CS173 Lecture B, October 8, 2015

Tandy Warnow

October 5, 2015

CS 173, Lecture B
October 8, 2015
Tandy Warnow

Today's material

We will cover:

- Recursively defined functions and how to write their running times recursively
- Running time analysis (and "Big Oh")

Specific algorithms:

- Bubblesort, as a recursive algorithm
- Mergesort, as a recursive algorithm

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

See Textbook section 14.7 for the last step.

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 $C = \max\{C_1, C_2\}$.
We will prove (using induction) that that $\forall n \in Z^+, t(n) \leq Cn^2$.

Bubblesort running time, continued

We rewrite the recurrence with $\leq$ and obtain the following recurrence for $t(n)$:

- $t(1) \leq C_1$, and
- $t(n) \leq C_2 n + t(n-1)$ if $n > 1$

Let $C = \max\{C_1, C_2\}$.
Our inductive hypothesis is that $\exists K \geq 1$ so that $\forall n \in \{1, 2, \ldots, K\}, t(n) \leq Cn^2$.
We note that $t(1) \leq C_1 \leq C = C * 1^2$, so the base case ($n = 1$) holds.

Bounding Bubblesort running time

We now wish to show that $t(K+1) \leq C(K+1)^2$.
Since $K+1 > 1$, we can use the recurrence relation and obtain $t(K+1) \leq C_2(K+1) + t(K) \leq C(K+1) + t(K)$.
Then we apply the I.H. to $t(K)$ and obtain $t(K) \leq CK^2$.
Hence

$$\begin{aligned} t(K+1) &\leq C(K+1) + t(K) \\ &\leq C(K+1) + C(K^2) \\ &= CK + C + CK^2 \\ &\leq CK^2 + 2CK + C \\ &= C(K+1)^2 \end{aligned}$$

Hence we have proved what we set out to prove. Since K was arbitrary, the theorem is true.

Mergesort - another sorting algorithm

MergeSort is another nice algorithm for sorting.
For the sake of this algorithm, assume $n = 2^k$.
Input: $A[1...n]$ of integers
Output: A in sorted order (smallest to largest)

- If $n = 1$, return the array
- Divide A into two arrays $A[1...m]$ and $A[m+1...n]$, where $m = \frac{n}{2}$.
- Recurse on $A[1...m]$ and on $A[m+1...n]$, so that each is sorted in increasing order
- Merge the two sorted arrays, by taking the smallest off the "top" of each array, and placing into a third array, until all elements moved into third array.

Running time analysis of Merge Sort

- If $n = 1$ the number of operations is C_1
- For $n > 1$,
  - the preprocessing is C_2
  - there are two subproblems, and each has $\frac{n}{2}$ elements, hence the recursive part uses $2t(\frac{n}{2})$
  - the postprocessing is $C_3 n$
- Let $C = C_1 + C_2 + C_3$

Hence, $t(n)$ (for $n > 1$) satisfies:

$$t(n) = 2t(\tfrac{n}{2}) + Cn, \text{ for some positive constant } C$$

It is not hard to see that for some constant $C' > 0$,
$t(n) \leq C' n \log n$ for all $n \in Z^+$.

Big Oh

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-oh is a way of communicating this.

Big Oh

Let $f : N \rightarrow R^+$ and $g : N \rightarrow R^+$ be two functions.
We will say that f is "big oh" of g if $\exists C > 0$ and $N \geq 0$ such that $f(n) \leq Cg(n)$ for all $n > N$.
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-oh

- 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 Oh

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 $f : Z^+ \rightarrow R^+$ and $g : Z^+ \rightarrow R^+$. How can we prove that f is $O(g)$?

- Find positive constants C and K such that $f(n) \leq Cg(n)$ 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 $f : R^+ \to R^+$ and $g : R^+ \to R^+$, 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 $\forall n > K, f(n) < Cg(n)$.
Equivalently, $\forall n > K, \frac{f(n)}{g(n)} < Cg(n)$.
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 $O(g)$.

Summary (so far)

We

- Talked about recursively defined functions and how to write their running times recursively
- Discussed "Big Oh"