CS16 Practice Midterm

due Never

The real test will consist of problems similar to some of the problems listed below and draw from the following topics. A really good way to study is to do all the problems below independently and then compare your solutions with a friend. If you actually solve the problems with friends, be sure you understand the solutions, and how to get to the solutions yourself. Memorizing the solutions to these problems will be of almost no value on the actual test. Make sure none of the following topics are foreign to you.
Note : This practice exam is longer than the actual exam will be, so don't worry if it takes you longer to attempt.

Topics to Study

Anything covered in lecture, projects, or homeworks before starting the graphs unit is fair game! Here is a list of topics you should be familiar with!

- Dynamic Programming
- Analysis and Big-O, Big-Omega, Big-Theta
- Amortized Analysis, Expected Analysis
- Expanding Data Structures (Stacks, Queues)
- Hashing, Sets, and Dictionaries
- Arrays
- Trees and Traversals
- Binary Search Trees
- Priority Queues, Heaps

Topics You Are Not Responsible For

- Hashing proof
- Proving expected runtimes (you should know *what* the expected runtimes of various algorithms/data structures are, but you will not need to prove that something has a certain expected runtime)
- Sorting
- Graphs
- DAGs and Topsort
- Decision Trees

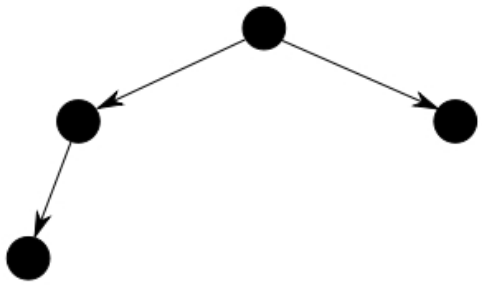
1 Dynamic Programming

Write the pseudocode for the most time-efficient algorithm you would use to solve each dynamic programming problem. **Please note that runtime is the most important part of this problem.**

> You're given a rope of length n and list of prices of rope of length i where $1 <= i <= n$. Find the optimal way to cut the rope into smaller ropes in order to maximize profit.
>
> Example:
> You're given the array lengths[] = [1,2,3,4,5], prices[] = [1,5,8,9,10], and a rope length of 4. Here the maximized profit would be to cut the rope into two pieces of length 2 each for a profit of 10.
>
> Write pseudocode for a way to solve this problem dynamically. A dynamic solution will break the large problem into smaller subproblems and then use the solutions to build up to solving the whole problem. *Note: Refer to HW3 for more practice with dynamic programming*

2 Tree Traversal

A binary tree of height h can have at most $2^{h+1} - 1$ nodes, as we showed in class. We define the *fullness* of a tree as the number of nodes in the tree divided by the maximum possible number of nodes for a tree of that height. Thus the tree below has a fullness of 4/7 (the height is 2, and there are 4 nodes, so the fullness is $4/(2^3 - 1) = 4/7$).



Design an efficient linear-time algorithm to compute, for any binary tree, the *minimum* fullness of any subtree of the tree (including the tree itself). In the example, there are four subtrees: the left leaf (fullness = 1), the right leaf (fullness = 1), the left subtree of the root (fullness 2/3) and the root (fullness = 4/7). The algorithm will return 4/7 because $4/7 < 2/3$.

3 Recurrence and Induction

1. The following code messes with the entries of an array, and then recursively calls itself on a smaller array. You can assume that the input array size n is a power of two.

```
def transform(arr):
    """transform: int array -> int array
    Purpose: takes an array of n ints, where n is a power of 2, and returns an array of n ints
    """

    if len(arr) == 1:
        return arr

    half = len(arr)/2
    sums = [0] * half   # build arrays of n/2 zeroes
    diffs = [0] * half
```

```
    for i in range(0, half):
        a = arr[2 * i]
        b = arr[2 * i + 1]
        sums[i] = a + b
        diffs[i] = a - b
    return transform(diffs) + sums  # Note: list1 + list2 concatenates
                                    # the lists: [1]+[2,3] => [1,2,3]
```

(a) Let $T(n)$ be the running time of transform on any array of size n. Write a recurrence relation for T. Be sure to include a base case.

(b) Use plug-n-chug with $n = 1, 2, 4$, and 8 to conjecture a big-O solution to the recurrence.

2. (a) Here's a recurrence relation for a different function, S:

- $S(1) = 1$
- $S(n) = n^2 + S(\frac{n}{2})$

The solution to this recurrence (for n a power of 2) is:

$$S(n) = \frac{4n^2 - 1}{3}$$

Prove this by induction.

Note: you need only handle the case where n is a power of two, i.e., where $n = 2^k$ for some non-negative integer k.

Hint (only read if stuck): Think about how this influences the induction step? In the induction step you are assuming S(p) is true, where p is any value of n, and then proving that S(the value that follows p) is true. However, if the sequence of numbers of n that are valid are when n is a power of 2, what is the number that follows p that is stille a valid number for n?

(b) Is the function S $O(1)$? Is it $O(n^3)$? Is it $\Omega(n^2/2)$? Is it $\Theta(n)$? Explain each case.

3. (a) A (0,2) binary tree T is one in which every node has out-degree zero or two, i.e., it has either two children or it's a leaf. Define a (0,2) binary tree recursively (in terms of itself). Take a look at HW5 for an example of a recursive definition of a tree.

(b) Prove by induction that the number of nodes in a regular binary tree is one more than the number of edges.

4 Amortized Analysis

As you saw in Homework 1, you can implement a queue with very little additional work by using two stacks, `in` and `out`. They both start out empty. To enqueue an item, you push it onto `in`. To dequeue an item, you pop it from `out`. Of course, that depends on `out` containing something! If `out` is empty, you first "pour" all the items from `in` into `out`, and then pop from `out`. This is what "pouring" looks like:

```
def pour():
    while not in.empty():
        out.push(in.pop())
```

(a) Draw a picture to indicate the state of the stacks `in` and `out` in an empty queue to which the following operations are applied: enq(A), enq(B), deq(), enq(C), deq(), deq(). You should draw a total of seven pictures, the first and last showing two empty stacks.

(b) Explain why enqueuing is worst-case $O(1)$ and dequeuing is worst-case $O(n)$, where n is the number of items in the queue.

(c) Explain why the amortized cost of dequeuing, in any sequence of n operations on an empty queue, is $O(1)$.

5 Hashing

What is a good hash function? What's a bad one? What are hashtables and how do they work?

6 Pseudocode

Suppose you have a stack and you want to know whether it's "almost empty". Write pseudocode for a method that returns "true" if the stack is either empty or has just one item in it. The Stack class you're enhancing does not keep track of its current size, n, but has an *isEmpty()* function. Your method should be $O(1)$, i.e., constant time.