Algorithm Efficiency

Written by Rachel J. Morris, last updated Feb 26, 2018

About

When we are designing a program that stores and works with data, we want to choose the most efficient way to store the data – but it isn't one-size-fits-all! How do we figure out the efficiency of an algorithm?

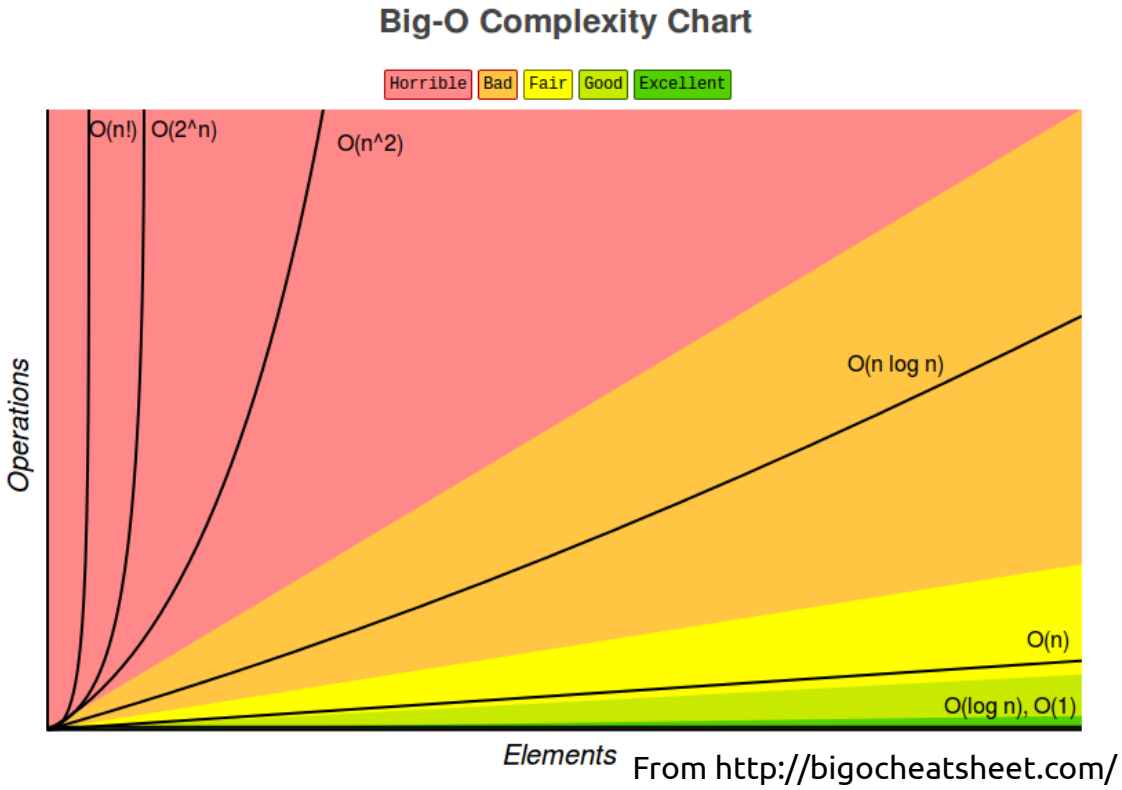
Topics

1. Introduction
2. Figuring out Time Complexity
3. Data Structures & their algorithms

1. Introduction

1. Introduction

As we are implementing functions in our data structures – such as searching for an item, or sorting data in a list – we will want to keep in mind the **algorithm efficiency**.

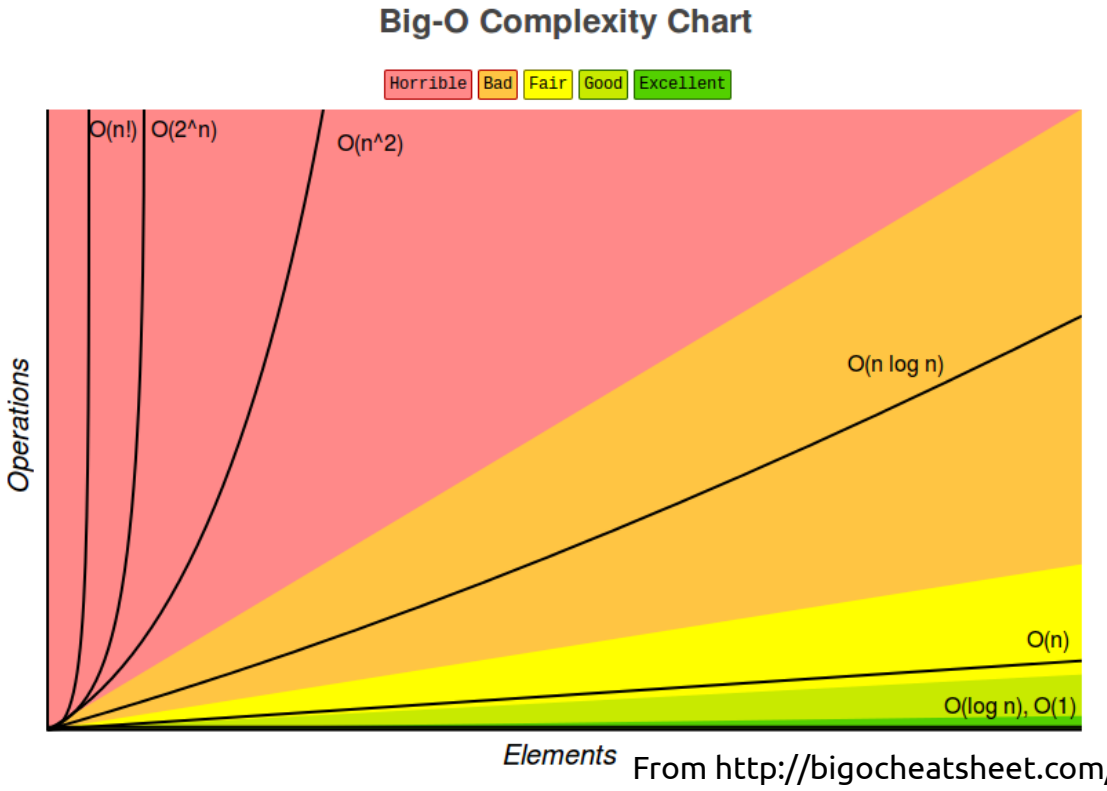


From http://bigocheatsheet.com/

1. Introduction

For example, with an *array* we can **access** elements in near-instantaneous time, but with a *linked structure* the access time's worst-case scenario is to iterate however many times big the list is.

When we are designing solutions, we will have to take into account what is most efficient for our *design*.



From http://bigocheatsheet.com/

1. Introduction

But how can we actually *tell* that one function is more efficient than another function? What are we analyzing to figure this out?

Let's look at the implementations for the **access** function.

Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

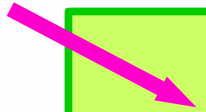
Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) *ptrCurrent = ptrCurrent→ptrNext*
3) Return *ptrCurrent->data*

1. Introduction

Let's look at the implementations for the **access** function.

For an array, we just need to do some simple math to get the memory address of the data we want.



Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) *ptrCurrent = ptrCurrent→ptrNext*
3) Return *ptrCurrent->data*

1. Introduction

Let's look at the implementations for the **access** function.

For an array, we just need to do some simple math to get the memory address of the data we want.

With a Linked List, we need to loop *n* times in order to get to the memory address.

Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) *ptrCurrent = ptrCurrent→ptrNext*
3) Return *ptrCurrent->data*

Let's look at the implementations for the **access** function.

So besides doing some addition and multiplication (a negligible amount of time), accessing an element of an array is ***virtually instantaneous.***

Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

But since we have to loop *n* times with a Linked List, the access time increases ***linearly*** – the more items, the longer it takes to get to the address we need.

Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) ptrCurrent = ptrCurrent→ptrNext
3) Return *ptrCurrent->data*

1. Introduction

Let's look at the implementations for the **access** function.

We don't care about the actual # of times we run through the loop, because we can generalize the algorithm's efficiency in terms of its *growth function.*

We can also discuss it in terms of **average case** and **worst case scenarios.**

Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) ptrCurrent = ptrCurrent→ptrNext
3) Return *ptrCurrent->data*

1. Introduction

Let's look at the implementations for the **access** function.

Worst case scenario, we have to start at item 0 and iterate over the entire list to get our item. As the list grows, the time grows linearly. Same for **average case**.

We write its efficiency as ***O(n)*** "Big-O of n"

Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) ptrCurrent = ptrCurrent→ptrNext
3) Return *ptrCurrent->data*

Let's look at the implementations for the **access** function.

With the virtually instant access time of an array, the time to access doesn't change with the size of the array – it will always be the same efficiency.

Accessing an item in a array is ***O(1).***

Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) *ptrCurrent = ptrCurrent→ptrNext*
3) Return *ptrCurrent->data*

Let's look at the implementations for the **access** function.

With the virtually instant access time of an array, the time to access doesn't change with the size of the array – it will always be the same efficiency.

Accessing an item in a array is ***O(1).***

Array

Access element *n*…
1) Return ARRAY_ADDRESS + SIZEOF(type) * n

Linked Structure

Access element *n*…
1) Set *ptrCurrent* to the first item.
2) Loop *n* times…
   1) *ptrCurrent = ptrCurrent→ptrNext*
3) Return *ptrCurrent->data*

1. Introduction

Growth Rate refers to the increase in execution time, given the increase in the size of the structure.

Big-O Notation is how we write out the efficiency of the algorithm, such as O(1), O(log n), O(n), $O(n^2)$, etc.

2. Figuring out Time Complexity

2. Figuring out Time Complexity

O(1): For functions that only do some set amount of commands, and the amount of commands doesn't change with the increase in size, we have O(1) (instant) time.

This is like for an array, we use the same amount of instructions to get to a specific element. The easiest way to identify these is that they don't contain loops.

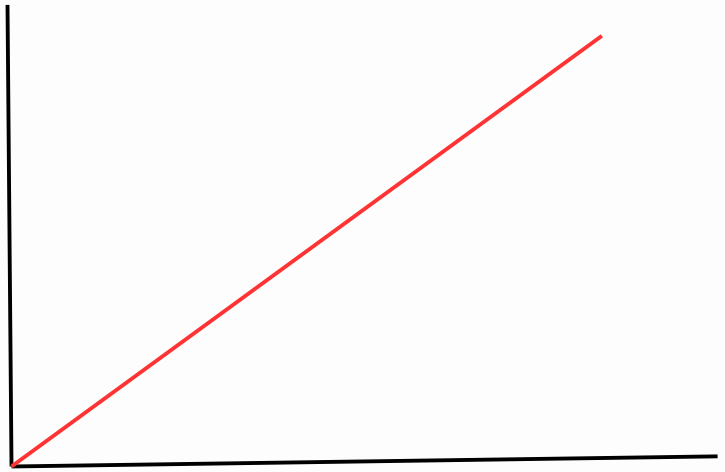
```
return myArray[5];
```

2. Figuring out Time Complexity

O(n): Algorithms whose growth rates are *linear* can be identified by having a single loop that, worst-case, iterates over all items in the structure.

As the structure grows in size, the time to run the algorithm also increases linearly.

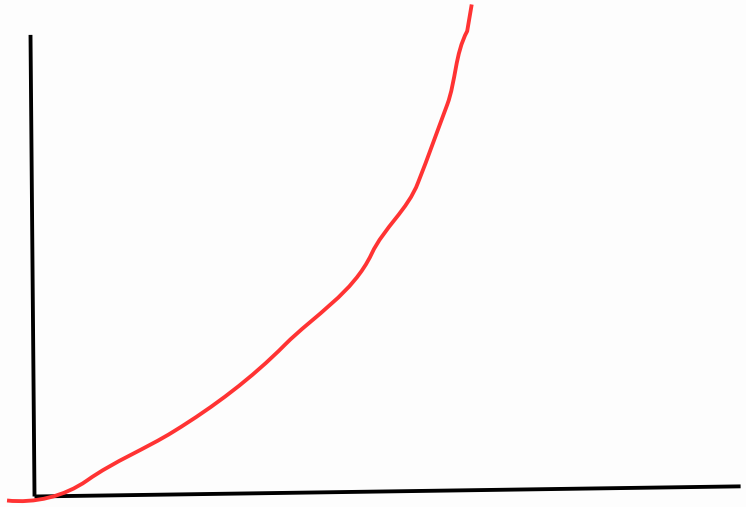
```
for ( int i = 0; i < n; i++ )
{
     if ( myArray[i] == searchItem )
     {
          return i;
     }
}
return -1;
```



2. Figuring out Time Complexity

O(n^2): Algorithms whose growth rates are *quadratic* can be identified by having two loops – one nested within another. You can think of this as iterating *n* times on the outside, and *n* times on the inside, resulting in *n x n* iterations.

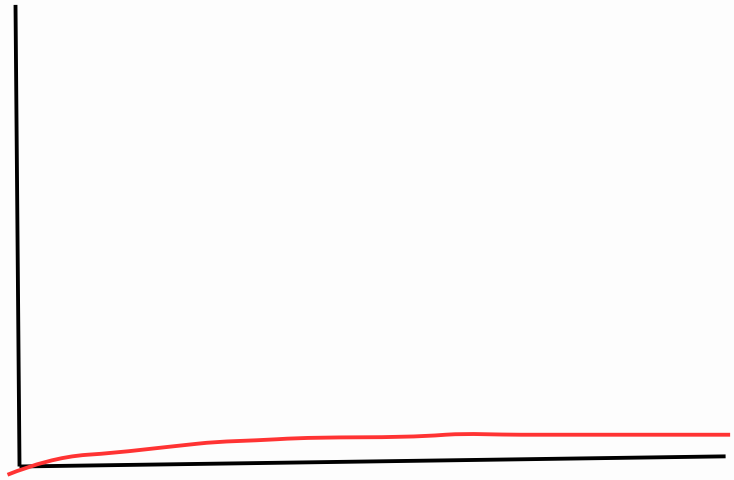
```
for ( int i = 0; i < n; i++ )
{
     for ( int j = 0; j < m; j++ )
     {
        if ( arrayOne[i] == arrayTwo[j] )
        {
              return true;
        }
     }
}
return false;
```



2. Figuring out Time Complexity

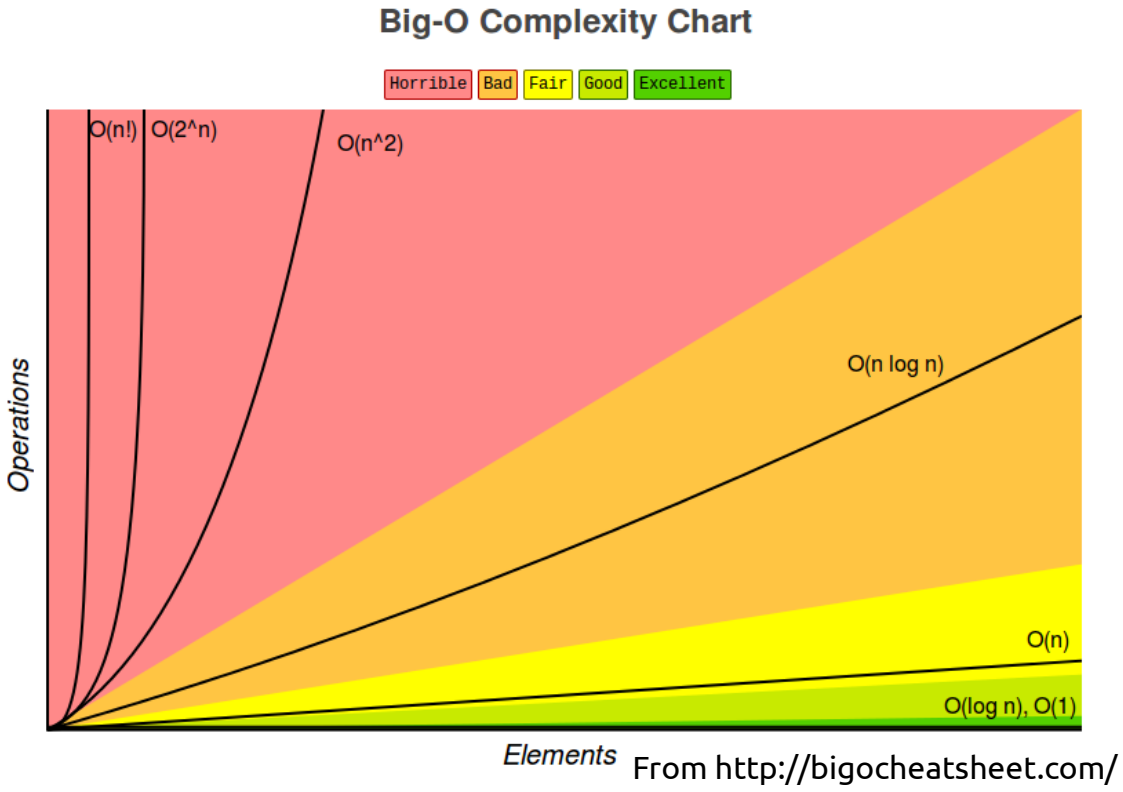
O(log n): Algorithms with *log n* efficiency reduce the "problem space" being iterated over with each pass through the loop. For example, with a binary tree, each time it generally *halves* the search pool each time it moves forward.

```
while ( ptr != nullptr )
{
     if ( *ptr < lookingFor )
     {
          ptr = ptr->left;
     }
     else
     {
          ptr = ptr->right;
     }
}
```



2. Figuring out Time Complexity

There are also other time complexities, but in this class we will mostly see **O(1), O(n), O(n2),** and **O(log n).**



From http://bigocheatsheet.com/

3. Data Structures & their algorithms

Array implementations - Access

Access	O(1)
Search	O(n)
Insert	O(n)
Delete	O(n)

An array's **access** time is instantaneous because we only need one instruction to access an element at an arbitrary position.

```
Get item i of an array:
   Return array[ i ]
```

Access	O(1)
Search	O(n)
Insert	O(n)
Delete	O(n)

Array implementations - Search

An (unsorted) array's **search** time is O(n) because we have to search through the list, one item at a time, to find an item. To confirm that the item isn't in the list, we have to look through the *entire* array.

```
Find x in array:
   For each item in array:
      if item == x:
         return x
   End for

   Return "not found"
```

Access	O(1)
Search	O(n)
Insert	O(n)
Delete	O(n)

Array implementations - Insert

An (unsorted) array's **insert** time is O(n) because we have to shift other elements over with a loop before we insert a new item in.

```
Insert x at index i in array of size n:
    For j = n to i in array:
        array[j] = array[j-1]
    End for

    array[i] = x
    itemCount += 1
```

Access	O(1)
Search	O(n)
Insert	O(n)
Delete	O(n)

Array implementations - Delete

An array's **delete** time is O(n), if the array requires elements to be contiguous
In this case, we have to iterate to shift everything backwards to fill gaps.

```
Delete item at index i in array of size n:
    For j = i to n in array:
        array[j] = array[j+1]
    End for

    itemCount -= 1
```

Access	O(n)
Search	O(n)
Insert	O(1)
Delete	O(1)

A Doubly Linked List's **access** time is O(n) because we have to iterate over *n* elements in the list to get to the item we're searching for.

Doubly Linked List - Access

```
Get item i in List:
   Counter = 0
   Pointer = First-Node

   While Counter < i:
      Pointer = Pointer->Next
      Counter += 1
   End While

   Return Pointer->Data
```

Access	O(n)
Search	O(n)
Insert	O(1)
Delete	O(1)

A Doubly Linked List's **search** time is O(n) because we potentially have to iterate over all the elements in the list to find a given item, or find that the item is not in the list.

Doubly Linked List - Search

```
Find item x in List:
   Pointer = First-Node

   While Counter != NULL:
      Pointer = Pointer→Next
      If Pointer->Data = x:
         Return Pointer
   End While

   Return "Not found"
```

Access	O(n)
Search	O(n)
Insert	O(1)
Delete	O(1)

A Doubly Linked List's **insert** time is O(1) because we don't have to iterate at all – we just create the node and update pointers.

Doubly Linked List - Insert

```
Add item x to end of List:
   Create newNode, set data to x

   Last->Next = newNode
   newNode->Prev = Last
   Last = newNode

   Size += 1
```

Didn't add pseudocode for adding 1st item in list.

Access	O(n)
Search	O(n)
Insert	O(1)
Delete	O(1)

A Doubly Linked List's **delete** time is O(1) because we aren't iterating here, either – we are freeing some data and updating pointers.

Doubly Linked List - Delete

```
Remove item at end of List:
    Last = Last->Previous
    Delete Last->Next
    Last->Next = NULL

    Size -= 1
```

Didn't add pseudocode for removing last item in list.

Access	O(log n)
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

For a Binary Search Tree, we have to traverse from the root node to the node we want, but each time we make a move downward (left or right), we cut out half the nodes and have less to search.
This is why it's O(log n).

Binary Search Tree – Access

```
Get item x from BST:
   Pointer = Root

   While Pointer != NULL:
      If Pointer->Data < x:
         Pointer = Pointer->Left
      Else:
         Pointer = Pointer->Right
   End While

   Return Pointer
```

Access	O(log n)
Search	O(log n)
Insert	O(log n)
Delete	O(log n)

We do the same traversal for an **Access** or a **Search** or an **Insert** or a **Delete**, because we always need to find the appropriate location when doing any of these operations.

Binary Search Tree – Access

```
Get item x from BST:
   Pointer = Root

   While Pointer != NULL:
      If Pointer->Data < x:
         Pointer = Pointer->Left
      Else:
         Pointer = Pointer->Right
   End While

   Return Pointer
```

Conclusion

Taking algorithm efficiency into account can help you choose the best structure for your design.

Are you accessing data more than you're inserting data? Maybe an Array implementation is a better choice than a Linked implementation.

Or, if you're adding and removing data more than you're accessing it, perhaps the Linked Implementation is better.

And Binary Search Trees are a nice in-between in efficiency.