CS173
Running Time and Big-O

Tandy Warnow

CS 173
Running Times and Big-O analysis
Tandy Warnow

Today's material

We will cover:

- Running time analysis
- Review of running time analysis of Bubblesort
- Review of running time analysis of Mergesort
- Review of running time analysis of recursive Fibonacci
- "Big-O"

Running time analyses

Often algorithms are described recursively or using divide-and-conquer. We will show how to analyze the running times by:

- writing the running time as a recurrence relation
- solving the running time
- proving the running time correct using induction

Analyzing running times

To analyze the running time of an algorithm, we have to know what we are counting, and what we mean.

First of all, we usually want to analyze the **worst case** running time.

This means an upper bound on the total running time for the operation, usually expressed as a function of its input **size**.

(For example, the size of a graph could be $n + m$, where n is the number of vertices and m is the number of edges.)

Running time analysis

We are going to count **operations**, with each operation having the same cost:

- I/O (reads and writes)
- Numeric operations (additions and multiplications)
- Comparisons between two values
- Logical operations

Running times of algorithms

Now we begin with algorithms, and analyzing their running times.

Let A be an algorithm which has inputs of size n, for $n \geq n_0$.

Let $t(n)$ describe the worst case largest running time of A on input size n.

Running times of recursively defined algorithms

Algorithms that are described recursively typically have the following structure:

- Solve the problem if $n = n_0$; else
  - Preprocessing
  - Recursion to one or more smaller problems
  - Postprocessing

As a result, their running times can be described by recurrence relations of the form

$$t(n_0) = C \text{ (some positive constant)}$$
$$t(n) = f(n) + \sum_{i \in I} t(i) + g(n) \text{ if } n > n_0$$

For the second bullet,

- $f(n)$ is the preprocessing time
- I is a set of dataset sizes that we run the algorithm on recursively
- $g(n)$ is the time for postprocessing

Example: Bubblesort

Bubblesort is an algorithm that sorts an array of integers by moving from left-to-right, swapping pairs of elements that are out of order.

From https://en.wikipedia.org/wiki/Bubble_sort:

```
procedure bubbleSort( A : list of sortable items )
n = length(A)
repeat
    swapped = false
    for i = 1 to n-1 inclusive do
        if A[i-1] > A[i] then
            swap(A[i-1], A[i])
            swapped = true
        end if
    end for
    n = n - 1
until not swapped
end procedure
```

Recursive bubblesort

Step 0: If $n = 1$, then Return A

Step 1: Scan array from left-to-right, swapping adjacent entries that are out of order

Step 2: Recurse on $A[0...n-2]$

Running time analysis

$t(n)$ is the running time for recursive BubbleSort on inputs of size n.

1. If $n = 1$, the running time is C_1 for some constant C_1.
2. The "preprocessing" takes place in Step 0 (checking to see if $n = 1$) and Step 1 (the left-to-right scan, swapping adjacent elements that are out of order), and uses no more than $C_2 n$ operations.
3. There is only one subproblem and it has $n - 1$ elements; hence the recursion takes $t(n - 1)$ operations.
4. There is no postprocessing stage for this algorithm.

Hence

- $t(1) = C_1$
- for $n > 1$, then $t(n) \leq C_2 n + t(n - 1)$

Bubblesort running time

We have

- $t(1) = C_1$
- $t(n) = C_2 n + t(n-1)$

Let $C = \max\{C_1, C_2\}$.

In an earlier class, we proved (using induction) that that $\forall n \in \mathbb{Z}^+, t(n) \leq Cn^2$.

Mergesort running time

Recall the Mergesort algorithm: divide into two sets, sort the two sets, then merge.
The running time analysis had

- $t(1) = C_1$
- $t(n) = 2t(\frac{n}{2}) + Cn$ for $n > 1$

Hence, there is some constant C' such that $t(n) \leq C'n \log n$ for all $n \in \mathbb{Z}$.

Recursive Fibonacci number calculation

Recall that recursive calculation of the n^{th} Fibonacci number:

- We store the two base cases, $F(1) = F(2) = 1$, in an array $Fib[1...n]$ (i.e., $Fib[1] = 1, Fib[2] = 1$).
- For $i = 3$ up to n, we compute $Fib[i]$ using the rule:
  - $Fib[i] = Fib[i-1] + Fib[i-2]$
- Return Fib[n]

The running time here is easy to analyze: there are n entries, and each one uses at most C operations for some constant C.
Hence the total time is at most Cn time.

Big-O

In all of this we have been writing upper bounds for running times, using constants.

But we didn't care about the constants, not really.

Big-O is a way of communicating this.

Big-O, see Rosen page 205

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be two functions.

We will say that f is "Big-O" of g if $\exists C > 0$ and $k \geq 0$ such that $|f(n)| \leq C|g(n)|$ for all $n > k$.

We write this as $f(n)$ is $O(g(n))$.

Note that this is just expressing that $g(n)$ is an *upper bound* for $f(n)$.

The bound need not be tight.

But also, we don't require that $f(n) \leq Cg(n)$ for all n... only for sufficiently large n.

Quick tricks about Big-O

- When comparing two polynomials, the degree is the only thing that matters.
  So n^5 grows faster than n^3.
  Therefore, n^3 is $O(n^5)$.
- When comparing two functions $f(n)$ and $g(n)$ where one or both are written as the sum of simpler functions, just compare the fastest growing terms.
  So $500n^2 + n^5$ is dominated by n^5, and $g(n) = 200n + \frac{n!}{200}$ is dominated by $\frac{n!}{200}$.
  Therefore, $500n^2 + n^5$ is $O(200n + \frac{n!}{200})$.

Big-O

True or False?

1. n^2 is $O(n^3)$?
2. n^3 is $O(n^3 - 200)$?
3. n^3 is $O(n^2)$?
4. n^2 is $O(n^3 - 200n)$?
5. $5n^2 + 1000$ is $O(n^3)$?
6. n^n is $O(n!)$?
7. $n!$ is $O(n^n)$?
8. 2^{logn} is $O(n)$?

Proving f is $O(g)$

Let $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ be two functions.

How can we prove that f is $O(g)$?

- Find positive constants C and k such that $|f(n)| \leq C|g(n)|$ for all $n \geq k$.
- Prove that $\exists C > 0$ and $K > 0$ such that for all $n > K$, $|\frac{f(n)}{g(n)}| < C$.

It is important to realize that f is $O(g)$ if and only it is possible to find these constants C and k.

How you find them is the challenge.

More tricks

Theorem: Suppose $f : \mathbb{N} \to \mathbb{R}$ and $g : \mathbb{N} \to \mathbb{R}$ and

$$\lim_{n\to\infty} |\frac{f(n)}{g(n)}| = C$$

for some constant C. Then f is $O(g)$!

Proof: When $\lim_{n\to\infty} |\frac{f(n)}{g(n)}| = C$, then $\exists k$ such that for all $n > k, |\frac{f(n)}{g(n)}| < C + 1$.

Hence, $|f(n)| < (C + 1)|g(n)|$ for all $n > k$.

Hence, f is $O(g)$.

L'Hôpital's rule

L'Hôpital's rule: Suppose f and g are differentiable functions defined on the positive reals such that

$$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$$

Suppose also that

$$\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = C < \infty$$

Then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$$

Using L'Hôpital's Rule

Let $f(n) = n$ and $g(n) = e^n$. We want to prove that f is $O(g)$.

Proof:

To prove that f is $O(g)$, it must be that $\exists C, k > 0$ such that $\forall n > K, |f(n)| < C|g(n)|$.

Equivalently, $\forall n > K, |\frac{f(n)}{g(n)}| < C$.

We try to compute $\lim_{n\to\infty} |\frac{f(n)}{g(n)}|$, but it isn't easy to see what it is.

Can we use L'Hôpital's Rule?

Using L'Hôpital's rule

Remember $f(n) = n, g(n) = e^n$.
Note:

- $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$.
- $f'(n) = 1$ and $g'(n) = e^n$.
- $\lim_{n\to\infty} \frac{f'(n)}{g'(n)} = \lim_{n\to\infty} \frac{1}{e^n} = 0$

Hence, by L'Hôpital's Rule, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = 0$

Therefore, $f(n) \leq g(n)$ for all "large enough n", and so f is $O(g)$.

Important points

Question: Suppose you know that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C < \infty$. Does it follow that $f(n) \leq Cg(n)$ for all large enough n?

Answer: No! Think carefully about what limits prove.

Important points

Question: Suppose $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist. Does it follow that f is not $O(g)$?

Answer: No! The limit may not exist for many reasons, and yet f can be $O(g)$.

Important points

Question: Suppose $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. Does it follow that f is not $O(g)$?

Answer: YES! If the limit of the ratio is ∞, then it is not possible for f to be $O(g)$.

Running times for algorithms

We talk about algorithms being "polynomial time" if their worst case running times are $O(p(n))$ for some polynomial p.

Examples:

- Bubblesort is $O(n^2)$
- Mergesort is $O(n \log n)$
- Longest Increasing Subsequence is $O(mn)$ (where the two strings have length m and n)

But the following are also true:

- Bubblesort is $O(n^3)$
- Mergesort is $O(n^5)$

Summary (so far)

We

- Talked about recursively defined functions and how to write their running times recursively
- Defined "Big-O"
- Provided techniques for establishing that a function f is $O(g)$
- Remembered that big-o is an *upper bound* and is not necessarily tight!