Simon Fraser University

Department of Mathematics

Bioinformatics Algorithms - Assignment 1

Name: Niloufar Saeidi

ID: 301590708

2.2

iterative algorithm

```python
def iterative(n):
    n_prime = [i+1 for i in n]
    length = len(n)
    indices = [0] * length
    while True:
        print(indices)
        i = length - 1
        while i >= 0:
            indices[i] += 1
            if indices[i] < n_prime[i]:
                break
            indices[i] = 0
            i -= 1

        if i < 0:
            break
```

We start from the last element and increase it from zero to n_d. Then set it to zero and increase the $d-1$th element one unit, keeping it the same for all possible values in the range n_d. And so on and so forth.

recursive algorithm

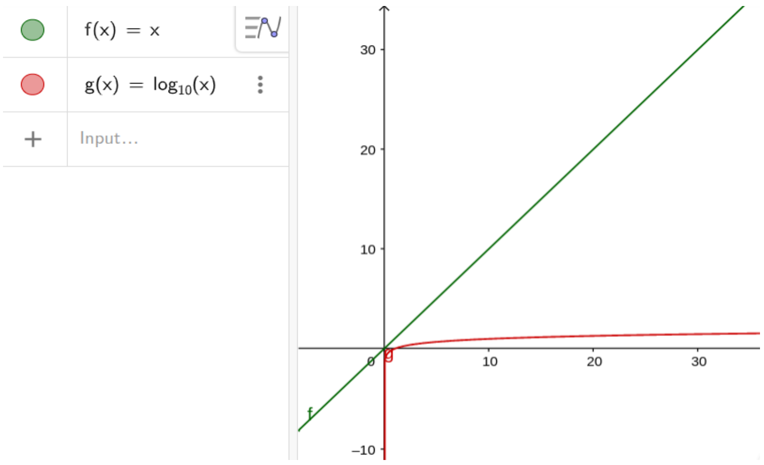
```python
n_prime = [i+1 for i in n]
def recursive(n_prime, idx=[]):
    if not n_prime:
        print(idx)
    else:
        window = n_prime[0]
        for i in range(window):
            recursive(n_prime[1:], idx + [i])
```

The exit case is when n_prime is empty. Each time the function is called, the left-most element of n_prime is not passed to the child function, and an iterator i is appended to idx which is empty at first. The values that are appended to idx are $[0, 1, \ldots, n\_i]$

2.3

- $log(n) = O(n)$: Correct

  because O provides the upper bound and so we know that for some $n > n_1$, $log(n)$ is always less than n.



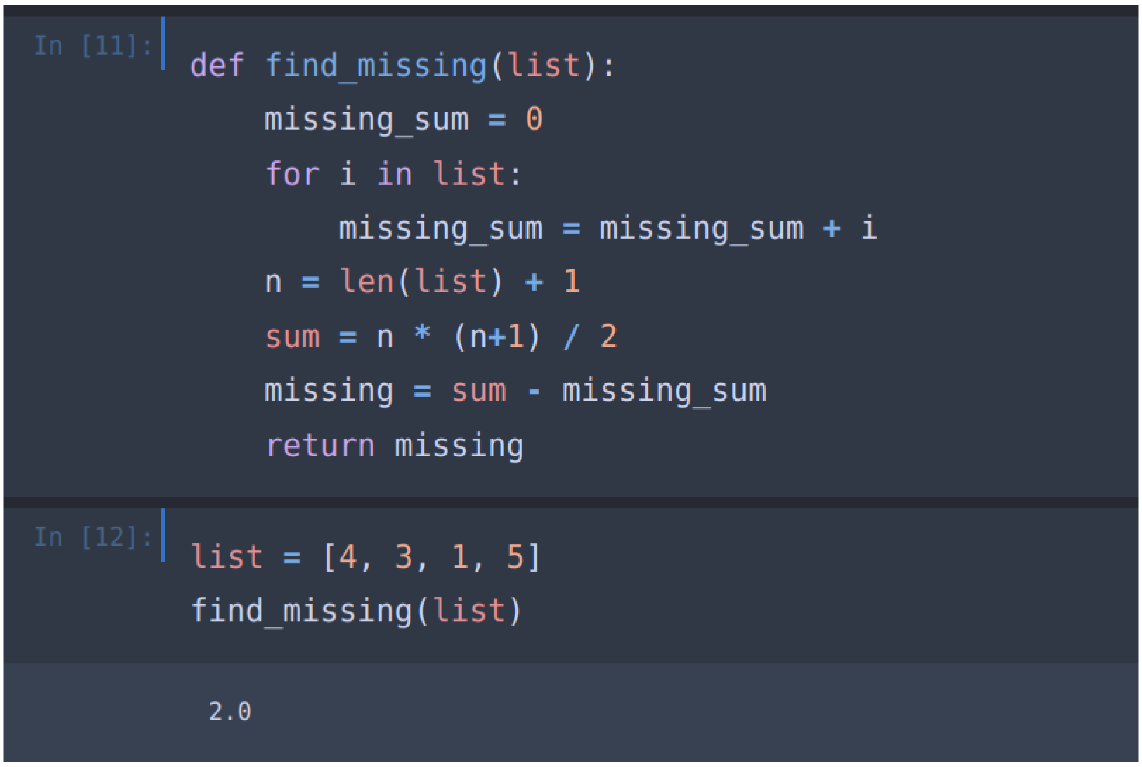
- $log(n) = \Omega(n)$: Wrong

  because Ω provides the lower bound

- $log(n) = \Theta(n)$: Wrong

  because Ω provides the tight bound

2.4



```
In [11]: def find_missing(list):
             missing_sum = 0
             for i in list:
                 missing_sum = missing_sum + i
             n = len(list) + 1
             sum = n * (n+1) / 2
             missing = sum - missing_sum
             return missing
```

```
In [12]: list = [4, 3, 1, 5]
         find_missing(list)
```

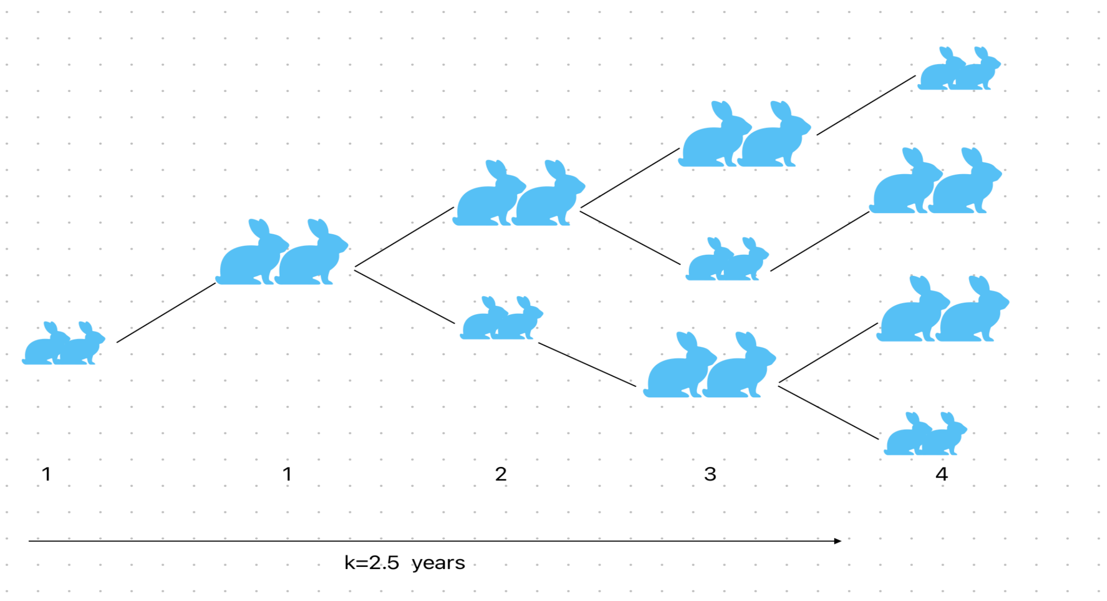
```
2.0
```

2.8

$F_n = F_{n-1} + F_{n-2} - F_{n-\lceil k+1 \rceil}$
We add 1 to k because we want to subtract the number of adults in the time step $n - \lceil k \rceil$, which is the number of kids in the step $n - \lceil k+1 \rceil$.

```
procedure ALG($n$)
    if $n == 1$ or $n == 2$ then
        return 1
    end if
    if $n == 3$ or $n == 4$ then
        return $n - 1$
    end if
    return ALG($n-1$)+ALG($n-2$)-ALG($n - \lceil k+1 \rceil$)
end procedure
```

Whether the number of rabbits exceeds the number of atoms in the universe or not depends on our assumptions, including the number of times rabbits reproduce during their lifetime, the amount of k, the initial population of rabbits, etc. As k gets smaller, it is more unlikely for the number of rabbits to ever exceed the number of atoms in the universe. Therefore, under our assumption where the number of initial rabbits is 1 and k is 2.5, it is unlikely



for them to become more than the number of atoms in the universe, which is around 10^80. For the given sequence, 1, 1, 2, 3, 4, my diagram looks like this: Where a pair of rabbits reproduce 3 times in their lifetime and do not get to see their last kids.

2.10

$f(n) = \Sigma_{i=1}^{n} = 1 + 2 + 3 + \ldots + n$
$\Rightarrow f(n) = n + (n-1) + (n-2) + \ldots + 1$
$\Rightarrow f(n) + f(n) = (n+1) + (n+1) + (n+1) + \ldots + (n+1)$
$\Rightarrow 2 * f(n) = n * (n-1)$
$\Rightarrow f(n) = \frac{n*(n+1)}{2}$

2.12

The answer is 15. M=40 is one of the examples for which the BETTERCHANGE algorithm does not work correctly. The reason is that the algorithm picks 25 first, and then it has to pick 10 and 5 to make it to 40. This will result in 3 coins, while the optimal solution is 2 coins, 20 and 20. If we add a coin with the value 15, this issue will be resolved and we can still have 2 coins if we pick 25 first.

2.16

If n =1, no one plays.
if n=2, player 1 wins.
if n=3, player 2 wins.
if n=4, player 1 wins. Because player 1 can make two piles of size 1 and 3, or 2 and 2. Then player two can make 3 piles with size 1 and 1 and 2. Then player 1 separates them into 4 piles and wins. We can deduce that whenever n is odd, player 2 wins, and whenever n is even, player 1 wins.
If we propose a binary tree for the game, we can prove that whether which player wins depends on n being even or odd only, and not anything else. No matter what action the players choose to take.