Agenda

Intros:

WWCode @ Alchemy Code Lab

Link to slides:
https://github.com/wwcodeportland/study-nights/tree/master/algorithms

Topic Summary:

Understanding Complexity

Lab Time:

Pair Programming + Algorithms

Algorithms Study Night

Leadership Team

Caterina
Director

Richa
Skills Development Lead

Shiyuan
Design Lead

Tricia
DevOps Lead

Sabina
Events Lead

Sarah Joy
JavaScript Lead

Keeley
Community Lead

Alia
Algorithms Lead

Upcoming Events - August

Featured Event:

- **Women + Tech Summer Soiree @ Cambia** - Thu, August 17th, 5:30 PM

Rest of August:

- **Roll Call: DevRel Summit 2017 @ Galvanize** - Fri, August 11th, 8:00 AM
- **Design + Product Study Night @ New Relic** - Tue, August 15th, 5:30 PM
- **JavaScript Study Night @ Metal Toad** - Wed, August 23rd, 5:30 PM

{short} Code of Conduct

Women Who Code (WWCode) is dedicated to providing an empowering experience for everyone who participates in or supports our community, regardless of gender, gender identity and expression, sexual orientation, ability, physical appearance, body size, race, ethnicity, age, religion, socioeconomic status, caste, or creed. Our events are intended to inspire women to excel in technology careers, and anyone who is there for this purpose is welcome. Because we value the safety and security of our members and strive to have an inclusive community, we do not tolerate harassment of members or event participants in any form. Our **Code of Conduct** applies to all events run by Women Who Code, Inc. If you would like to report an incident or contact our leadership team, please submit an **incident report form**.

Resources

WWCode @ Meetup.com

WWCode @ Slack

WWCode @ Github

Big-O CheatSheet

What is Complexity?

Measure of time or space taken by an algorithm to run. It depends on your implementation.

Why is it important?

Helps you assess what resources are needed and if you can optimize the code. Efficient code is always appreciated.

Types of Complexity

“Complexity”

Synonyms:
- Algorithm Complexity
- Time Complexity
- Runtime Complexity

How fast or slow a particular algorithm performs

Time versus the input size **n**.

“Space Complexity”

Synonyms:
- Data Structure Complexity

Total space taken by the algorithm versus the input size **n**.

Includes both Auxiliary (extra/temporary) space and space used by input.

Time complexity

A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc.

The way around is to estimate **efficiency** of each algorithm **asymptotically** (when the input **n** gets really big). It measures the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Types of Notations for Complexity

1. **Big Oh** denotes "fewer than or the same as" <expression> iterations.
   example: **O(n)**

2. Big Omega denotes "more than or the same as" <expression> iterations.
   example Ω(n)

3. Big Theta denotes "the same as" <expression> iterations.
   example: Θ(n)

4. Little Oh denotes "fewer than" <expression> iterations.
   example: o(n)

5. Little Omega denotes "more than" <expression> iterations.
   example: ω(n)

Average, Best, and Worst Case

These are generally exactly what they sound like.

Best Case scenario: for searching for a value in an array would mean finding it in the first spot.

Worst Case scenario: would be searching for a value and finding in the last/worst possible place.

Average Case scenario: is in between best and worst.

Common rule: when not specified complexity refers to worst-case scenario complexity

Why is it always simple expressions

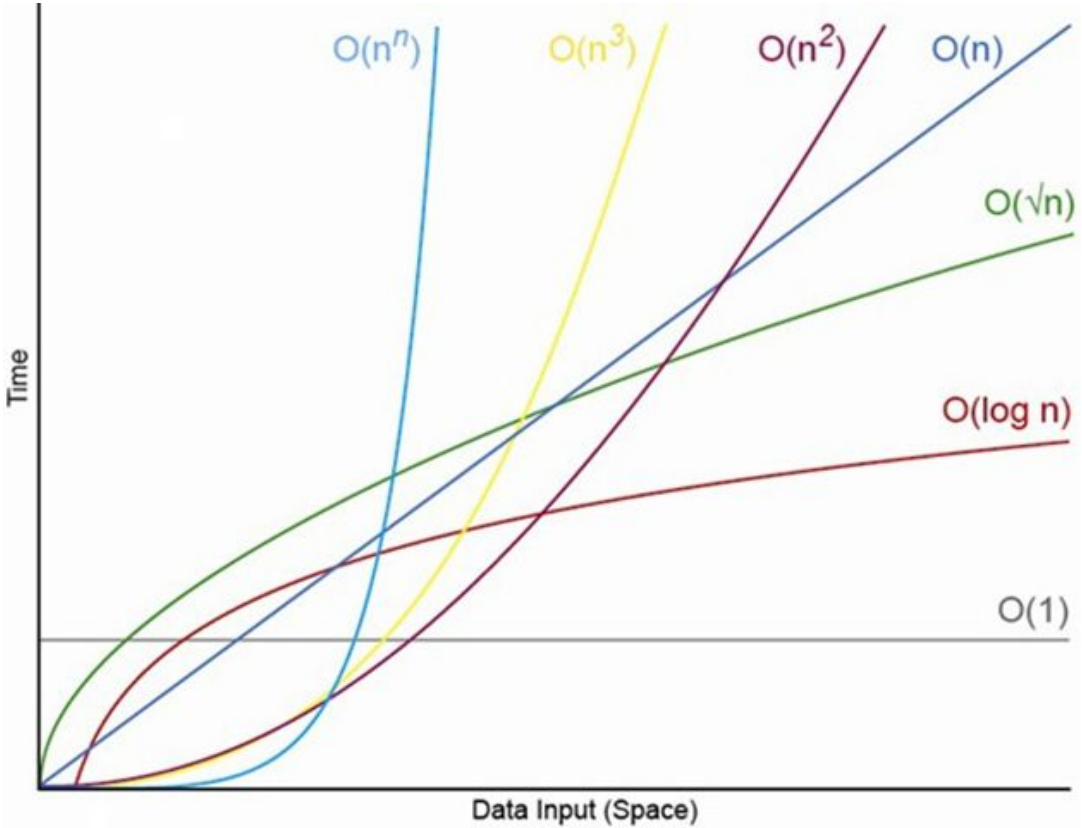
Reminder: Estimate **efficiency** of each algorithm **asymptotically** (when the input **n** gets really big)

So if you end up with an expression like **$O(6n^2+ 8n +10)$** when n gets really big the part that “weighs the most” is n^2 as it will get much bigger than the rest.

As a result simplified notation is **$O(n^2)$**

if n = 1,000,000 you don’t care that it will be $6.000008*10^{12}$ you just want to know that it is “of the order of” 10^{12} as $n^2 = 1.000000*10^{12}$

Asymptotes comparison



Calculating Time Complexity using Big O Notation

Say you have a statement, something like

```
total++;
```

Just running that statement would take a constant amount of time. O(1)

Now let’s say you want to run that statement above, but in a for loop:

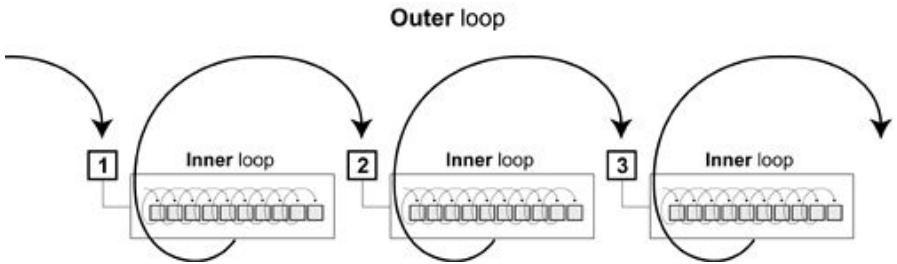
```
for(int i = 0; i < n; i++){
    total++;
}
```

Running this statement would take a constant amount of time, N times. O(n)

Calculating Time Complexity Cont.

Oftentimes you'll find you might need to have nested for loops in a program.

```
for(int i=0; i < n; i++)
{
  for(int j=0; j < n;j++)
  {
    total++;
  }
}
```



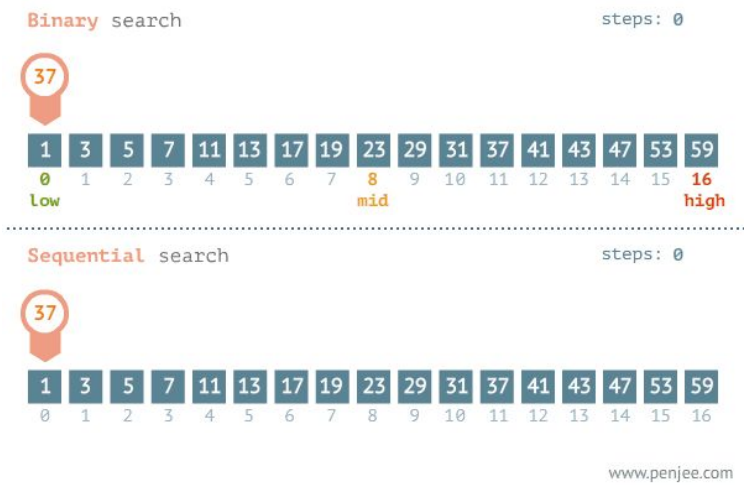
Something like this would mean that for every value in N, you are going through the entire group of N. This makes this a complexity of $O(n^2)$.

Calculating Time Complexity Cont.

The following example is trying to find the target number. Rather than increment/decrement by only 1, this example divides the distance between the low and high values by two at each iteration.

```
int target; //some number between low and high
while(low <= high)
{
  mid = (low + high) / 2;
  if (target < mid)
    high = mid;
  else if (target > mid)
    low = mid;
  else break;
}
```

This code would be O(log(n)).



Calculating space complexity

Add-up complexity for each of the variables that “live at the same time” in comparison to your input.

Example:

```
public static int getLargestItem(int[] arrayOfItems) {
    int largest = Integer.MIN_VALUE;
    for (int item : arrayOfItems) {
        if (item > largest) {
            largest = item;
        }
    }
    return largest;
}
```

O(1)

Calculating space complexity

For recursion total space used by all recursive calls active at that time

Example:

```
int fib(int n){
    int f[n+1];
    f[1] = f[2] = 1;
    for (int i = 3; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```

O(n)

Calculating space complexity

Example:

```
int fib(int n){
    int a = 1, b = 1;
    for (int i = 3; i <= n; i++) {
        int c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

O(1)

Data Structure Operation Complexity

Understanding the level of effort that goes into any operation on a data structure is important. For example, accessing the value of something stored in a particular spot in an array is a constant complexity of O(1). You just go to the spot and get the value.

For other data structures it is more complicated. With a linked list if you want to access a node, you have to traverse the linked list from the beginning until you reach the correct node. This leads to the complexity being O(n).

Inserting and Deleting values from an array are considered O(n) because you would have to move every value after the one inserted up or back one space. Inserting and Deleting with Linked Lists is constant, because you change the nodes that are referenced in the inserted/deleted node, which is a O(1) effort.

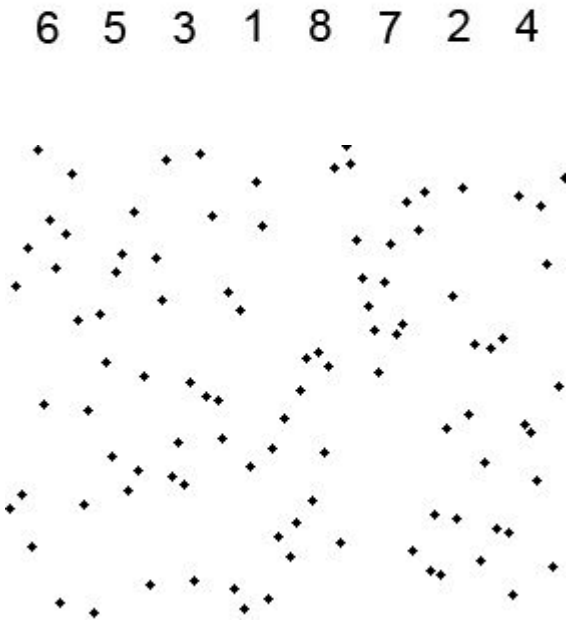
```
function insertAfter(Node node, Node newNode) // insert newNode after node
    newNode.next := node.next
    node.next    := newNode
```

```
function removeAfter(Node node) // remove node past this one
    obsoleteNode := node.next
    node.next := node.next.next
    destroy obsoleteNode
```

Calculating the Complexity of Bubblesort

A sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items, and swaps them if they are in the wrong order.

```
function bubbleSort(arr) {
  const ret = Array.from(arr)
  let swapped
  do {
    swapped = false
    for (let i = 1; i < ret.length; ++i) {
      if (ret[i - 1] > ret[i]) {
        [ret[i], ret[i - 1]] = [ret[i - 1],
ret[i]]
        swapped = true
      }
    }
  } while (swapped)
  return ret
}
```



Calculating the Complexity of Mergesort

An efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output.

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

6 5 3 1 8 7 2 4

Calculating the Complexity of Mergesort Cont.

```
function merge_sort(list m)
    // Base case. A list of zero or one elements is sorted,
by definition.
    if length of m ≤ 1 then
        return m
    // Recursive case. First, divide the list into
equal-sized sublists
    // consisting of the first half and second half of the
list.
    // This assumes lists start at index 0.
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right
    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)
```

```
function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume
them.
    // (Only one of the following loops will actually be
entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

Calculating the Complexity of Quicksort

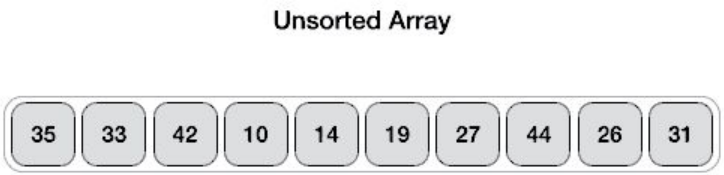
Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

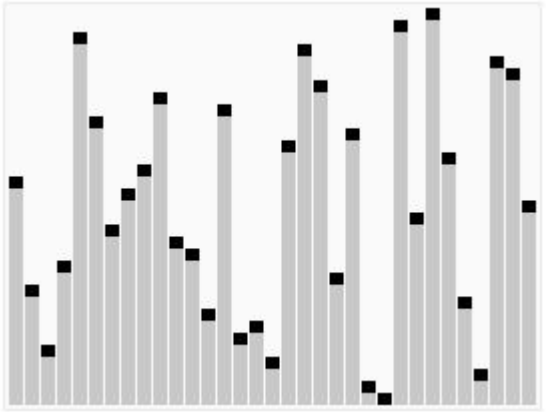
The steps are:

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.





Lomuto vs. Hoare Partition Scheme Algorithms

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo - 1
    for j := lo to hi - 1 do
        if A[j] < pivot then
            i := i + 1
            swap A[i] with A[j]
    if A[hi] < A[i + 1] then
        swap A[i + 1] with A[hi]
    return i + 1
```

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[lo]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot

        do
            j := j - 1
        while A[j] > pivot

        if i >= j then
            return j

        swap A[i] with A[j]
```

Practice

1 | The Triforce

- Problem Statement

2 | French License Plates

- Problem Statement

3 | Coding Game

- CodinGame