Recursion, Induction, Dynamic Programming

CS16: Introduction to Algorithms & Data Structures
Summer 2021
Outline

- Recursion
- Recurrence relations
- Plug & chug
- Induction
- Strong vs. weak induction
Collaboration policy

- **Can and should** discuss assignments with other students!
- Still **cannot** share code or written solutions
The Scouting Problem
recursive: 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: Factorial, Fibonacci
- At each level, the problem gets easier/smaller
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 :-)
    - Especially for recursively-defined data structures (e.g. trees)
  - Can be hard to think about and to analyze :-(

Factorial

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

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

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

# call factorial(3)
```

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

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

- call `factorial(3)`
  - level #1: $3 \neq 1$ so $3 \times \text{factorial}(2)$
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

- call \texttt{factorial}(3)
  - level #1: \(3 \neq 1\) so \(3 \times \texttt{factorial}(2)\)
    - level #2: \(2 \neq 1\) so \(2 \times \texttt{factorial}(1)\)
Recursive Factorial — Simulation

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

call \textbf{factorial}(3)

- \textbf{level \#1: } 3 \neq 1 \text{ so } 3 \times \textbf{factorial}(2)
  - \textbf{level \#2: } 2 \neq 1 \text{ so } 2 \times \textbf{factorial}(1)
    - \textbf{level \#3: } 1 == 1 \text{ so return } 1
def \texttt{factorial}(n):\n  \textbf{if} n == 1:\n    \textbf{return} 1
  \textbf{else}:\n    \textbf{return} n * \texttt{factorial}(n-1)

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

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

- call **factorial(3) = 6**
  - fact(3): 3 ≠ 1 so 3 × 2
    - level #2: 2 ≠ 1 so 2 × 1
      - level #3: 1 == 1 so return 1
Wait a minute!!

you keep *calling* factorial but never actually *implemented* it
Recursive Factorial — Simulation

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
Recursive Factorial — Simulation

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(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))
```
Example: recursive \texttt{array\_max}

\begin{verbatim}
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
\end{verbatim}

\[
array\_max([5,1,9,2], 4) = [9]
\]

\[
max(2, array\_max([5,1,9,2], 3) = [9])
\]

\[
max(9, array\_max([5,1,9,2], 2) = [5])
\]

\[
max(1, array\_max([5,1,9,2], 1) = [5])
\]

21
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…

```python
def factorial(n):
    out = 1
    for i in range(1, n+1):
        out = i * out
    return out

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

- Functions that express run time recursively

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

- part 1: # of operations in general case
- part 2: # of operations in 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 call
- base: constant # ops for comp and return

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

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

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

- **general case**
- **base case**
Plug & Chug

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

**general case**

**base case**

\[
\begin{align*}
T(1) &= c_0 \\
T(2) &= c_1 + T(1) = c_1 + c_0 \\
T(3) &= c_1 + T(2) = c_1 + c_1 + c_0 = 2c_1 + c_0 \\
T(4) &= c_1 + T(3) = c_1 + 2c_1 + c_0 = 3c_1 + c_0 \\
T(5) &= c_1 + T(4) = c_1 + 3c_1 + c_0 = 4c_1 + c_0 \\
&\vdots \\
T(n) &= c_1 + T(n-1) = \ldots = \ldots = (n-1)c_1 + c_0
\end{align*}
\]

- **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 we guessed at a Big-O

- How can we be sure?
- We prove it using Induction
Induction

- Proof technique to prove statements about infinite sets of natural numbers

- Can also be used for recursively-defined structures like trees

- To prove that a statement $P$ is true for all positive numbers $n=1, 2, 3, 4, \ldots$
  
  - prove that a statement $P$ is true for $n=1$
  
  - prove that if $P$ is true for $n=k$ then $P$ is true for $n=k+1$
Steps to an Inductive Proof

- Base case
  - prove that statement $P$ is true for base case
- Inductive hypothesis
  - assume that $P$ is true for some case $n = k$
- Inductive step
  - prove that if $P$ is true for $n = k$ then $P$ is true for $n = k+1$
- Conclusion
  - Then $P$ must be true for all $n$
Induction

Inductive step:

Base case:
Induction for `array_max`

- **P(n):** \( T(n) = T(n - 1) + c_1 \), w/ \( T(1) = c_0 \) is equal to
  \[
  f(n) = (n - 1) \cdot c_1 + c_0
  \]

- Prove for base case: \( n=1 \)
  - \( T(1) = c_0 \) and \( f(1) = (1 - 1) \cdot c_1 + c_0 = c_0 \)

- Inductive assumption: \( n=k \)
  - assume \( T(k) = f(k) \)

- Inductive step: \( T(k + 1) = T(k) + c_1 \)
  \[
  = (k - 1) \cdot c_1 + c_0 + c_1 \\
  = k \cdot c_1 + c_0 = f(k + 1)
  \]
Induction Example #2

\( P(n) : A(n) = \sum_{i=1}^{n} 2i \) is equal to \( f(n) = n \cdot (n + 1) \)

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

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

- **Inductive step**
  \[
  A(k+1) = \sum_{i=1}^{k+1} 2i \\
  = \sum_{i=1}^{k} 2i + 2 \cdot (k + 1) \\
  = k \cdot (k + 1) + 2 \cdot (k + 1) \\
  = (k + 1) \cdot (k + 2) \\
  = f(k + 1)
  \]
Another Induction Example

\[ P(n): \quad A(n) = \sum_{i=1}^{n} i \quad \text{is equal to} \quad f(n) = \frac{n \cdot (n + 1)}{2} \]

- Prove base case: \( n=1 \)
  - \( A(1) = 1 \) and \( f(1) = \frac{1 \cdot (1 + 1)}{2} = 1 \)
- Induction assumption: \( n=k \)
  - \( A(k) = f(k) \) which means \( \sum_{i=1}^{k} i = \frac{k \cdot (k + 1)}{2} \)
- Prove induction step!
Another Induction Example

- Prove induction step

$$A(k + 1) = \sum_{i=1}^{k+1} i$$

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

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

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

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

$$= f(k + 1)$$
Strong vs. Weak Induction

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

- Strong induction
  - induction step assumes statement is 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:
Dynamic programming
Factorial, again

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

- $T(1) = c_0$
- $T(n) = c_1 + T(n-1)$
- What’s the big-O runtime? $O(n)$
Fibonacci

- Defined recursively
  - $F_0 = 0$, $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
Fibonacci (Recursive)

function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)

- $T(0) = c_0$
- $T(1) = c_1$
- $T(n) = c_2 + T(n-1) + T(n-2)$
- What’s the big-O runtime?
Fibonacci (Recursive)

```
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
```

- How many times does `fib` get called for `fib(4)`?
  - 8 times
- At each level it makes twice as many recursive calls as the level below.
- For `fib(n)` it makes approximately `2^n` recursive calls.
- Algorithm is $O(2^n)$

On my computer, computing the 60th Fibonacci number takes ~2 days.
Computing 60! is ~instantaneous.
Dynamic programming to the rescue!
What is Dynamic Programming?

- Algorithm design paradigm/framework
  - Design efficient algorithms for optimization problems
- Optimization problems
  - “find the best solution to problem X”
  - “what is the shortest path between u and v in G”
  - “what is the minimum spanning tree in G”
- Can also be used for non-optimization problems
When is Dynamic Programming Applicable?

- Condition #1: sub-problems
  - The problem can be solved recursively
  - Can be solved by solving sub-problems
- Condition #2: overlapping sub-problems
  - Same sub-problems need to be solved many times
- Core idea
  - Solve each sub-problem once and store the solution
  - Use stored solution when you need to solve sub-problem again
Steps to Solving a Problem w/ DP

- What are the sub-problems?
- What is the “magic” step?
  - Given solutions to sub-problems…
  - …how do I combine them to get solution to the problem?
- In which order should I solve sub-problems?
  - so that solutions to sub-problems are available when I need them
- Design iterative algorithm
  - that solves sub-problems in right order and stores their solution
Fibonacci (Dynamic Programming)
Fibonacci (Dynamic Programming)

- Given \( n \) compute
  - \( \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \)
- with base cases \( \text{Fib}(0) = 0 \) and \( \text{Fib}(1) = 1 \)

- What are the **sub-problems**?
  - \( \text{Fib}(n-1), \text{Fib}(n-2), \ldots, \text{Fib}(1), \text{Fib}(0) \)

- What is the **magic** step?
  - \( \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) \)

*Magic step is usually not provided!*
Fibonacci (Dynamic Programming)

- In which order should I solve sub-problems?
  - Fib(0), Fib(1), …, Fib(n-1), Fib(n)
Fibonacci (Dynamic Programming)

- Design iterative **algorithm**

```python
function Fib(n):
    fibs = []
    fibs[0] = 0
    fibs[1] = 1
    for i from 2 to n:
        fibs[i] = fibs[i-1] + fibs[i-2]
    return fibs[n]
```
Fibonacci (Dynamic Programming)

- What’s the runtime of `dynamicFib()`?
  - Calculates Fibonacci numbers from 0 to `n`
  - Performs $O(1)$ ops for each one
  - Runtime is $O(n)$

- We reduced runtime of algorithm
  - From exponential to linear
  - with dynamic programming!
Seams
Finding Low Importance Seams

- **Idea:** remove *seams* not columns
  - (vertical) seam is a path from top to bottom
  - that moves left or right by at most one pixel per row
Finding Low Importance Seams

- How many seams in a $c \times r$ image?
  - At each row the seam can go Left, Right or Down
  - It chooses 1 out of 3 dirs at all but last row $r$
  - So about $3^{r-1}$ seams from some starting pixel
  - There are $c$ starting pixels so total number of seams is
    - about $c \times 3^{r-1}$

- For square $n \times n$ image
  - there are about $n3^{n-1}$ possible seams
Finding Low Importance Seams

- Brute force algorithm
  - Try every possible seam & find least important one
- What is running time of brute force algorithm?
  - If image is $n \times n$ brute force takes about $n^3n^{-1}$
  - So brute force is $\Omega(2^n)$ (i.e., exponential)
Seamcarve

- What is the runtime of Seamcarve?
- The algorithm
  - Iterate over all pixels from bottom to top
  - Populate costs and dirs arrays
  - Create seam by choosing minimum value in top row and tracing downward
- How many operations per pixel?
  - A constant number of operations per pixel (4)
- Constant number of operations per pixel means algorithm is linear
  - $O(n)$ where $n$ is number of pixels
Seamcarve

- How can we possibly go from
  - exponential running time with brute force
  - to linear running time with Seamcarve?
- What is the secret to this magic trick?

Dynamic Programming!
Designing Seamcarve

- What are the subproblems?
  - lowest cost seam (LCS) starting at is

\[
\min( \text{LCS}(\text{ }\text{ }), \text{LCS}(\text{ }\text{ }), \text{LCS}(\text{ }\text{ }))
\]

- Are they overlapping?
  - Yes!
  - ex: LCS( ) is subproblem of LCS( ) and LCS( )
Designing Seamcarve

- What is the magic step?
  - \[ \min(\text{LCS}(), \text{LCS}(), \text{LCS}()) \]

- Which order should I use?
  - to solve LCS problem at cell \((i,j)\)
  - we need to have solved problem at cells below
Designing Seamcarve

- Algorithm
  - compute cost of LCS for each cell going bottom up
  - store cost of LCS in an auxiliary 2D array...
  - ...so we can reuse them

Cost(  ) = Val(  ) + min( Cost(  ), Cost(  ), Cost(  ))
Designing Seamcarve

- Problem
  - Costs array only gives us cost of LCS at cell
  - We need the seam. What happened?
  - We used
    \[
    \text{Cost}(\text{cell}) = \text{Val}(\text{cell}) + \min( \text{Cost}(\text{left}), \text{Cost}(\text{up}), \text{Cost}(\text{diagonal}))
    \]
  - But recall that at “seam level” we had
    \[
    \text{LCS}(\text{cell}) = \begin{array}{c}
    \text{cell} \\
    \min( \text{LCS}(\text{left}), \text{LCS}(\text{up}), \text{LCS}(\text{diagonal}))
    \end{array}
    \]
Designing Seamcarve

- It’s OK!
  - We can keep track of minimum LCS
  - at each step in auxiliary structure Dirs
Readings

- Induction handout on course page