Recursion & Induction

CS16: Introduction to Algorithms & Data Structures
Spring 2018
Outline

- Recursion
- Recurrence relations
- Plug & chug
- Induction
- Strong vs. weak induction
“Something defined in terms of itself”
Recursion

- What is a recursive problem?
  - a problem defined in terms of itself
- What is a recursive function?
  - a function defined in terms of itself
  - example: Fibonacci
- **Note**: at each level problem/function/pic gets easier/smaller
- How can we solve recursive problems?
Recursive Algorithms

- Algorithms that call themselves
  - Call themselves on smaller inputs (sub-problems)
  - Combine the results to find solution to larger input
- Recursive algorithms
  - Can be very easy to describe & implement :-)
  - Can be hard to analyze :-(
Factorial

iterative: \( n! = \prod_{i=1}^{n} i = n \times (n - 1) \times \cdots \times 1 \)

recursive: \( n! = n \times (n - 1)! \)
Recursive Factorial — Simulation

```python
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

- call `factorial(3)`
```
Recursive Factorial — Simulation

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

- call `factorial(3)`
  - level #1: 3! = 1 so 3 * `factorial(2)`
Recursive Factorial — Simulation

```python
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- **call** `factorial(3)`
  - **level #1**: $3! = 1$ so $3 \times \textbf{factorial}(2)$
    - **level #2**: $2! = 1$ so $2 \times \textbf{factorial}(1)$
Recursive Factorial — Simulation

```python
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call `factorial(3)`
  - **level #1**: $3! = 1$ so $3 \times \text{factorial}(2)$
    - **level #2**: $2! = 1$ so $2 \times \text{factorial}(1)$
      - **level #3**: $1! = 1$ so return $1$
Recursive Factorial — Simulation

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

call factorial(3)

  level #1: 3! = 1 so 3 \times \textbf{factorial}(2)

    level #2: 2! = 1 so 2 \times 1

      level #3: 1! = 1 so return 1
Recursive Factorial — Simulation

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

- call factorial(3)
  - level #1: 3! = 1 so 3 x 2
    - level #2: 2! = 1 so 2 x 1
      - level #3: 1 == 1 so return 1
Recursive Factorial — Simulation

```python
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial(3) = 6**
  - **fact(3):** $3! = 1$ so $3 \times 2$
    - **level #2:** $2! = 1$ so $2 \times 1$
      - **level #3:** $1 == 1$ so return **1**
Example: recursive `array_max`

```python
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

Activity #1
Example: recursive `array_max`

```python
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

Activity #1

2 min
Example: recursive `array_max`

```python
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

Activity #1
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
Example: recursive `array_max`

```python
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

`array_max([5,1,9,2], 4) = max(2, array_max([5,1,9], 3))`

= `max(2, max(9, array_max([5,1], 2)))`

= `max(2, max(9, max(1, array_max([5], 1))))`

= `max(2, max(9, max(1, 5)))`

= `max(2, max(9, 5))`

= `max(2, 9)`

= 9

*Note*: we keep entire array but only show relevant items
Running Time of Recursive Algos

- Difficult to analyze :-(
- With iterative algorithms
  - we can count # of ops per loop
- How can we count # ops in a recursive step?
  - We can’t…
Recurrence Relations

- Functions that express run time recursively
  - part 1: # of operations in base case
  - part 2: # of operations in general case

\[
T(n) = 2 \cdot T(n - 1) + 10, \quad \text{with} \quad T(1) = 8
\]

- general case
- base case
Example: recursive `array_max`

```python
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

\[ T(n) = T(n-1) + c_1, \quad \text{with} \quad T(1) = c_0 \]

- general: constant \# ops for comp & max + cost of recursive
- base: constant \# ops for comp and return

**What about Big-Oh?**
Big-Oh from Recurrence Relation

- Step #1: Plug & Chug
  - algebraic manipulations to guess a Big-Oh expression
- Step #2: Induction
  - prove that Big-Oh expression is correct
Example: recursive \texttt{array\_max}

\[
T(n) = T(n - 1) + c_1, \quad \text{with} \quad T(1) = c_0
\]

- general case
- base case

Activity #2
Example: recursive `array_max`

\[ T(n) = T(n - 1) + c_1, \quad \text{with} \quad T(1) = c_0 \]

- **general case**
- **base case**

Activity #2

3 min
Example: recursive `array_max`

\[ T(n) = T(n - 1) + c_1, \quad \text{with} \quad T(1) = c_0 \]

*general case*

*base case*

Activity #2
Example: recursive `array_max`

\[ T(n) = T(n - 1) + c_1, \quad \text{with} \quad T(1) = c_0 \]

- general case
- base case

Activity #2
Example: recursive array_max

\[ T(n) = T(n-1) + c_1, \quad \text{with} \quad T(1) = c_0 \]

- general case
- base case

Activity #2
Plug & Chug

\[ T(1) = c_0 \]
\[ T(2) = c_1 + T(1) = c_1 + c_0 \]
\[ T(3) = c_1 + T(2) = c_1 + c_1 + c_0 = 2c_1 + c_0 \]
\[ T(4) = c_1 + T(3) = c_1 + 2c_1 + c_0 = 3c_1 + c_0 \]
\[ T(5) = c_1 + T(4) = c_1 + 3c_1 + c_0 = 4c_1 + c_0 \]

\[ \vdots \]
\[ T(n) = c_1 + T(n-1) = (n-1)c_1 + c_0 \]

- Recurrence solution: closed form expression

\[ T(n) = (n - 1) \cdot c_1 + c_0 = O(n) \]
Are we done?

- That was just a guess...not a proof!
  - plugged & chugged to find a pattern
  - and then guessed
- How can we be sure?
- We prove it using Induction
Induction

- Proof technique to prove statements about well-ordered sets
  - well-ordered: order between elements
  - example: the integers, recurrence relations

- Idea
  - prove if statement true for some case, statement true for next case
  - prove statement for base case

- Example for integers
  - prove statement for \( n = 1 \)
  - prove that if statement is true for \( n = k \) then true for \( n = k+1 \)
Induction

Inductive step:

Base case:
Induction for **array_max**

- The solution of \( T(n) = T(n - 1) + c_1, T(1) = c_0 \) is
  \[
  (n - 1) \cdot c_1 + c_0
  \]

- Base case: \( n=1 \)
  - \( T(1) = c_0 \)
  - \( (1 - 1)c_1 + c_0 = c_0 \)

- Inductive assump: \( n=k \)
  \[
  T(k) = (k - 1) \cdot c_1 + c_0
  \]

- Inductive step
  \[
  T(k + 1) = c_1 + T(k) = c_1 + (k - 1) \cdot c_1 + c_0 = k \cdot c_1 + c_0
  \]

Rec. rel.

Simplify
Induction Example #2

\[ A(n) = 2 + 4 + \cdots + 2n = n \cdot (n + 1) \]

- **Base case:** \( n = 1 \)
  - \( 2 \cdot 1 \) and \( 1 \cdot (1 + 1) = 1 \cdot 2 = 2 \)

- **Inductive assumption:** \( n=k \)
  - \( A(k) = 2 + 4 + \cdots + 2k = k \cdot (k + 1) \)

- **Inductive step** \( A(k+1) = 2 + 4 + \cdots + 2k + 2 \cdot (k + 1) \)
  - \[ = k \cdot (k + 1) + 2 \cdot (k + 1) \]
  - **factor out** \( (k + 1) \)
  - \[ = (k + 1) \cdot (k + 2) \]
Induction Example #3

\[ P(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \]

Activity #3
Induction Example #3

\[ P(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \]

Activity #3

4 min
Another Induction Example

\[ P(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \]

Activity #3

3 min
Another Induction Example

\[ P(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \]
Another Induction Example

\[ P(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \]

Activity #3

1 min
Another Induction Example

\[ P(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \]
Another Induction Example

\[ P(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n + 1)}{2} \]

- Prove base case: \( n=1 \)
  \[ \sum_{i=1}^{1} i = 1 \quad \text{and} \quad \frac{1 \cdot (1 + 1)}{2} = 1 \]

- Induction assumption: \( n=k \)
  \[ \sum_{i=1}^{k} i = \frac{k \cdot (k + 1)}{2} \]

- Prove induction step!
Another Induction Example

- Prove induction step

\[
\sum_{i=1}^{k+1} i = 1 + 2 + \cdots + k + (k + 1)
\]

\[
= \sum_{i=1}^{k} i + (k + 1)
\]

\[
= \frac{k \cdot (k + 1)}{2} + (k + 1)
\]

\[
= \frac{(k + 1) \cdot (k + 2)}{2}
\]

**Induction assumption**

\[
\sum_{i=1}^{k} i = \frac{k \cdot (k + 1)}{2}
\]

- factor out \((k + 1)\)
Strong vs. Weak Induction

- Weak induction
  - induction step assumes true for \( n=k \) and
  - proves true for \( n=k+1 \)

- Strong induction
  - induction step assumes true for \( n=1, 2, \ldots, k \) and
  - proves true for \( n=k+1 \)

- Strong vs. weak refers to assumption
  - not strength of proof
Strong vs. Weak Induction

Weak:

Strong:
Readings

- Induction handout on course page
Announcements

› Sections have started!

› Clinic is starting this week! Wednesdays 8-10pm in CIT 219

› Let us know if:
  › You don’t receive graded Homework 1 via email by Thursday

› **Homework 2** due **Friday 5:00pm**

› **Seamcarve** due **Monday 11:59pm**

› Thursday is **Python Lab #2**
  › Same room you went to last week