Induction

Instructors: Sam McCauley and Dan Barowy

March 4, 2022

Admin

- Any questions?

Motivating Recursion (Induction Intro)

Where to start?

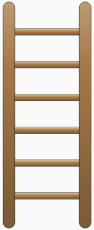
```
public static int numX(String s) {
   if(s.length() == 0) {
      return 0;
   }
   if(s.charAt(s.lengt()-1) == 'X') {
      return 1 + numX(s.substring(0,s.length() - 1));
   else {
      return numX(s.substring(0,s.length() - 1));
   }
}
```

- If `s` has length 0, then this algorithm correctly returns 0.
- What if `s` has length 1?

Our strategy

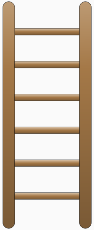
- Start with base case
- Slowly argue that it works for larger and larger strings

Visualizing our Strategy



- How can we climb to the top of a ladder?
- Here’s a two step process:
  - Figure out how to get on some rung of the ladder
  - Figure out a method to get from one rung to the next rung
- I always make it to the top of the ladder?

Our strategy



Ladder analogy: each step of the ladder is the length of the string in our recursive method. We want to show that our method is correct for a string of a certain length.

- Start with base case *Figure out how to get on the ladder*
- Slowly argue that it works for larger and larger strings *From one rung of the ladder, how can we get to the next rung?*

Induction

Induction

- Formalizes this idea
- Incredibly powerful and widely-used proof technique, especially in computer science and discrete math
- Two motivations:
  - *Prove* that your approach works (and that mathematical equations hold)
  - Analyze your code inductively: tracking down what happens when code behaves unexpectedly.

Induction: Classic Example

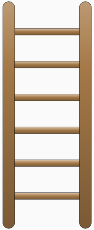
- Let's prove that for all $n \geq 1$:

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}.$$

- Also can be written:

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

Induction



- Sometimes say that we are proving a *collection* of statements
- I could say “numX is correct”.
  - Or I could say “numX is correct for strings of length 1, and strings of length 2, and strings of length 3, . . . ”
- I could say that $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$.
  - Or I could say that $1 = 1(2)/2$, and $1 + 2 = 2(3)/2$, and $1 + 2 + 3 = 3(4)/2 \ldots$
- Climbing the ladder is like proving these statements one at a time. Taken together, I’ve proven the full statement.

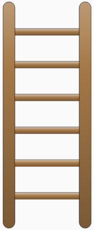
Induction Recipe

Any inductive proof needs (let's say we're doing induction on n):

- A **Base Case**: need to show that the proof is correct for some value
- An **Inductive Hypothesis**: write the assumption you are making for a *given* n
- An **Inductive Step:** Prove that if you assume the inductive hypothesis for n, you can prove it for $n + 1$.

You should **always write all three steps explicitly** when doing an induction in this class.

Induction Recipe (End of ladder analogy)



Any inductive proof needs (let's say we're doing induction on n):

- A **Base Case**: need to show that the proof is correct for some value get on ladder
- An **Inductive Hypothesis**: write the assumption you are making for a *given* n make it clear what the "rungs" of the ladder are
- An **Inductive Step:** Prove that if you assume the inductive hypothesis for n, you can prove it for $n + 1$. get from one rung to the next

Induction: Classic Example

- Let's prove that for all $n \geq 1$:

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}.$$

- Base case?
  - Base case $n = 1$. If $n = 1$, $1 = 1(2)/2$; works!
- Inductive hypothesis: for some n, we have

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}.$$

Inductive Step

Goal: let's assume the inductive hypothesis for n. Can we use it to show the inductive hypothesis for $n+1$?

- Assume that

$$1+2+3+\ldots+n=\frac{n(n+1)}{2}.$$

- Then

$$\begin{aligned} 1+2+3+\ldots+n+(n+1) &= \frac{n(n+1)}{2}+(n+1) \\ &= \frac{n^2+n+2n+2}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

- But this is the inductive hypothesis for $n+1$! So we are done.

What have we done?

- We have shown that for all n,

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}.$$

- Questions?

On the board

- For all n,

$$1 + 2 + 4 + \ldots + 2^n = 2^{n+1} - 1$$

- Can also be written

$$\sum_{i=0}^{n} 2^n = 2^{n+1} - 1$$

- Remember: base case, inductive hypothesis, inductive step

Proving statements about algorithms

(We'll be using this a lot on Monday, and with more interesting examples)

- How do we know `contains()` works on a `Vector`?
- Base case: `contains()` returns true if the element is in position `0` in the vector; otherwise it moves past the `0`th element without returning
- Inductive hypothesis: if the element is in the first i positions of the vector then `contains()` returns true; otherwise it moves past the first i elements without returning
- Inductive step: assume the inductive hypothesis for i. If the element is in position $\leq i$, then `contains` returns true by the inductive hypothesis. If the element is in position $i+1$, then `contains` does not return while looking at the first i elements by the inductive hypothesis. It examines the $i+1$st element, and returns true if it is found; otherwise, `contains` moves to the next element.

Comparing performance of extending a `Vector`

- Let's say we insert n items into a `Vector` one at a time; each time we use `add()`. Let's assume that the `Vector` starts with size 1.
- First, let's say `ensureCapacity` adds 1 to the capacity of the `Vector` each time. How many operations does this take in total?
- Let's write the total operations.
- It takes $O(i)$ operations to call `ensureCapacity` on a vector of size i.
- Total operations:

$$O(1) + O(2) + \ldots + O(n)$$

- How do we add O? We can just use the definition: we can upper bound $O(i)$ with ci.

Comparing performance of extending a `Vector`

- Let's say we insert n items into a `Vector` one at a time; each time we use `add()`. Let's assume that the `Vector` starts with size 1.
- First, let's say `ensureCapacity` adds 1 to the capacity of the `Vector` each time. How many operations does this take in total?
- Total operations:

$$O(1) + O(2) + \ldots + O(n)$$

- Total operations:

$$c_1 + 2c_1 + 3c_1 + \ldots + c_1 n = c_1 (1 + 2 + 3 + \ldots + n) =$$

$$c_1 \frac{n(n+1)}{2} = c_1 n^2/2 + c_1 n/2 = O(n^2)$$

Comparing performance of extending a `Vector`

- Let's say we insert n items into a `Vector` one at a time; each time we use `add()`. Let's assume that the `Vector` starts with size 1.
- First, let's say `ensureCapacity` adds 1 to the capacity of the `Vector` each time. How many operations does this take in total? Answer: $O(n^2)$
- Then, let's say `ensureCapacity` adds 10 to the capacity of the `Vector` each time. How many operations does this take in total?
- Finally, let's say `ensureCapacity` doubles the size of the `Vector` each time it is called. How many operations does this take in total?

Comparing performance of extending a `Vector`

- Let's say we insert n items into a `Vector` one at a time; each time we use `add()`. Let's assume that the `Vector` starts with size 1.
- Then, let's say `ensureCapacity` adds 10 to the capacity of the `Vector` each time. How many operations does this take in total?
- $$O(10) + O(20) + \ldots + O(n)$$
- Rewrite as:

$$c_2 10 + c_2 20 + \ldots + c_2 n = 10c_2 \cdot 1 + 10c_2 \cdot 2 + \ldots + 10c_2 \cdot n/10 =$$

- Rewrite as:

$$= 10c_2 \frac{\frac{n}{10}\left(\frac{n}{10} + 1\right)}{2} = O(n^2)$$

Comparing performance of extending a `Vector`

- Let's say we insert n items into a `Vector` one at a time; each time we use `add()`. Let's assume that the `Vector` starts with size 1.
- First, let's say `ensureCapacity` adds 1 to the capacity of the `Vector` each time. How many operations does this take in total? Answer: $O(n^2)$
- Then, let's say `ensureCapacity` adds 10 to the capacity of the `Vector` each time. How many operations does this take in total? Answer: $O(n^2)$
- Finally, let's say `ensureCapacity` doubles the size of the `Vector` each time it is called. How many operations does this take in total?

Comparing performance of extending a `Vector`

- Let's say we insert n items into a `Vector` one at a time; each time we use `add()`. Let's assume that the `Vector` starts with size 1.
- Finally, let's say `ensureCapacity` doubles the size of the `Vector` each time it is called. How many operations does this take in total?
- Total operations:

$$O(1) + O(2) + O(4) + \ldots + O(n)$$

- Total operations:

$$c_1 + 2c_1 + 4c_1 + \ldots + c_1 n = c_1 (1 + 2 + 4 + \ldots + n) \leq$$

- Reminder: $1 + 2 + 4 + \ldots + 2^i = 2^{i+1} - 1$
- Substitute $i = \log_2 n$:

$$c_1(2^{i+1} - 1) = 2c_1 n - c_1 = O(n)$$

Comparing performance of extending a `Vector`

- Let's say we insert *n* items into a `Vector` one at a time; each time we use `add()`. Let's assume that the `Vector` starts with size 1.
- First, let's say `ensureCapacity` adds 1 to the capacity of the `Vector` each time. How many operations does this take in total? Answer: $O(n^2)$
- Then, let's say `ensureCapacity` adds 10 to the capacity of the `Vector` each time. How many operations does this take in total?
- Finally, let's say `ensureCapacity` doubles the size of the `Vector` each time it is called. How many operations does this take in total? Answer: $O(n)$

We save a *factor n* in number of operations by doubling vector size every time!

Printing a linked list

Let’s look at two ways to print a singly linked list. First, we call `get()` on each index. Second, we iterate through the nodes of the list.

How long does each take?

Printing a linked list

```
public String toString() {
   String ret = "";
   for(int i = 0; i < count; i++) {
      ret += get(i).toString();
   }
}
```

```
public String toString() {
   Node<E> current = head;
   String ret = "";
   while(head != null) {
      ret += current.value().toString();
      current = current.next();
   }
}
```

Printing a linked list

- First method: calling `get()` takes $O(n)$ time.
- `get()` is called n times
- Total is $O(n^2)$ time.
- Second method: calling `next()` and `toString()` are $O(1)$ time
- Called n times. Total: $O(n)$ time.

Linked List Discussion

Tail pointer for singly linked list?

- Singly linked lists have very slow operations at the end of the list
- What if we maintain a `tail` pointer to the last element of the list? What can we maintain efficiently?
- How about `addLast()`?
- How about `removeLast()`?

Doubly vs Singly Linked List Tradeoffs

- What are some advantages of a doubly linked list?
- What are some advantages of a singly linked list?