COMPLEXITY ANALYSIS

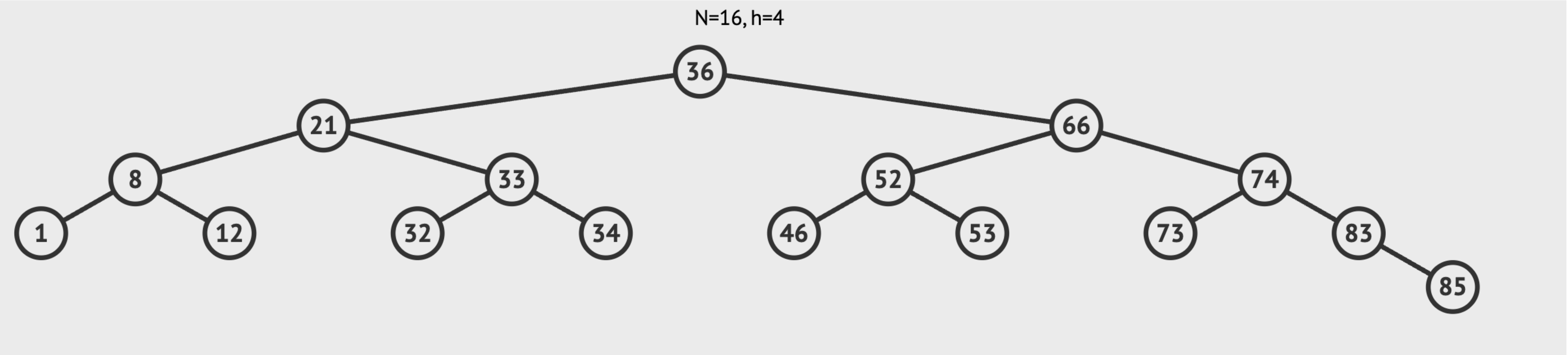
Problem Solving with Computers-II

Review: Big O and BST

- Big O: what does $T(n) = O(f(n))$ mean?
- What are the operations in a bst and how fast do they run?
- Std:: set vs. custom BST (lab03)

Balanced Binary Search Trees

- Definition: A Balanced tree is a tree whose height is O(log n)
  - Example of balanced BSTs: AVL trees, red black trees (std::set)
- Visualize: https://visualgo.net/bn/bst



Balanced BST time complexity (std::set)

```
      8
    /   \
  4       12
 / \      / \
2   6   10   14
/    \
1     7
```

Given a balanced BST with n nodes, which operation(s) have a time complexity of O(log n)?

A. min/max
B. search
C. successor
D. All of the above

Discuss best case/worst case for each operation

Amortized Analysis

```
       8
     /   \
    4     12
   / \    / \
  2  6  10  14
 /    \
1      7
```

What is the worst case time complexity of this code?

```
void printSetValues(const std::set<int>& s){
    for (int value : s) {
        std::cout << value << " ";
    }
}
```

A. O(1) B.O(log n) C. O(n) D. O(nlogn)

Note: Worst case time complexity of successor is O(log n)

Comparing algorithms

```
      8
    /   \
   4     12
  / \    / \
 2   6  10  14
/     \
1      7
```

Which code is faster to find a key in a set (s)?

```
A. bool find(const std::set<int>& s, int key){
       for (int value : s) {
          if(value == key) return true;
       }
       return false;
   }

B. bool find(const std::set<int>& s, int key){
       set<int>::iterator it = s.find(key);
       if(it != s.end()) return true
       return false;
   }

C. Both are equally fast!
```

Finding common keys

Given a std::set with N unique integer keys and a std::vector with M integer keys (not necessarily unique), you need to find all keys common to both, returning a std::set of the found keys. Two solutions are implemented (see handout for code):

- **Solution 1**: Iterate over the M vector keys, using std::set::find to check if each key is in the set.
- **Solution 2**: Iterate over the N set keys, using std::find on the unsorted vector to check if each key is in the vector.

What is the time complexity of these solutions?

Assume the number of common keys is bounded a constant K

Finding common keys (contd)

- **Solution 1**: Iterate over the M vector keys, using std::set::find to check if each key is in the set.
- **Solution 2**: Iterate over the N set keys, using std::find on the unsorted vector to check if each key is in the vector.

Which of the following correctly describes the time complexity of these solutions?

Option	Solution 1	Solution 2
A	O(M * N)	O(N * M)
B	O(M * log N)	O(N * M)
C	O(M)	O(N * M)
D	O(M * log N)	O(N * log M)

Space Complexity

S(n) = auxiliary memory needed to compute F(n)

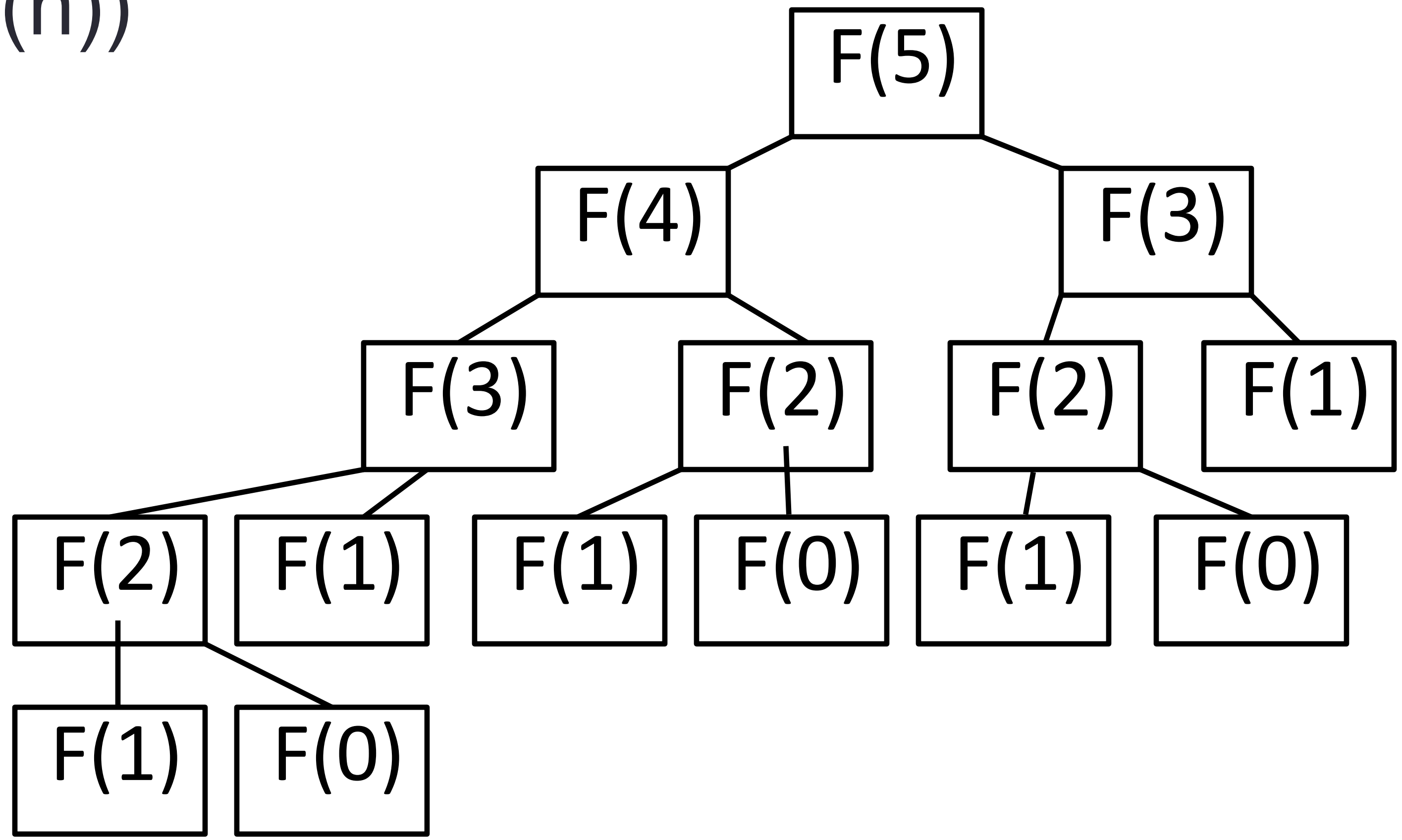
In general space complexity includes space to store inputs + auxiliary space. But for this class assume auxilliary space only

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```

What is S(n)? Express your answer in Big-O notation

What is S(n)? Express your answer in Big-O notation

A. $O(1)$
B. $O(\log(n))$
C. $O(n)$
D. $O(n^2)$
E. $O(2^n)$



Tree of recursive calls needed to compute F(5)

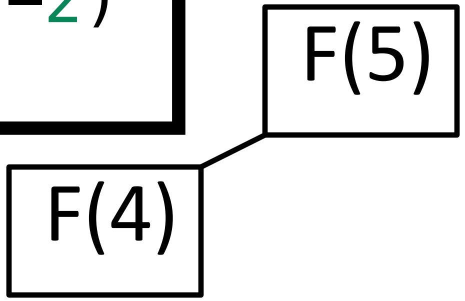
S(n) relates to maximum depth of the recursion

```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```

F(5)

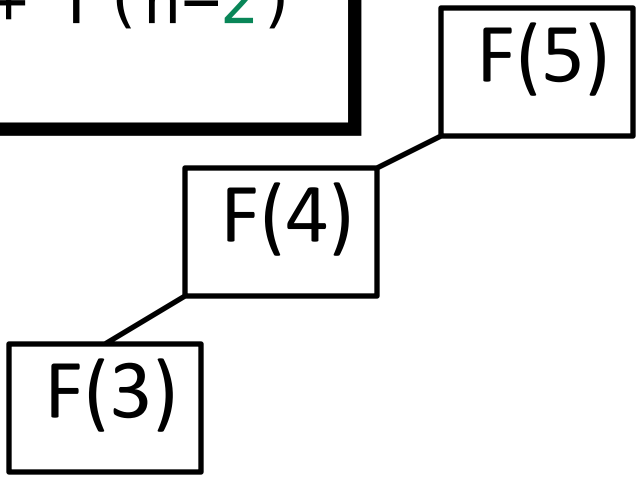
S(n) relates to maximum depth of the recursion

```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



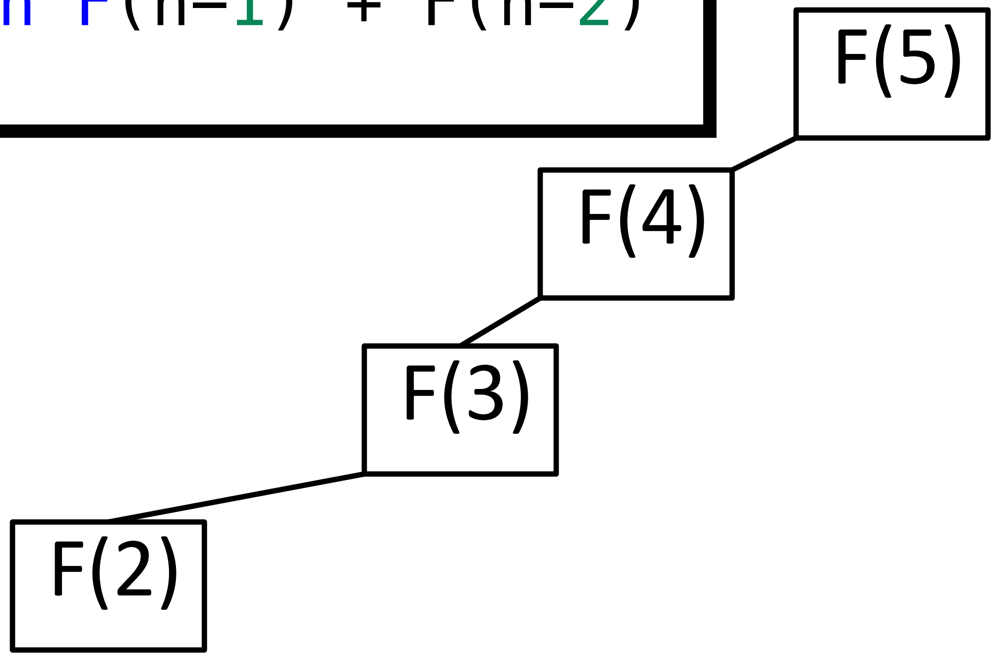
S(n) relates to maximum depth of the recursion

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



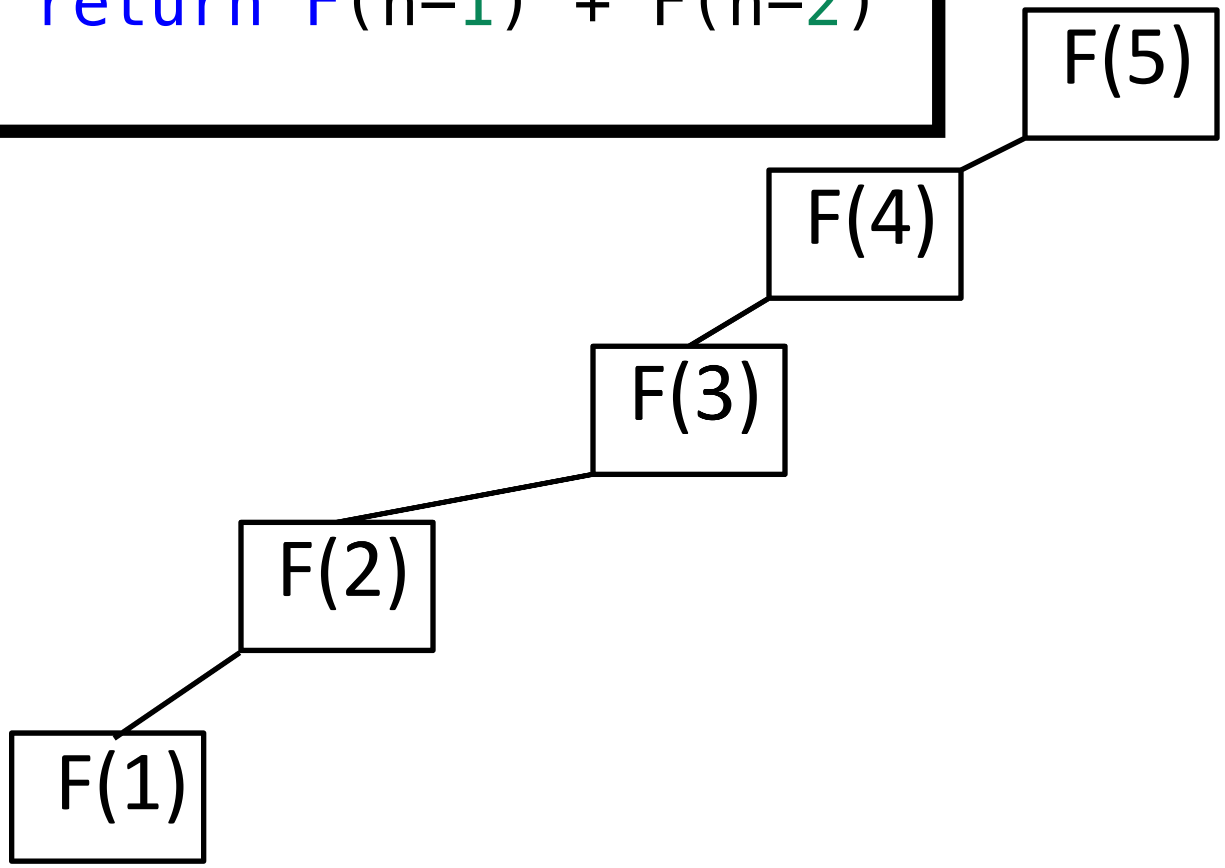
S(n) relates to maximum depth of the recursion

```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



S(n) relates to maximum depth of the recursion

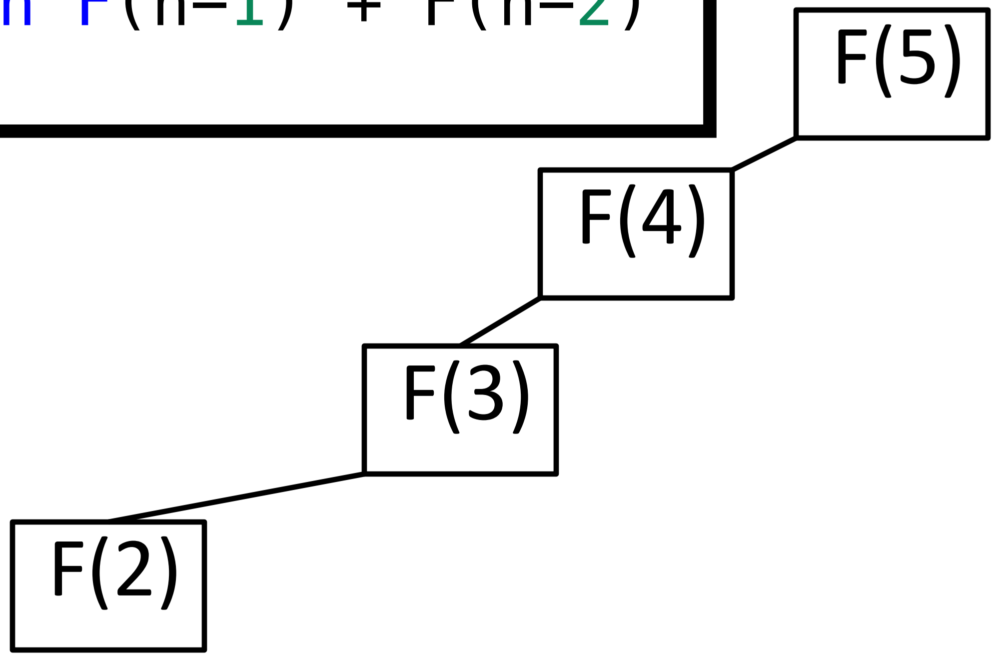
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

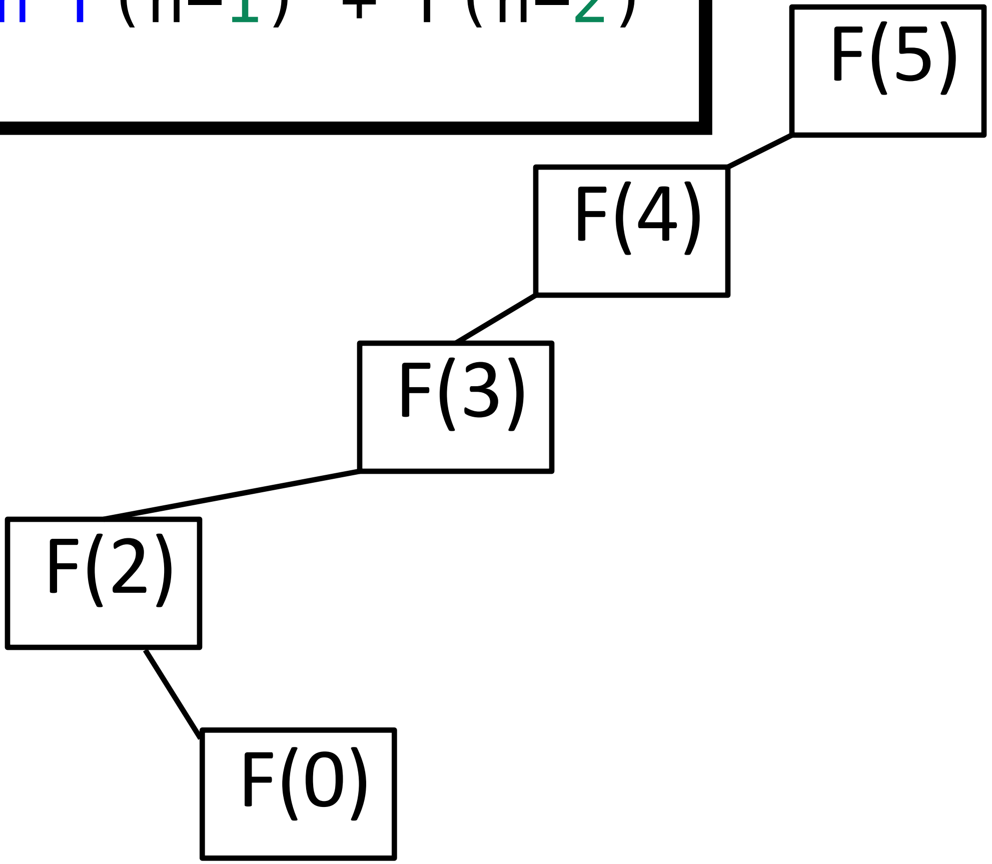
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

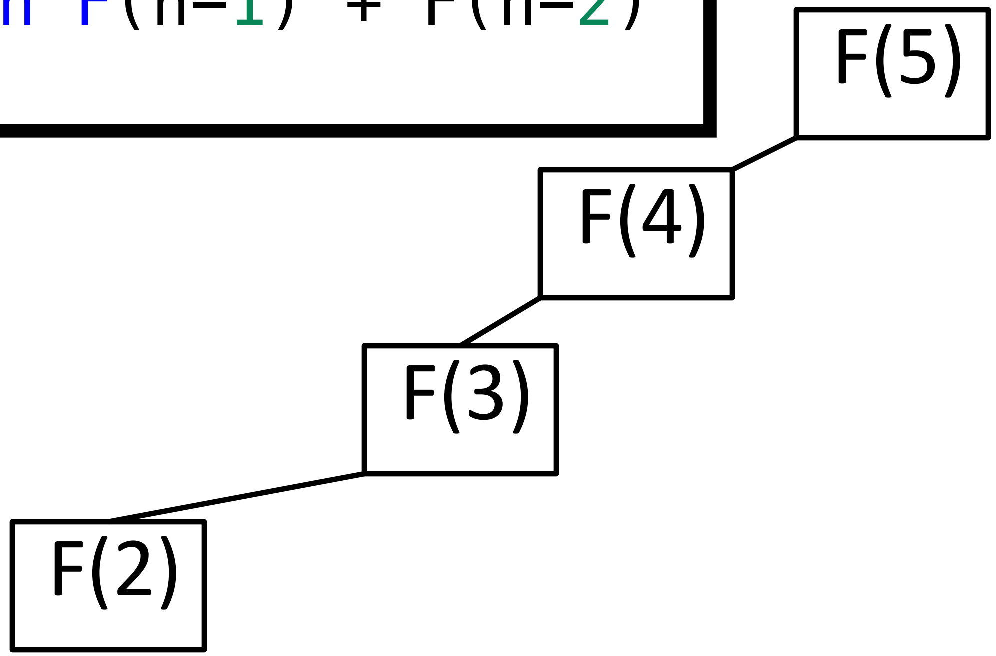
```
F(int n){
    if(n <= 1) return 1
    return F(n−1) + F(n−2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

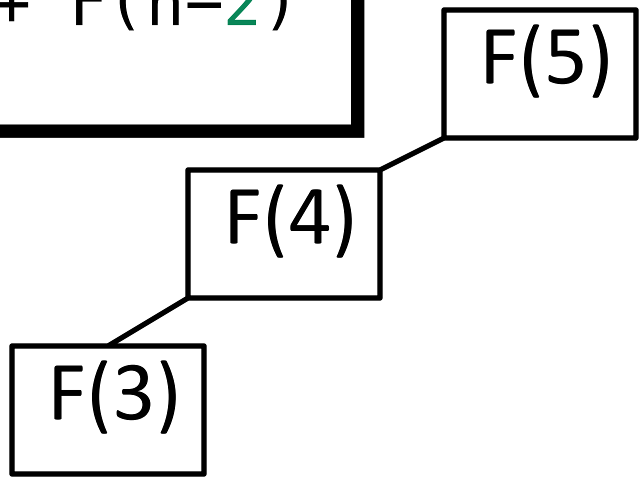
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

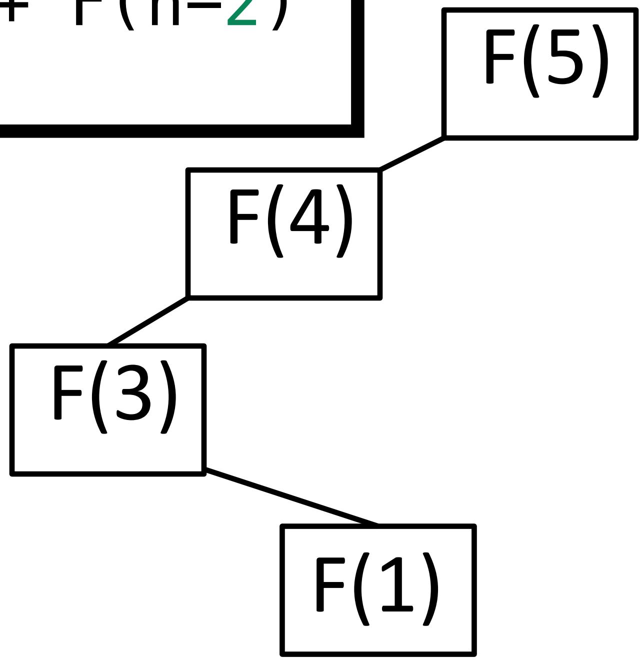
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

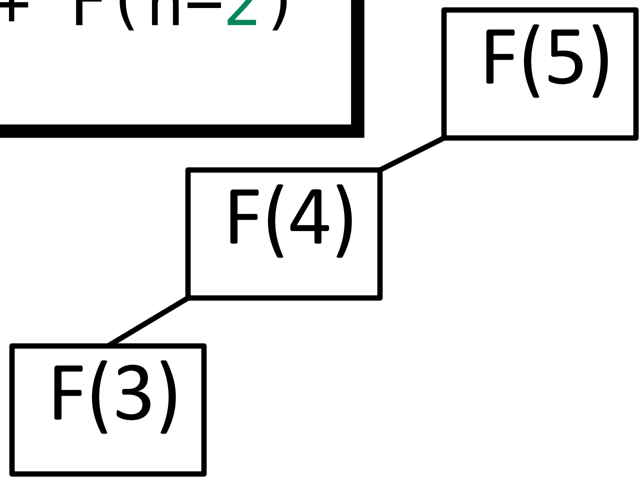
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

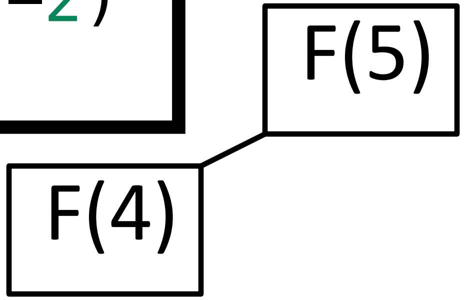
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

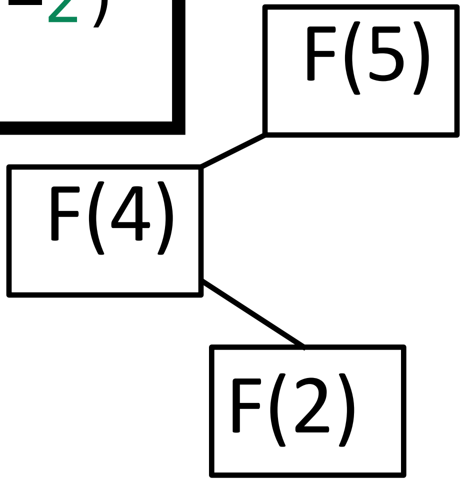
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

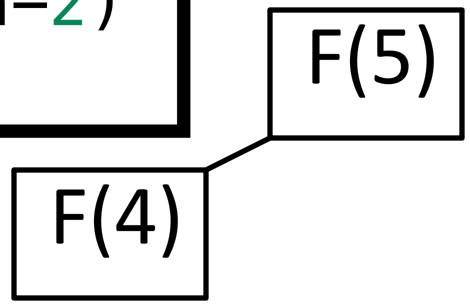
```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

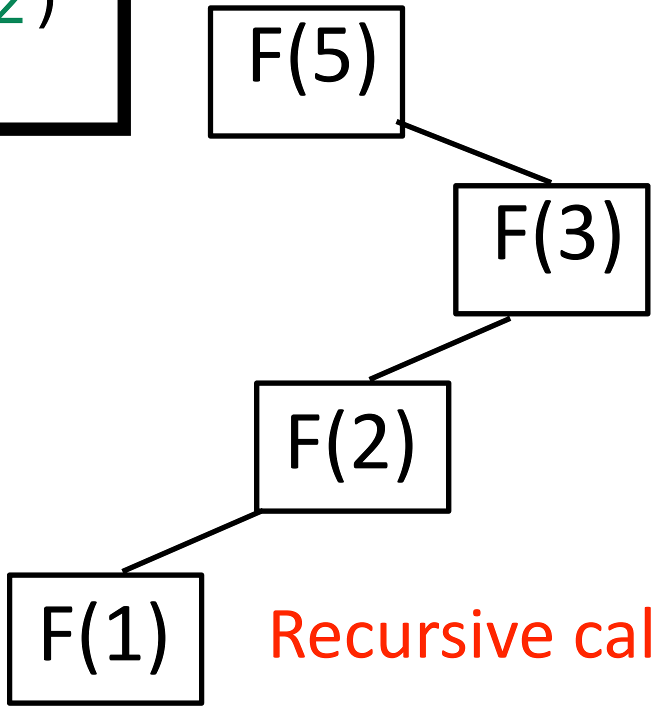
```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Maximum depth of the recursion = 5

S(n) relates to maximum depth of the recursion

```
F(int n){
    if(n <= 1) return 1
    return F(n-1) + F(n-2)
}
```



Recursive calls continue

Maximum depth of the recursion for F(n) = n

Therefore, S(n) = O(n)

Which algorithm is more space efficient?

A.

```
F(int n){
   if(n <= 1) return 1
   return F(n-1) + F(n-2)
}
```

B.

```
F(int n){
    Initialize A[0 . . . n]
    A[0] = A[1] = 1

    for i = 2 : n
      A[i] = A[i-1] + A[i-2]

    return A[n]
}
```

C. *Both are the same: O(n)*