Agenda

1. Icebreaker
2. Mini Assignment/Inductive proofs
3. Dynamic Programming
4. Recursion and recurrence relations
5. Induction
6. Expanding Stacks and Queues
Icebreaker

Go around the room and give your name, pronouns, class year, and if you were a sea creature, which would you be?
What are the steps of an Inductive Proof?

- Base Case
- Inductive Hypothesis
- Inductive Step
- Conclusion
Mini Assignment: Induction Question 2

Use mathematical induction to prove that $1 + 2 + 3 + \ldots + n = \frac{n(n + 1)}{2}$ for all positive integers $n$.

**Solution:**

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

**Base Case:** Show $p(1)$ is true.

Left Side = 1
Right Side = $1(1 + 1)/2 = 1$
Both sides of the statement are equal hence $p(1)$ is true.

**Inductive Hypothesis:** We now assume that $p(k)$ is true

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

**Inductive Step:** and show that $p(k + 1)$ is true by adding $k + 1$ to both sides of the above statement

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

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

The last statement may be written as

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

Which is the statement $p(k + 1)$. 
Mini Assignment: Recurrence Relations

Give recurrence relations for the following problems:

a) The Fibonacci Sequence

```python
fib(number):
    if number is <= 2:
        return 1
    else:
        return fib(number-1) + fib(number-2)
```

b) Factorial

```python
factorial(number):
    if number is 1:
        return number
    else:
        return number * factorial(number - 1)
```

a) \( a_n = 1 + a_{n-1} + a_{n-2} \)

b) \( a_n = 1 + a_{n-1} \)
Topic Map

- Dynamic Programming
- Recursion and Recurrence Relations
- Induction
- Expanding Stacks/Queues
Dynamic Programming
What is dynamic programming?

- Core idea:
  - solve each sub-problem once and store the solution (usually in a look up table)
  - use stored solution when you need to solve sub-problem again
General Steps

● Don’t be afraid to begin with the greedy solution
● Think of what repetitive calculations can be “saved” so the problem could be solved faster
● Think about how the problem can be broken down into subproblems
   ○ Oftentimes this involves coming up with some kind of equation to relate the current step to earlier ones.
● Design Iterative Solution
Practice Problem 1

Given a cost matrix and a position in the matrix, return the cost of the minimum cost path to reach the position from \((0, 0)\). Each cell of the matrix has the cost of that cell. You can only traverse down, right and diagonally lower cells from a given cell.

What are the sub-problems? Discuss different approaches.
Practice Problem 1

General logic for minimum cost path:
1. Create a total array which is the same size as the cost array
2. Base case: Initialize total[0][0] = cost[0][0]
3. General case:
   total[row][col] = min(total[row-1][col-1], total[row-1][col], total[row][col-1]) + cost[row][col]
Practice Problem 2

Convert some amount of money $M$ into a given list of denominations (decreasing order), using the smallest possible number of coins. Return the smallest number of coins (not the denominations used).

*Group discussion: Greedy Solution*
greedy_change(amt, denoms):
    remainder = amt // remaining amount to make change for
    pieces = [ ] // output array of denominations (number of each denomination)
    for k = 0 to denom.length: // for each denomination (starting with largest)
        pieces[k] = remainder/denoms[k]
        remainder = remainder % denoms[k] // practice mod!
    return sum(pieces)
However...

Let’s say the coin denominations were 4, 3, 1. If you were trying to make 6, the algorithm would produce one 4 piece, and 2 one pieces, where in reality you would want 2 three pieces. How can we change our solution to solve this issue?
Convert some amount of money $M$ into a given a list of denominations (decreasing order), using the smallest possible number of coins. Return the smallest number of coins (not the denominations used).

*Hand simulate! Come up with a strategy, then write pseudocode if you have time.*
def get_best_num_coins(amt, denoms):
    # get_best_num_coins: int, list[int] → int
    # purpose:
    # consumes:
    # produces:

    best_num_coins = []
    # number of coins needed for subproblem for amount 0
    best_num_coins[0] = 0
    # for each subproblem (each amount to make change for)
    for curr_amt = 1 to amt:  # O(amt)
        # min stores the current best min no. coins to make ‘curr_amt’ amount
        min_num_coins = infinity
        # for each denomination
        for coin_val in denoms:  # O(denoms.length)
            if curr_amt >= coin_val:
                # if 1 + subproblem is better than current best
                if best_num_coins[curr_amt - coin_val] + 1 <
                    min_num_coins:
                    # update the min_num_coins tracker
                    min_num_coins = best_num_coins[curr_amt - coin_val] + 1
                # store min no. of coins to make ‘curr_amt’ in the array
                best_num_coins[curr_amt] = min_num_coins
    return best_num_coins[amt]
Recursion and Recurrence Relations
What is recursion?

- a problem defined in terms of itself
- examples of functions that can be implemented recursively: Fibonacci, Factorial
- can be seen in functions/algorithms that call themselves
  - ex. \( F(n) = F(n-1) + F(n-2) \)
What is a recurrence relation?

- Functions that express runtime recursively
- For example, the runtime for $n = 2$ depends on the runtime of $n = 1$
- Two parts: general case, base case
Practice problem (as a section): Towers of Hanoi
Practice problem (as a section):
Towers of Hanoi

RECURRENTCE

The game of Hanoi Tower is to play with a set of disks of graduated size with holes in their centers and a playing board having three spokes for holding the disks. The object of the game is to transfer all the disks from spoke A to spoke C by moving one disk at a time without placing a larger disk on top of a smaller one. The minimal number of moves required to solve the problem with $n$ disks can be modeled by the following recurrence relation:

$$a_n = 2a_{n-1} + 1, \quad n \geq 1$$

$$a_1 = 1$$
a.k.a “closed form expression” - an expression that does not rely on recurrence
Induction
What is induction?

- proof technique useful for recurrence relations, problems that ask you to prove “for all n > 1”, etc.
- **two main parts:**
  - prove claim is true for base case
  - prove that if claim is true for case n, it is true for case n+1
- **four steps:** base case, inductive hypothesis, inductive step, conclusion
Claim: The runtime of Towers of Hanoi can be written as
\[ a_n = 2a_{n-1} + 1, \quad n \geq 1 \]

Prove that its Big-O runtime is \( 2^n - 1 \).
Expanding Stacks / Queues
Intro to Stacks and Queues

- **stack**: LIFO (last-in-first-out)
  - push/pop
- **queue**: FIFO (first-in-first-out)
  - enqueue/dequeue
- both *abstract data types* - must be implemented by you
- how to make expandable stacks/queues that can hold indefinite number of elements
Expanding Stack

- array and pointer that keeps track of last element pushed

- two strategies
  - incremental - increase array size by constant $e$ when full
  - doubling - double array size when full

- both have runtime of $O(1)$ when not expanding, $O(n)$ when expanding
- however, amortized runtime differs
  - cost of sequence *per call* vs worst case runtime which is total

$O(n)$ for incremental, $O(1)$ for doubling
Expanding Queue

- array, a pointer to head, and a pointer to tail
- when tail is at end of array, array is not necessarily full
  - tail “wraps around” front until it reaches head
  - keep this in mind when expanding!
    - array[0] is not necessarily always the head