CSE373: Data Structures and Algorithms

Lecture 4: Asymptotic Analysis

Catie Baker

Spring 2015
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

Find an integer in a sorted array

```java
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    ???
}
```
Linear search

Find an integer in a sorted array

```java
// 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]
- \(c_1\) steps
- \(= O(1)\)

Worst case?
- k is not in arr
- \(c_2 \times (\text{arr.length})\)
- \(= O(\text{arr.length})\)
Binary search

Find an integer in a sorted array

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

```java
// 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: $c_1$ steps = $O(1)$
Worst case: $T(n) = c_2$ steps + $T(n/2)$ where $n$ is $hi-lo$
  
  • $O(\log n)$ where $n$ is $\text{array.length}$
  
  • Solve recurrence equation to know that...

```java
// requires array is sorted
// returns whether $k$ is in array
boolean find(int[] arr, int k) {
    return help(arr, k, 0, arr.length); // 1
}

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?
   - \( T(n) = c2 + T(n/2) \) \( T(1) = c1 \)

2. “Expand” the original relation to find an equivalent general expression *in terms of the number of expansions*.
   - \( T(n) = c2 + c2 + T(n/4) \)
     - \( = c2 + c2 + c2 + T(n/8) \)
     - \( = ... \)
     - \( = c2(k) + T(n/(2^k)) \)

3. Find a closed-form expression by setting *the number of expansions* 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 \( T(n) = c2 \log_2 n + T(1) \)
   - So \( T(n) = c2 \log_2 n + c1 \) (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-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: \( g(n) \) is in \( O(f(n)) \) if there exist positive constants \( c \) and \( n_0 \) such that

\[ g(n) \leq c f(n) \quad \text{for all } n \geq n_0 \]

- 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 \( f(n) = n^2 \)
    - \( c=5 \) and \( n_0 =10 \) is more than good enough
    - \( (3\cdot10^2)+17 \leq 5\cdot10^2 \) 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 \( g(n) = 1000n \) and \( f(n) = n^2 \)
  – To prove \( g(n) \) is in \( O(f(n)) \), find a valid \( c \) and \( n_0 \)
  – The “cross-over point” is \( n=1000 \)
    • \( g(n) = 1000 \times 1000 \) and \( f(n) = 1000^2 \)
    – So we can choose \( n_0=1000 \) and \( c=1 \)
      • Many other possible choices, e.g., larger \( n_0 \) and/or \( c \)

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

\[
g(n) \leq c f(n) \quad \text{for all } n \geq n_0
\]
Example 2, using formal definition

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

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

\[
g(n) \leq c f(n) \quad \text{for all } n \geq n_0
\]
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.
- Consider:
  
  $g(n) = 7n+5$
  $f(n) = n$
  
  - These have the same asymptotic behavior (linear)
    - So $g(n)$ is in $O(f(n))$ even through $g(n)$ is always larger.
    - The $c$ allows us to provide a coefficient so that $g(n) \leq c f(n)$
  
  - In this example:
    - To prove $g(n)$ is in $O(f(n))$, have $c = 12$, $n_0 = 1$
      $(7*1)+5 \leq 12*1$
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)
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 $f(n)$
  - $g(n)$ is in $\Theta(f(n))$ if both $g(n)$ is in $O(f(n))$ and $g(n)$ is in $\Omega(f(n))$
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… $\le$
  - 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… $\ge$
  - Example: array sum is $\omega(\log n)$ but not $\omega(n)$
What we are analyzing

• The most common thing to do is give an $O$ 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 $O(f(n))$ since you can always make a slower 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)
Big-Oh 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 $5 \times 10^{17}$
  – 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
  – If you care about performance for small $n$ then the constant factors can matter
Addendum: 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”
private static void bubbleSort(int[] intArray) {
    int n = intArray.length;
    int temp = 0;

    for(int i=0; i < n; i++){
        for(int j=1; j < (n - i); j++){
            if(intArray[j-1] > intArray[j]){ //swap the elements!
                temp = intArray[j-1];
                intArray[j-1] = intArray[j];
                intArray[j] = temp;
            }
        }
    }
}

i     j
0     n-1
1     n-2
2     n-3
3     n-4
...   ...
n-2   1
n-1   0

1+2+3+..+(n-2)+(n-1) = n(n-1)/2 (number of iterations)

Each iteration takes c1

O(n^2)