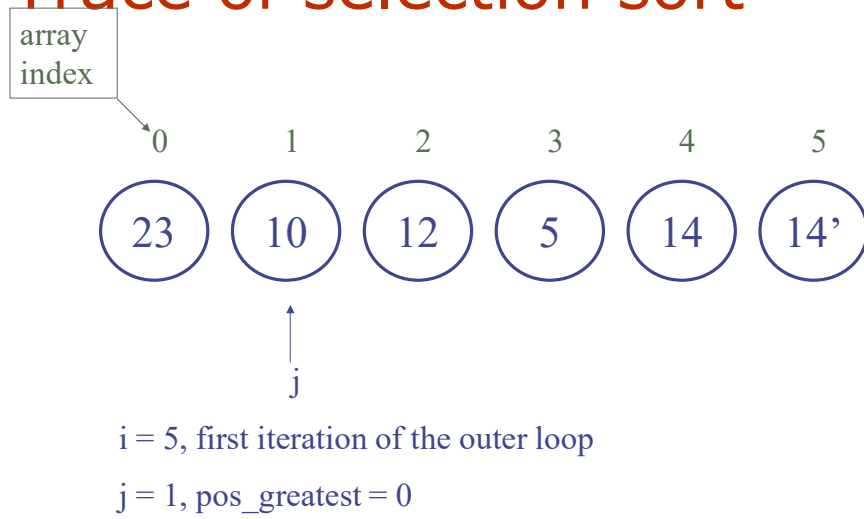


$i = 5$, first iteration of the outer loop

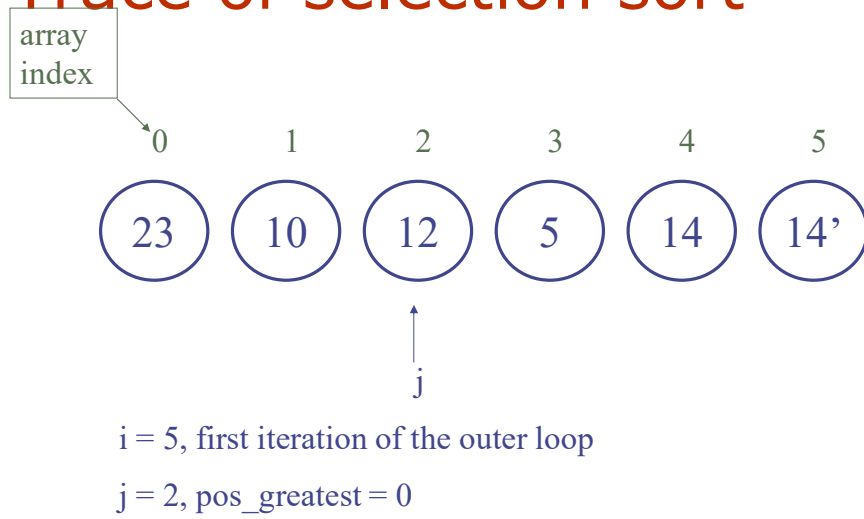
Trace of selection sort



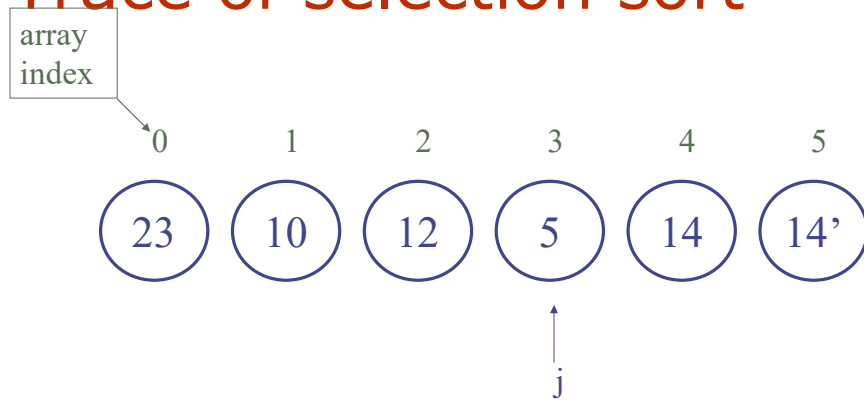
Trace of selection sort



Trace of selection sort



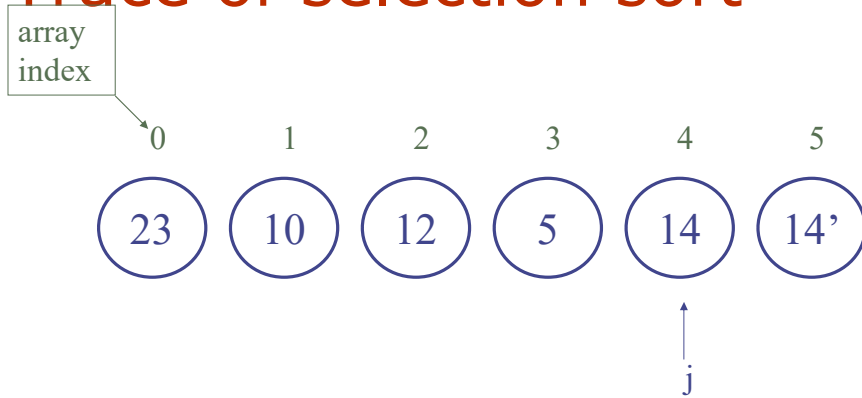
Trace of selection sort



$i = 5$, first iteration of the outer loop

$j = 3$, $\text{pos_greatest} = 0$

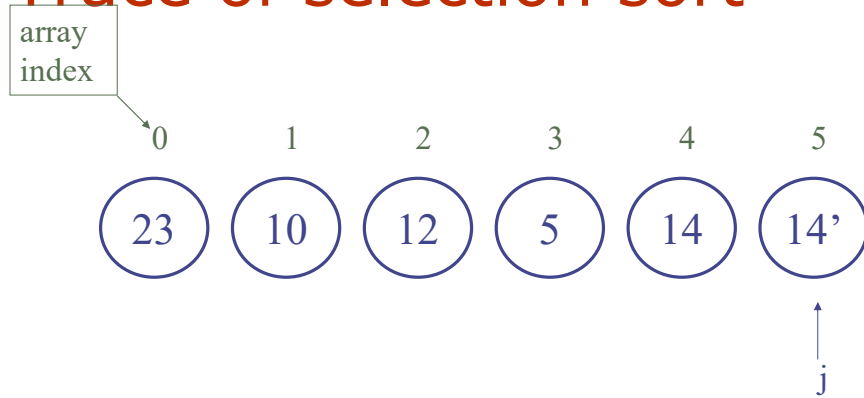
Trace of selection sort



$i = 5$, first iteration of the outer loop

$j = 4$, $\text{pos_greatest} = 0$

Trace of selection sort

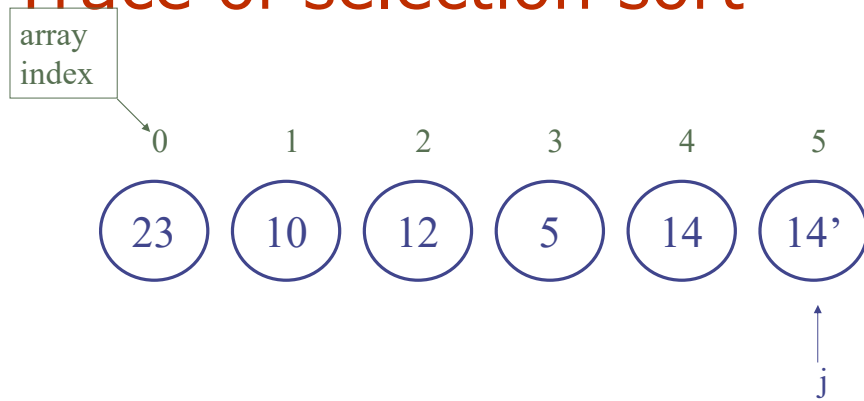


$i = 5$, first iteration of the outer loop

$j = 5$, $\text{pos_greatest} = 0$

First loop keeps the pos of greatest as index 0

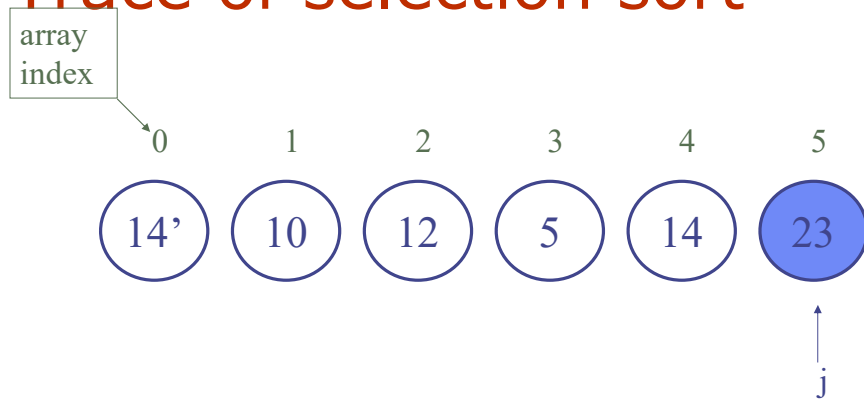
Trace of selection sort



$i = 5$, first iteration of the outer loop

swap element at $\text{pos_greatest} = 0$ to 5

Trace of selection sort

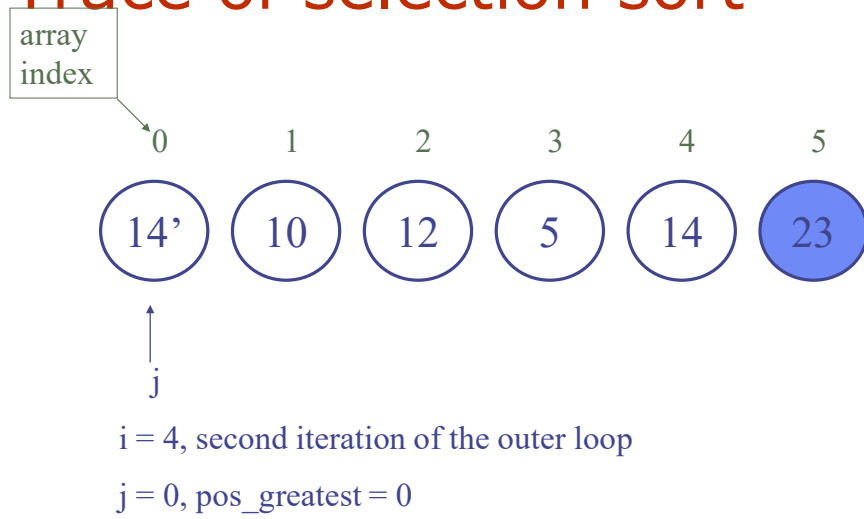


$i = 5$, first iteration of the outer loop

swap element at $\text{pos_greatest} = 0$ to 5

Then it does the swap of the greatest to the end element.

Trace of selection sort

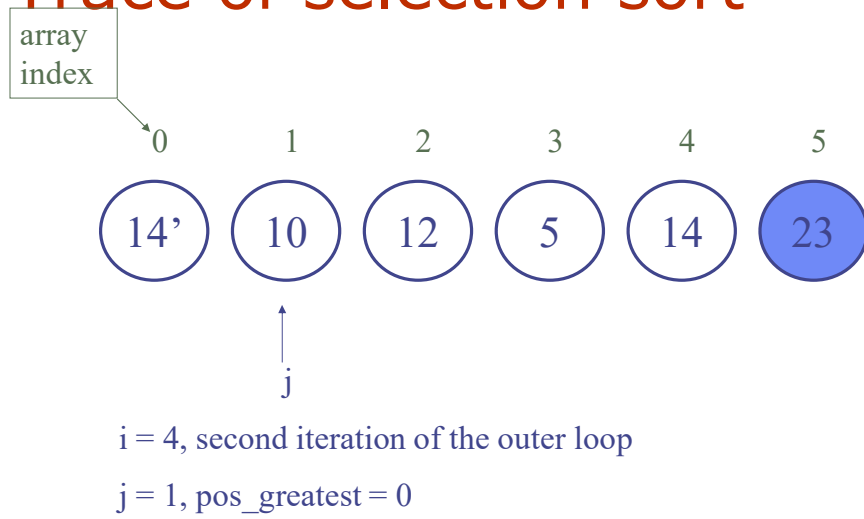


49

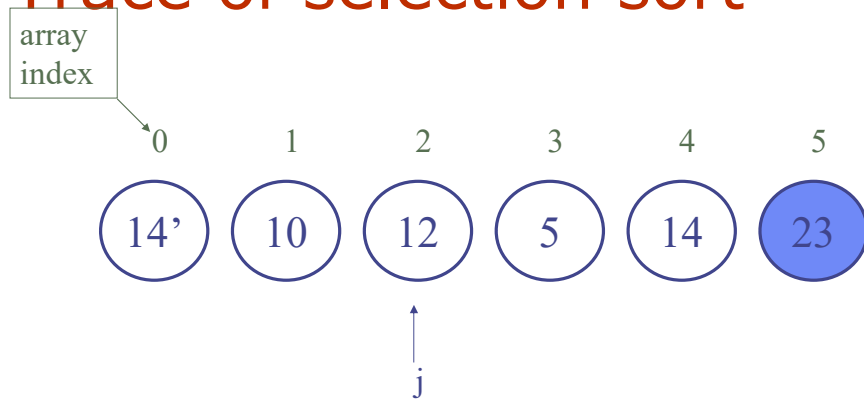
The next iterations are then similar.

So we just step through again quickly.

Trace of selection sort



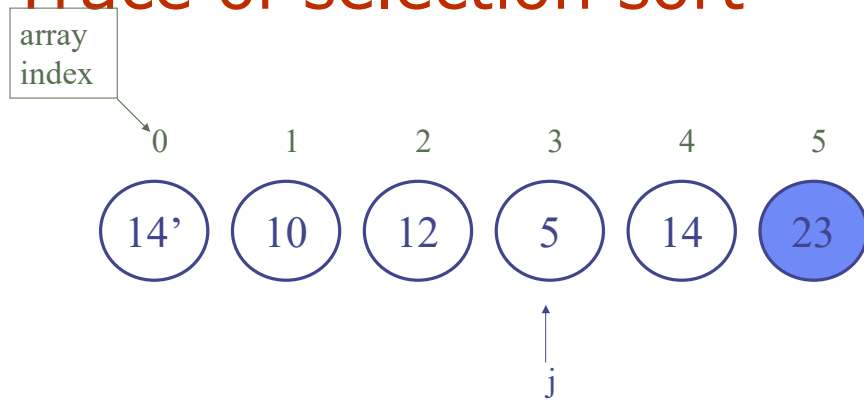
Trace of selection sort



$i = 4$, second iteration of the outer loop

$j = 2$, $\text{pos_greatest} = 0$

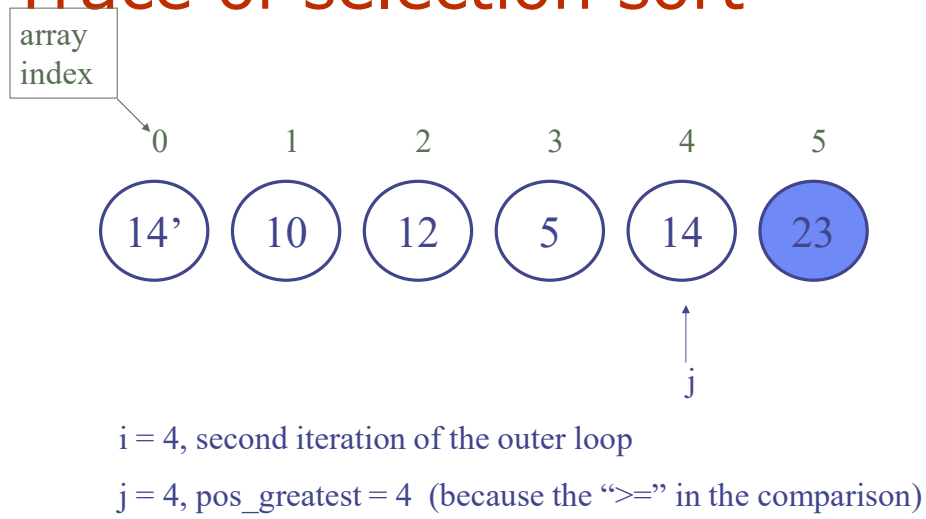
Trace of selection sort



$i = 4$, second iteration of the outer loop

$j = 3$, $\text{pos_greatest} = 0$

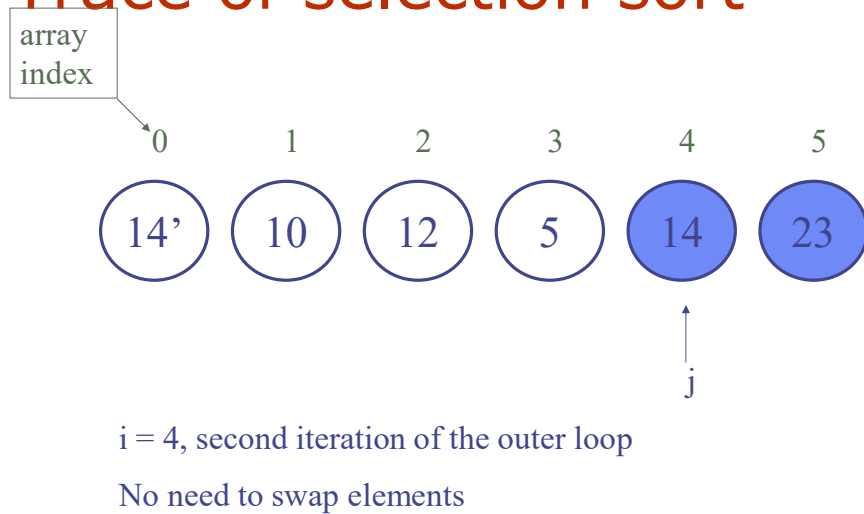
Trace of selection sort



53

The 14 on the right does not need to move. Note this can depend on details of the coding.

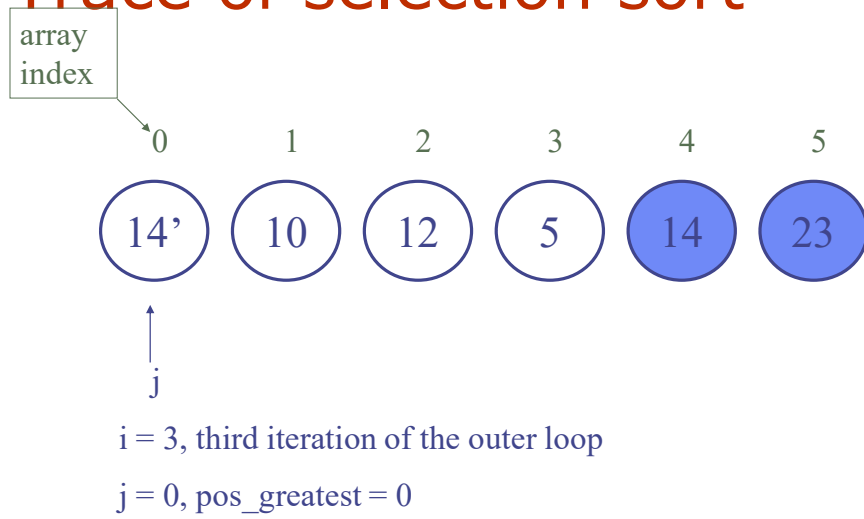
Trace of selection sort



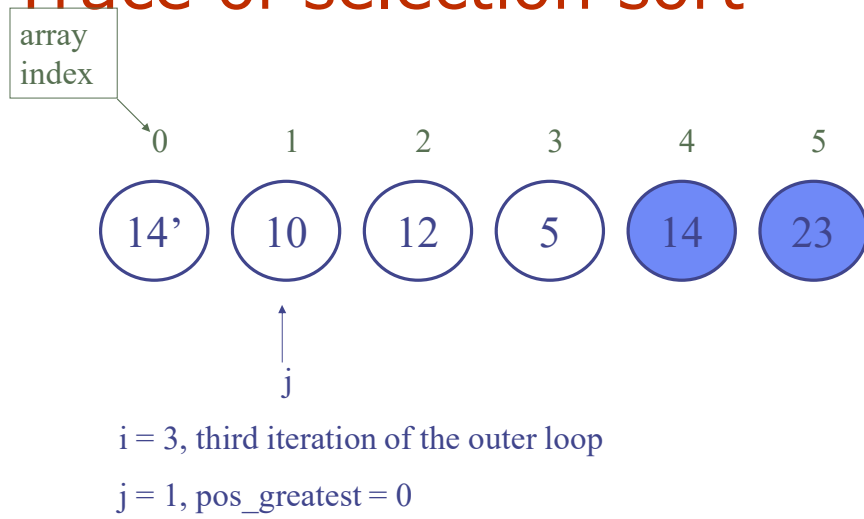
54

Again the elements marked in blue are finished with and will not need to move so the inner loop gets smaller.

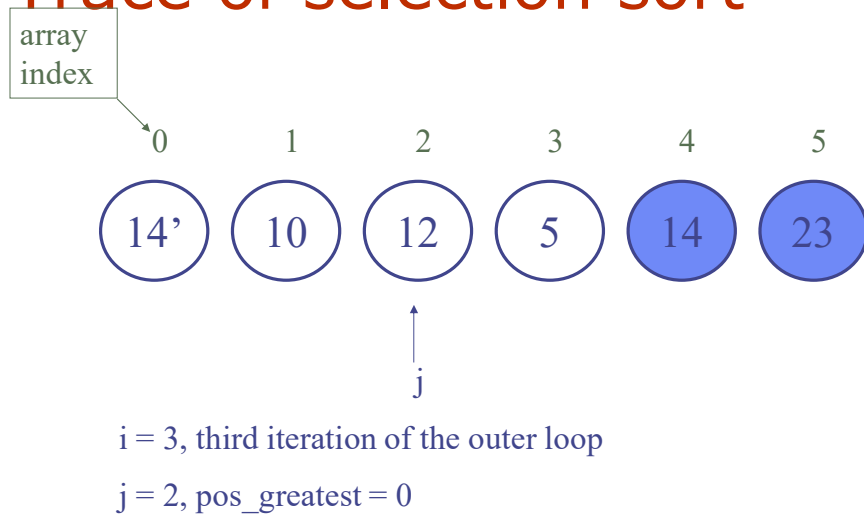
Trace of selection sort



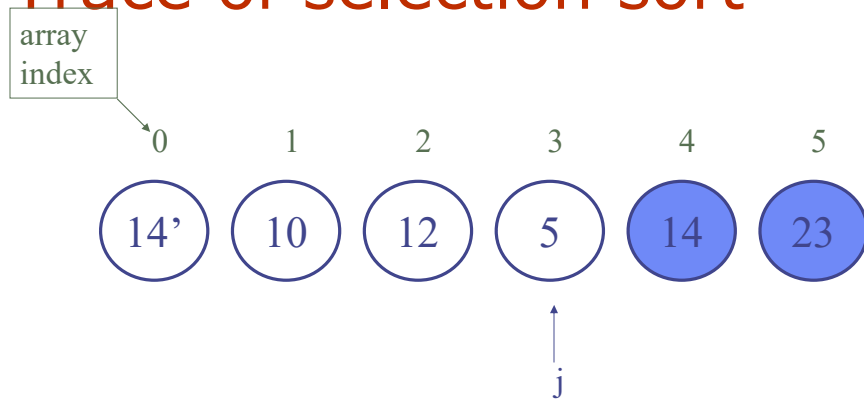
Trace of selection sort



Trace of selection sort



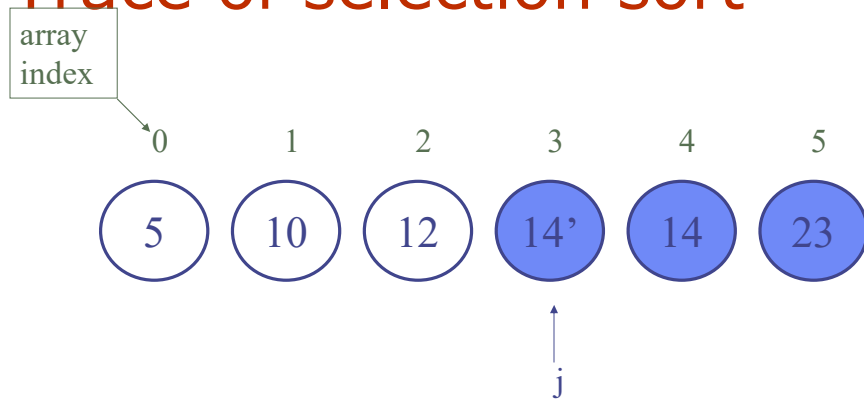
Trace of selection sort



$i = 3$, third iteration of the outer loop

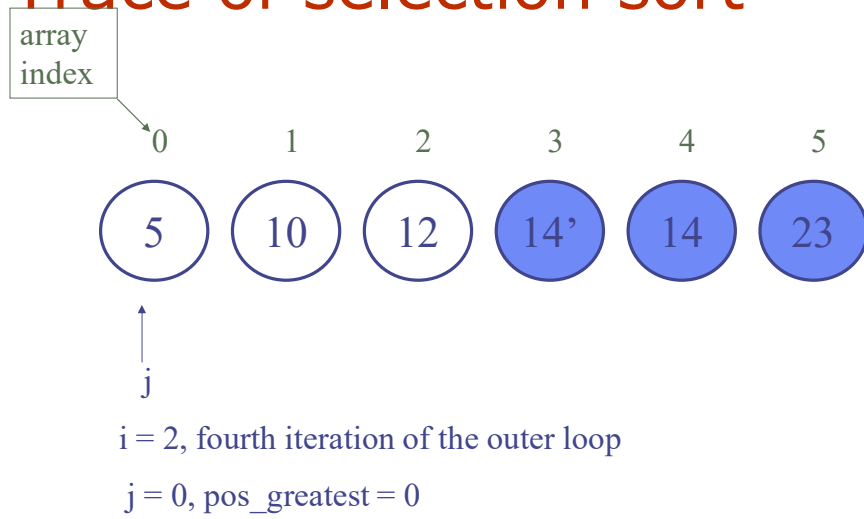
$j = 3$, $\text{pos_greatest} = 0$

Trace of selection sort

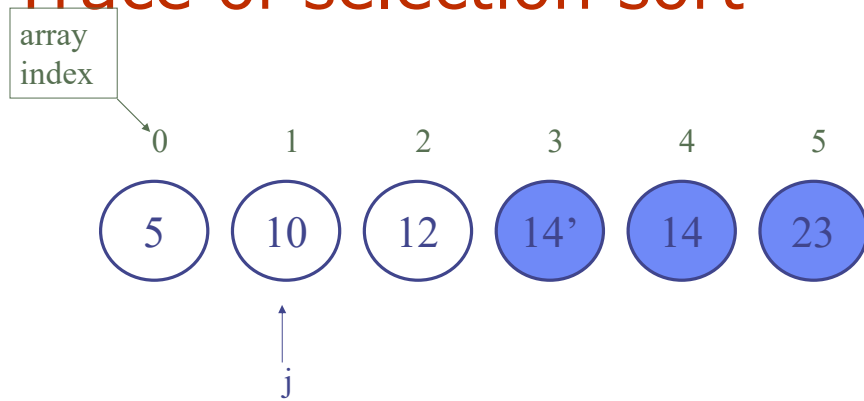


$i = 3$, third iteration of the outer loop
swap elements at pos_greatest and 3

Trace of selection sort



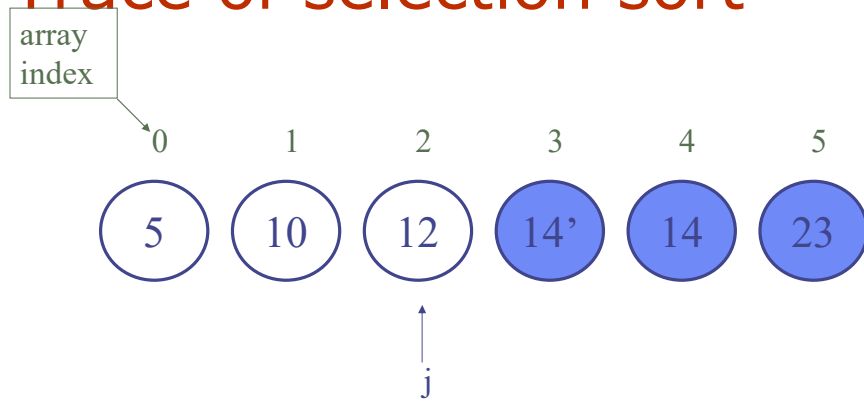
Trace of selection sort



$i = 2$, fourth iteration of the outer loop

$j = 1$, pos_greatest = 1 (changed!)

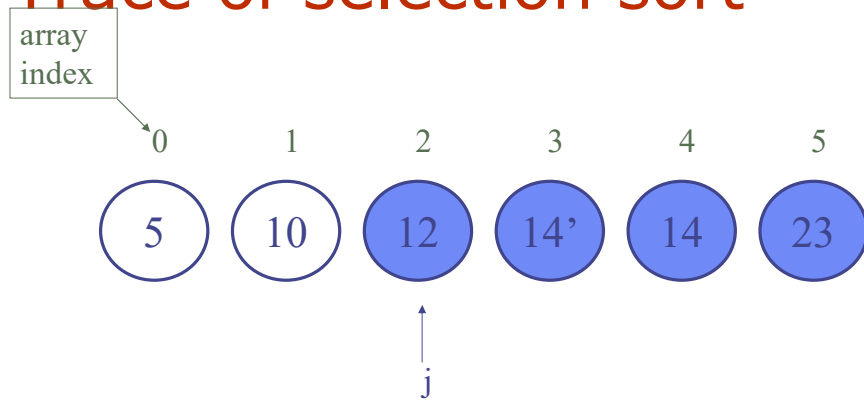
Trace of selection sort



$i = 2$, fourth iteration of the outer loop

$j = 2$, $\text{pos_greatest} = 2$ (changed again!)

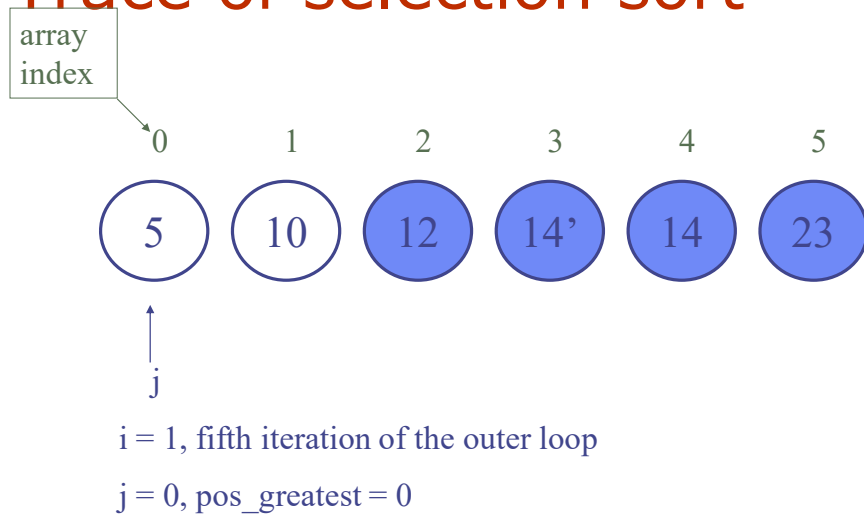
Trace of selection sort



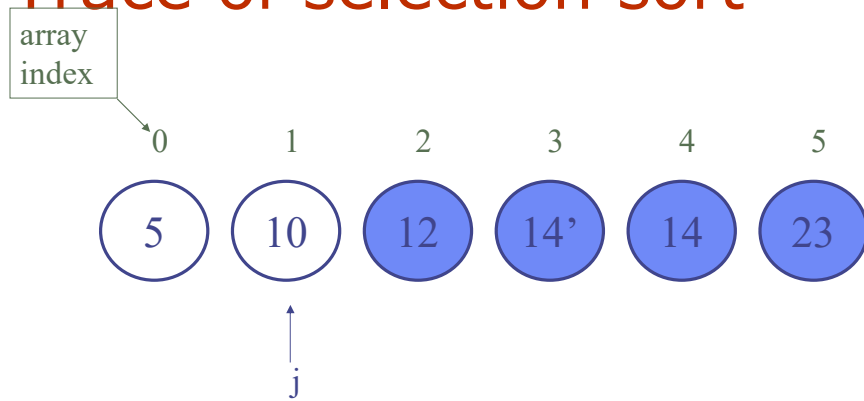
$i = 2$, fourth iteration of the outer loop

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

Trace of selection sort



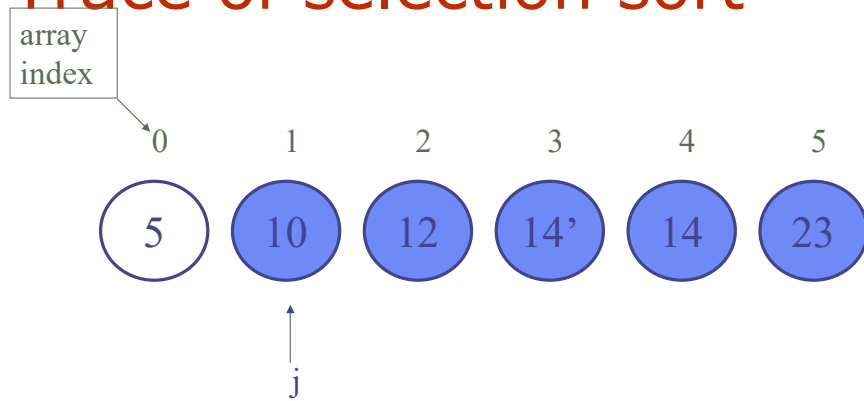
Trace of selection sort



$i = 1$, fifth iteration of the outer loop

$j = 1$, pos_greatest = 1 (changed)

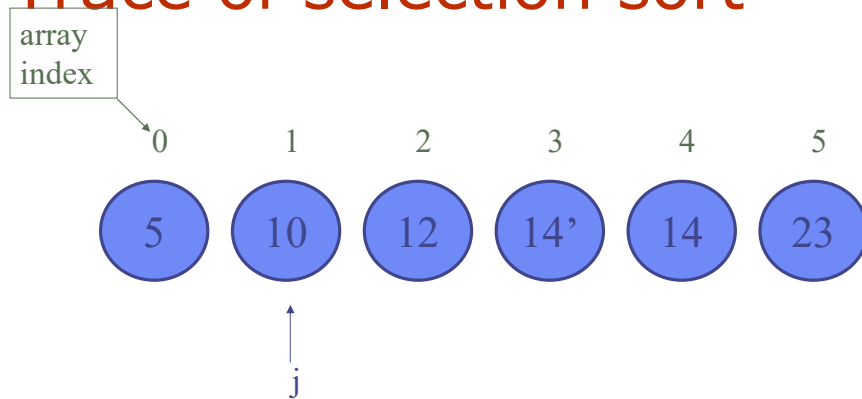
Trace of selection sort



$i = 1$, fifth iteration of the outer loop

swap element at pos_greatest with element at position i
(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?

67

As was obvious from earlier the 14 and 14' had been swapped and so this implementation is not stable.

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 $O(n^2)$

68

Analysis of the complexity is virtually the same as for bubble sort.

We skip the details, but make sure you understand this.

Since it is doing fewer swaps, then the big-Oh cannot get worse, and so it is still $O(n^2)$.

Doing the full details would be very similar to slide 36 and is left as an exercise.

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


69

For the last algorithm we do Insertion sort.

This has a different idea, namely to keep the front of the list sorted, and go through elements one at a time and insert them into the correct place.

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

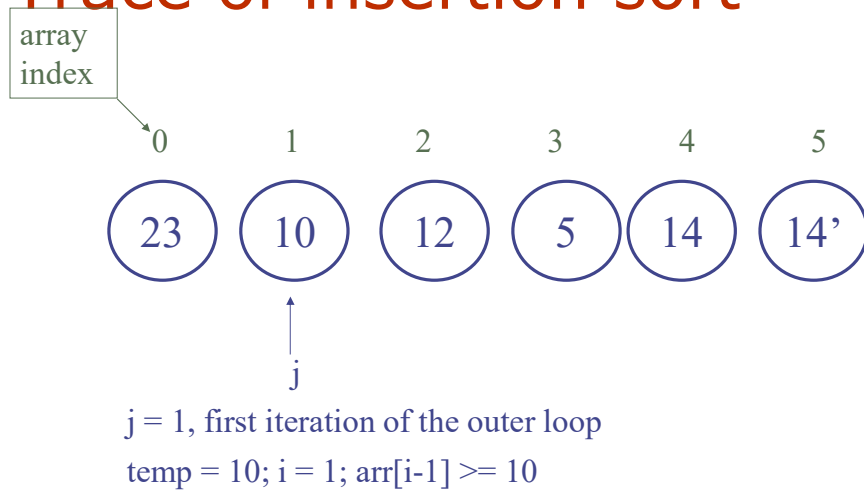
70

This time the outer loop works through in the forward direction.

The inner loop walks backwards and swaps either elements until it has found the correct place.

Again the easiest way to see this is from an animation.

Trace of insertion sort

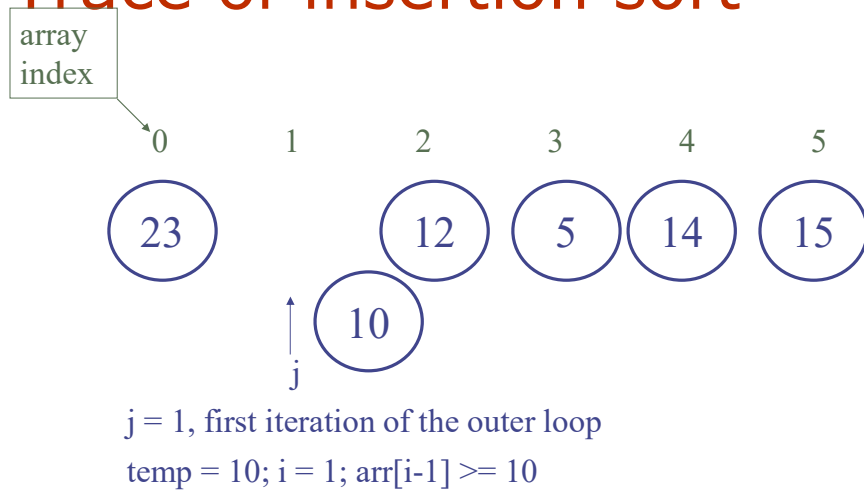


71

At first the start of the list is just the 23. B

But now consider the 10, and insert at the right place.

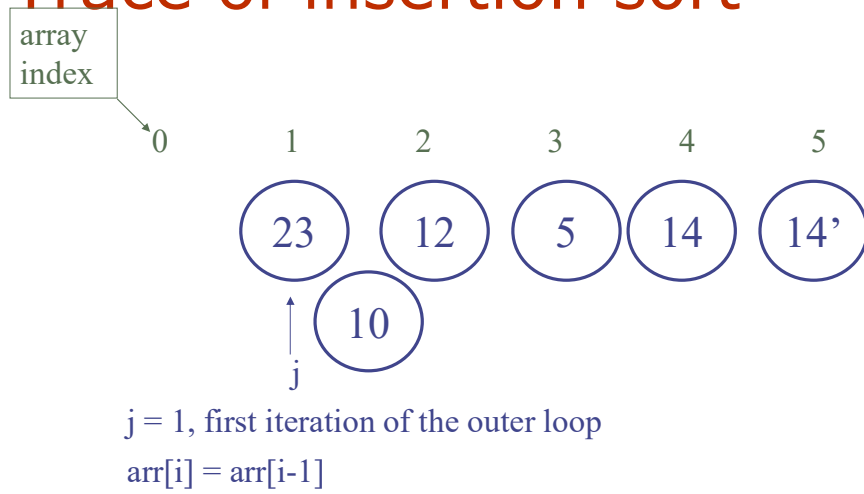
Trace of insertion sort



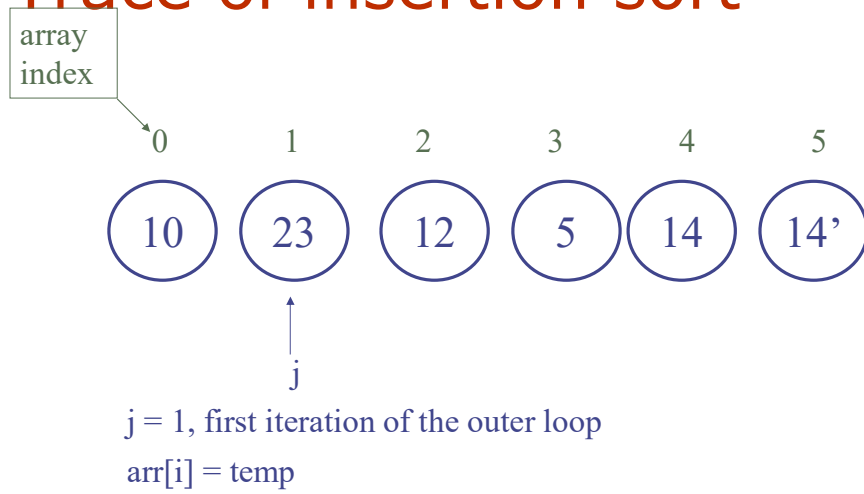
72

We pick up the 10 and move it backwards until it hits something smaller.

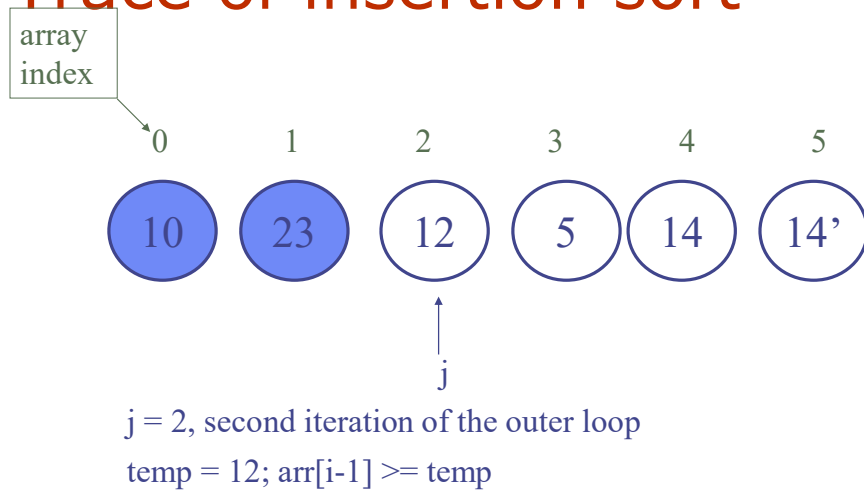
Trace of insertion sort



Trace of insertion sort



Trace of insertion sort

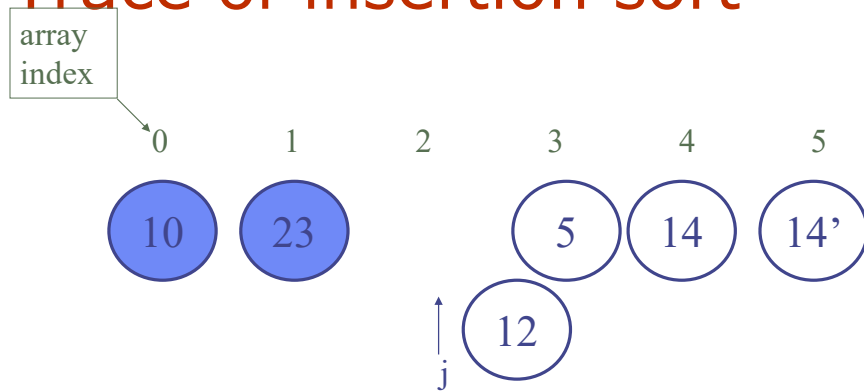


75

The blue nodes are now the front of the list.

Now we pick up the 12 and step backwards to insert.

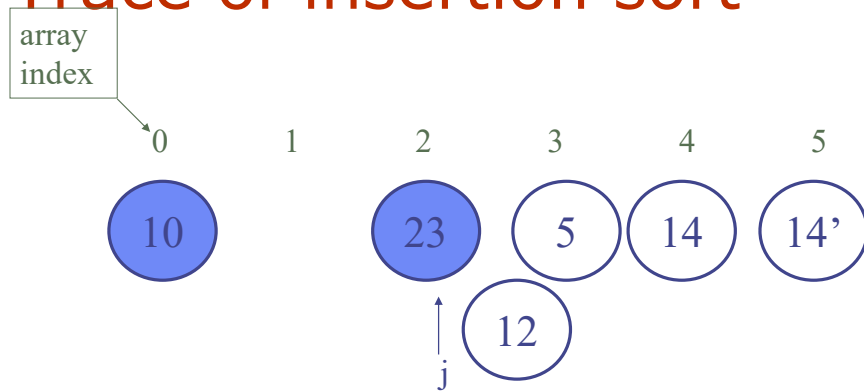
Trace of insertion sort



$j = 2$, second iteration of the outer loop

$\text{temp} = 12; \text{arr}[i-1] \geq \text{temp}$

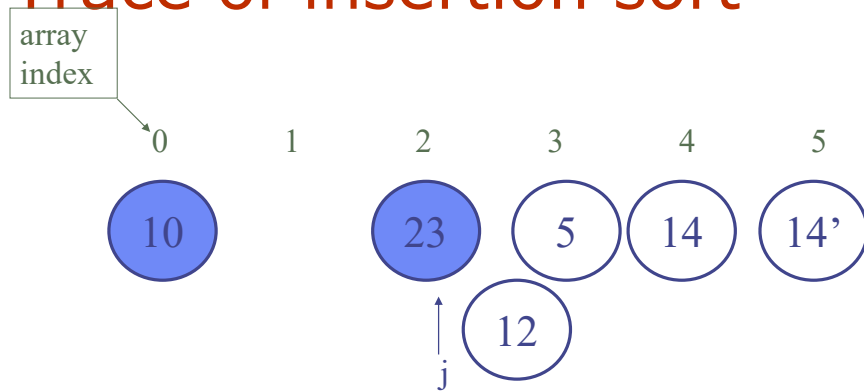
Trace of insertion sort



$j = 2$, second iteration of the outer loop

$arr[i-1] = arr[i]$

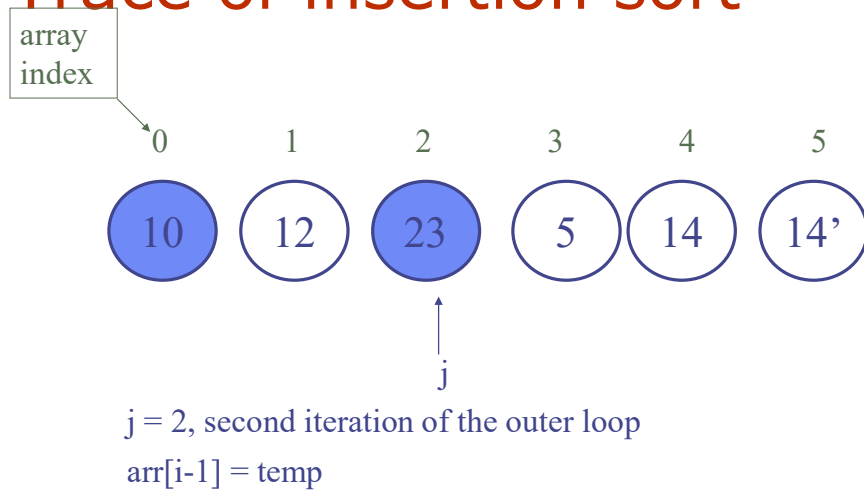
Trace of insertion sort



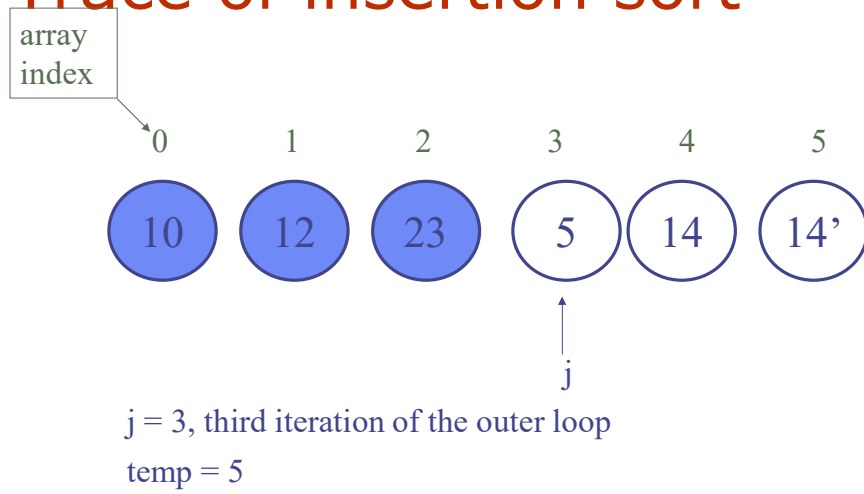
j = 2, second iteration of the outer loop

arr[i-1] < temp

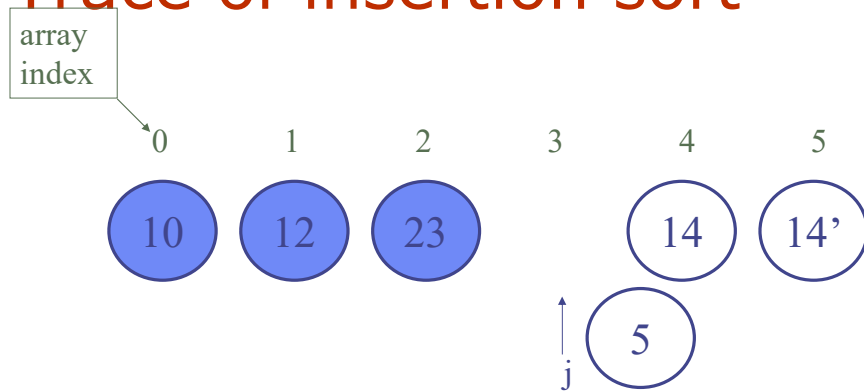
Trace of insertion sort



Trace of insertion sort



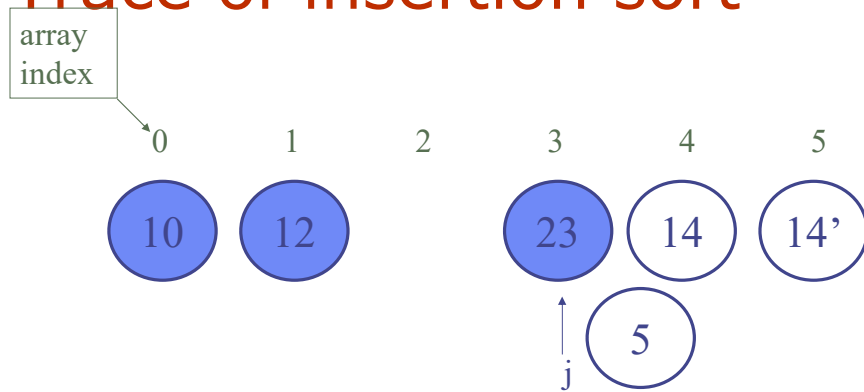
Trace of insertion sort



$j = 3$, third iteration of the outer loop

$\text{arr}[i-1] \geq \text{temp}$

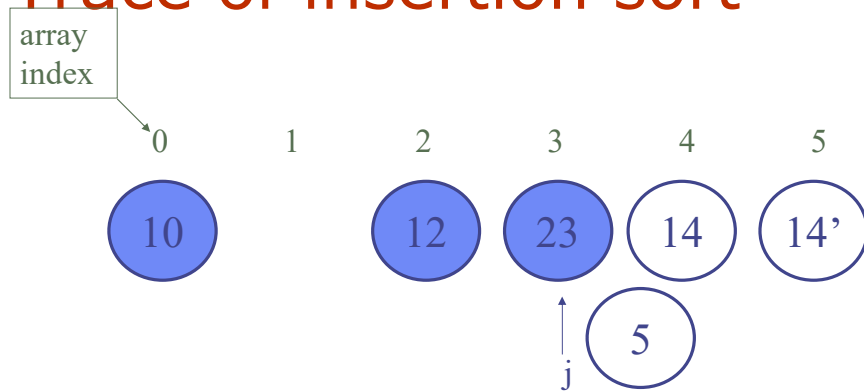
Trace of insertion sort



$j = 3$, third iteration of the outer loop

$\text{arr}[i-1] \geq \text{temp}$

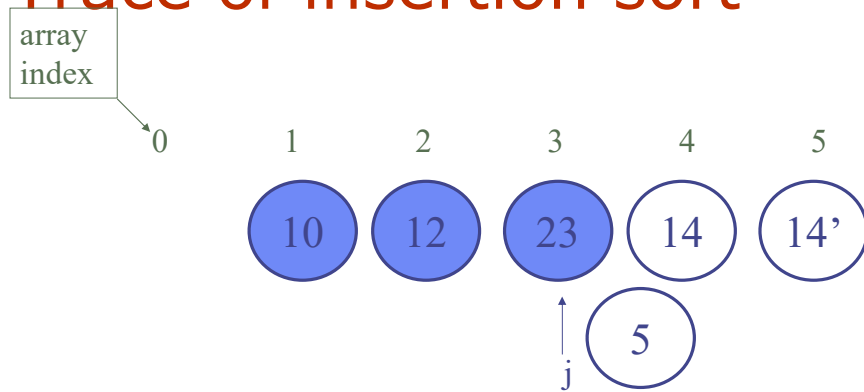
Trace of insertion sort



$j = 3$, third iteration of the outer loop

$\text{arr}[i-1] \geq \text{temp}$

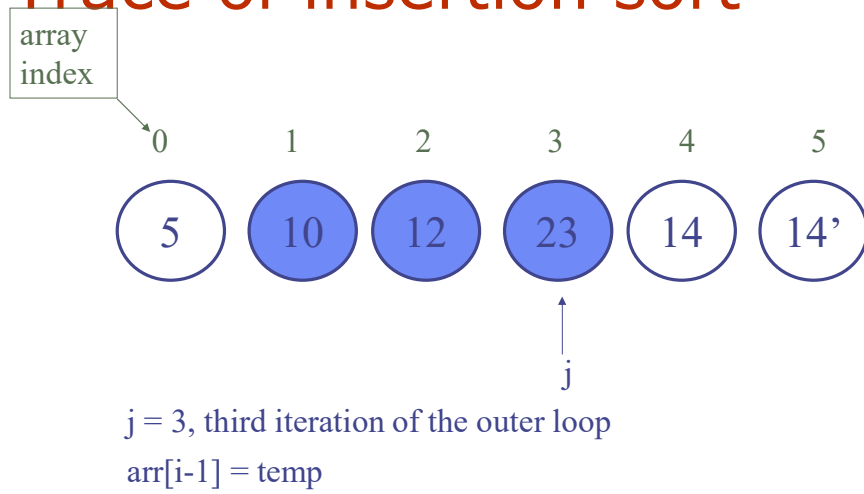
Trace of insertion sort



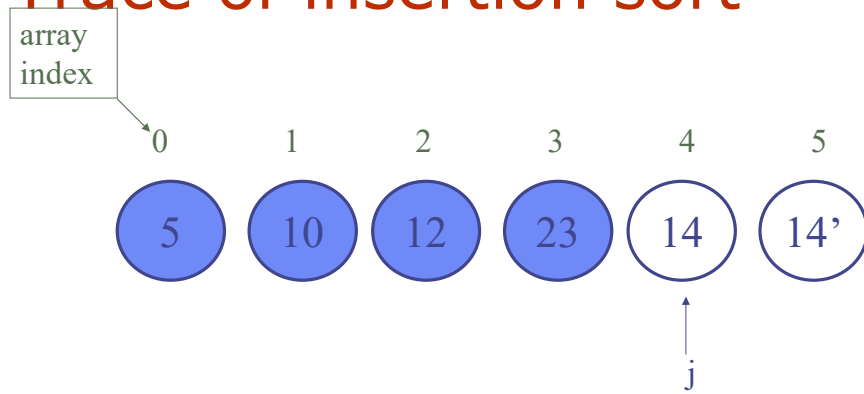
$j = 3$, third iteration of the outer loop

$\text{arr}[i-1] \geq \text{temp}$

Trace of insertion sort



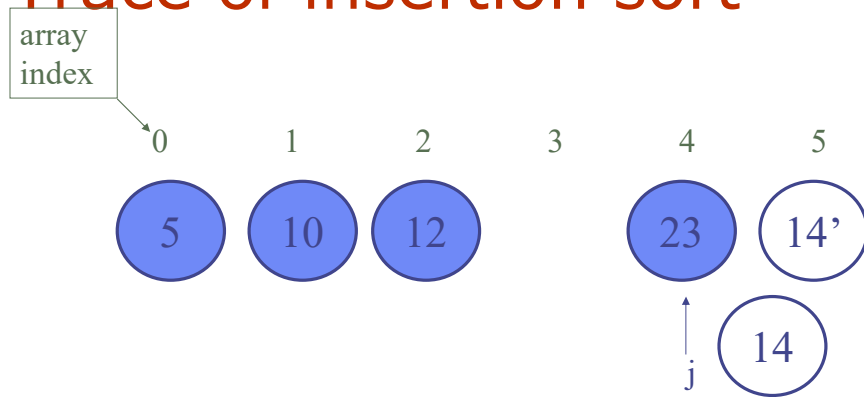
Trace of insertion sort



j = 4, fourth iteration of the outer loop

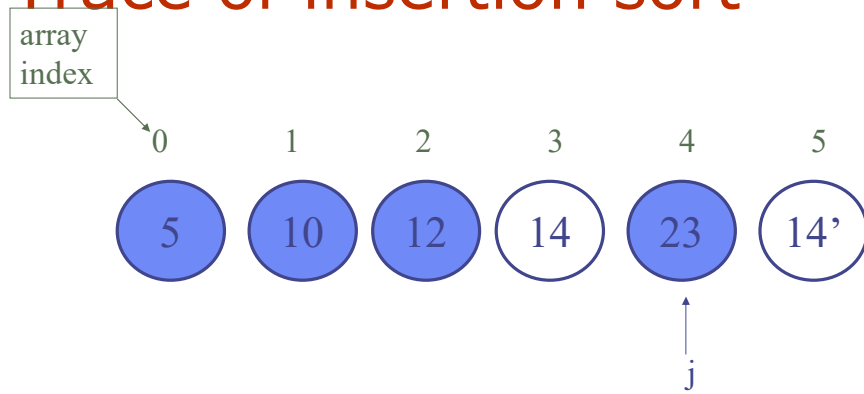
temp = 14

Trace of insertion sort



$j = 4$, fourth iteration of the outer loop
 $\text{arr}[i-1] \geq \text{temp}$

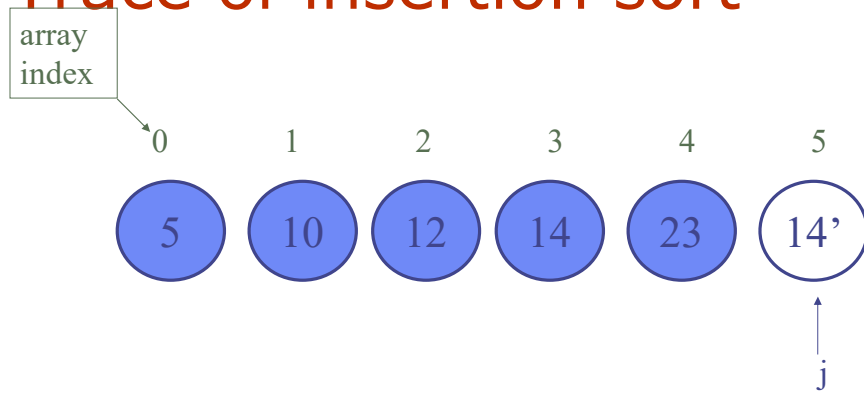
Trace of insertion sort



$j = 4$, fourth iteration of the outer loop

$arr[i-1] = temp$

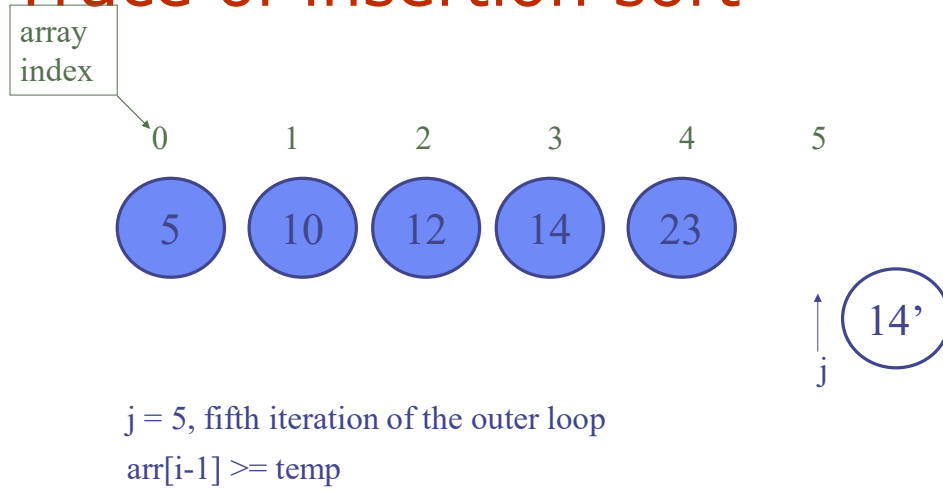
Trace of insertion sort



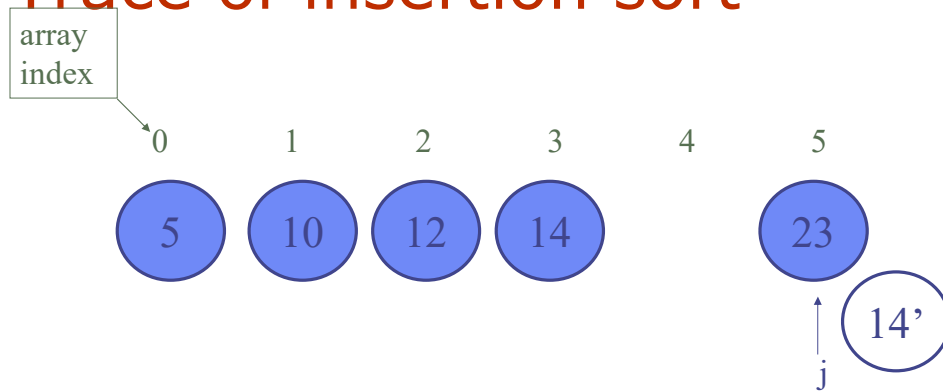
j = 5, fifth iteration of the outer loop

temp = 15

Trace of insertion sort

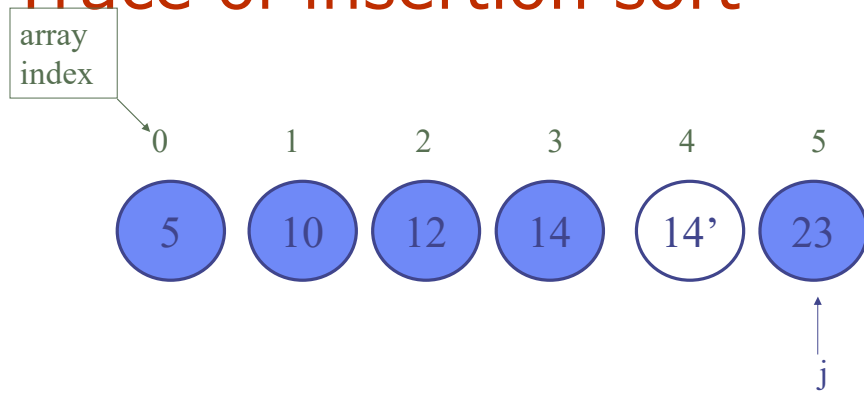


Trace of insertion sort



$j = 5$, fifth iteration of the outer loop
 $arr[i-1] \geq temp$

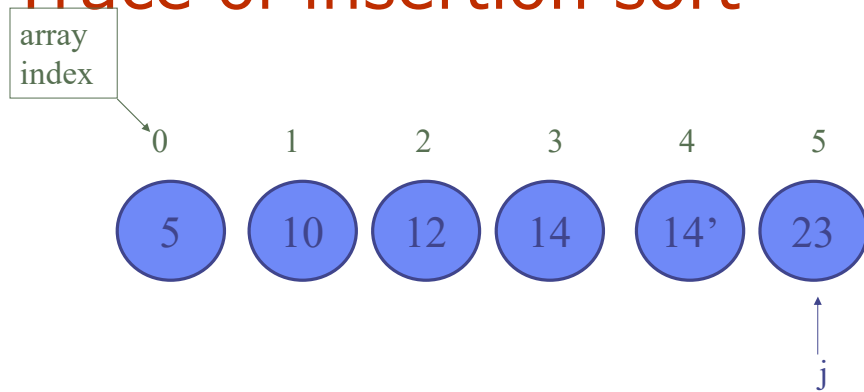
Trace of insertion sort



$j = 5$, fifth iteration of the outer loop

$\text{arr}[i-1] = \text{temp}$

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 $O(n)$

94

What about the complexity?

It is again very similar, there are two nested loops, and to the number of comparisons will be the same.

If the array is already sorted then the insertion will not actually need to walk all the list and will become $O(n)$.

“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?

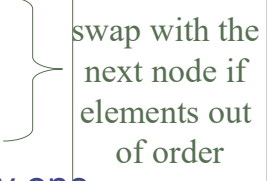
- Is bubble sort also workable for linked lists?
- Bubble sort is just as efficient (or rather inefficient) on linked lists.
 - We can easily bubble sort even for a singly linked list.

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

The code is hard to read but has a similar structure.

The border is the border between the start portion that is not yet sorted, and the end portion that is sorted.

The border starts at the right, and then gets moved towards the left.

The inner loop scans through from the head up until the border.

Doing swaps as needed.

Bubble sort of a linked list

In the previous slide

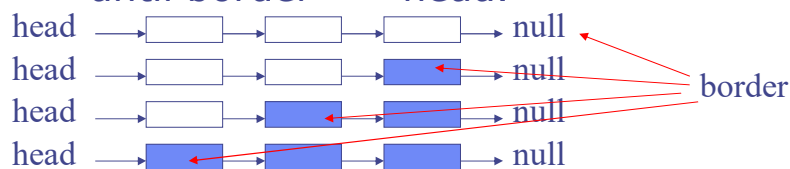
```
E element v = current.element;  
current.element = current.next.element;  
current.next.element = v;
```

- 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.

Note that the swap is not done using surgery on the links, but just using swapping of the contents.

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.



Simple Sorting Algorithms

102

Again the pattern is exactly the same.

On each scan the right hand portion of “in the final place” is increased.

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;
```

103

The selection sort is basically identical.

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...):

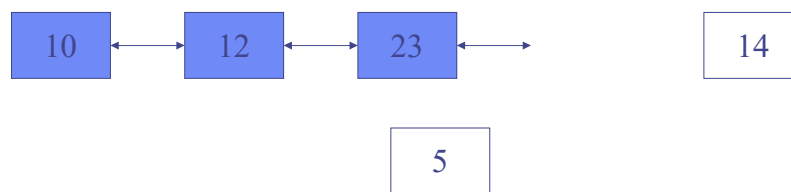


104

For insertion sort it can be harder as the insertion meant walking backwards to find the right 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...):

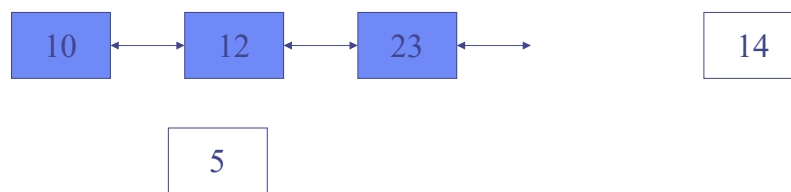


105

Again the idea is exactly the same of inserting a new element into list that is already sorted.

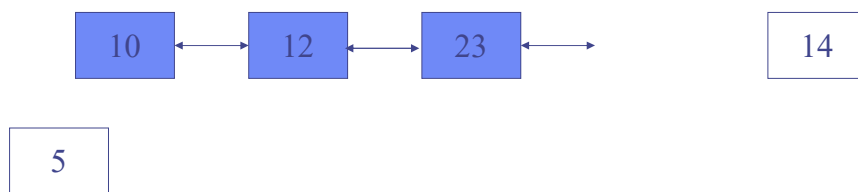
Insertion sort on linked lists

- 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

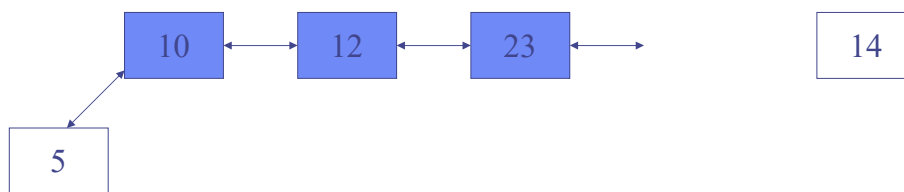
- This is a suitable sorting method for doubly linked lists
- We can just insert a node in a sorted portion of linked list in constant time, don't need to shift other nodes to make space for it (but need to find the place...):



We carry on walking backwards

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...):

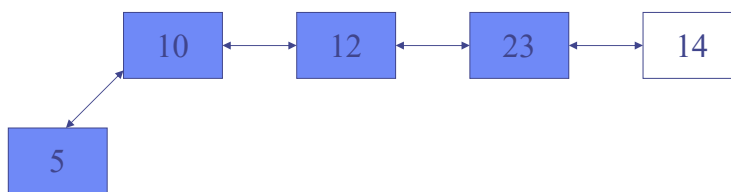


108

Since we are doing lists then we do not need to actually move the nodes, but will need to readjust the links between them.

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.