# CSE332: Data Abstractions

# Lecture 2: Algorithm Analysis

Ruth Anderson

Winter 2015

# *Announcements*

- Project 1 – phase A due Monday
- Homework 1 – (out soon) due next Friday (normally due on Wed)
- Office Hours – posted
- Calendar & Midterm date - coming soon

# *Today – Algorithm Analysis*

- What do we care about?
- How to compare two algorithms
- Analyzing Code
- Recurrence relations
- Asymptotic Analysis

# *Algorithm Analysis*

- Correctness:
  - Does the algorithm do what is intended.

- Performance:
  - Speed — time complexity
  - Memory — space complexity

- Why analyze?
  - To make good design decisions
  - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

# *Correctness*

Correctness of an algorithm is established by proof. Common approaches:

- (Dis)proof by counterexample
- Proof by contradiction
- Proof by induction
  - Especially useful in recursive algorithms

# *Proof by Induction*

- **Base Case**: 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.

# *Mathematical induction*

Suppose *P(n)* is some predicate (involving integer *n*)

- Example: $n \geq n/2 + 1$ (for all n ≥ 2)

To prove *P(n)* for all integers $n \geq c$, it suffices to prove

1. *P(c)* – called the "basis" or "base case"
2. If *P(k)* then *P(k+1)* – called the "induction step" or "inductive case"

We will use induction:

To show an algorithm is correct or has a certain running time *no matter how big a data structure or input value is*

(Our "*n*" will be the data structure or input size.)

***P(n)* = "  the sum of the first *n* powers of 2 (starting at $2^0$) is $2^n$-1  "**

# *Inductive Proof Example*

Theorem: *P(n)* holds for all $n \geq 1$

Proof: By induction on *n*

- Base case, *n*=1: Sum of first power of 2 is $2^0$, which equals 1.
  And for n=1, $2^n$-1 equals 1.
- Inductive case:
  - Inductive hypothesis: Assume the sum of the first *k* powers of 2 is $2^k$-1
  - Show, given the hypothesis, that the sum of the first (*k*+1) powers of 2 is $2^{k+1}$-1

From our inductive hypothesis we know:

$$1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

Add the next power of 2 to both sides…

$$1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2^k - 1 + 2^k$$

We have what we want on the left; massage the right a bit

$$1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2(2^k) - 1 = 2^{k+1} - 1$$

# *Note for homework*

Proofs by induction will come up a fair amount on the homework

When doing them, be sure to state each part clearly:

- What you're trying to prove
- The base case
- The inductive case
- The inductive hypothesis
  - In many inductive proofs, you'll prove the inductive case by just starting with your inductive hypothesis, and playing with it a bit, as shown above

# *How should we compare two algorithms?*

# *Gauging performance*

- Uh, why not just run the program and time it
  - Too much *variability*, not reliable or *portable*:
    - Hardware: processor(s), memory, etc.
    - OS, Java version, libraries, drivers
    - Other programs running
    - Implementation dependent
  - Choice of input
    - Testing (inexhaustive) may *miss* worst-case input
    - Timing does not *explain* relative timing among inputs (what happens when *n* doubles in size)

- Often want to evaluate an *algorithm*, not an implementation
  - Even *before* creating the implementation ("coding it up")

# *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

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!

# *Today – Algorithm Analysis*

- What do we care about?
- How to compare two algorithms
- Analyzing Code
- Recurrence relations
- Asymptotic Analysis

# *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 of reality*: a very useful "lie".)

| | |
|---|---|
| Consecutive statements | Sum of time of each statement |
| Conditionals | Time of condition plus time of slower branch |
| Loops | Num iterations * time for loop body |
| Function Calls | Time of function's body |
| Recursion | Solve *recurrence equation* |

# *Complexity cases*

We’ll start by focusing on two cases:

- **Worst-case complexity**: max # steps algorithm takes on “most challenging” input of size N

- **Best-case complexity**: min # steps algorithm takes on “easiest” input of size N

# *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: 6 “ish” steps = $O(1)$

Worst case: 5 “ish” * (arr.length)

= $O$(arr.length)

# *Analyzing Recursive Code*

- Computing run-times gets interesting with recursion
- Say we want to perform some computation recursively on a list of size n
  - Conceptually, in each recursive call we:
    - Perform some amount of work, call it w(n)
    - Call the function recursively with a smaller portion of the list

- So, if we do w(n) work per step, and reduce the problem size in the next recursive call by 1, we do total work:

$$T(n)=w(n)+T(n-1)$$

- With some base case, like T(1)=5=O(1)

# *Example Recursive code: sum array*

Recursive:

- Recurrence is some constant amount of work O(1) done *n* times

```
int sum(int[] arr){
  return help(arr,0);
}
int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

Each time **help** is called, it does that O(1) amount of work, and then calls **help** again on a problem one less than previous problem size.

Recurrence Relation: $T(n) = O(1) + T(n\text{-}1)$

# *Solving Recurrence Relations*

- Say we have the following recurrence relation:

  T(n)=6 “ish”+T(n-1)

  T(1)=9 “ish” ←base case

- Now we just need to solve it; that is, reduce it to a closed form.
- Start by writing it out:

  T(n)=6+T(n-1)

  =6+6+T(n-2)

  =6+6+6+T(n-3)

  =6+6+6+…+6+T(1) = 6+6+6+…+6+9

  =6k+T(n-k)

  =6k+9, where k is the # of times we expanded T()

- We expanded it out n-1 times, so

  T(n)=6k+T(n-k)

  =6(n-1)+T(1) = 6(n-1)+9

  =6n+3 = O(n)

Or When does n-k=1?
Answer: when k=n-1

Best case:

Worst case:

# *Binary search*

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

Find an integer in a *sorted* array

- Can also be done non-recursively but “doesn’t matter” here

```
// 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: 9 "ish" steps = $O(1)$

Worst case: $T(n)$ = 10 "ish" + $T(n/2)$ 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?
   - $T(n) = 10 + T(n/2)$ $T(1) = 15$
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

# *Solving Recurrence Relations*

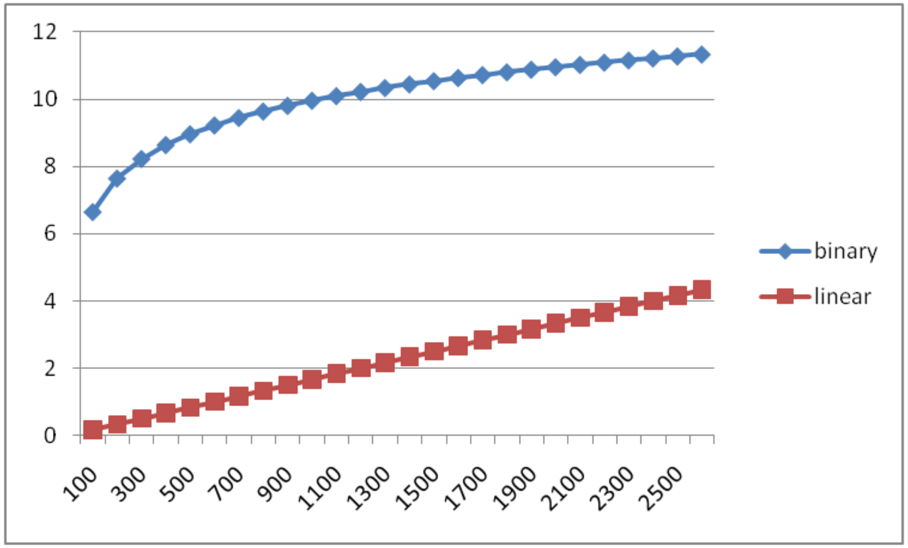
1. Determine the recurrence relation. What is the base case?
   - $T(n) = 10 + T(n/2)$ $T(1) = 15$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
   - $T(n) = 10 + 10 + T(n/4)$
     $= 10 + 10 + 10 + T(n/8)$
     $= \ldots$
     $= 10k + T(n/(2^k))$ (where k is 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
   - $n/(2^k) = 1$ means $n = 2^k$ means $k = \log_2 n$
   - So $T(n) = 10 \log_2 n + 15$ (get to base case and do it)
   - So $T(n)$ is $O(\log n)$

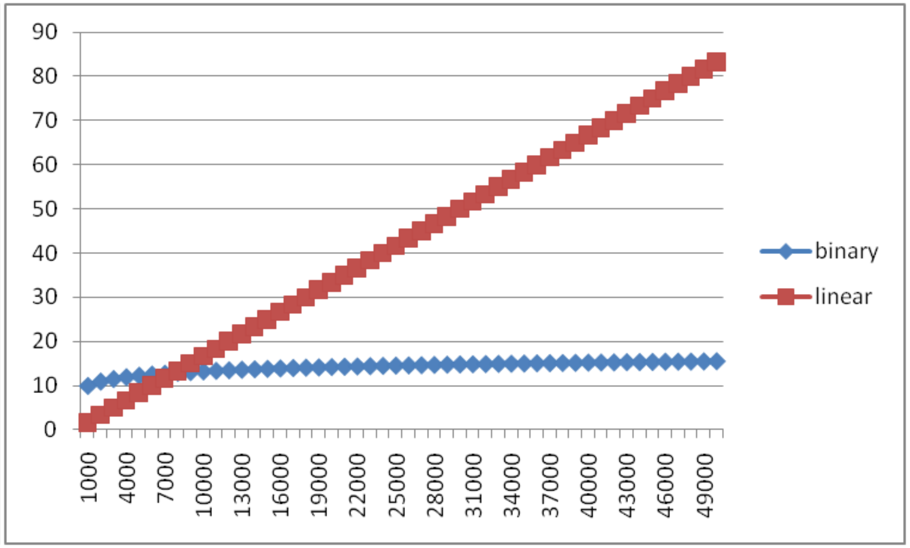
# *Ignoring constant factors*

- So binary search is $O(\mathtt{log}\ n)$ and linear is $O(n)$
  - But which will actually be faster?
  - Depending on constant factors and size of n, in a particular situation, linear search could be faster….
- Could depend on constant factors
  - How *many* assignments, additions, etc. for each *n*
  - And could depend on size of *n*
- **But** there exists some $n_0$ such that for all $n > n_0$ binary search wins
- Let's play with a couple plots to get some intuition…

# *Example*

- Let's try to "help" linear search
  - Run it on a computer 100x as fast (say 2010 model vs. 1990)
  - Use a new compiler/language that is 3x as fast
  - Be a clever programmer to eliminate half the work
  - So doing each iteration is 600x as fast as in binary search
- Note: 600x still helpful for problems without logarithmic algorithms!



# *Another example: sum array*

Two “obviously” linear algorithms: $T(n) = O(1) + T(n\text{-}1)$

Iterative:

```
int sum(int[] arr){
  int ans = 0;
  for(int i=0; i<arr.length; ++i)
     ans += arr[i];
  return ans;
}
```

Recursive:

- Recurrence is $c + c + \ldots + c$ for $n$ times

```
int sum(int[] arr){
  return help(arr,0);
}
int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

# *What about a <u>binary</u> version of sum?*

```
int sum(int[] arr){
   return help(arr,0,arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo==hi)   return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

Recurrence is $T(n) = O(1) + 2T(n/2)$

- 1 + 2 + 4 + 8 + …   for `log` *n* times
- $2^{(\log n)} - 1$ which is proportional to *n* (by definition of logarithm)

Easier explanation: it adds each number once while doing little else

"Obvious": You can't do better than *O(n)* – have to read whole array

# *Parallelism teaser*

- But suppose we could do two recursive calls *at the same time*
  - *Like having a friend do half the work for you!*

```
int sum(int[]arr){
   return help(arr,0,arr.length);
}
int help(int[]arr, int lo, int hi) {
   if(lo==hi)   return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

- If you have as many "friends of friends" as needed, the recurrence is now $T(n) = O(1) + 1T(n/2)$
  - $O($**`log`** $n)$ : same recurrence as for **`find`**

# *Really common recurrences*

Should know how to solve recurrences but also recognize some really common ones:

| | |
|---|---|
| $T(n) = O(1) + T(n\text{-}1)$ | linear |
| $T(n) = O(1) + 2T(n/2)$ | linear |
| $T(n) = O(1) + T(n/2)$ | logarithmic |
| $T(n) = O(1) + 2T(n\text{-}1)$ | exponential |
| $T(n) = O(n) + T(n\text{-}1)$ | quadratic |
| $T(n) = O(n) + T(n/2)$ | linear |
| $T(n) = O(n) + 2T(n/2)$ | $O(\mathrm{n}\ \mathbf{log}\ \mathrm{n})$ |

Note big-Oh can also use more than one variable

- Example: can sum all elements of an *n*-by-*m* matrix in $O(nm)$