Insertion sort

Acknowledgement:

Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.3, p.144-8

Credit: Prof. Douglas Wilhelm Harder, ECE, University of Waterloo, Ontario, Canada

Outline

- The insertion sort – Time complexity $\mathbf{O}(n^2)$
- We shall discuss:

1) The algorithm
2) An example
3) Pseudo-code
4) Run-time and space analysis
   - worst case
   - average case
   - best case
5) Real-time use cases
6) Advantages and disadvantages
7) Summary

Background

Consider the following observations:

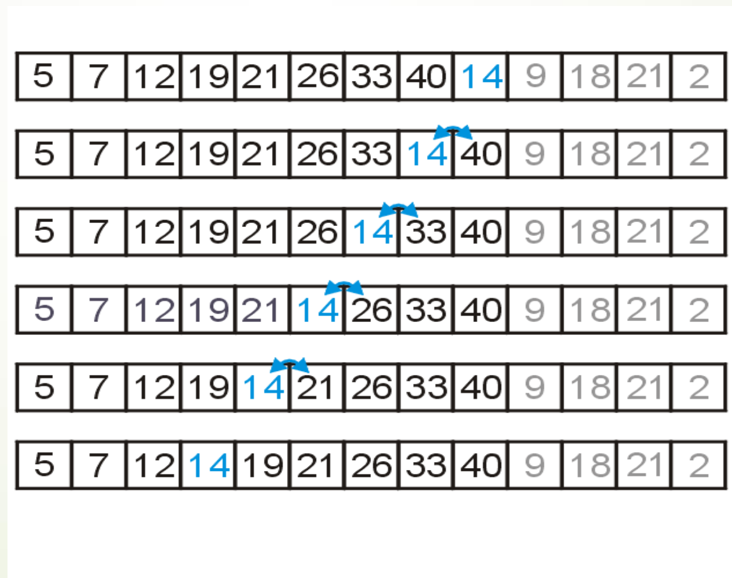
- A list with one element is sorted.
- In general, if we have a sorted list of k items, we can insert a new item to create a sorted list of size $k + 1$.
- For example, consider this sorted array containing of eight sorted entries.

5	7	12	19	21	26	33	40	14	9	18	21	2

- Suppose we want to insert 14 into this array leaving the resulting array sorted.

Background

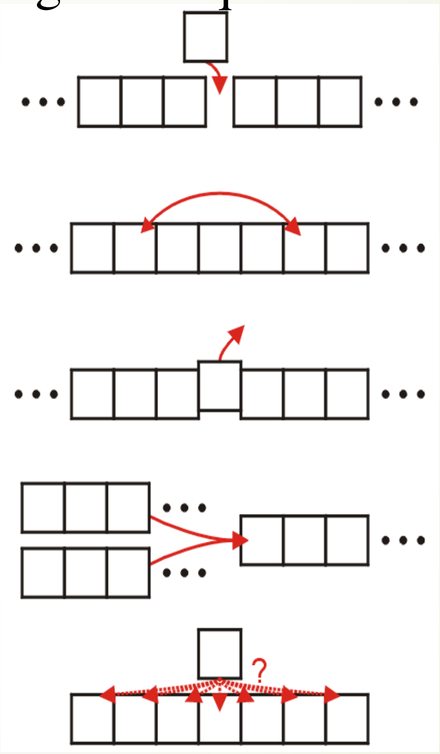
- Starting at the end, if the number is greater than 14, copy it to the right.
  - Once an entry less than 14 is found, insert 14 into the resulting vacancy



Background

Recall the five sorting techniques:

- Insertion
- Exchange
- Selection
- Merging
- Distribution

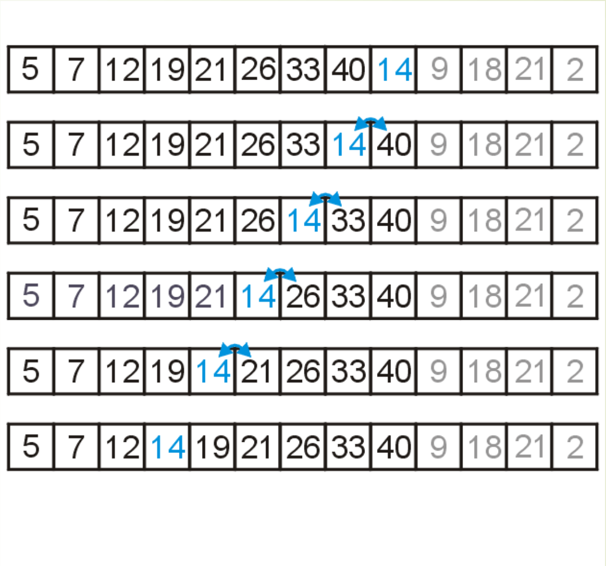


Clearly insertion sort falls into the first category

The Algorithm

Code for this would be:

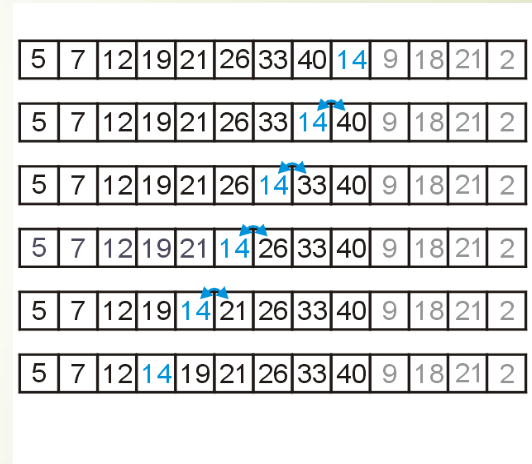
```
for ( int j = k; j > 0; --j )
{
  if ( array[j - 1] > array[j] )
  {
   swap( array[j - 1], array[j] )
  }
  else
  {
   // As soon as we don't need to swap, the (k + 1)st
   // is in the correct location
   break;
  }
}
```



Implementation Analysis

This code segment would be embedded in a function call such as

```
void insertion_sort( Type *const array, int const n )
{
   for ( int k = 1; k < n; ++k )
   {
    for ( int j = k; j > 0; --j )
     {
       if ( array[j - 1] > array[j] )
         swap( array[j - 1], array[j] );
       else
       {
         // As soon as we don't need to swap,
         // the (k + 1)st is in the correct location
         //
         break;
       }
     }
   }
}
```



Run-time Analysis

The **Θ**(1)-initialization of the outer for-loop is executed once

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
    {
        for ( int j = k; j > 0; --j )
        {
            if ( array[j - 1] > array[j] )
                swap( array[j - 1], array[j] );
            else
            {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Run-time Analysis

This $\mathbf{\Theta}(1)$- condition will be tested n times at which point it fails

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
    {
        for ( int j = k; j > 0; --j )
        {
            if ( array[j - 1] > array[j] )
                swap( array[j - 1], array[j] )
            else
            {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Run-time Analysis

Thus, the inner for-loop will be executed a total of $n - 1$ times

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
    {
        for ( int j = k; j > 0; --j )
        {
            if ( array[j - 1] > array[j] )
            {
                swap( array[j - 1], array[j] );
            }
            else
            {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Run-time Analysis

In the worst case, the inner for-loop is executed a total of *k* times

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
    {
        for ( int j = k; j > 0; --j )
        {
            if ( array[j - 1] > array[j] )
            {
                swap( array[j - 1], array[j] );
            }
            else
            {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Run-time Analysis

The body of the inner for-loop runs in **Θ**(1) in either case

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
    {
        for ( int j = k; j > 0; --j )
        {
            if ( array[j - 1] > array[j] )
                swap( array[j - 1], array[j] );
            else
            {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Thus, the worst-case run time is

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

Run-time Analysis

Problem: we may break out of the inner loop…

```
void insertion_sort( Type *const array, int const n )
{
   for ( int k = 1; k < n; ++k )
     {
       for ( int j = k; j > 0; --j )
        {
           if ( array[j - 1] > array[j] )
           {
             swap( array[j - 1], array[j] );
           }
           else
           {
              // As soon as we don't need to swap, the (k + 1)st
              // is in the correct location
              break;
           }
        }
    }
}
```

Run-time Analysis

Recall: each time we perform a swap, we remove an inversion

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
     {
        for ( int j = k; j > 0; --j )
         {
            if ( array[j - 1] > array[j] )
            {
                swap( array[j - 1], array[j] );
            }
            else
            {
               // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Run-time Analysis

As soon as a pair `array[j - 1] <= array[j]`, we are finished

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
     {
        for ( int j = k; j > 0; --j )
        {
            if ( array[j - 1] > array[j] )
                swap( array[j - 1], array[j] );
            else
            {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Run-time Analysis

Thus, the body is run only as often as there are inversions

```
void insertion_sort( Type *const array, int const n )
{
    for ( int k = 1; k < n; ++k )
    {
        for ( int j = k; j > 0; --j )
   {
            if ( array[j - 1] > array[j] )
                swap( array[j - 1], array[j] );
            else
            {
               // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

If the number of inversions is d, the run time is $\mathbf{\Theta}(n + d)$

Consequences of Our Analysis

A random list will have $d = \mathbf{O}(n^2)$ inversions

- The average random list has $d = \mathbf{\Theta}(n^2)$ inversions
- Insertion sort, however, will run in $\mathbf{\Theta}(n)$ time whenever $d = \mathrm{O}(n)$

Other benefits:

- The algorithm is easy to implement
- Even in the worst case, the algorithm is fast for small problems
- Considering these run times, it appears to be approximately 10 instructions per inversion

Size	Approximate Time (ns)
8	175
16	750
32	2700
64	8000

Consequences of Our Analysis

Unfortunately, it is not very useful in general:

- Sorting a random list of size $2^{23} \approx 8{,}000{,}000$ would require approximately one day

Doubling the size of the list quadruples the required run time

- An optimized quick sort requires less than 4 sec on a list of the above size

Consequences of Our Analysis

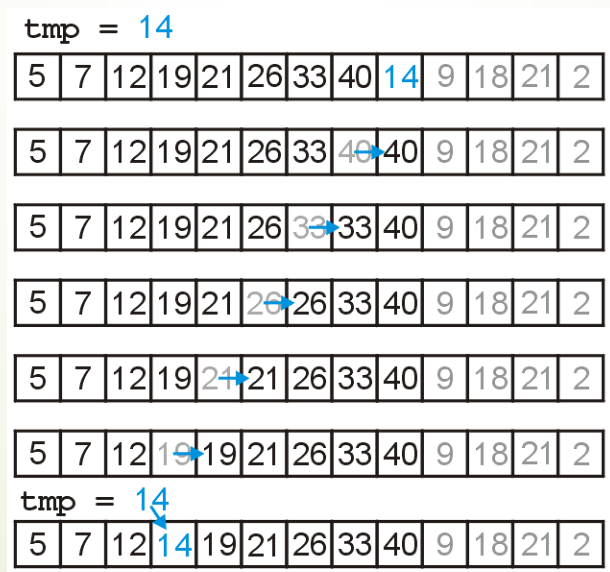
The following table summarizes the run-times of insertion sort

Case	**Run Time**	**Comments**
Worst	$\Theta(n^2)$	Reverse sorted
Average	$O(d + n)$	Slow if $d = \omega(n)$
Best	$\Theta(n)$	Very few inversions: $d = O(n)$

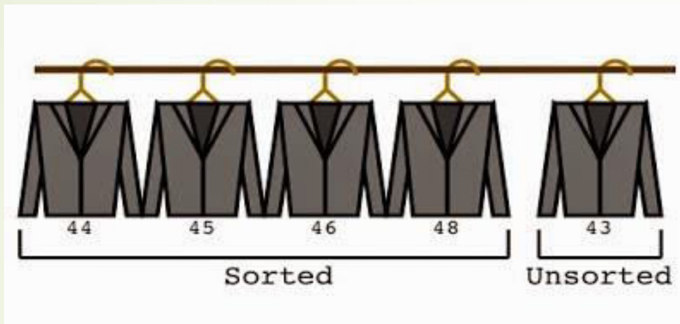
The Algorithm (rewind ..)

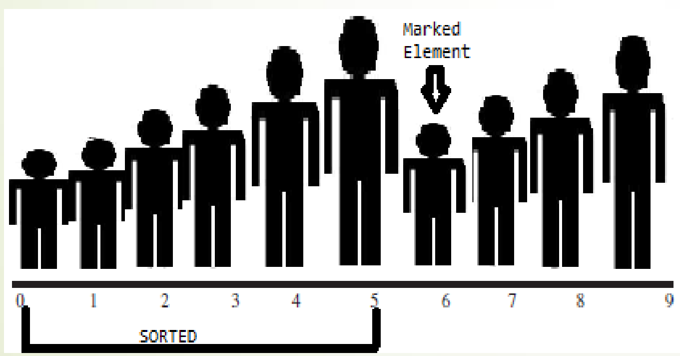
Now, swapping is expensive, so we could just temporarily assign the new entry

- this reduces assignments by a factor of 3
- speeds up the algorithm by a factor of two



Applications of Insertion sorting





Advantages

1. Simple to implement and **in-place** algorithm, meaning it requires no extra space. Thus, it is memory efficient
2. Maintains **relative order** of the input data in case of two equal values (stable)
3. Good for sorting '**almost sorted**' arrays - performs much better than other sorting algorithms
4. Sorting Small Sub-lists in **Quick-sort**
5. Use when we have smaller number of elements to sort in an array
6. The more the sequence is ordered the closer is run time to linear time O(n)
7. Used to design Tim sort (sorting algorithm runs in **Python** sort() API)

Limitations

1. Not suitable for large data sets
2. Useful only when sorting a list of few items
3. Performs worst when the required items are in reverse order

Summary

Insertion Sort:

- Insert new entries into growing sorted lists
- Run-time analysis
  - Actual and average case run time : $\mathbf{O}(n^2)$
  - Detailed analysis : $\Theta(n + d)$
  - Best case ($\mathbf{O}(n)$ inversions) : $\Theta(n)$
- Memory requirements: $\Theta(1)$