CS 4104: Data and Algorithm Analysis

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Fall 2010

Factorial Growth (1)

Which function grows faster? $f(n) = 2^n$ or $g(n) = n!$

How about $h(n) = 2^{2n}$?

Factorial Growth (1)

Which function grows faster? $f(n) = 2^n$ or $g(n) = n!$

How about $h(n) = 2^{2n}$?

	n	1	2	3	4	5	6	7	8
$g(n)$	$n!$	1	2	6	24	120	720	5040	40320
$f(n)$	2^n	2	4	8	16	32	64	128	256
$h(n)$	2^{2n}	4	16	64	256	1024	4096	16384	65536

Factorial Growth (1)

Consider the recurrences:

$$h(n) = \begin{cases} 4 & n = 1 \\ 4h(n-1) & n > 1 \end{cases}$$

$$g(n) = \begin{cases} 1 & n = 1 \\ ng(n-1) & n > 1 \end{cases}$$

I hope your intuition tells you the right thing.

But, how do you PROVE it?

Induction? What is the base case?

Using Logarithms (1)

$n! \geq 2^{2n}$ iff $\log n! \geq \log 2^{2n} = 2n$. Why?

Using Logarithms (1)

$n! \geq 2^{2n}$ iff $\log n! \geq \log 2^{2n} = 2n$. Why?

$$\begin{aligned} n! &= n \times (n-1) \times \cdots \times \frac{n}{2} \times (\frac{n}{2} - 1) \times \cdots \times 2 \times 1 \\ &\geq \frac{n}{2} \times \frac{n}{2} \times \cdots \times \frac{n}{2} \times 1 \times \cdots \times 1 \times 1 \\ &= (\frac{n}{2})^{n/2} \end{aligned}$$

Using Logarithms (1)

$n! \geq 2^{2n}$ iff $\log n! \geq \log 2^{2n} = 2n$. Why?

$$\begin{aligned} n! &= n \times (n-1) \times \cdots \times \frac{n}{2} \times (\frac{n}{2} - 1) \times \cdots \times 2 \times 1 \\ &\geq \frac{n}{2} \times \frac{n}{2} \times \cdots \times \frac{n}{2} \times 1 \times \cdots \times 1 \times 1 \\ &= (\frac{n}{2})^{n/2} \end{aligned}$$

Therefore

$$\log n! \geq \log(\frac{n}{2})^{n/2} = (\frac{n}{2}) \log(\frac{n}{2}).$$

Need only show that this grows to be bigger than 2n.

Using Logarithms (2)

$$\begin{array}{rrcl} & (\frac{n}{2})\log(\frac{n}{2}) & \geq & 2n \\ \iff & \log(\frac{n}{2}) & \geq & 4 \\ \iff & n & \geq & 32 \end{array}$$

So, $n! \geq 2^{2n}$ once $n \geq 32$.

Now we could prove this with induction, using 32 for the base case.

- What is the tightest base case?
- How did we get such a big over-estimate?

Logs and Factorials

We have proved that $n! \in \Omega(2^{2n})$.

We have also proved that $\log n! \in \Omega(n \log n)$.

From here, its easy to prove that $\log n! \in O(n \log n)$, so $\log n! = \Theta(n \log n)$.

This does **not** mean that $n! = \Theta(n^n)$.

- Note that $\log n = \Theta(\log n^2)$ but $n \neq \Theta(n^2)$.
- The log function is a “flattener” when dealing with asymptotics.

A Simple Sum (1)

```
sum = 0; inc = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++) {
    sum = sum + inc;
    inc++;
  }
```

Use summations to analyze this code fragment. The number of assignments is:

$$2 + \sum_{i=1}^{n}(\sum_{j=1}^{i} 2) = 2 + \sum_{i=1}^{n} 2i = 2 + 2\sum_{i=1}^{n} i$$

A Simple Sum (2)

Give a good estimate.

- Observe that the biggest term is $2 + 2n$ and there are n terms, so its at most:

A Simple Sum (2)

Give a good estimate.

- Observe that the biggest term is $2 + 2n$ and there are n terms, so its at most: $2n + 2n^2$

A Simple Sum (2)

Give a good estimate.

- Observe that the biggest term is $2 + 2n$ and there are n terms, so its at most: $2n + 2n^2$
- Actually, most terms are much less, and its a linear ramp, so a better estimate is:

A Simple Sum (2)

Give a good estimate.

- Observe that the biggest term is $2 + 2n$ and there are n terms, so its at most: $2n + 2n^2$
- Actually, most terms are much less, and its a linear ramp, so a better estimate is: about n^2.

A Simple Sum (2)

Give a good estimate.

- Observe that the biggest term is $2 + 2n$ and there are n terms, so its at most: $2n + 2n^2$
- Actually, most terms are much less, and its a linear ramp, so a better estimate is: about n^2.

Give the exact solution.

- Of course, we all know the closed form solution for $\sum_{i=1}^{n} i$.
- And we should all know how to prove it using induction.
- But where did it come from?

A Problem-Specific Approach

Observe that we can "pair up" the first and last terms, the 2nd and $(n-1)$th terms, and so on. Each pair sums to:

A Problem-Specific Approach

Observe that we can “pair up” the first and last terms, the 2nd and $(n-1)$th terms, and so on. Each pair sums to: $n+1$.

The number of pairs is:

A Problem-Specific Approach

Observe that we can “pair up” the first and last terms, the 2nd and $(n-1)$th terms, and so on. Each pair sums to: $n+1$.

The number of pairs is: $n/2$.

Thus, the solution is:

A Problem-Specific Approach

Observe that we can “pair up” the first and last terms, the 2nd and $(n-1)$th terms, and so on. Each pair sums to: $n+1$.

The number of pairs is: $n/2$.

Thus, the solution is: $(n+1)(n/2)$.

A Little More General

Since the largest term is n and there are n terms, the summation is less than n^2.

If we are lucky, the solution is a polynomial.

Guess: $f(n) = c_1 n^2 + c_2 n + c_3$.
$f(0) = 0$ so $c_3 = 0$.
For $f(1)$, we get $c_1 + c_2 = 1$.
For $f(2)$, we get $4c_1 + 2c_2 = 3$.
Setting this up as a system of 2 equations on 2 variables, we can solve to find that $c_1 = 1/2$ and $c_2 = 1/2$.

More General (2)

So, if it truely is a polynomial, it **must** be

$$f(n) = n^2/2 + n/2 + 0 = \frac{n(n+1)}{2}.$$

Use induction to prove. Why is this step necessary?

Why is this not a universal approach to solving summations?

An Even More General Approach

Subtract-and-Guess or **Divide-and-Guess** strategies.

To solve sum f, pick a known function g and find a pattern in terms of $f(n) - g(n)$ or $f(n)/g(n)$.

Find the closed form solution for

$$f(n) = \sum_{i=1}^{n} i.$$

Guessing (cont.)

Examples: Try $g_1(n) = n$; $g_2(n) = f(n-1)$.

n	1	2	3	4	5	6	7	8
$f(n)$	1	3	6	10	15	21	28	36
$g_1(n)$	1	2	3	4	5	6	7	8
$f(n)/g_1(n)$	2/2	3/2	4/2	5/2	6/2	7/2	8/2	9/2
$g_2(n)$	0	1	3	6	10	15	21	28
$f(n)/g_2(n)$		3/1	4/2	5/3	6/4	7/5	8/6	9/7

What are the patterns?

$\frac{f(n)}{g_1(n)} =$

$\frac{f(n)}{g_2(n)} =$

Solving Summations (cont.)

Use algebra to rearrange and solve for $f(n)$

$$\frac{f(n)}{n} = \frac{n+1}{2}$$

$$\frac{f(n)}{f(n-1)} = \frac{n+1}{n-1}$$

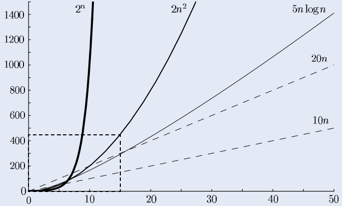
Solving Summations (cont.)

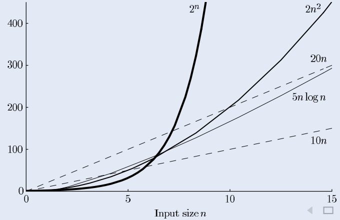
$$\begin{aligned}
\frac{f(n)}{f(n-1)} &= \frac{n+1}{n-1} \\
f(n)(n-1) &= (n+1)f(n-1) \\
f(n)(n-1) &= (n+1)(f(n)-n) \\
nf(n) - f(n) &= nf(n) + f(n) - n^2 - n \\
2f(n) &= n^2 + n = n(n+1) \\
f(n) &= \frac{n(n+1)}{2}
\end{aligned}$$

Important Note: This is **not a proof** that $f(n) = n(n+1)/2$. Why?

Growth Rates

Two functions of n have different **growth rates** if as n goes to infinity their ratio either goes to infinity or goes to zero.





Estimating Growth Rates

Exact equations relating program operations to running time require machine-dependent constants.

Sometimes, the equation for exact running time is complicated to compute.

Usually, we are satisfied with knowing an approximate growth rate.

Example: Given two algorithms with growth rate $c_1 n$ and $c_2 2^{n!}$, do we need to know the values of c_1 and c_2?

Consider n^2 and $3n$. PROVE that n^2 must eventually become (and remain) bigger.

Proof by Contradiction

Assume there are some values for constants r and s such that, for all values of n,

$$n^2 < rn + s.$$

Then, $n < r + s/n$.

But, as n grows, what happens to s/n?

Since n grows toward infinity, the assumption must be false.

Some Growth Rates (1)

Since n^2 grows faster than n,

- 2^{n^2} grows faster than 2^n.
- n^4 grows faster than n^2.
- n grows faster than $\sqrt{n}$.
- $2 \log n$ grows <u>no slower</u> than $\log n$.

Some Growth Rates (2)

Since $n!$ grows faster than 2^n,

- $n!!$ grows faster than $2^n!$.
- $2^{n!}$ grows faster than 2^{2^n}.
- $n!^2$ grows faster than 2^{2n}.
- $\sqrt{n!}$ grows faster than $\sqrt{2^n}$.
- $\log n!$ grows no slower than n.

Some Growth Rates (3)

If f grows faster than g, then

- Must $\sqrt{f}$ grow faster than $\sqrt{g}$?
- Must $\log f$ grow faster than $\log g$?

$\log n$ is related to n in exactly the same way that n is related to 2^n.

- $2^{\log n} = n$

Fibonacci Numbers (Iterative)

$f(n) = f(n-1) + f(n-2)$ for $n \geq 2$; $f(0) = f(1) = 1$.

```
long Fibi(int n) {
  long past, prev, curr;
  past = prev = curr = 1;     // curr holds Fib(i)
  for (int i=2; i<=n; i++) {  // Compute next value
    past = prev; prev = curr; // past holds Fib(i-2)
    curr = past + prev;       // prev holds Fib(i-1)
  }
  return curr;
}
```

The cost of Fibi is easy to compute:

Fibonacci Numbers (Recursive)

```
int Fibr(int n) {
  if ((n <= 1) return 1;            // Base case
  return Fibr(n-1) + Fibr(n-2);     // Recursive call
}
```

What is the cost of Fibr?

Analysis of Fibr

Use divide-and-guess with $f(n-1)$.

n	1	2	3	4	5	6	7
$f(n)$	1	2	3	5	8	13	21
$f(n)/f(n-1)$	1	2	1.5	1.666	1.625	1.615	1.619

Following this out, it appears to settle to a ratio of 1.618.

Assuming $f(n)/f(n-1)$ really tends to a fixed value x, let's verify what x must be.

$$\frac{f(n)}{f(n-2)} = \frac{f(n-1)}{f(n-2)} + \frac{f(n-2)}{f(n-2)} \rightarrow x + 1$$

Analysis of Fibr (cont.)

For large n,

$$\frac{f(n)}{f(n-2)} = \frac{f(n)}{f(n-1)} \frac{f(n-1)}{f(n-2)} \rightarrow x^2$$

If x exists, then $x^2 - x - 1 \rightarrow 0$.

Using the quadratic equation, the only solution greater than one is

$$x = \frac{1+\sqrt{5}}{2} \approx 1.618.$$

What does this say about the growth rate of f?

Order Notation

little oh	$f(n) \in o(g(n))$	$<$	$\lim f(n)/g(n) = 0$
big oh	$f(n) \in O(g(n))$	$\leq$	
Theta	$f(n) = \Theta(g(n))$	$=$	$f = O(g)$ and $g = O(f)$
Big Omega	$f(n) \in \Omega(g(n))$	$\geq$	
Little Omega	$f(n) \in \omega(g(n))$	$>$	$\lim g(n)/f(n) = 0$

I prefer "$f \in O(n^2)$" to "$f = O(n^2)$"

- While $n \in O(n^2)$ and $n^2 \in O(n^2)$, $O(n) \neq O(n^2)$.

Note: Big oh does not say how good an algorithm is – only how bad it CAN be.

If $\mathcal{A} \in O(n)$ and $\mathcal{B} \in O(n^2)$, is $\mathcal{A}$ better than $\mathcal{B}$?

Perhaps... but perhaps better analysis will show that $\mathcal{A} = \Theta(n)$ while $\mathcal{B} = \Theta(\log n)$.

Limitations on Order Notation

Statement: Algorithm $\mathcal{A}$'s resource requirements grow slower than Algorithm $\mathcal{B}$'s resource requirements.

Is $\mathcal{A}$ better than $\mathcal{B}$?

Potential problems:

- How big must the input be?
- Some growth rate differences are trivial
  - Example: $\Theta(\log^2 n)$ vs. $\Theta(n^{1/10})$.
- It is not always practical to reduce an algorithm's growth rate
  - Shaving a factor of n reduces cost by a factor of a million for input size of a million.
  - Shaving a factor of $\log \log n$ saves only a factor of 4-5.

Practicality Window

In general:

- We have limited time to solve a problem.
- We have a limited input size.

Fortunately, algorithm growth rates are USUALLY well behaved, so that Order Notation gives practical indications.