

# Working with Ordered Data

Steven Zeil

July 9, 2013

## Contents

<b>1</b>	<b>Sorting</b>	<b>2</b>
1.1	Insertion Sort . . . . .	2
<b>2</b>	<b>Binary Search</b>	<b>5</b>

# 1 Sorting

## Sorting

- Text discusses
    - bubble sort
    - section sort
    - insertion sort
  - We will look at insertion sort - the fastest of these three and the only one that sees practical use.
  - Much faster (but more complicated) algorithms in CS361
- .....

## 1.1 Insertion Sort

### Insertion Sort

Basic idea:

- We know how to insert one element in order
    - [ordered insertion](#) ↗ (8)
  - What if we built up an entire array that way?
- .....

## Getting Started

```

template <class T>
void insertionSort (T* array, int size)
{
    for (int firstOutOfOrder = 1;❷
        firstOutOfOrder < size; )❸
    {
        addInOrder (array, firstOutOfOrder, ❶
                    array[firstOutOfOrder]);
    }
}

```

.....

- ❶ Basically, this algorithm does repeated calls to `addInOrder`, adding a different element each time
  - At any given moment, all items in `array[0..firstOutOfOrder-1]` should be ordered
- ❷ We start the loop at 1, not 0, because, by definition a list of one element (`array[0..0]`) is already ordered
- ❸ We don't put `firstOutOfOrder++` in the increment position of the loop, because `addInOrder` already increments its second parameter, and we don't want it incremented twice each time around the loop.

### All for One

So these two functions, together, should sort the array.

```

template <class T>
void insertionSort (T* array, int size)
{
    for (int firstOutOfOrder = 1;
        firstOutOfOrder < size; )
    {
        addInOrder (array, firstOutOfOrder,
                    array[firstOutOfOrder]);
    }
}

```

```

}
}

template <class T>
int addInOrder (T* array, int& size, T value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        array[toBeMoved+1] = array[toBeMoved];
        --toBeMoved;
    }
    // Insert the new value
    array[toBeMoved+1] = value;
    ++size;
    return toBeMoved+1;
}

```

.....

## and One for All

Now let's merge them...

```

template <class T>
void insertionSort (T* array, int size)
{
    for (int firstOutOfOrder = 1; firstOutOfOrder < size;
        ++firstOutOfOrder)
    {
        T value = array[firstOutOfOrder];
        int toBeMoved = firstOutOfOrder - 1;
        while (toBeMoved >= 0 && value < array[toBeMoved]) {
            array[toBeMoved+1] = array[toBeMoved];
            --toBeMoved;
        }
    }
}

```

```

}
// Insert the new value
array[toBeMoved+1] = value;
}
}

```

.....

You can play with the both the insertion sort algorithm and the other sorting algorithms mentioned in the text [here](#).

## 2 Binary Search

### Binary Search

- Searching ordered array for a value  $x$
- Pick element in the middle
  - If  $x < \text{array}[\text{mid}]$  then  $x$  must be in the lower half of the array
  - If  $x > \text{array}[\text{mid}]$  then  $x$  must be in the upper half of the array
  - With just one comparison, we have eliminated half the possible locations
- Rinse, lather, repeat

### Binary Search Code

```

int binarySearch(const int list[], int listLength,
                int searchItem)
{
    int first = 0;
    int last = listLength - 1;

```

```
int mid; ❶

bool found = false;

while (first <= last && !found) ❷
{
    mid = (first + last) / 2; ❸

    if (list[mid] == searchItem) ❹
        found = true;
    else
        if (list[mid] > searchItem) ❺
            last = mid - 1;
        else
            first = mid + 1;
}

if (found)
    return mid;
else
    return -1;
}
```

❶ BTW, this is poor style.

- It's a *very* bad habit to declare variables without immediately initializing them.
  - Leads to lots of errors due to later accessing uninitialized variables

❷ Keep looping until we find it or until *first* and *last* bump into each other.

❸ Look at the item halfway between *first* and *last*

❹ Did we find *searchItem*?

⑥ If not, cut the search area in half by moving either *first* or *last*.

.....  
The easiest way to understand this algorithm is probably to just [try it](#).

- Binary search only works on ordered data. Either generate an ordered array or sort the one you have before attempting binary search.
- Try searching for values that are present in the array, and for values that are not in there at all.
- Try searching for values near the middle of the array, and for the lowest and highest values in the array.

### What's So Special About Binary Search?

- Each time around the loop we cut our search space in half
  - Suppose we start with 128 elements
  - After 1 iteration, we have narrowed the possible locations to 64
  - After 2 iterations, we have narrowed the possible locations to 32
  - After 3 iterations, we have narrowed the possible locations to 16
  - After 4 iterations, we have narrowed the possible locations to 8
  - After 5 iterations, we have narrowed the possible locations to 4
  - After 6 iterations, we have narrowed the possible locations to 2
  - After just 7 times around the loop, we will either have found what we are looking for or proven that it isn't in the array!

.....

**Speed of Binary Search**

How many array elements will we examine while doing binary search?

Array Size	comparisons
1	1
2	2
4	3
8	4
16	5
32	6
64	7
128	8

Compare to sequential and ordered search which visit, on average, half of the array locations per search.

.....

**Speed of Binary Search (cont.)**

Array Size	comparisons
256	9
512	10
1024	11
2048	12
4096	13
8192	14
16384	15
32768	16

.....

For an array with over 32000 elements, a sequential search would look, on average, at more than 16000 elements before finding the one we wanted.

Binary search does the same after examining just 16.

That's a 1000x speedup!