CI583: Data Structures and Operating Systems

Three simple sorting algorithms

Simple sorting algorithms

The next algorithms we consider are three of the simplest algorithms for sorting data. We will see that they are all quite inefficient and would not be our first choice if we needed to sort large amounts of data.

However, they are a good place to start, mainly because they are simple to describe and implement. What's more, they are "good enough" in some circumstances and use very little additional memory as they run (i.e. additional to the data being sorted).

One of them, insertion sort is often used as part of a more sophisticated algorithm (quicksort) we will encounter in a later lecture.

Simple sorting algorithms

We saw in Week One that we can find an element in a sorted dataset very efficiently, and the most common reason to keep a dataset sorted is for efficient search and retrieval.

Relational databases manage indexes to do just this. An index is a sorted structure containing column data that often needs to be searched, such as a primary key. It takes longer to insert to a table with indexes on it, but searching is much faster.

Simple sorting algorithms

After this lecture, it will benefit you to experiment with the applets on studentcentral that visualise the various algorithms. Do this carefully and with the slides handy at the same time.

Bubble sort

Bubble sort is the simplest, easiest to describe, and just about the least efficient sorting algorithm there is. There are several variations of bubble sort, and we present a simple version that could be optimised in various ways.

The general idea is for larger values to move ("bubble") up through the collection until everything is in the right place. Each element, e, is compared with its neighbour, e'. If $e > e'$, then their places are swapped. We continue in this way until we reach the end of the collection, which is the end of the first pass.

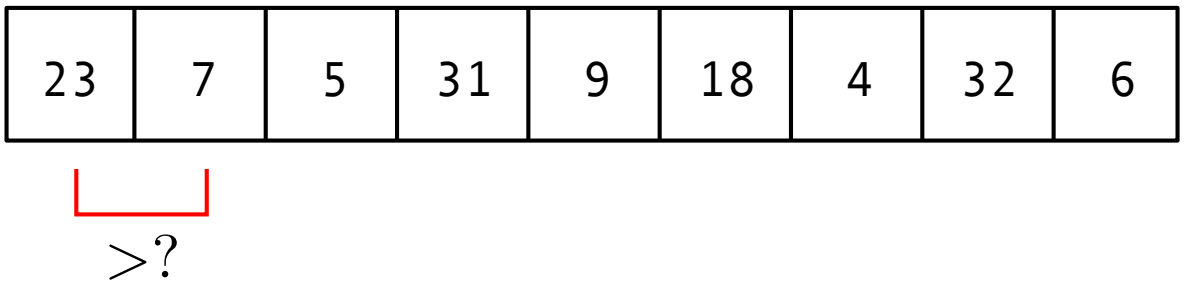
Bubble sort

After the first pass, the last element in the collection is the largest one, and that position is sorted.

On the next pass, we only need to consider $n - 1$ values, where n is the size of the input, and on the one after that, $n - 2$, and so on. If, on any given pass, there are no swaps then the data is sorted and we stop.

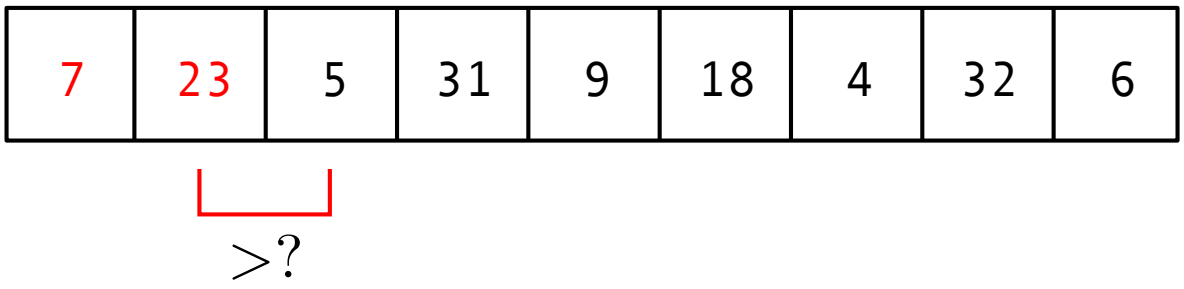
Bubble sort

One pass of bubble sort:



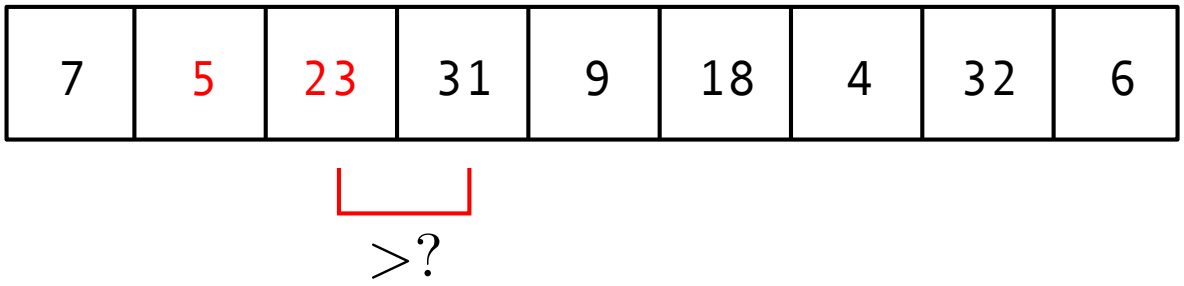
Bubble sort

One pass of bubble sort:



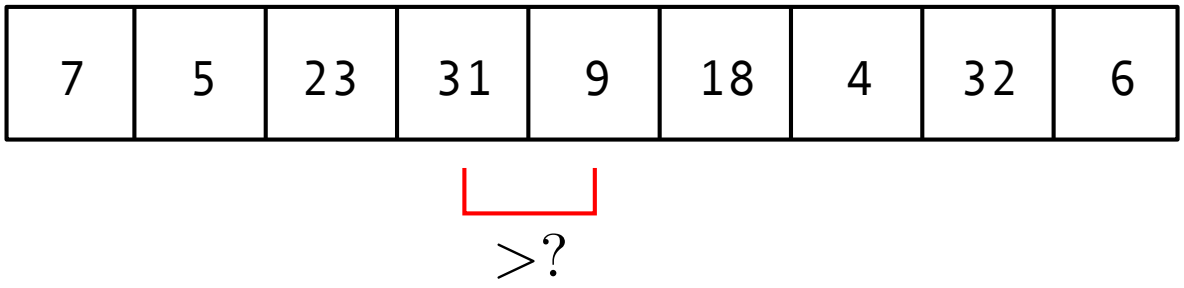
Bubble sort

One pass of bubble sort:



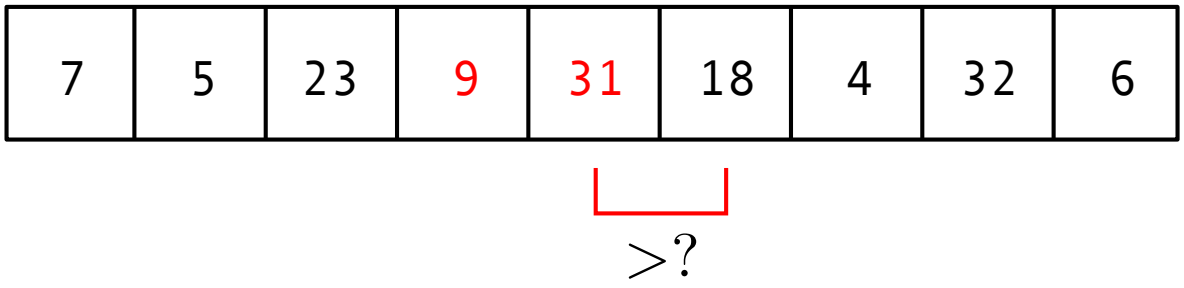
Bubble sort

One pass of bubble sort:



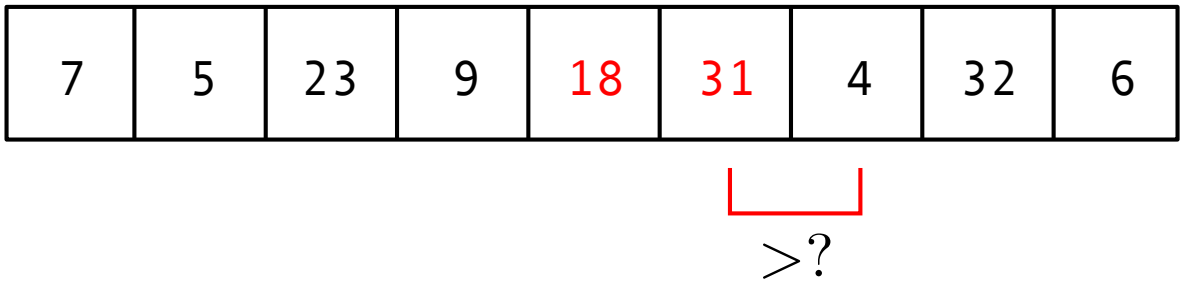
Bubble sort

One pass of bubble sort:



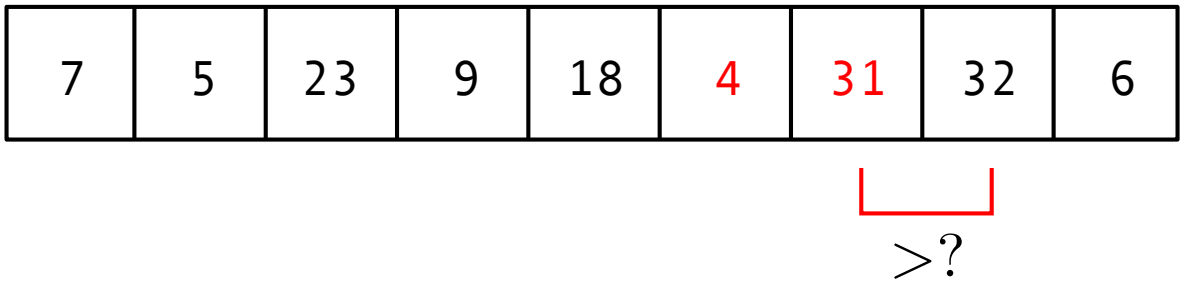
Bubble sort

One pass of bubble sort:



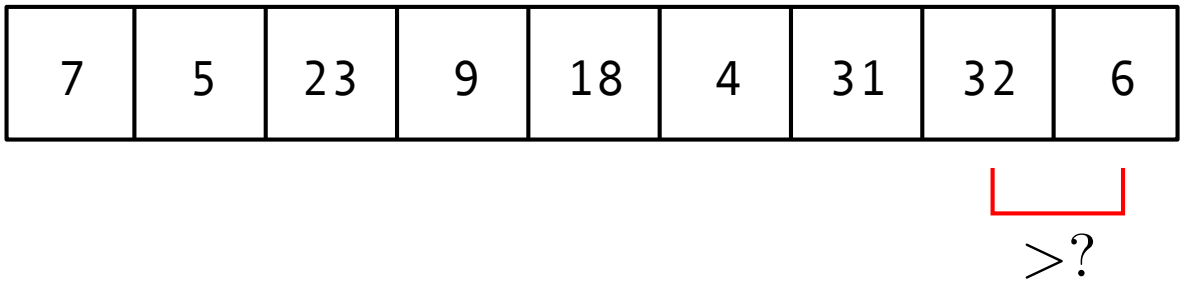
Bubble sort

One pass of bubble sort:



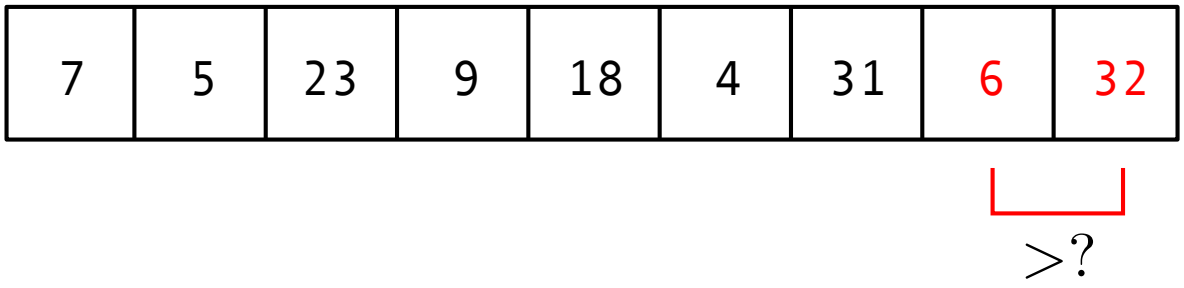
Bubble sort

One pass of bubble sort:



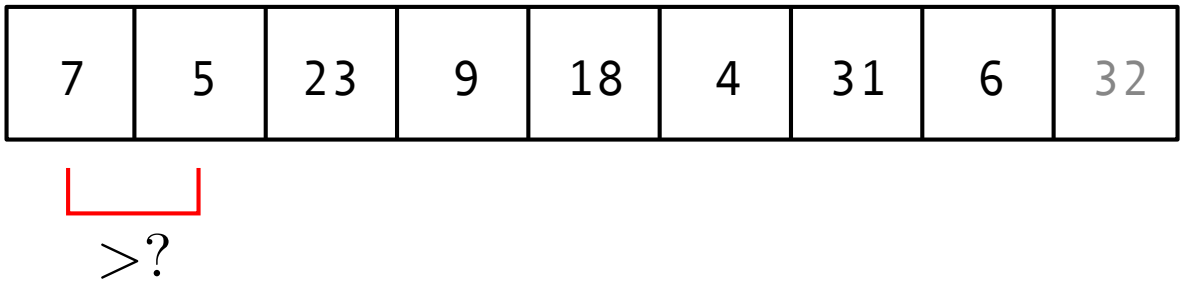
Bubble sort

One pass of bubble sort:



Bubble sort

One pass of bubble sort:



Bubble sort

(This is more low-level than pseudo-code should normally be, but I think it's the clearest way to present the algorithm.)

```
procedure BubbleSort(arr, n)          ▷ n is the length of arr
    swapped          ▷ boolean – did we make any swaps this pass?
    for i from n − 1 down to 0 do                  ▷ loop backwards
        swapped ← false
        for j from 0 up to i do                    ▷ loop forwards
            if arr[j] > arr[j + 1] then
                swap(arr[j], arr[j + 1])
                swapped ← true
            end if
        end for
        if swapped = false then
            break
        end if
    end for
end procedure
```

On the first pass, bubble sort carries out $n - 1$ comparisons. In the best case, there are no swaps and the algorithm terminates. We will concentrate on the worst case.

The worst case is when the input was in reverse order: at the beginning of the first pass, the largest element is in first place and is swapped all the way to the end. At the beginning of the second pass, the second largest element is in first place, etc.

So, the first pass does $n-1$ comparisons, the second $n-2$, and so on.

Let $W(n)$ be the worst case for n elements. Then

$$\begin{aligned} W(n) &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)n}{2} \\ &= \frac{n^2-n}{2} \\ &\approx \frac{1}{2}n^2 \\ &= O(n^2) \end{aligned}$$

Bubble sort
Complexity

As a rule of thumb, note that that any bubble sort implementation will have inner and outer loops over the same collection, or some equivalent structure.

When we see this pattern we can take it to mean $O(n^2)$, as we are carrying out an $O(n)$ task n times. The simplest sorting algorithms are all in this order.

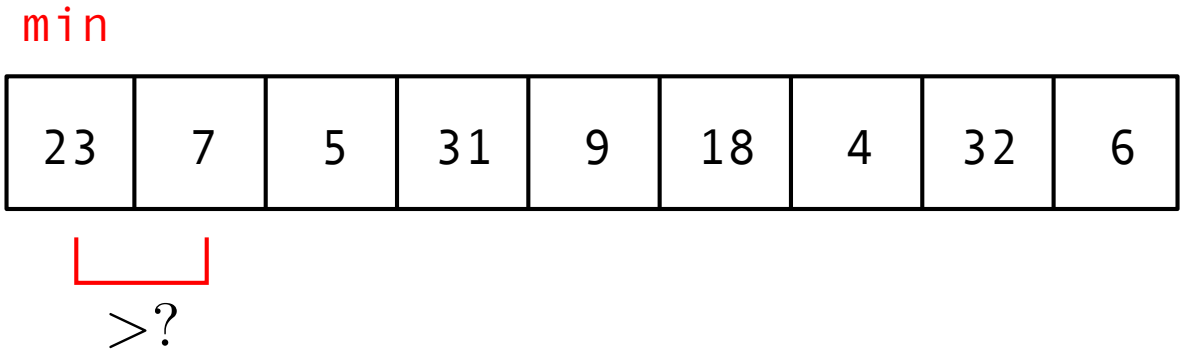
Selection sort

The idea behind selection sort is that we look through the whole collection for the smallest element, then put that in first place. On the next pass, we do the same thing but start with the second element, and so on.

Selection sort requires the same number of comparisons as bubble sort, but the number of swaps required is $O(n)$. In bubble sort the same element may be moved many times, but in selection sort each element is likely to be moved much less often.

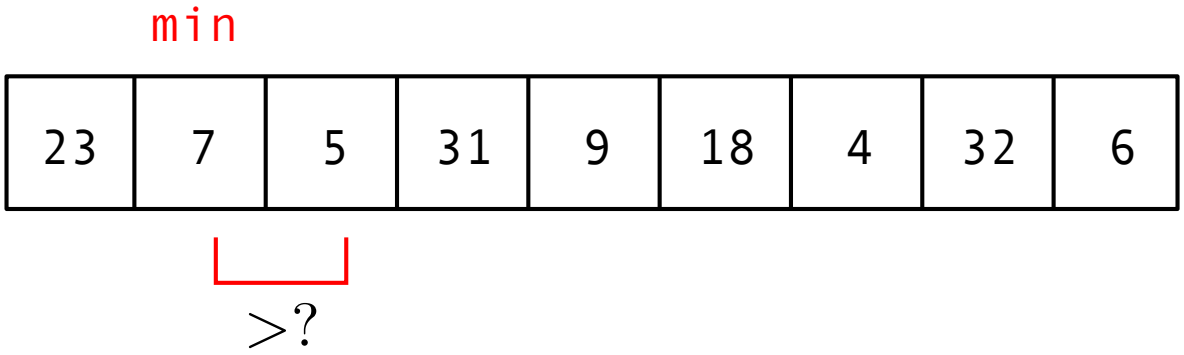
Selection sort

One pass of selection sort:



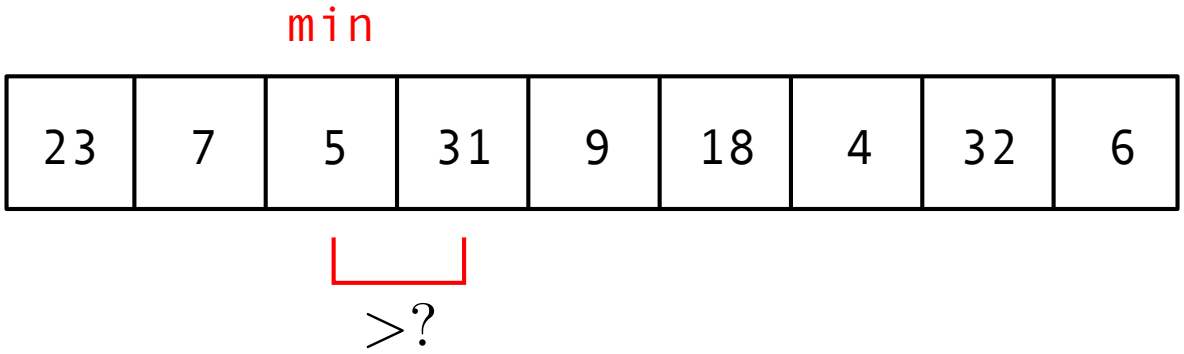
Selection sort

One pass of selection sort:



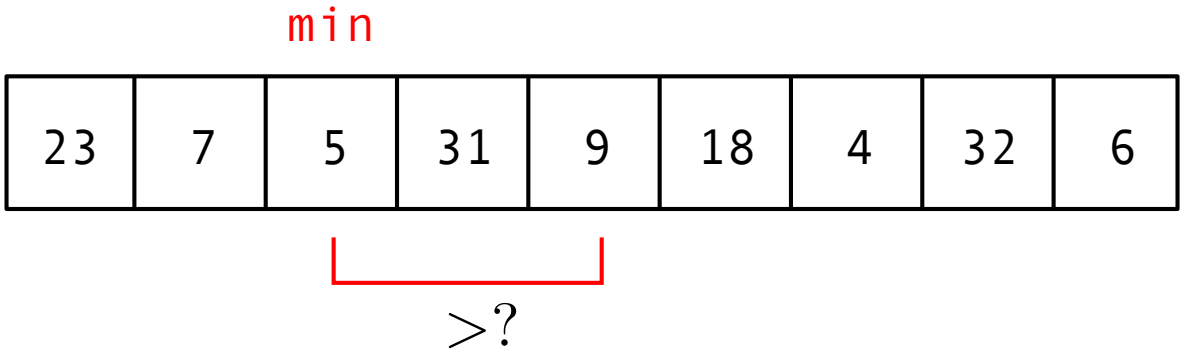
Selection sort

One pass of selection sort:



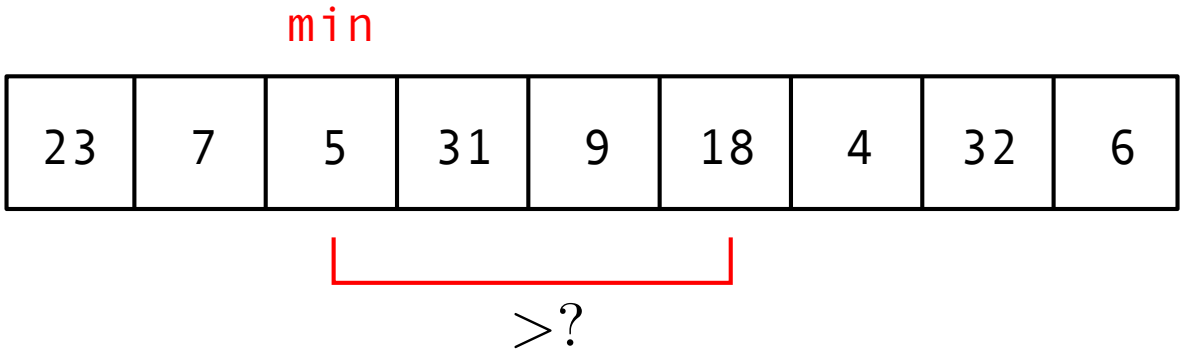
Selection sort

One pass of selection sort:



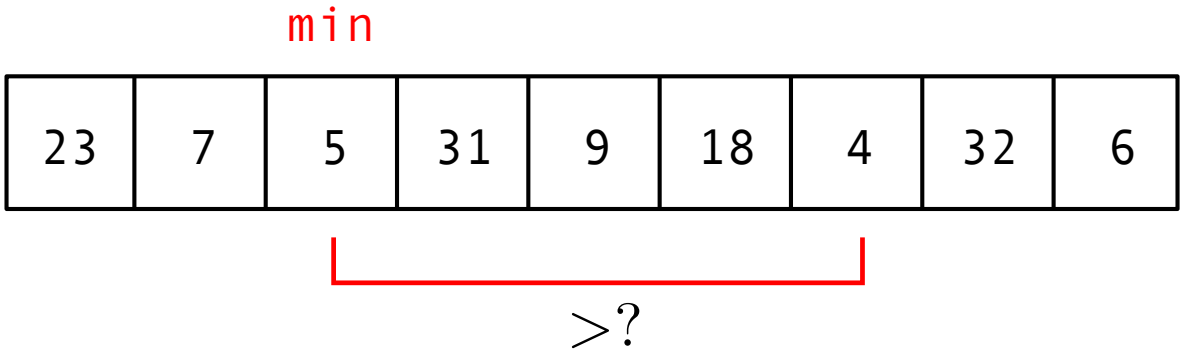
Selection sort

One pass of selection sort:



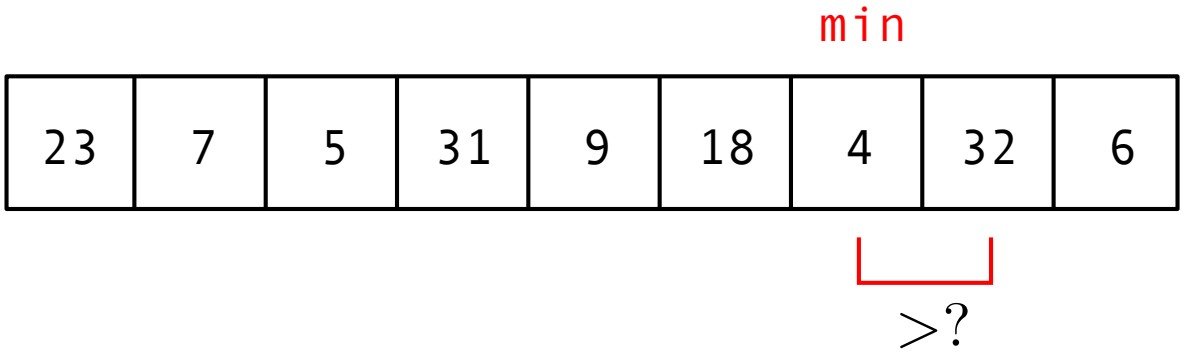
Selection sort

One pass of selection sort:



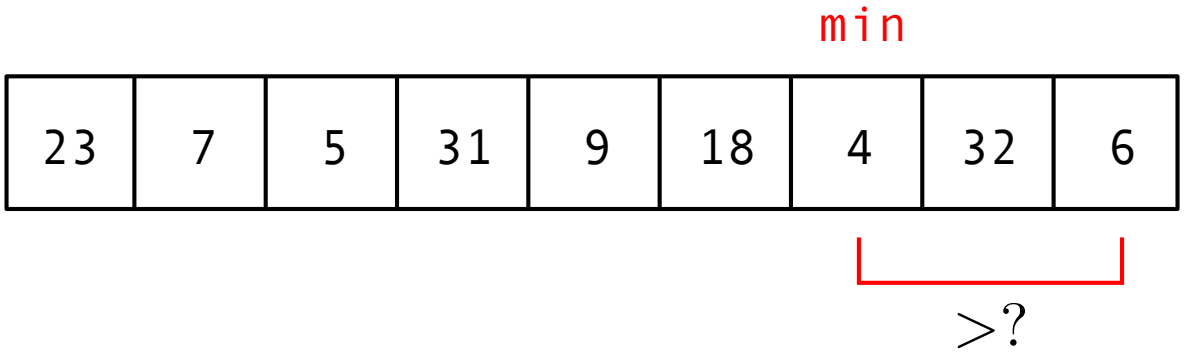
Selection sort

One pass of selection sort:



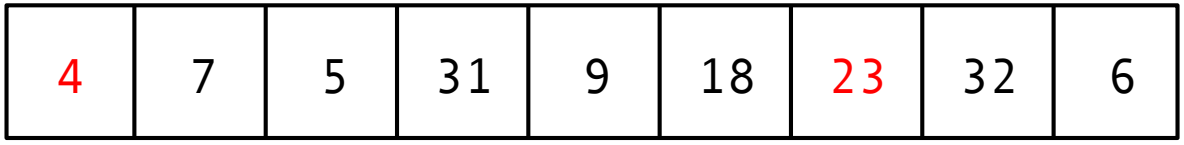
Selection sort

One pass of selection sort:



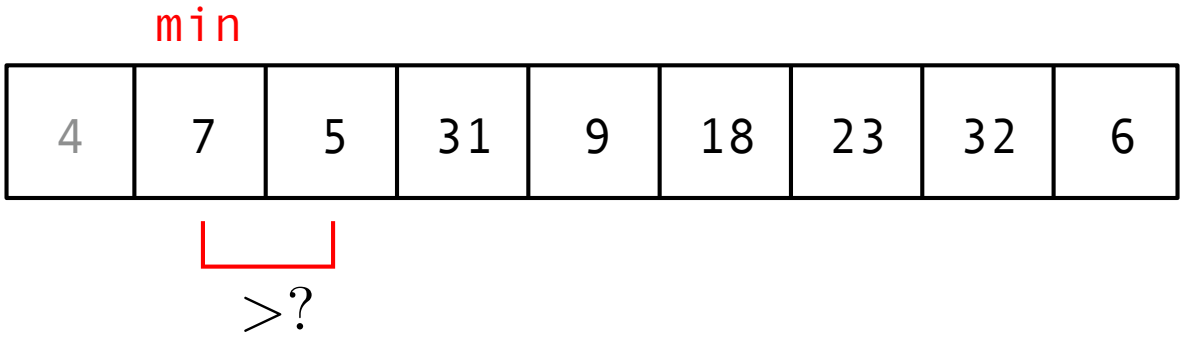
Selection sort

One pass of selection sort:



Selection sort

One pass of selection sort:



Selection sort

```
procedure SelectionSort(arr, n)        ▷ n is the length of arr
    for out from 0 up to n − 2 do   ▷ stop at one before the last element
        min ← out
        for in from out + 1 up to n − 1 do
            if arr[min] > arr[in] then
                min ← in
            end if
        end for
        swap(out, min)
    end for
end procedure
```

In the worst case, we can see that selection sort is $O(n^2)$, by the same reasoning as for bubble sort.

However, it was easy to optimise the best case for bubble sort by detecting that the input was already sorted and ending after the first pass. Since we don't necessarily compare every element to each other in selection sort, this isn't so easy.

Still, making fewer swaps gives selection sort better average performance.

Insertion sort

Most of the time, insertion sort has the best performance of the simple sorts we're looking at today. It is still $O(n^2)$ but about twice as fast as bubble sort.

The idea is to start by sorting the first two elements then, as we move along the collection, to insert each element in the right place in the sorted part of the collection.

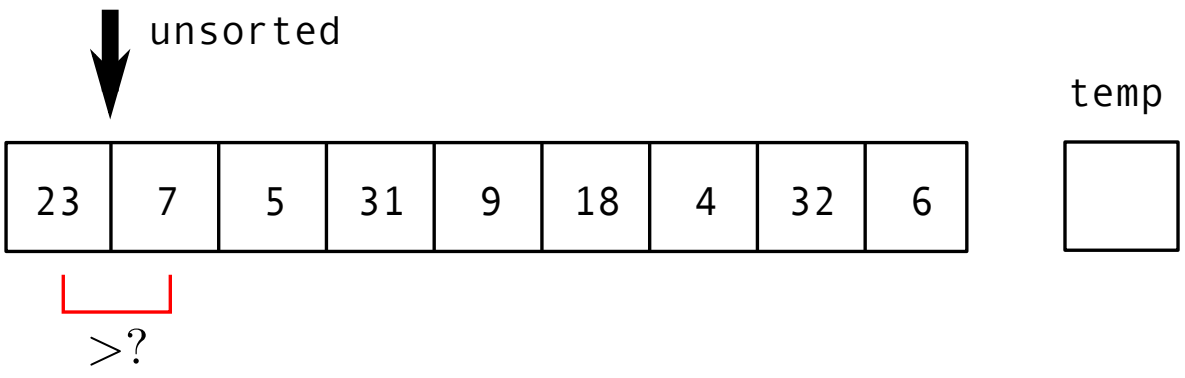
Insertion sort

So, part of the input is always sorted and we keep inserting items into that part. The sorted part grows and the unsorted part shrinks until there is nothing left to do.

We need to keep track of which part of the collection is sorted, and we need to store temporary values as we make room for an element to be moved.

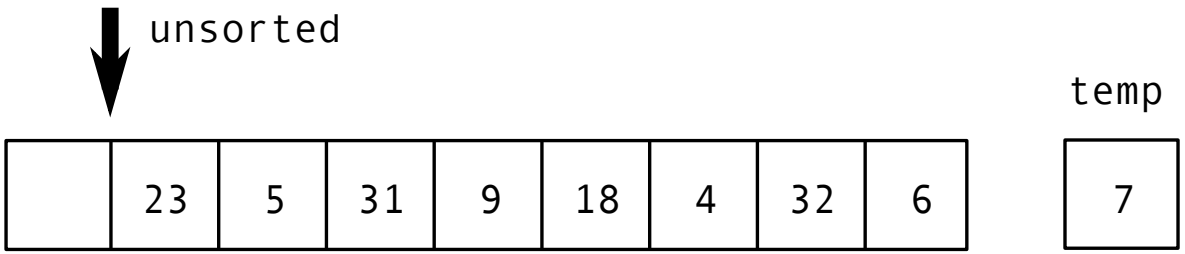
Insertion sort

Part of a run of insertion sort:



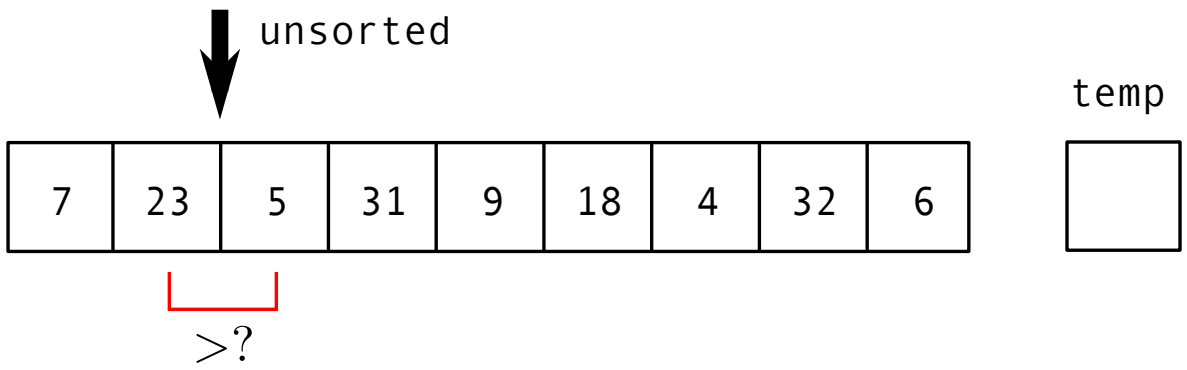
Insertion sort

Part of a run of insertion sort:



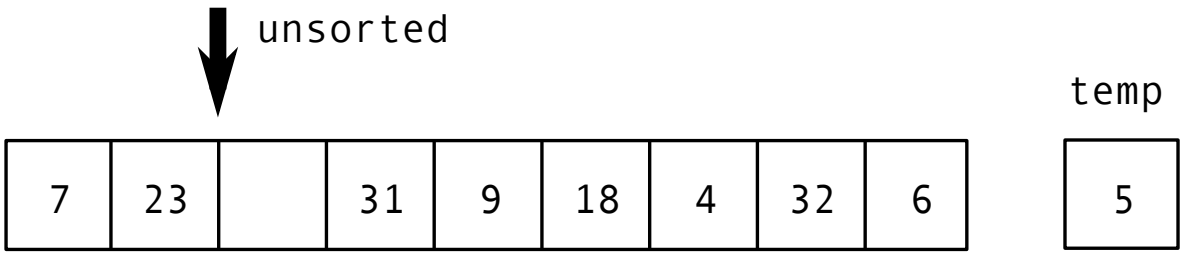
Insertion sort

Part of a run of insertion sort:



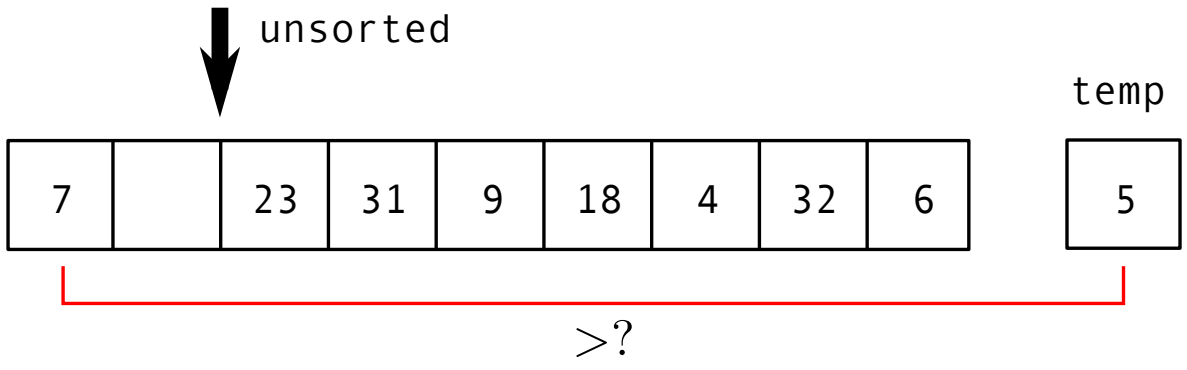
Insertion sort

Part of a run of insertion sort:



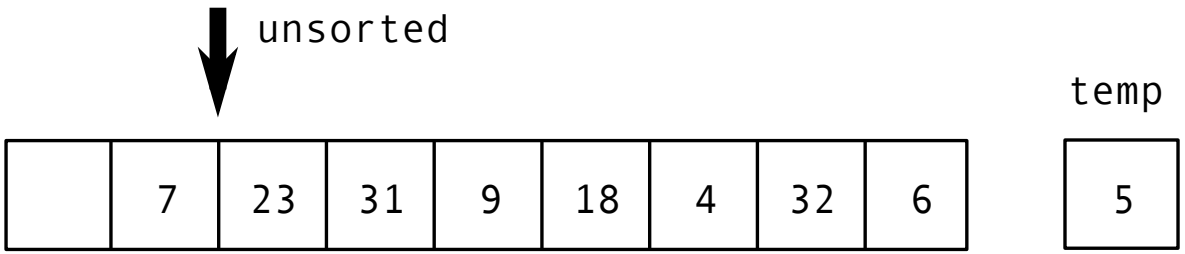
Insertion sort

Part of a run of insertion sort:



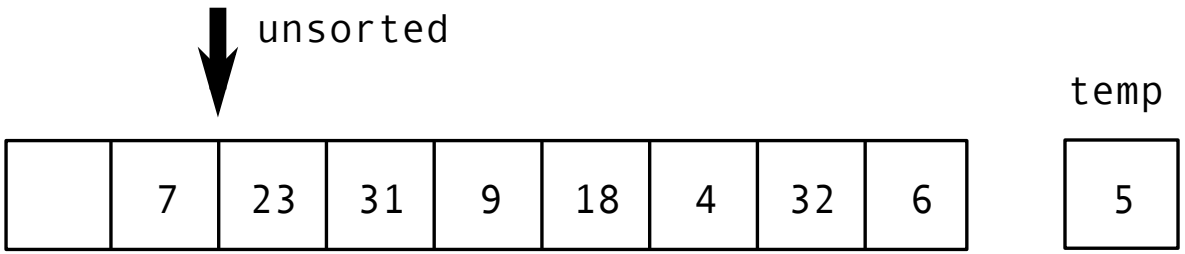
Insertion sort

Part of a run of insertion sort:



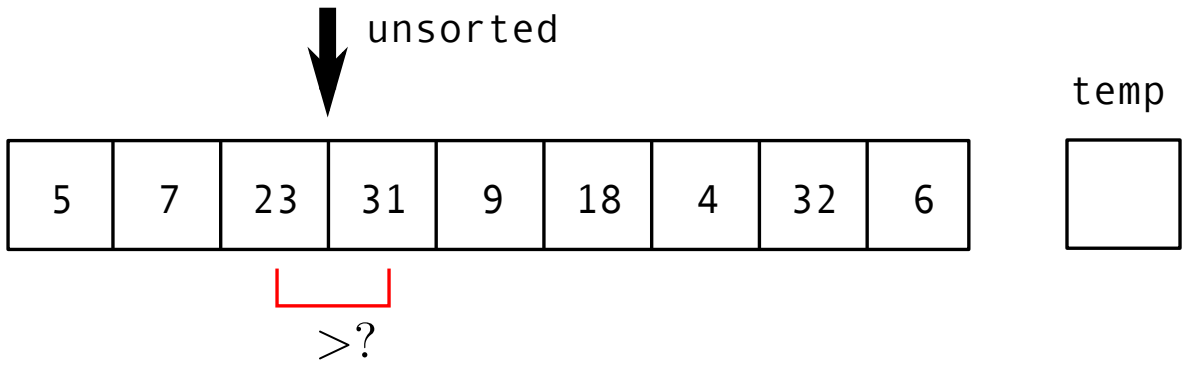
Insertion sort

Part of a run of insertion sort:



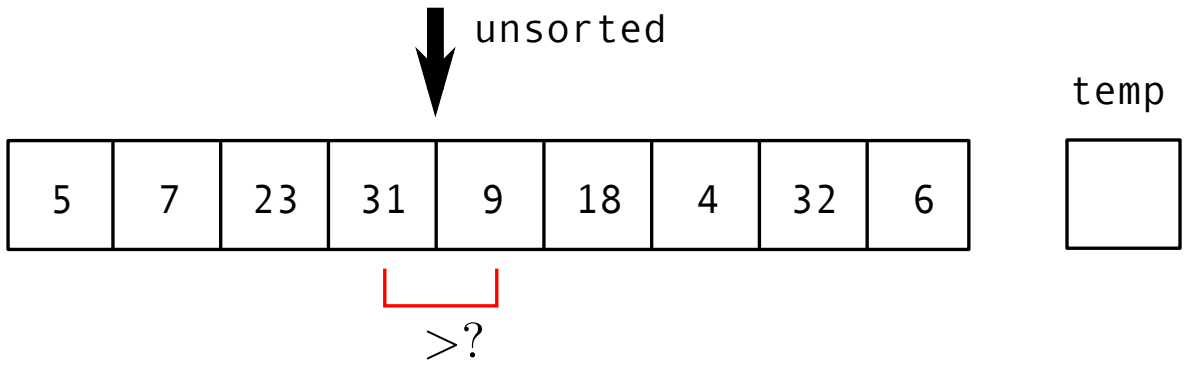
Insertion sort

Part of a run of insertion sort:



Insertion sort

Part of a run of insertion sort:



Insertion sort

You will probably have to think about this and compare it to the applet to convince yourself that it's true.

```
procedure Insertion Sort(arr, n)        ▷ n is the length of arr
    for out from 1 up to n − 1 do       ▷ out is the dividing line
        tmp ← arr[out]
        in ← out                        ▷ start shifts at out
        while in > 0 and arr[in − 1] > tmp do
            arr[in] ← arr[in − 1]
            in ← in − 1
        end while
        arr[in] ← tmp
    end for
end procedure
```

The previous two sorts reduce the size of the problem at each pass. In this case we increase it. On the first pass, we make one comparison, on the second, a maximum of two, and so on.

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

So the worst case is the same: $O(n^2)$. However, insertion sort performs much better when the data is sorted or "almost" sorted.