Assignment 5

Dynamic Programming

Task 1 Dynamic programming

Your friend Jesse has made the following code for calculating the n^{th} Fibonacci number:

```
1 def fib(n: int) -> int:
2     if n <= 1:
3         return n
4     return fib(n - 2) + fib(n - 1)
```

He asks you for help optimizing it, and you decide to analyze the code by drawing up a tree of the recursive calls when you call `fib(5)`

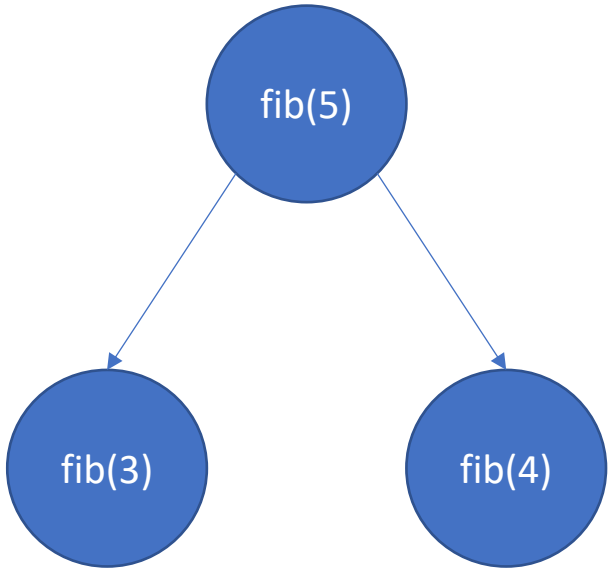
a) Draw the tree and use it to analyze the runtime of the recursion, do you notice a pattern in the tree?


```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```

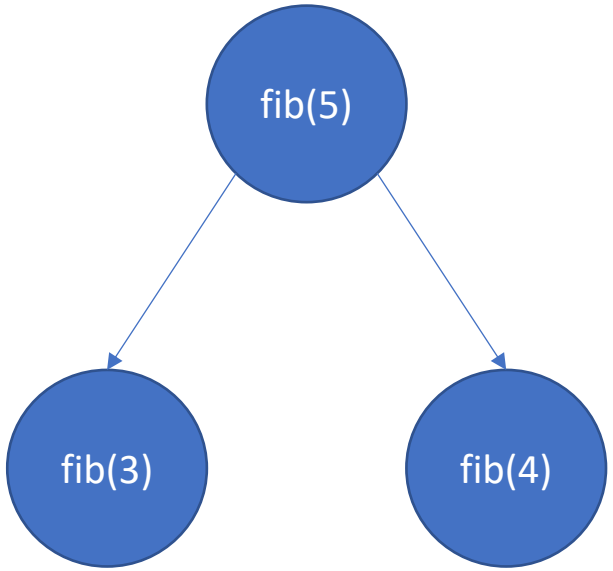

```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```


```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```

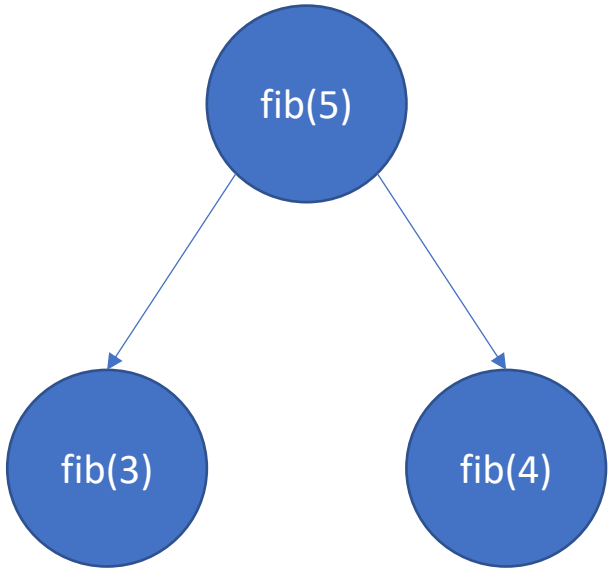
```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```



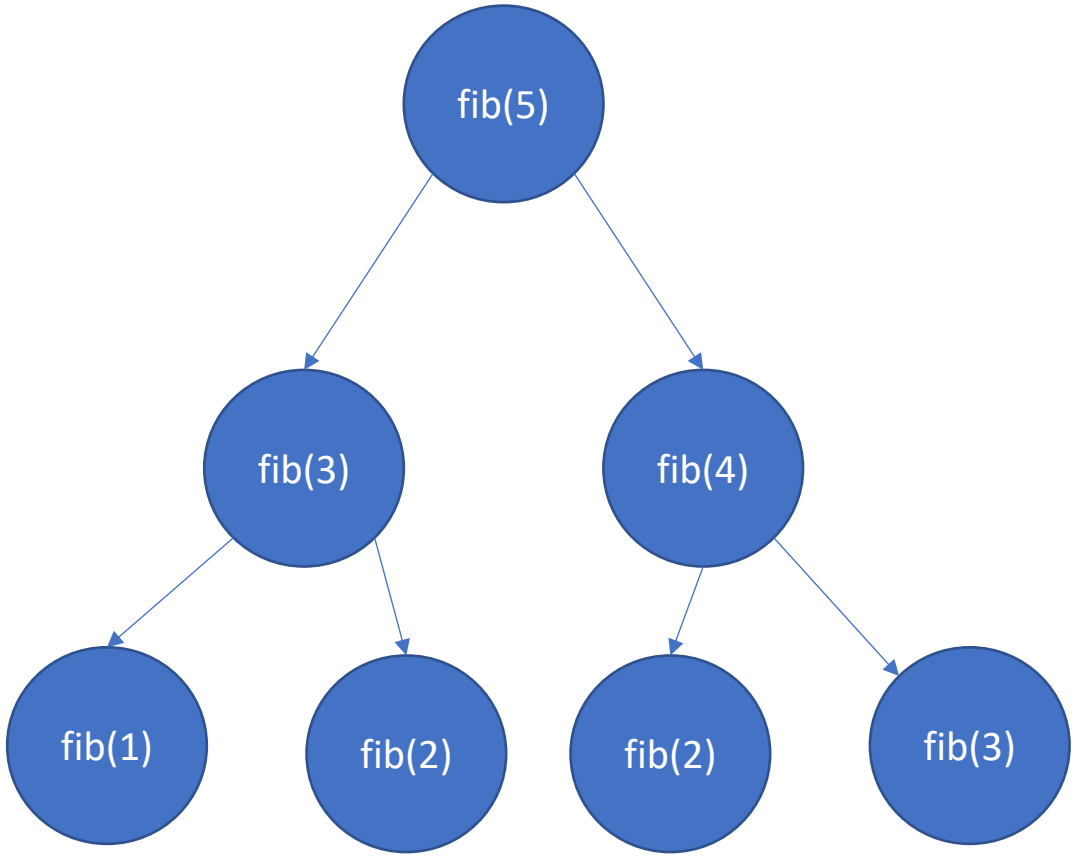
```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```



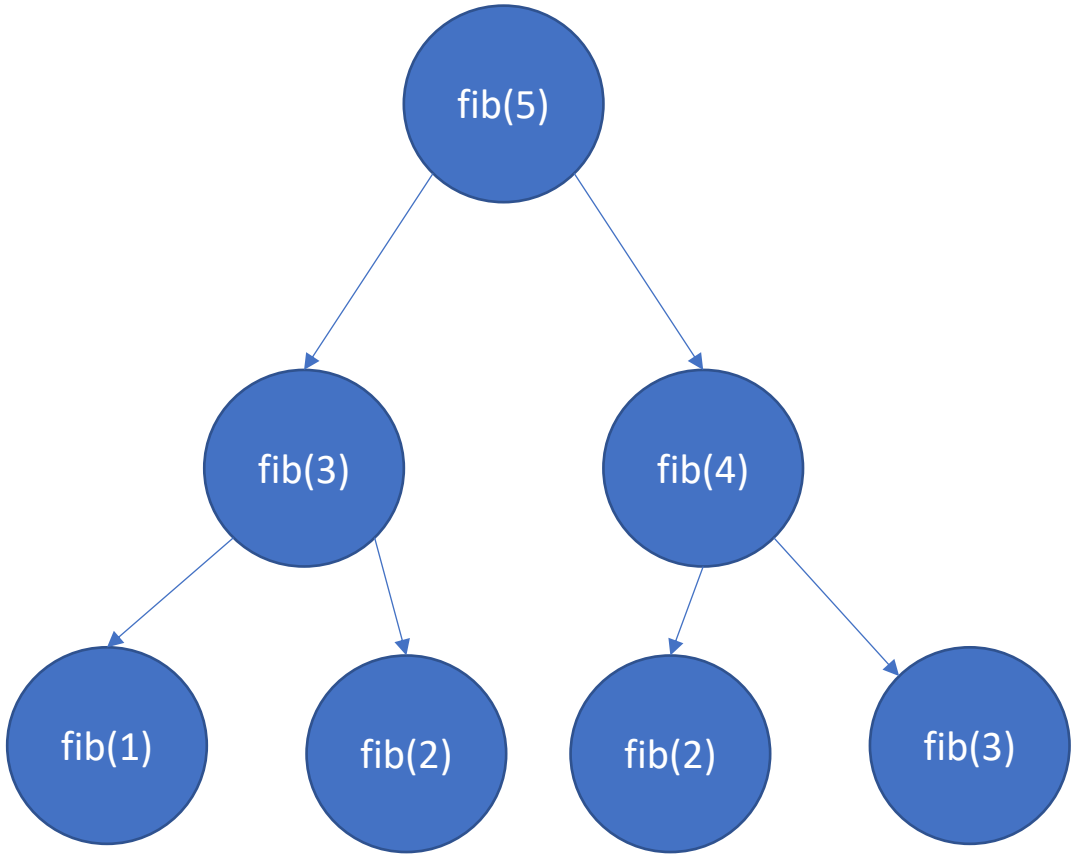
```python
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```



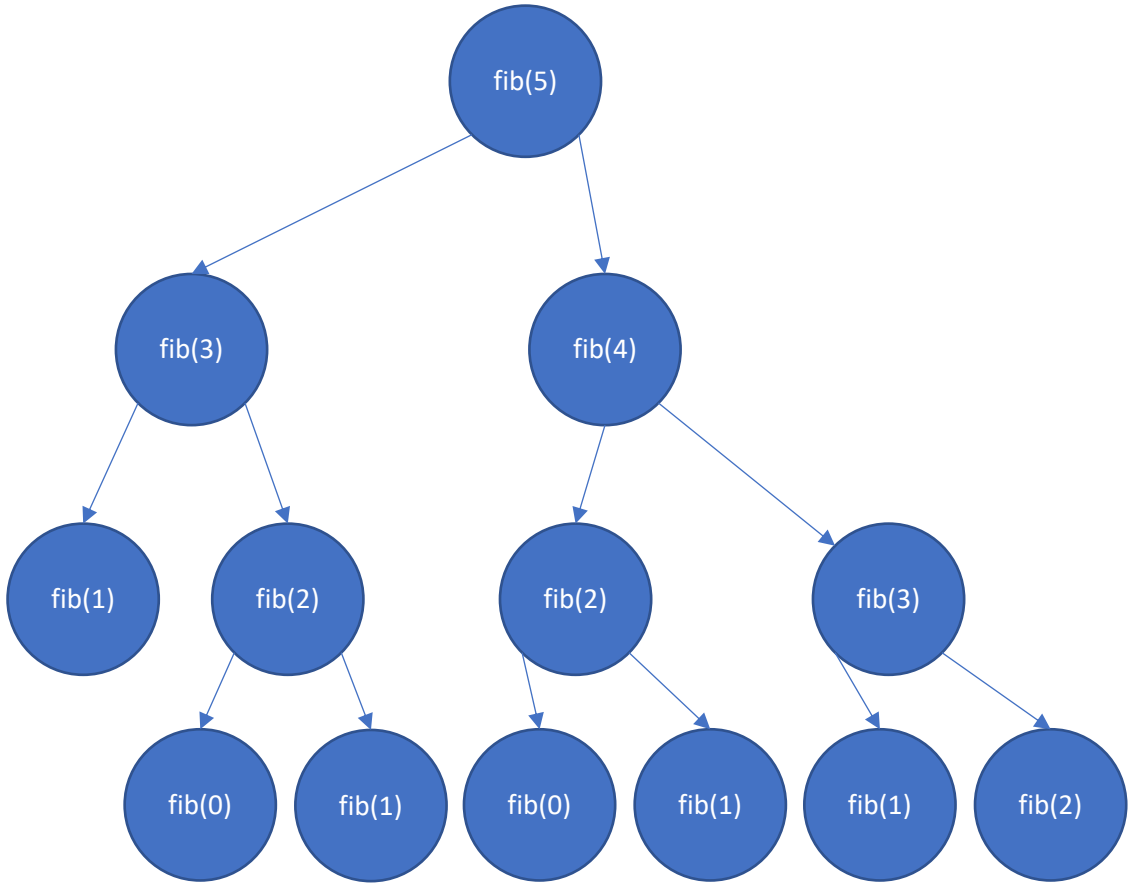
```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```



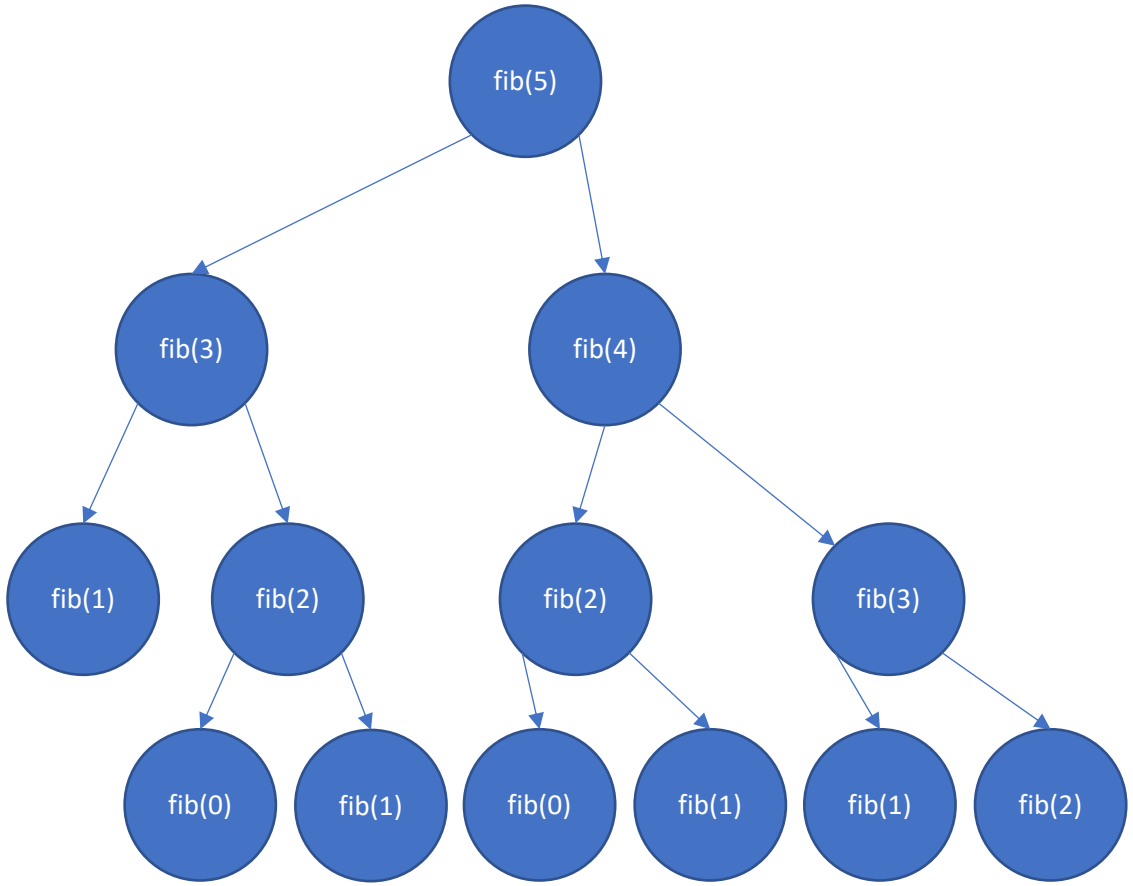
```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```



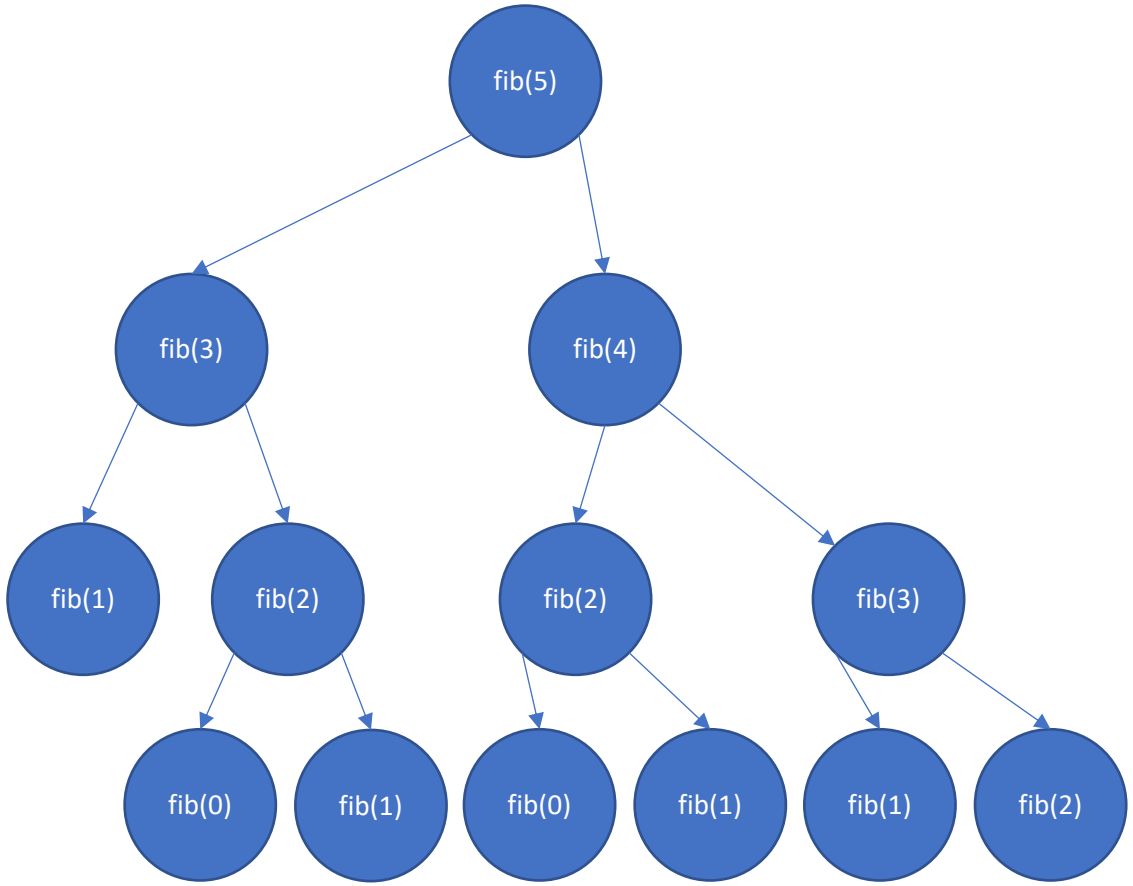
```python
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```



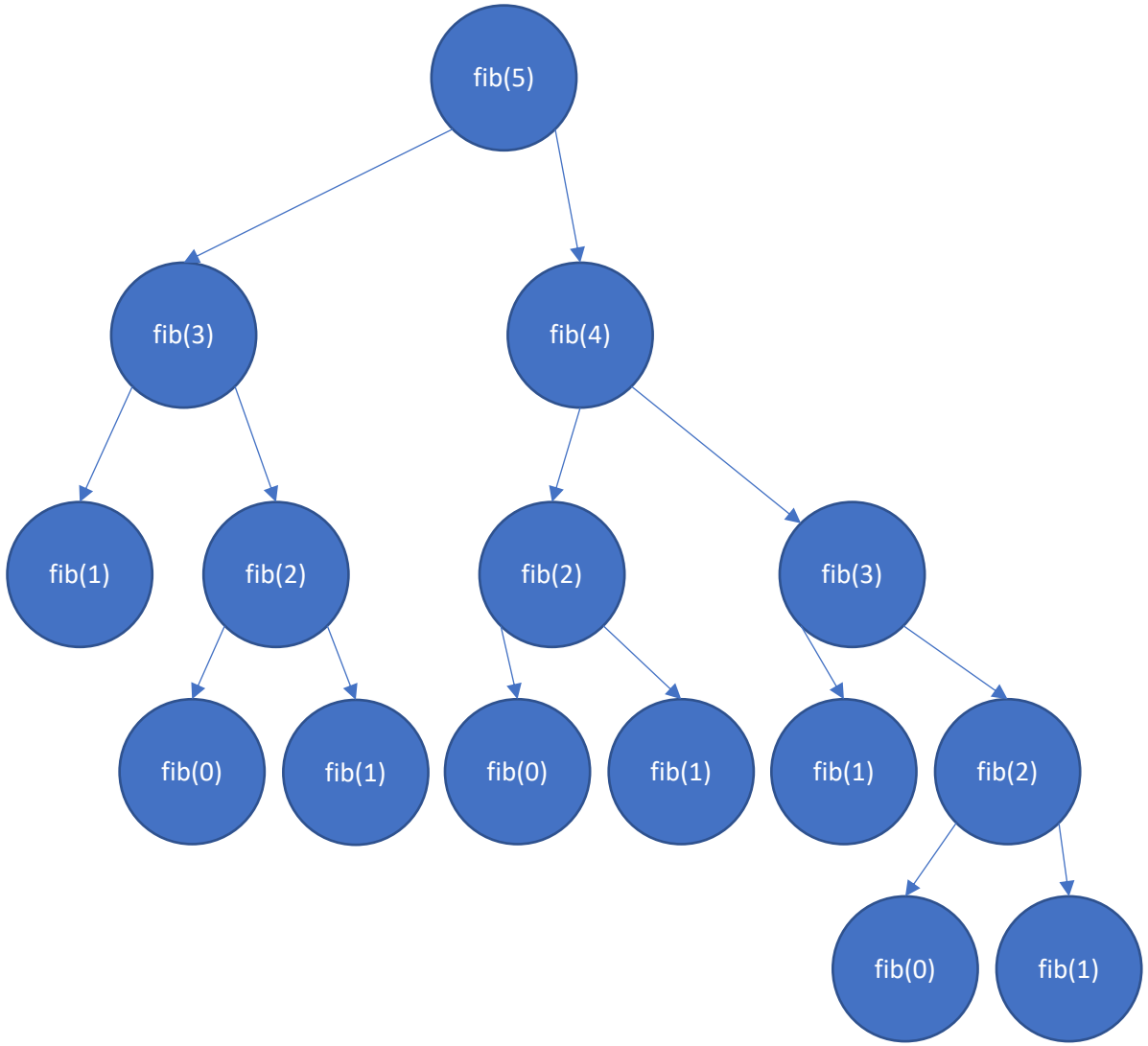
```python
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```



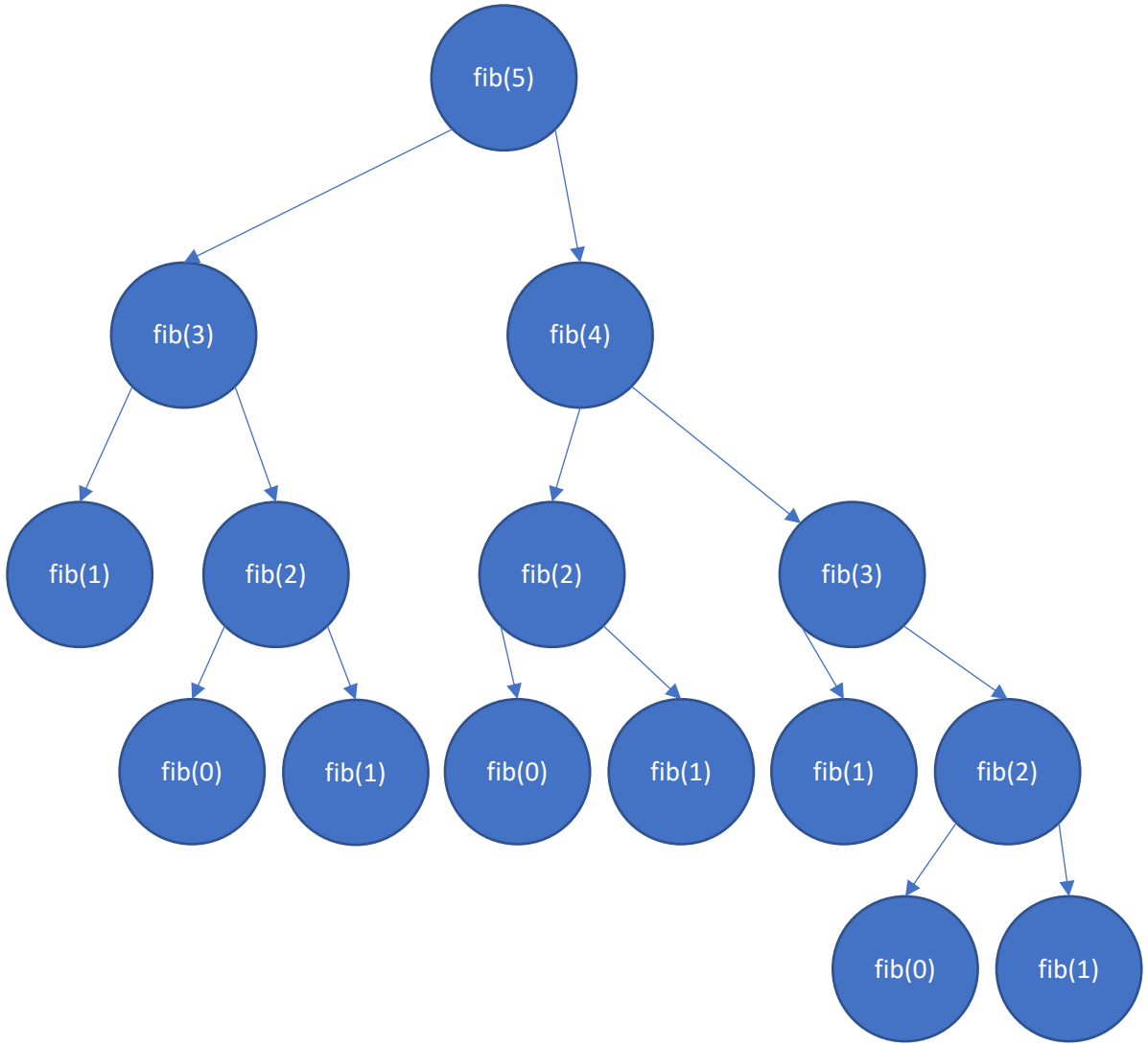
```python
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```



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

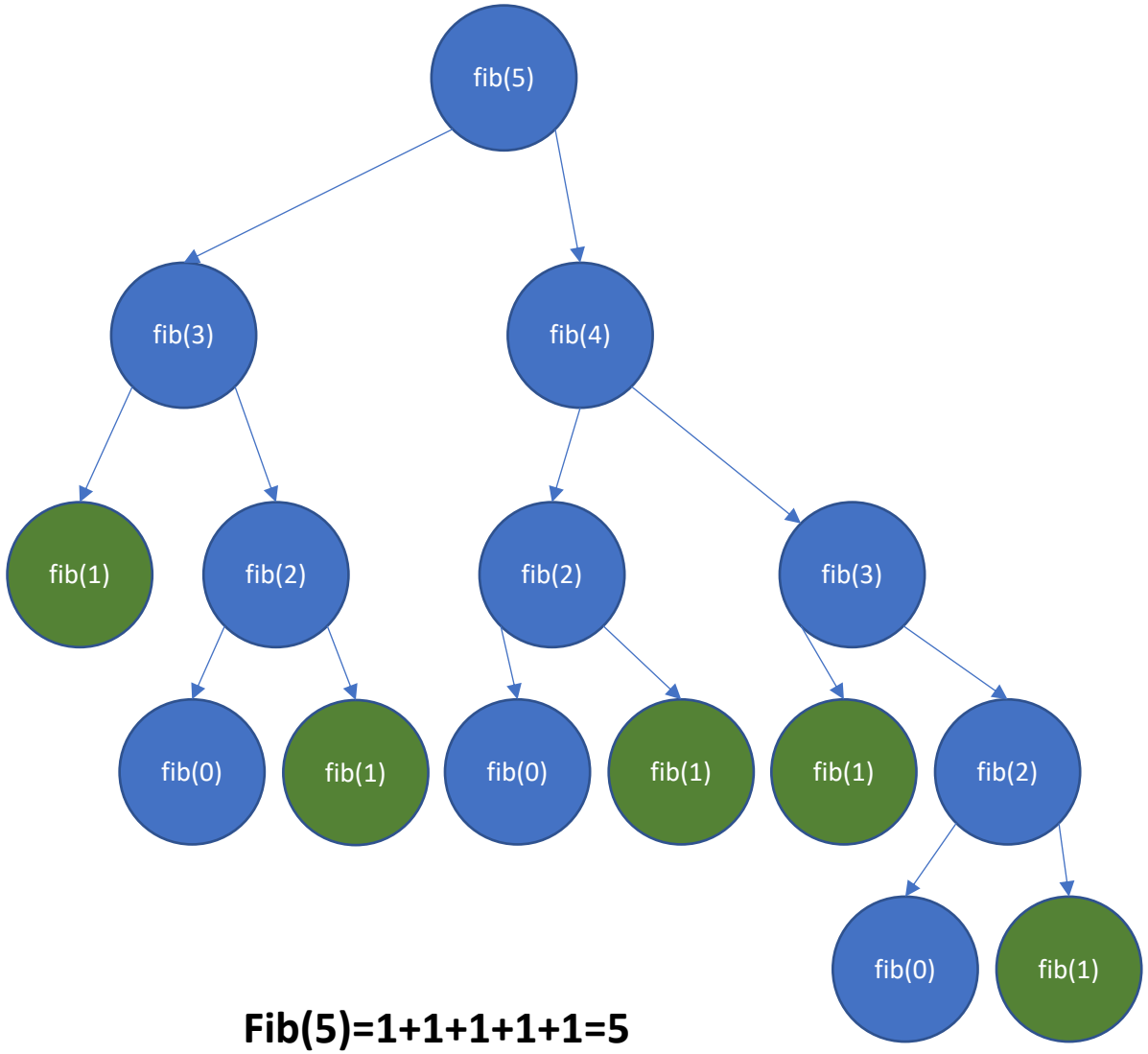


```
1  def fib(n: int) -> int:
2      if n <= 1:
3          return n
4      return fib(n - 2) + fib(n - 1)
```



→ The number of nodes will increase by 2^n → The runtime is $O(2^n)$

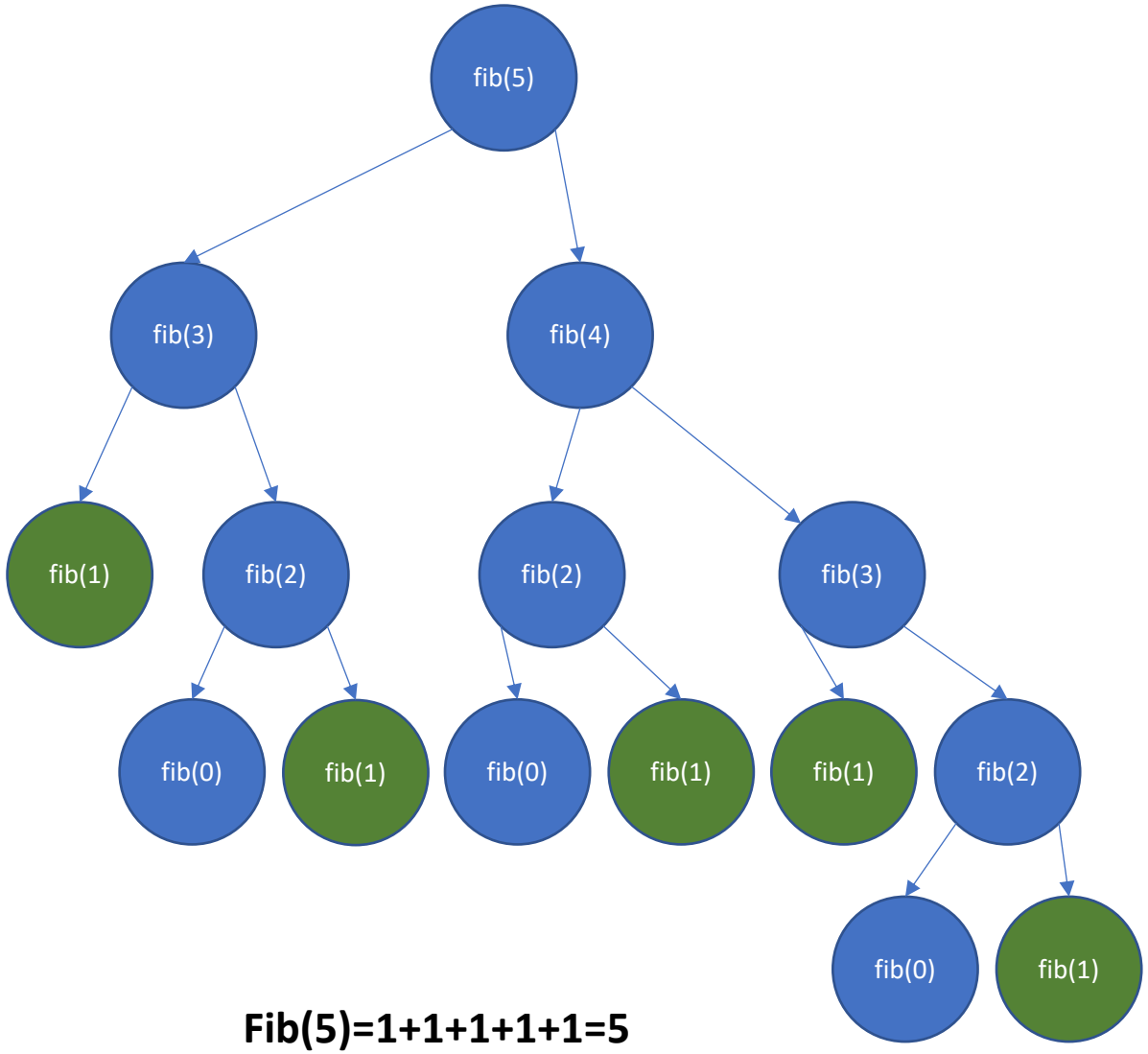
```python
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```



Fib(5)=1+1+1+1+1=5

→ The number of nodes will increase by 2^n → The runtime is $O(2^n)$

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



Fib(5)=1+1+1+1+1=5

→ The number of nodes will increase by 2^n

Part 1 Theory

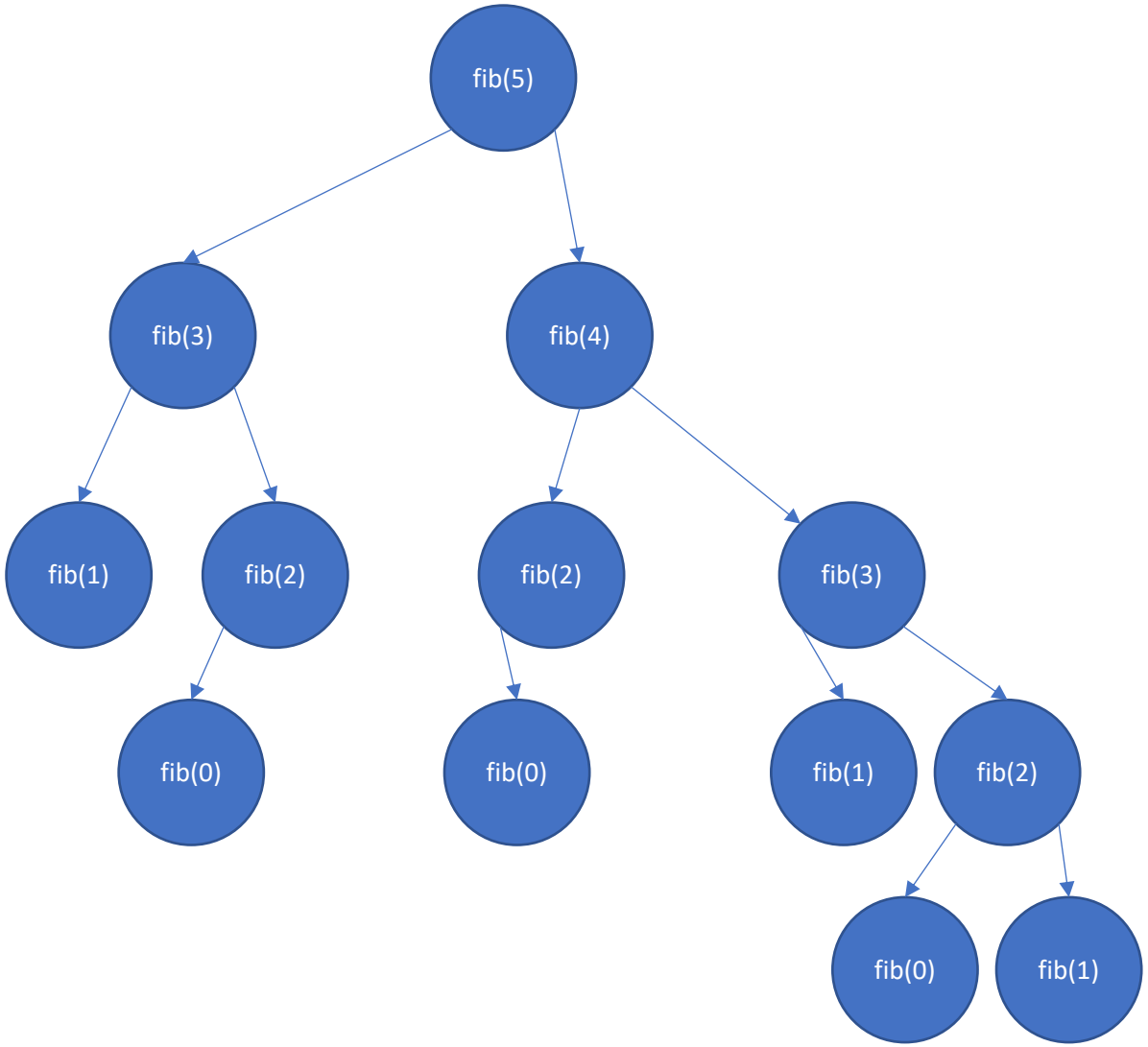
Task 1 Dynamic programming

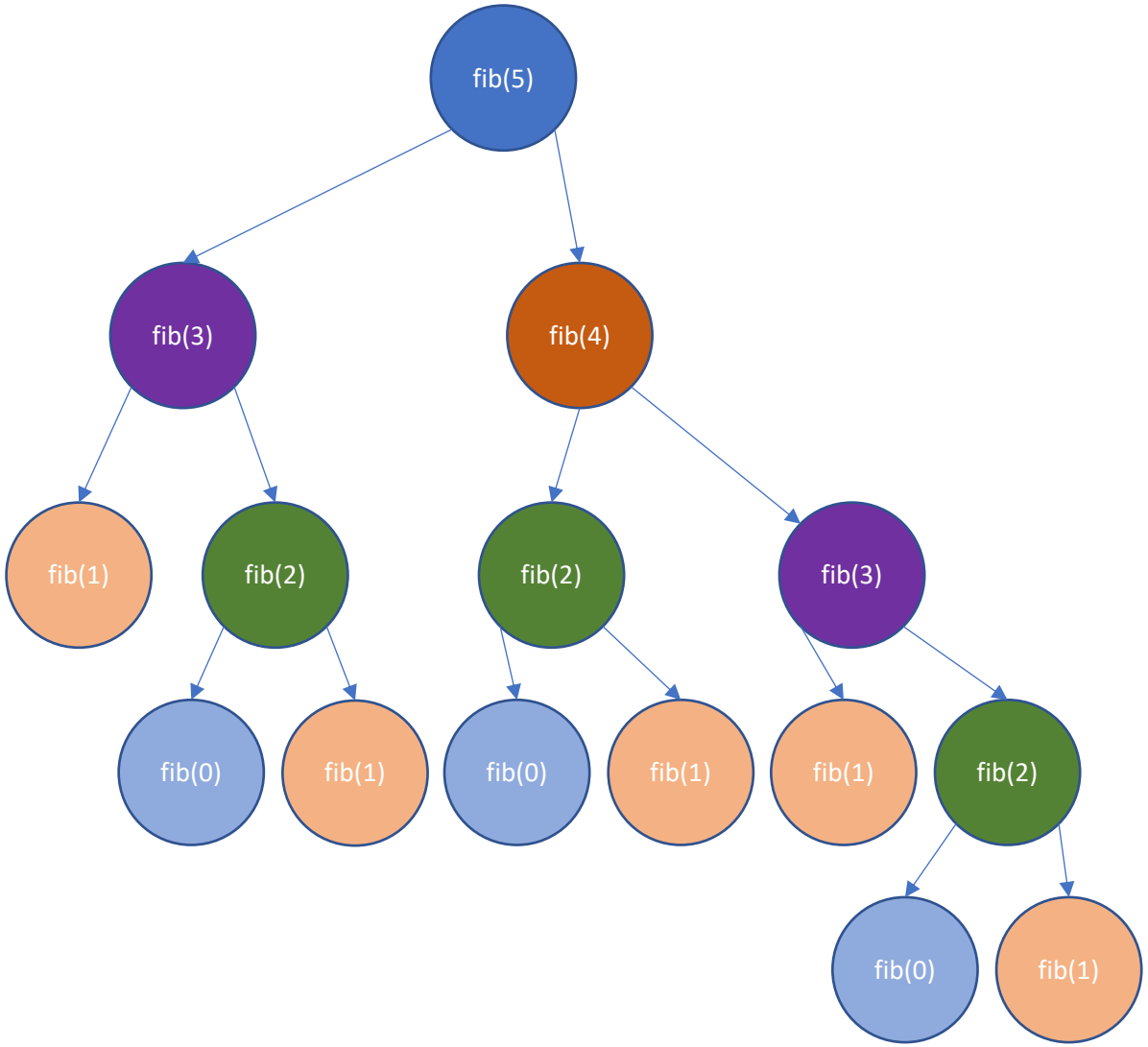
Your friend Jesse has made the following code for calculating the n^{th} Fibonacci number:

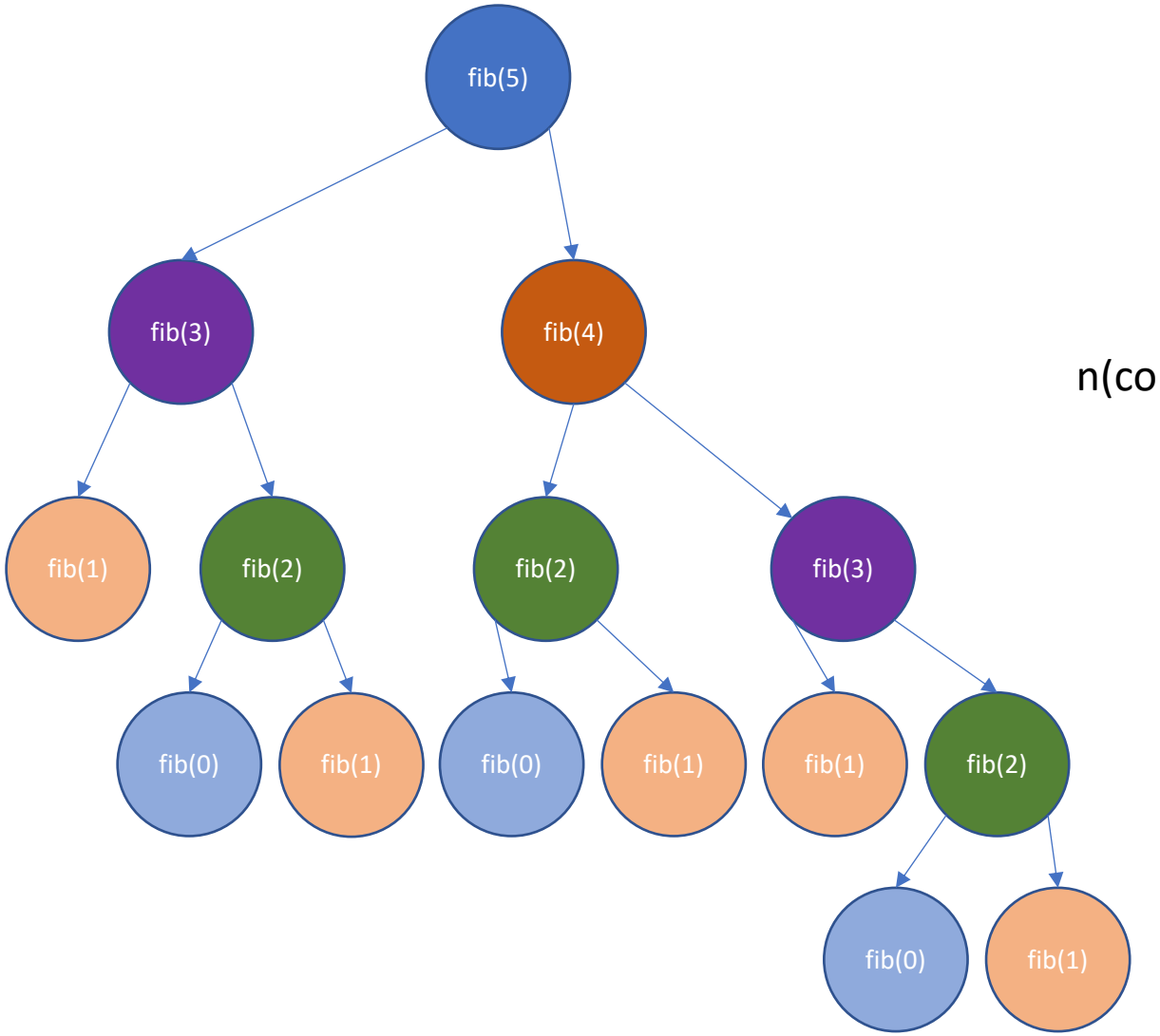
```
def fib(n: int) -> int:
    if n <= 1:
        return n
    return fib(n - 2) + fib(n - 1)
```

He asks you for help optimizing it, and you decide to analyze the code by drawing up a tree of the recursive calls when you call `fib(5)`

b) You decide to help Jesse. Based on your observations from task a), describe an algorithm that solves the same problem in linear time


fib(5)
fib(3)
fib(4)
fib(1)
fib(2)
fib(2)
fib(3)
fib(0)
fib(0)
fib(1)
fib(2)
fib(0)
fib(1)


fib(5)
fib(3)
fib(4)
fib(1)
fib(2)
fib(2)
fib(3)
fib(0)
fib(1)
fib(0)
fib(1)
fib(1)
fib(2)
fib(0)
fib(1)


fib(5)
fib(3)
fib(4)
fib(1)
fib(2)
fib(2)
fib(3)
fib(0)
fib(1)
fib(0)
fib(1)
fib(1)
fib(2)
fib(0)
fib(1)

n(colours)=5

Part 1 Theory

Task 1 Dynamic programming

Your friend Jesse has made the following code for calculating the n^{th} Fibonacci number:

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

He asks you for help optimizing it, and you decide to analyze the code by drawing up a tree of the recursive calls when you call `fib(5)`

b) You decide to help Jesse. Based on your observations from task a), describe an algorithm that solves the same problem in linear time

Solution:
Since calls like `fib(1)`, `fib(2)`, etc. , are repeated. We should save the results in a memoization table while running the algorithm. So instead of instantly calling the function recursively, we should check the table. And when a call returns, it should also save its return value to the table. Since reading to and from the table takes constant time, we can make this algorithm run in linear time $O(n)$, as we have to call all n numbers, but only once, since we get them from the memoization table the second time around.

This solution is an example of a memoization (top-down) approach, but a bottom-up approach is also accepted here. The point is that the student should understand the concept of dynamic programming.

Part 1 Theory

Task 1 Dynamic programming

Your friend Jesse has made the following code for calculating the n^{th} Fibonacci number:

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

He asks you for help optimizing it, and you decide to analyze the code by drawing up a tree of the recursive calls when you call `fib(5)`

c) What properties must a problem have for it to be able to be solved with dynamic programming?

Part 1 Theory

Task 1 Dynamic programming

Your friend Jesse has made the following code for calculating the n^{th} Fibonacci number:

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

He asks you for help optimizing it, and you decide to analyze the code by drawing up a tree of the recursive calls when you call `fib(5)`

c) What properties must a problem have for it to be able to be solved with dynamic programming?

Solution:
The problem needs to have a collection of subproblems that overlap (think of how the problems overlap in the Fibonacci algorithm above). The solution can be calculated from the solution to the subproblems. There is a natural order from smalles to largest, that allows us to determine the soluton to a subproblem.

Task 2 Smoothie algorithm

Your friend Walter has made an algorithm based on dynamic programming. The algorithm makes the best-tasting smoothie based on the fruit and vegetables in Walter's pantry. You look at the algorithm and observe that it only calculates and returns the taste levels of the optimal smoothie. But it does not give the actual ingredients. Walter argues it's a trivial difference, and adding said functionality would be easy. What do you think?

Task 2 Smoothie algorithm

Your friend Walter has made an algorithm based on dynamic programming. The algorithm makes the best-tasting smoothie based on the fruit and vegetables in Walter's pantry. You look at the algorithm and observe that it only calculates and returns the taste levels of the optimal smoothie. But it does not give the actual ingredients. Walter argues it's a trivial difference, and adding said functionality would be easy. What do you think?

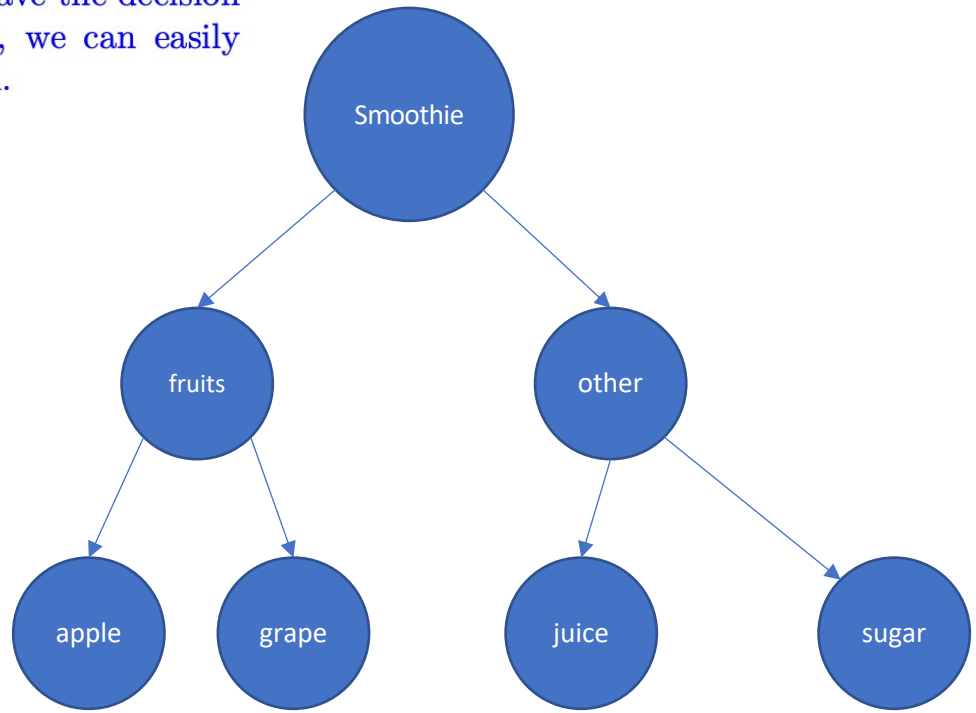
Solution:
Walter is right. Dynamic programming bases itself on a series of decisions. If we save the decision made for each subproblem (i.e. the ingredient used for the current smoothie), we can easily reconstruct the ingredient list without it affecting the complexity of the algorithm.

Task 2 Smoothie algorithm

Your friend Walter has made an algorithm based on dynamic programming. The algorithm makes the best-tasting smoothie based on the fruit and vegetables in Walter's pantry. You look at the algorithm and observe that it only calculates and returns the taste levels of the optimal smoothie. But it does not give the actual ingredients. Walter argues it's a trivial difference, and adding said functionality would be easy. What do you think?

Solution:
Walter is right. Dynamic programming bases itself on a series of decisions. If we save the decision made for each subproblem (i.e. the ingredient used for the current smoothie), we can easily reconstruct the ingredient list without it affecting the complexity of the algorithm.

Save this -->



Task 3 Shortest-path

Bellman-Ford and Dijkstra's algorithms are both algorithms for finding the shortest path in a graph. Explain their differences, in both the problem they solve and how they solve it.

Task 3 Shortest-path

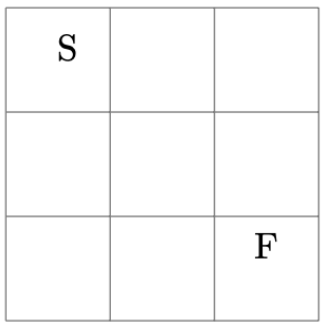
Bellman-Ford and Dijkstra's algorithms are both algorithms for finding the shortest path in a graph. Explain their differences, in both the problem they solve and how they solve it.

Solution:
Both algorithms solve the shortest path problem on a weighted directed graph, meaning the shortest (lowest weight) path from one vertice to every other vertice. Dijkstra's algorithm only works when there are no negative-weight edges in the graph. Bellman-ford works when there are negative edges, but no negative cycles in the graph.

Task 4 Grid traveling

a) Given a $n \times m$ grid where you are only allowed to move to the right or downwards from one cell to another. In how many ways can you travel from the top left corner (S) to the bottom right corner(F) when the grid has the following sizes:

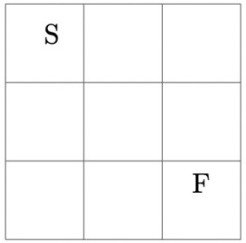
i. 3×3 Grid



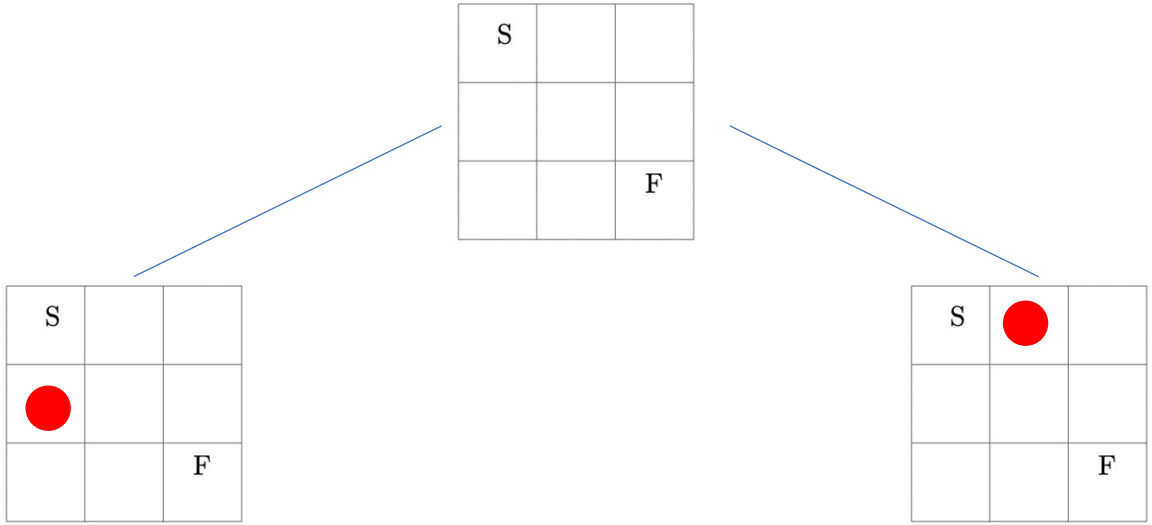
ii. 2×3 Grid

iii. 1×3 Grid

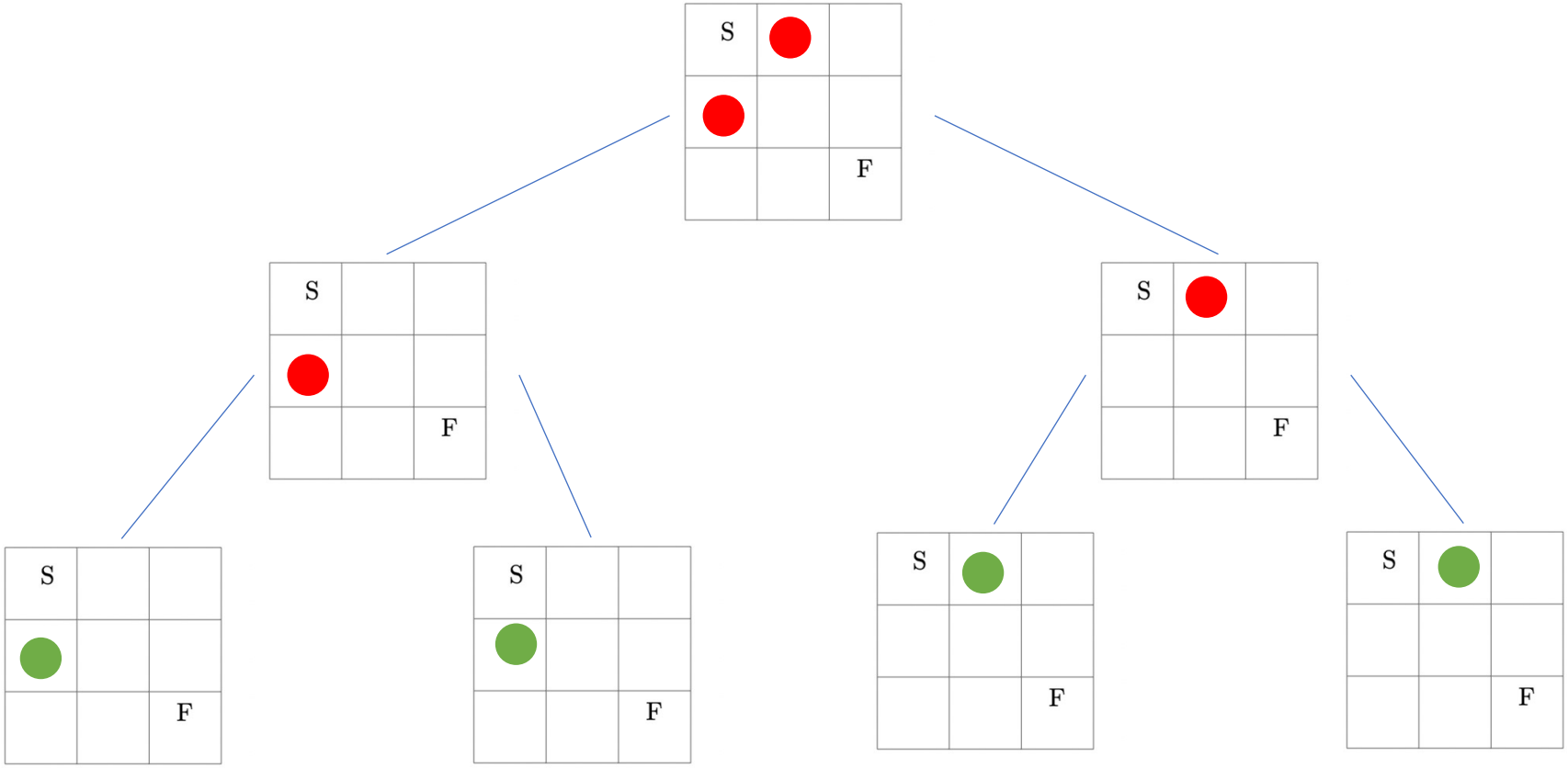
i) 3x3



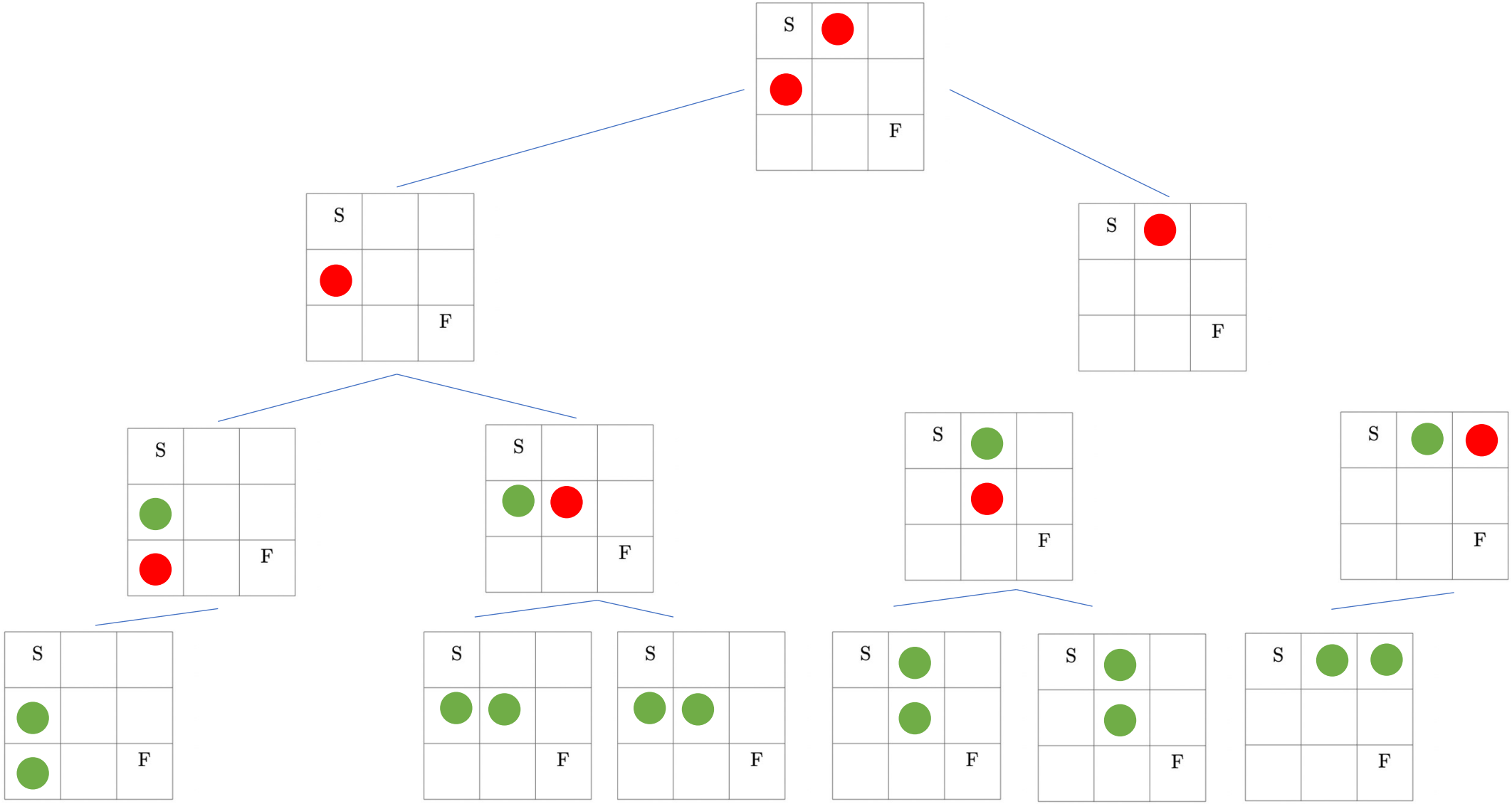
i) 3x3



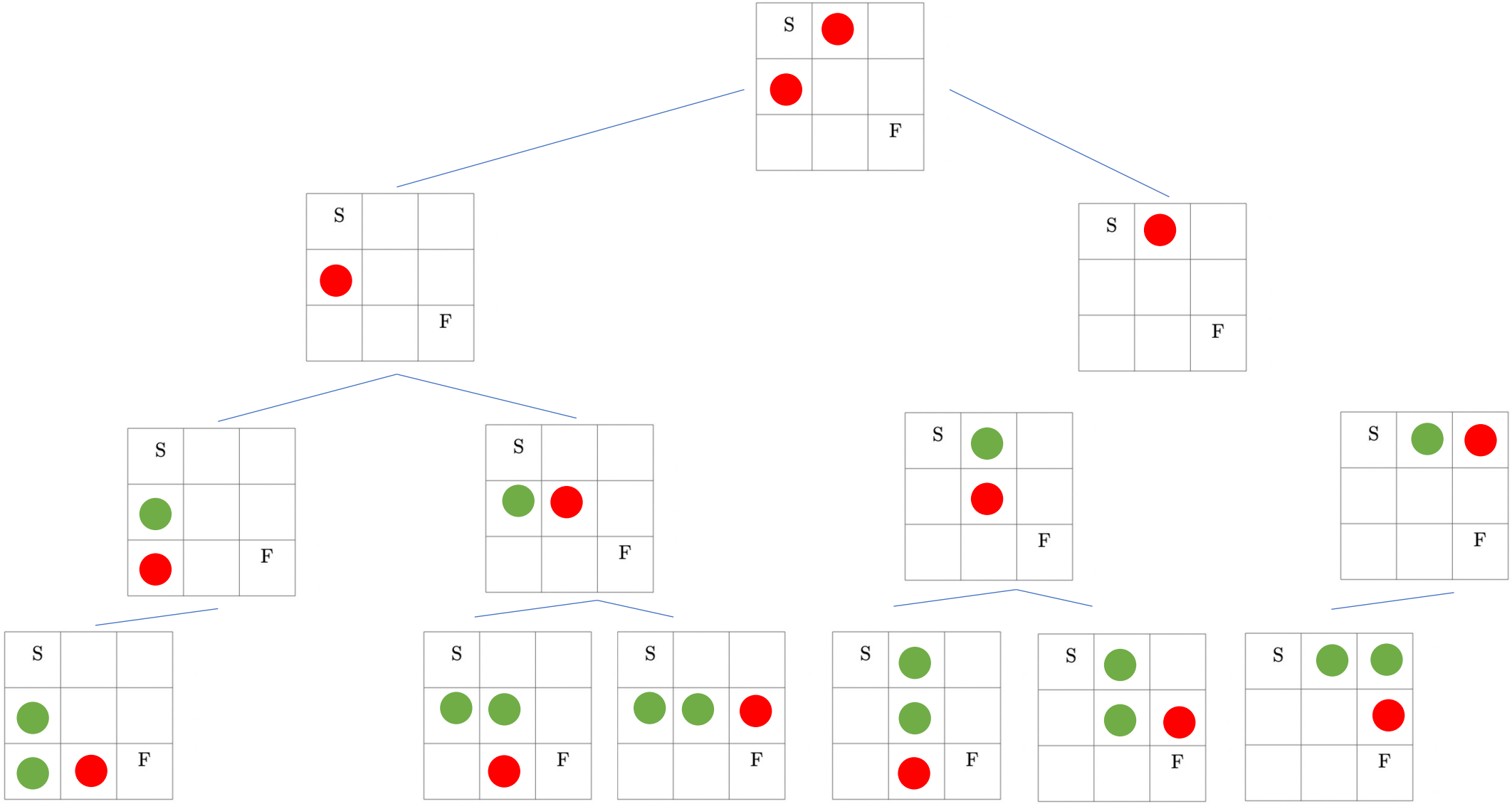
i) 3x3



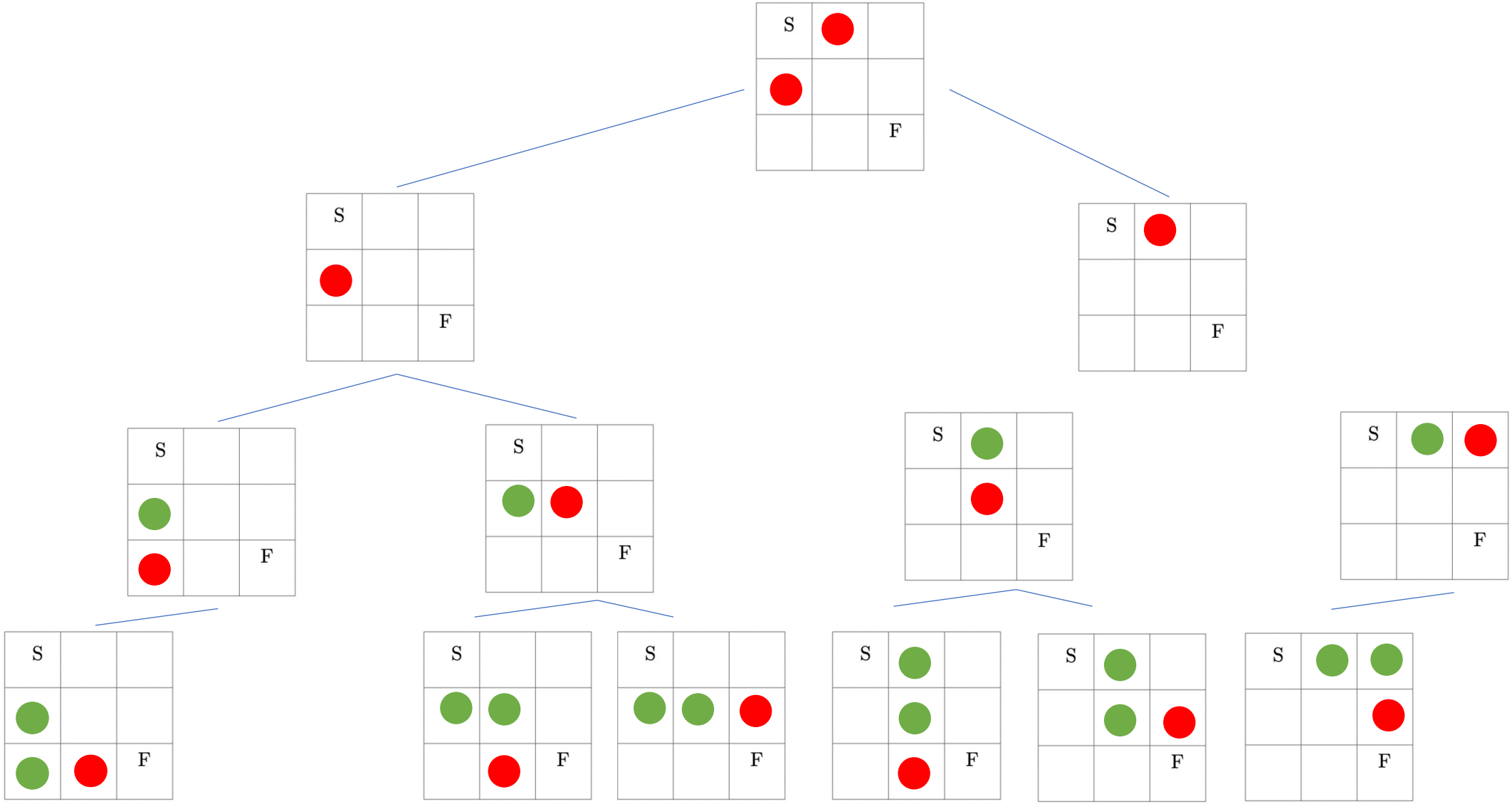
i) 3x3



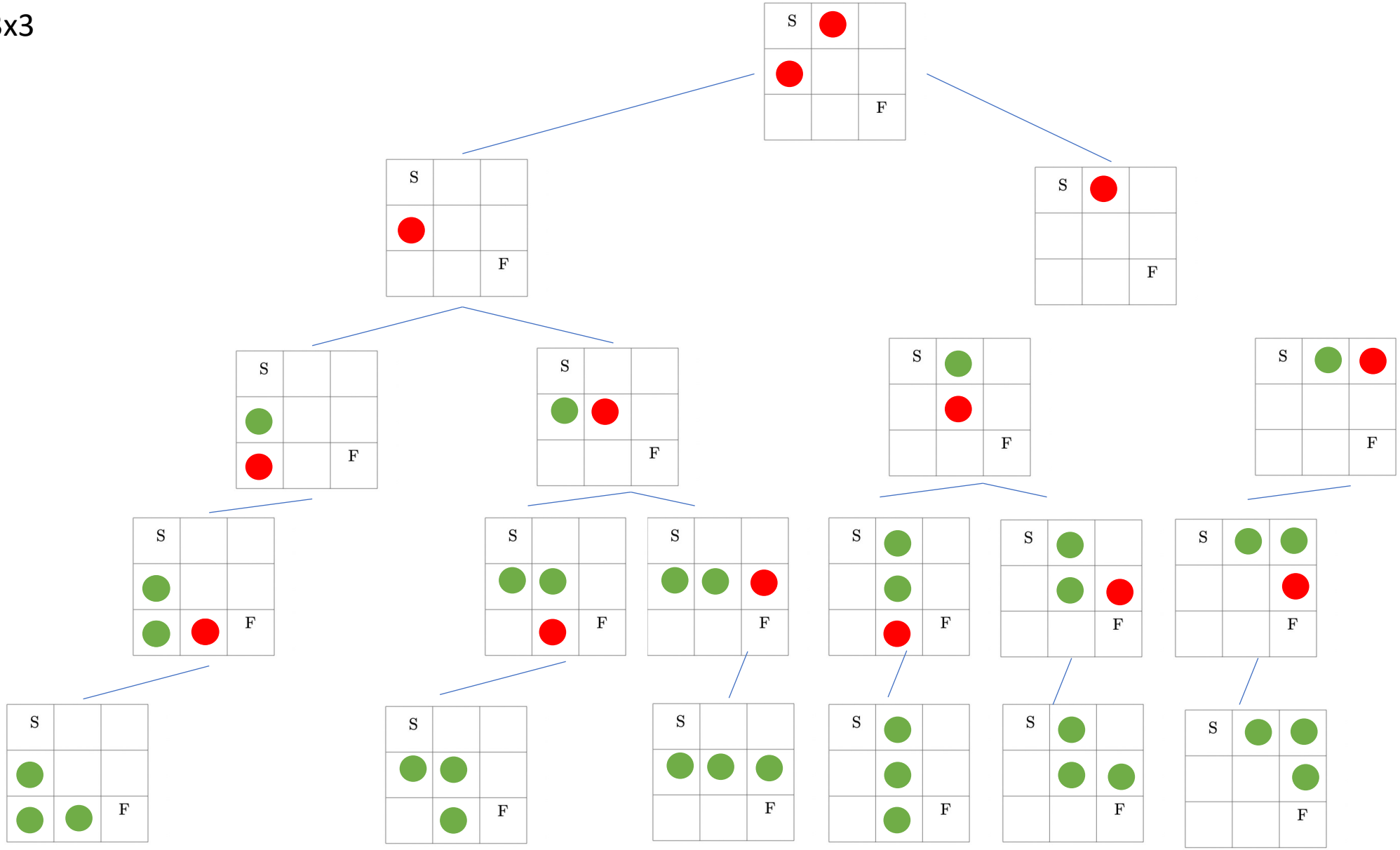
i) 3x3



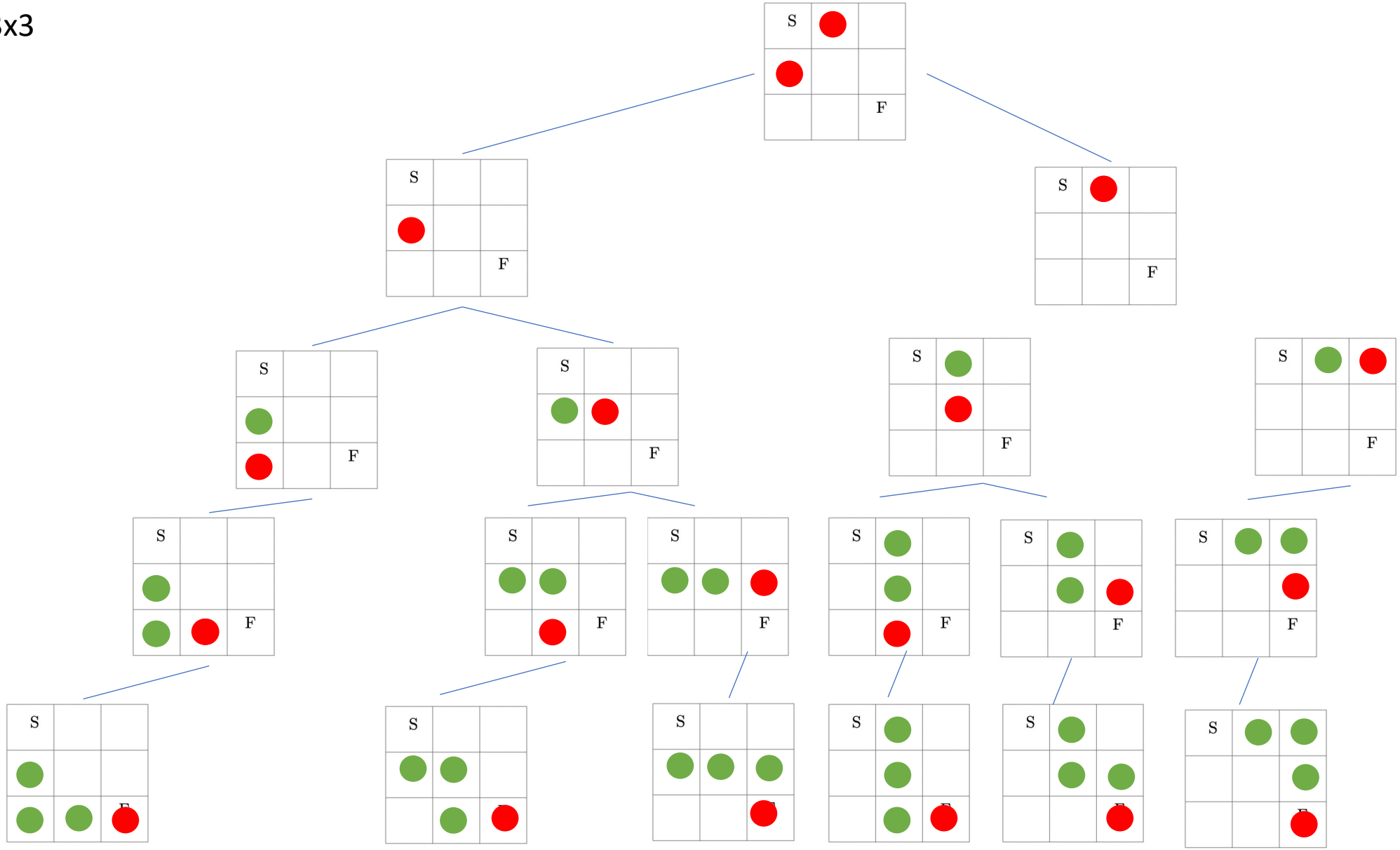
i) 3x3



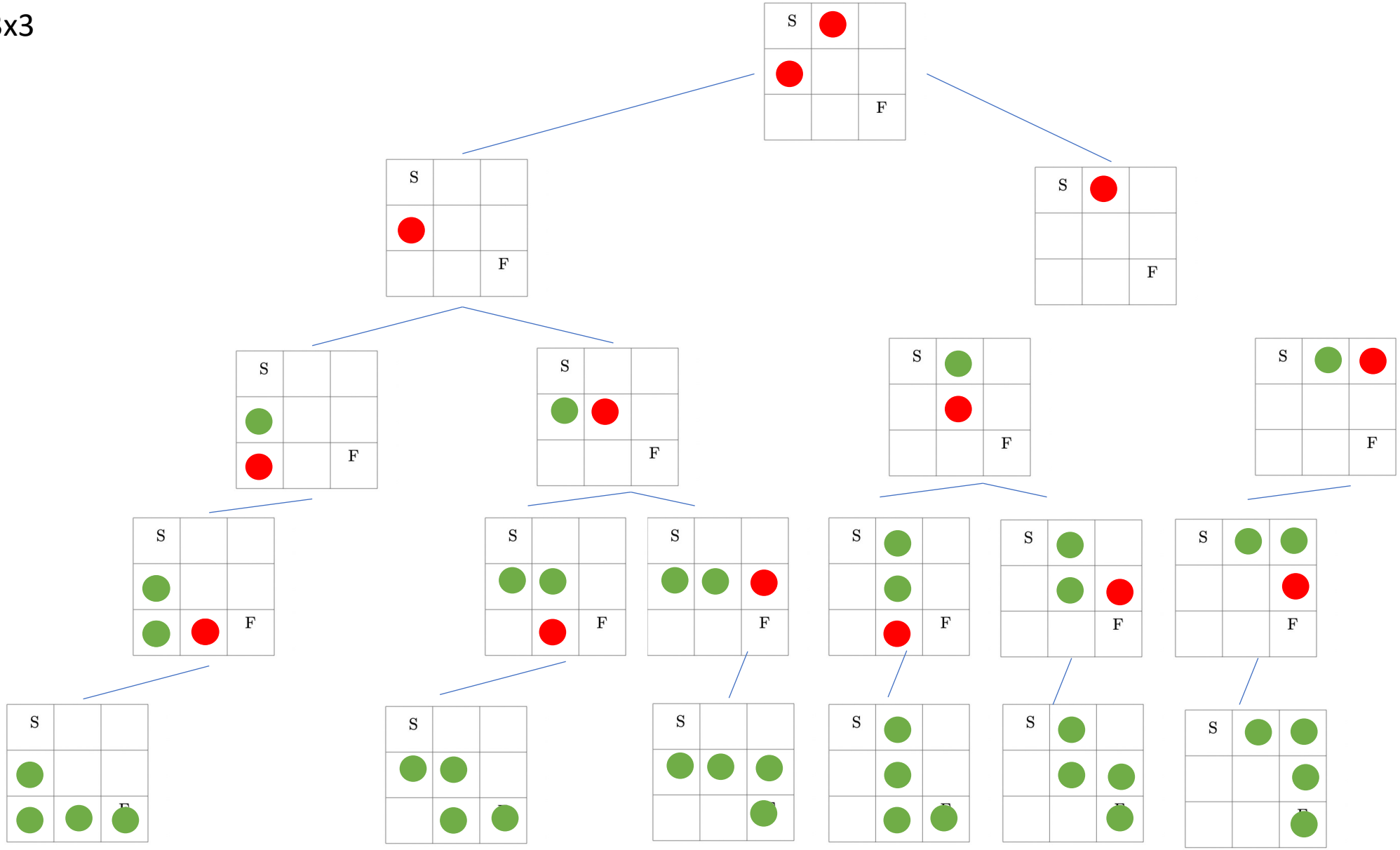
i) 3x3



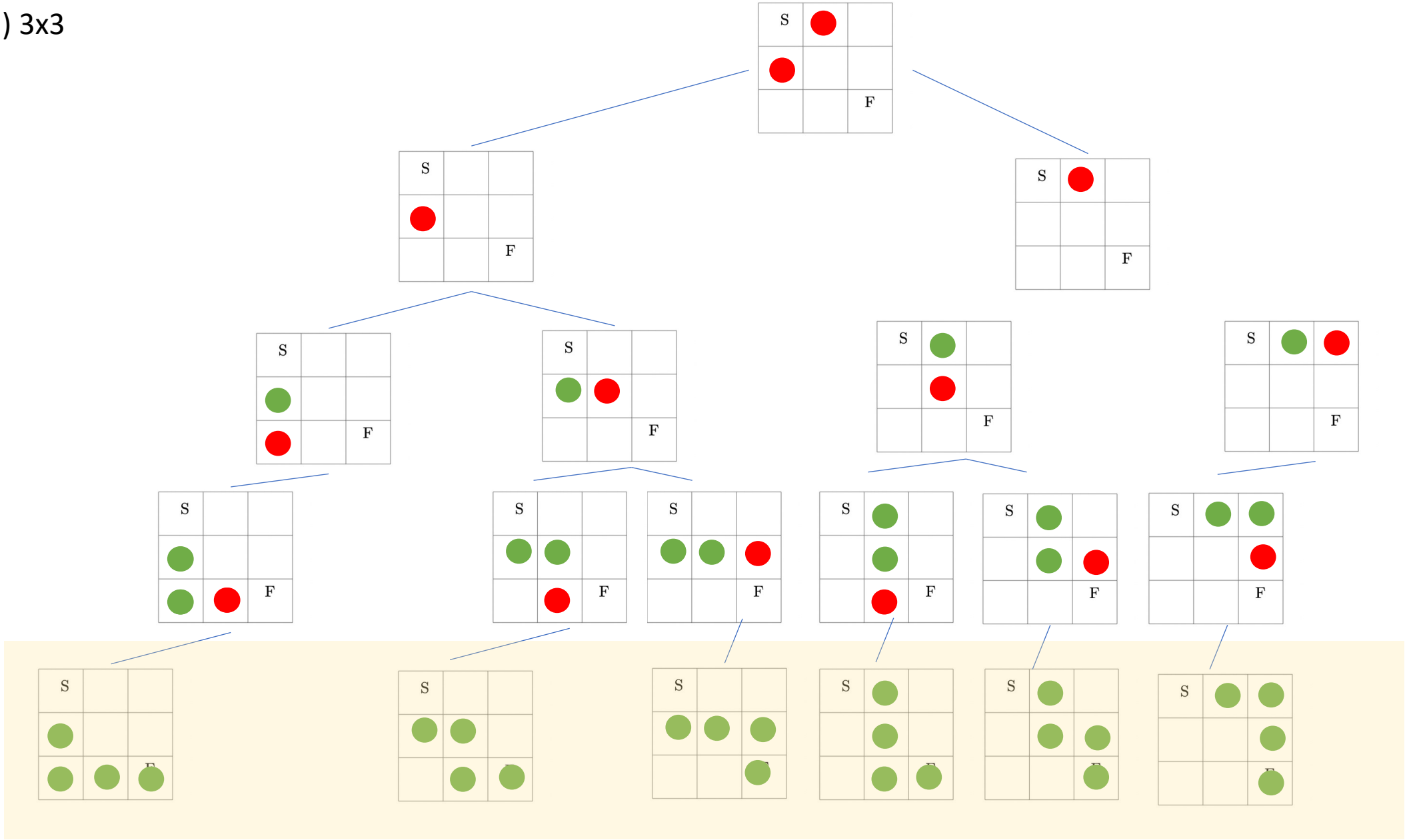
i) 3x3



i) 3x3



i) 3x3



n=6

Same procedure for ii) and iii)

b) You want to make an algorithm to calculate the amount of ways you can travel from the top left corner to the bottom right corner in an $n \times m$ grid, but you are not sure how to do it.

Your friend Saul claims that he has made an algorithm that solves this problem, he shows you the following pseudocode:

```
gridTravel(n,m):
    Array A[0...m, 0...n]
    A[1,1] = 1

    for j = 1,2,...,n:
        for i = 1,2,...,m:
            current = A[i,j]
            A[i+1][j] += current
            A[i][j+1] += current
        Endfor
    Endfor
    return A[m, n]
```

Does Saul's pseudocode work? Explain why or why not.

b) You want to make an algorithm to calculate the amount of ways you can travel from the top left corner to the bottom right corner in an $n \times m$ grid, but you are not sure how to do it.

Your friend Saul claims that he has made an algorithm that solves this problem, he shows you the following pseudocode:

```
1  gridTravel(n,m):
2      Array A[0...m, 0...n]
3      A[1,1] = 1
4
5      for j = 1,2,...,n:
6          for i = 1,2,...,m:
7              current = A[i,j]
8              A[i+1][j] += current
9              A[i][j+1] += current
10         Endfor
11     Endfor
12     return A[m, n]
```

Does Saul's pseudocode work? Explain why or why not.

Solution:
Yes! Saul's code works by solving a base case for our subproblems and adding them to solve the whole problem. A position x and y on the grid represents the number of unique paths we can make on a grid of size x*y from start to finish. We initialize [1,1] to be 1, as a grid of size 1'*'1 only has one path from start to finish. On line 8 and 9 we add the current step to the right and downward-facing cell, as we know that if we expand the grid, we have at least one path. If we keep expanding our grid, while adding the previous number of paths, we'll have the correct number of paths in the cell [n'*'n].

Task 5 Sequence alignment

a) In the sequence alignment problem, what does α_{pq} and δ represent, how do they relate to the cost?

Task 5 Sequence alignment

a) In the sequence alignment problem, what does α_{pq} and δ represent, how do they relate to the cost?

Solution:
δ represents the gap penalty, while α_{pq} represents the mismatch penalty, and the cost represents the sum of the gap and mismatch penalties.

b) Which of the following alignments of PALETTE and PALATE, has the lowest cost? Explain your answer.

Assume that $\delta = 2$ and $\alpha_{pq} = 1$ if p is not equal to q, and $\alpha_{pq} = 0$ if p is equal to q.

```
# Allignment 1
PALETTE
PALATE-

# Allignment 2
PALETTE
PALAT-E

# Allignment 3
P-ALETTE
-PALAT-E
```

b) Which of the following alignments of PALETTE and PALATE, has the lowest cost? Explain your answer.

Assume that $\delta = 2$ and $\alpha_{pq} = 1$ if p is not equal to q, and $\alpha_{pq} = 0$ if p is equal to q.

```
# Allignment 1
PALETTE
PALATE-

# Allignment 2
PALETTE
PALAT-E

# Allignment 3
P-ALETTE
-PALAT-E
```

Alignment 1:
E and A do not match, as well as T and E. Meanwhile, there is a gap at the end. The sum of the gap penalty and the mismatch penalty causes the total cost to be 4.

b) Which of the following alignments of PALETTE and PALATE, has the lowest cost? Explain your answer.

Assume that $\delta = 2$ and $\alpha_{pq} = 1$ if p is not equal to q, and $\alpha_{pq} = 0$ if p is equal to q.

```
# Allignment 1
PALETTE
PALATE-

# Allignment 2
PALETTE
PALAT-E

# Allignment 3
P-ALETTE
-PALAT-E
```

Alignment 2:
Here E and A do not match up, and there is also a gap. The sum of the gap penalty and the mismatch penalty causes the total cost to be 3.

b) Which of the following alignments of PALETTE and PALATE, has the lowest cost? Explain your answer.

Assume that $\delta = 2$ and $\alpha_{pq} = 1$ if p is not equal to q, and $\alpha_{pq} = 0$ if p is equal to q.

```
# Allignment 1
PALETTE
PALATE-

# Allignment 2
PALETTE
PALAT-E

# Allignment 3
P-ALETTE
-PALAT-E
```

Alignment 3:
There are three gaps, and the E and E do not line up. The sum of the gap penalty and the mismatch penalty causes the total cost to be 7.