Problem 1

Problem. Compute the $n$'th Fibonacci number,

\[ F_0 = 0, \quad F_1 = 1, \quad \forall n > 1, \quad F_n = F_{n-1} + F_{n-2} \]

Algorithm. Compute recursively with memoization (attempt)

```python
def fib_memoize(n):
    fib_memo = {0: 0, 1: 1}
    if n in fib_memo:
        return fib_memo[n]
    else:
        ans = fib_memoize(n-1) + fib_memoize(n-2)
        fib_memo[n] = ans
        return ans
```
PROBLEM. determine experimentally the running time of set intersection.

ALGORITHM.

```python
import timeit

def intersect(x, y):
    a = set(range(10 ** x))
    b = set(range(10 ** y))
    return b.intersection(a)

t = timeit.Timer('intersect(3,3)',
                'from __main__ import intersect')
print min(t.repeat(10,1))
```
On Homework

- Homework 1: max 9/10, mean 3.1, standard deviation 3.0
- New late homework policy.
- Start early, test and debug your code.
- Credit all sources.
GRAPH ALGORITHMS
Graph traversal

**Problem.** Given $G$ and a node $s$, find the set of nodes reachable from $s$ (i.e., the connected component of $G$ containing $s$).

**Generic Algorithm.**

- $R = \{s\}$.
- while there is an edge $(u, v)$ where $u \in R$, $v \notin R$, add $v$ to $R$

**Examples.**

- **Breadth-First Search.** explore in successive layers, based on distance from $s$.
- **Depth-First Search.** keep following the first edge, and backtrack when you reach a “dead end”.
- Both algorithms produce a rooted tree, along with an ordering of vertices.
DFS(u):
explored[u] = True
for each neighbor v of u:
    if not explored[v]:
        T.append([u,v])
        DFS(v)

PROPERTIES.

▶ Suppose \((x, y)\) is an edge in \(G\) not in the DFS tree \(T\). Then one of \(x\) or \(y\) is an ancestor of the other.
BFS(s):

discovered = [False] * n, discovered[s] = True

i = 0, L[0] = [s], T = []

while L[i] is not empty:
    L[i+1] = []

    for each node u in L[i]:
        # add all undiscovered neighbors of u
        for each neighbor v of u:
            if not discovered[v]:
                discovered[v] = True
                T.append([u,v])
                L[i+1].append(v)

    i = i+1
BFS(s):

discovered = [False] * n, discovered[s] = True
L = [s], T = []
while (L):
    u = L.pop(0)  # FIFO, add all undiscovered neighbors of u
    for each neighbor v of u:
        if not discovered[v]:
            discovered[v] = True
            T.append([u,v])
            L.append(v)

REMARKS.

- L is a queue (= union of L[0],..,L[n-1])
- Running time: O(m + n)
DFS(s):

\[\text{explored} = [\text{False}] \ast n, \ L = [s], \ T=[]\]

\[\text{while} \ (L):\]

\[u = L.\text{pop()} \ # \text{LIFO, explore } u \text{ next if unexplored}\]

\[\text{if not } \text{explored}[u]:\]

\[\text{explored}[u] = \text{True}\]

\[T.\text{append}([u,\text{parent}[u]])\]

\[L.\text{extend}[\text{nb}[u]] \ # \text{add neighbors of } u \text{ to stack}\]

**REMARKS.**

- Same as recursive implementation, except each adjacency list is processed in reverse order.
- L is a stack, possibly with repetitions
- Running time: \( O(m + n) \)
Testing Bipartiteness

**DEFINITION.** An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue s.t. the end-points of every edge has different colors.

**FACT.** A graph $G$ is bipartite iff it does not contain an odd cycle.

**PROBLEM.** Given a graph $G$, determine if it is bipartite.

**ALGORITHM.**
- Run a BFS.
- Color odd layers blue, even layers red
- Check every edge has end-points with different colors

**RUNNING TIME.** linear time $O(m + n)$. 
THE END