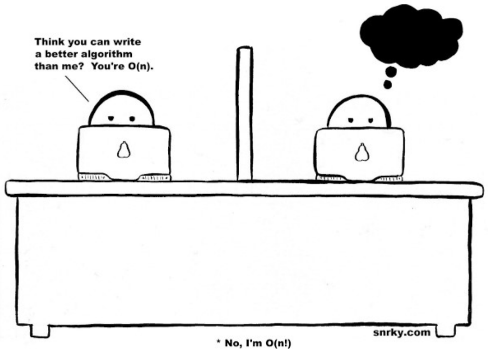
CSE373: Data Structures and Algorithms

Asymptotic Analysis (Big O, Ω, and Θ)

Steve Tanimoto
Autumn 2016

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

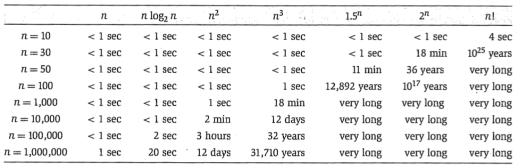


Big-O: Common Names

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	"$n \log n$"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant)
$O(k^n)$	exponential (where k is any constant > 1)
$O(n!)$	factorial

Comparing Function Growth (e.g., for Running Times)

- For a processor capable of one million instructions per second



	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Efficiency

- What does it mean for an algorithm to be *efficient*?
  – We 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"

Gauging Performance

- Uh, why not just run the program and time it?
  – Too much *variability*, not reliable or *portable*:
    - Hardware: processor(s), memory, etc.
    - Software: 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 5, 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* but practical.)

Control Flow	Time required
Consecutive statements	Sum of times of statements
Conditionals	Sum of times of test and slower branch
Loops	Number of iterations × time of body
Calls	Time of called function's body
Recursion	(Solve a *recurrence equation*)

Analyzing Code

1. Add up time for all parts of the algorithm
   e.g. number of iterations = $(n^2 + n)/2$
2. Eliminate low-order terms i.e. eliminate n: $(n^2)/2$
3. Eliminate coefficients i.e. eliminate 1/2: (n^2)

Examples:
- $4n + 5$ $\in O(n)$
- $0.5n \log n + 2n + 7$ $\in O(n \log n)$
- $n^3 + 2^n + 3n$ $\in O(2^n)$ EXPONENTIAL GROWTH!
- $n \log (10n^2) + 2n \log (n)$ $\in O(n \log n)$

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: about 6 steps = $O(1)$
Worst case: $\sim 6*(\text{arr.length})$
$\in O(\text{arr.length})$

Binary Search

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array
- Can also be done non-recursively

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

- $T(n) \in O(\log n)$ 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?
   - $T(n) = 10 + T(n/2)$ $T(1) = 10$
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
   - $n/(2^k) = 1$ implies $n = 2^k$ implies $k = \log_2 n$
   - So $T(n) = 10 \log_2 n + 8$ (get to base case and do it)
   - So $T(n)$ is in $O(\log n)$

Ignoring Constant Factors

- So binary search's runtime is in $O(\log n)$ and linear's is in $O(n)$
  - 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 size of 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

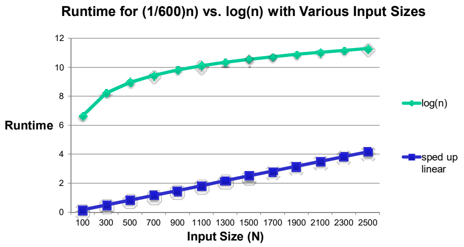
Run it on a computer 100x as fast (say 2015 model vs. 1990)

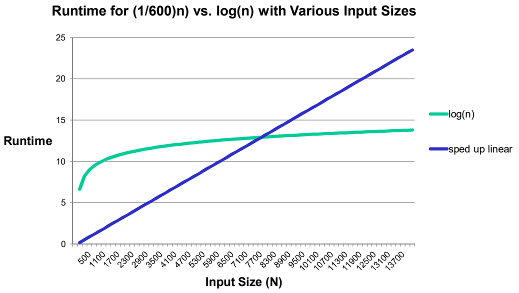
Use a new compiler/language that is 3x as fast

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: $T(n) = c + T(n-1)$

Iterative:

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

Recursive:

- Recurrence is $k + k + \dots + k$ 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 Recursive 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 $T(n) = 1 + 2T(n/2)$; $T(1) = 1$

$1 + 2 + 4 + 8 + \ldots$ 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": We can't do better than *O(n)* : we 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 $T(n) = c + 1T(n/2)$
  - *O*(`log` *n*) : same recurrence as for `find`

Common Recurrences

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

$T(n) = c + T(n-1)$	linear
$T(n) = c + 2T(n/2)$	linear
$T(n) = c + T(n/2)$	logarithmic: *O*(`log` n)
$T(n) = c + 2T(n-1)$	exponential
$T(n) = c\,n + T(n-1)$	quadratic
$T(n) = c\,n + T(n/2)$	linear
$T(n) = c\,n + 2T(n/2)$	*O*(n `log` n)

Note big-O 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-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*)

For example, $(3n^2+17)$ **is in** $O(n^2)$

Confusingly, some people also say/write:

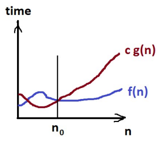
- $(3n^2+17)$ **is** $O(n^2)$
- $(3n^2+17) = O(n^2)$

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

Big-O, Formally

Definition:

f(*n*) is in O(g(*n*)) if there exist constants *c* and n_0 such that $f(n) \le c\,g(n)$ for all $n \ge n_0$



- 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 $n_0=10$ will work.
- This is "less than or equal to"
  - So $3n^2+17$ is also in $O(n^5)$ and in $O(2^n)$ etc.

More Examples, Using Formal Definition

- Let $f(n) = 1000n$ and $g(n) = n^2$
  - A valid proof is to find valid c and n_0
  - The "cross-over point" is $n=1000$
  - 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 constants
c and n_0 such that $f(n) \leq c\,g(n)$ for all $n \geq n_0$

More Examples, Using Formal Definition

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

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

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

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

Aesthetics of a Big-O Demonstrations

- Sometimes, $f(n)$ is clearly "dominated" by $g(n)$.
- That happens when f is in $O(g)$, but g is not in $O(f)$.
- For example $2n$ is in $O(n^3)$ but n^3 is not in $O(2n)$.
- Then to show $f(n)$ is in $O(g(n))$, it is *good form* to use $c = 1$ and the smallest n_0 that works with it, or the smallest integer value of n_0 that works with it.
- We show $2n$ is in $O(n^3)$ by taking $c=1$ and $n_0 = 2^{1/2}$ or $n_0 = 2$.

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

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 in $O(n^2)$
  - 3^n is not in $O(2^n)$

(This all follows from the formal definition)

Big-O: Common Names (Again)

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	"$n \log n$"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant)
$O(k^n)$	exponential (where k is any constant > 1)

"exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1"
- A savings account accrues interest exponentially (k=1.01?)
- If you don't know k, you probably don't know its exponential

More Asymptotic Notation

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

Correct Terms, in Theory

A common error is to say **$O(\text{f}(n))$** when you mean **$\Theta(\text{f}(n))$**
- 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 $O(\texttt{log}\ n)$

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 $o(n)$
  - "strictly greater than"
- "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 $\omega(\texttt{log}\ n)$ but not $\omega(n)$
  - "strictly less than"

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: $(\texttt{log}\ n)$ 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 $\Omega(\texttt{log}\ n)$ in the worst-case
    - *No* algorithm can do better
    - A ***problem*** cannot be $O(\text{f}(n))$ 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 *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

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 Analysis 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-O Summary

- Big-O notation 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".