CS 310: Order Notation (aka Big-O and friends)

Chris Kauffman

Week 1-2

Logistics

At Home

- Read Weiss Ch 1-4: Java Review
- Read Weiss Ch 5: Big-O
- Get your java environment set up
- Compile/Run code for Max Subarray problem form first lecture

Goals

- Finish up Course Mechanics
- Basic understanding of Big O and friends

How Fast/Big?

Algorithmic time/space complexity depend on problem size

- Often have some input parameter like n or N or (M, N) which indicates problem size
- Talk about time and space complexity as *functions* of those parameters
- Example: Two algorithms to find the maximum element for an input array of size N,
  - One algorithm finds the maximum element using $5 * N + 3$ operations
  - Another finds the max element in $N^2 + 2N + 7$ operations.
- Example: Two algorithms solve the Max Sub Array problem for an input array of size N,
  - Using 7 units of memory in addition to the input array
  - Using an additional $9 + (N \times (N + 1))/2$ units of memory
- Big-O notation: bounding how fast functions grow based on input

It's Show Time!

Not *The* Big O

Just Big O

$T(n)$ is $O(F(n))$ if there are positive constants c and n_0 such that

- When $n \geq n_0$
- $T(n) \leq cF(n)$

Bottom line:

- If $T(n)$ is $O(F(n))$
- Then $F(n)$ grows as fast or faster than $T(n)$

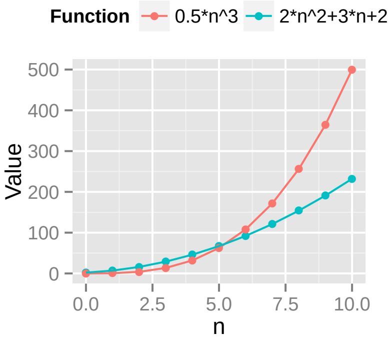
Show It

Show

$$f(n) = 2n^2 + 3n + 2 \text{ is } O(n^3)$$

- Pick $c = 0.5$ and $n_0 = 6$

n	$f(n)$	$0.5n^3$
0	2	0
1	7	0
2	16	4
3	29	13
4	46	32
5	67	62
6	92	108
7	121	171



How about the opposite? Show

$$g(n) = n^3 \text{ is } O(2n^2 + 3n + 2)$$

Basic Rules

- Constant additions disappear
  - $N + 5$ is $O(N)$
- Constant multiples disappear
  - $0.5N + 2N + 7$ is $O(N)$
- Non-constant multiples multiply:
  - Doing a constant operation $2N$ times is $O(N)$
  - Doing a $O(N)$ operation $N/2$ times is $O(N^2)$
  - Need space for half an array with N elements is $O(N)$ space overhead
- Function calls are not free (including library calls)
  - Call a function which performs 10 operations is $O(1)$
  - Call a function which performs $N/3$ operations is $O(N)$
  - Call a function which copies object of size N takes $O(N)$ time and uses $O(N)$ space

Bounding Functions

- Big O: Upper bounded by ...
  - $2n^2 + 3n + 2$ is $O(n^3)$ and $O(2^n)$ and $O(n^2)$
- Big Omega: Lower bounded by ...
  - $2n^2 + 3n + 2$ is $\Omega(n)$ and $\Omega(\log(n))$ and $\Omega(n^2)$
- Big Theta: Upper and Lower bounded by
  - $2n^2 + 3n + 2$ is $\Theta(n^2)$
- Little O: Upper bounded by but not lower bounded by...
  - $2n^2 + 3n + 2$ is $o(n^3)$

Growth Ordering of Some Functions

Name	Lead Term	Big-Oh	Example
Constant	$1, 5, c$	$O(1)$	$2.5, 85, 2c$
Log-Log	$\log(\log(n))$	$O(\log\log n)$	$10 + (\log\log n + 5)$
Log	$\log(n)$	$O(\log(n))$	$5\log n + 2$
			$\log(n^2)$
Linear	n	$O(n)$	$2.4n + 10$
			$10n + \log(n)$
N-log-N	$n\log n$	$O(n\log n)$	$3.5n\log n + 10n + 8$
Super-linear	$n^{1.x}$	$O(n^{1.x})$	$2n^{1.2} + 3n\log n - n + 2$
Quadratic	n^2	$O(n^2)$	$0.5n^2 + 7n + 4$
			$n^2 + n\log n$
Cubic	n^3	$O(n^3)$	$0.1n^3 + 8n^{1.5} + \log(n)$
Exponential	a^n	$O(2^n)$	$8(2^n) - n + 2$
		$O(10^n)$	$100n^{500} + 2 + 10^n$
Factorial	$n!$	$O(n!)$	$0.25n! + 10n^{100} + 2n^2$

Constant Time Operations

The following take O(1) Time (Constant Time)

- Arithmetic operations (add, subtract, divide, modulo)
  - Integer ops usually practically faster than floating point
- Accessing a stack variable
- Accessing a field of an object
- Accessing a single element of an array
- Doing a primitive comparison (equals, less than, greater than)
- *Calling* a function/method but NOT waiting for it to finish

The following take more than O(1) time (how much more)?

- Raising an arbitrary number to arbitrary power
- Allocating an array
- Checking if two `Strings` are equal
- Determining if an array or `ArrayList` `contains()` an object

Common Patterns

- Adjacent Loops Additive: $2 \times n$ is $O(n)$

```
for(int i=0; i<N; i++){
  blah blah blah;
}
for(int j=0; j<N; j++){
  yakkety yack;
}
```

- Nested Loops Multiplicative usually polynomial
  - 1 loop, $O(n)$
  - 2 loops, $O(n^2)$
  - 3 loops, $O(n^3)$
- Repeated halving usually involves a logarithm
  - Binary search is $O(\log n)$
  - Fastest sorting algorithms are $O(n \log n)$
  - Proofs are harder, require solving recurrence relations

Lots of special cases so be careful

Practice

Two functions to revers an array. Discuss

- Big-O estimates of runtime of both
- Big-O estimates of memory overhead of both
  - Memory overhead is the amount of memory in addition to the input required to complete the method
- Which is practically better?
- What are the exact operation counts for each method?

reverseE

```
public static
void reverseE(Integer a[]){
  int n = a.length;
  Integer b[] = new Integer[n];
  for(int i=0; i<n; i++){
    b[i] = a[n-1-i];
  }
  for(int i=0; i<n; i++){
    a[i] = b[i];
  }
}
```

reverseI

```
public static void
reverseI(Integer a[]){
  int n = a.length;
  for(int i=0; i<n/2; i++){
    int tmp = a[i];
    a[i] = a[n-1-i];
    a[n-1-i] = tmp;
  }
  return;
}
```

Much Trickier Exercise

```
public static String toString( String [ ] arr ) {
  String result = "";
  for( String s : arr ){
    result = result + s + " ";
  }
  return result;
}
```

- Give a Big-O estimate for the runtime
- Give a Big-O estimate for the memory overhead

Multiple Input Size

What if "size" has two parameters?

- $m \times n$ matrix
- Graph with m vertices and n edges
- Network with m computers and n cables between them

Exercise: Sum of a 2D Array

Give the runtime complexity of the following method.

```
public int sum2D(int [][] A){
  int M = A.length;
  int N = A[0].length;
  int sum = 0;
  for(int i=0; i<M; i++){
    for(int j=0; j<N; j++){
      sum += A[i][j];
    }
  }
  return sum;
}
```

What if I have no idea?

Analyzing a complex algorithm is hard. More in CS 483.

- Most analyses in here will be straight-forward
- Mostly use the common patterns

If you haven't got a clue looking at the code, *run it and check*

- This will give you a much better sense

Observed Runtimes of Maximum Subarray

	Figure 5.4	Figure 5.5	Figure 7.20	Figure 5.8
N	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.000001	0.000000	0.000001	0.000000
100	0.000288	0.000019	0.000014	0.000005
1,000	0.223111	0.001630	0.000154	0.000053
10,000	218	0.133064	0.001630	0.000533
100,000	NA	13.17	0.017467	0.005571
1,000,000	NA	NA	0.185363	0.056338

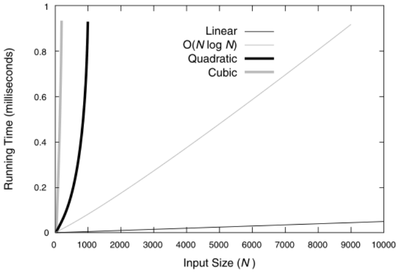
figure 5.10

Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

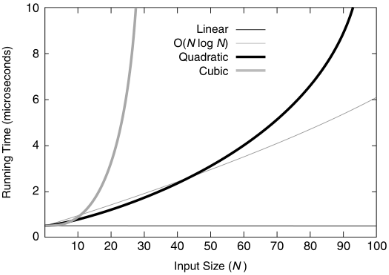
Weiss pg 203

Idealized Functions

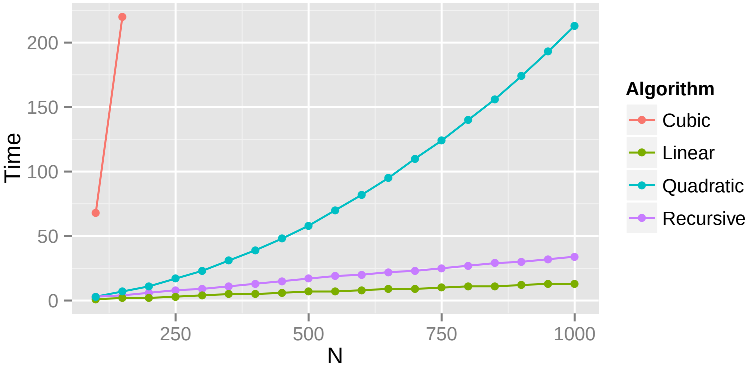
Smallish Inputs



Larger Inputs



Actual Data for Max-Subarray



- Where did this data come from?
- Does this plot confirm our analysis?
- How would we check?

Playing with `MaxSumTestBetter.java`

Let's generate part of the data, demo in
`w01-1-code/MaxSumTestBetter.java`

- Edit: Running a main, n=100 to 100,000, multipy by 10
- Try in DrJava
- Demo interactive loop

Analysis

Linear

```
> summary(linmod)

Coefficients:
             Estim Pr(>|t|)
(Intercept)   7.26   <2e-16 ***
poly(N, 1)   16.25   <2e-16 ***
poly(N, 2)   -0.34    0.287
poly(N, 3)   -0.01    0.962
```

Why these coefficients?

Quadratic

```
> summary(quadmod)

Coefficients:
             Estim Pr(>|t|)
(Intercept)  83.89   <2e-16 ***
poly(N, 1)  278.16   <2e-16 ***
poly(N, 2)   54.75   <2e-16 ***
poly(N, 3)   -0.24    0.562
```

Take-Home

Today

Order Analysis gives big picture of runtime and memory complexity of algorithms

- Different functions grow at different rates
- Big O upper bounds
- Big Theta tightly bounds
- Standard tricks to roughly figure out complexity of functions

Next Time

- What are the limitations of Big-O?
- Reading: finish Ch 5, Ch 15 on `ArrayList`
- Suggested practice: Exercises 5.39 and 5.44 which explore string concatenation, why obvious approach is slow for lots of strings, alternatives