CSE332: Data Abstractions

Lecture 3: Asymptotic Analysis

Dan Grossman

Spring 2012

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 because probably any algorithm is "plenty good" for small inputs (if *n* is 10, probably anything is fast)

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"

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 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: 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

- 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: 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

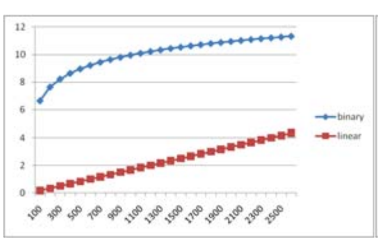
1. Determine the recurrence relation. What is the base case?
   - $T(n) = 10 + T(n/2)$ $\qquad T(1) = 8$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
   - $T(n) = 10 + 10 + T(n/4)$
     $= 10 + 10 + 10 + T(n/8)$
     $= \ldots$
  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

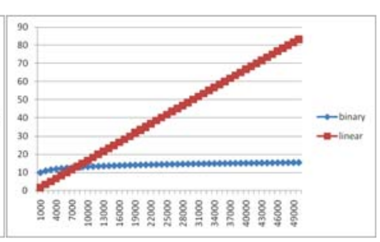
Ignoring constant factors

- So binary search is $O(\log n)$ and linear is $O(n)$
  - But which is 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: $T(n) = O(1) + T(n-1)$

Iterative:

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

Recursive:

- Recurrence is $k + k + \ldots + k$ 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 binary version?

```
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 $T(n) = O(1) + 2T(n/2)$

- $1 + 2 + 4 + 8 + \ldots$ for `log` n times
- $2^{(\log n)} - 1$ which is proportional to n (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 (see previous lecture)
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	$O(n$ `log` $n)$

Note big-Oh can also use more than one variable

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

Asymptotic notation

About to show formal definition, which amounts to saying:

1. Eliminate low-order terms
2. Eliminate coefficients

Examples:

- $4n + 5$
- $0.5n$ `log` $n + 2n + 7$
- $n^3 + 2^n + 3n$
- n `log` $(10n^2)$

Big-Oh relates functions

We use O on a function f(n) (for example n^2) to mean *the set of functions* with asymptotic behavior ***less than or equal to*** f(n)

So $(3n^2+17)$ **is in** $O(n^2)$

- $3n^2+17$ and n^2 have the same asymptotic behavior

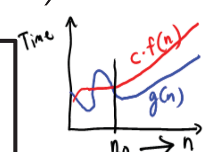
Confusingly, we also say/write:

- $(3n^2+17)$ **is** $O(n^2)$
- $(3n^2+17)$ **=** $O(n^2)$

But we would never say $O(n^2) = (3n^2+17)$

Formally Big-Oh (Dr? Ms? Mr? ☺)

Definition:
g(n) is in O(f(n)) if there exist constants
c and n_0 such that g(n) $\leq$ c f(n) for all $n \geq n_0$



- To show g(n) is in O(f(n)), pick a c large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"
  - Example: Let g(n) = $3n^2+17$ and f(n) = n^2
    c=5 and n_0=10 is more than good enough

- This is "less than or equal to"
  - So $3n^2+17$ is also $O(n^5)$ and $O(2^n)$ etc.

More examples, using formal definition

- Let $g(n) = 4n$ and $f(n) = n^2$
  - A valid proof is to find valid c and n_0
  - The "cross-over point" is n=4
  - So we can choose n_0=4 and c=1
    - Many other possible choices, e.g., larger n_0 and/or c

> Definition:
> g(n) is in O(f(n)) if there exist constants c and n_0 such that $g(n) \leq c\, f(n)$ for all $n \geq n_0$

More examples, using formal definition

- Let $g(n) = n^4$ and $f(n) = 2^n$
  - A valid proof is to find valid c and n_0
  - We can choose n_0=20 and c=1

> Definition:
> g(n) is in O(f(n)) if there exist constants c and n_0 such that $g(n) \leq c\, f(n)$ for all $n \geq n_0$

What's with the c

- The constant multiplier c is what allows functions that differ only in their largest coefficient to have the same asymptotic complexity

- Example: $g(n) = 7n+5$ and $f(n) = n$
  - For any choice of n_0, need a c > 7 (or more) to show g(n) is in O(f(n))

> Definition:
> g(n) is in O(f(n)) if there exist constants c and n_0 such that $g(n) \leq c\, f(n)$ for all $n \geq n_0$

What you can drop

- Eliminate coefficients because we don't have units anyway
  - $3n^2$ versus $5n^2$ doesn't mean anything when we have not specified the cost of constant-time operations (can re-scale)

- Eliminate low-order terms because they have vanishingly small impact as n grows

- Do NOT ignore constants that are not multipliers
  - n^3 is not $O(n^2)$
  - 3^n is not $O(2^n)$

(This all follows from the formal definition)

More Asymptotic Notation

- Upper bound: $O(f(n))$ is the set of all functions asymptotically less than or equal to f(n)
  - g(n) is in $O(f(n))$ if there exist constants c and n_0 such that $g(n) \leq c\, f(n)$ for all $n \geq n_0$
- Lower bound: $\Omega(f(n))$ is the set of all functions asymptotically greater than or equal to f(n)
  - g(n) is in $\Omega(f(n))$ if there exist constants c and n_0 such that $g(n) \geq c\, f(n)$ for all $n \geq n_0$
- Tight bound: $\theta(f(n))$ is the set of all functions asymptotically equal to f(n)
  - Intersection of $O(f(n))$ and $\Omega(f(n))$ (use *different* c values)

Correct terms, in theory

A common error is to say $O(f(n))$ when you mean $\theta(f(n))$
- Since a linear algorithm is also $O(n^5)$, it's tempting to say "this algorithm is exactly $O(n)$"
- That doesn't mean anything, say it is $\theta(n)$
- That means that it is not, for example $O(\log n)$

Less common notation:
- "little-oh": intersection of "big-Oh" and *not* "big-Theta"
  - *For all* c, there exists an n_0 such that… $\leq$
  - Example: array sum is $o(n^2)$ but not $o(n)$
- "little-omega": intersection of "big-Omega" and *not* "big-Theta"
  - *For all* c, there exists an n_0 such that… $\geq$
  - Example: array sum is $\omega(\log n)$ but not $\omega(n)$

What we are analyzing

- The most common thing to do is give an O or θ bound to the worst-case running time of an algorithm
- Example: binary-search algorithm
  - Common: $\theta(\texttt{log}\ n)$ running-time in the worst-case
  - Less common: $\theta(1)$ in the best-case (item is in the middle)
  - Less common: Algorithm is $\Omega(\texttt{log log}\ n)$ in the worst-case (it is not really, really, really fast asymptotically)
  - Less common (but very good to know): the find-in-sorted-array ***problem*** is $\Omega(\texttt{log}\ n)$ in the worst-case
    - *No* algorithm can do better (without parallelism)
    - A ***problem*** cannot be $O(f(n))$ since you can always find a slower algorithm, but can mean ***there exists*** an algorithm

Other things to analyze

- Space instead of time
  - Remember we can often use space to gain time
- Average case
  - Sometimes only if you assume something about the distribution of inputs
    - See CSE312 and STAT391
  - Sometimes uses randomization in the algorithm
    - Will see an example with sorting; also see CSE312
  - Sometimes an *amortized guarantee*
    - Will discuss in a later lecture

Summary

Analysis can be about:

- The problem or the algorithm (usually algorithm)
- Time or space (usually time)
  - Or power or dollars or …
- Best-, worst-, or average-case (usually worst)
- Upper-, lower-, or tight-bound (usually upper or tight)

Usually asymptotic is valuable

- Asymptotic complexity focuses on behavior for large n and is independent of any computer / coding trick
- But you can "abuse" it to be misled about trade-offs
- Example: $n^{1/10}$ vs. $\texttt{log}\ n$
  - Asymptotically $n^{1/10}$ grows more quickly
  - But the "cross-over" point is around $5 * 10^{17}$
  - So if you have input size less than 2^{58}, prefer $n^{1/10}$
- For *small* n, an algorithm with worse asymptotic complexity might be faster
  - Here the constant factors can matter, if you care about performance for small n

Timing vs. Big-Oh Summary

- Big-oh is an essential part of computer science's mathematical foundation
  - Examine the algorithm itself, not the implementation
  - Reason about (even prove) performance as a function of n
- Timing also has its place
  - Compare implementations
  - Focus on data sets you care about (versus worst case)
  - Determine what the constant factors "really are"
  - Will do some timing on the projects too