CSE373: Data Structures and Algorithms

Lecture 4: Asymptotic Analysis

Lauren Milne

Summer 2015

Administrivia

- Questions on Homework 1? Due Wednesday at 10:59 pm.
- TA Session tomorrow, mostly on induction
- Today
  - Algorithmic Analysis!

Algorithm Analysis

- As the size of an algorithm's input grows, we want to know
  - How long it takes to run (time)
  - How much room it takes to run (space)
- We use Big-O notation to compare algorithm runtimes
  - Ignore constants and lower order terms
  - Independent of implementation
  - Big-O of (n^3 + $10n\log^2 n$ + 5)?
- Make assumptions
  - "basic" operations take constant time
- Always analyze worst possible case
  - Slower branch of conditional
  - Worst possible input

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]

c1 steps

= $O(1)$

Worst case?

k is not in arr

c2*(arr.length)

= O(arr.length)

Binary search

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array

```
// requires sorted array
// 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:

c1 steps = $O(1)$

Worst case:

$T(n) = c2 + T(n/2)$ where c2 is constant and n is **`hi-lo`**

$O($**`log`** $n)$ where n is **`arr.length`** (*recurrence equation*)

```
// requires sorted array
// 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 and the base case.
   - $T(n) = c2 + T(n/2)$ $\qquad\qquad$ $T(1) = c1$

What is T(n/2)?

What is T(n/4)?

Solving Recurrence Relations

1. Determine the recurrence relation and the base case.
   – $T(n) = c2 + T(n/2)$ $\qquad T(1) = c1$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions "k"*.
   – $T(n) = c2 + c2 + T(n/4)$
   $= c2 + c2 + c2 + T(n/8)$
   $= \ldots$
   $= c2(k) + T(n/(2^k))$
3. Find a closed-form expression: find *the number of expansions* to reach the 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 $T(n) = c2 \log_2 n + c1$
   – So $T(n)$ is $O(\log n)$

Ignoring constant factors

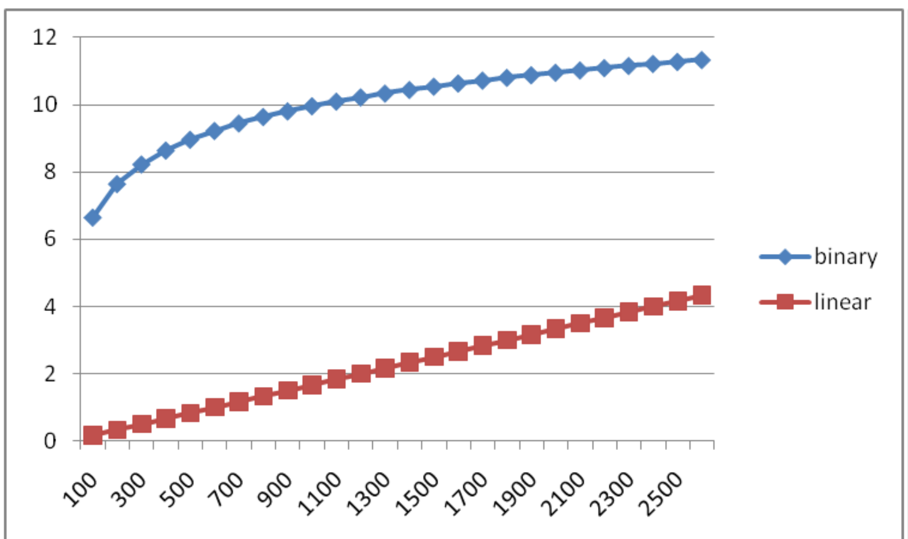
- So binary search is $O(\texttt{log}\ n)$ and linear is $O(n)$
  - But which is faster?

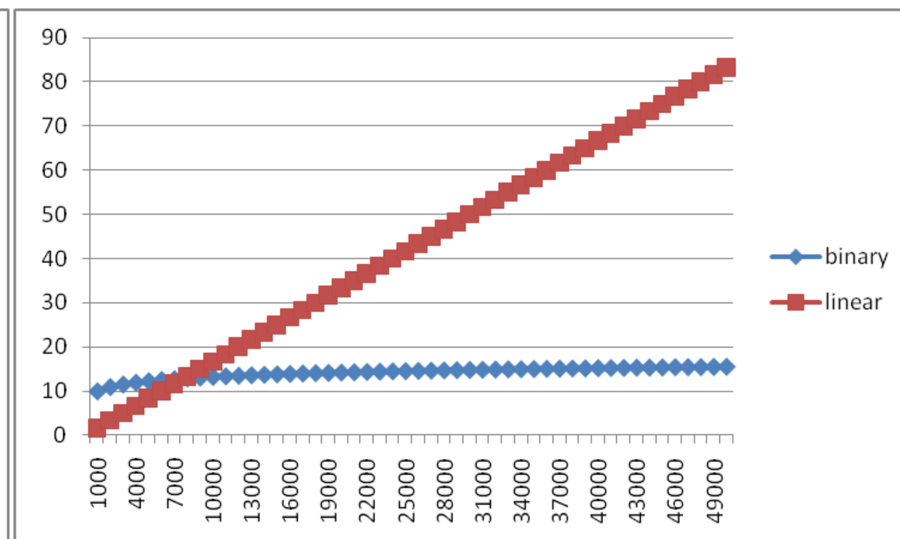
- Depends on constant factors
  - How *many* assignments, additions, etc. for each n
    - *E.g.* $T(n) = 5{,}000{,}000n$ vs. $T(n) = 5n^2$
  - And could depend on overhead unrelated to n
    - *E.g.* $T(n) = 5{,}000{,}000 + \log n$ vs. $T(n) = 10 + n$

- But there exists some n_0 such that for all $n > n_0$ binary search wins

Example

- Let’s try to “help” linear search
  - 100x faster computer
  - 3x faster compiler/language
  - 2x smarter programmer (eliminate half the work)
  - Each iteration is 600x as fast as in binary search





Big-O, formally

Definition:

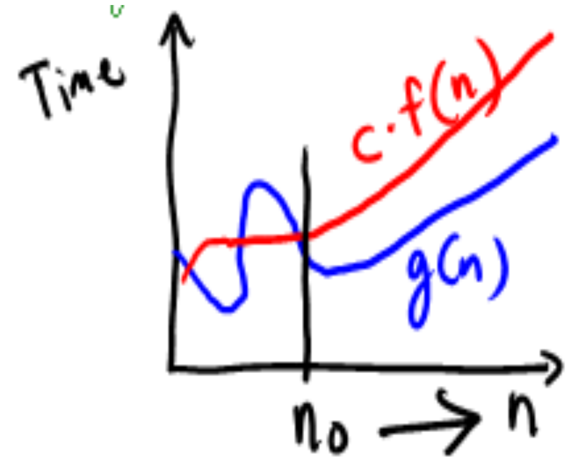
$g(n)$ is in $O(\ f(n)\ )$ if there exists positive constants c and n_0 such that

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

Big-O, formally

Definition:

> $g(n)$ is in $O(\ f(n)\ )$ if there exists positive 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"
  - n_0 large enough to "cover the lower-order terms"
- Example:
  - Let $g(n) = 3n^2+17$ and $f(n) = n^2$

    What could we pick for c and n_0?

    $c = 5$ and $n_0 = 10$

    $(3*10^2)+17 \leq 5*10^2$ so $3n^2+17$ is $O(n^2)$

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*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
>
> $g(n) \leq c\ f(n)$ 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. $f(n) = 1*2^{20}$

> Definition: $g(n)$ is in $O(\ f(n)\ )$ if there exist positive constants c and n_0 such that
>
> $g(n) \leq c\ f(n)$ 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 $O(2^n)$

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 $\theta(\ f(n)\ )$

- A linear algorithm is in both O(n) and $O(n^5)$
- Better to 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 $\omega(n)$

Summary

Analysis can be about:

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

- We generally will give an *O* upper bound to the worst-case running time of an algorithm

Big-O Caveats

- Asymptotic complexity focuses on behavior for large *n*
- You can be misled about trade-offs using it
- Example: $n^{1/10}$ vs. **`log`** *n*
  - Asymptotically $n^{1/10}$ grows more quickly
  - "Cross-over" point is around $5 * 10^{17}$
  - So for any smaller input, prefer $n^{1/10}$
- For *small n*, an algorithm with worse asymptotic complexity might be faster

Addendum: Timing vs. Big-O Summary

- Big-O
  - Examine the algorithm itself, not the implementation
  - Reason about performance as a function of *n*

- Timing
  - Compare implementations
  - Focus on data sets other than worst case
  - Determine what the constants actually are

Bubble Sort

```
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
…	…
n-2	1
n-1	0

Number of iterations
0+1+2+3+..+(n-2)+(n-1)
= n(n-1)/2

Each iteration takes c1

$O(n^2)$