COMP S265F Design and Analysis of Algorithms
Lab 3: Fibonacci Numbers, Binary Tree, and Dynamic Programming

This lab introduces the Fibonacci numbers, how to print a binary tree using the Python package `graphviz`, and a powerful algorithm design technique called *Dynamic Programming*.

1. Fibonacci numbers

Leonardo of Pisa (a.k.a. Fibonacci) invented the *Fibonacci numbers* when he was studying the population dynamics of rabbits. He simplified the problem with the following assumptions:

- A pair of rabbits gives birth to a pair of children every year.
- These children are too young to have children of their own until two years later.
- Rabbits never die.

Then, the number of *pairs* of rabbits can be expressed as the following function of years:

- $F(0) = 0$ (We don't have any pairs of rabbits.)
- $F(1) = 1$ (We start with one pair A of rabbits in Year 1.)
- $F(2) = 1$ (The pair A is too young to have their own children in Year 2.)
- $F(3) = 2$ (A gives birth to a pair B.)
- $F(4) = 3$ (A gives birth to another pair C, but B is too young to have children.)
- $F(5) = 5$ (There are two new pairs by A and B.)

Therefore, the *Fibonacci numbers* $F(n)$ can be defined, as follows:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for any $n \geq 2$

In general, to compute $F(n)$ for Year n, we can see that all the previous rabbits are still there $(F(n-1))$ plus that every pair in two years ago gives birth to a new pair $(F(n-2))$.

It is well-known that Fibonacci number has a closed-form expression $F(n) = \dfrac{\phi^n - (1-\phi)^n}{\phi - (1-\phi)}$, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. By careful calculations, we can approximate $F(n) \approx 2^{0.694n}$.

We can use recursion to find the n-th Fibonacci number $F(n)$:

01fib.py

```python
from sys import stdin

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

def main():
    """Reads a user-specified non-negative integer n,
    and then prints the n-th Fibonacci number F(n)."""
```

```

    n = int(stdin.readline())
    print(f"F({n}) = {fib(n)}")

if __name__ == "__main__":
    print(main.__doc__)
    main()
```

Time complexity. Let $T(n)$ be the number of steps to compute `fib(n)`. Therefore, we have

- $T(n) = 2$ for $n = 0, 1$
- $T(n) = T(n-1) + T(n-2) + 3$ for $n \geq 2$

Comparing the recurrences of $T(n)$ and $F(n)$, we immediately see that $T(n) \geq F(n) = \Omega(2^{0.694n})$. That is, the time complexity of `fib(n)` is exponential to n; it takes a long time to compute small inputs like $F(50)$.

2. Fibonacci trees: A binary tree

A *Fibonacci tree* represents the recursive call structure of the Fibonacci computation. The root of a Fibonacci tree is the value n that represents the n-th Fibonacci number $F(n)$. In general, the root has two children: the left is the value $n-1$ for $F(n-1)$ and the right is the value $n-2$ for $F(n-2)$. All other internal nodes (except $n = 0, 1$) have two children defined similarly. Therefore, a Fibonacci tree is a binary tree.

We can define a class `Node` to represent a tree node:

02fib.py

```python
from sys import stdin

num_node = 0

class Node:
    def __init__(self, n, left, right):
        global num_node
        self.id = str(num_node)
        num_node += 1
        self.n = n
        self.left = left
        self.right = right

def fib(n):
    if n <= 1:
        return Node(n, None, None)
    else:
        return Node(n, fib(n-1), fib(n-2))

def traverse(node):
    if node.left == None:
        print(f"#{node.id} [{node.n}] has no children")
    else:
        print(f"#{node.id} [{node.n}] has two children #{node.left.id} [{node.left.n}] and #{node.right.id} [{node.right.n}]")
        traverse(node.left)
        traverse(node.right)

def main():
    """Reads a user-specified non-negative integer n,
    and then prints the structure of the Fibonacci tree for F(n)."""

    n = int(stdin.readline())
```

```python
    root = fib(n)
    traverse(root)

if __name__ == "__main__":
    print(main.__doc__)
    main()
```

In the constructor of `Node`, the keyword `global` makes the variable `num_node` a global variable (otherwise, it is a variable local to the method). The above program assumes that all tree nodes have a unique id, and it outputs the Fibonacci tree structure using lines with following formats:

- "#x [n] has two children #y [$n-1$] and #z [$n-2$]" if a node n with id x has two child nodes $n-1$ with id y and $n-2$ with id z.
- "#x [n] has no children" if a node n with id x does not have any children.

It would be more helpful and insightful to display the Fibonacci tree by a graph. We may use the Python package `graphviz` (which can be installed by `conda install python-graphviz`). You can instantiate a `Graph` object that has the following three methods:

- `node(`x`, label=`L`)`: Create a node with id x and displayed it as L.
- `edge(`x`, `y`, label=`L`)`: Create an edge connecting nodes with id x and y and add an edge label L.
- `view()`: Display the graph as a PDF file.

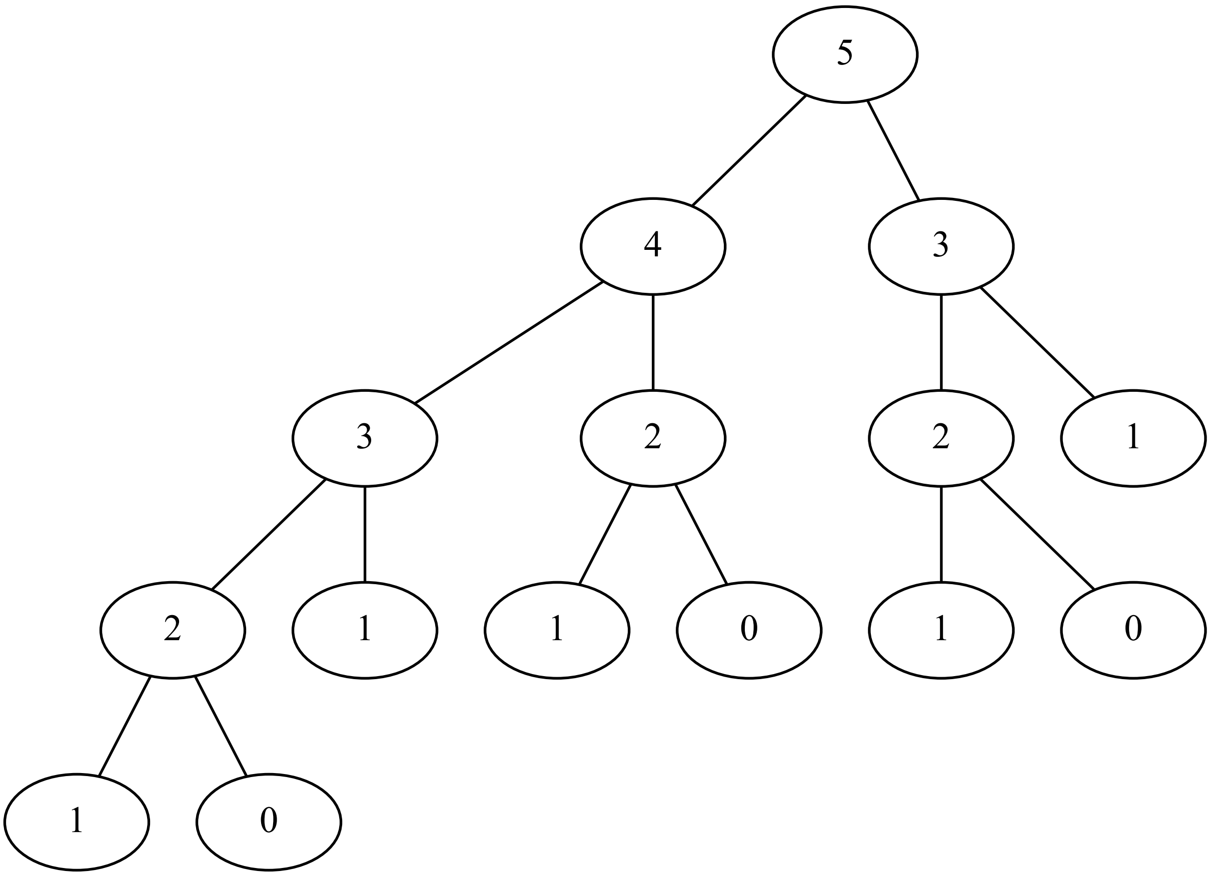
Then we can revise the program, as follows:

02fibtree.py

```python
from sys import stdin
from graphviz import Graph

num_node = 0
tree = Graph()

class Node:
    def __init__(self, n, left, right):
        global num_node
        self.id = str(num_node)
        num_node += 1
        self.n = n
        self.left = left
        self.right = right

def fib(n):
    global tree
    if n <= 1:
        node = Node(n, None, None)
        tree.node(node.id, label=str(n))
    else:
        left_node = fib(n-1)
        right_node = fib(n-2)
        node = Node(n, left_node, right_node)
        tree.node(node.id, label=str(node.n))
        tree.edge(node.id, left_node.id)
        tree.edge(node.id, right_node.id)
    return node

def main():
    """Reads a user-specified non-negative integer n,
    and then prints the structure of the Fibonacci tree for F(n)."""
```

```python
    global tree

    n = int(stdin.readline())
    root = fib(n)
    tree.view()

if __name__ == "__main__":
    print(main.__doc__)
    main()
```

Below is the output of the Fibonacci tree for $F(5)$:



3. A polynomial-time algorithm: Dynamic programming

Let's try to understand why `fib(n)` is so slow. The Fibonacci tree shown in Section 2 shows the cascade of recusive invocations triggered by a single call to `fib(5)`. Notice that many computations are repeated! For example, `fib(2)` was repeated three times, and `fib(3)` was repeated twice.

We can apply a technique called *Dynamic Programming*. The idea is to simply store the results of the subproblems, so that we do not have to re-compute them when needed later. We will use a dictionary `f` to store the subproblem results:

03fib.py

```python
from sys import stdin
f = {}

def fib(n):
    global f

    if n in f: # if F(n) was computed before
        return f[n]

    # if F(n) was not computed before
    if n <= 1:
        f[n] = n
    else:
        f[n] = fib(n-1) + fib(n-2)
    return f[n]

def main():
    """Reads a user-specified non-negative integer n,
    and then prints the n-th Fibonacci number F(n)."""

    n = int(stdin.readline())
    print(f"F({n}) = {fib(n)}")

if __name__ == "__main__":
    print(main.__doc__)
    main()
```

Time complexity. Now, we only compute `fib(k)` once for every $k \leq n$, and then we can obtain its result by simply using $f[k]$. Therefore, the time complexity of the improved algorithm is $O(n)$. Notice that we have improved the time complexity from exponential to linear!

4. Exercises

Question 1. Using both the ***original*** and ***improved*** Euclid's Algorithm, find the greatest common divisor (*g.c.d.*) of the following numbers.

(a) $48, 36$

(b) $133, 728$

Question 2. Consider the following algorithm in pseudocode that prints an n-integer array in ascending order.

```
fuction(num_array):
    while num_array is not empty:
        min_num = num_array[0]
        for each number x in num_array:
            if min_num > x:
                min_num = x
        print(min_num)
        delete min_num from num_array
```

(a) What is its time complexity?

(b) Where is the bottleneck in its computation?

(c) How can we reduce the number of steps used in this bottleneck?