UGANDA CHRISTIAN UNIVERSITY
A Center of Excellence in the Heart of Africa

Design and Analysis of Algorithms

Chapter 5: Dynamic Programming

Dynamic Programming

- Wikipedia definition: "*method for solving complex problems by breaking them down into simpler subproblems*“
- *This definition will make sense once we see some examples - Actually, we'll only see problem solving examples.*
- *General, powerful Algorithm Design Technique.*

Dynamic Programming

- A method for solving classical problems (Like Divide & Conquer; it's not a specific algorithm)

- 'programming' here is not referring to software. The word itself is older than the computer. 'programming' means any tabular method to accomplish a task.

- Richard Bellman introduced it in 1949. He developed the method with Lester Ford to find the shortest path in a graph.

Dynamic Programming

- For what type of problems DP is useful?
  - the problems that can be broken into subproblems
- Think of DP as kind of exhaustive search.
- Perspectives:
  - ~~ Careful brute force - search all possibilities but do it in a smart way to get the optimal solution
  - ~~ Subproblems + re-use (the solution)

Divide & Conquer	**Dynamic Programming**
you deal with independent subproblems	you deal with overlapping subproblems

Steps for Solving DP Problems

1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

Each step is important

1 SAMUEL 7:12

Dynamic Programming

- Dynamic programming is a very powerful, general tool for solving optimization problems.
- Once understood it is relatively easy to apply, but many people have trouble understanding it.

Greedy Algorithms

- Greedy algorithms focus on making the best local choice at each decision point.
- For example, a natural way to compute a shortest path from x to y might be to walk out of x, repeatedly following the cheapest edge until we get to y. WRONG!
- In the absence of a correctness proof greedy algorithms are very likely to fail.

Inductive definitions

- Factorial
  - $f(0) = 1$
  - $f(n) = n \times f(n-1)$
- Insertion sort
  - isort([ ]) = [ ]
  - $\text{isort}([x_1,x_2,..,x_n]) = \text{insert}(x_1,\text{isort}([x_2,\ldots,x_n]))$

… Recursive programs

```
int factorial(n):
  if (n <= 0)
    return(1)
  else
    return(n*factorial(n-1))
```

Optimal substructure property

- Solution to original problem can be derived by combining solutions to subproblems

- factorial(n-1) is a subproblem of factorial(n)
  - So are factorial(n-2), factorial(n-3), …, factorial(0)
  - isort([x_2,…,x_n]) is a subproblem of isort([x_1,x_2,…,x_n])
  - So is isort([x_i,…,x_j]) for any $1 \le i \le j \le n$

Evaluating subproblems

Fibonacci numbers

- fib(0) = 0
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

Computing fib(5)

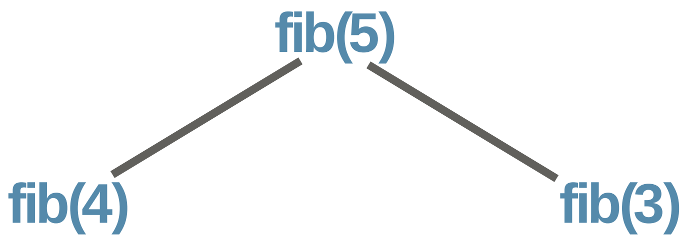
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
                fib(n-2)
  return(value)
```

fib(5)

1 SAMUEL 7:12

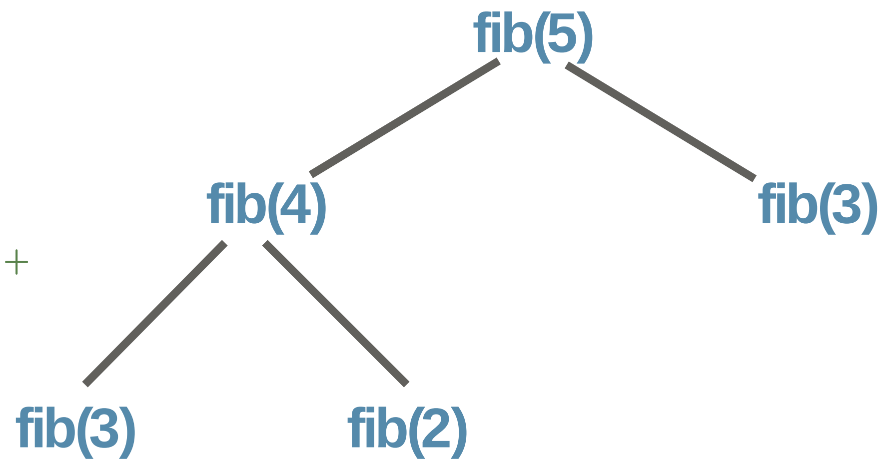
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



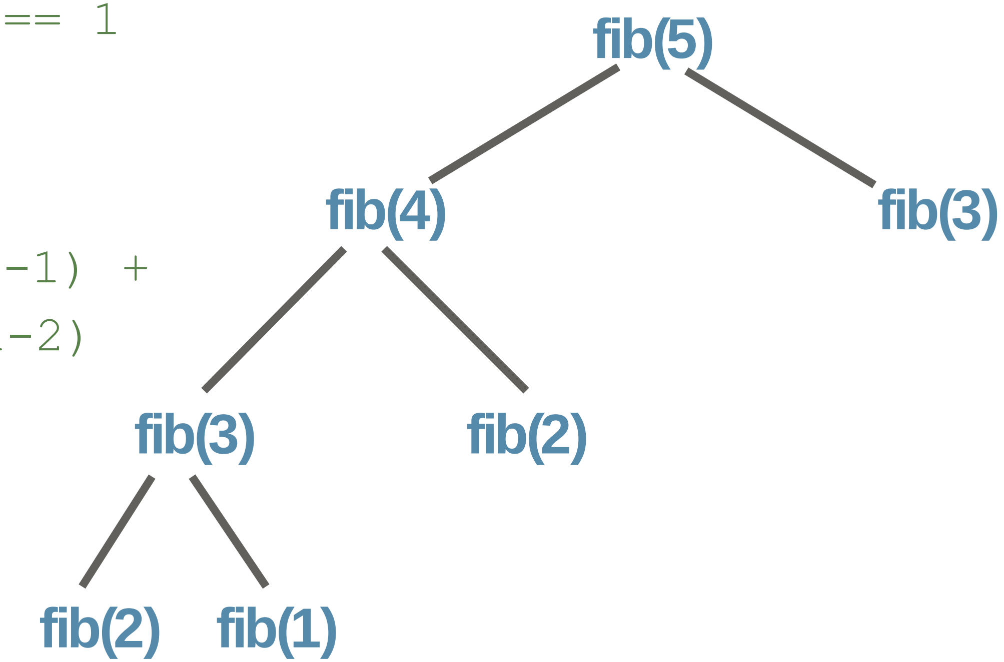
Computing fib(5)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



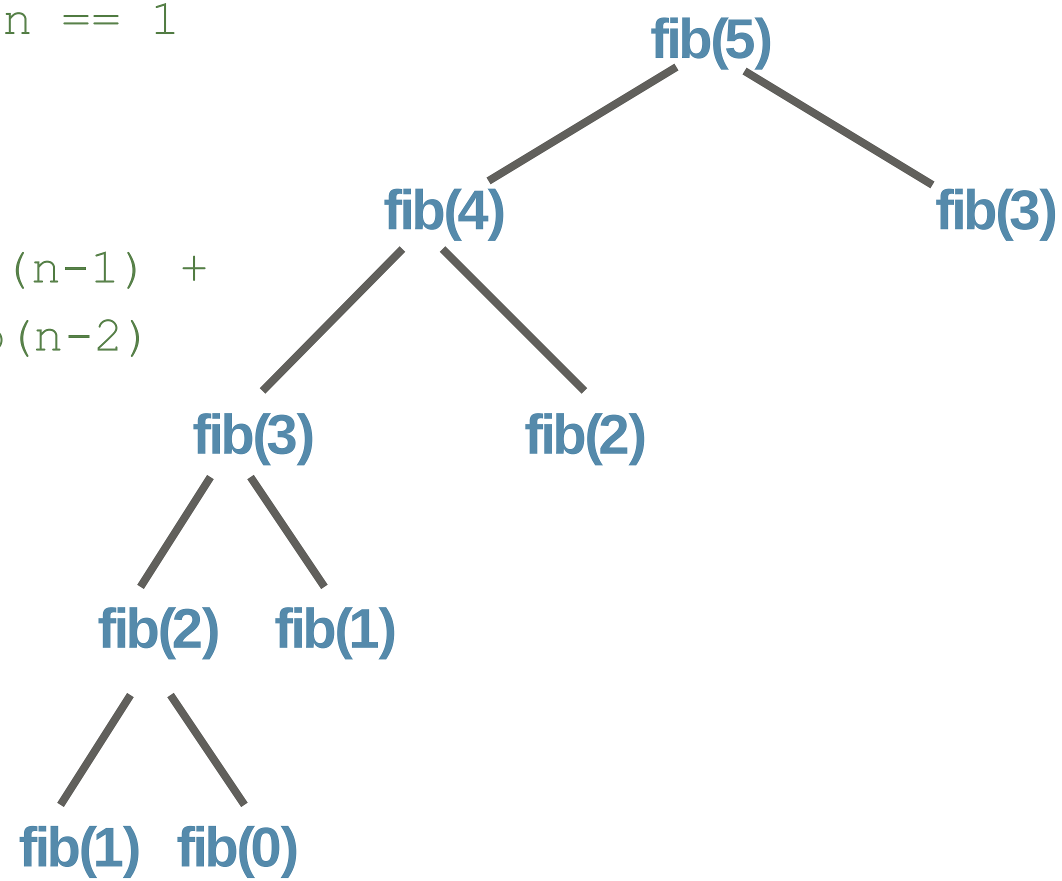
Computing fib(5)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



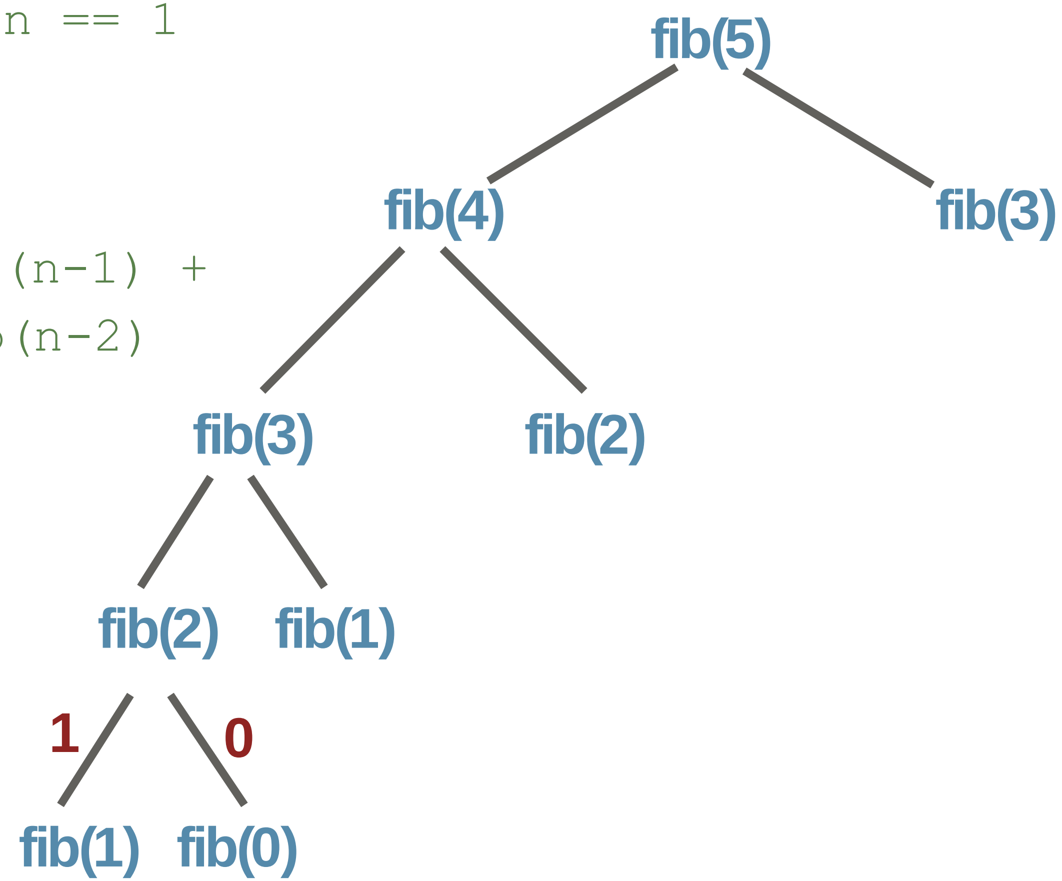
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



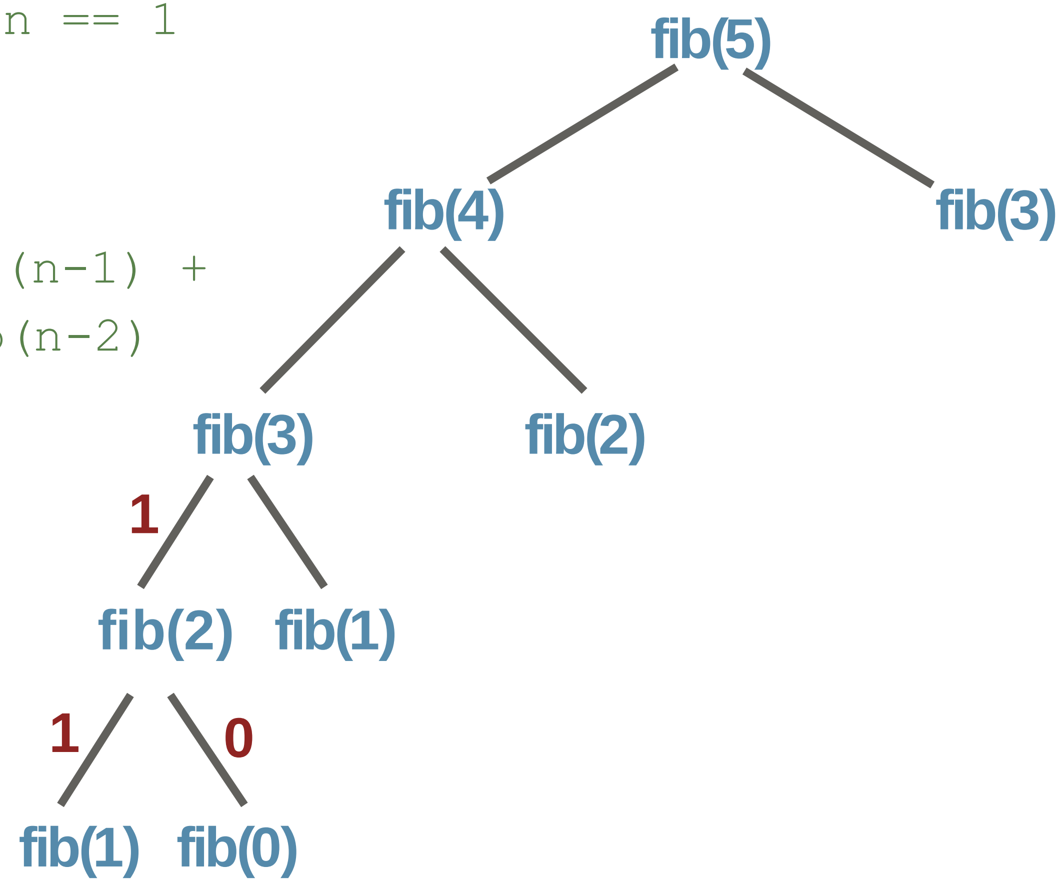
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



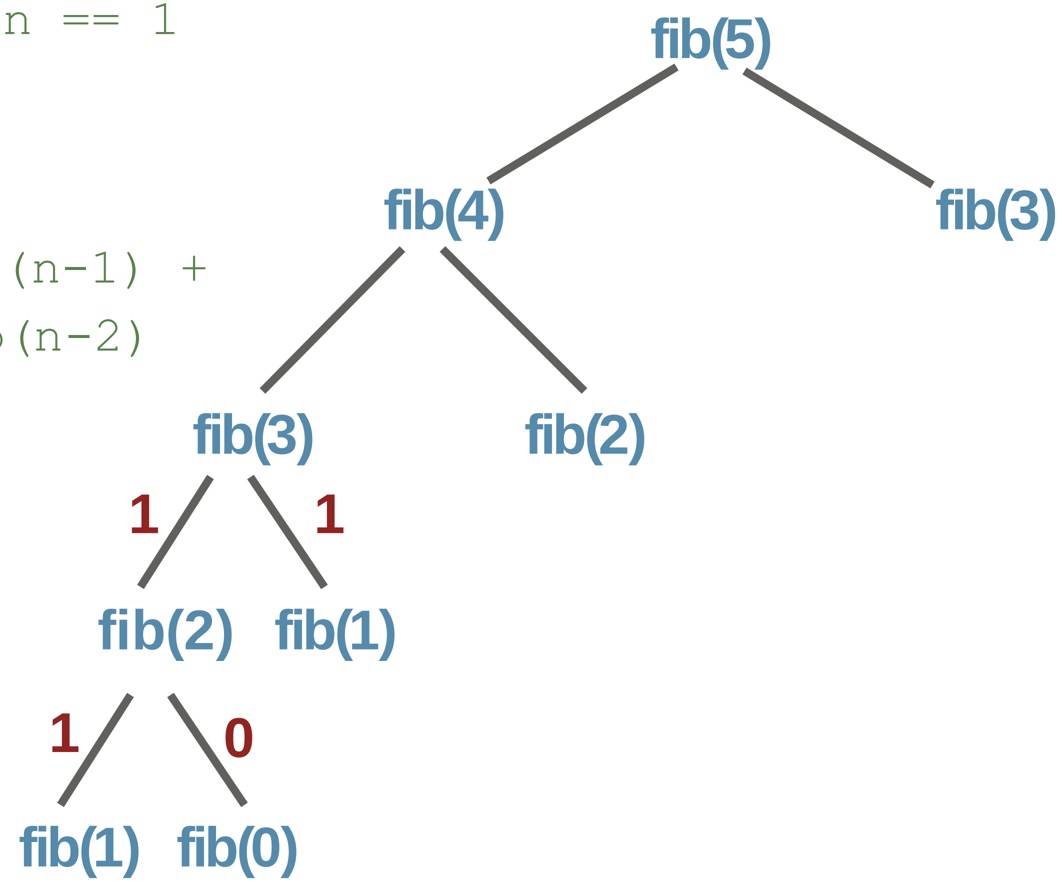
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
               fib(n-2)
   return(value)
```



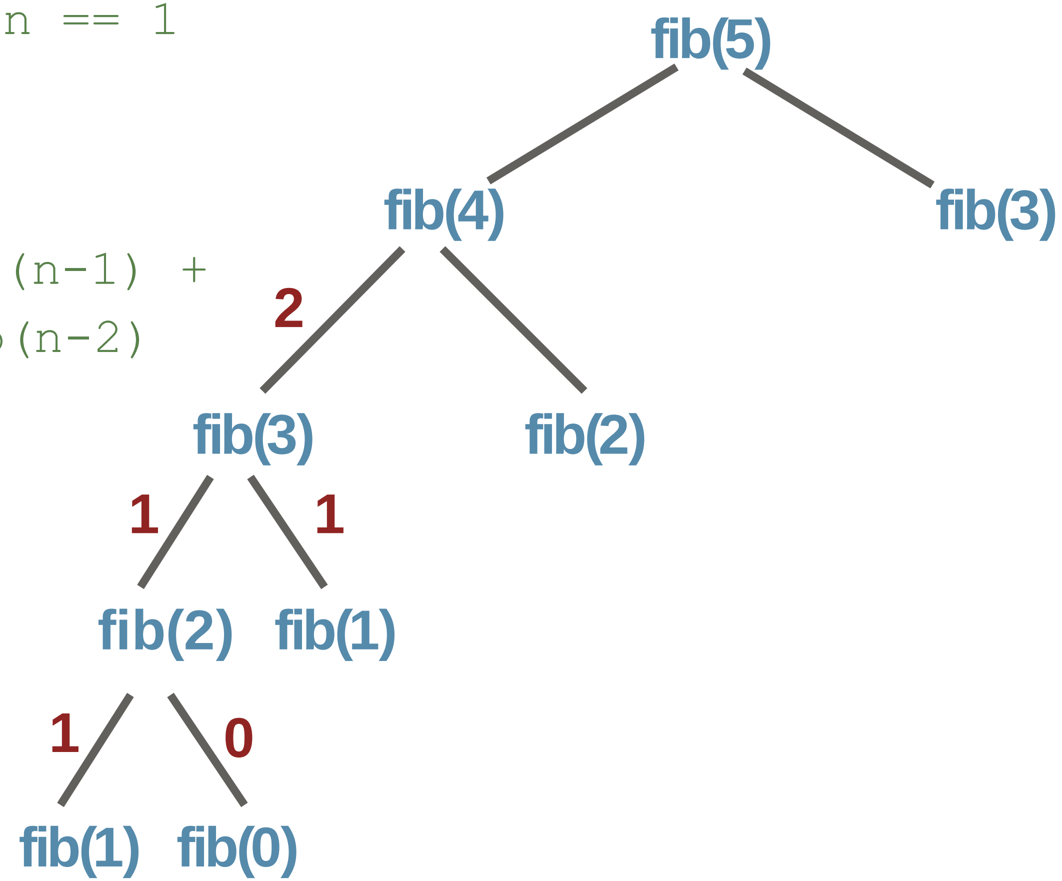
Computing fib(5)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



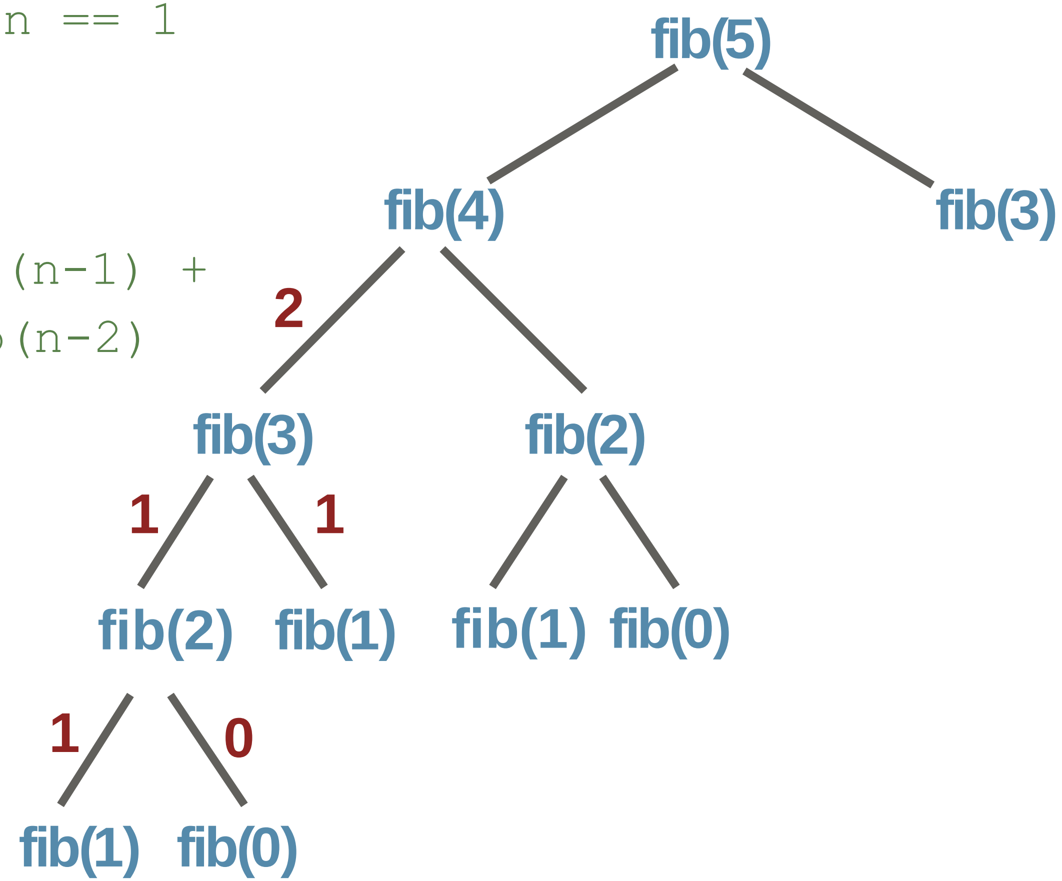
Computing fib(5)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



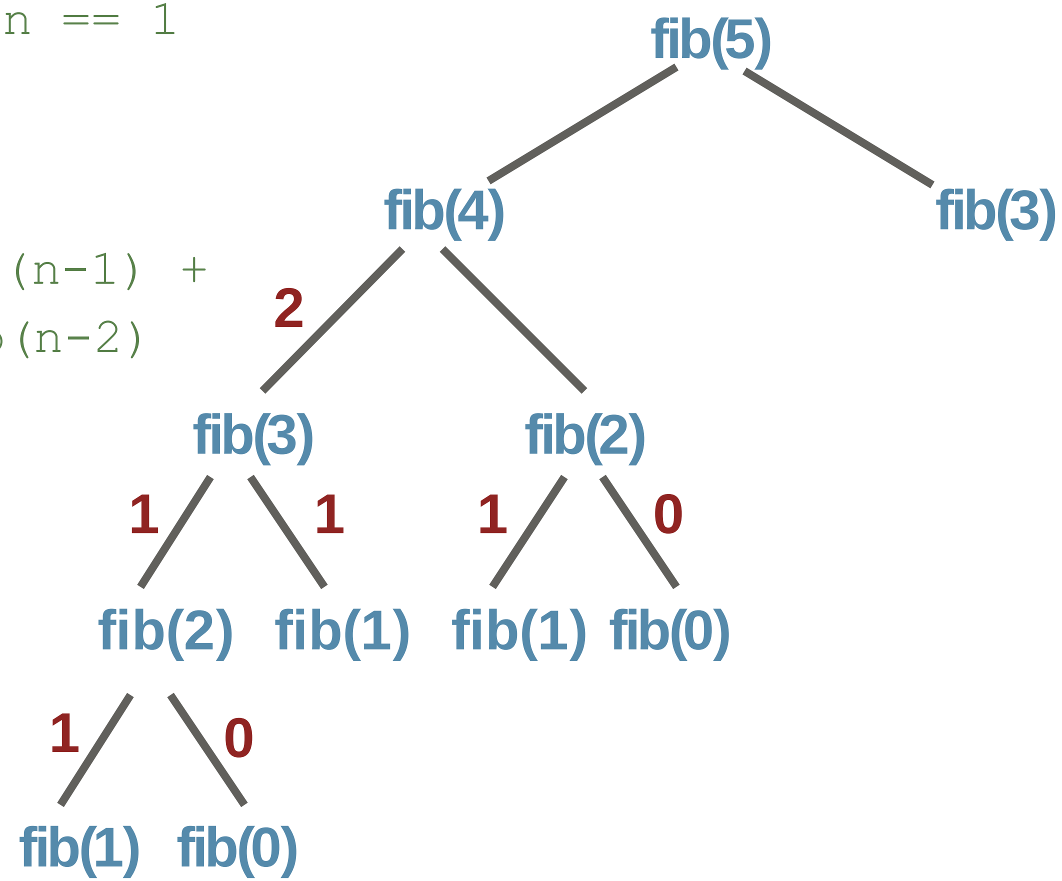
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



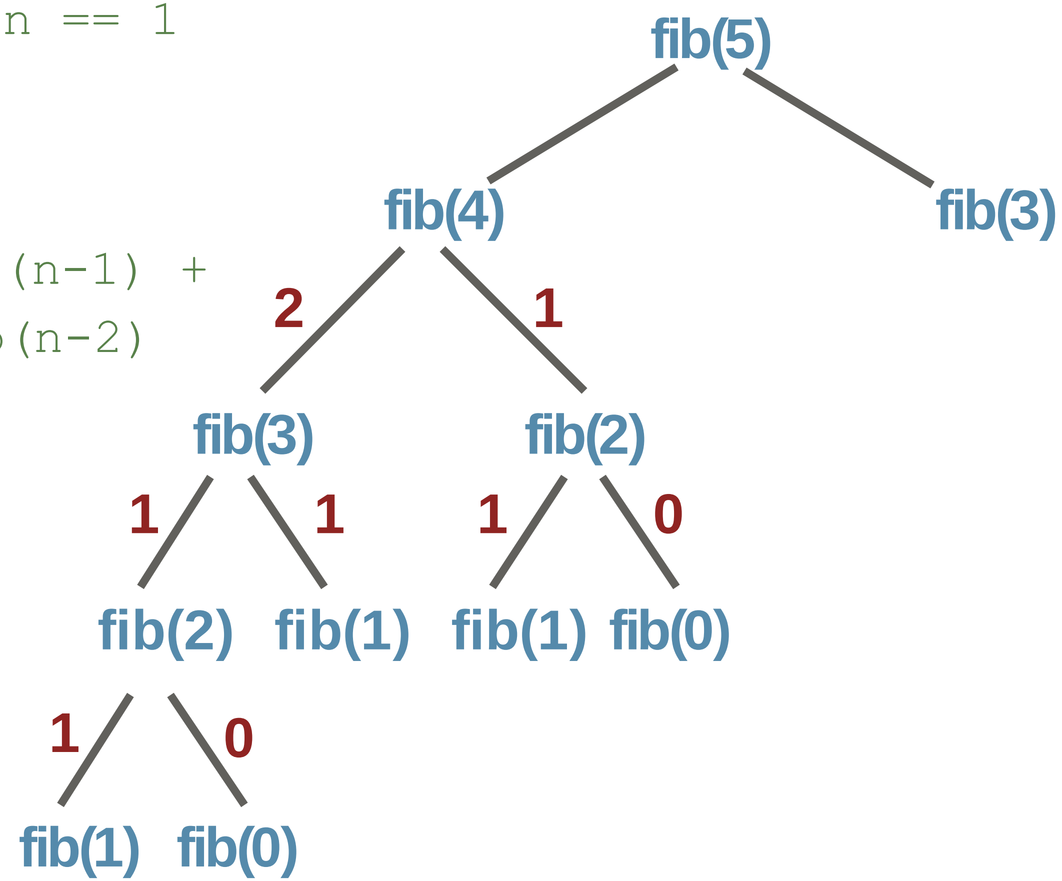
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



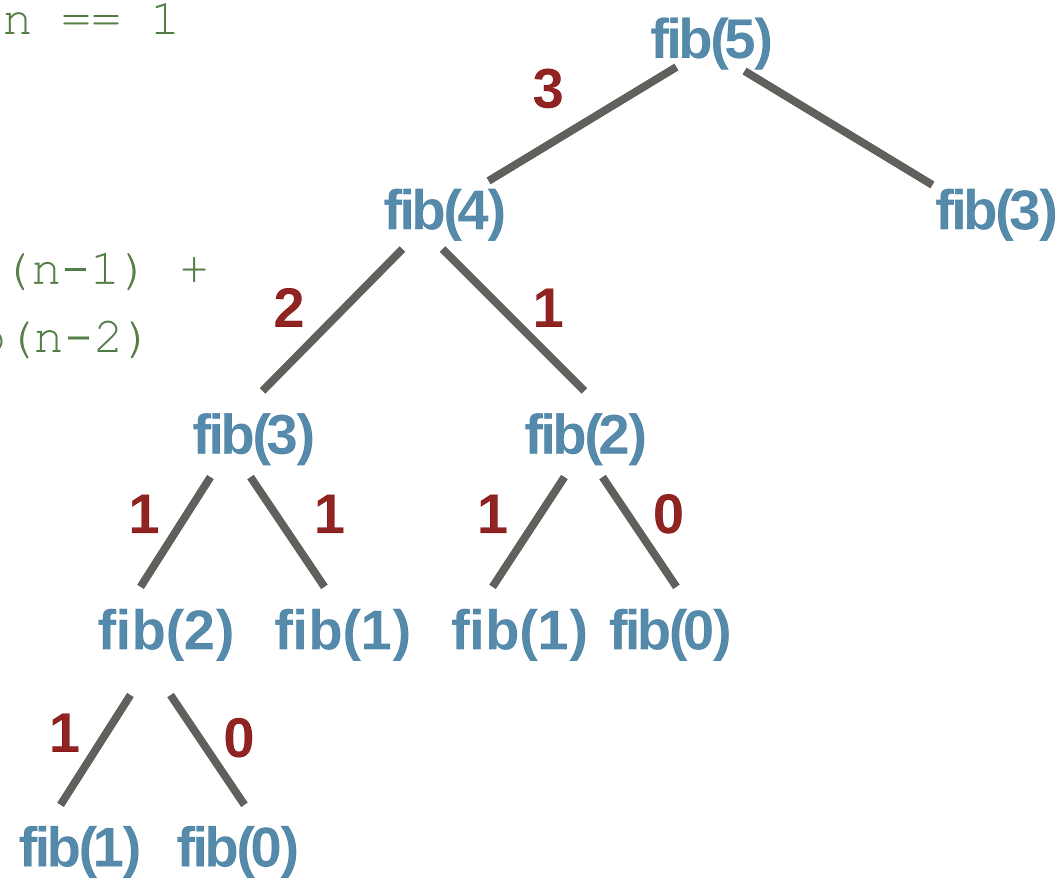
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



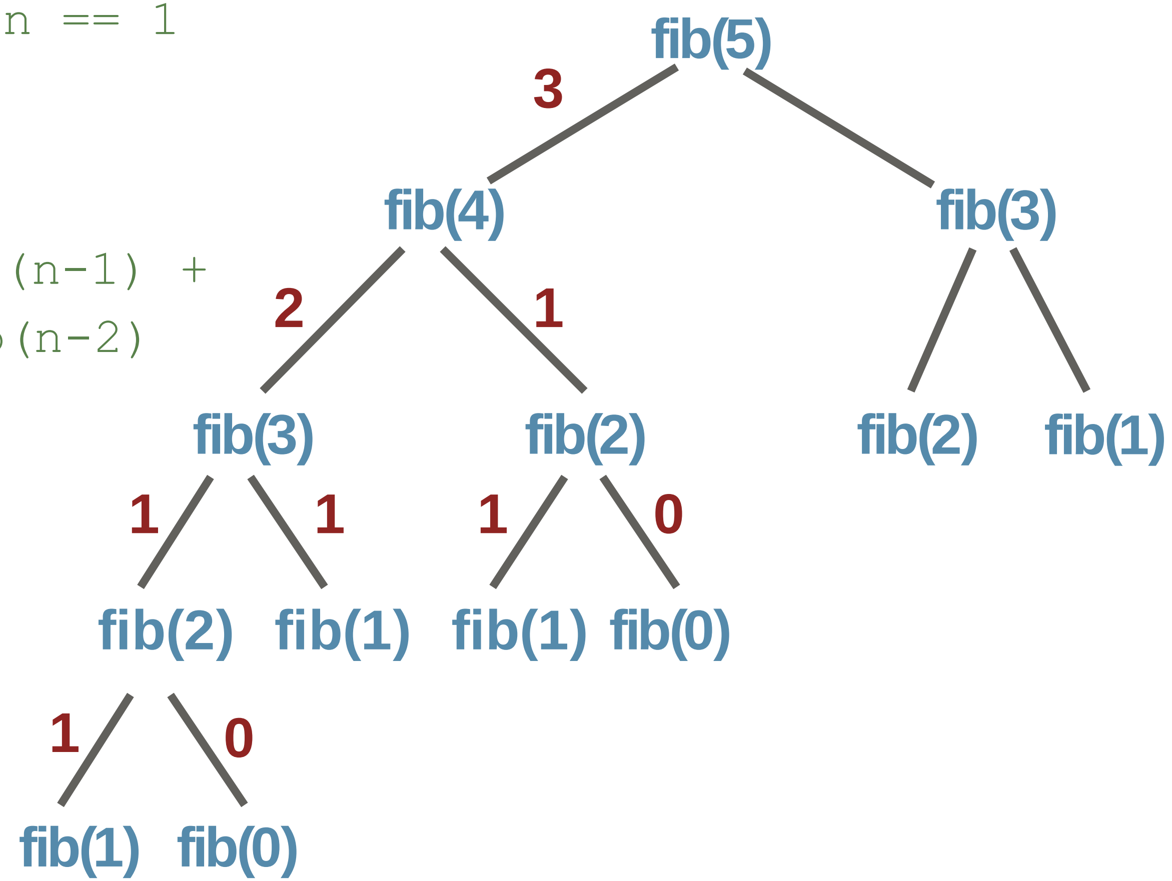
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



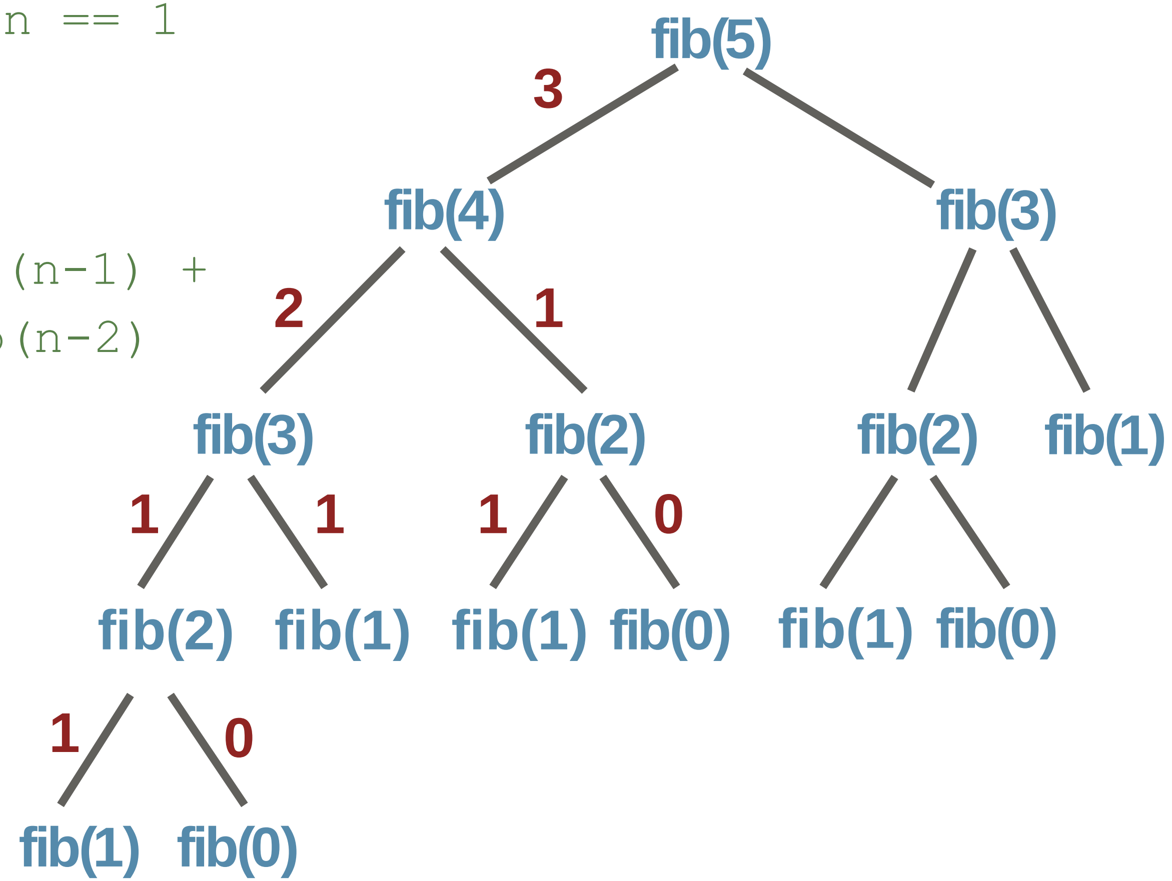
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



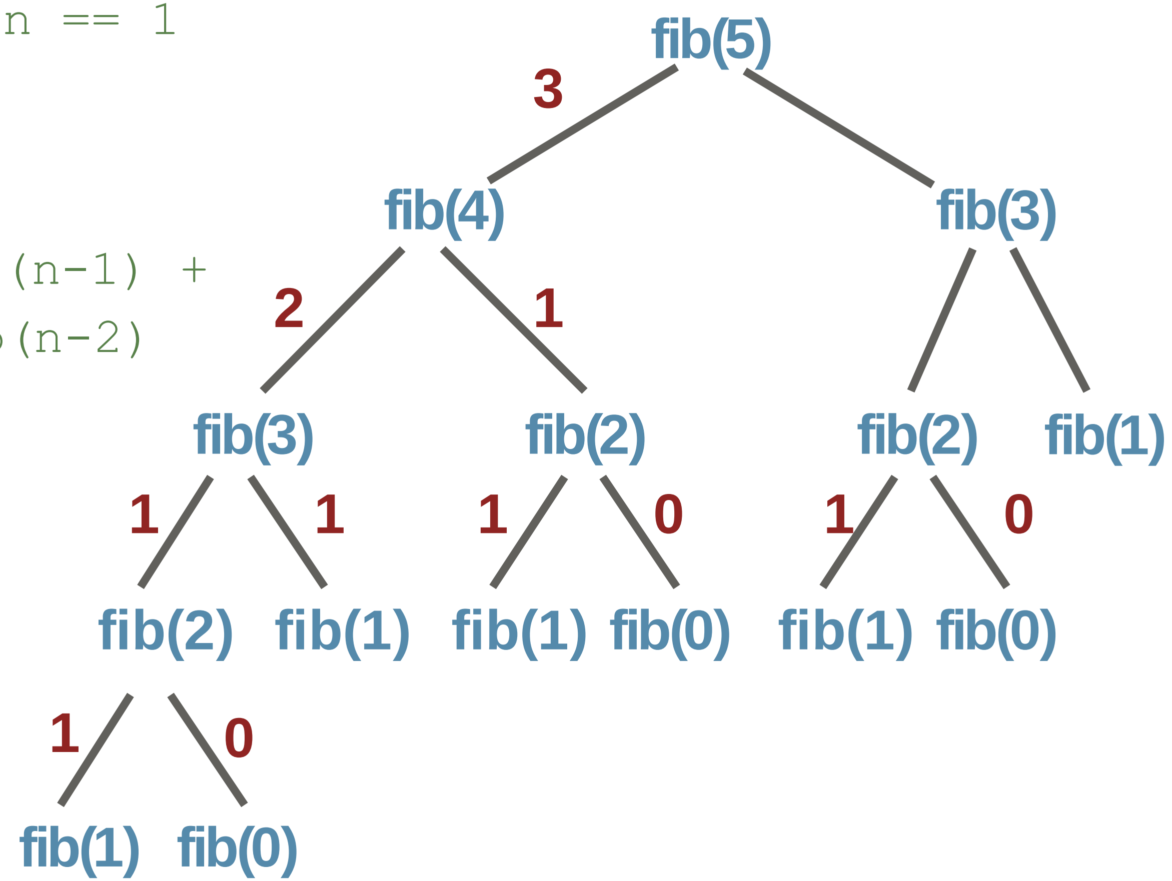
Computing fib(5)

```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



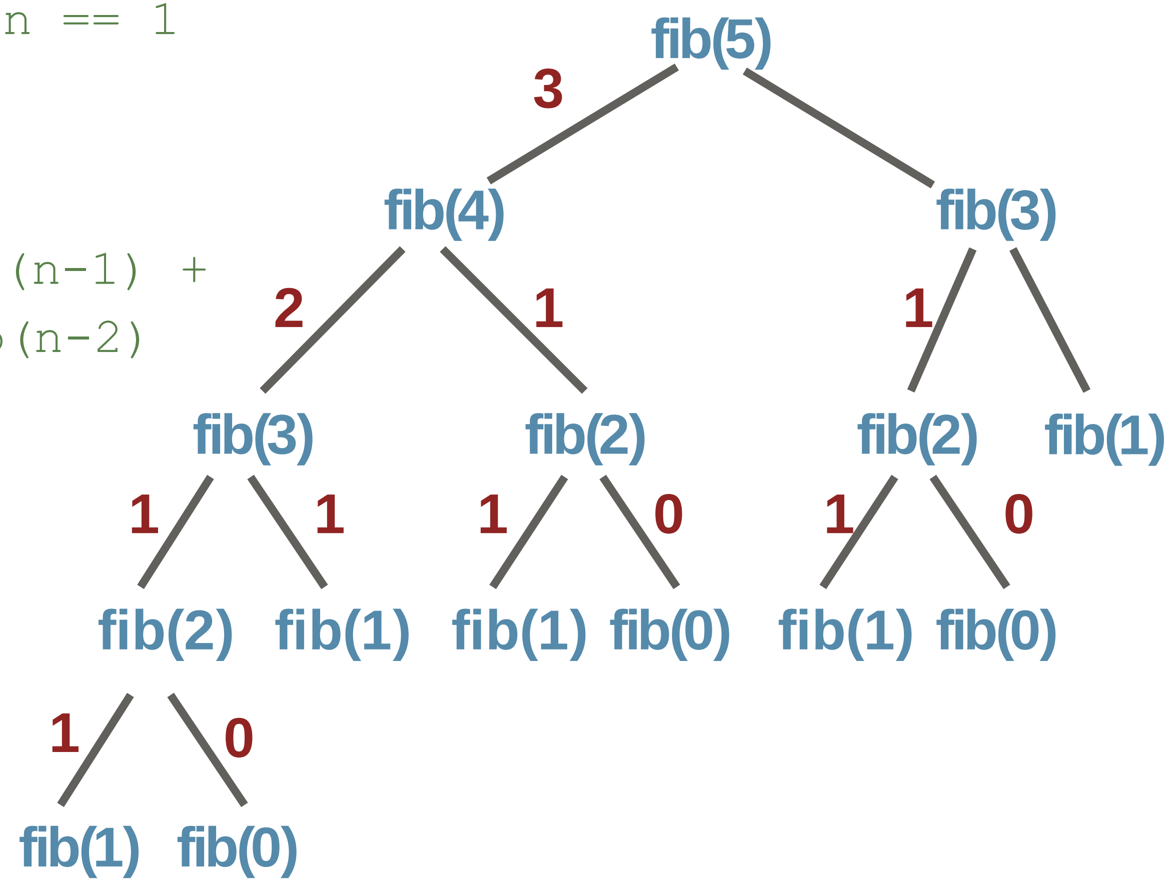
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



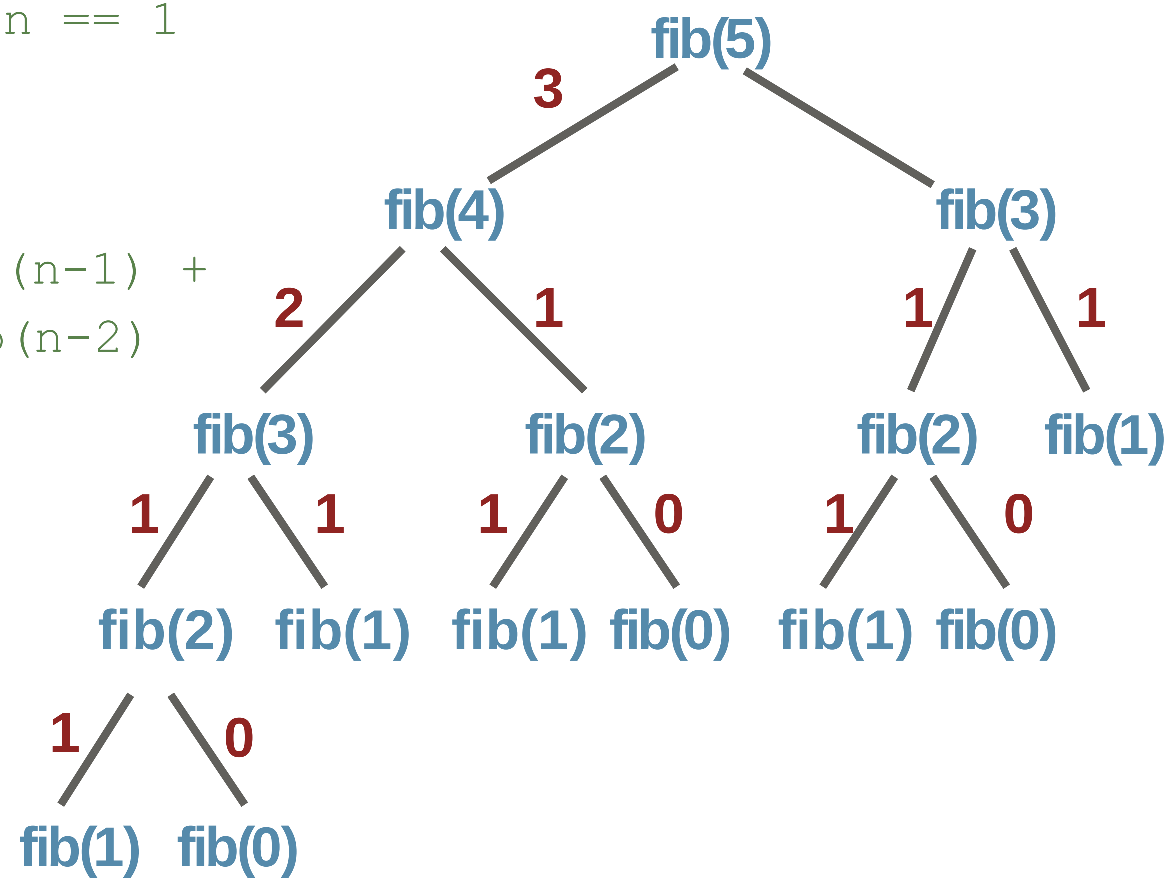
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



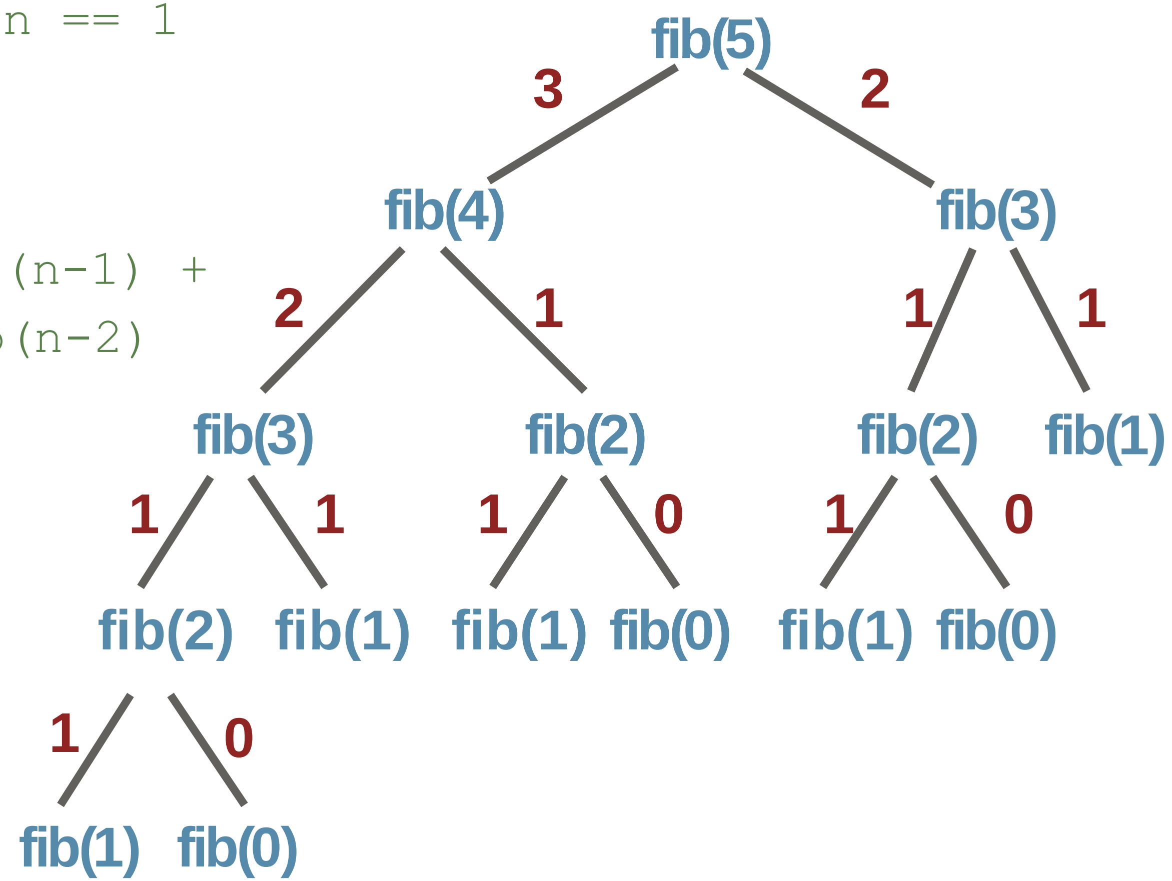
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



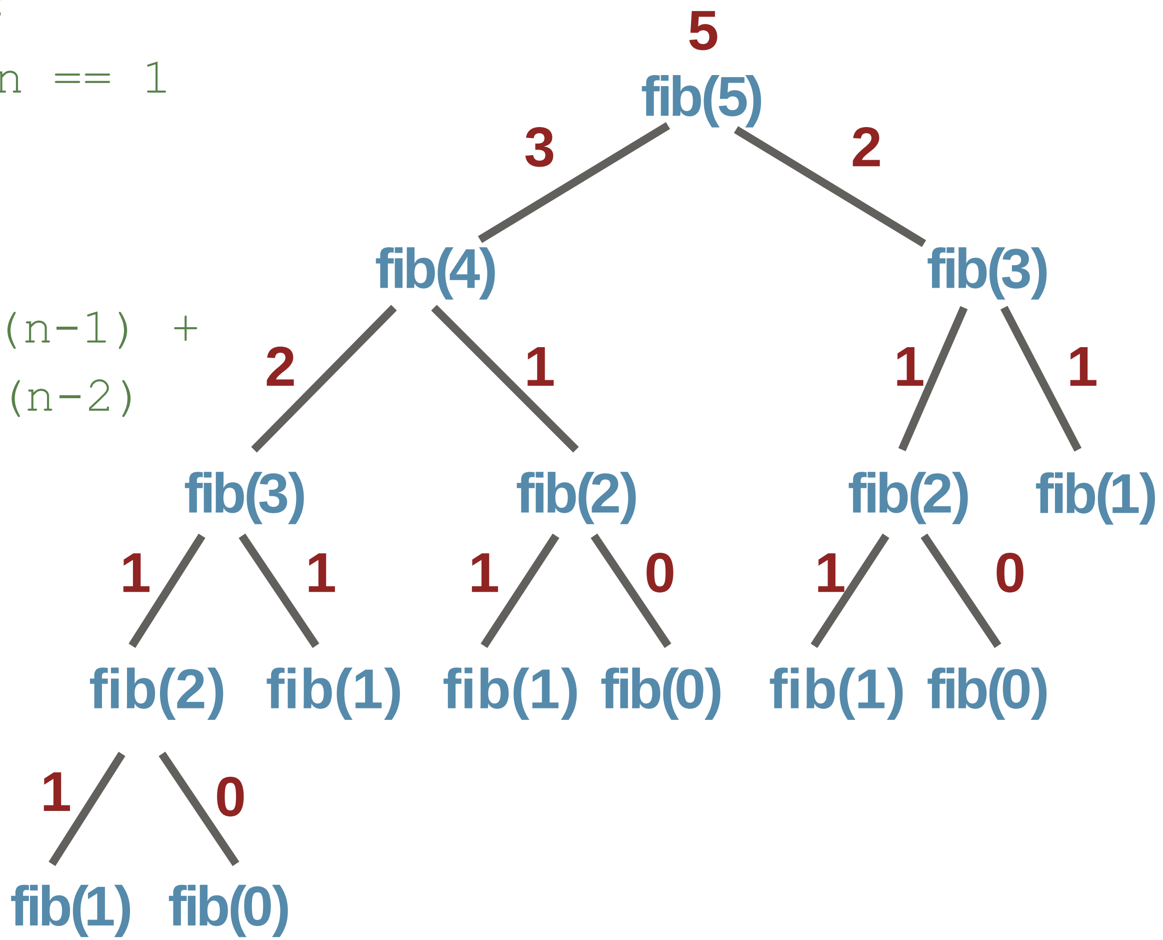
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



Computing fib(5)

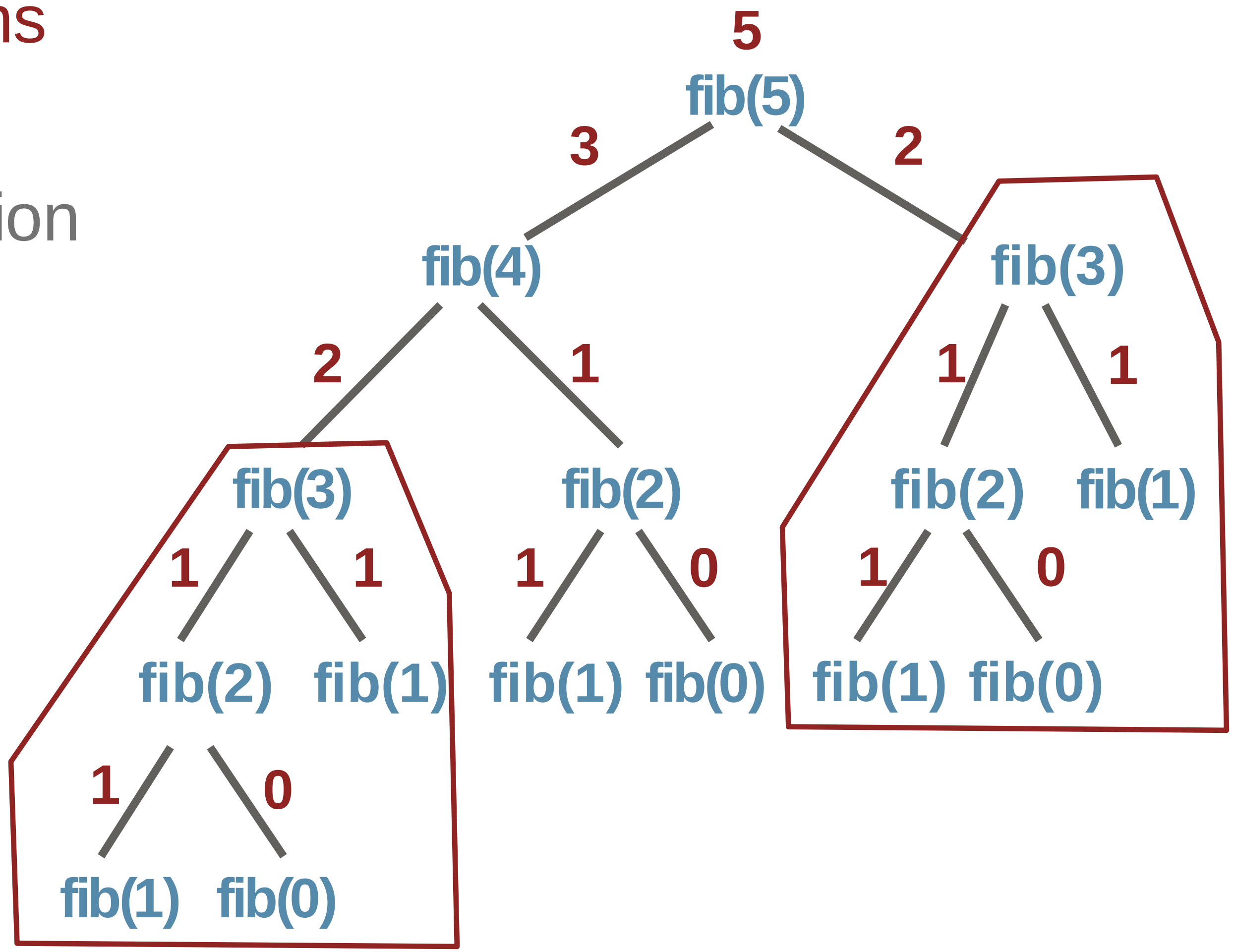
```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



Computing fib(5)

Overlapping subproblems

- Wasteful recomputation
- Computation tree grows exponentially



Python Implementation

```
def fib(n):
  if n == 0:
     return 0
  elif n == 1:
     return 1
  else:
     return fib(n-1) + fib(n-2)

print([fib(n) for n in range(15)])
```

Never re-evaluate a subproblem

- Build a table of values already computed
  - Memory table
- Memoization
  - Remind yourself that this value has already been seen before

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

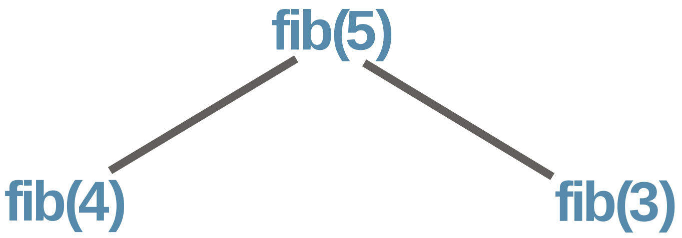
fib(5)

k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

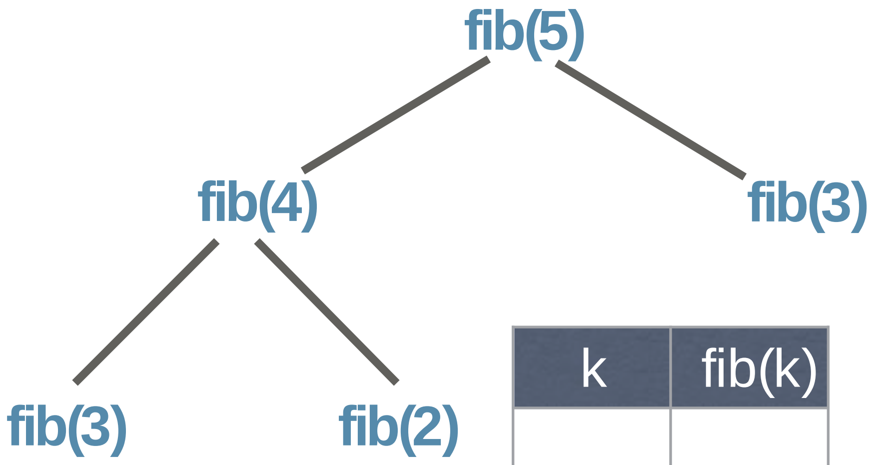


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

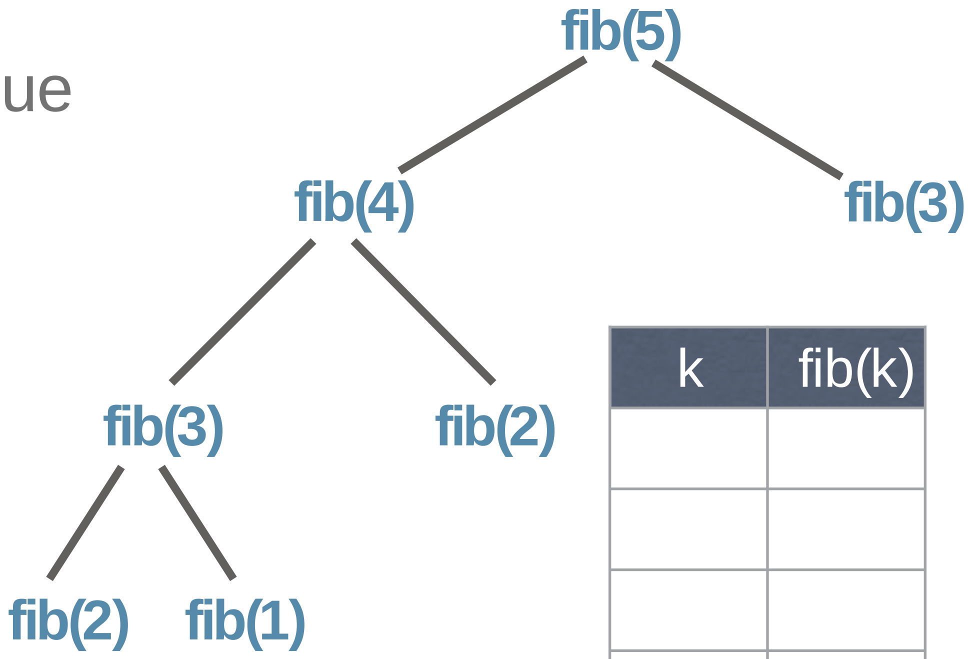


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

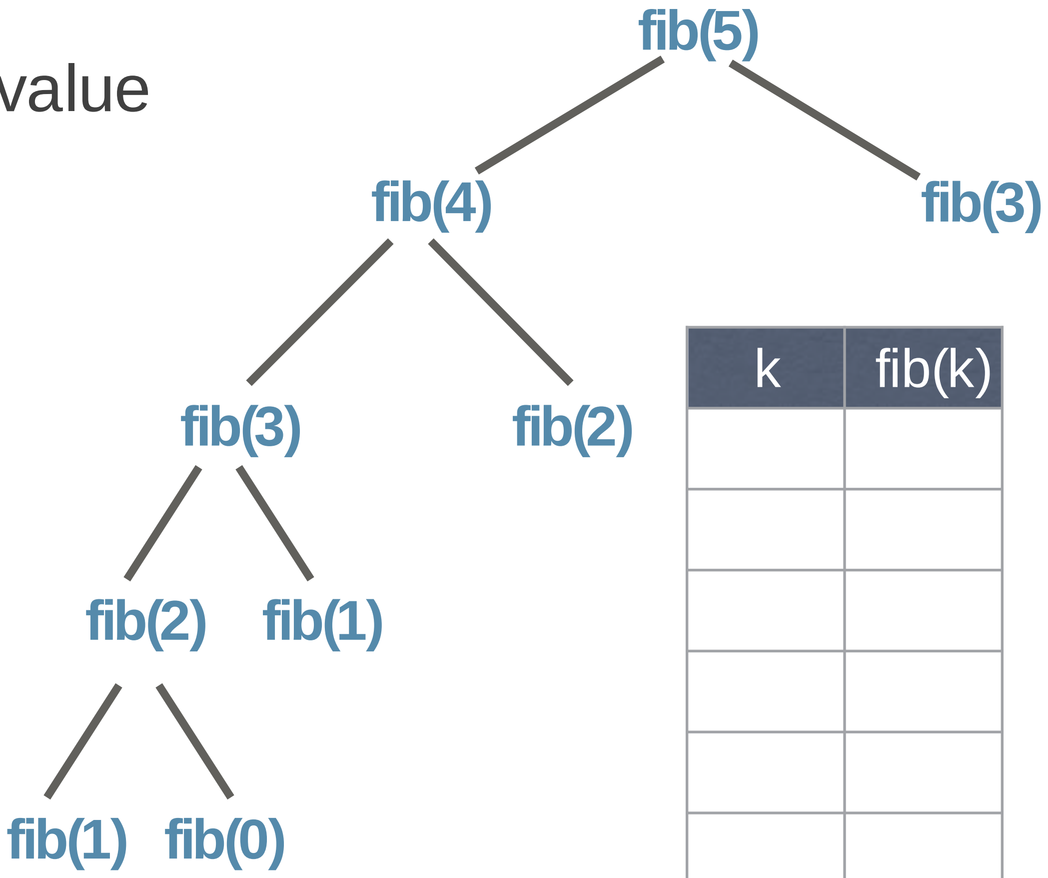


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

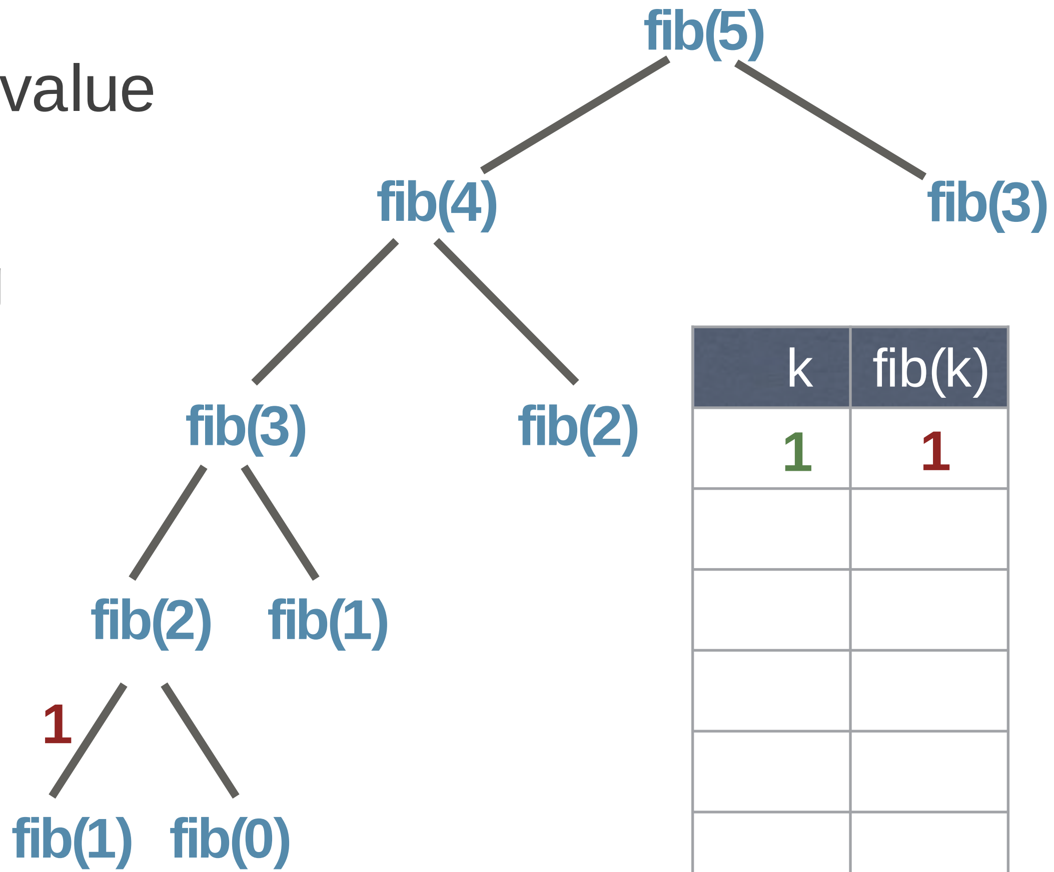


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

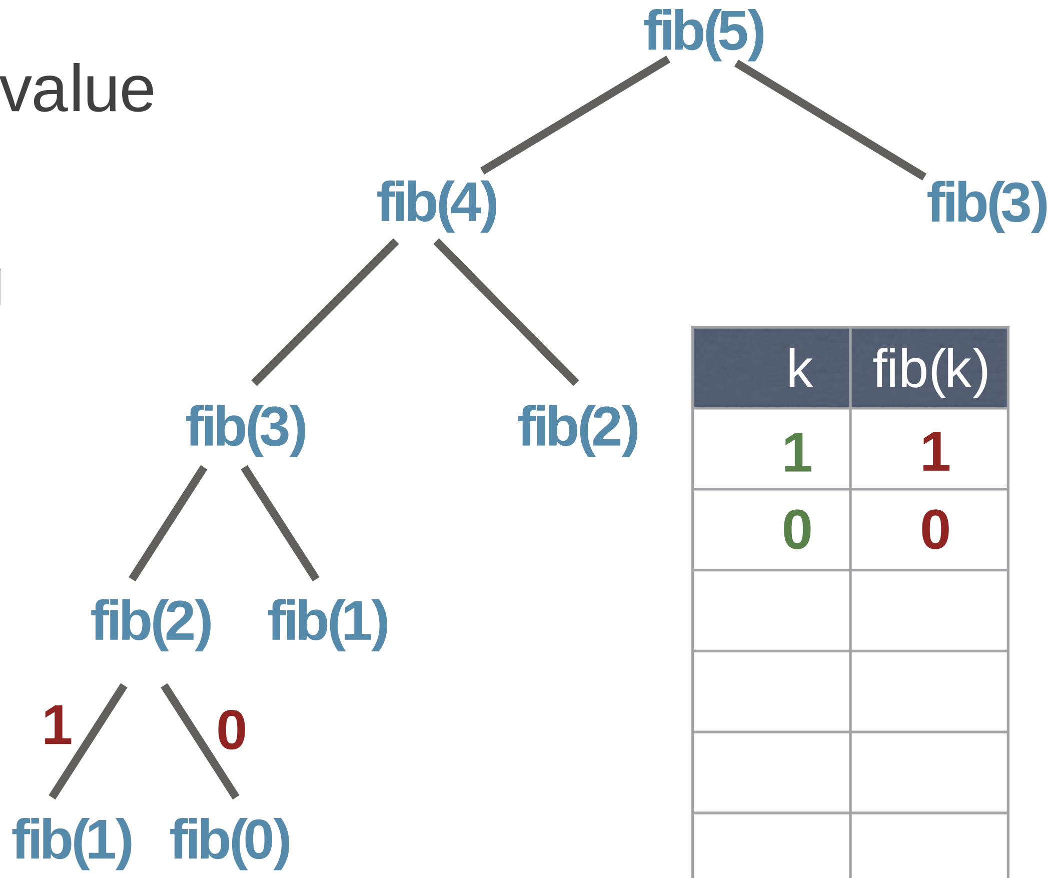


k	fib(k)
1	1

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

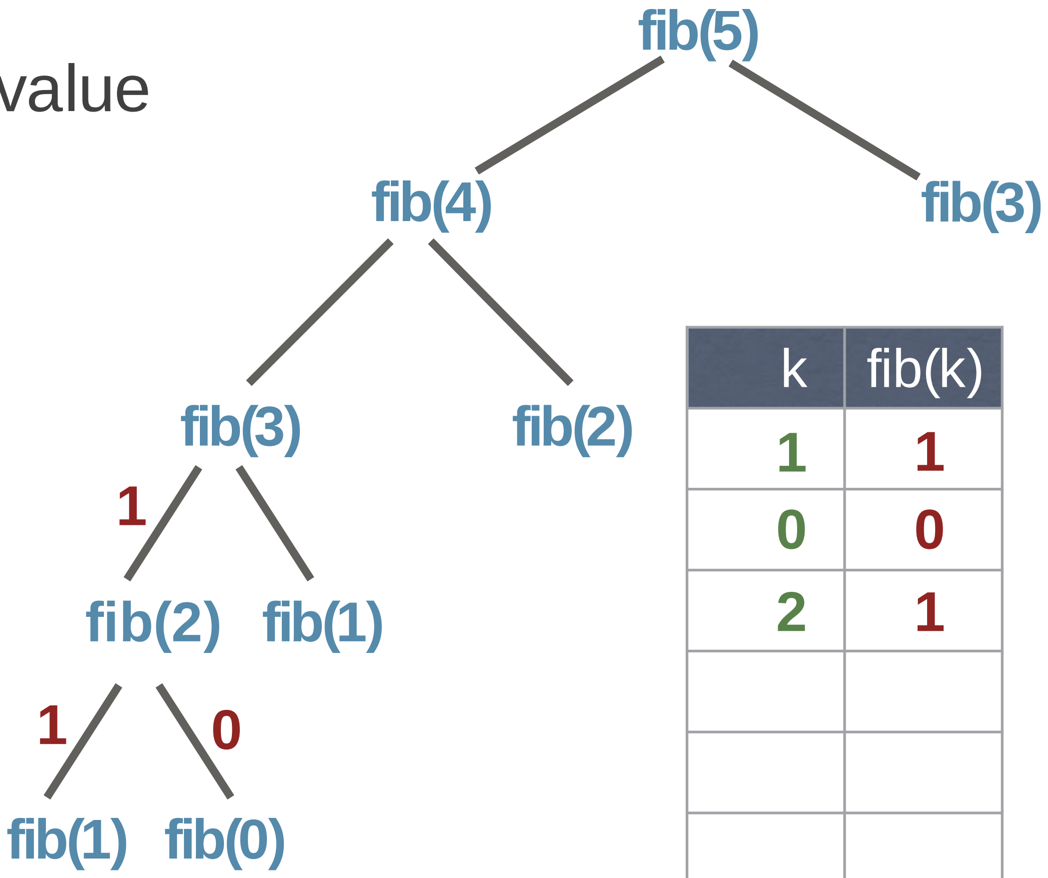


k	fib(k)
1	1
0	0

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

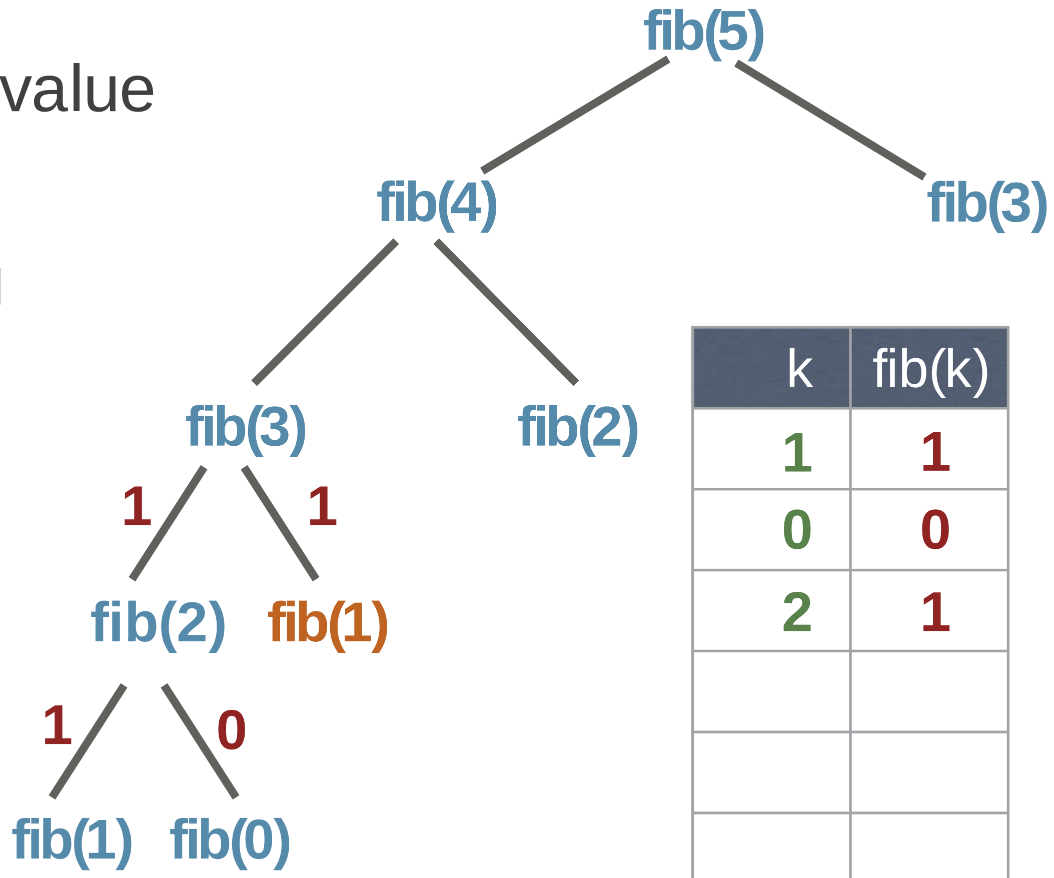


k	fib(k)
1	1
0	0
2	1

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

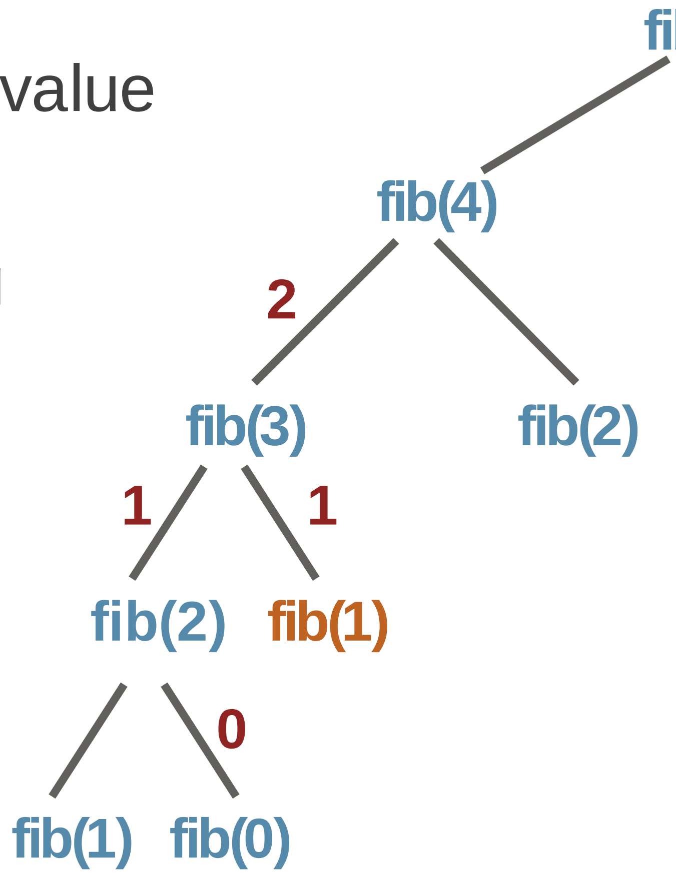


k	fib(k)
1	1
0	0
2	1

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

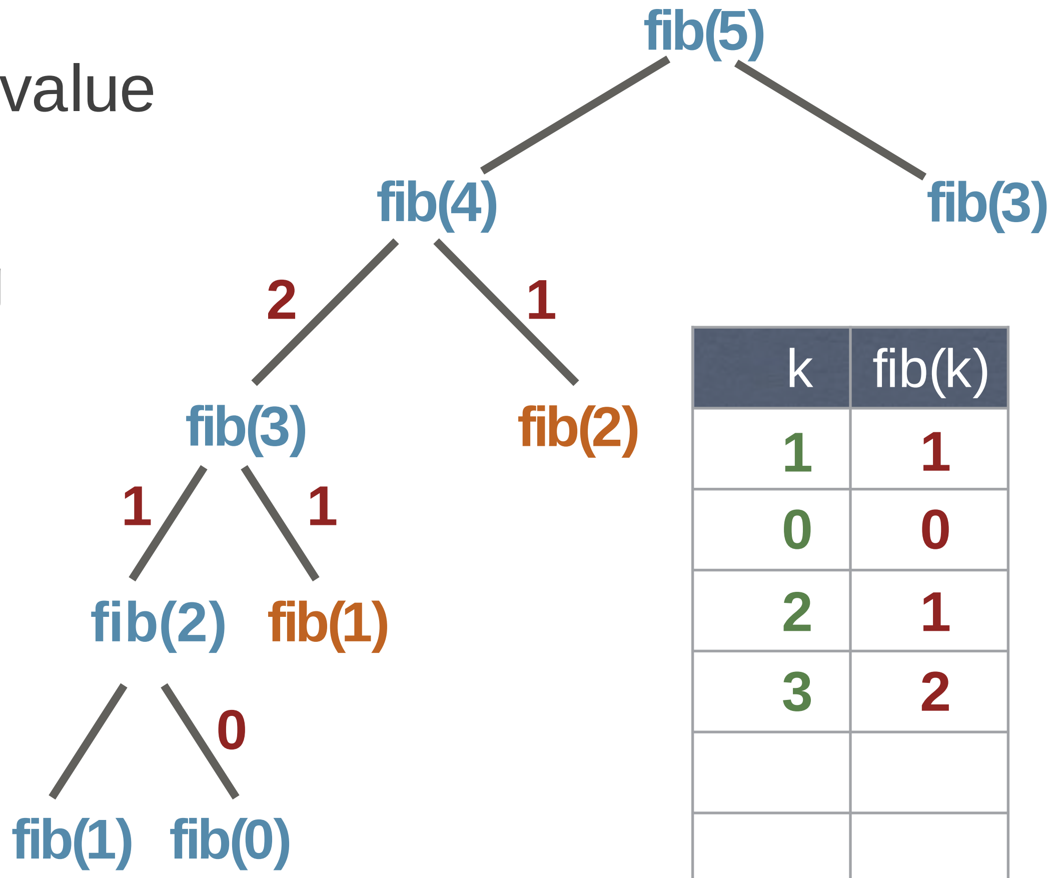


k	fib(k)
1	1
0	0
2	1
3	2

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

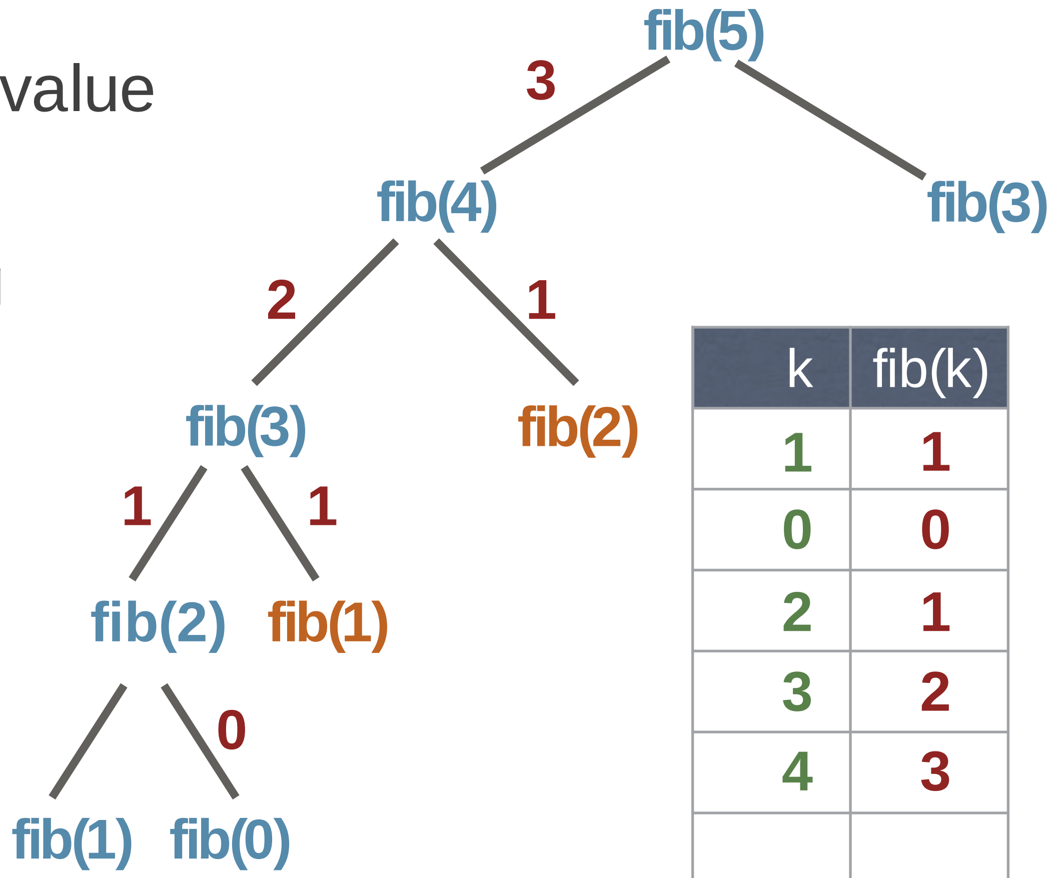


k	fib(k)
1	1
0	0
2	1
3	2

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

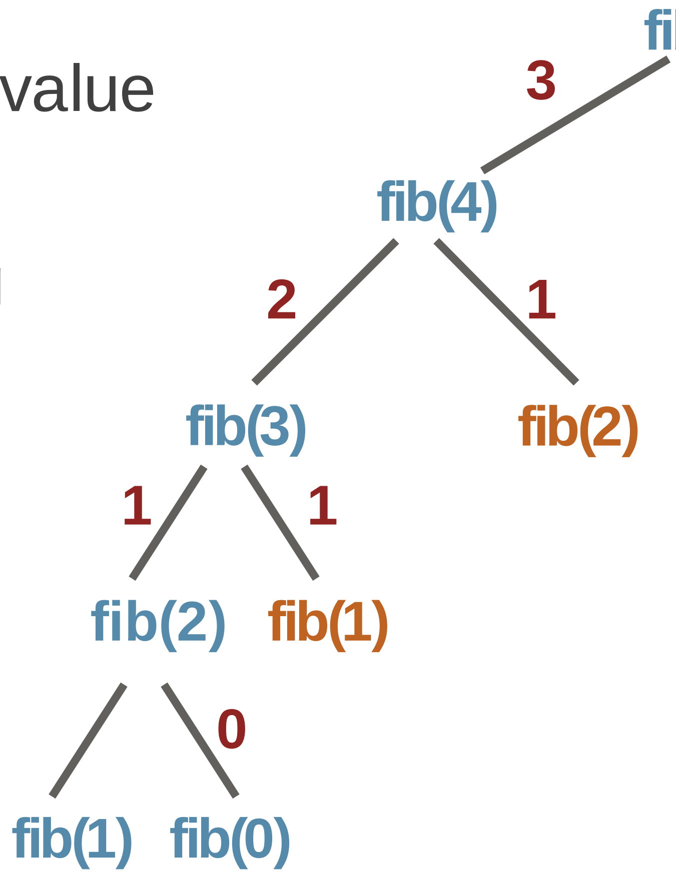


k	fib(k)
1	1
0	0
2	1
3	2
4	3

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

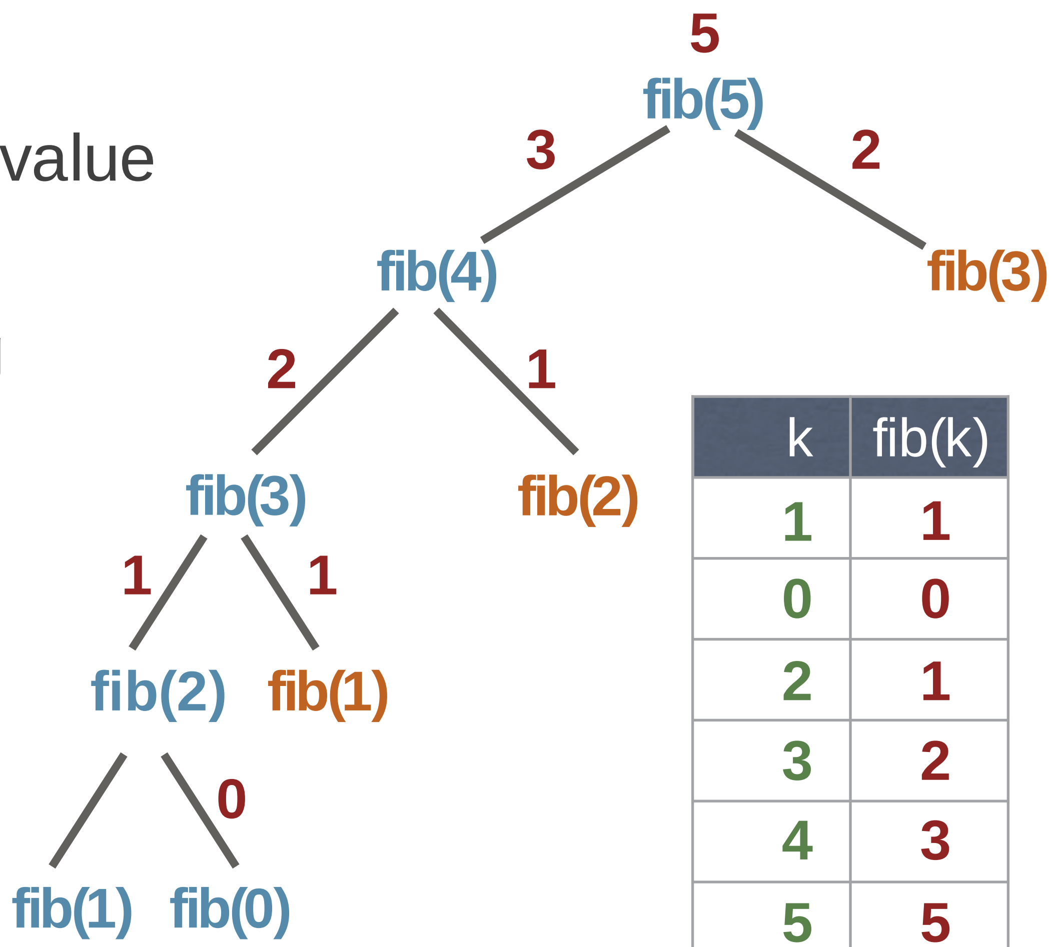


k	fib(k)
1	1
0	0
2	1
3	2
4	3

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear



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

Memoized fibonacci

```
function fib(n):
  if fibtable[n]
    return(fibtable[n])
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

In general

```
function f(x,y,z):
  if ftable[x][y][z]
    return(ftable[x][y][z])
    value = expression in terms of subproblems
  ftable[x][y][z] = value return(value)
```

Python Implementation

```
def fib(n, memo):
    if memo[n] is not None:
         return memo[n]
    if n == 1 or n == 2:
        result = 1
    else:
        result = fib(n-1, memo) + fib(n-2, memo)
    memo[n] = result
    return result
def fib_memo(n):
    memo = [None] * (n+1)
    return fib(n, memo)
fib_memo(5)
fib_memo(1000) # recursive error due to too many recursive calls. # Try bottom-up dynamic programming
```

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

fib(5)

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

fib(5)

fib(4)

fib(3)

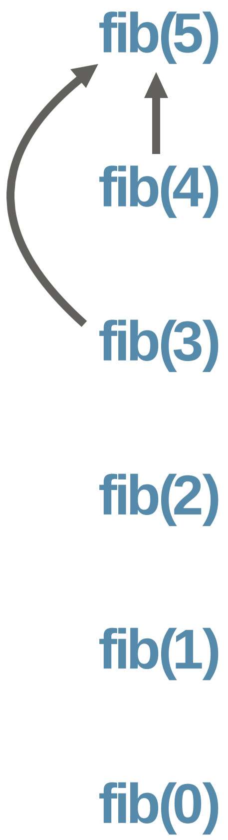
fib(2)

fib(1)

fib(0)

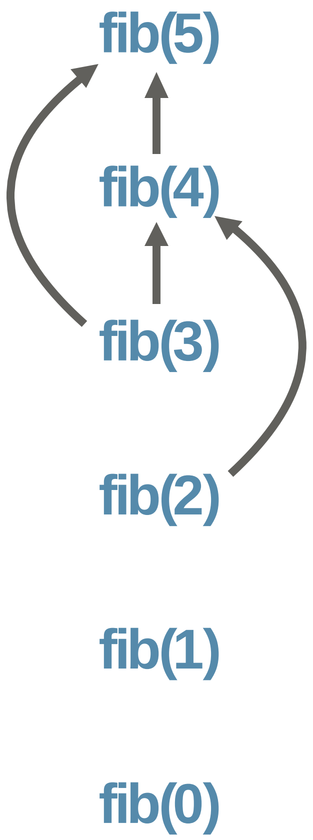
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order



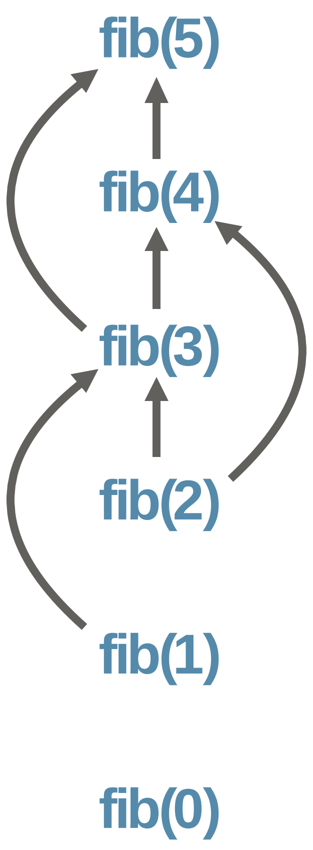
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order



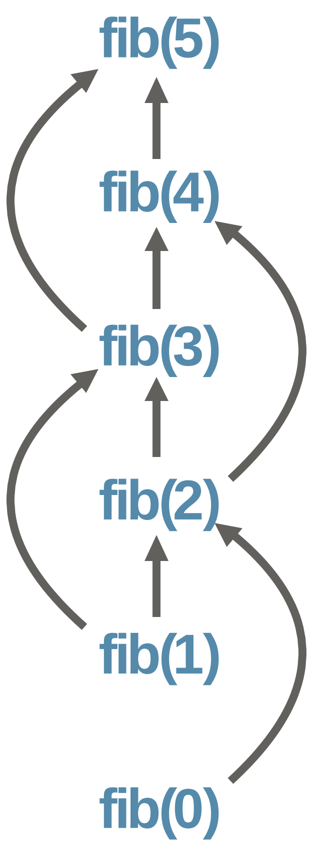
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order



Dynamic programming

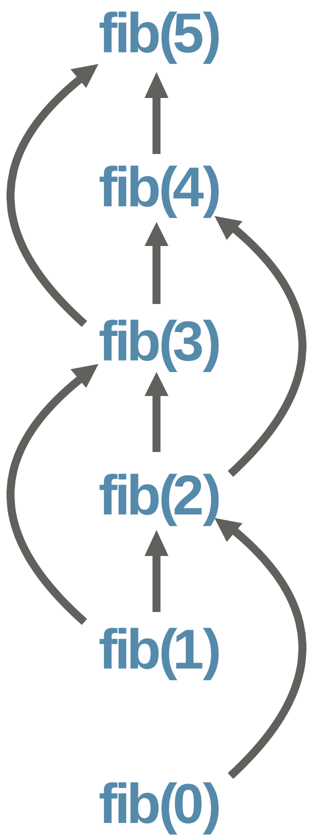
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order



Dynamic programming

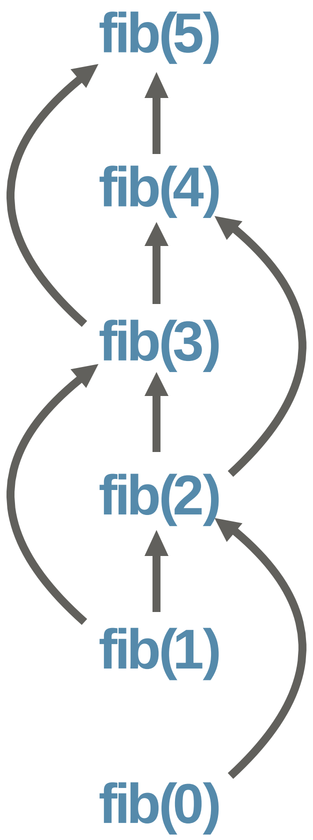
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

k	0	1	2	3	4	5
fib(k)						



Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

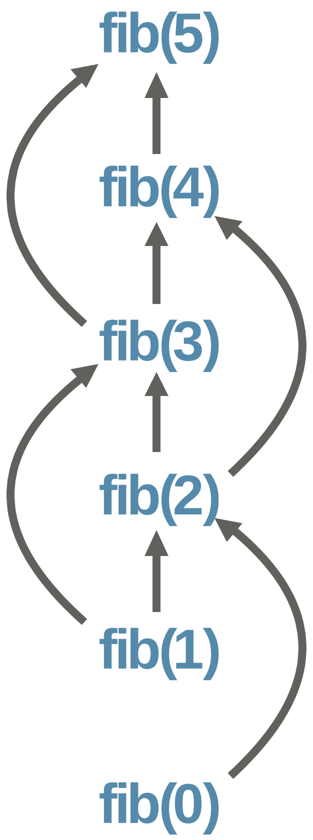


k	0	1	2	3	4	5
fib(k)	**0**					

Dynamic programming

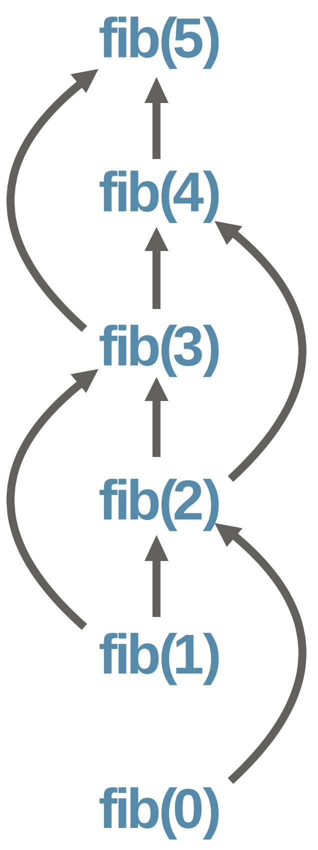
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

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



Dynamic programming

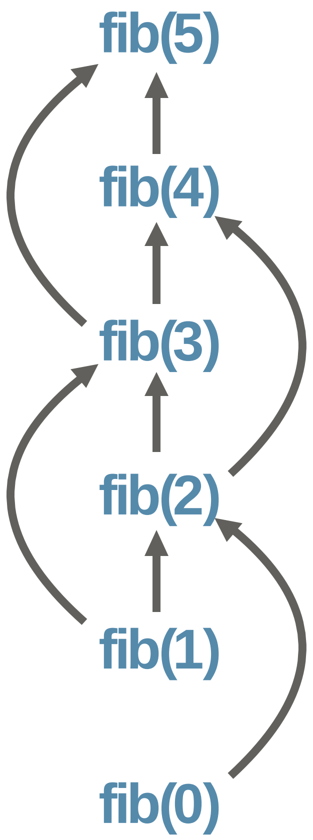
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order



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

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

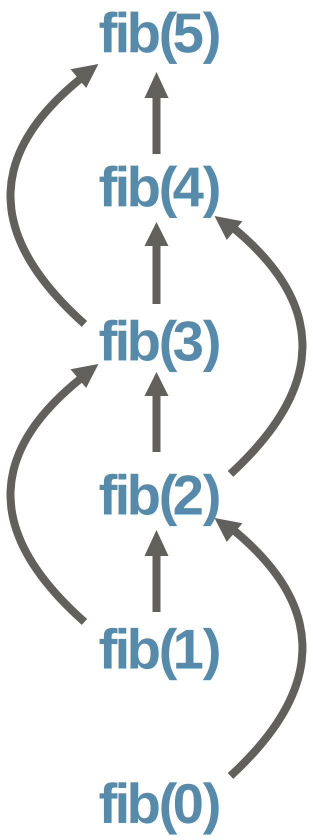


k	0	1	2	3	4	5
fib(k)	0	1	1	2		

Dynamic programming

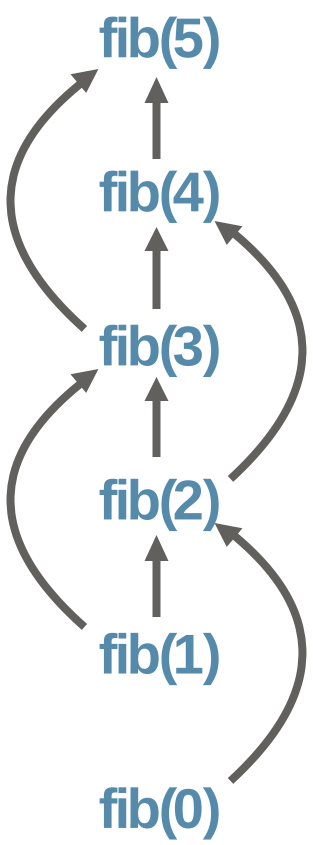
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a DAG
- Solve subproblems in topological order

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



Dynamic programming

- Anticipate what the memory table looks like
    - Subproblems are known from problem structure
    - Dependencies form a DAG
- Solve subproblems in topological order



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

Dynamic programming fibonacci

```
function fib(n):
  fibtable[0] = 0
  fibtable[1] = 1
  for i = 2,3,..n
    fibtable[i] = fibtable[i-1] +
                  fibtable[i-2]

  return(fibtable[n])
```

Python Implementation | Bottom-up DP

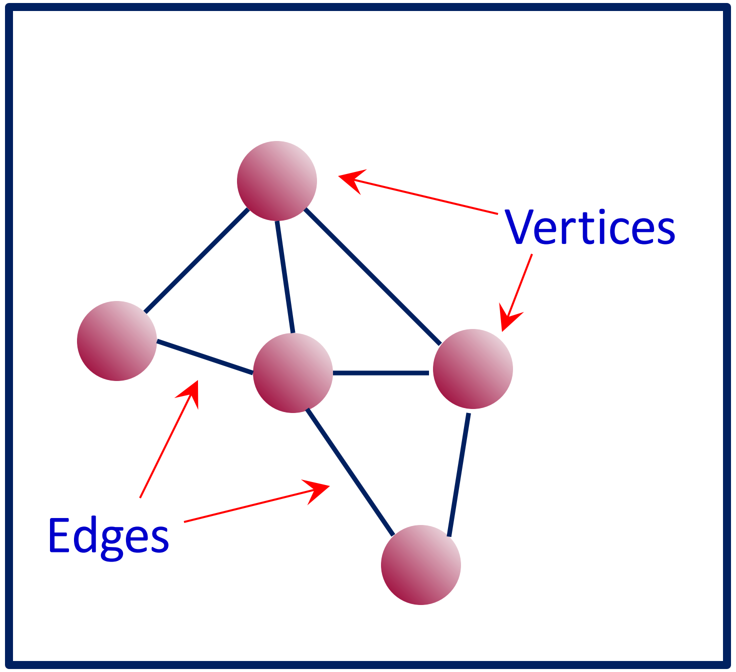
```python
def fib_bottom_up(n):
    if n ==1 or n== 2:
        return 1
    bottom_up = [None]*(n+1)
    bottom_up[1] = 1
    bottom_up[2]= 1
    for k in range(3, n+1):
        bottom_up[k] = bottom_up[k-1] + bottom_up[k-2]
    return bottom_up[n]

fib_bottom_up(5)
```

Graph Theory – A Brief!

Graph – A Graph is defined as a set of:

- V = **Nodes** (vertices)
- E = **Edges** (links, arcs) between pairs of nodes
- Denoted by $G = (V, E)$.
- Graph Size Parameters:
  - **Order** of G: number of vertices, $n = |V|$,
  - **Size** of G: number of edges, $m = |E|$.

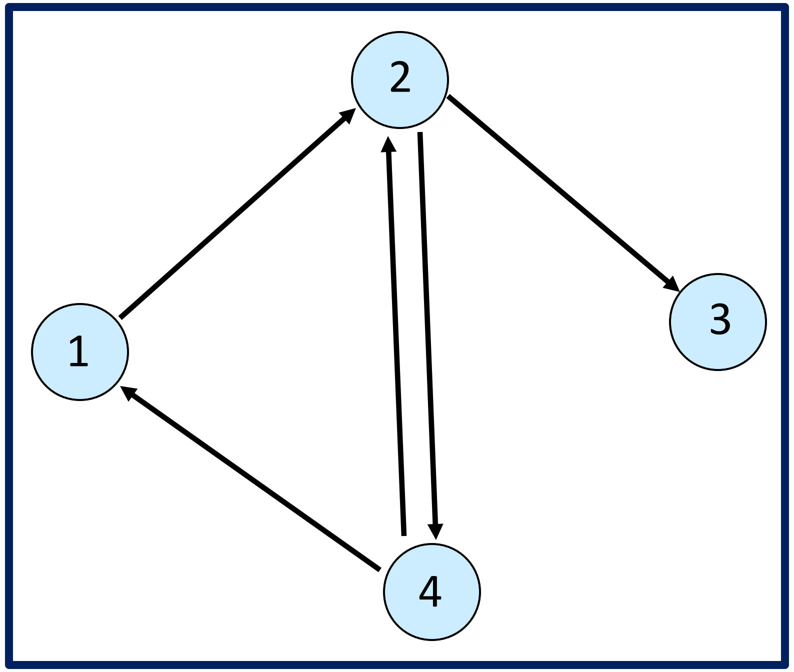


- The ***running time of algorithms*** are usually measured in terms of the order and size of a graph
- **Path** – Path represents a sequence of edges between two vertices

Directed Graph

An edge $e \in E$ of a directed graph is represented as an ordered pair (u,v), where

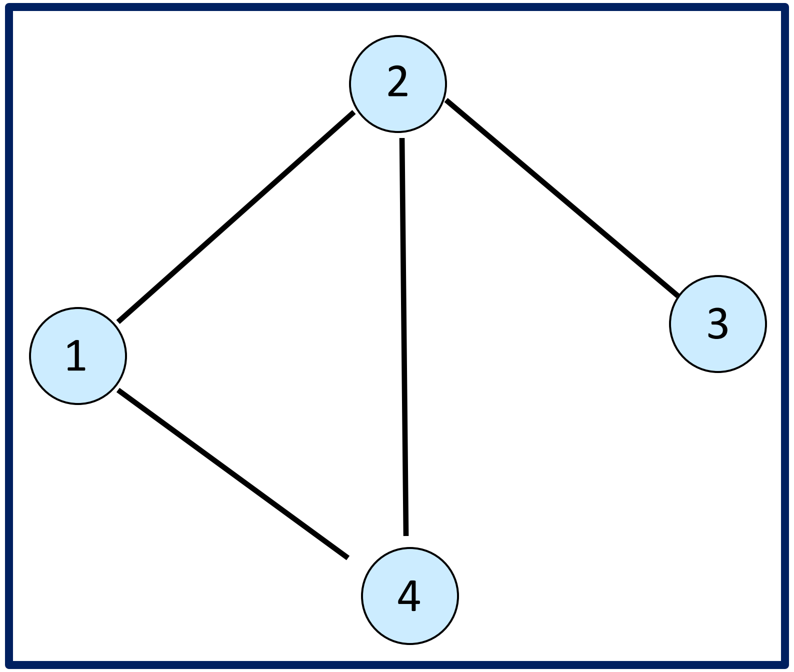
- $u, v \in V$
- u is the ***initial*** vertex
- v is the ***terminal*** vertex.
- u / v



V = { 1, 2, 3, 4}, | V | = 4
E = {(1,2), (2,3), (2,4), (4,1), (4,2)}, | E | = 5

Undirected Graph

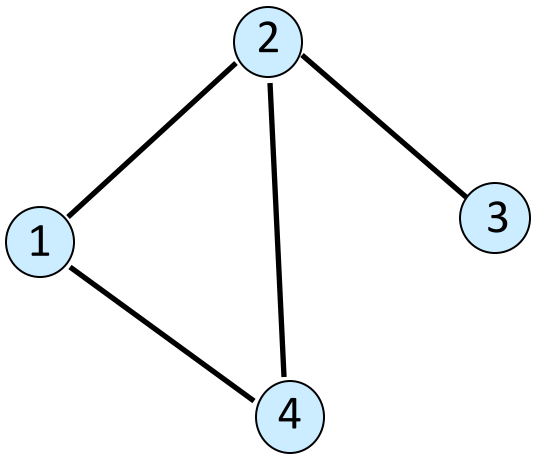
An edge $e \in E$ of an undirected graph is represented as an unordered pair **(u,v) = (v,u),** where $u, v \in V$, $u \neq v$



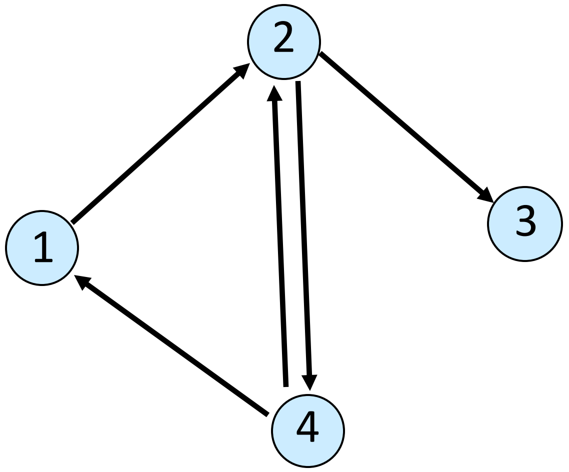
$V = \{1, 2, 3, 4\}$, $|V| = 4$
$E = \{(1,2), (2,3), (2,4), (4,1)\}$, $|E| = 4$

Degree of a Vertex

Degree of a vertex in an undirected graph is the number of edges incident on it. In a directed graph, the *out degree* of a vertex is the number of edges leaving it and the *in degree* is the number of edges entering it.



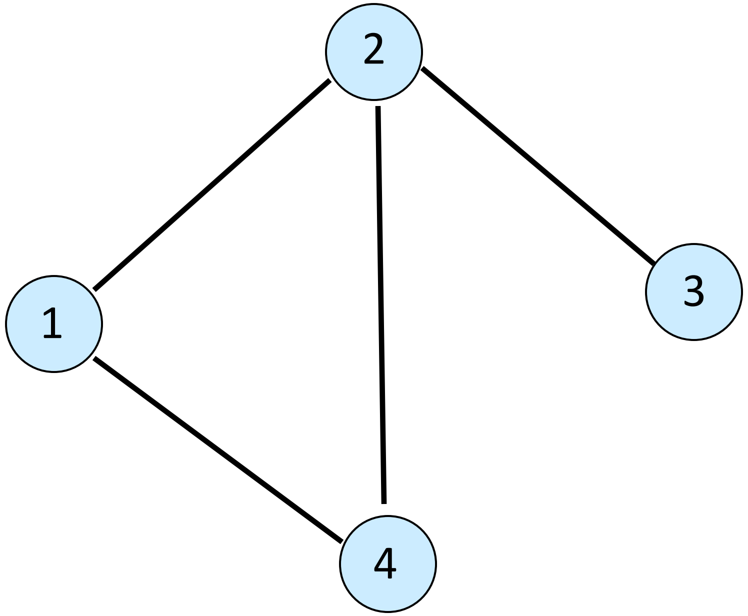
The *degree* of vertex 2 is 3



The *in degree* of vertex 2 is 2 and the *out degree* of vertex 3 is 0

Adjacent/Neighbor Nodes

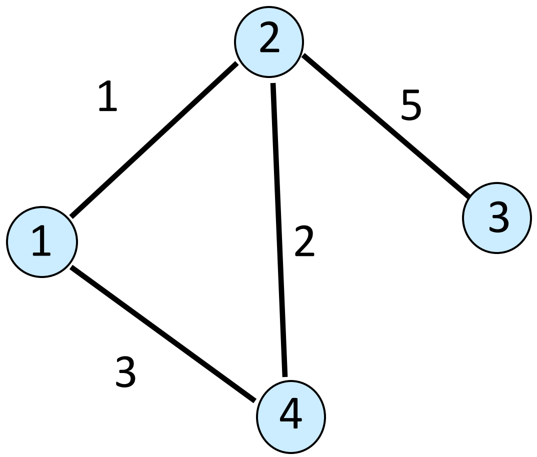
If an edge $e=\{u,v\} \in E$, u and v are **adjacent** or **neighbors**

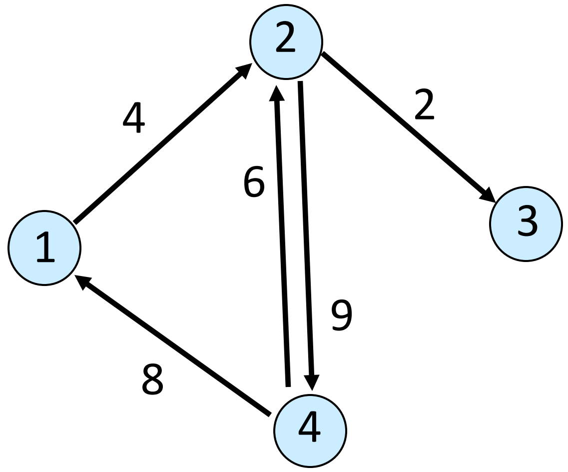


(1,2), (2,3), (2,4), (4,1) are all Adjacent/Neighbor Nodes

Weighted Graph

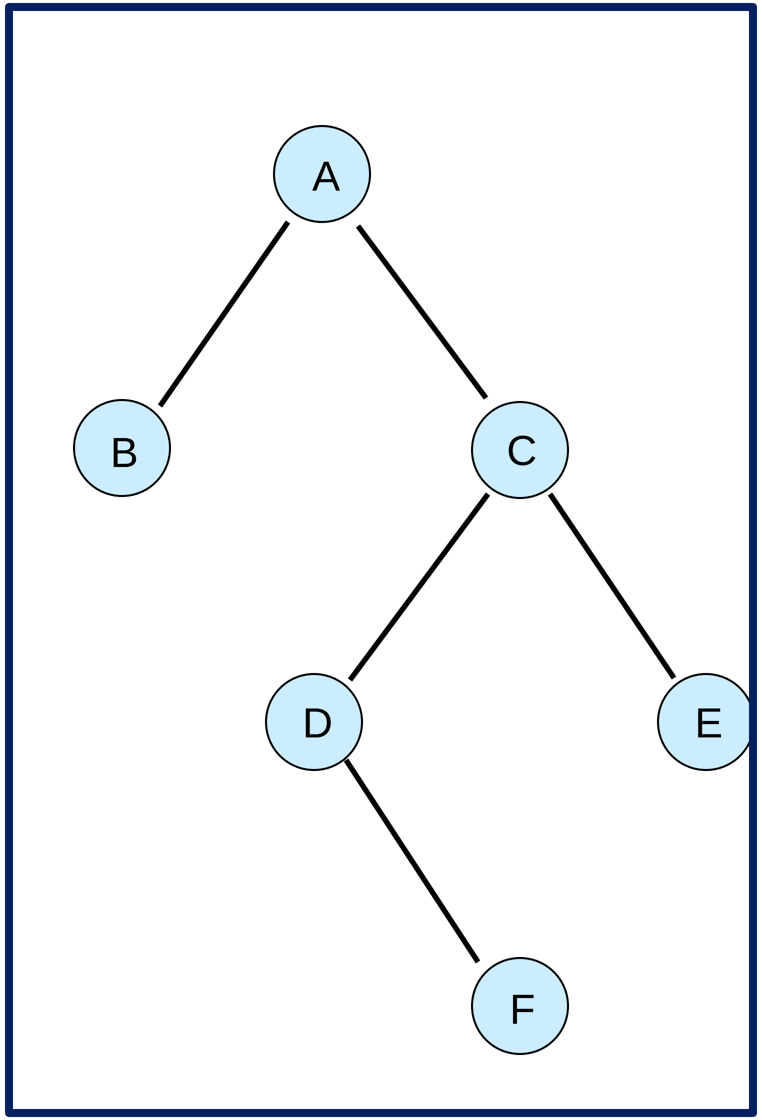
A *weighted graph* is a graph for which each edge has an associated *weight*





Trees

- An undirected graph is a **tree** if it is connected and does not contain a cycle.
- For an undirected tree Graph G, the following statements are equivalent:
  - Any two vertices in G are connected by unique simple path
  - G is connected, but if any edge is removed from E, the resulting graph is disconnected
  - G is connected, and $|E| = |V| - 1$
  - G is acyclic, and $|E| = |V| - 1$
  - G is acyclic, but if any edge is added to E, the resulting graph contains a cycle



Graph Representation: Adjacency Matrix

Adjacency Matrix $|V| \times |V|$ matrix with $A_{uv} = 1$ if (u, v) is an edge

- Symmetric matrix for *undirected* graphs (not for directed graphs)
- Space: proportional to $|V|^2$.
  - Not efficient for ***sparse graphs***
  - Algorithms might have longer running time if this representation is used
- Checking if (u, v) is an edge consumes O(1) time.
- Identifying all edges consumes $O(|V|^2)$ time.

Graph Representation: Adjacency List

- Two common data structures for representing graphs:
  - Adjacency lists
  - Adjacency matrix

Adjacency List - Node indexed array of lists

- Two representations of each edge
- Space proportional to $|E| + |V|$
- Checking if (u, v) is an edge consumes $O(deg(u))$ time
- Identifying all edges takes $O(|E|+|V|)$ time
- Requires $O(|E|+|V|)$ space. *Good* for dealing with *sparse* graphs

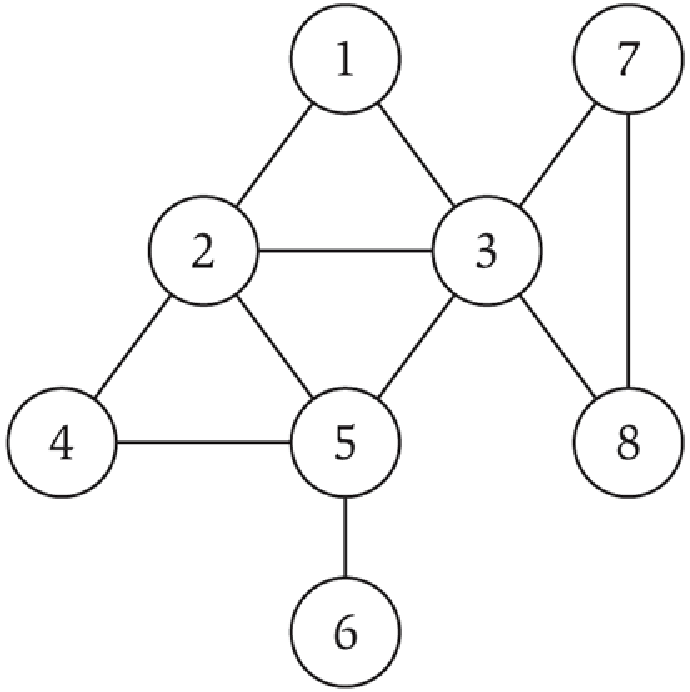
Graph Representation

Adjacency Matrix

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Adjacency List

1	2	3			
2	1	3	4	5	
3	1	2	5	7	8
4	2	5			
5	2	3	4	6	
6	5				
7	3	8			
8	3	7			



Basic Operations

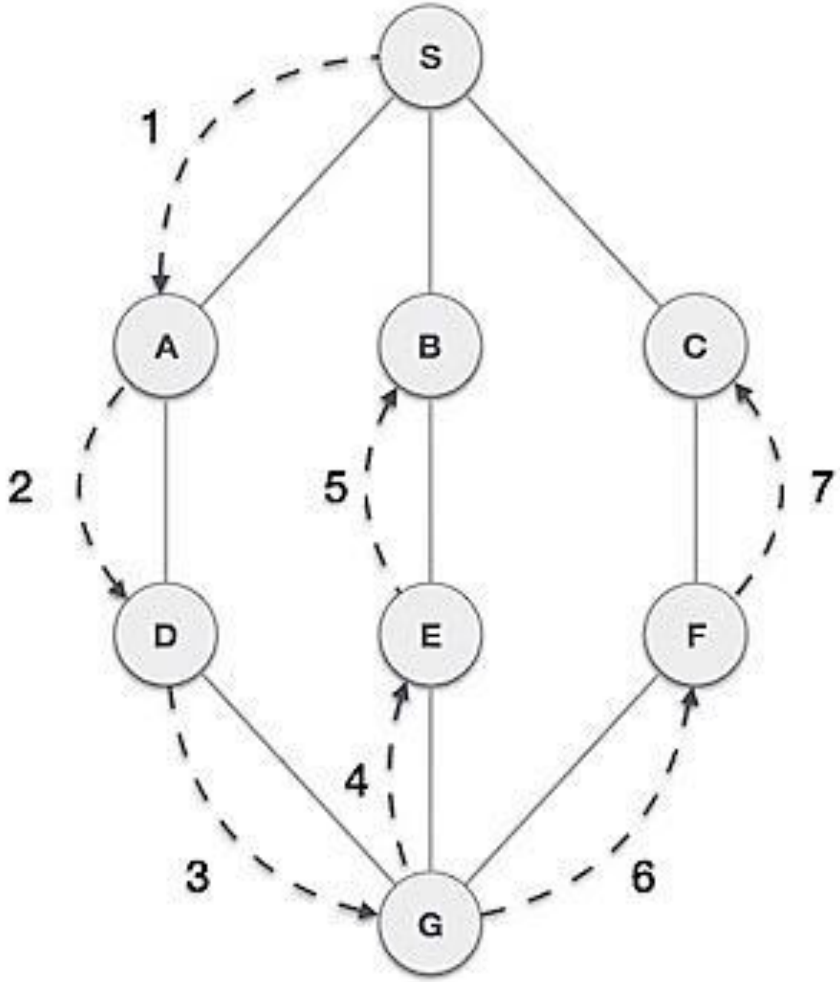
1 SAMUEL 7:12

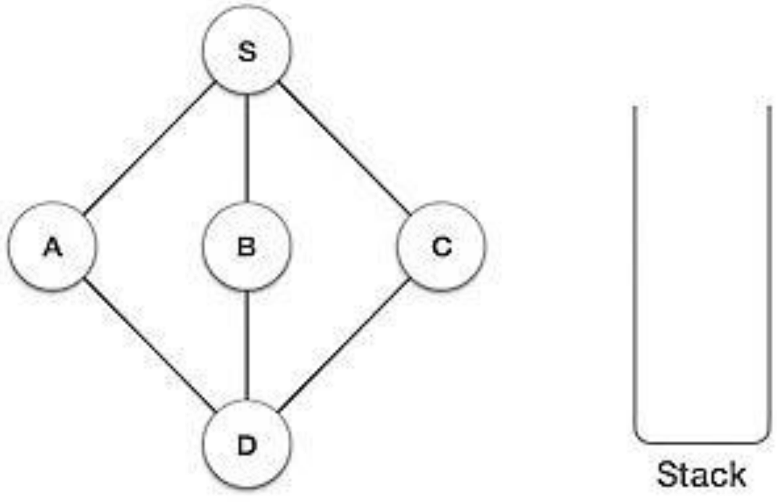
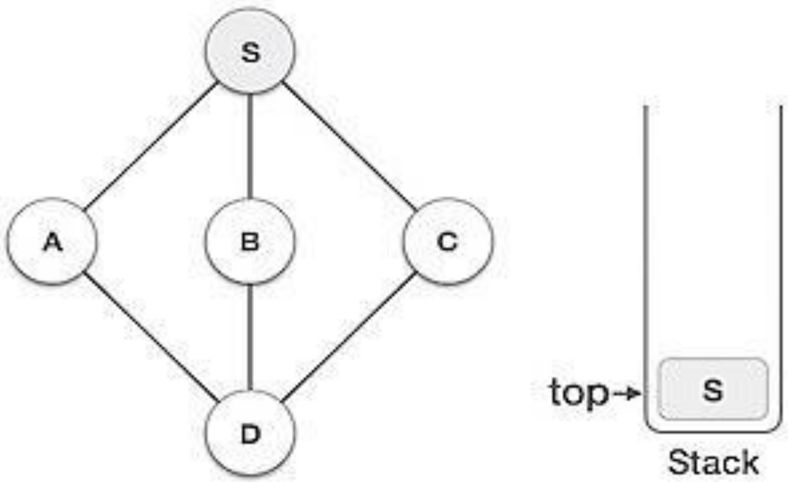
- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

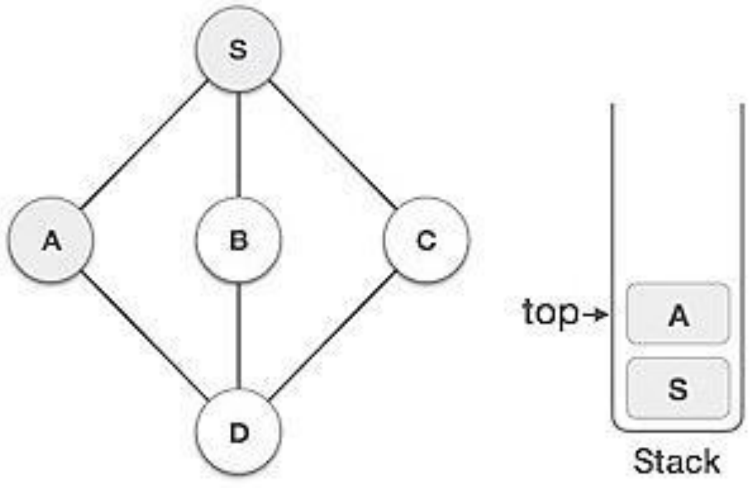
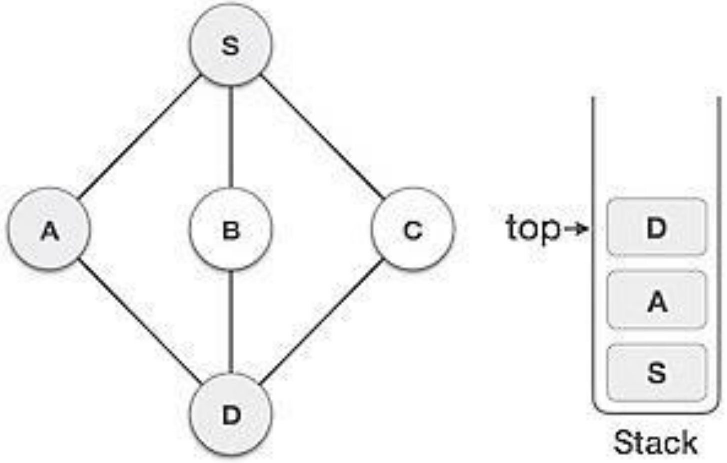
Depth First Traversal

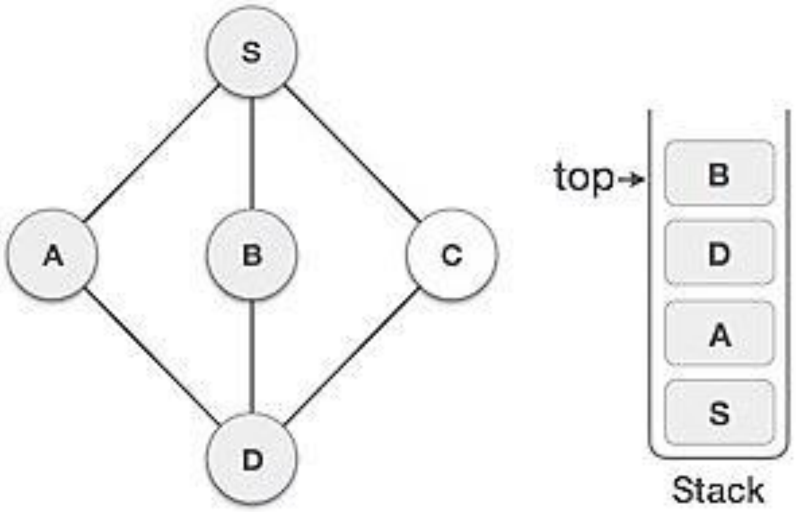
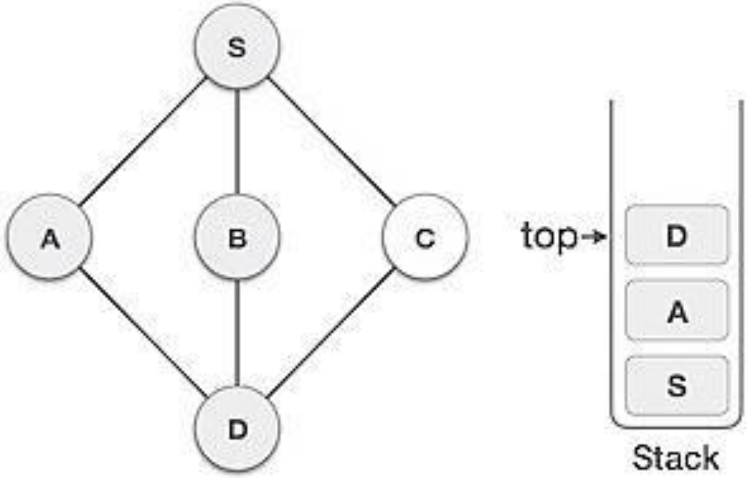
- Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

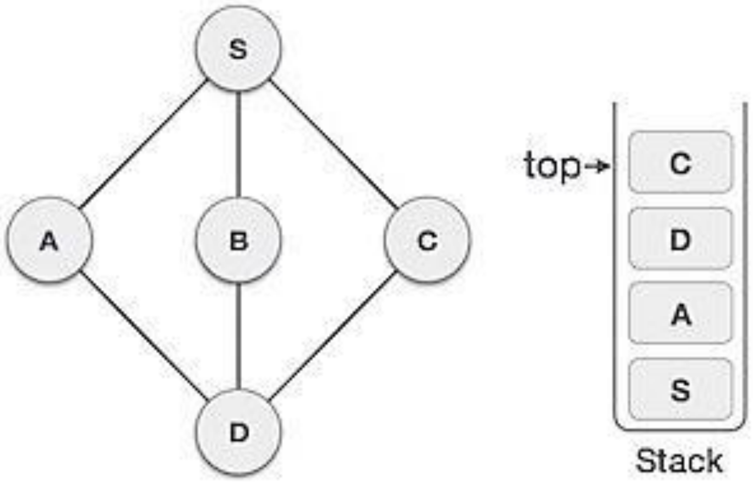
Rule 1 – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
Rule 2 – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
Rule 3 – Repeat Rule 1 and Rule 2 until stack is empty.


S
1
A
B
C
2
5
7
D
E
F
4
3
6
G

Step	Traversal	Description
1.		Initialize the stack
2.		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.

3.		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4.		Visit **D** and mark it visited and put onto the stack. Here we have **B** and **C** nodes which are adjacent to **D** and both are unvisited. But we shall again choose in alphabetical order.

5.		We choose **B**, mark it visited and put onto stack. Here **B** does not have any unvisited adjacent node. So we pop **B** from the stack.
6.		We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of stack.

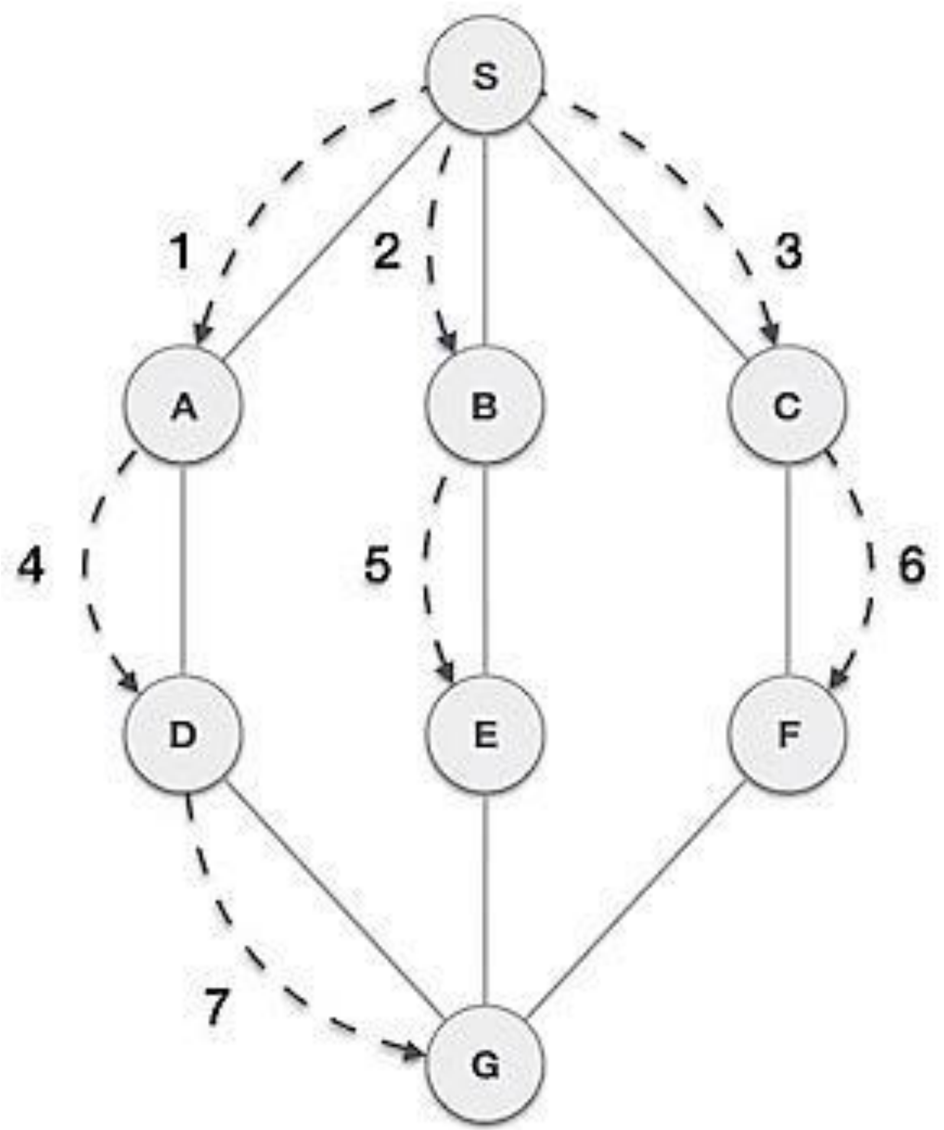
7.		Only unvisited adjacent node from **D** is **C** now. So, we visit **C**, mark it visited and put it onto the stack.

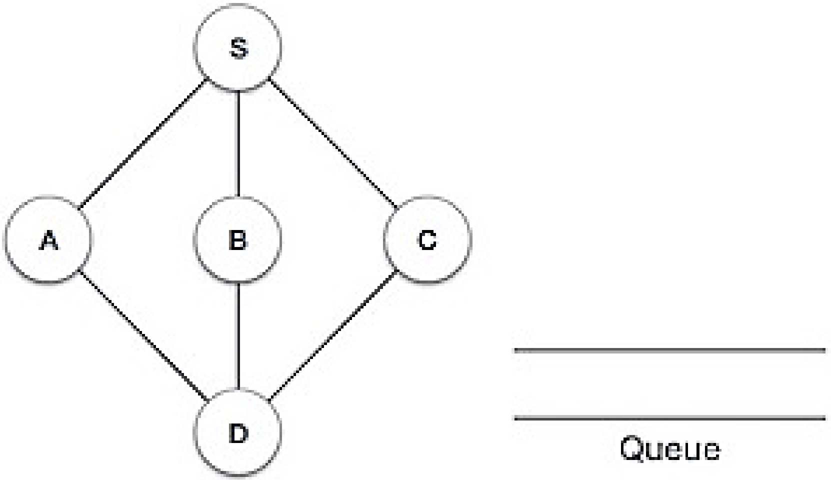
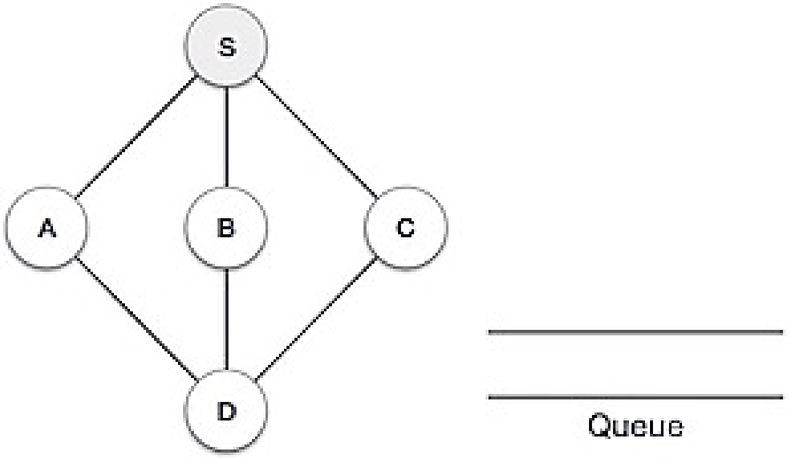
1 SAMUEL 7:12

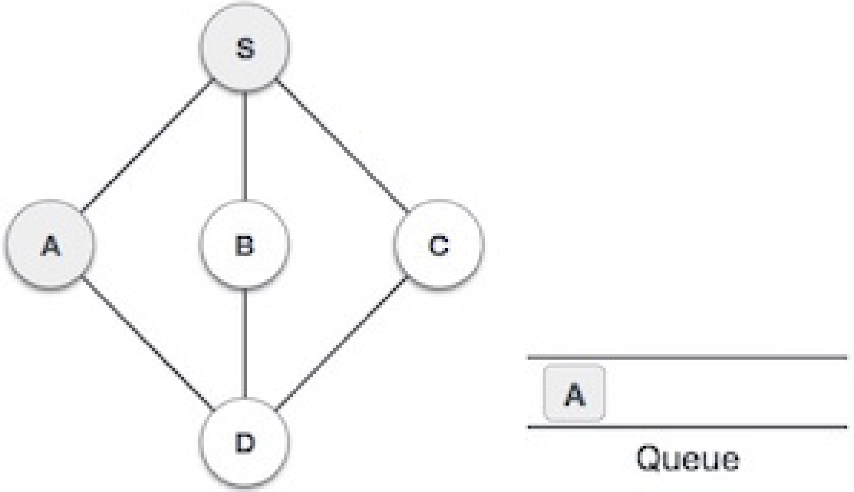
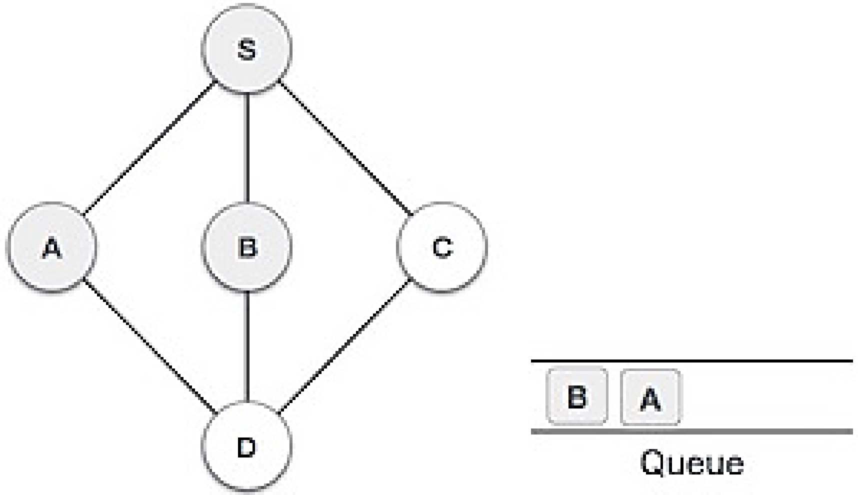
Breadth First Traversal

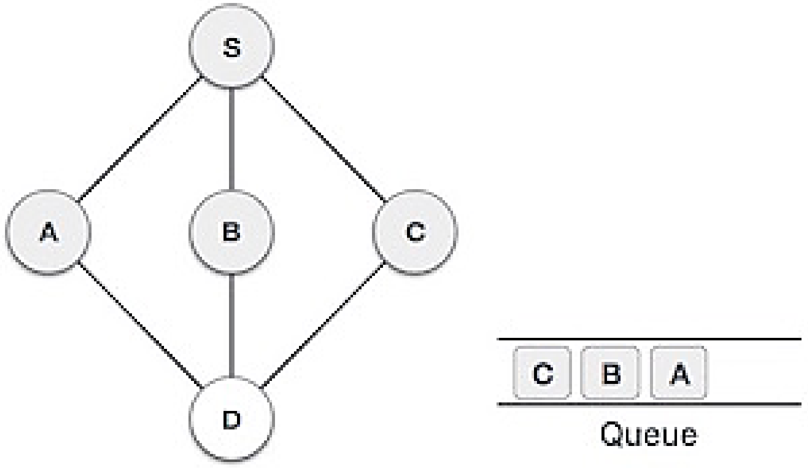
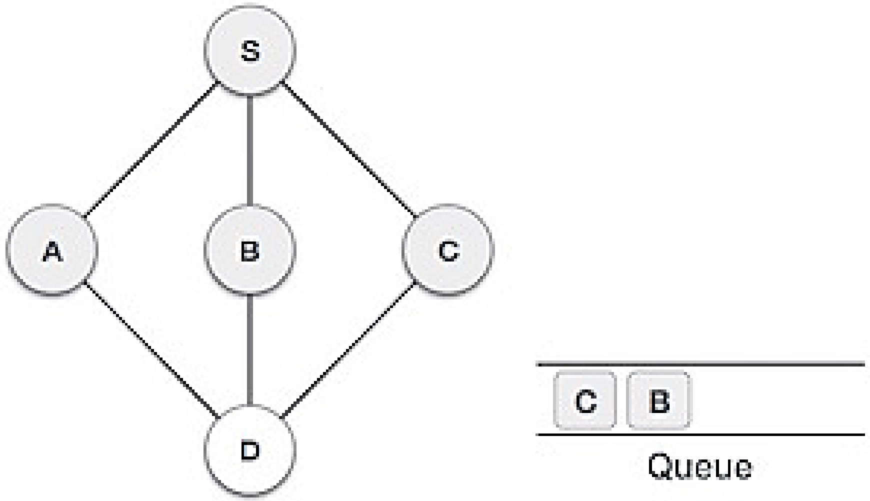
- Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

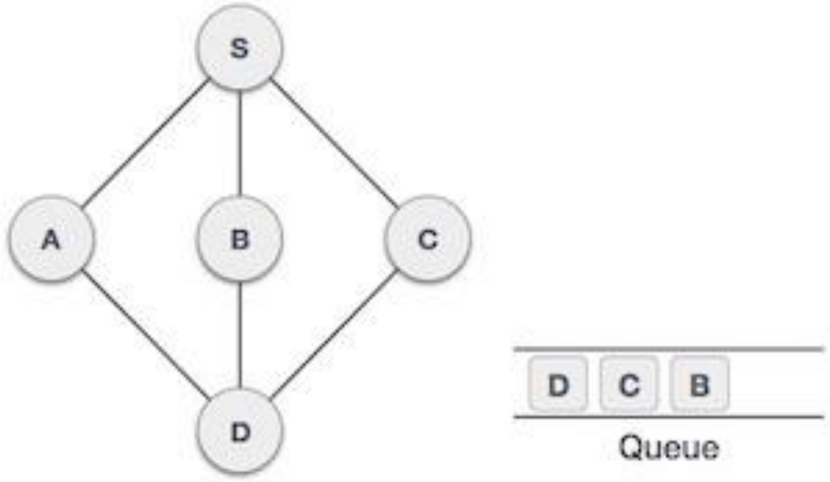
- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.


S
1
2
3
A
B
C
4
5
6
D
E
F
7
G

1		Initialize the queue.
2		We start from visiting **S**(starting node), and mark it visited.

3		We then see unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A** mark it visited and enqueue it.
4		Next unvisited adjacent node from **S** is **B**. We mark it visited and enqueue it.

5		Next unvisited adjacent node from **S** is **C**. We mark it visited and enqueue it.
6		Now **S** is left with no unvisited adjacent nodes. So we dequeue and find **A**.

7		From **A** we have **D** as unvisited adjacent node. We mark it visited and enqueue it.

Motivation

- Many algorithms use a graph representation to represent data or the problem to be solved

Examples:

- Cities with distances between
- Roads with distances between intersection points
- Course prerequisites
- Network
- Social networks

Social Network Graph Example

1 SAMUEL 7:12

```
from collections import deque

class SocialNetwork:
  def __init__(self):
        self.graph = {}

  def add_user(self, user):
    if user not in self.graph:
        self.graph[user] = set()

  def add_connection(self, user1, user2):
    self.add_user(user1)
    self.add_user(user2)
    self.graph[user1].add(user2)
    self.graph[user2].add(user1)
```

```
def mutual_friends(self, user1, user2):
  if user1 not in self.graph or user2 not in self.graph:
  return "Invalid users"

  queue = deque([(user1, [])])
  visited = set()

  while queue:
  current_user, path = queue.popleft()

  if current_user == user2:
        return path

  for friend in self.graph[current_user]:
        if friend not in visited:
        queue.append((friend, path + [friend]))
        visited.add(friend)

  return "No mutual friends"
```

Social Network Graph Example

```python
# Example usage:
social_network = SocialNetwork()
social_network.add_connection(”John", ”Luke")
social_network.add_connection(”Joe", "Diana")
social_network.add_connection(”John", ”Diana")
social_network.add_connection(”Diana", ”Karen")

user1 = ”John"
user2 = ”Karen"
result = social_network.mutual_friends(user1, user2)
print(f"Mutual friends between {user1} and {user2}:
{result}")
```

Summary

- Memoization
  - Store values of subproblems in a table
  - Look up the table before making a recursive call
- Dynamic programming:
  - Solve subproblems in topological order of dependency
    - Dependencies must form a dag (why?) Iterative evaluation

Exercise

You are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. The weight of a path is the sum of the weights of the edges in the path. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What is the running time of your algorithm?