Algorithm Analysis II

Spring 2017 - Aaron S. Crandall, PhD

Today’s Outline

- Announcements
- Thing of the Day
- Algorithm Analysis - More examples

Announcements

- PA1 is in your repos. It’s due Sunday the 17th at 11:59pm.
- MA2 is in your repos. It’s due Wednesday the 13th at 11:59pm.
- Open lab / Tutor is: Kimi Phan <kimberlee.phan@wsu.edu>
  - Hours: T/Th 5-8 pm in Sloan 353 (center room)
- LUG should still be in Dana on Saturday doing tutoring
- Internship panel with Google, SEL and Student Interns
  - Sept 12th - 5:30-7:00pm in Stephenson Hall Lounge
  - More events via the PPEL site: https://vcea.wsu.edu/ppel/

Thing of the Day: This is a fish controlling where it goes

- Vision processing by overhead camera
- Robotics system for motor controls
- Basically, it’s a reverse submarine now

Some pointers on using Git in this class

- I push your assignments to your repos using a script
- Each assignment will go on it's own branch (PA1, MA2, etc)
- You can work entirely in that branch or merge it into the master, your call
    - I would probably keep them in their own branches until they're completed
- To change which branch you're on: git checkout [branch name]
    - You can easily see which branches there are on the web interface, or git branch -a
    - If you commit and push only in a branch it can't affect any of your other branches
- How to find your commit hashes? git log

Last class recap

- We discussed:
  - We started on what Big-O is
  - Comparing function growth over input size
  - Some ways to derive the Big-O for a given algorithm

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

- When looking at the running time, 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, and constants thrown away
  - We only care about the growth rate over large N values for analysis

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

Bubblesort

How long does this take?

```
#include<iostream>
using namespace std;
int main(){
    int a[50],n,i,j,temp;
    for(i=1;i<n;++i){
      for(j=0;j<(n-i);++j)
            if(a[j]>a[j+1]) {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
       }
    }
}
```

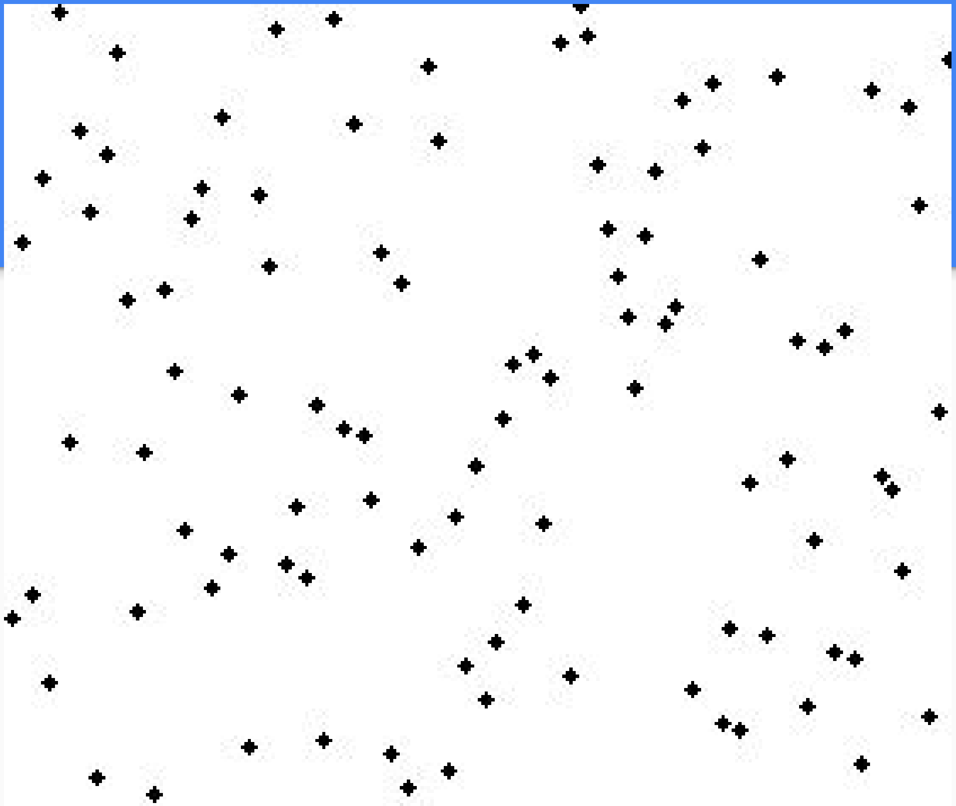
How about this one?

```
void bubblesort( vector & a ) {
    for( int i = 1; !clean_pass && i < list.size(); i++ ) {
        clean_pass = 1;
        for(int j=0; j < (list.size() - i); j++)
            if( list[j] > list[j+1] ) {
                int temp = list[j];
                list[j] = list[j+1];
                list[j+1] = temp;
                clean_pass = 0;
            }
        }
    }
}
```

BS visualized

Rabbits vs. Turtles in Bubble Sort

- Values move quickly down the list
- Values move slowly up the list
- Rabbits vs. Turtles
- This kind of behavior has implications for your algorithms and expected behaviors.
- The algorithm you use for a given application can matter, especially given time constraints.

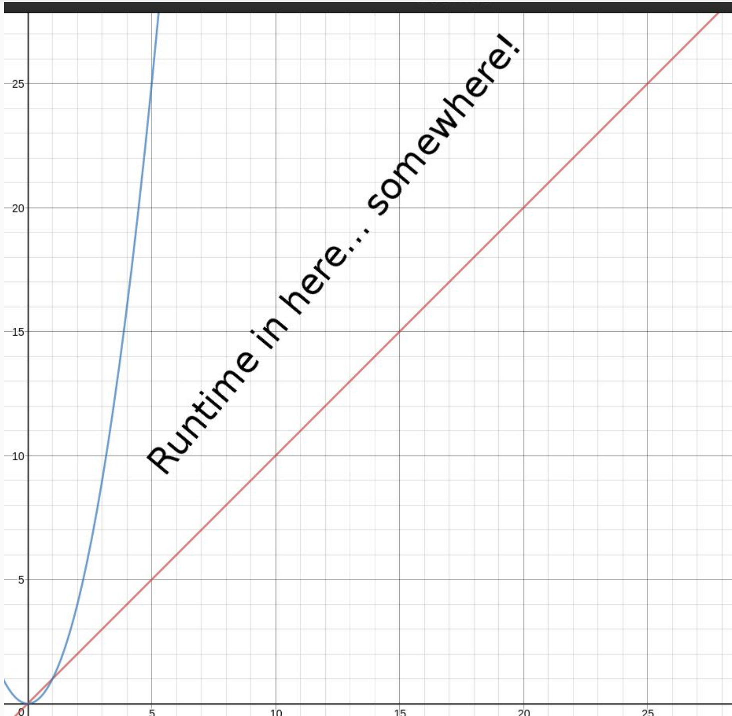


Using the time command

- Linux (unix) has a ‘time’ command to time how long it takes to run a process.
- time [program with options]
- Will be used in assignments to clock program execution
    - Real -> Wallclock time from start to end of program
    - User -> Actual processing time of program
    - Sys -> Kernel processing time for the program

Why does the time vary with executions?

- Real definitely varies because other programs happen
- User time varies because of input variability
  - Input can make algorithms vary radically!
    - Bubblesort goes from N to N^2
- Sys varies if kernel needs to do extra bookkeeping while your program runs
  - Memory management, I/O operations, definitely networking overhead



Now for some C++11

- new and delete
  - Or… the source of much memory leakage.
  - Talk to your doctor about remedies
- lvalues and rvalues
  - New reference types! Welcome to 1990’s
  - Named variables are lvalues
  - Literals are rvalues
- lvalue references
  - type & varname
  - Makes an alias to the variable varname
- rvalue references
  - type && varname
  - Alias to an rvalue (can be temp or const)

```
auto whichList = theLists[ myhash( x,
                    theLists.size( ) ) ];
```

(badness - makes a copy)

```
auto & whichList = theLists[ myhash( x,
                    theLists.size( ) ) ];
```

(goodness - makes an lvalue reference)

Better loops! A foreach loop is wunderbar

```
for( auto x : arr )  // broken - makes copies of x
        ++x;

for( auto & x : arr )  // works! - makes refs to x
        ++x;


swap( x, y );
void swap( double & a, double & b );
```

Rvalue option for parameter passing:

```
string randomItem( vector<string> && arr );
 // Primarily used to to a std::move on arr
 // This is important if passed a const or
temporary thing instead of a larger lvalue item
```

C++11 - Big Five

- Copy Constructor
- Move Constructor
- Copy Assignment Operator
- Move Assignment Operator
- Destructor

std::move() and std::swap() are new C++11 features. They're designed to make things significantly more efficient in data operations with objects.

```
class foo
{
  public:
    foo()
      : p{new resource{}}
    { }

    foo(const foo& other)
      : p{new resource{*(other.p)}}
    { }

    foo(foo&& other)
      : p{other.p}
    {
      other.p = nullptr;
    }

    foo& operator=(const foo& other)
    {
      if (&other != this) {
        delete p;
        p = nullptr;
        p = new resource{*(other.p)};
      }

      return *this;
    }
```

```
    foo& operator=(foo&& other)
    {
      if (&other != this) {
        delete p;
        p = other.p;
        other.p = nullptr;
      }

      return *this;
    }

    ~foo()
    {
      delete p;
    }

  private:
    resource* p;
};
```

There’s LOTS more about C++11 we could cover, but these are the key elements

- The spec tried to address many efficiency advantages of references
- Many languages use these, but C++ wasn’t up to par
- Modern C++ code will definitely be using the lvalue and rvalue styles so getting a grip on them will be very important in a career

Now, back to some algorithm analysis review & new materials

The IF statement rule

```
if( condition )
        S1
Else
        S2
```

* Use the larger of S1 or S2
* Yes, if you know the ratio of the two you could do a deeper analysis for a tighter bound, but the default is to just take the larger branch cost

A different example (and why)

```
function sample(k)
  if k < 2
     return 0
  return 1 + sample(k/2)
```

- What is the time complexity of this algorithm?
- What is the space complexity? … or: what Crandall hasn’t shown you yet!
- What if it was sample(k/3)?

When worst case analysis breaks down

- Big-O is often much bigger than average running time
- This can be an opportunity to empirically derive running behavior to update your O(N) calculations. For example:
  - Determining the ratio of S1:S2 in IF statements
  - Getting a distribution of orderliness in input data (pre-sorted or not)
  - This is not always possible since you’re now working in distributions, so you get stuck with worst case analysis

What to analyze in an algorithm?

- Options include:
  - T_ave(N)
  - T_worst(N)
  - T_optimal(N)
- T_optimal(N) <= T_ave(N) <= T_worst(N)
- Do implementation details matter for algorithms analysis?
  - Copying big arrays vs. pass by reference example
  - TL;DR: No, implementation isn’t about algorithm analysis
  - That said: it matters in the real world when you code

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.

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.
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, $\boxed{41}$, 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, $\boxed{56}$, 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, $\boxed{51}$, 54, 56, 65, 72, 77, 83

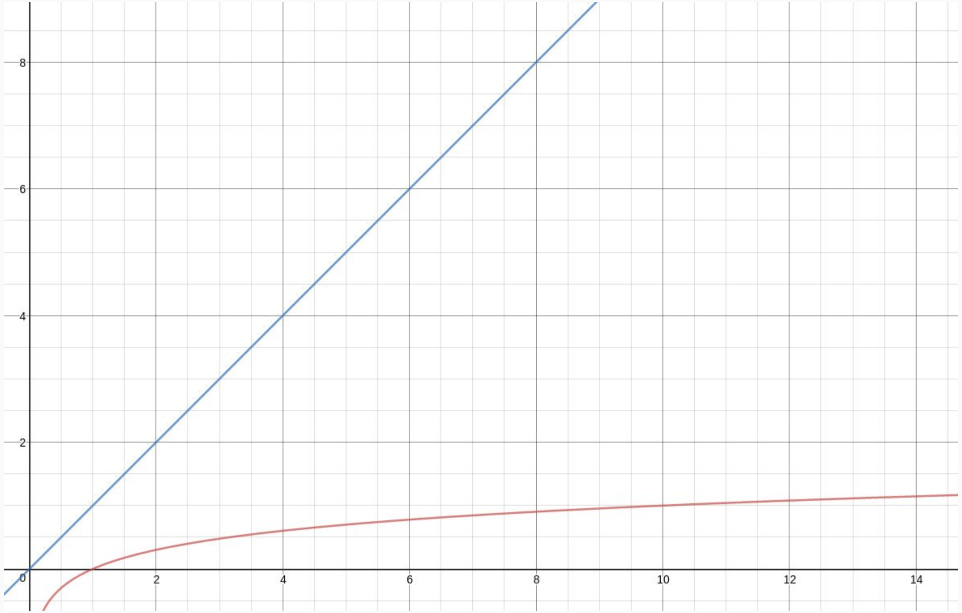
Binary Search Code

- With Big-O we are normally 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… 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 runtime is O(log(n)).
- And how does that compare to the BruteForceSearch algorithm which is O(n)?

- BinarySearch wins!



The Core Lesson

- If a loop is halved over and over or doubled over and over, it is O(log(N)).
    - Possibly O(e^N) if it’s a really bad algorithm, like recursive Fibonacci
- 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 jumps in each iteration?

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

Increases by 2 each time, but not by a multiplicative 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.

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

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

What’s the runtime?

A common situation

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

What's the runtime? O(N^2)

O(N) * O(N) = O(N^2)

Another common situation

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

What’s the runtime?

Another common situation

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

What’s the runtime? O(N log(N))

O(log(N)) * O(N) = O(N log(N))

What about this one?

- What was the runtime of binarySearch?

```
for(int i = 1; i<n; i++) {
    for(int j = 1; j<n; j+=2) {
        binarySearch(preSortedArray, j);
    }
}
```

So… what would the total runtime be?

What about this one?

- What was the runtime of binarySearch?

```
for(int i = 1; i<n; i++) {
	for(int j = 1; j<n; j+=2) {
		binarySearch(preSortedArray, j);
	}
}
```

So… what would the total runtime be? O(log(N) * N * N) -> O(N^2 log(N))

How about this one?

```
int counter = 1;
while(counter < n) {
	binarySearch(preSortedArray, counter);
	counter *= 2;
}
```

How about this one?

```
int counter = 1;
while(counter < n) {
        binarySearch(preSortedArray, counter);
        counter *= 2;
}
```

The loop takes O(log(N)) time and the binary search takes O(log(N)) time, so it becomes O(log(N)^2)

In Summary (if we even get here!)

- Big-O is the asymptotic run time for an algorithm
- 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)

Friday:

Looking at C++11 and the Big-Five

std::swap & std::move