The Standard Template Library

...

Written by Rachel J. Morris, last updated 1/19/17

Topics:

- Coming up with algorithms to search a linear array
- Comparison of efficiency

Searching

Let's say that we have an array of sorted data:

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

What kind of algorithm can we come up with to

(a) see if that item is in the array, and

(b) return the index (position) where that item is in the array.

Searching

Let's say that we have an array of sorted data:

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

Sometimes, a good *starting point* is what seems most obvious – perhaps starting at the beginning, checking each item, as we traverse towards the end of the list.

Searching

Let’s say that we have an array of sorted data:

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

Sometimes, a good *starting point* is what seems most obvious – perhaps starting at the beginning, checking each item, as we traverse towards the end of the list.

Searching

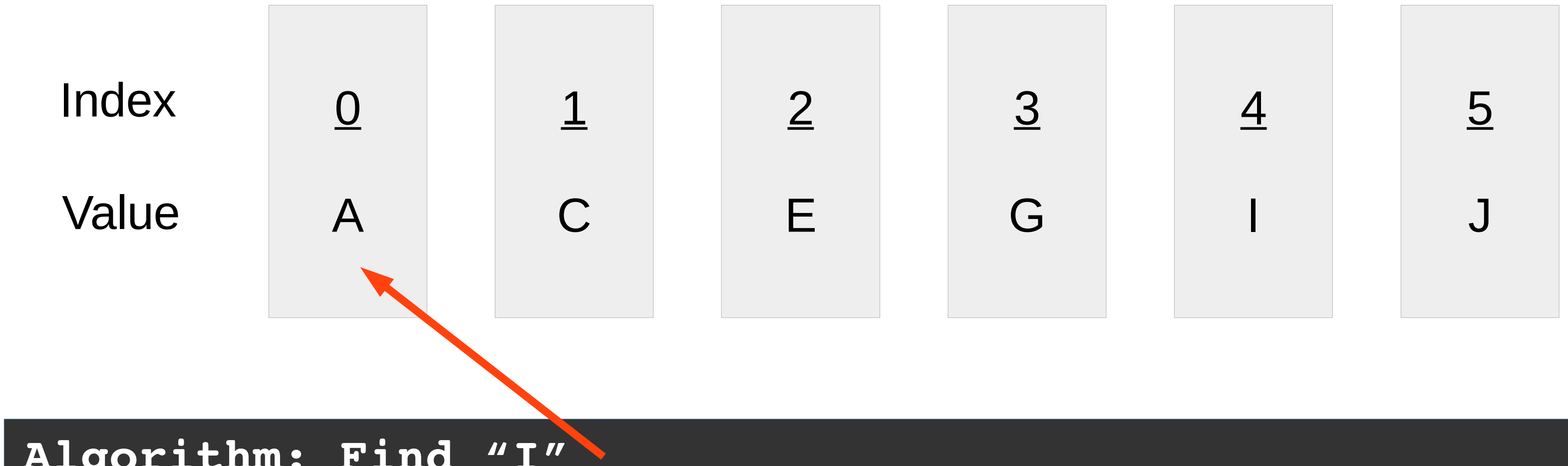
Let's say that we have an array of sorted data:

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find "I"
```

Searching

Let’s say that we have an array of sorted data:

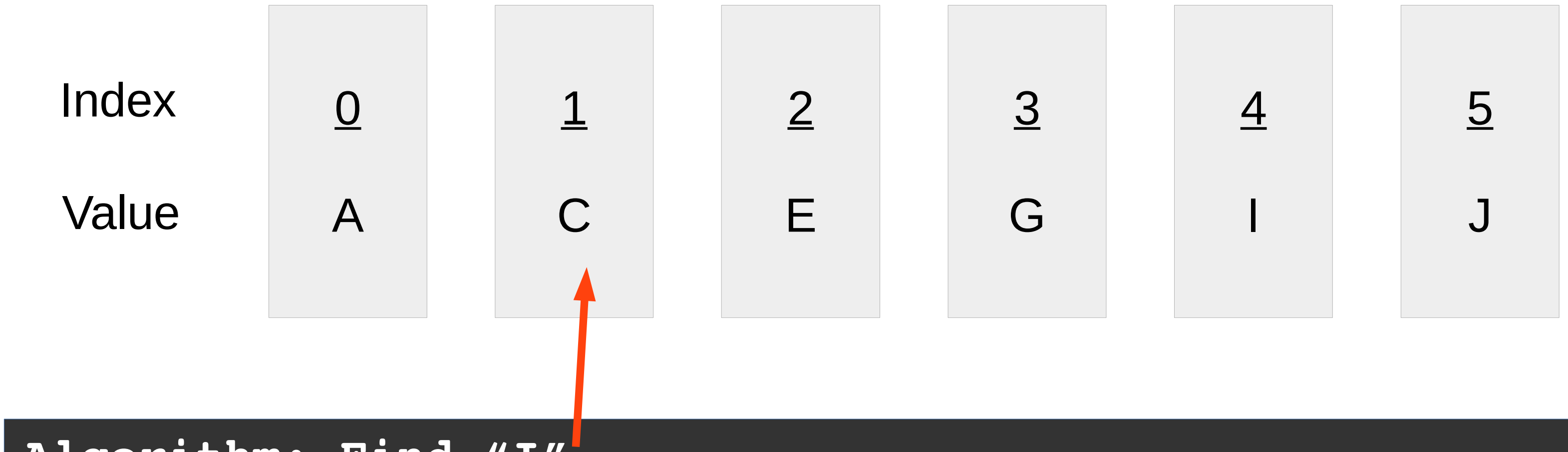


Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” at position 0?  No, that’s “A”.
```

Searching

Let’s say that we have an array of sorted data:



Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” at position 0?   No, that’s “A”.
Is “I” at position 1?   No, that’s “C”.
```

Searching

Let’s say that we have an array of sorted data:

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” at position 0?   No, that’s “A”.
Is “I” at position 1?   No, that’s “C”.
Is “I” at position 2?   No, that’s “E”.
```

Searching

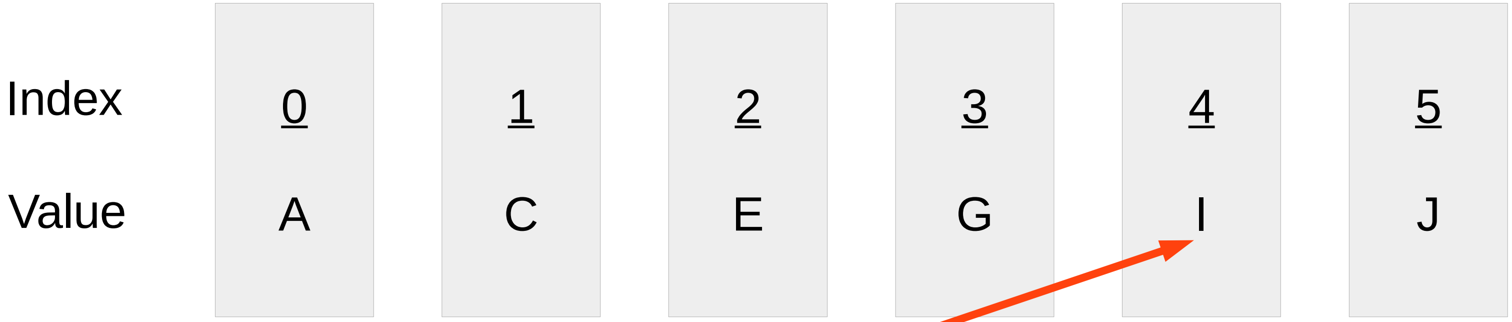
Let’s say that we have an array of sorted data:

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” at position 0?  No, that’s “A”.
Is “I” at position 1?  No, that’s “C”.
Is “I” at position 2?  No, that’s “E”.
Is “I” at position 3?  No, that’s “G”.
```

Searching

Let’s say that we have an array of sorted data:



Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” at position 0?  No, that’s “A”.
Is “I” at position 1?  No, that’s “C”.
Is “I” at position 2?  No, that’s “E”.
Is “I” at position 3?  No, that’s “G”.
Is “I” at position 4?  Yes! Found at position 4.

5 comparisons performed to find “I”.
```

Searching

```
Algorithm: Find “I”
Is “I” at position 0?  No, that’s “A”.
Is “I” at position 1?  No, that’s “C”.
Is “I” at position 2?  No, that’s “E”.
Is “I” at position 3?  No, that’s “G”.
Is “I” at position 4?  Yes! Found at position 4.

5 comparisons performed to find “I”.
```

For a small array, 5 comparisons isn’t a big deal.

However, if this array were much bigger, a search could take a long time!

Searching

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

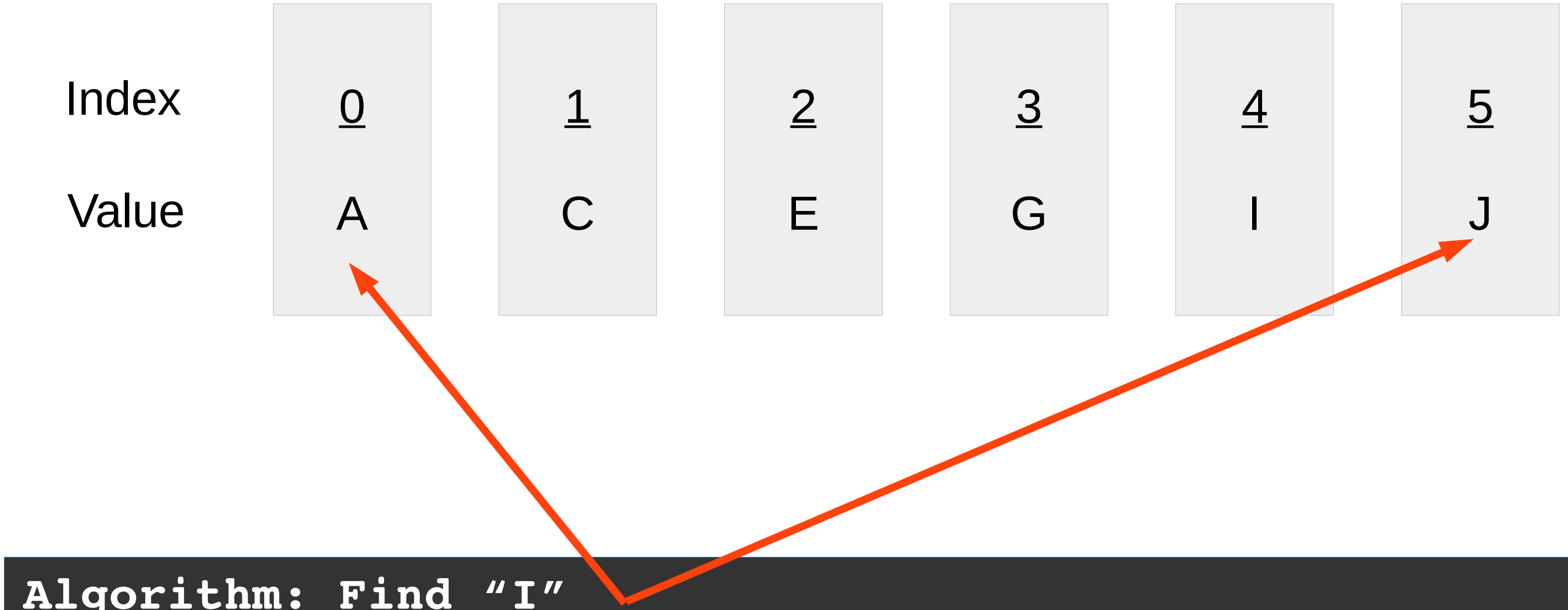
Also, this array contains sorted data, so we could write an algorithm that performs more efficiently than just starting at the beginning and moving forward!

Searching

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

**As a simple example, the next-obvious search algorithm might be to either move from beginning-to-end, or end-to-beginning, based on whether our search term is closer to the minimum value (at 0),
or the maximum value (at 5).**

Searching



Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” closer to “A” at 0, or “J” at 5?     “J”.
    Search from end to begin!
```

Searching

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” closer to “A” at 0, or “J” at 5?    “J”.
   Search from end to begin!
Is “I” at position 5?  No, that’s, “J”.
```

Searching

Index	0	1	2	3	4	5
Value	A	C	E	G	I	J

```
Algorithm: Find “I”
Is “I” closer to “A” at 0, or “J” at 5?     “J”.
    Search from end to begin!
Is “I” at position 4?   Yes, it is at position 4!

2 comparisons performed to find “I”.
```

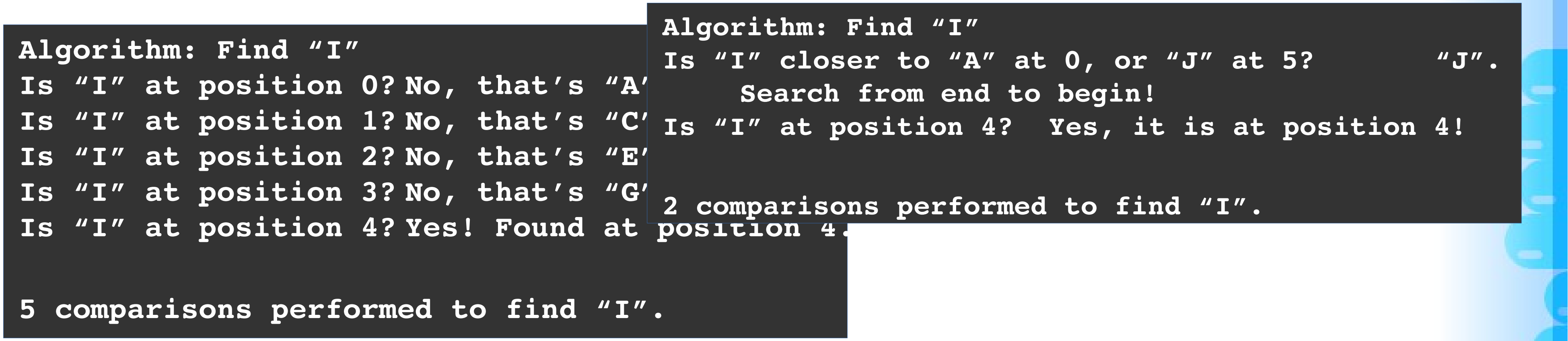
Searching

```
Algorithm: Find “I”
Is “I” at position 0?   No, that’s “A”.
Is “I” at position 1?   No, that’s “C”.
Is “I” at position 2?   No, that’s “E”.
Is “I” at position 3?   No, that’s “G”.
Is “I” at position 4?   Yes! Found at position 4.

5 comparisons performed to find “I”.
```

So when we have sorted data, we can come up with something more intelligent to try to find our search item faster!

```
Algorithm: Find “I”
Is “I” closer to “A” at 0, or “J” at 5?     “J”.
   Search from end to begin!
Is “I” at position 4?   Yes, it is at position 4!

2 comparisons performed to find “I”.
```

Searching



```
Algorithm: Find “I”
Is “I” at position 0? No, that’s “A’
Is “I” at position 1? No, that’s “C’
Is “I” at position 2? No, that’s “E’
Is “I” at position 3? No, that’s “G’
Is “I” at position 4? Yes! Found at position 4.

5 comparisons performed to find “I”.
```

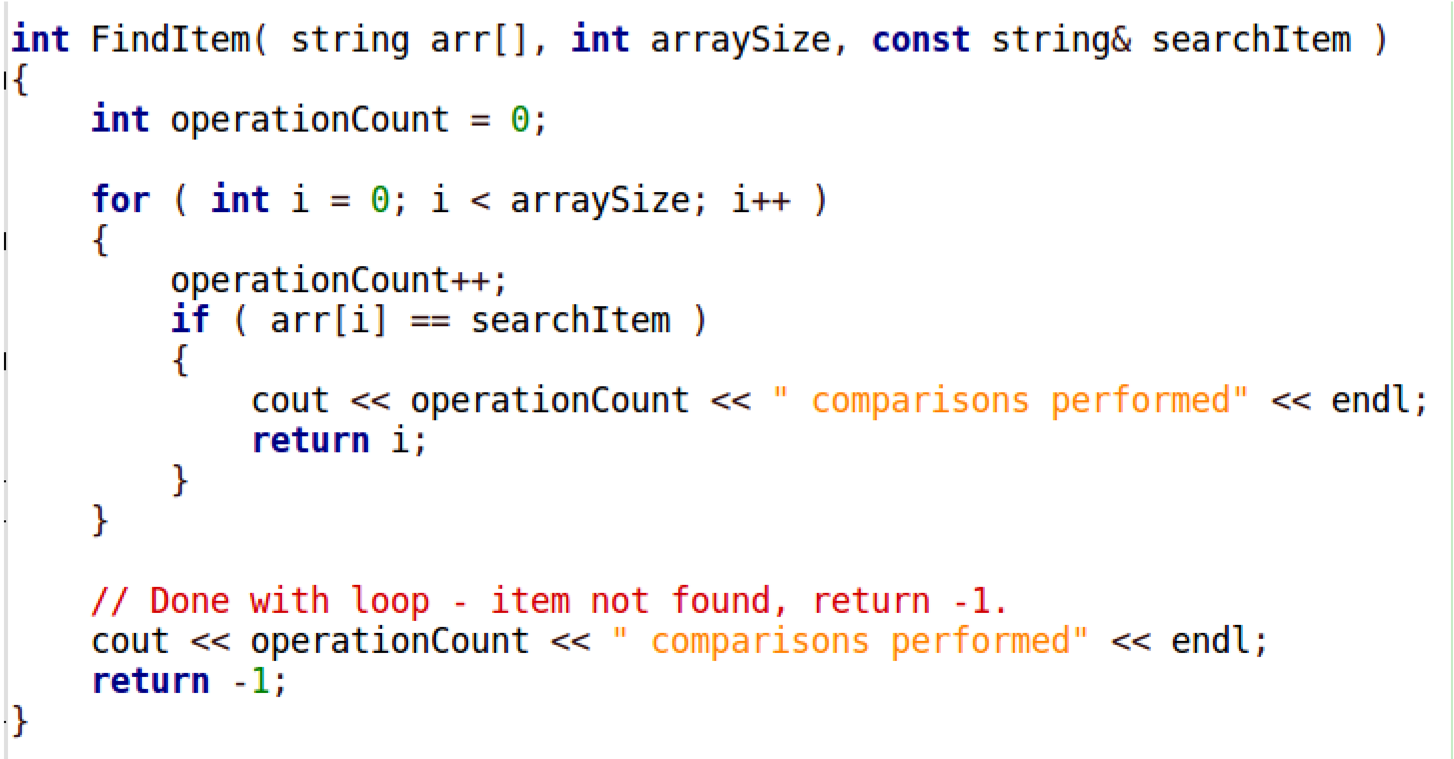
```
Algorithm: Find “I”
Is “I” closer to “A” at 0, or “J” at 5?       “J”.
    Search from end to begin!
Is “I” at position 4?  Yes, it is at position 4!

2 comparisons performed to find “I”.
```

The amount of comparisons performed is how we can tell which one is more efficient than the other. Another way to think of it is, how many times did we go through one cycle of the loop?

Searching

0	1	2	3	4	5
A	C	E	G	I	J

We can step through this search algorithm by using our IDE's debugging tools (like break points) to see it in action!



```
int FindItem( string arr[], int arraySize, const string& searchItem )
{
    int operationCount = 0;

    for ( int i = 0; i < arraySize; i++ )
    {
        operationCount++;
        if ( arr[i] == searchItem )
        {
            cout << operationCount << " comparisons performed" << endl;
            return i;
        }
    }

    // Done with loop - item not found, return -1.
    cout << operationCount << " comparisons performed" << endl;
    return -1;
}
```