Complexity Analysis

CSE 373
Data Structures & Algorithms
Linda Shapiro
Spring 2013

Today's Outline

- **Announcements**
  - Assignment #1, due Fri, April 12 at 11pm
  - Assignment #2, posted Fri April 12, due Friday April 19
- **Algorithm Analysis**
  - How to compare two algorithms?
  - Analyzing code
  - Big-Oh

Comparing Two Algorithms…

- How do *you* do it?
- Two algorithms for finding the nth value in an array:

1. for i := 0 to n-1 do {temp := v[i]}; return temp;

2. return v[n-1];

What we want

- Rough Estimate
- Ignores Details
- What does rough mean?

- Why do we ignore details?

Big-O Analysis

- Ignores "details"
- What do I mean by O(n)? O(n^2)? O(1)?

Definition of BIG OH

- Suppose I analyze my code and find it runs in time proportional to some function T(N)
- e.g. $T(N) = 4N^2 + 6N + 85$

- Def. "BIG OH"
  T(N) = O(f(N)) if there are positive constants c and n_0 such that T(N) <= cf(N) when N >= n_0.
- For N >= 1
  $T(N) = 4N^2 + 6N + 85 <= 4N^2 + 6N^2 + 85N^2 = 95N^2$
  $T(N) = O(N^2)$
- T(N) is "order N^2". It is dominated by the N^2 term.

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

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!

What is 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!

- So we want to look at what happens 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 *recurrence equation*

Example

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){
   ???
}
```

Linear 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){
   for(int i=0; i < arr.length; ++i)
      if(arr[i] == k)
        return true;
   return false;
}
```

Best case:

Worst case:

Linear 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){
   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)

Average?

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);
}
```

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…

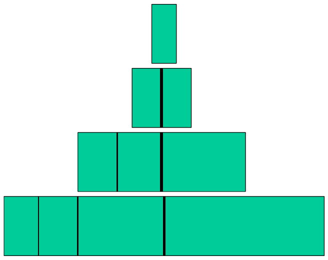
```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   return help(arr,k,0,arr.length);
}
boolean help(int[]arr, int k, int lo, int hi) {
   int mid = (hi+lo)/2;
   if(lo==hi)       return false;
   if(arr[mid]==k)  return true;
   if(arr[mid]< k)  return help(arr,k,mid+1,hi);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

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

For Binary Search

Number of Cuts



N = 1 K = 0

N = 2 K = 1

N = 4 K = 2

N = 8 K = 3

N = 16 K = ?

Binary Search

- In general, if N is a power of 2, ie. if $N = 2^K$ then $K = \log_2 N$ cuts are required.
- If N is not a power of 2, just round it up to the next one.

Recurrence Relation for Binary Search

$T(1)$ = about 13

$$\begin{aligned} T(N) &= 10 + T(N/2) \\ &= 10 + 10 + T(N/4) \\ &= 10 + 10 + 10 + T(N/8) \\ &\ldots\ldots \\ &= 10 * \text{number of cuts} + T(1) \\ &= 10 * \log_2 N + 13 \\ &= c1 * \log_2 N + c2 \\ &<= (c1+c2) * \log_2 N \\ &= O(\log_2 N) \end{aligned}$$

Prove by Induction that $T(N) = O(\log_2 N)$

- For $N = 1$, $T(1) = O(\log_2 1) = \log_2 2^0 = 0$ cuts.
- Assume it is true for $N = 2^K$ that there are k cuts and it is $O(k)$.
- When $N = 2^{k+1}$, we first cut it once in half, getting two halves each of which is size 2^k.

  According to our assumption (above), either of the halves would then get k cuts, so we'd end up with k + 1 cuts. Thus for size $N = 2^{k+1}$, we get k+1 cuts, so the search is $O(k+1)$.
- This is called the guess and prove method for solving recurrence relations. It says to figure it out intuitively first, and then prove it by induction or other methods.