Asymptotic Analysis

Anna Baynes

CSC 130

Programming via Recursion

- Write a recursive function to find the sum of the first n integers stored in array v

```
int findSum(int v[], int n)
{
   if (n <= 0)
      return 0
   else (findSum(v, n-1) + a(n-1));
}
```

Proof by Induction

- **Basis Step:** The algorithm is correct for a base case or two by inspection
- **Inductive Hypothesis (n=k):** Assume that the algorithm works correctly for the first k cases
- **Inductive Step (n=k+1):** Given the hypothesis above, show that the k+1 case will be calculated correctly

Program Correctness by Induction

- Basis Step: sum(v,0) = 0
- Inductive Hypothesis (n=k): Assume sum(v,k) correctly returns sum of first k elements of v, i.e. v[0] + v[1] + … v[k-1]
- Inductive Step (n=k+1): sum(v,n) returns v[k]+sum(v,k) = (by inductive hypothesis) v[k]+(v[0]+v[1]+...+v[k-1]) = v[0]+v[1]+...+v[k-1]+v[k]

Asymptotic Analysis

- Rough Estimate
- Ignores Details
  - Or really: independent of details
- What are some details we should ignore?

Big-O Analysis

- What are some details we should ignore?
  - Speed of machine
  - Programming language used
  - Amount of memory
  - Order of input
  - Size of input Compiler

Analysis of Algorithms

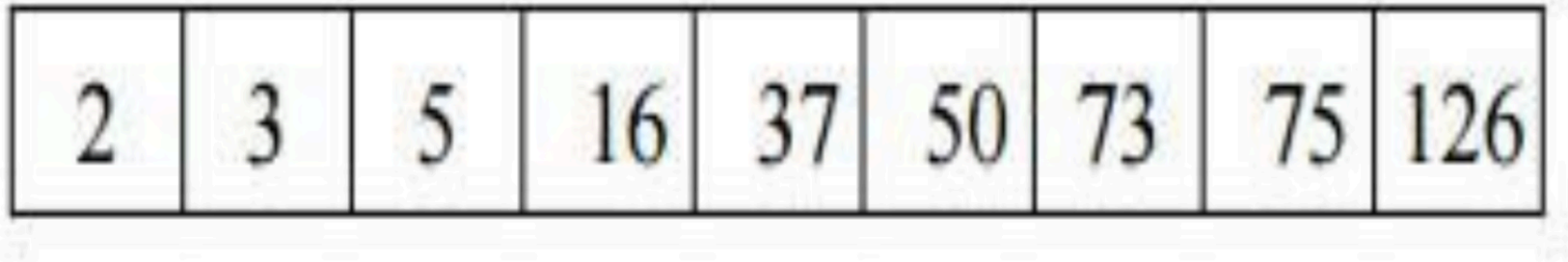
- Efficiency measure
  - How long the program runs (time complexity)
  - How much memory it uses (space complexity)
- Why analyze at all?
  - Decide which one to implement before going to the trouble
  - Given code, idea of where bottlenecks will be – without running and timing

Asymptotic Analysis

- One “detail” we won't ignore – problem size, # elements
- Complexity as a function of input size n
  - T(n) = 4n+5
  - T(n) = 0.5n log n – 2 n +7
  - T(n) = 2^n + n^3 +3n
- What happens as n grows?

Why Asmptotic Analysis?

- Most algorithms are fast for small n
  - Time difference too small to be noticeable
  - External things dominate (OS, disk I/O, …)
- BUT n is often large in practice
  - Databases, internet, graphics, …
- Time difference really shows up as n grows!




```
bool ArrayFind (int array[ ], int n, int key) {



}
```

Linear Search Analysis

```
bool LinearArrayFind(int array[ ], int n, int key ) {
   for( int i = 0; i < n; i++ ) {
      if( array[i] == key )
         // Found it!
         return true;
      }
   return false;
}
```

Best case?

Worst case?

Analyzing Code

- **Basic Java operations** - Constant Time
- **Consecutive statements –** Sum of times
- **Conditionals –** Larger branch plus test
- **Loops –** Sum of iterations
- **Function calls –** Cost of function body
- **Recursive functions –** Solve recurrence relation, Number of calls * work for each call

Binary Search Analysis

```
bool BinArrayFind(int array[ ], int low, int high, int
key ) {
   // The subarray is empty
   if( low > high ) return false;
  //Search this subarray recursively
   int mid = (high + low) /2;
   if( key == array[mid] ) return true;
   else if (key < array[mid]){
       return BinArrayFind( array, low, mid-1, key);
   else {
       return BinArrayFind( array, mid+1, high, key);
    }
}
```

Solving Recurrence Relations

- It takes O(1) time to do the comparisons, then it cuts the search range in half
  - $T(N) = T(N/2) + 1$
- Repeat the recurrence (basically expanding the relation)...
  - $T(N) = T(N/4) + 2$
  - $= T(N/8) + 3$
  - $= T(N/2^k) + k$
- Round up N to the nearest power of 2: $N \leq 2^m$
  - $T(N) \leq T(2^m/2^k)+k$
- Let k = m
  - $T(N) \leq T(2^m/2^m)+m = T(1) + m = 1 + m = O(m)$
  - If $N=2^m$, then making m = log N. So **T(N) = O(log N)**

Linear Search vs Binary Search

- Linear Search vs Binary Search
  - Linear: Best case 4 at [0], Worst 3n+2
  - Binary: Best case 4 at [mid], Worst 4 log n +4

- Which algorithm is better? What tradeoffs can you make?

Asymptotic Analysis

- Asymptotic analysis looks at the order of the running time of the algorithm
  - A valuable tool when the input gets “large”
  - Ignores the *effects of different machines* or *different implementations of the same algorithm*

- Intuitively, to find the asymptotic runtime, throw away the constants and low-order terms
  - Linear search is T(n) = 3n + 2 becomes O(?)
  - Binary search is T(n) = 4 log n + 4 becomes O(?)

Big-O Common Names

- constant: $O(1)$
- logarithmic: $O(\log n)$
- linear: $O(n)$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$ (k is a constant)
- exponential: $O(c^n)$ (c is a constant > 1)

Remember

- The fastest algorithm has the slowest growing function for its runtime

Asymptotic Analysis

- Eliminate low order terms
  - 4n + 5 =>
  - 0.5 n log n + 2n + 7 =>
  - n^3 + 2^n + 3n =>

- Eliminate coefficients
  - 4n =>
  - 0.5 n log n =>
  - n log n^2 =>

Properties of logs

- We will assume logs to base 2 unless specified otherwise
- log AB = log A + log B
- Log A/B = log A – log B
- log(A^B) = B log A
- Any base k log is equivalent to base 2

Definition of Order Notation

- Upper bound: $T(n) = O(f(n))$ Big-O
  - Exist constants c and n' such that
    - $T(n) <= c\ f(n)$ for all $n >= n'$
- Lower bound: $T(n) = \Omega(g(n))$ Omega
  - Exist constants c and n' such that
    - $T(n) >= c\ g(n)$ for all $n >= n'$
- Tight bound $T(n) = \Theta(f(n))$ Theta
  - When both hold: $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Order Notation: Definition

O(f(n)) : a set or class of functions

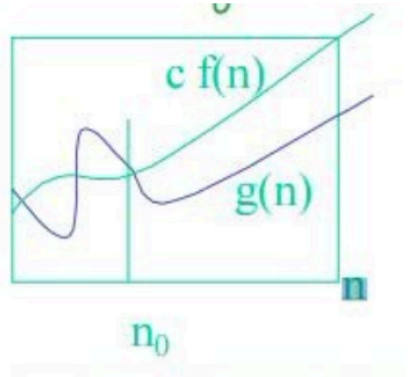
$g(n) \in O(f(n))$ iff there exist consts c and n_0

Such that:

$g(n) \leq c\, f(n)$ for all $n \geq n_0$

Example

- g(n) =1000n vs. f(n) = n^2
- Is g(n) $\in$ O(f(n)) ?
  - Pick: n_0 = 1000, c = 1
  - 1000n ≤ 1 * n^2 for all n ≥ 1000
  - So g(n) $\in$ O(f(n))

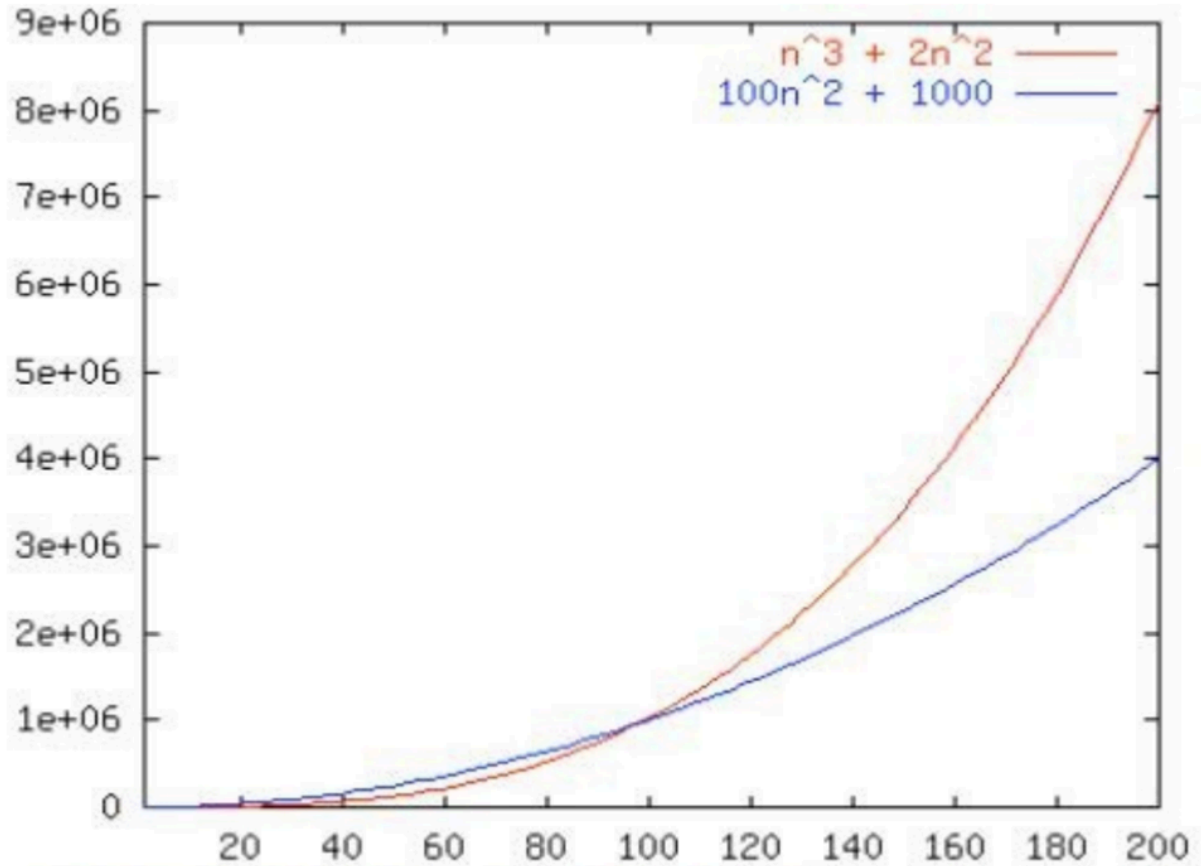


- Small cases, really don't matter. As long as it's eventually an upper bound, it fits the defintion
- If f(n) is in O(n)... what about is f(n) in O(n^2)?

Notation Notes

- Note: Sometimes, you'll see the notation:
  - $g(n) = O(f(n))$.
- This is equivalent to:
  - $g(n) \in O(f(n))$.

Order Notation: Example



$100n^2 + 1000 \leq 5\ (n^3 + 2n^2)$ for all $n \geq 19$

So $f(n) \in O(\ g(n)\ )$

Meet the Family

- **O(f(n))** is the set of all functions asymptotically less than or equal to f(n)
  - **o(f(n))** is the set of all functions asymptotically strictly less than f(n)
- **Ω(f(n))** is the set of all functions asymptotically greater than or equal to f(n)
  - **ω(f(n))** is the set of all functions asymptotically strictly greater than f(n)
- **θ(f(n))** is the set of all functions asymptotically equal to f(n)

Big-Omega et al. Intuitively

Asymptotic Notation	Mathematics Relation
O	$\leq$
Ω	$\geq$
θ	$=$
o	$<$
ω	$>$

Kinds of Asymptotic Analysis

- Running time may depend on actual data input, not just length of input
- Distinguish
  - worst case
    - your worst enemy is choosing input
  - best case
  - average case
    - assumes some probabilistic distribution of inputs
  - amortized
    - average time over many operations

Types of Analysis

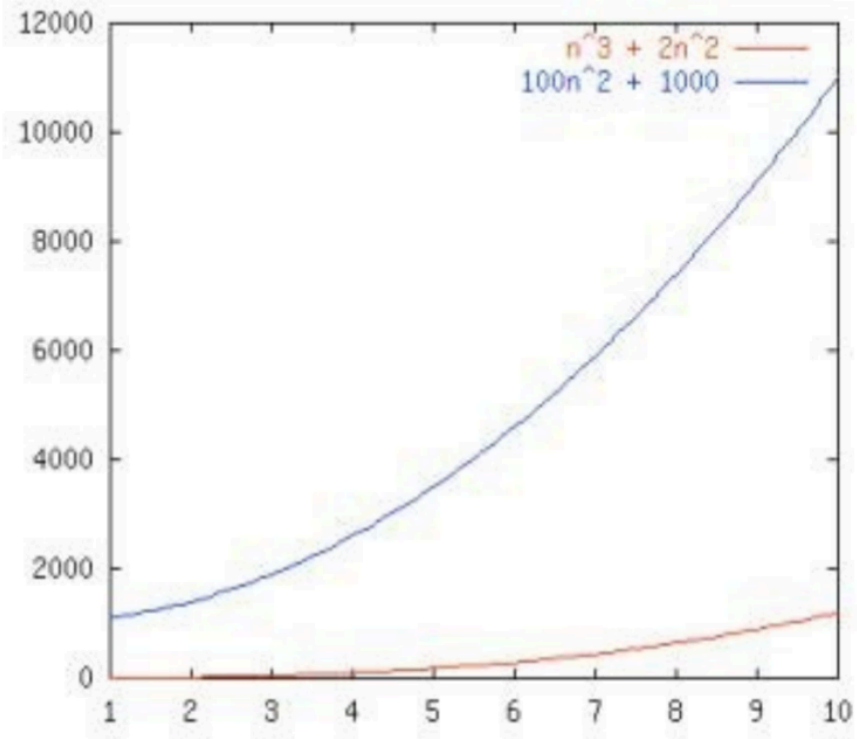
- Two orthogonal axes:
  - bound flavor
    - upper bound (O, o)
    - lower bound (Ω, ω)
    - asymptotically tight (θ)
  - analysis case
    - worst case (adversary)
    - average case
    - best case
    - “amortized”

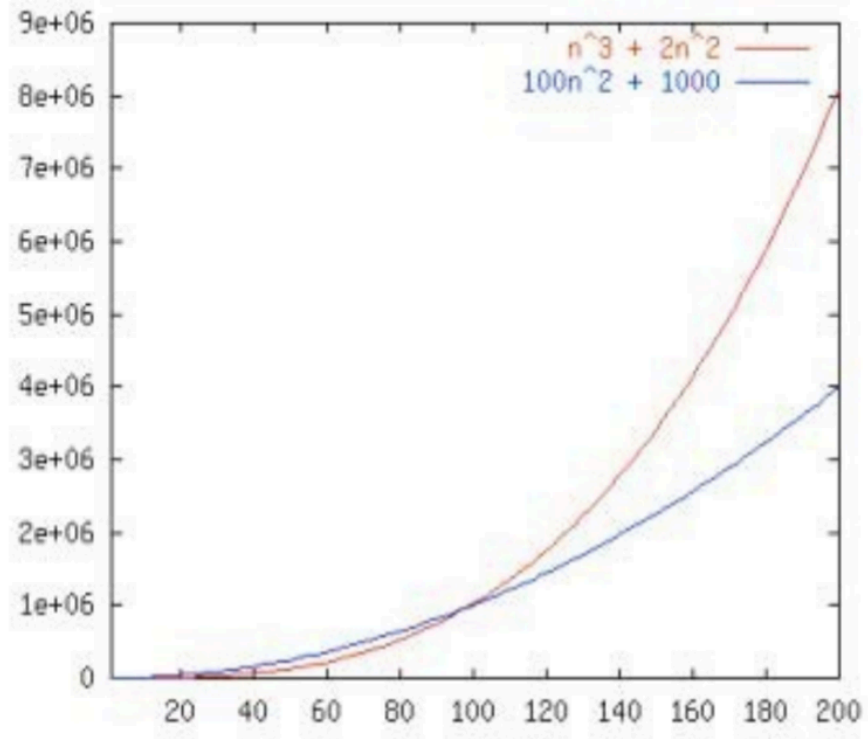
Which Function Grows Faster

$n^3 + 2n^2$ vs. $100n^2 + 1000$

Which Function Grows Faster

$n^3 + 2n^2$ vs. $100n^2 + 1000$





Nested Loops

```
for i = 1 to n do
    for j = 1 to n do
        sum = sum + 1
```

Nested Loops

```
for i = 1 to n do
    for j = 1 to n do
        if (cond) {
            do_stuff(sum)
        } else {
            for k = 1 to n*n
                sum += 1
```

Interview Question of the Day

- We're storing data for 100,000 restaurants. What is the restaurant with the best recommendation?
  - What questions can you ask? How can we do this?
- Now find the 30 best restaurants
- Now find the top N best restaurants

Interview Question of the Day

- We're storing data for 100,000 restaurants. What is the restaurant with the best recommendation?
  - Look at the number of positive reviews left for each restaurant
  - Iterate over the array O(n)
- Now find the 30 best restaurants
  - Iterate over it 30 times... O(n) 30 is constant