1 Recurrence and Induction

(a) Recurrence relation for $T$.

- $T(1) = c_0$
- $T(n) = T\left(\frac{n}{2}\right) + c_1n$

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

(i) $T(1) = c_0$

(ii) $T(2) = T(1) + c_1 \times 2 = 2c_1 + c_0$

(iii) $T(4) = T(2) + c_1 \times 4 = 4c_1 + 2c_1 + c_0 = 6c_1 + c_0$

(iv) $T(8) = T(4) + c_1 \times 8 = 8c_1 + 6c_1 + c_0 = 14c_1 + c_0$

We can see that $T(n)$ is a linear function, with recurrence relation $T(n) = c_0 + (2n - 2)c_1$ which makes transform $O(n)$.

(c) Here’s a recurrence relation for a different function, $S$:

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

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

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

Proof by Induction

Base case:

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

Assume true for $n = k$ (Inductive assumption):

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

Show true for $n = 2k$ (General case):

$$S(2k) = (2k)^2 + S(k) = 4k^2 + \frac{4k^2 - 1}{3} = \frac{16k^2 - 1}{3} = \frac{4(2k)^2 - 1}{3} = \frac{4(2k)^2 - 1}{3}$$

Since $P(k) \rightarrow P(2k)$, and $P(1)$ is true, $P(k)$ is true where $n = 2^k$ for some non-negative integer $k$.

(d) The function $S$ is $O(n^3)$ and $\Omega(n^2/2)$. 
2 Tree Induction

(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. Give a recursive definition of a (0,2) binary tree.

Base case: 1 node without any children is a (0,2) tree

Recursive definition: A (0,2) tree is a binary tree where both the left and right children are (0,2) binary trees

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

$P(n)$: For a tree of size $n$, there are $n-1$ edges.

Base case: $P(1) = 0$. This is trivially true, because a tree with only one node has no edges

Assumption: $P(k)$ is true, i.e. for a tree of size $k$, there are $k-1$ edges

Inductive step: Prove that $P(k+1)$ is also true:

Take any tree with $k+1$ nodes. In any binary tree, there must be at least one leaf node. We can choose any of those leaf nodes and delete it. This means we will also delete the edge connecting it to its parent. Now we are left with a tree of size $k$. By our inductive assumption, this tree has $k-1$ edges. Since we had to remove one edge (and one node) to reduce our tree of size $k+1$ to one of size $k$, we have shown that the number of edges in a tree of size $k+1$ is $k-1+1 = k$.

Alternatively, consider a tree with $k$ nodes, to add a node to the tree, it must be connected to the tree by one new edge. Now the tree with $k$ nodes had $k-1$ edges by the inductive assumption, therefore the addition of one node with one edge takes the number of edges to $k$, and the number of nodes to $k+1$.

Conclusion: Since we have proven $P(1)$ and shown that $P(k) \Rightarrow P(k+1)$, we have shown that for every binary tree with one or more nodes, the number of edges is one less than the number of nodes, i.e. that $P(n)$ is true.

3 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:

```python
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.

Enqueuing is worst case $O(1)$ because it is only a push() on the first stack (and, of course, a push is a constant time operation). Dequeuing is worst case $O(n)$ because if the out stack is empty, every
element in the queue will have to be moved from the in to the out stack before it can be popped from the out stack.

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

Though each step is \( O(n) \), the next \( n \) dequeue operations will be constant since we’ve moved those elements to the out stack. So for \( n \) operations, the amortized cost of dequeuing is \( O(1) \).

4 Hashing

What is a good hash function? What’s a bad one? What are hashtables and how do they work? Explain the difference between a hashset and a hashtable.

A good hash function uniformly distributes the inputs across all buckets, regardless of whether or not the input is random. A bad hash function is one that maps many inputs to a few buckets. Hashtables are arrays of arrays (or linked lists) that map keys to (key,value) pairs using a hash function. A hashset is the same as hashtable except that the value is the key - the hash function is used to determine the location where the key itself is stored.

5 Search

If we knew \( n \) we could use binary search to find \( b \) (or find out it does not exist) in \( O(\log n) \) time. As pointed out in the problem, we do not know \( n \). We can, however, check if \( n \) is at most some integer \( i \) by checking whether \( f(i + 1) \) evaluates to -1 or not. Observe that \( n \) must lie between two consecutive powers of 2; if \( k = \lfloor \log_2 n \rfloor \), then \( n \in [2^k, 2^{k+1}) \). Therefore, we can find an upper bound on \( n \) by checking increasing powers \( k \) of 2 and stop as soon as \( f(2^k) \) evaluates to -1. By the observation above, this will take exactly \( \lfloor \log_2 n \rfloor + 1 \) which is \( O(\log n) \) steps. Moreover, the bound \( 2^k \) we find is at most twice as large as \( n \), so performing binary search for \( b \) in the range \([1, 2^k]\) instead of \([1, n]\) takes just \( O(\log n) \) steps as well. When implementing that binary search we should be careful, though, since \( f \) is not exactly increasing in \([1, 2^k]\) (it is until \( n \), but then it becomes just -1). We can overcome this by treating -1 as if it were \(+\infty\).

Note that once we find the upper bound \( 2^k \) we could first binary search for \( n \) in the range \([1, 2^k]\) and then search for \( b \) in \([1, n]\). That would be less efficient, though not asymptotically less efficient.
One possible solution:

```python
def find(f, p):
    curr = 1
    while (f(curr) < p and f(curr) > 0):
        curr = curr * 2
    return search(curr/2, curr)

def search(low, high, f, p):
    if f(low) == p return True
    if f(high) == p return True
    if low + 1 == high return False
    if f((high-low)/2 + low) > p
        return search(low, (high-low)/2 + low, f, p)
    return search((high-low)/2 + low, high, f, p)
```

Another possible solution:

```python
def find(f, p):
    #
    # Consumes: f -> function
    #          p -> integer
    # Produces: boolean
    # Purpose: Finds if there exists an integer b such that f(b) = p
    #
    return search(f, p, 1, 0.5)

def search(f, p, c, i):
    #
    # Consumes: f -> function
    #          p -> integer (goal value)
    #          c -> integer (current test value)
    #          i -> integer (interval value)
    # Produces: boolean
    # Purpose: Find's if there exists an integer b such that f(b) = p
    #
    if i < 0.5
        return False
    elif f(c) == p
        return True
    elif f(c) == -1 or f(c) > p
        return search(f, p, c-i, i/4)
    else
        return search(f, p, c+i, 2i)
```