CS 209
Data Structures and Mathematical Foundations

03 / 29 / 2024

Instructor: Michael Eckmann

Today's Topics

- Questions?/Comments?
- Finish the Divide and Conquer (D&C) applied to MaxCSS
- Apply Divide and Conquer to Towers of Hanoi
- Stacks and Queues

Divide and Conquer

- The divide and conquer technique is a way of
  - converting a problem into smaller problems that can be solved individually and then
  - combining the answers to these subproblems in some way to solve the larger problem
- DIVIDE = divide into smaller problems and solve them recursively, except the base case(s)
- CONQUER = compute the solution to the overall problem by using the solutions to the smaller problems solved in the DIVIDE part.

Divide and Conquer for MaxCSS

- Apply divide and conquer to the Maximum contiguous subsequence problem.
- We can divide the seqeuence in half each time, like MergeSort does.
- Don't divide when subsequence is length 1. This is base case and the answer is simply the value of the element or 0 if it is negative.
- We will get an answer for each half.
- The answer to the larger problem (the sequence comprising the two halves) is either
  - The answer to the left half
  - The answer to the right half
  - Or the max that spans the two halves

Divide and Conquer for MaxCSS

- The overall result can be either
  - the max on the left side OR
  - the max on the right side OR
  - the max that spans both sides.

Divide and Conquer for MaxCSS

- Maximum sum of a contiguous subsequence of
  - seq[left .. right]

- Conquer part:
  - compute the maxLeftBorderSum
  - compute the maxRightBorderSum
  - decide which is larger
    - maxLeft or
    - maxRight or
    - maxLeftBorderSum + maxRightBorderSum

Towers of Hanoi

- Three pegs
- Start with different size discs on first peg
- Move them all to the third peg using the following rules
  - Move only one disc at a time
  - Can't place a larger disc on a smaller disc
  - All discs must be on some peg except for the disc in transit between pegs
- Let's see a web implementation.

Towers of Hanoi

- Do a solution with 3 discs.
- Do a solution with 4 discs.
- See any patterns emerging?
- Is it recursive?

Towers of Hanoi

- To move N discs from first peg to third
  - Move topmost N-1 discs from first peg to second
  - Move the largest disc from the first peg to third
  - Move the N-1 discs from the second peg to the third

Towers of Hanoi

- During the solution, each peg takes on a role
  - Either the start peg, end peg, or intermediate peg.
  - Notice:
    - In the first and third parts, notice that they are the same as the whole, with the roles of the pegs changing.
    - First part:
      - Move from: first peg (start)
      - To: second peg (end)
      - Using as intermediate: third peg (intermediate)
    - Third part:
      - Move from: second peg (start)
      - To: third peg (end)
      - Using as intermediate: first peg (intermediate)

Towers of Hanoi

```
def move_tower(num_discs, start, end, intermed):
    if num_discs == 1:
        move_one_disc(start, end)
    else:
        move_tower(num_discs - 1, start, intermed, end)
        move_one_disc(start, end)
        move_tower(num_discs - 1, intermed, end, start)

def move_one_disc(p1, p2):
    print('Move the disc from peg', p1, 'to peg', p2)
```

Towers of Hanoi

- Let's put this in a program and run it.
- Does anyone think that an iterative solution would be possible? If so, do you think it'll be easier?
- You can think about this on your own if you wish and try to come up with an iterative (that is, non-recursive) solution.

Queues and Stacks

- A queue is a data structure that has the following characteristics
  - It is linear
  - Uses FIFO (first in, first out) processing
- Queue operations
  - Enqueue – add an item to the *rear* of the queue
  - Dequeue – remove an item from the *front* of the queue
  - Empty – returns true if the queue is empty
- What's significant about a queue is that there are no insert in anywhere or remove from anywhere in the queue. The only place to add something to the queue is the rear, and the only place to remove something from the queue is the front.

Queues and Stacks

- A stack is a data structure that has the following characteristics
  - It is linear
  - Uses LIFO (last in, first out) processing
- Stack operations
  - Push – add an item to the *top* of the stack
  - Pop – remove an item from the *top* of the stack
  - Empty – returns true if the stack is empty
  - Peek – retrieve information about the item on *top* of the stack without removing it
- The only allowable ways to put an item into and to get an item from the stack is via push and pop. There are no insert in anywhere or remove from anywhere in the stack.
- What if there was no peek? Is it redundant --- could a series of the existing operations achieve the same functionality?

Queues and Stacks

- Can anyone think of real world examples that are naturally modeled by queues?
- Can anyone think of a real world example that is naturally modeled by a stack?
- Let's see visual representations of a queue and a stack on the board.

Queues and Stacks

- Can anyone think of real world examples that are naturally modeled by queues?
  - Line of people at grocery store checkout
  - Line of airplanes waiting for takeoff
- Can anyone think of a real world example that is naturally modeled by a stack?
  - Plates at a salad bar
    - A customer takes the top plate (pop)
    - When new plates come out, they are "pushed" to the top of the stack.
  - Why is this example not a queue?

Queues and Stacks

- Method/function calls
  - A new call to a function causes its local data and other state information (where in the method does it return to, etc.) to be pushed onto the stack
  - The function calls finish in reverse order, so when a method call ends, its local data/state is popped off the stack

  – You've seen me write information about function calls on the board in stack format, specifically when the functions were recursive.

Queues and Stacks

- Could we implement a Queue with
  - a linked list
  - an list
- Could we implement a Stack with
  - a linked list
  - an list

Queues and Stacks

- Let's implement a stack with a python list
  - what will be our instance variables?