CSE373: Data Structures and Algorithms

Algorithm 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!

Algorithm Analysis

As the "size" of an algorithm's input grows (integer, length of array, size of queue, etc.), we want to know
- How much longer does the algorithm take to run? (time)
- How much more memory does the algorithm need? (space)

Because the curves we saw are so different, often care about only "which curve we are like"

Separate issue: Algorithm *correctness* – does it produce the right answer for all inputs
- Usually more important, naturally

Algorithm Analysis: A first example

- Consider the following program segment:

```
x = 0
for i = 1 to n do
   for j = 1 to i do                (pseudocode)
       x = x + 1
```

- What is the value of x at the end?

i	**j**	**x**
1	1 to 1	1
2	1 to 2	3
3	1 to 3	6
4	1 to 4	10
…		
n	1 to n	?

Number of times x gets incremented is
= 1 + 2 + 3 + … + (n-1) + n
= n·(n+1)/2

Analyzing the loop

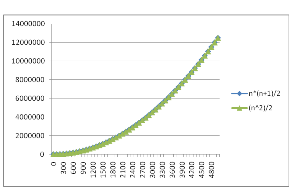
- Consider the following program segment:

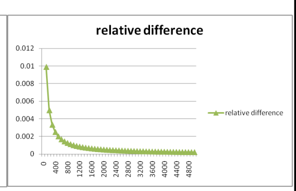
```
x = 0
for i = 1 to n do
   for j = 1 to i do
       x = x + 1
```

- The total number of loop iterations is n·(n+1)/2
  - This is a very common loop structure, worth memorizing
  - This is *proportional to* n^2 , and we say $O(n^2)$, "big-Oh of"
    - n·(n+1)/2 = $(n^2+ n)/2$
    - For large enough n, the lower order and constant terms are irrelevant, as are the assignment statements
    - See plot… $(n^2+ n)/2$ vs. just $n^2/2$

Lower-order terms don't matter

n·(n+ 1)/2 vs. just $n^2/2$





We just say $O(n^2)$

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

Note: "exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1"

Big-O 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

Analyzing code

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*: a very useful "lie".)

Consecutive statements	Sum of times
Conditionals	Time of test plus slower branch
Loops	Sum of iterations
Calls	Time of call's body
Recursion	Solve *recurrence equation* *(next lecture)*

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:
- $4n + 5$ = $O(n)$
- $0.5n \log n + 2n + 7$ = $O(n \log n)$
- $n^3 + 2^n + 3n$ = $O(2^n)$
- $n \log (10n^2)$
  = $2n \log (10n)$ = $O(n \log n)$