CSE 250: Recursion

Lecture 11

Sept 22, 2023

Reminders

- PA1 Implementation due Sun, Sept 24 at 11:59 PM
  - Implement a Sorted Linked List
  - Extra office hours **Davis 338A 6-8 PM Saturday**
- WA2 due Sun, Oct 1 at 11:59 PM
  - Will be released over the next weekend
- Midterm 1 on Oct 2 in-class
  - Covers: Asymptotics, Sequences/Lists, Arrays, Linked Lists, Recursion
  - Bounds: Tight Upper/Lower, Unqualified vs Amortized vs Expected

ArrayList

`ArrayList`: An array with empty space at the end.

`add(v)`

- Use an empty slot if one is available
- When you run out of space, double the size.

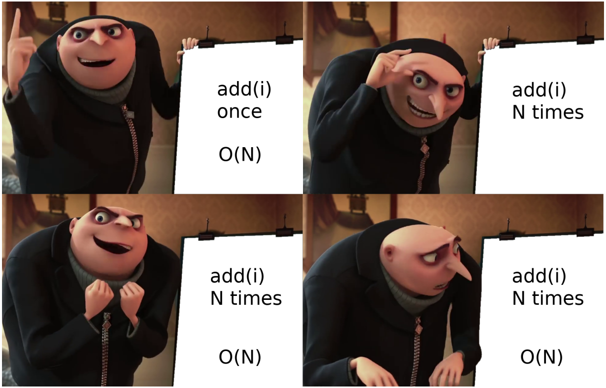
`add(idx, v)`

- As `add`, but shift elements $\geq$ `idx` right one space first.

`remove(idx)`

- Shift elements $>$ `idx` left one spot.

Huh?



Despicable Me; ©2010 Universal Pictures

Amortized Runtimes

$$T_{add}(N) = \begin{cases} \theta(1) & \textbf{if } \textit{capacity} > \textit{size} \\ \theta(N) & \textbf{otherwise} \end{cases}$$

$T_{add}(N) \in O(N)$

- Any **one** call could be $O(N)$
- But the $O(N)$ case happens rarely.
  - ... rarely enough (with doubling) that the expensive call *amortizes* over the cheap calls.

ArrayList

- Double the size
  - Copy 2^i elements (Put $\theta(2^i)$ on a credit card)
  - Get $2^i - 1$ $\theta(1)$ freebie inserts (Pay off over $\theta(2^i)$ calls to add)
- Contrast with always adding k slots
  - Copy $k \cdot i$ elements (Put $k \cdot i$ on a credit card)
  - Get k $\theta(1)$ freebie inserts (Pay off over $\theta(k)$ calls to add)

When doubling, the time you have to 'pay off' your card grows with the amount that goes on the card.

Amortized Runtime

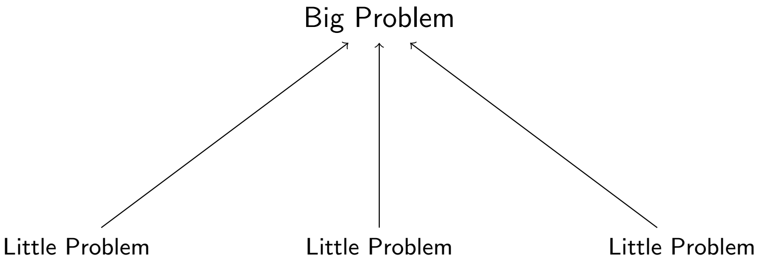
- The tight *unqualified* upper bound on `add(i)` is O(N)
  Any one call to `add(i)` could take up to $O(N)$.
- The tight *amortized* upper bound on `add(i)` is O(1)
  N calls to `add(i)` average out to $O(1)$ each.
  ($O(N)$ for all N calls)
  (Amortized lets you use a credit card, as long as you pay it off)

Recursion

Recursion

https://www.etsy.com/listing/916447505/ukrainian-nesting-doll-nesting-dolls

Algorithms



Recursive Algorithm: When the little problem is the same as the big problem, just smaller.

Factorial

$$\begin{aligned} 439! &= 439 \cdot 438 \cdot 437 \cdot 436 \cdot 435 \cdot 434 \cdot 433 \cdot 432 \cdot \ldots \\ &= 439 \cdot 438! \end{aligned}$$

$$438! = 438 \cdot 437 \cdot 436 \cdot 435 \cdot 434 \cdot 433 \cdot 432 \cdot \ldots$$

Factorial

$$N! = N \cdot (N-1)!$$

$$\underbrace{N!}_{\text{big problem}} = N \cdot \underbrace{(N-1)!}_{\text{smaller (same) problem}}$$

```
public long factorial(long N)
{
  return N * factorial(N-1);
}
```

StackOverflowError

Factorial

- $1! = 1$ — Base Case
- $N! = N \cdot (N-1)!$ — Recursive Case

```
public long factorial(long N)
{
  if(N <= 1){ return 1; }
  else { return N * factorial(N-1); }
}
```

Fibonacci

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Fibonacci

- $\texttt{Fib}(0) = 1$ (Base Case)
- $\texttt{Fib}(1) = 1$ (Base Case)
- $\texttt{Fib}(N) = \texttt{Fib}(N-1) + \texttt{Fib}(N-2)$ (Recursive Case)

```
public long fib(long N)
{
  if(n <= 1){ return 1; }
  else { fib(n-1) + fib(n-2) }
}
```

Towers of Hanoi

Live Demo!

Towers of Hanoi

Task: Move n blocks from **A** to **C**

Base Case ($n = 1$)

1. Move the Block from **A** to **C**

Base Case ($n \geq 2$)

1. Move $n - 1$ blocks from **A** to **B**
2. Move the n'th block from **A** to **C**
3. Move $n - 1$ blocks from **B** to **C**

Towers of Hanoi

```java
public void move(Tower from, Tower to, Tower other, int n)
{
  if(n == 1)
  {
    from.moveOneTo(to);
  }
  else
  {
    move(from, other, to, n-1);
    from.moveOneTo(to);
    move(other, to, from, n-1);
  }
}
```

How do we compute the complexity of recursive algorithms?

Factorial

What is the complexity of Factorial?

```
    public long factorial(long N)
    {
      if(N <= 1){ return 1; }
      else { return N * factorial(N-1); }
    }
```

$$T_{factorial}(N) = \begin{cases} \theta(1) & \textbf{if } N \leq 1 \\ \theta(1) + ???\, T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Runtime growth functions have base and recursive cases too.

Factorial

$$T_{factorial}(N) = \begin{cases} \theta(1) & \textbf{if } N \leq 1 \\ \theta(1) + T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Solve for $T_{factorial}(N)$.

Induction

Solve for $T_{factorial}(N)$.

Induction

1. Generate a hypothesis.
2. Prove the hypothesis for the base case.
3. Prove the hypothesis inductively.

Generate a Hypothesis

Hypothesis: Solve for increasing values of N

N	1	2	3	4	5	6
$T(N)$	$1 \cdot \theta(1)$	$2 \cdot \theta(1)$	$3 \cdot \theta(1)$	$4 \cdot \theta(1)$	$5 \cdot \theta(1)$	$6 \cdot \theta(1)$

What's the pattern? $(N \cdot \theta(1))$

Hypothesis: $T(N) \in \theta(N)$

- There is some $c_{high} > 0$ such that $T(n) \leq c_{high} \cdot n$ $\leftarrow$
- There is some $c_{low} > 0$ such that $T(n) \geq c_{low} \cdot n$

Algebra with θ

Remember, $\theta(N)$ used in a math equation is shorthand for:
$f(N)$ where $f(N) \in \theta(N)$

So $\theta(1)$ is shorthand for some constant c.

Factorial

$$T_{factorial}(N) = \begin{cases} \theta(1)c_1 & \textbf{if } N \leq 1 \\ \theta(1)c_2 + T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Goal: Show that $T_{factorial}(N) \leq c \cdot N$ for some $c > 0$ $(N > N_0)$

Factorial

$$T_{factorial}(N) = \begin{cases} c_1 & \textbf{if } N \leq 1 \\ c_2 + T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Goal: Show that $T_{factorial}(N) \leq c \cdot N$ for some $c > 0$ $(N > N_0)$

$$T_{factorial}(1) \stackrel{?}{\leq} 1 \cdot c$$

$$c_1 \stackrel{?}{\leq} 1 \cdot c \checkmark$$

Factorial

$$T_{factorial}(N) = \begin{cases} c_1 & \textbf{if } N \leq 1 \\ c_2 + T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Goal: Show that $T_{factorial}(N) \leq c \cdot N$ for some $c > 0$ $(N > N_0)$

$$T_{factorial}(2) \stackrel{?}{\leq} 2 \cdot c$$

$$c_2 + T_{factorial}(1) \stackrel{?}{\leq} 2 \cdot c$$

$$c_2 + c_1 \stackrel{?}{\leq} 2 \cdot c \checkmark$$

Factorial

$$T_{factorial}(N) = \begin{cases} c_1 & \textbf{if } N \leq 1 \\ c_2 + T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Goal: Show that $T_{factorial}(N) \leq c \cdot N$ for some $c > 0$ ($N > N_0$)

$$T_{factorial}(2) \overset{?}{\leq} 2 \cdot c$$

$$c_2 + T_{factorial}(1) \overset{?}{\leq} 2 \cdot c$$

$$c_2 + T_{factorial}(1) \overset{?}{\leq} c + c$$

$$c_2 \overset{?}{\leq} c \checkmark$$

Factorial

$$T_{factorial}(N) = \begin{cases} c_1 & \textbf{if } N \leq 1 \\ c_2 + T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Goal: Show that $T_{factorial}(N) \leq c \cdot N$ for some $c > 0$ $(N > N_0)$

$$T_{factorial}(3) \stackrel{?}{\leq} 3 \cdot c$$

$$c_2 + T_{factorial}(2) \stackrel{?}{\leq} 3 \cdot c$$

$$c_2 + T_{factorial}(2) \stackrel{?}{\leq} c + 2c$$

$$c_2 \stackrel{?}{\leq} c \checkmark$$

Factorial

This is boring!

Can't I automate the proof?

Induction

1. Generate a hypothesis. $T(N) \in O(N)$
2. Prove the hypothesis for the base case. $\exists c : T(N) \leq c \cdot N$
3. Prove the hypothesis inductively. $\ldots$
   - Assume you've proved it for case $N - 1$
   - Prove that it holds for case N

Factorial

$$T_{factorial}(N) = \begin{cases} c_1 & \textbf{if } N \leq 1 \\ c_2 + T_{factorial}(N-1) & \textbf{otherwise} \end{cases}$$

Goal: Show that $T_{factorial}(N) \leq c \cdot N$ for some $c > 0$ $(N > N_0)$

Assume: $T_{factorial}(N-1) \leq c \cdot N - 1$

$$T_{factorial}(N) \stackrel{?}{\leq} N \cdot c$$
$$c_2 + T_{factorial}(N-1) \stackrel{?}{\leq} N \cdot c$$
$$c_2 + T_{factorial}(N-1) \stackrel{?}{\leq} c + (N-1)c$$
$$c_2 \stackrel{?}{\leq} c \checkmark$$

Induction

We showed there exists a c such that...

- $T(1) \leq 1 \cdot c$ (Base Case)
- if $T(N-1) \leq (N-1) \cdot c$ then $T(N) \leq N \cdot c$ (Inductive Proof)

So...

1. $T(1) \leq 1 \cdot c$ (Base Case)
2. $T(2) \leq 2 \cdot c$ (#1 + Inductive Proof)
3. $T(3) \leq 3 \cdot c$ (#2 + Inductive Proof)
4. $T(4) \leq 4 \cdot c$ (#3 + Inductive Proof)
5. $T(5) \leq 5 \cdot c$ (#4 + Inductive Proof)
6. $T(6) \leq 6 \cdot c$ (#5 + Inductive Proof)
7. ...

The proof holds for any $N \geq 1 \rightarrow T(N) \in O(N)$ ✓

Factorial

What is the complexity of Factorial?

```
public long factorial(long N)
{
  if(N <= 1){ return 1; }
  else { return N * factorial(N-1); }
}
```

Answer: $O(N)$[1]

How much memory does it use?

[1] Technically it's $\theta(N)$, but we haven't proven $T(N) \in \Omega(N)$

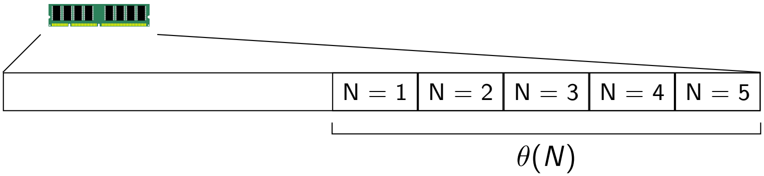
Stack Frames

Every time you call a function, it allocates some memory for local variables (e.g., `N`).

This chunk of memory is called a **Stack Frame**.

This is where the term `StackOverflowError` comes from.

Stack Frames



Factorial (as a loop)

```
public long factorial(long N)
{
  long total = 1;
  for(long i = N; i > 0; i--)
  {
    total *= i
  }
  return total
}
```

Factorial

Why does this work?

Factorial (as a loop)

```
public long factorial(long N)
{
  if(N <= 1){ return 1; }
  else { return N * factorial(N-1); }
}
```

Each call to factorial only makes **one** recursive call.

Factorial (as a loop)

- Is $\texttt{N} > 1$? $\longleftarrow$ Requires stack frame
- Compute $\texttt{arg} = \texttt{N} - 1$ $\longleftarrow$ Requires stack frame
- Call `factorial(arg)`
- Compute $\texttt{N} \times$ result $\longleftarrow$ Requires stack frame
- Return

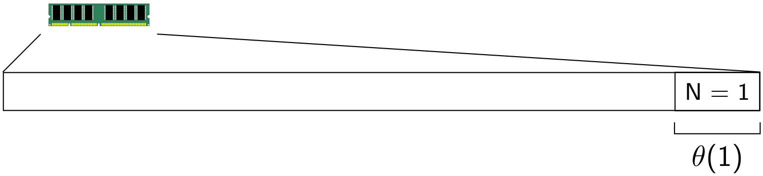
Factorial (as a loop)

```
public long factorial(long N, long total)
{
  if(N <= 1){ return total; }
  else { return factorial(N-1, N * total); }
}
```

Factorial (as a loop)

- Is $\mathtt{N} > 1$? ← Requires stack frame
- Compute $\mathtt{arg1} = \mathtt{N} - 1$ ← Requires stack frame
- Compute $\mathtt{arg2} = \mathtt{N} \times$ total ← Requires stack frame
- Call `factorial(arg1, arg2)`
- Return ← Stack frame unnecessary

Stack Frames



Tail Recursion

If the recursive call is the *last* operation before the return, most languages optimize the recursion away[2].

This is called **Tail Recursion**

[2]... but not Java

Fibonacci

Time permitting...

Fibonacci

What's the complexity:

```
public long fib(long N)
{
  if(n <= 1){ return 1; }
  else { fib(n-1) + fib(n-2) }
}
```