CSE 332 Data Abstractions:

Algorithmic, Asymptotic, and Amortized Analysis

Kate Deibel

Summer 2012

Announcements

- Project 1 posted
- Homework 0 posted
- Homework 1 posted this afternoon
- Feedback on typos is welcome

- New Section Location: CSE 203
  - Comfy chairs! :O
  - White board walls! :o
  - Reboot coffee 100 yards away :)
  - Kate's office is even closer :/

Today

- Briefly review math essential to algorithm analysis
  - Proof by induction
  - Powers of 2
  - Exponents and logarithms

- Begin analyzing algorithms
  - Big-O, Big-Ω, and Big-Θ notations
  - Using asymptotic analysis
  - Best-case, worst-case, average case analysis
  - Using amortized analysis

If you understand the first n slides,
you will understand the n+1 slide

MATH REVIEW

Recurrence Relations

Functions that are defined using themselves (think recursion but mathematically):

- $F(n) = n \cdot F(n-1)$, $F(0) = 1$
- $G(n) = G(n-1) + G(n-2)$, $G(1)=G(2) = 1$
- $H(n) = 1 + H(\lfloor n/2 \rfloor)$, $H(1)=1$

Some recurrence relations can be written more simply in closed form (non-recursive)

$\lfloor x \rfloor$ is the floor function (first integer $\leq x$)

$\lceil x \rceil$ is the ceiling function (first integer $\geq x$)

Example Closed Form

$H(n) = 1 + H(\lfloor n/2 \rfloor), H(1)=1$

- $H(1) = 1$
- $H(2) = 1 + H(\lfloor 2/2 \rfloor) = 1 + H(1) = 2$
- $H(3) = 1 + H(\lfloor 3/2 \rfloor) = 1 + H(1) = 2$
- $H(4) = 1 + H(\lfloor 4/2 \rfloor) = 1 + H(2) = 3$

…

- $H(8) = 1 + H(\lfloor 8/2 \rfloor) = 1 + H(4) = 4$

…

$H(n) = 1 + \lfloor \log_2 n \rfloor$

Mathematical Induction

Suppose P(n) is some predicate (with integer n)

- Example: $n \geq n/2 + 1$

To prove P(n) for all $n \geq c$, it suffices to prove

1. P(c) – called the "basis" or "base case"
2. If P(k) then P(k+1) – called the "induction step" or "inductive case"

When we will use induction:

- To show an algorithm is correct or has a certain running time no matter how big a data structure or input value is
- Our "n" will be the data structure or input size.

Induction Example

The sum of the first n powers of 2 (starting with zero) is given the by formula:

$$P(n) = 2^n-1$$

Theorem: P(n) holds for all $n \geq 1$

Proof: By induction on n

Base case: n=1.

- Sum of first power of 2 is 2^0 , which equals 1.
- And for n=1,

$$2^n-1 = 2^1-1 = 2-1 = 1$$

Induction Example

The sum of the first n powers of 2 (starting with zero) is given the by formula:

$$P(n) = 2^n-1$$

Inductive case:

- Assume: sum of the first k powers of 2 is 2^k-1
- Show: sum of the first (k+1) powers is $2^{k+1}-1$
- $P(k+1) = 2^0+2^1+\ldots+2^{k+1-2}+2^{k+1-1}$

$$= (2^0+2^1+\ldots+2^{k-1})+2^k$$

$$= (2^k-1)+2^k \quad \text{since } P(k)=2^0+2^1+\ldots+2^{k-1}= 2^k-1$$

$$= 2\cdot 2^k-1$$

$$= 2^{k+1}-1$$

Powers of 2

- A bit is 0 or 1
- n bits can represent 2^n distinct things
  - For example, the numbers 0 through 2^n-1

Rules of Thumb:

- 2^{10} is 1024 / “about a thousand”, kilo in CSE speak
- 2^{20} is “about a million”, mega in CSE speak
- 2^{30} is “about a billion”, giga in CSE speak

In Java:

- **int** is 32 bits and signed, so “max int” is 2^{31} - 1 which is about 2 billion
- **long** is 64 bits and signed, so “max long” is 2^{63} - 1

Therefore…

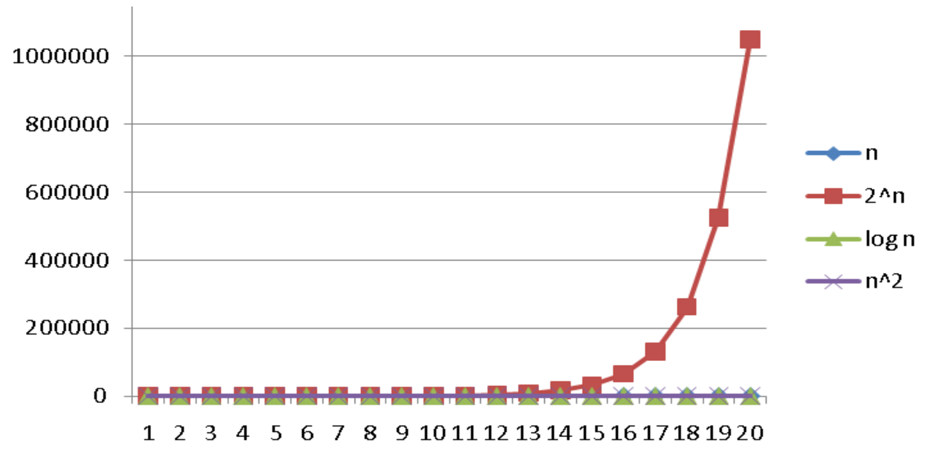
One can give a unique id to:

- Every person in the U.S. with 29 bits
- Every person in the world with 33 bits
- Every person to have ever lived with ≈38 bits
- Every atom in the universe with 250-300 bits
- So if a password is 128 bits long and randomly generated, do you think you could guess it?

Logarithms and Exponents

- Since so much in CS is in binary,
  `log` almost always means $\log_2$
- Definition: $\log_2 x = y$ if $x = 2^y$
- So, $\log_2$ 1,000,000 = "a little under 20"
- Just as exponents grow *very* quickly, logarithms grow *very* slowly

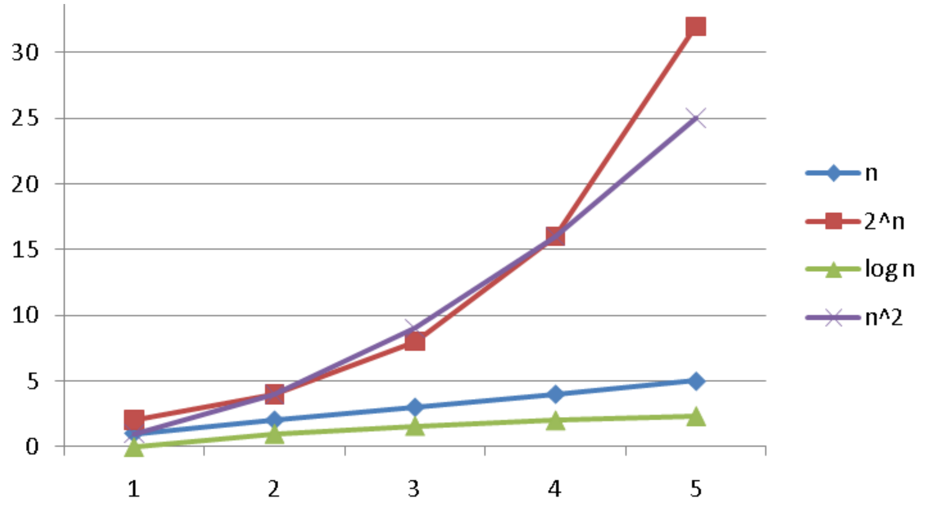
See Excel file on course page to play with plot data!



Logarithms and Exponents

- Since so much in CS is in binary,
  `log` almost always means $\log_2$
- Definition: $\log_2 x = y$ if $x = 2^y$
- So, $\log_2$ 1,000,000 = "a little under 20"
- Just as exponents grow *very* quickly, logarithms grow *very* slowly

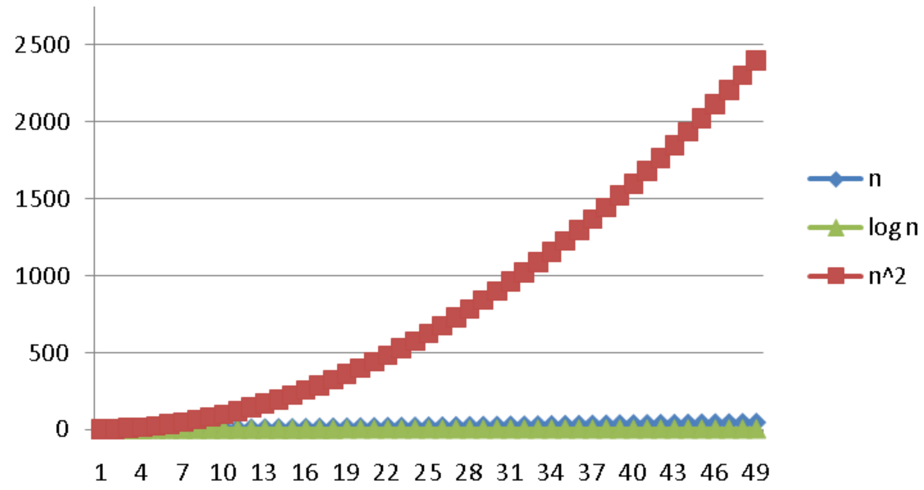
See Excel file on course page to play with plot data!



Logarithms and Exponents

- Since so much in CS is in binary,
  `log` almost always means $\log_2$
- Definition: $\log_2 x = y$ if $x = 2^y$
- So, $\log_2$ 1,000,000 = "a little under 20"
- Just as exponents grow *very* quickly, logarithms grow *very* slowly

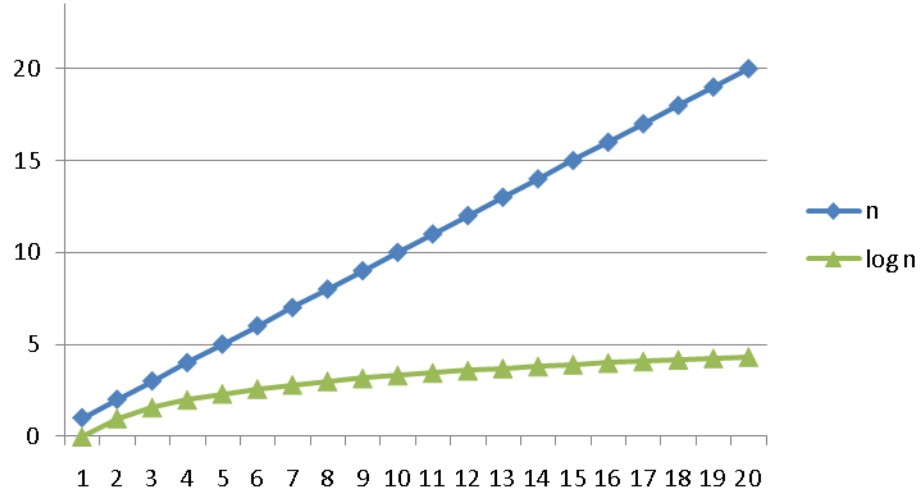
See Excel file on course page to play with plot data!



Logarithms and Exponents

- Since so much in CS is in binary,
  `log` almost always means $\log_2$
- Definition: $\log_2 x = y$ if $x = 2^y$
- So, $\log_2$ 1,000,000 = "a little under 20"
- Just as exponents grow *very* quickly, logarithms grow *very* slowly

See Excel file on course page to play with plot data!



Logarithms and Exponents

- log(A*B) = log A + log B
- $\log(N^k) = k \log N$
- log(A/B) = log A – log B
- log(log x) is written log log x
  - Grows as slowly as 2^{2^x} grows fast
- (log x)(log x) is written $\log^2 x$
  - It is greater than log x for all x > 2

Logarithms and Exponents

Any base B log is equivalent to base 2 log within a constant factor

In particular,

$$\log_2 x = 3.22 \log_{10} x$$

In general,

$$\log_B x = (\log_A x) / (\log_A B)$$

This matters in doing math but not CS!

In algorithm analysis, we tend to not care much about constant factors

Get out your stopwatches… or not

ALGORITHM ANALYSIS

Algorithm Analysis

As the "size" of an algorithm's input grows (array length, size of queue, etc.):

- Time: How much longer does it run?
- Space: How much memory does it use?

How do we answer these questions?

For now, we will focus on time only.

One Approach to Algorithm Analysis

Why not just code the algorithm and time it?

- Hardware: processor(s), memory, etc.
- OS, version of Java, libraries, drivers
- Programs running in the background
- Implementation dependent
- Choice of input
- Number of inputs to test

The Problem with Timing

- Timing doesn’t really evaluate the algorithm but merely evaluates a specific implementation
- At the core of CS is a backbone of theory & mathematics
  - Examine the algorithm itself, **not** the implementation
  - Reason about performance as a function of n
  - Mathematically prove things about performance
- Yet, timing has its place
  - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
  - Ex: Benchmarking graphics cards

Basic Lesson

Evaluating an algorithm?

Use asymptotic analysis

Evaluating an implementation?

Use timing

Goals of Comparing Algorithms

Many measures for comparing algorithms

- Security
- Clarity/ Obfuscation
- Performance

When comparing performance

- Use large inputs because probably any algorithm is "plenty good" for small inputs ($n < 10$ always fast)
- Answer should be independent of CPU speed, programming language, coding tricks, etc.
- Answer is general and rigorous, complementary to "coding it up and timing it on some test cases"

Assumptions in Analyzing Code

Basic operations take constant time

- Arithmetic (fixed-width)
- Assignment
- Access one Java field or array index
- Comparing two simple values (is $x < 3$)

Other operations are summations or products

- Consecutive statements are summed
- Loops are (cost of loop body) $\times$ (number of loops)

What about conditionals?

Worst-Case Analysis

- In general, we are interested in three types of performance
  - Best-case / Fastest
  - Average-case
  - Worst-case / Slowest
- When determining worst-case, we tend to be pessimistic
  - If there is a conditional, count the branch that will run the slowest
  - This will give a loose bound on how slow the algorithm may run

Analyzing Code

What are the run-times for the following code?

Answers are

1.
```
for(int i=0;i<n;i++)
 x = x+1;
```
$\approx 1+4n$

2.
```
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        x = x + 1
```
$\approx 4n^2$

3.
```
for(int i=0;i<n;i++)
    for(int j=0; j <= i); j++)
        x = x + 1
```
$\approx 4(1+2+\ldots+n)$

$\approx 4n(n+1)/2$

$\approx 2n^2+2n+2$

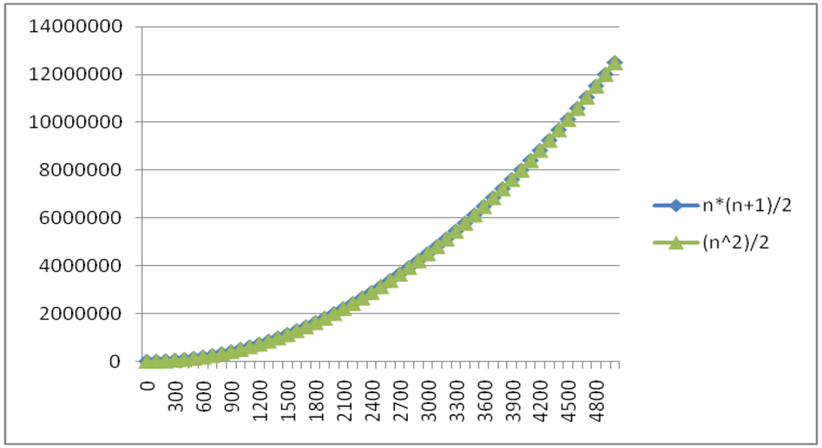
No Need To Be So Exact

Constants do not matter

- Consider $6N^2$ and $20N^2$
- When N >> 20, the N^2 is what is driving the function's increase

Lower-order terms are also less important

- $N*(N+1)/2$ vs. just $N^2/2$
- The linear term is inconsequential

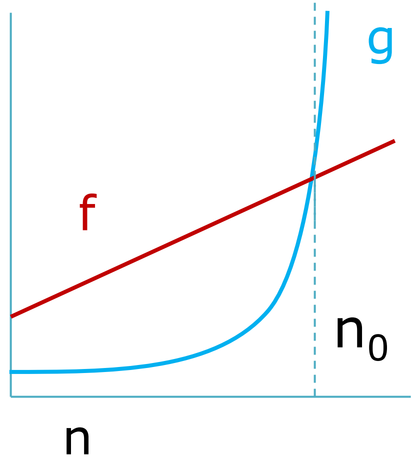


We need a better notation for performance that focuses on the dominant terms only

Big-Oh Notation

- Given two functions f(n) & g(n) for input n, we say f(n) is in O(g(n)) iff there exist positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \quad \text{for all } n \geq n_0$$



- Basically, we want to find a function g(n) that is eventually always bigger than f(n)

The Gist of Big-Oh

Take functions f(n) & g(n), consider only the most significant term and remove constant multipliers:

- $5n+3 \rightarrow n$
- $7n+.5n^2+2000 \rightarrow n^2$
- $300n+12+n\log n \rightarrow n \log n$
- $-n \rightarrow$??? A negative run-time?

Then compare the functions; if $f(n) \leq g(n)$, then $f(n)$ is in $O(g(n))$

A Big Warning

Do NOT ignore constants that are not multipliers:

n^3 is $O(n^2)$ is FALSE

3^n is $O(2^n)$ is FALSE

When in doubt, refer to the rigorous definition of Big-Oh

Examples

- True or false?

1. $4+3n$ is $O(n)$ — True
2. $n+2 \log n$ is $O(\log n)$ — False
3. $\log n+2$ is $O(1)$ — False
4. n^{50} is $O(1.1^n)$ — True

Examples (cont.)

For $f(n)=4n$ & $g(n)=n^2$, prove $f(n)$ is in $O(g(n))$

A valid proof is to find valid c and n_0

When n=4, f=16 and g=16, so this is the crossing over point

We can then chose $n_0 = 4$, and c=1

We also have infinitely many others choices for c and n_0, such as $n_0 = 78$, and c=42

Big Oh: Common Categories

From fastest to slowest

$O(1)$	constant (or $O(k)$ 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 constant)
$O(k^n)$	exponential (where constant $k > 1$)

Caveats

- Asymptotic complexity focuses on behavior for large *n* and is independent of any computer/coding trick, but results can be misleading
- Example: $n^{1/10}$ vs. **`log`** *n*
  - Asymptotically $n^{1/10}$ grows more quickly
  - But the "cross-over" point is around $5 * 10^{17}$
  - So if you have input size less than 2^{58}, prefer $n^{1/10}$
  - Similarly, an $O(2^n)$ algorithm may be more practical than an $O(n^7)$ algorithm

Caveats

- Even for more common functions, comparing O() for small n values can be misleading
  - Quicksort: O(n log n) (expected)
  - Insertion Sort: $O(n^2)$(expected)
  - In reality Insertion Sort is faster for small n's so much so that good QuickSort implementations switch to Insertion Sort when n<20

Comment on Notation

- We say $(3n^2+17)$ **is in** $O(n^2)$
- We may also say/write is as
  - $(3n^2+17)$ **is** $O(n^2)$
  - $(3n^2+17)$ **=** $O(n^2)$
  - $(3n^2+17) \in O(n^2)$
- But it's not '=' as in 'equality':
  - We would never say $O(n^2) = (3n^2+17)$

Big Oh's Family

- Big Oh: 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 $n \geq n_0$

- Big Omega: 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 $n \geq n_0$

- Big Theta: Tight bound: $\Theta(\ f(n)\ )$ is the set of all functions asymptotically equal to $f(n)$
  - Intersection of $O(\ f(n)\ )$ and $\Omega(\ f(n)\ )$

Regarding use of terms

Common error is to say $O(f(n))$ when you mean $\Theta(f(n))$

- People often say O() to mean a tight bound
- Say we have f(n)=n; we could say f(n) is in O(n), which is true, but only conveys the upper-bound
- Somewhat incomplete; instead say it is $\Theta(n)$
- That means that it is not, for example $O(\texttt{log}\ n)$

Less common notation:

- "little-oh": like "big-Oh" but strictly less than
  - Example: sum is $o(n^2)$ but not $o(n)$
- "little-omega": like "big-Omega" but strictly greater than
  - Example: sum is $\omega(\texttt{log}\ n)$ but not $\omega(n)$

Putting them in order

$$\omega(\ldots) < \Omega(\ldots) \leq f(n) \leq O(\ldots) < o(\ldots)$$

Do Not Be Confused

- Best-Case does not imply $\Omega(f(n))$
- Average-Case does not imply $\Theta(f(n))$
- Worst-Case does not imply $O(f(n))$

- Best-, Average-, and Worst- are specific to the algorithm
- $\Omega(f(n))$, $\Theta(f(n))$, $O(f(n))$ describe functions
  - One can have an $\Omega(f(n))$ bound of the worst-case performance (worst is at least f(n))
  - Once can have a $\Theta(f(n))$ of best-case (best is exactly f(n))

Now to the Board

- What happens when we have a costly operation that only occurs some of the time?

- Example:

  My array is too small. Let's enlarge it.

  Option 1: Increase array size by 10
  Copy old array into new one

  Option 2: Double the array size
  Copy old array into new one

We will now explore amortized analysis!

Stretchy Array (version 1)

StretchyArray:

maxSize: positive integer (starts at 1)

array: an array of size maxSize

count: number of elements in array

```
put(x): add x to the end of the array
    if maxSize == count
        make new array of size (maxSize + 5)
        copy old array contents to new array
        maxSize = maxSize + 5
    array[count] = x
    count = count + 1
```

Stretchy Array (version 2)

StretchyArray:

```
    maxSize: positive integer (starts at 0)
    array: an array of size maxSize
    count: number of elements in array

    put(x): add x to the end of the array
        if maxSize == count
            make new array of size (maxSize * 2)
            copy old array contents to new array
            maxSize = maxSize * 2
        array[count] = x
        count = count + 1
```

Performance Cost of put(x)

In both stretchy array implementations, put(x)is defined as essentially:

```
if maxSize == count
    make new array of bigger size
    copy old array contents to new array
    update maxSize
array[count] = x
count = count + 1
```

What f(n) is put(x) in O(f(n))?

Performance Cost of put(x)

In both stretchy array implementations, put(x)is defined as essentially:

if maxSize == count	O(1)
make new array of bigger size	O(1)
copy old array contents to new array	O(n)
update maxSize	O(1)
array[count] = x	O(1)
count = count + 1	O(1)

In the worst-case, put(x) is O(n) where n is the current size of the array!!

But…

- We do not have to enlarge the array each time we call put(x)
- What will be the average performance if we put n items into the array?

$$\frac{\sum_{i=1}^{n} \text{cost of calling put for the ith time}}{n} = \mathrm{O}(?)$$

- Calculating the average cost for multiple calls is known as *amortized analysis*

Amortized Analysis of StretchyArray Version 1

i	maxSize	count	cost	*comments*
	0	0		Initial state
1	5	1	0 + 1	Copy array of size 0
2	5	2	1	
3	5	3	1	
4	5	4	1	
5	5	5	1	
6	10	6	5 + 1	Copy array of size 5
7	10	7	1	
8	10	8	1	
9	10	9	1	
10	10	10	1	
11	15	11	10 + 1	Copy array of size 10
⁞	⁞	⁞	⁞	⁞

Amortized Analysis of StretchyArray Version 1

i	maxSize	count	cost	*comments*
	0	0		Initial state
1	5	1	0 + 1	Copy array of size 0
2	5	2	1	
			1	
			1	
			1	
			5 + 1	Copy array of size 5
			1	
8	10	8	1	
9	10	9	1	
10	10	10	1	
11	15	11	10 + 1	Copy array of size 10
⋮	⋮	⋮	⋮	⋮

Every five steps, we have to do a multiple of five more work

Amortized Analysis of StretchyArray Version 1

Assume the number of puts is n=5k

- We will make n calls to array[count]=x
- We will stretch the array k times and will cost:

$$0 + 5 + 10 + \ldots + 5(k-1)$$

Total cost is then:

$n + (0 + 5 + 10 + \ldots + 5(k-1))$

$= n + 5(1 + 2 + \ldots +(k-1))$

$= n + 5(k-1)(k-1+1)/2$

$= n + 5k(k-1)/2$

$\approx n + n^2/10$

Amortized cost for put(x) is

$$\frac{n + \frac{n^2}{10}}{n} = 1 + \frac{n}{10} = O(n)$$

Amortized Analysis of StretchyArray Version 2

i	maxSize	count	cost	*comments*
	1	0		Initial state
1	1	1	1	
2	2	2	1 + 1	Copy array of size 1
3	4	3	2 + 1	Copy array of size 2
4	4	4	1	
5	8	5	4 + 1	Copy array of size 4
6	8	6	1	
7	8	7	1	
8	8	8	1	
9	16	9	8 + 1	Copy array of size 8
10	16	10	1	
11	16	11	1	
⋮	⋮	⋮	⋮	⋮

Amortized Analysis of StretchyArray Version 2

i	maxSize	count	cost	*comments*
	1	0		Initial state
1	1	1	1	
2	2	2	1 + 1	Copy array of size 1
3	4	3	2 + 1	
4	4	4	1	
5	8	5	4 + 1	
6	8	6	1	
7	8	7	1	
8	8	8	1	
9	16	9	8 + 1	Copy array of size 8
10	16	10	1	
11	16	11	1	
⋮	⋮	⋮	⋮	⋮

Enlarge steps happen basically when i is a power of 2

Amortized Analysis of StretchyArray Version 2

Assume the number of puts is $n=2^k$

- We will make n calls to array[count]=x
- We will stretch the array k times and will cost:

$$\approx 1 + 2 + 4 + \ldots + 2^{k-1}$$

Total cost is then:

$\approx n + (1 + 2 + 4 + \ldots + 2^{k-1})$

$\approx n + 2^k - 1$

$\approx 2n - 1$

Amortized cost for put(x) is

$$\frac{2n-1}{n} = 2 - \frac{1}{n} = O(1)$$

The Lesson

With amortized analysis, we know that over the long run (on average):

- If we stretch an array by a constant amount, each put(x) call is O(n) time
- If we double the size of the array each time, each put(x) call is O(1) time