CSE373: Data Structures and Algorithms

# Asymptotic Analysis

Steve Tanimoto
Winter 2016

This lecture material represents the work of multiple instructors at the University of Washington. Thank you to all who have contributed!

## Efficiency

- What does it mean for an algorithm to be *efficient*?
  - We primarily care about *time* (and sometimes *space*)
- Is the following a good definition?
  - "An algorithm is efficient if, when implemented, it runs quickly on real input instances"
  - Where and how well is it implemented?
  - What constitutes "real input?"
  - How does the algorithm *scale* as input size changes?



## Gauging efficiency (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

*We will focus on large inputs* because probably any algorithm is "plenty good" for small inputs (if *n* is 10, probably anything is fast)
- Time difference really shows up as n grows

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!



## We usually care about worst-case running times

- Has proven reasonable in practice
  - Provides some guarantees
- Difficult to find a satisfactory alternative
  - What about average case?
  - Difficult to express full range of input
  - Could we use randomly-generated input?
  - May learn more about generator than algorithm



## 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?
k is in arr[0]
6ish steps
= $O(1)$

Worst case?
k is not in arr
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: about 8 steps = $O(1)$

Worst case: $T(n) \approx 10$ steps + $T(n/2)$ 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?
   – $T(n) = 10 + T(n/2)$ $\quad T(1) = 8$
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 argument to T in the right-hand side* to a value (e.g. 1) which reduces the problem to a base case
   – $n/(2^k) = 1$ means $n = 2^k$ means $k = \log_2 n$
   – So $T(n) = 10 \log_2 n + T(1)$
   – So $T(n) = 10 \log_2 n + 8$ (get to base case and do it)
   – So $T(n)$ is $O(\log n)$



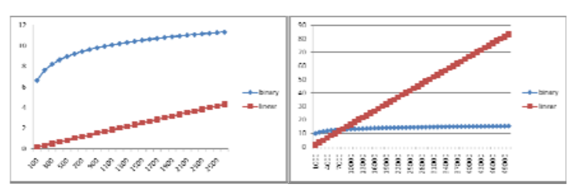
## Ignoring constant factors

- 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 $n$
    - *E.g.* $T(n) = 5{,}000{,}000n$ vs. $T(n) = 5n^2$
  – And could depend on overhead unrelated to $n$
    - *E.g.* $T(n) = 5{,}000{,}000 + \log n$ vs. $T(n) = 10 + n$
- 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 2014 model vs. 1994)
  – 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



## Big-O 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

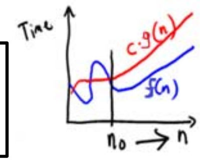
Confusingly, we also say/write:

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

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



## Big-O, formally




Definition: f($n$) is in O( g($n$) ) if there exist positive constants $c$ and $n_0$ such that

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

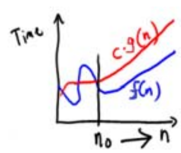
This is equivalent to:

We say that f($n$) is in O( g($n$) )
if and only if
it's possible to demonstrate that f(n) beyond some value of n becomes bounded by a positive multiple of g(n).

That value of n is $n_0$. The positive multiple of g(n) is c · g(n)



## Showing Big-O




- To show f($n$) is in O( g($n$) ), pick a $c$ large enough to "cover the constant factors" and $n_0$ large enough to "cover the lower-order terms"
  – Example: Let f($n$) = $3n^2+17$ and g($n$) = $n^2$
    $c$=5 and $n_0$=10 is more than good enough
    ($3*10^2$)+17 ≤ $5*10^2$ so $3n^2+17$ is O($n^2$)
- This is "less than or equal to"
  – So $3n^2+17$ is also $O(n^5)$ and $O(2^n)$ etc.
    - But usually we're interested in the tightest upper bound.



## Example 1, using formal definition

- Let f($n$) = 1000$n$ and g($n$) = $n^2$
  – To prove f($n$) is in O(g($n$)), find a valid $c$ and $n_0$
  – The "cross-over point" is $n$=1000
    - f($1000$) = 1000*1000 and g($1000$) = $1000^2$
  – So we can choose $n_0$=1000 and $c$=1
    - Many other possible choices, e.g., larger $n_0$ and/or $c$

Definition: f($n$) is in O( g($n$) ) if there exist positive constants $c$ and $n_0$ such that

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



## Example 2, using formal definition

- Let f($n$) = $n^4$ and f($n$) = $2^n$
  – To prove f($n$) is in O(g($n$)), find a valid $c$ and $n_0$
  – We can choose $n_0$=20 and $c$=1
    - f($20$) = $20^4$ vs. g($20$) = $1*2^{20}$
- Note: There are many correct possible choices of $c$ and $n_0$

Definition: f($n$) is in O( g($n$) ) if there exist positive constants $c$ and $n_0$ such that

f($n$) ≤ $c$ g($n$) 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
- Consider:
  f($n$) = 7$n$+5
  g($n$) = $n$

– These have the same asymptotic behavior (linear)
  - So f($n$) is in O(g($n$)) even through f($n$) is always larger
  - The $c$ allows us to provide a coefficient so that f($n$) ≤ $c$ g($n$)

– In this example:
  - To prove f($n$) is in O(g($n$)), have $c$ = 12, $n_0$ = 1
    (7*1)+5 ≤ 12*1

## 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
- 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)

CSE 373 Winter 2016 19

## More Asymptotic Notation

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

CSE 373 Winter 2016 20

## Correct terms, in theory

A common error is to say $O(\ g(n)\ )$ when you mean $\Theta(\ g(n)\ )$
- 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-O" 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 $\omega(n^2)$

CSE 373 Winter 2016 21

## What we are analyzing

- The most common thing to do is give an $O$ upper bound to the worst-case running time of an algorithm
- Example: binary-search algorithm
  - Common: $O(\log n)$ running-time in the worst-case
  - Less common: $\Theta(1)$ in the best-case (item is in the middle)
  - Less common (but very good to know): the find-in-sorted-array ***problem*** is $\Omega(\log n)$ in the worst-case
    - *No* algorithm can do better
    - A ***problem*** cannot be $O(g(n))$ since you can always make a slower algorithm

CSE 373 Winter 2016 22

## 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 *probability distribution* of inputs
  - Sometimes uses randomization in the algorithm
    - Will see an example with sorting
  - Sometimes an *amortized guarantee*
    - Average time over any sequence of operations
    - Will discuss in a later lecture

CSE 373 Winter 2016 23

## 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)

CSE 373 Winter 2016 24

## Big-O Caveats

- 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. `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
  - If you care about performance for small *n* then the constant factors can matter



## Addendum: Timing vs. Big-O Summary

- Big-O 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"