As we write data structures that implement algorithms for tasks such as insert, search, or even sort, we need to also be mindful of the efficiency of the functions that we're implementing. Two aspects of this are time and space – how long does it take to run? How much space in memory does it take?

We need to be able to compare these to choose the best algorithms for the job (or take into account the trade-offs and make a good decision), but also to generally just become a better programmer and avoid writing inefficient code.

Section 1: Growth rates

Two algorithms may run similarly quickly for small sets of data, but if their growth rates are different, then the time it takes to process many more items can vary a lot.

Question 1: Given two algorithms with different growth rates, figure out which runs faster for a given amount of data. Here, *t* represents "time" and *n* represents amount of items.

a.

Algorithm	**Time to process 100 items**
Algorithm 1: Linear growth $t = 1000n$	100,000
Algorithm 2: Quadratic growth $t = n^2$	10,000
MOST EFFICIENT:	ALGORITHM 2

b.

Algorithm	**Time to process 2000 items**
Algorithm 1: Linear growth $t = 1000n$	2,000,000
Algorithm 2: Quadratic growth $t = n^2$	4,000,000
MOST EFFICIENT:	ALGORITHM 1

c.

Algorithm	**Time to process 100 items**
Algorithm 1: Linear growth $t=100n$	10,000
Algorithm 2: Cubic growth $t=n^3$	1,000,000
Algorithm 3: Exponential growth $t=2^n$	1.267×10^{30}
MOST EFFICIENT:	ALGORITHM 1

If we have a linear growth rate, it is easy to see how processing twice as many items results in twice the amount of time, or processing half requires half the time.

But how does processing more items end up resulting in quadratic or exponential growth? Are there any algorithms that are *more efficient* than a linear growth rate?

Section 2: Algorithm execution time

An algorithm's execution time is related to the number of operations it requires. This is usually expressed in terms of the number, *n*, of items the algorithm must process. Counting an algorithm's operations – if possible – is a way to assess its efficiency.

For a simple loop such as:

```
for ( int i = 0; i < 10; i++ )
{
  // Do a thing
}
```

It will always execute 10 times. We use "Big O" notation to denote the efficiency of an algorithm, so this one in particular would be O(10).

1. *For the given algorithms, write down the O(n) values.*

a.

```
int s = 0;

for ( int j = 0; j < 4; j++ )
{
  s++;
}
```

(___/1)

= O(4)

b. (___/1)

```
int s = 0;
for ( int i = 0; i < 2; i++ )
{
  for ( int j = 0; j < 4; j++ )
  {
    s++;
  }
}
```

~~O(8)~~ O(8)

c. (___/1)

```
int s = 0;

for ( int i = 0; i < 4; i++ )
{
  for ( int j = i; j < 4; j++ )
  {
    s++;
  }
}
```

O(10)

It might help to write out all the steps of the loop...

Iteration	i's value	j's vlaue
1	0	0
2	0	1

Section 3: Generalizing O(n)

Often in a program, we cannot count every time a loop is going to occur. Therefore, we use *n* as shorthand to specify *n* amount of operations.

For a simple for loop, then:

```
for ( int i = 0; i < 10; i++ )
{
  // Do a thing
}
```

We say that its efficiency is O(n), and for a nested for loop:

```
for ( int i = 0; i < 4; i++ )
{
  for ( int j = 0; j < 4; j++ )
  {
    // Do a thing
  }
}
```

We say that this is O(n^2), because we end up doing the inner-most operation $n \times n$ times. This is taking into account the *worst-case scenario*, where we have to traverse all loops fully.

Cheatsheet

- **For loops – The running time for a for loop is the run time of the internal statements, times the number of iterations.**
- **Nested loops – You should analyze these from the inside → out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.**
- **Consecutive statements – These just add, so when we are generalizing, the one we count is the one with the larger run time.**
- **If / Else – Count whichever one has the larger run time.**

2. For the given algorithms, write down the generalized O(*n*) values.

a. (__/1)

```
for ( int i = 0; i < 8; i++ )
{
  for ( int j = 0; j < 8; j++ )
  {
    for ( int k = 0; k < 8; k++ )
    {
      s++;
    }
  }
}
```

$\rightarrow O(n^3)$

b. (__/1)

```
Node* ptrCurrent = ptrFirst;

while ( ptrCurrent != nullptr )
{
  ptrCurrent = ptrCurrent->ptrNext;
}
```

$O(n)$

Is this similar to using a for loop to iterate from some beginning to some end?
What is the O(*n*) value?

c. (__/1)

```
Node* ptr1 = list1.ptrFirst;
Node* ptr2 = list2.ptrFirst;

int common = 0;

while ( ptr1 != nullptr )
{
  // See if this element is in the 2nd list...
  while ( ptr2 != nullptr )
  {
    if ( ptr2->data == ptr1->data )  { common++; }
    ptr2 = ptr2->ptrNext;
  }
  ptr1 = ptr1->ptrNext;
}
```

$O(n^2)$

Section 3: Comparing algorithm efficiencies

For different types of data structures, there are trade-offs in efficiency; it is instantaneous to access any given item in an array, but searching it can be slow. You will always have to traverse a binary search tree to find an item, but searching is faster than any given unsorted away.

When choosing the best tool for the job, it is be important to figure out whether your program will spend a lot of time changing the list of data, or searching the list of data, or both.

Insertion

```
void Push(T data)
{
    SinglyLinkedNode<T>* newNode = new
        SinglyLinkedNode<T>;
    newNode->data = data;

    if (m_size == 0)
    {
        m_ptrFirst = newNode;
        m_ptrLast = newNode;
    }
    else
    {
        m_ptrLast->ptrNext = newNode;
        m_ptrLast = newNode;
    }

    m_size++;
}
```

This method has no loops in it, so it is effectively "instantaneous" - so we say, O(1).

Searching

```
bool IsInList( T data )
{
    SinglyLinkedNode<T>* ptrCurrent =
        m_ptrFirst;

    while ( ptrCurrent != nullptr )
    {
        if ( ptrCurrent->data == data )
        {
            return true;
        }
        ptrCurrent = ptrCurrent->ptrNext;
    }

    return false;
}
```

This method loops until it gets to the end of the list, or until it finds the data. Worst case scenario, it goes through the whole list and doesn't find anything – so it is O(n).

3. Find O(*n*) for both of these functions, and write down which is faster. *Don't worry about whether we've covered these yet or not.*

a. Finding items at some index in linked lists... (__/2)

Singly Linked List

```
T& Get( int index )
{
    // Locate item
    SinglyLinkedNode<T>* ptrCurrent = m_ptrFirst;

    int counter = 0;
    while ( counter != index )
    {
        ptrCurrent = ptrCurrent->ptrNext;
        counter++;
    }

    return ptrCurrent->data;
}
```

$O(n^2)$

Doubly Linked List (written smarter)

```
T& Get( int index )
{
    // Faster to go from start to end, or end to start?
    if ( (m_itemCount - 1) - index < index )
    {
        DoublyLinkedNode<T>* ptrCurrent = m_ptrLast;
        int counter = m_itemCount - 1;

        while ( counter != index )
        {
            ptrCurrent = ptrCurrent->ptrPrev;
            counter--;
        }
    }
    else
    {
        DoublyLinkedNode<T>* ptrCurrent = m_ptrFirst;
        int counter = 0;

        while ( counter != index )
        {
            ptrCurrent = ptrCurrent->ptrNext;
            counter++;
        }
    }

    return ptrCurrent->data;
}
```

$O(n)$

(___/2)

b. Searching...

Unsorted Array

```
template <typename T>
bool Contains( T arr[], int itemCount, T value )
{
    for ( int i = 0; i < itemCount; i++ )
    {
        if ( arr[i] == value )
        {
            return true;
        }
    }
    return false;
}
```

O(n)

Binary Search Tree

```
bool Contains( T value )
{
    return ( RecursiveFind( m_ptrRoot, value ) != nullptr );
}

Node<T>* RecursiveFind( Node<T>* node, T value )
{
    if ( node->data == value )
        return node;

    else if ( value < node->data && node->ptrLeft == nullptr )
        return nullptr;

    else if ( value > node->data && node->ptrRight == nullptr )
        return nullptr;

    else if ( value < node->data )
        return RecursiveFind( node->ptrLeft, value );

    else if ( value > node->data )
        return RecursiveFind( node->ptrRight, value );
}
```

O(n)

What is the worst case time to search a binary tree? Here's a hint: Each time it calls RecursiveFind again, it *halves* its search pool by traversing left or right...

We might need to cover more on algorithm efficiency before you can answer!

c. **Searching...**

Unsorted Array

```
template <typename T>
bool Contains( T arr[], int itemCount, T value )
{
    for ( int i = 0; i < itemCount; i++ )
    {
        if ( arr[i] == value )
        {
            return true;
        }
    }
    return false;
}
```

O(n)

Dictionary

```
string Get( const string& key )
{
    int index = Hash( key );
    return m_strings[index].value;
}
```

O(1)

We haven't covered dictionaries yet, but you should be able to tell what the access time is here – do you see *any* loops? What's the access time of an array, like m_strings?

d. Insertions...

Unsorted Array

```
template <typename T>
bool Contains( T arr[], int itemCount, T value )
{
    for ( int i = 0; i < itemCount; i++ )
    {
        if ( arr[i] == value )
        {
            return true;
        }
    }
    return false;
}
```

O(1)

Binary Search Tree

```
void RecursiveInsert( Node<T>* node, Node<T>* newNode )
{
  if ( newNode->data < node->data && node->ptrLeft == nullptr )
    node->ptrLeft = newNode;
  else if ( newNode->data < node->data && node->ptrLeft != nullptr )
    RecursiveInsert( node->ptrLeft, newNode );

  else if ( newNode->data > node->data && node->ptrRight == nullptr )
    node->ptrRight = newNode;
  else if ( newNode->data > node->data && node->ptrRight != nullptr )
    RecursiveInsert( node->ptrRight, newNode );
}
```

O(log(n))

This is similar to searching in a Binary Search Tree, since in order to insert anything, you have to find a *place* for it. Its insertion time, therefore, is the same as its search and access times, which end up being ***logarithmic***...