CSE373: Data Structures and Algorithms
Lecture 4: Asymptotic Analysis

Aaron Bauer
Winter 2014
Previously, on CSE 373

• We want to analyze algorithms for efficiency (in time and space)
• And do so generally and rigorously
  – not timing an implementation
• We will primarily consider worst-case running time
• Example: find an integer in a sorted array
  – Linear search: $O(n)$
  – Binary search: $O(\log n)$
  – Had to solve a recurrence relation to see this
Another example: sum array

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

Iterative:

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

Recursive:

```java
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);
}
```

Recursive:

- Recurrence is \( k + k + \ldots + k \) for \( n \) times
What about a binary version?

```java
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!

```java
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 $T(n) = O(1) + 1 T(n/2)$
  – $O(\log n)$: same recurrence as for `find`
Really common recurrences

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

\[ T(n) = O(1) + T(n-1) \] \quad \text{linear}
\[ T(n) = O(1) + 2T(n/2) \] \quad \text{linear}
\[ T(n) = O(1) + T(n/2) \] \quad \text{logarithmic}
\[ T(n) = O(1) + 2T(n-1) \] \quad \text{exponential}
\[ T(n) = O(n) + T(n-1) \] \quad \text{quadratic (see previous lecture)}
\[ T(n) = O(n) + 2T(n/2) \] \quad 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 $O(n^2) = (3n^2+17)$
Big-O, formally

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

• 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 \( f(n) = n^2 \)
    \[ c=5 \text{ and } n_0 =10 \text{ is more than good enough} \]

• 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 \( g(n) = 1000n \) and \( f(n) = n^2 \)
  – 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:
\[
g(n) \text{ is in } O(f(n)) \text{ if there exist constants } c \text{ and } n_0 \text{ such that } g(n) \leq c f(n) \text{ for all } n \geq n_0
\]
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:

\[ g(n) \text{ is in } O(f(n)) \text{ if there exist constants } c \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0 \]
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:

\[ g(n) \text{ is in } O(f(n)) \text{ if there exist constants } c \text{ and } n_0 \text{ such that } g(n) \leq c f(n) \text{ for all } n \geq n_0 \]
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)
Big-O: Common Names (Again)

O(1)     constant (same as O(k) for constant k)
O(log n) logarithmic (probing)
O(n)     linear (single-pass)
O(n log n) “n log n” (mergesort)
O(n^2)   quadratic (nested loops)
O(n^3)   cubic (more nested loops)
O(n^k)   polynomial (where is k is any constant)
O(k^n)   exponential (where k is any constant > 1)
**Big-O running times**

- For a processor capable of one million instructions per second

<table>
<thead>
<tr>
<th></th>
<th>( n )</th>
<th>( n \log_2 n )</th>
<th>( n^2 )</th>
<th>( n^3 )</th>
<th>( 1.5^n )</th>
<th>( 2^n )</th>
<th>( n! )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 10 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>( n = 30 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>10^{25} years</td>
</tr>
<tr>
<td>( n = 50 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 100 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>10^{17} years</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 1,000 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 10,000 )</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 100,000 )</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>( n = 1,000,000 )</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
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… ≤
  • 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… ≥
  • 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 $\theta$ bound to the worst-case running time of an algorithm

• Example: binary-search algorithm
  – Common: $\theta(\log n)$ running-time in the worst-case
  – Less common: $\theta(1)$ in the best-case (item is in the middle)
  – Less common: Algorithm is $\Omega(\log \log n)$ 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 $\Omega(\log n)$ in the worst-case
    • No algorithm can do better
    • 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 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 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. $\log n$
  – Asymptotically $n^{1/10}$ grows more quickly
  – But the “cross-over” point is around $5 \times 10^{17}$
  – 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”