Algorithm Analysis I

Spring 2017 - Aaron S. Crandall, PhD

Today’s Outline

- Announcements
- Thing of the Day
- Algorithm Analysis

Announcements

- TA schedules should be up today
- Programming assignment #1 is going to be given out today
    - Will be BST-height calculating and graphing
    - If you don’t have your EECS account and git.eecs.wsu.edu login done...

Thing of the Day: Fingerprint theft points to digital danger

- With off-the-shelf gear, researchers on the National Institute of Informatics in Tokyo photographed fingertips at ranges of up to three metres, and used the ensuing photos to idiot a fingerprint recognition system.
- Basically, the images are so high res that they can generate an image able to be used for authentication/login systems
- http://newsonahand.com/fingerprint-theft-poin

https://goo.gl/jkbrFw

Attendance!

Last classes recap

- We discussed:
  - 122 data structures review
  - A little on how we use time and space to compare algorithms
    - All based on input size (N)
- Git talk by Andrew -
  - Basics of using Git:
    - Clone a repository
    - Edit files
    - Add files
    - Commit changes
    - Push changes

Formal Definition of Algorithm Complexity

- T(N) = O(f(N)) when +[c, n_o] such that T(N) <= cf(N) when N >= n_o
- T(N) = Ω(g(N)) when +[c, n_o] such that T(N) >= cg(N) when N >= n_o
- T(N) = Θ(h(N)) if and only if T(N) = O(h(N)) and T(N) = Ω(h(N))
- T(N) = o(p(N)) if +[c] there exists n_o such that T(N) < cp(N) when N > n_o
  - T(N) = o(p(N)) if T(N) = O(p(N)) and T(N) != Θ(p(N))

What is T(N)? - Book is... annoying

- T(N) is the maximum time for a function to run
- It is more specific than O(N), since O(N) is only of the order:
  - T(n) = n^2 + n+1
  - O(n) = n^2
- They should have defined T(N) more clearly.

“How Long” means wallclock time here

Bounds

For f(N):

- O(g(N)) is an UPPER bound of f(N) -- “Worst case can be no more than”
- Ω(g(N)) is a LOWER bound of f(N) -- “Best case can be no faster than”

These are normally done on the order of the algorithm, not the details, such as constants. We normally only care about Big-O because it’s worst case.

- Θ(g(N)) is when O(g(N)) = Ω(g(N)) -- “It must be exactly”
    - You’ll find Theta (Θ) also used as an average case… but that’s lazy

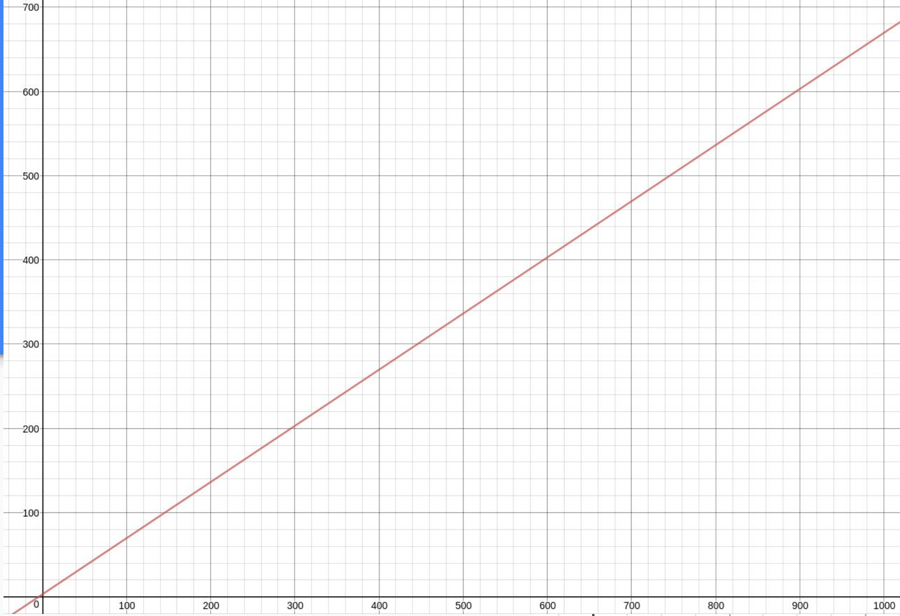
Why can we drop constants and other bounds in Big-O?

- Only the dominating factor matters for large N values:
  - 1000N vs N^2
  - 1000N +1,000,000 vs N^2
  - N^3 vs N^2 + 30,000
  - N^3 vs N^3 + N^2
- Lower-order terms can generally be ignored for Big-O analysis, and constants thrown away if there’s a higher order factor
  - We only care about the growth rate over large N values for Big-O analysis
  - There’s plenty of work in the small N space too - example is for N <= 10 in sorting

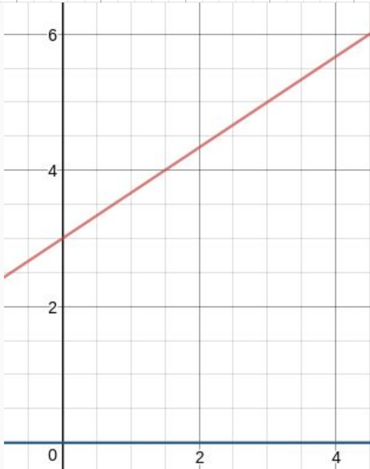
Typical Growth Rates

Function	Name
c	Constant
log N	Logarithmic
log^2 N	Log-squared
N	Linear
N log N	(We’ll see this in sorting *a lot*)
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Book example: copying data



- Copy parameters:
  - 3 second delay to initialize
  - Download is 1.5MB/s (12 Mb/s)
  - For an N MB file
- T(N) = N/1.5 + 3
- 1,500M file ~= 1,003 sec   (constant is 0.3% of the time)
- 750M file ~= 503 sec   (constant is 0.6% of the time)
  - For a large file, the 3 second startup time isn't a significant factor
  - The linear function dominates over the constant when N >= n_o



Analysis Example: Search in List

Search Problem: Given an integer k and an array of integers:

A_0 , A_1 , A_2 , A_3 , A_4… A_{N-1}

which are pre-sorted, find i such that $A_i = k$. (Return −1 if k is not in the list.)

For example, {-32, 2, 3, 9, 45, 1002}. Given that k = 9, the program will return 3. i.e., the number 9 lives in the 3rd position.

Note: start counting positions from 0.

Brute Force Search

```
public int bruteForceSearch(int k, int[] array){
        for(int i=0; i<array.length; i++){
                if(a[i] = = k){
                        return i;                       /*found it!*/
                }
        }
        return –1;                                      /*didn’t find, not in array*/
}

// Takes O(N) time
// Takes how much space?
```

An Alternative Algorithm: Binary Search

1) Start in the middle of array.
2) If that is the correct number return.
3) If not, then check if the correct number is larger or smaller than the number in the current position.
4) Take correct half of the array and go to the middle of that one.
5) Repeat.

Binary Search Example

1) Let's look for k = 54.
2) Start in middle of array

   11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83
3) Is 54 bigger than 41? Yes. So look in upper half of array.

   11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83
4) Is 54 bigger than 56? No. So take lower half of remaining array.

   11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83

Binary Search Example

5) Is 54 bigger than 51? Yes, so take upper half of remaining array.

11, 13, 21, 26, 29, 36, 40, 41, 45, 51, 54, 56, 65, 72, 77, 83

6) And 51 is in the 9th position (starting from 0 … stupid array counting).

7) Note that we decreased the size of the search by roughly ½ each step.

So here’s some code that will do this “binary search”…

The binary search code

```
public int binarySearch(int k, int[] array){
        int left = -1;
        int right = array.length;          /*left and right are the array bounds*/
        while(left+1 ! = right) {          /*stop when left and right meet */
                int middle = (left+right)/2; /*find the middle point*/
                if(k < array[middle])        /*in left half*/
                        right = middle;      /*new right is the old middle*/
                if(k == array[middle])       /*found it!*/
                        return middle;       /*new right is the old middle*/
                if(k > array[middle])        /*in right half*/
                        left = middle;       /*new left is the old middle*/
        }
        return –1;                         /*didn't find it. Not in array*/
}
```

Binary search code

Ahhh, a “while” loop. So how many times does it iterate?

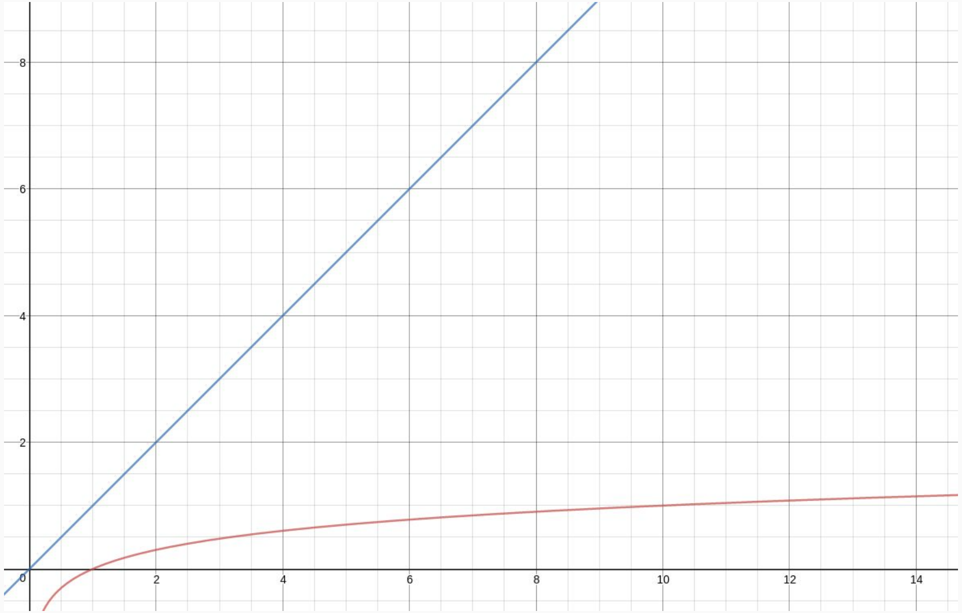
Like “for” loops the Big-O answer is just the number of passed through the loop times the most costly statement on the inside.

Binary search code

- With Big-O we are always looking for the worst case scenario.
- The worst case is that the array size has to be halved until we are down to an array size of 1 (just like the example).
- Example: Once through for size 32, then size 16, 8, 4, 2, 1.
- How many times through the loop?
- Just flip it around… to get the series: 1, 2, 4, 8, 16, 32, …, 2^{i-1}
  where i is the number of times through the loop.

Binary Search Analysis

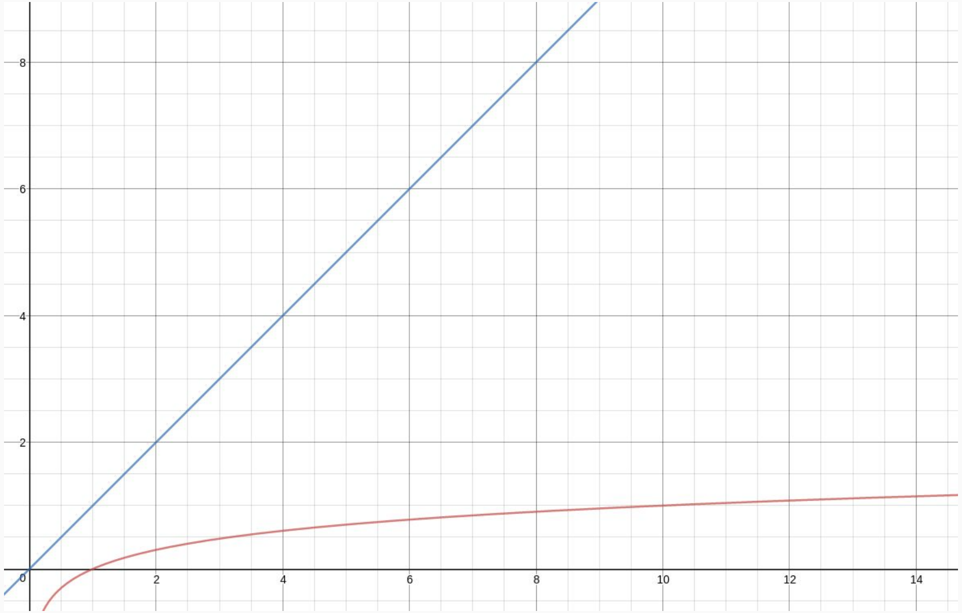
- So the array size, n = 2^{i-1} .
- So i = (log(n)/log(2)) + 1.
- So the run time is O(log(n)).
- And how does that compare to the BruteForceSearch algorithm which is O(n)?



Binary Search Analysis

- So the array size, n = 2^{i-1} .
- So i = (log(n)/log(2)) + 1.
- So the run time is O(log(n)).
- And how does that compare to the BruteForceSearch algorithm which is O(n)?

- Binary search definitely wins



The Core Lesson

- If a loop is halved over and over or doubled over and over, it is some form of O(log(N)).
  - Possibly O(e^N) if it’s a really bad algorithm, like recursive Fibonacci
- So, if a loop increases by a constant multiplicative factor each iteration, it’s O(log(N))

The log(N) example

```
for(int i = 1; i<n; i *= 37){
        Total++;
}
```

Claim: i increases by a factor of 37 each time, so takes log(N) time.

Proof:

i = 1, 37, 37^2 , 37^3 , … 37^{k-1} where 37^{k-1} is the last number that doesn't exceed n

(k is the number of iterations).

So 37^{k-1} ≤ n which means log(37^{k-1}) ≤ log(n). Therefore, k-1≤ log(n)/log(37). So the *max* number of iterations is k = (log(n)/log(37))+1. Therefore the runtime is O(log(n)).

What about linear behavior?

```
for(int i = 0; i<n; i += 2){
	total++;
}
```

Increases by 2 each time, but not by a <u>multiplicative</u> factor of 2. So not log(n).

What is the run time?

i = 0, 2, 4, 6, 8, … So this will run for n/2 iterations. So the runtime is O(n). The constant value (the div 2) is dropped in Big-O notation because it's a constant scaling factor not influenced by how big n is (it does matter for T(N)).

Increasing by constant time

- When a loop increases or decreases by a constant amount each iteration, then its growth rate is O(N).
- Example:

```
for(float x = 27.2; x > -n; x -= log(1.3))
    { total++; }
```

Is that log(1.3) going to introduce a O(log(N)) kind of behavior?

A common situation: Simple iterative loop

```
for(int i = 1; i < n; i++){
	for(int j = 1; j < n; j++){
		total++;
	}
}
```

Remember how I said that you work out how many iterations a loop goes, then multiply that by the largest Big-O factor it has inside of the loop?

A common situation: Simple iterative loop

```
for(int i = 1; i < n; i++){
	for(int j = 1; j < n; j++){
		total++;
	}
}
```

Outer loop goes n times. Inner loop goes n times. n*n means:

O(N^2)

Another common situation

```
for(int i = 1; i < n; i++){
	for(int j = 1; j < n; j *= 2){
		total++;
	}
}
```

Another common situation

```
for(int i = 1; i < n; i++){
    for(int j = 1; j < n; j *= 2){
        total++;
    }
}
```

Outer loop goes n times. Inner loop goes log(n) times so: n * log(n)

$O(N \log(N))$

How about some more complexity?

```
for(int i = 1; i<n; i++){                     // Outer loop goes n times
        for(int j = 1; j<n; j*=2){            // InnerA goes log(n) times
                total++;                      // Cost == O(1)
        }
        for(int k = 1; k<n; k++){             // InnerB goes n times
                total++;                      // Cost == O(1)
}       }
```

Result: O(n) * (O(log(n)) + O(n)) → O(n^2) + O(n log(n)) *Which one dominates?*

How about some more complexity?

```
for(int i = 1; i < n; i++){                 // Outer loop goes n times
    for(int j = 1; j < n; j*=2){            // InnerA goes log(n) times
        total++;  }                         // Cost == O(1)
    for(int k = 1; k < n; k++){             // InnerB goes n times
        total++;  }                         // Cost == O(1)
}
for(int x = 1; x < n; x++){ total++; }      // Runs n times
```

Result: O(n) * (O(log(n)) + O(n)) + O(n)

O(n^2) + O(n log n) + O(n)

In Summary (if we even get here!)

- Big-O is the asymptotic run time for an algorithm
  - (once N gets “big enough”)
- All lower run time elements in the analysis can be dropped for a large N
- Halving work each time gets O(log(N))
- Increasing in a linear fashion gets you O(N)

NO CLASS MONDAY - I’ll be reverse engineering laser tag technology

Start reading chapter 3 and review chapter 2 again - it’s heavy material