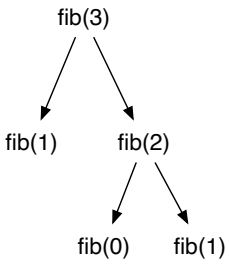
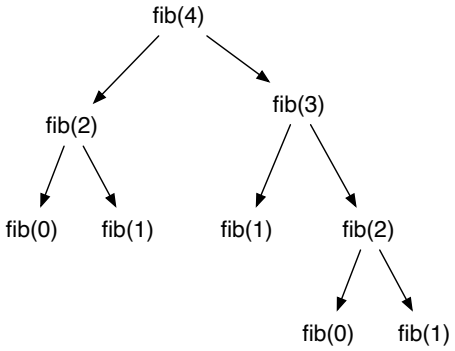
Homework 1:

Problem 1:

```
def fib(n):
   if n <= 1:
       return n
   return fib(n-1)+fib(n-2)
```

The Fibonacci function can be implemented with recursive calls. Let $T(n)$ denote the number of recursive calls when fib(n) is called. For example, $T(0) = T(1) = 0$, $T(2) = 2$, $T(3) = 4$, and $T(4) = 8$. We can illustrate this in the following figure where the number of arrows gives the number of recursive calls.





What is the recurrence expression for $T(n+2)$?

Aside:

We can see from the figure that the problem with this recursion is the multiple calls to the function with the same argument. It turns out using Mathematica that a closed form expression for T is

$$\left(\frac{1}{2}\left(1+\sqrt{5}\right)\right)^n + \left(\frac{2}{1+\sqrt{5}}\right)^n \cos(\pi n) + \frac{\left(\frac{1}{2}\left(1+\sqrt{5}\right)\right)^n - \left(\frac{2}{1+\sqrt{5}}\right)^n \cos(\pi n)}{\sqrt{5}} - 2$$

where of course $\cos(n\pi)$ oscillates between 1 and -1. The function is $\Omega((\frac{1+\sqrt{5}}{2})^n)$.

Problem 2:

```
def fib(n):
   lista = [0,1]
   for i in range(n-2):
      lista.append( lista[-1]+lista[-2] )
   return lista[n]
```

A much better implementation of the Fibonacci numbers uses linear lists in Python. Recall that `lista[-1]` is the last element of a list, `lista[-2]` is the penultimate number in the list and that `append` appends the argument to the list.
Assuming that the append operation always takes constant time, what is the asymptotic runtime of this function. What will the function return if *n* is negative, zero, or one? Use induction to show that the function correctly calculates the Fibonacci numbers.

Problem 3:

Loop invariants can actually help us to find programming solutions. In Quicksort, we alternate between partitioning an array and recursion on the obtained partitions of the array. The partitioning step reorders the elements of an array `A` around an object `p` such that

1. all elements in `A[:low]` are less than `p`,
2. all elements in `A[low:high]` are equal to `p`,
3. all elements in `A[high:]` are greater than `p`.

Show that the following pseudo-code (actually Python code) is correct using the invariant

1. all elements in `A[:low]` are less than `p`,
2. all elements in `A[low:mid]` are equal to `p`,
3. all elements in `A[high:]` are greater than `p`.
4. also `low <= mid <= high`

The invariant says nothing about the elements in `A[mid:high]`.

```
def partition(A, p):
    low = 0; high = len(A); mid=0
    while mid<high:
        a = A[mid]
        if a < p:
            A[mid] = A[low]
            A[low] = a
            low += 1
            mid += 1
        elif a == p:
            mid += 1
        else:   # a > p
            A[mid] = A[high-1]
            A[high-1] = a
            high = high-1
```

Show also that the algorithm terminates. The following drawing might be helpful in understanding the invariants. There, `low` points to the array element just after the "smaller than p" array, etc.