

Project 3
CPSC 335
7/21/2023

Group 7:
Kyler Farnsworth, Michael Daza, Andrew Young

1)

Pseudocode:

```
function find(parent, i)
    if parent[i] is equal to i
        return i
    else
        return find(parent, parent[i])

function union(parent, rank, x, y)
    x_root = find(parent, x)
    y_root = find(parent, y)

    if rank[x_root] is less than rank[y_root]
        parent[x_root] is assigned to y_root
    else if rank[x_root] is greater than rank[y_root]
        parent[y_root] is assigned to x_root
    else
        parent[y_root] is assigned to x_root
        rank[x_root] is incremented by 1

function kruskal(edges, num_vertices)
    sorted_edges = sort(edges, by=lambda x: x[1]) # Sort edges by weight in ascending order
    parent = array with elements from 0 to num_vertices - 1
    rank = array with num_vertices elements initialized to 0
    min_spanning_tree = empty list

    for each edge in sorted_edges
        u, v = first and second elements of edge
        weight = third element of edge

        if find(parent, u) is not equal to find(parent, v) # Check if adding this edge creates a cycle
            add [edge[0], weight] to min_spanning_tree
            union(parent, rank, u, v)

    return min_spanning_tree
```

Mathematical Analysis:

Time Complexity: $O(n \log n)$

The time complexity of Kruskal's algorithm in the provided code is $O(n \log n)$, where n is the number of edges in the graph. This complexity comes from the sorting step. The loop that iterates through sorted edges adds $O(n)$ time to the complexity. However when combining these we just reach a final complexity of $O(n \log n)$

Code:

```

def kruskal(edges):
    # sort edges by weight
    sorted_edges = []
    for i, siblings in enumerate(edges):
        for sibling in siblings:
            if i < sibling[0]:
                sorted_edges.append((i, sibling[0], sibling[1]))
    sorted_edges.sort(key=lambda x: x[2])

    # initialize disjoint set
    parent = [i for i in range(len(edges))]
    rank = [0 for _ in range(len(edges))]

    def find(x):
        if parent[x] != x:
            parent[x] = find(parent[x])
        return parent[x]

    def union(x, y):
        root_x = find(x)
        root_y = find(y)
        if root_x == root_y:
            return False
        if rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_x] = root_y
            if rank[root_x] == rank[root_y]:
                rank[root_y] += 1
        return True

    # build minimum spanning tree
    mst_edges = []
    for edge in sorted_edges:
        if union(edge[0], edge[1]):
            mst_edges.append(edge)

    # build result
    result = [[] for _ in range(len(edges))]
    for edge in mst_edges:
        result[edge[0]].append([edge[1], edge[2]])
        result[edge[1]].append([edge[0], edge[2]])

    return result

edges = [
    [[1, 3], [2, 5]],
    [[0, 3], [2, 10], [3, 12]],
    [[0, 5], [1, 10]],
    [[1, 12]]
]
print(kruskal(edges))

```

```
(env) student@tuffix-vm:~/Desktop$ python csp3.py  
[[[1, 3], [2, 5]], [[0, 3], [3, 12]], [[0, 5]], [[1, 12]]]
```

2a)

Pseudocode:

```
def FibSequence(n):  
    list = [0, 1]  
    if n is an existing index in list:  
        return list[n]  
    else:  
        list.append(FibSequence(n-1) + FibSequence(n-2))  
        return (FibSequence(n-1) + FibSequence(n-2))
```

Mathematical Analysis:

Time Complexity: $f(n) = 2T(n) + 3$

In the worst case scenario, we can expect this recursive function to have the following values: $r = 2$, $d = 1$, and $k = 0$.

Since $r < d^k$, we can apply Case 1 of the Master's Theorem to determine that the time complexity is $O(n^2)$.

My solution to this problem trades a small portion of the small space complexity for a significant savings in time complexity. Rather than returning the number of $\text{fib}(n-1) + \text{fib}(n-2)$ every time, I'm first checking the list, List, to see if the value already exists and then I'm returning the value if it is in the list. If the value is not already in the list, then we recursively call the function until we reach an index that has already been added. The amount of space required to store the list is nominal when compared to the amount of time it takes to recursively find larger values in the fibonacci sequence.

Code:

```
def fibSequence(n):
    list = [0, 1]
    if n <= 1:
        return list[n]
    elif (n+1 < len(list)):
        return list[n+1]
    else:
        fibNumber = fibSequence(n-1) + fibSequence(n-2)
        list.append(fibNumber)
    return (fibNumber)

print("The 15th number in the Fibonacci Sequence: ", fibSequence(15))
```

2b)**Pseudocode:**

#First create algorithm for equation 3 to be used when creating algorithms for equations 4 and 5

def fibonacci_equation3(n):

#Set and return a variable equal to the result of equation 3 using the given n

```
    final_num = (((1 + math.sqrt(5))**n) -
                  ((1 - math.sqrt(5))**n))/((2**n)*math.sqrt(5))
    return final_num
```

#Create an algorithm for equation 4

def fibonacci_equation4():

#Ask the user for an input for p and check if the number given is positive and non-floating. Use a while loop so that if their input a value that doesn't work it will ask them again until they provide a valid value

```
number_checker = False
while number_checker == False:
    try:
        p = int(input("Enter a positive integer p: "))
        if p <= 0:
            print("Please enter a positive integer.")
        else:
            number_checker = True
    except ValueError:
        print("Invalid input. Please enter a positive integer.")
```

#Set a variable equal to the fibonacci number at position p

```
p_fibonacci = fibonacci_equation3(p)
```

#Ask the user for an input for n and calculate the resulting final value using equation 4 and the provided values for p and n

```
n = int(input("Enter n: "))
n_fibonacci = (p_fibonacci)*(((1 + math.sqrt(5))/(2))**(n - p))
return n_fibonacci
```

#Create an algorithm for equation 5

```
def fibonacci_equation5():
```

#Ask the user for an input for n, corresponding to the position of the desired Fibonacci number

```
n = int(input("Enter the number corresponding to the position of desired Fibonacci number: "))
```

#Find the value of the Fibonacci number one position behind specified n value using equation 3 and assign it to a variable

```
n_minus_one_fibonacci = fibonacci_equation3(n - 1)
```

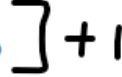
#Use the previous fibonacci number and equation 5 to find our final fibonacci number and return that value

```
n_fibonacci = ((n_minus_one_fibonacci)*((1 + math.sqrt(5))/(2)))
return n_fibonacci
```

Mathematical Analysis:

1st algorithm:

```
def fibonacci_equation3(n):  
    final_num = (((1 + math.sqrt(5))**n) -  
                 ((1 - math.sqrt(5))**n)) / ((2**n)*math.sqrt(5))  
    return final_num
```



The time function of our first algorithm is quite easy to calculate since it only consists of one line, meaning its time function would be:

$$T(n) = 1$$

The efficiency class of this algorithm can be proven using a Proof by Limits, where:

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = L$$

With the above limit and the assumption that the efficiency class is likely $O(1)$, we can derive the following formula:

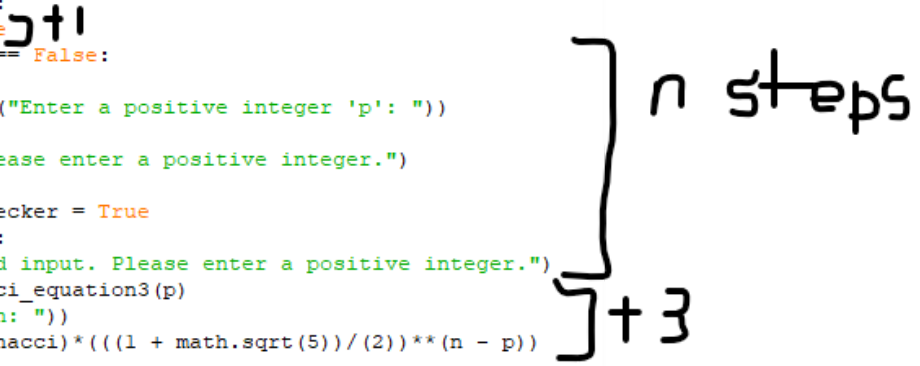
$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{1}{1} \\ &= \lim_{n \rightarrow \infty} 1 \\ &= 1 \end{aligned}$$

We can therefore conclude for our first algorithm:

$$1 \in O(1)$$

2nd algorithm:

```
def fibonacci_equation4():  
    number_checker = False  
    while number_checker == False:  
        try:  
            p = int(input("Enter a positive integer 'p': "))  
            if p <= 0:  
                print("Please enter a positive integer.")  
            else:  
                number_checker = True  
        except ValueError:  
            print("Invalid input. Please enter a positive integer.")  
    p_fibonacci = fibonacci_equation3(p)  
    n = int(input("Enter n: "))  
    n_fibonacci = (p_fibonacci)*(((1 + math.sqrt(5))/(2))**(n - p))  
    return n_fibonacci
```



Since this algorithm consists of a while loop that repeats as long as the user is inputting invalid values and 4 other lines of code, its time function would be:

$$T(n) = 1 + n + 3$$

$$T(n) = n + 4$$

The efficiency class of this algorithm can be proven using a Proof by Limits, where:

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = L$$

With the above limit and the assumption that the efficiency class is likely $O(n)$, we can derive the following formula:

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{n + 4}{n} \\ &= \lim_{n \rightarrow \infty} \frac{n}{n} + \frac{4}{n} \\ &= \lim_{n \rightarrow \infty} \frac{n}{n} + \lim_{n \rightarrow \infty} \frac{4}{n} \\ &= \lim_{n \rightarrow \infty} \frac{n}{n} \\ &= \lim_{n \rightarrow \infty} 1 \\ &= 1 \end{aligned}$$

We can therefore conclude for our second algorithm:

$$n + 4 \in O(n)$$

3rd Algorithm:

```
def fibonacci_equation5():
    n = int(input("Enter the number corresponding to the position desired Fibonacci number: "))
    n_minus_one_fibonacci = fibonacci_equation3(n - 1)
    n_fibonacci = ((n_minus_one_fibonacci)*((1 + math.sqrt(5))/(2)))
    return n_fibonacci
```

Since this algorithm only consists of three lines of code and no loops, it's time function would be:

$$T(n) = 3$$

The efficiency class of this algorithm can be proven using a Proof by Limits, where:

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = L$$

With the above limit and the assumption that the efficiency class is likely $O(1)$, we can derive the following formula:

$$\lim_{n \rightarrow \infty} 3$$

$$= 3$$

We can therefore conclude for our third algorithm:

$$3 \in O(1)$$

a) Completed algorithms:

```
import math

def fibonacci_equation3(n):
    final_num = (((1 + math.sqrt(5))**n) -
                  ((1 - math.sqrt(5))**n))/((2**n)*math.sqrt(5))
    return final_num

def fibonacci_equation4():
    number_checker = False
    while number_checker == False:
        try:
            p = int(input("Enter a positive integer 'p': "))
            if p <= 0:
                print("Please enter a positive integer.")
            else:
                number_checker = True
        except ValueError:
            print("Invalid input. Please enter a positive integer.")
    p_fibonacci = fibonacci_equation3(p)
    n = int(input("Enter n: "))
    n_fibonacci = (p_fibonacci)*((1 + math.sqrt(5))/(2))**(n - p)
    return n_fibonacci

def fibonacci_equation5():
    n = int(input("Enter the number corresponding to the position desired Fibonacci number: "))
    n_minus_one_fibonacci = fibonacci_equation3(n - 1)
    n_fibonacci = ((n_minus_one_fibonacci)*((1 + math.sqrt(5))/(2)))
    return n_fibonacci
```

b)

To print out the first 20 terms of the sequences created by equations 4 and 5, we first created versions of our earlier algorithms for equations 4 and 5 that did not require user input as we wanted to use a for loop in a later algorithm that would print out the values. We then created the algorithm that we would use to print out the first 20 values, starting by creating two lists that we would append values to and creating said for loop that would iterate from 0 - 19. We also made sure to include if statements for specific cases such as when $i = 0$ as we didn't want the program to be using negative values. The code and final outputs are shown below:

```
def no_user_input_fibonacci_equation4(p,n):
    p_fibonacci = fibonacci_equation3(p)
    n_fibonacci = (p_fibonacci)*((1 + math.sqrt(5))/(2))**(n - p))
    return n_fibonacci

def no_user_input_fibonacci_equation5(n):
    n_minus_one_fibonacci = fibonacci_equation3(n - 1)
    n_fibonacci = ((n_minus_one_fibonacci)*((1 + math.sqrt(5))/(2)))
    return n_fibonacci

def first_20_numbers():
    list_of_equation4 = []
    list_of_equation5 = []
    for i in range(20):
        p = i - 1
        if i == 0:
            p = 0
        q = i
        if i == 0:
            q = 1
        list_of_equation4.append(no_user_input_fibonacci_equation4(p,i))
        list_of_equation5.append(no_user_input_fibonacci_equation5(q))
    print("First 20 results of equation 4: ")
    print(list_of_equation4)
    print("First 20 results of equation 5: ")
    print(list_of_equation5)
    return

>>> first_20_numbers()
First 20 results of equation 4:
[0.0, 0.0, 1.618033988749895, 1.618033988749895, 3.23606797749979, 4.854101966249686, 8.090169943749476,
12.944271909999163, 21.034441853748636, 33.9787137637478, 55.01315561749644, 88.99186938124424, 144.0050
2499874068, 232.99689437998495, 377.00191937872563, 609.9988137587106, 987.0007331374364, 1596.999546896
147, 2584.0002800335837, 4180.999826929731]
First 20 results of equation 5:
[0.0, 0.0, 1.618033988749895, 1.618033988749895, 3.23606797749979, 4.854101966249686, 8.090169943749476,
12.944271909999163, 21.034441853748636, 33.9787137637478, 55.01315561749644, 88.99186938124424, 144.0050
2499874068, 232.99689437998495, 377.00191937872563, 609.9988137587106, 987.0007331374364, 1596.999546896
147, 2584.0002800335837, 4180.999826929731]
...
```

c)

Since both of our sequences for equation 4 and 5 are exactly the same, the differences between the two would be 0. However, this is due to the fact that we set $p = i - 1$, meaning that equation 4 essentially calculated the Fibonacci sequence the same way equation 5 would and that is why the sequences ended up identical. If one were to use different values of p , they would end up

with a very different set of values, as demonstrated below with two different values of p and their resulting sequences:

If $p = 1$:

```
First 20 results of equation 4:
[0.0, 1.0, 1.618033988749895, 2.618033988749895, 4.23606797749979, 6.854101966249686, 11.090169943749476,
17.944271909999163, 29.03444185374864, 46.978713763747805, 76.01315561749645, 122.99186938124426, 199.0050249987407,
321.996894379985, 521.0019193787257, 842.9988137587108, 1364.0007331374366, 2206.9995468961474,
3571.000280033584, 5777.999826929732]
```

If $p = 5$:

```
First 20 results of equation 4:
[0.0, 0.7294901687515772, 1.1803398874989486, 1.9098300562505262, 3.0901699437494745, 5.000000000000001,
8.090169943749476, 13.090169943749476, 21.180339887498953, 34.27050983124843, 55.45084971874739, 89.72135954999582,
145.17220926874322, 234.89356881873906, 380.06577808748233, 614.9593469062214, 995.0251249937037,
1609.9844718999254, 2605.009596893629, 4214.994068793554]
```

d)

```
>>> fibonacci_equation5()
Enter the number corresponding to the position desired Fibonacci number: 2
1.618033988749895
>>> fibonacci_equation5()
Enter the number corresponding to the position desired Fibonacci number: 3
1.618033988749895
>>> 1.618033988749895/1.618033988749895
1.0
>>> fibonacci_equation5()
Enter the number corresponding to the position desired Fibonacci number: 29
514228.9999985933
>>> fibonacci_equation5()
Enter the number corresponding to the position desired Fibonacci number: 30
832040.0000008704
>>> 832040.0000008704/514228.9999985933
1.6180339887543225
```

As shown in the image above, the ratio of F_3/F_2 when using equation 5 would only be 1, however the ratio of F_{30}/F_{29} when using equation 5 would be 1.6180339887543225, exactly the golden ratio and proving the maxim that as n increases the ratio of two consecutive Fibonacci numbers approaches the golden ratio.