Insertion sort

Outline

This topic discusses the insertion sort

We will discuss:

- the algorithm
- an example
- run-time analysis
  - worst case
  - average case
  - best case

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 $k + 1$

Background

For example, consider this sorted array containing of eight sorted entries

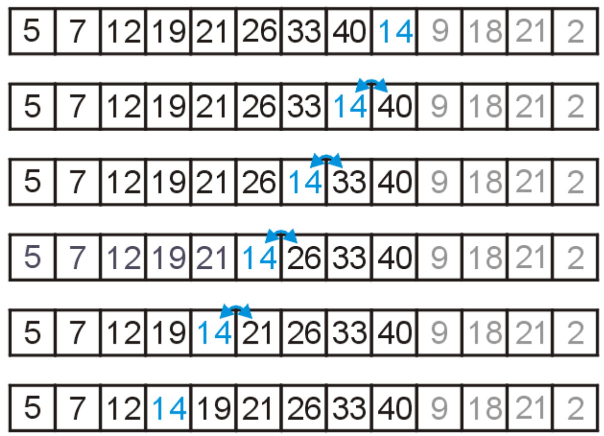
5	**7**	**12**	**19**	**21**	**26**	**33**	**40**	14	9	18	21	2

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

Background

Starting at the back, 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



The Algorithm

For any unsorted list:

- Treat the first element as a sorted list of size 1

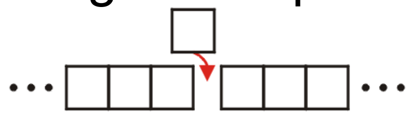
Then, given a sorted list of size $k - 1$

- Insert the k^{th} item in the unsorted list into it into the sorted list
- The sorted list is now of size $k + 1$

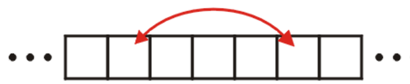
The Algorithm

Recall the five sorting techniques:

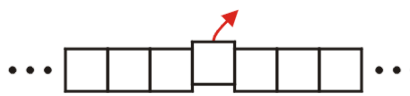
- Insertion



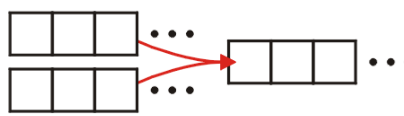
- Exchange



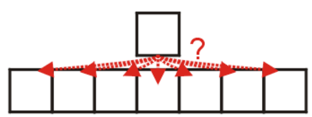
- Selection



- Merging



- Distribution

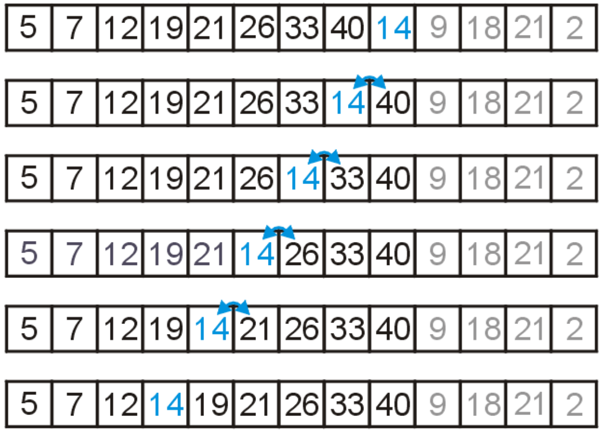


Clearly insertion falls into the first category

The Algorithm

Code for this would be:

```
for ( int j = k; j > 0; --j ) {
    if ( array[j - 1] > array[j] ) {
        std::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 and Analysis

This would be embedded in a function call such as

```
template <typename Type>
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] ) {
                std::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 and Analysis

Let's do a run-time analysis of this code

```
template <typename Type>
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] ) {
                std::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 and Analysis

The $\Theta(1)$-initialization of the outer for-loop is executed once

```
template <typename Type>
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] ) {
                std::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 and Analysis

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

```
template <typename Type>
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] ) {
                std::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 and Analysis

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

```
template <typename Type>
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] ) {
                std::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 and Analysis

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

```
template <typename Type>
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] ) {
                std::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 and Analysis

The body of the inner for-loop runs in $\Theta(1)$ in either case

```
template <typename Type>
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] ) {
                std::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

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

Implementation and Analysis

Problem: we may break out of the inner loop…

```
template <typename Type>
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] ) {
                std::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 and Analysis

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

```
template <typename Type>
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] ) {
                std::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 and Analysis

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

```
template <typename Type>
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] ) {
                std::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 and Analysis

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

```
template <typename Type>
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] ) {
                std::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 $\Theta(n + d)$

Consequences of Our Analysis

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

- The average random list has $d = \Theta(n^2)$ inversions
- Insertion sort, however, will run in $\Theta(n)$ 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 $4\ \mathrm{s}$ 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: $d = O(n)$

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: $\Theta(1)$

References

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.1, p.80-82.

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, p.2-4, 6-9.

[3] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §8.2, p.262-5.

[4] Edsger Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM 11 (3), pp.147–148, 1968.

[5] Donald Knuth, *Structured Programming with Goto Statements*, Computing Surveys 6 (4): pp.261–301, 1972.

References

Wikipedia, http://en.wikipedia.org/wiki/Insertion_sort

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.1, p.80-82.

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, p.2-4, 6-9.

[3] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §8.2, p.262-5.

[4] Edsger Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM 11 (3), pp.147–148, 1968.

[5] Donald Knuth, *Structured Programming with Goto Statements*, Computing Surveys 6 (4): pp.261–301, 1972.