

Asymptotic Analysis

CSE 373
Data Structures & Algorithms
Ruth Anderson
Winter 2012

Today's Outline

- **Announcements**
 - Assignment #1, due Thurs, Jan 12 at 11pm
 - Assignment #2, posted soon, due Fri Jan 20 at BEGINNING of lecture
- **Algorithm Analysis**
 - How to compare two algorithms?
 - Analyzing code
 - Big-Oh

1/11/2012

cse 373 12wi - Asymptotic Analysis

2

Comparing Two Algorithms...

1/11/2012

cse 373 12wi - Asymptotic Analysis

3

What we want

- Rough Estimate
- Ignores Details

1/11/2012

cse 373 12wi - Asymptotic Analysis

4

Big-O Analysis

- Ignores "details"

1/11/2012

cse 373 12wi - Asymptotic Analysis

5

Gauging performance

- Uh, why not just run the program and time it?
 - Too much variability; not reliable:
 - Hardware: processor(s), memory, etc.
 - OS, version of Java, libraries, drivers
 - Programs running in the background
 - Implementation dependent
 - Choice of input
 - Timing doesn't really evaluate the *algorithm*; it evaluates an *implementation* in one very specific scenario

1/11/2012

cse 373 12wi - Asymptotic Analysis

6

Comparing algorithms

- When is one *algorithm* (not *implementation*) better than another?
- Various possible answers (clarity, security, ...)
 - But a big one is *performance*: for sufficiently large inputs, runs in less time (our focus) or less space

We will focus on large inputs (*n*) because probably any algorithm is "plenty good" for small inputs (if *n* is 10, probably anything is fast enough)

Answer will be *independent* of CPU speed, programming language, coding tricks, etc.

Answer is general and rigorous, complementary to "coding it up and timing it on some test cases"

- Can do analysis before coding!

Why Asymptotic 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!

Analyzing code ("worst case")

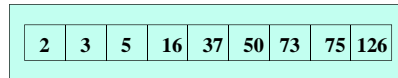
Basic operations take "some amount of" *constant time*

- Arithmetic (fixed-width)
- Assignment
- Access one Java field **or array index**
- Etc.

(This is an *approximation*.)

Consecutive statements	Sum of times
Conditionals	Time of test plus slower branch
Loops	Sum of iterations
Calls	Time of call's body
Recursion	Solve <i>recurrence equation</i>

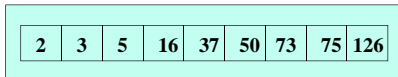
Example



Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    ???
}
```

Linear search

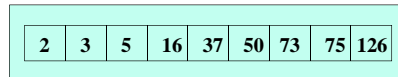


Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    for(int i=0; i < arr.length; ++i)
        if(arr[i] == k)
            return true;
    return false;
}
```

Best case:
Worst case:

Linear search



Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    for(int i=0; i < arr.length; ++i)
        if(arr[i] == k)
            return true;
    return false;
}
```

Best case: 6ish steps = $O(1)$
Worst case: 6ish*(arr.length) = $O(arr.length)$

Binary search

2	3	5	16	37	50	73	75	126
---	---	---	----	----	----	----	----	-----

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    return help(arr,k,0,arr.length);
}
boolean help(int[] arr, int k, int lo, int hi) {
    int mid = (hi+lo)/2; //i.e., lo+(hi-lo)/2
    if(lo==hi) return false;
    if(arr[mid]==k) return true;
    if(arr[mid]< k) return help(arr,k,mid+1,hi);
    else return help(arr,k,lo,mid);
}
```

1/11/2012

cse 373 12wi - Asymptotic Analysis

13

Binary search

Best case: 8ish steps = $O(1)$

Worst case: $T(n) = 10ish + T(n/2)$ where n is $hi-lo$

- $O(\log n)$ where n is `array.length`
- Solve *recurrence equation* to know that...

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    return help(arr,k,0,arr.length);
}
boolean help(int[] arr, int k, int lo, int hi) {
    int mid = (hi+lo)/2;
    if(lo==hi) return false;
    if(arr[mid]==k) return true;
    if(arr[mid]< k) return help(arr,k,mid+1,hi);
    else return help(arr,k,lo,mid);
}
```

1/11/2012

cse 373 12wi - Asymptotic Analysis

14

Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
 $T(n) = 10 + T(n/2)$ $T(1) = 13$ "ish"
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case

1/11/2012

cse 373 12wi - Asymptotic Analysis

15

Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
 $T(n) = 10 + T(n/2)$ $T(1) = 13$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
 $T(n) = 10 + 10 + T(n/4)$
 $= 10 + 10 + 10 + T(n/8)$
 $= \dots$
 $= 10k + T(n/(2^k))$
3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case
 - $n/(2^k) = 1$ means $n = 2^k$ means $k = \log_2 n$
 - So $T(n) = 10 \log_2 n + 13$ (get to base case and do it)
 - So $T(n)$ is $O(\log n)$

1/11/2012

cse 373 12wi - Asymptotic Analysis

16