Module 2 Problem Set

1. [10 pts] From problem set 1: Find an efficient algorithm to compute x^m, for arbitrary non-negative integer m.
Hint: used divide and conquer.

Prove your algorithm is correct

2. [10 pts] Suppose that we choose 3 as our factor in the "Big Box Store" algorithm. That is, when the stack is full we allocate 3x the amount of current storage. What is the complexity of M operations in this case? [Big Oh answer].

3. [10 pts] From problem set 1: Suppose you were given a list of numbers that were in ascending order. Some evil mastermind, however, has rearranged the list to "cycle around" from some middle point. Find an efficient algorithm to determine the original starting point from the cyclic list.

Prove your algorithm is correct

4. [10 pts] Repeat question 2 with a factor of 11. That is, when the stack is full, allocate 11x the current size.
What is the complexity of M operations in case where factor = 11?
Give your answer (and the associated reasoning) in Big Oh notation.

5. [20 pts] For this problem, let i be the sets in a union/find problem.
Let f[i] be the find array associated with this union/find problem.

Draw the tree associated with the f array below:

```
         i    1 2 3 4 5 6 7 8 9 10
              ----------------------
      f[i]    2 2 4 2 6 7 2 4 5 6
```

6. [20 pts] Can the find array of problem 3 be a result of running weighted quick union?
Explain why this is impossible OR ELSE give a sequence of union-find operations that produce the above table.

7. [20 pts] Suppose we introduce the "add_new_set()" operation. The operation will return a new whole number corresponding to a new disjoint set. This means that the number of nodes in our graph may change dynamically. Assuming add_new_set is O(1), what is the complexity of a sequence of add_new_set, union, and find operations?

1)

```
def my_pow(x, m):
  if(m == 0):
    return 1
  if(m == 1):
    return x
  pow_half = my_pow(x*x, m//2)
  elif(m % 2 == 0):
    return pow_half
  else:
    return x * pow_half
```

We prove using induction.
The base case:
m=0, the function return 1. $x^0 = 1$. where x is not 0. This is trivial.
m=1. the functions return x. $x^1 = x$. This is trivial.

We assume the algorithms works and is correct for all $m = 0, 1, 2, 3, ..., k$.

We now see that if $k+1$ is even, the algorithm calls $my\_pow(x*x, m//2)$, where $m//2$ is floor division. By the induction hypothesis, the algorithm is correct for $m//2$. Therefore, the algorithm is correct for $k+1$ when m is even.

We see that if $k+1$ is odd, the algorithm calls $x * my\_pow(x*x, m//2)$. By the induction hypothesis, the algorithm is correct for $m//2$. Therefore, the algorithm is correct for $k+1$ when m is odd.

Since we covered both cases, the algorithm is correct for all m.

2)
For copying we see: $\sum_{j=1}^{k} 3^{j-1} = (1/2)(3^k - 1)$
$T_{copy}(M) = (1/2)(3^{(log_3(M)+1)} - 1) = (1/2)(3M - 1)$ where M is power of 3.
$T_{push}(M) = M + M + (1/2)(3M - 1) = O(M)$
$T_{pop} = M = O(M)$
So with 3 as factor, M stack operations is still $O(M)$.

3)

```
def find_start(li):
  lo = 0
  hi = len(li)-1
  mid = 0
  while(hi>=lo):
    mid = (hi+lo) //2
    if(li[mid] > li[hi]):
      lo = mid+1
    elif(li[mid] < li[hi]):
      hi = mid
    else:
      return mid
```

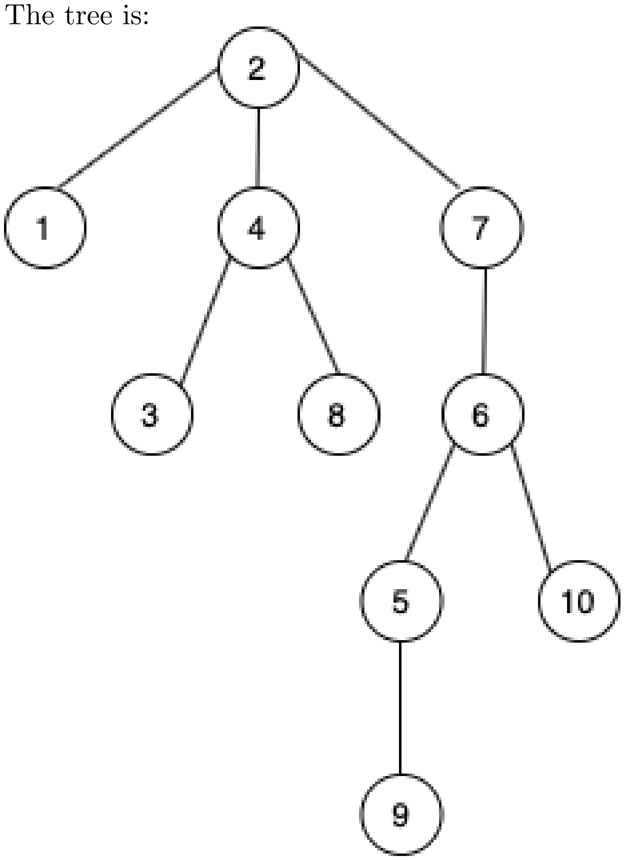
We prove using induction.
Base case: If list is 1 element, lo, hi, and mid are 0, so find_start(li) returns the mid element, which is the lowest starting number. If list is 2 elements, the algorithm picks index 0 (since floor division) as mid and compares that with index 1, so lo = 0, mid=0, hi=1. Comparing li[mid] to li[hi] in the if statements,

we either update lo = 0+1 or hi = 0 based on the comparison, and from there, returns mid by first base case.

We assume the algorithm works and is correct the for all lists of size $n = 0, 1, 2, 3, ..., k$.
We need to show it works for k+1. With k+1, find_start(li) starts with lo=0, hi=k, mid = (hi+lo)//2 in first iteration of loop and compares li[mid] to li[hi]. If li[mid] > li[hi], lo=mid+1 and If li[mid] < li[hi], hi = mid. So we throw away half of the list and the we only look at elements lo=0 to hi=(hi+lo)//2 or mid = (hi+lo)//2 + 1 to hi = k. By the inductive hypothesis, the algorithm is correct for either of these 2 cases. Therefore, the algorithm is correct for all lists.

4)
For copying we see:
$\sum_{j=1}^{k} 11^{j-1} = (1/10)(11^k - 1)$
$T_{copy}(M) = (1/10)(11^{(log_{11}(M)+1)} - 1) = (1/10)(11M - 1)$ where M is power of 11.
$T_{push}(M) = M + M + (1/10)(11M - 1) = O(M)$
$T_{pop} = M = O(M)$
So with 11 as factor, M stack operations is still $O(M)$.

5)
The tree is:



6)
The find array cannot be result of running weighted quick union. It's height is 5 from nodes 2,7,6,5,9. With weighted quick union the height is at most log_2 (N), so log_2 (10) is 3. In this problem, the height 5 is greater than 3.

7) This problem doesn't specify the type of running algorithm. The complexity of a sequence of M add_new_set, union, and find operations will depend on the algorithm and if is M >> N or N << M.
There is a pretty detailed paper I read about this. With empty data structure, pay it forward, naive linking of M sequence and M $>= N$, the complexity is $O(MlogN)$. The proof is complex and nontrivial.
`https://e-maxx.ru/bookz/files/dsu/Worst-Case%20Analysis%20of%20Set%20Union%20Algorithms.%20Tarjan,%20Leeuwen.pdf`

The naive set union algorithm runs in $O(N + MN)$ for a sequence M of the 3 operations, since add_new_set and link is $O(1)$, Union = $O(N)$ and find=$O(1)$.