Dynamic Programming

Introduction

Prepared by Pawan R.

Inductive definition

Recursive program

Inductive definition

Fact(n)

Fact(0)=1

Fact(n)= n-1 * Fact(n)

sort(A)

sort([])=[]

sort(a,b,c,d)= fit(d, sort[a,b,c])

Recursive program

```
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)

sort([b,c,d]) is a subproblem of sort([a,b,c,d])
So are sort([c,d]), sort([b,c])….

Fibonacci numbers

Inductive definition

fib(0) = 0

fib(1) = 1

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

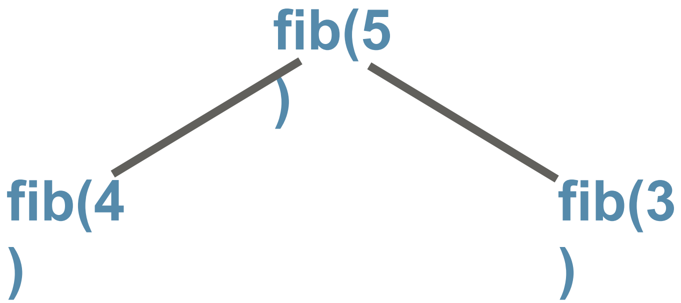
Recursive Program

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

 return(value)
```

Prepared by Pawan R.

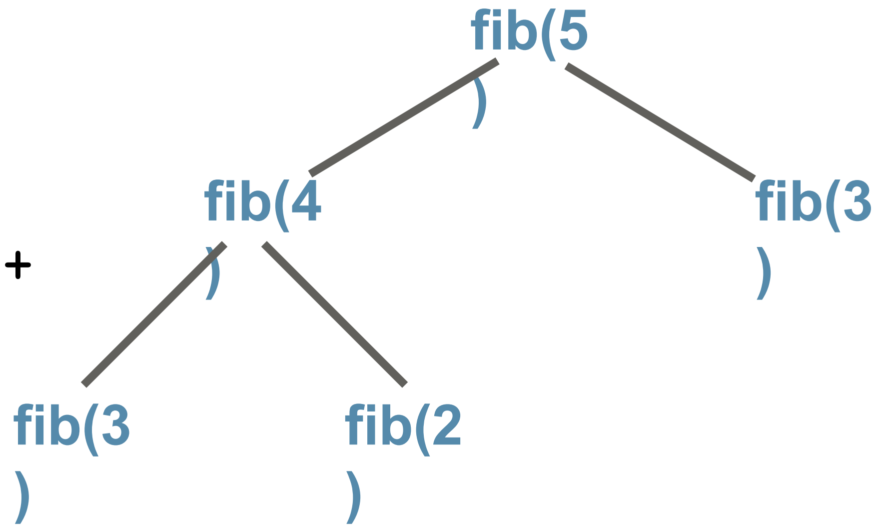
Computing fib(5)

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



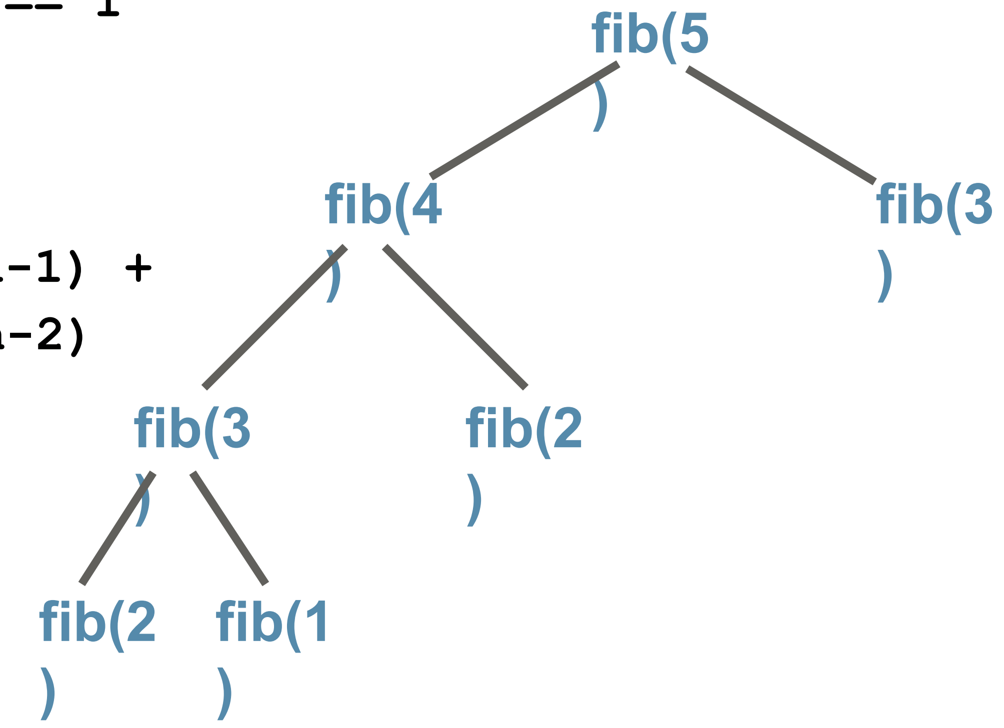
Computing fib(5)

```
function fib(n):
   if n == 0 or n == 1
     value = n

   else
     value = fib(n-1) +
             fib(n-2)

   return(value)
```



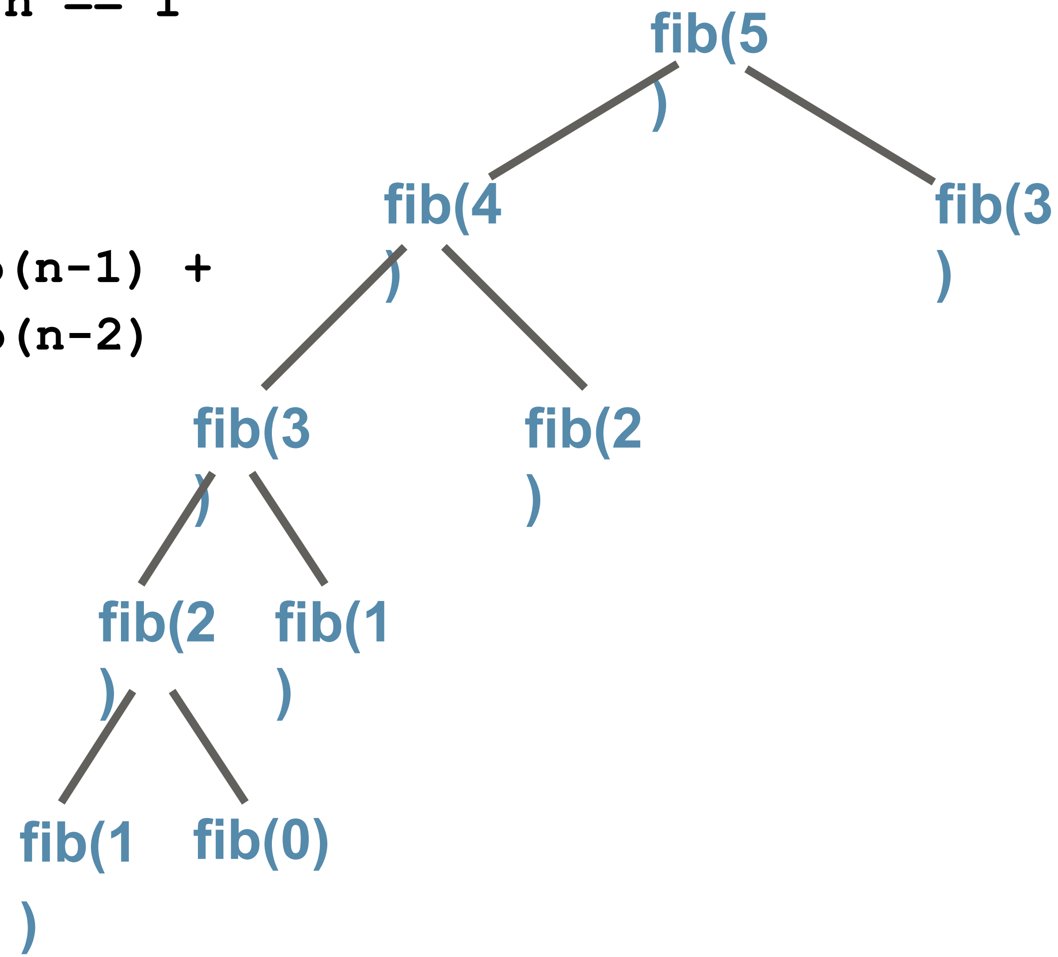
Computing fib(5)

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



Computing fib(5)

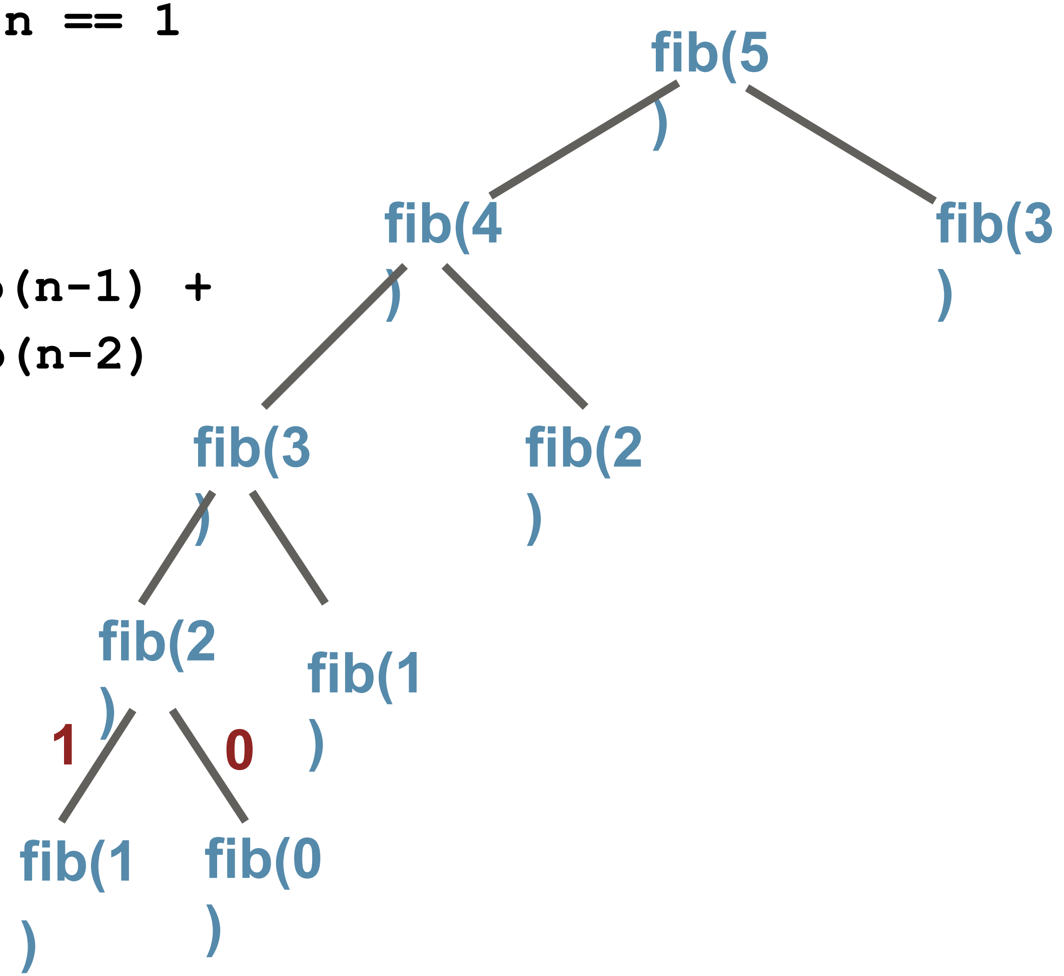
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

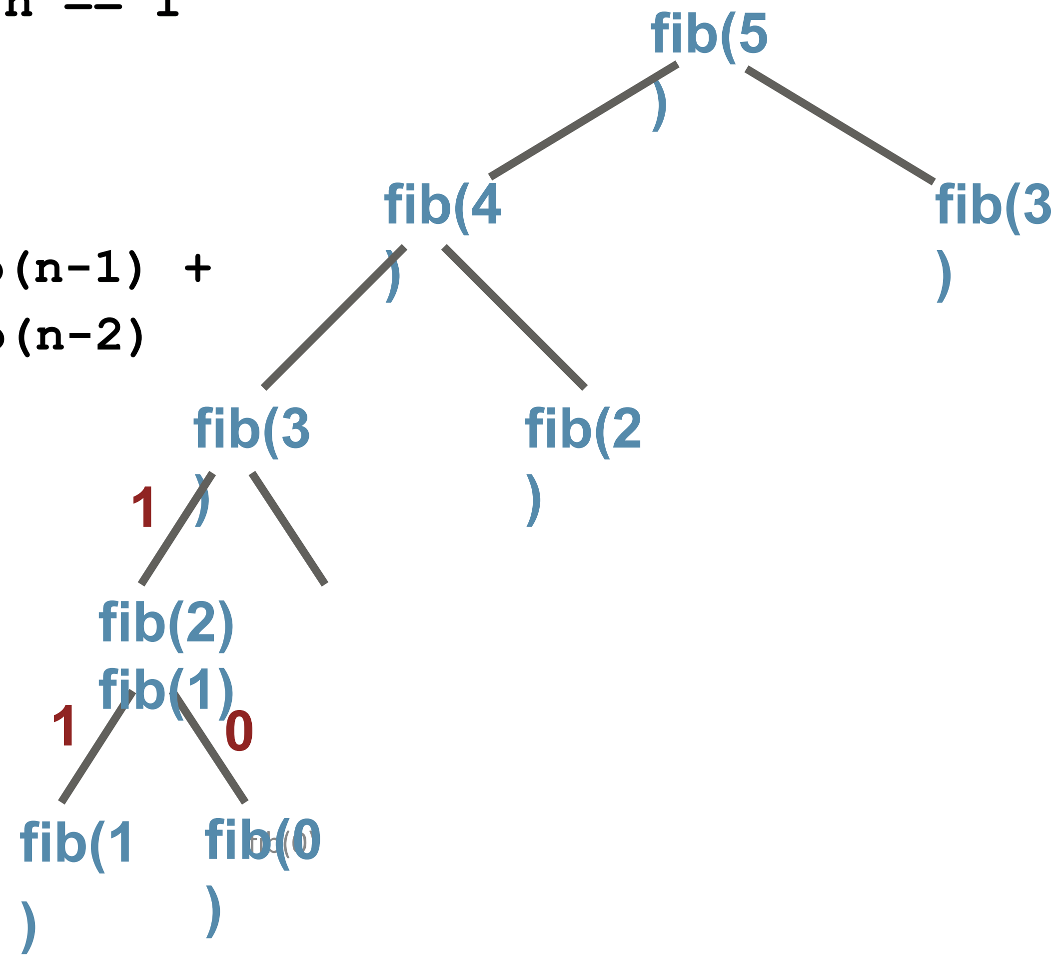
Computing fib(5)

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



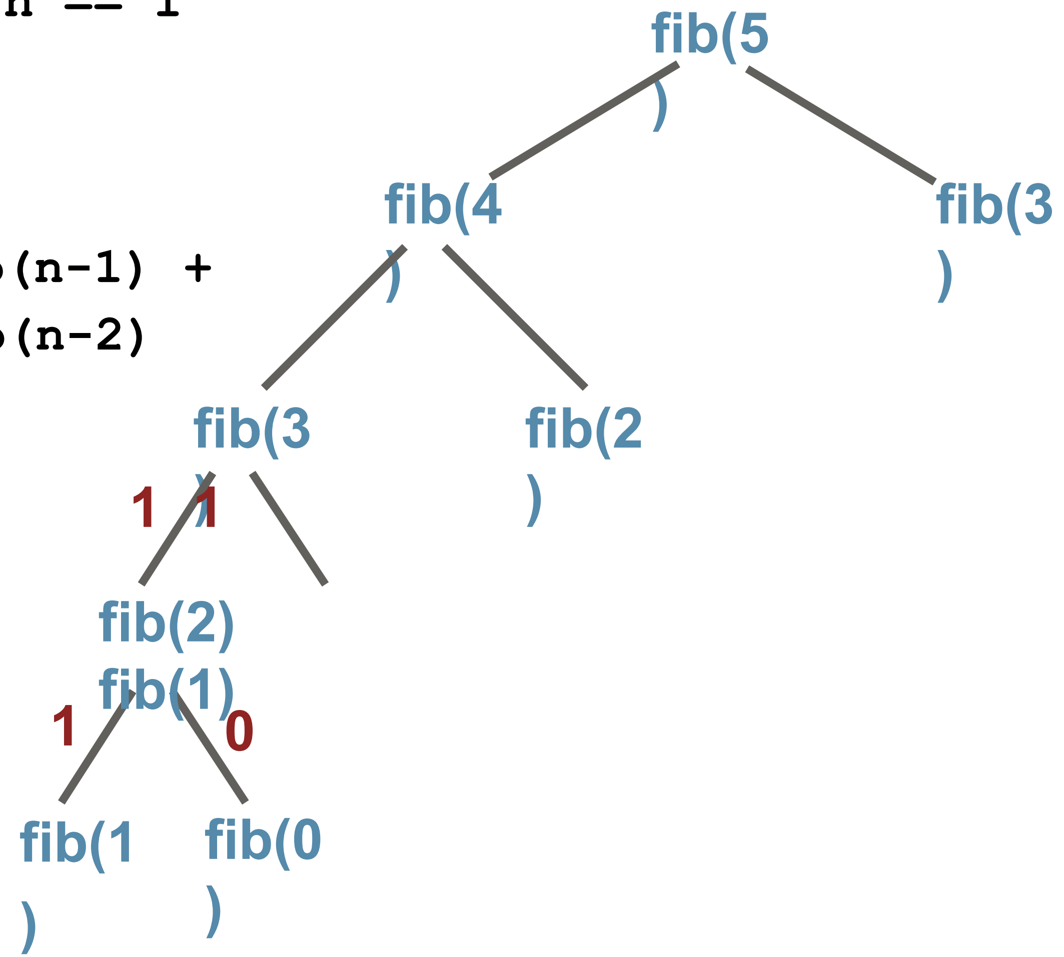
Computing fib(5)

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



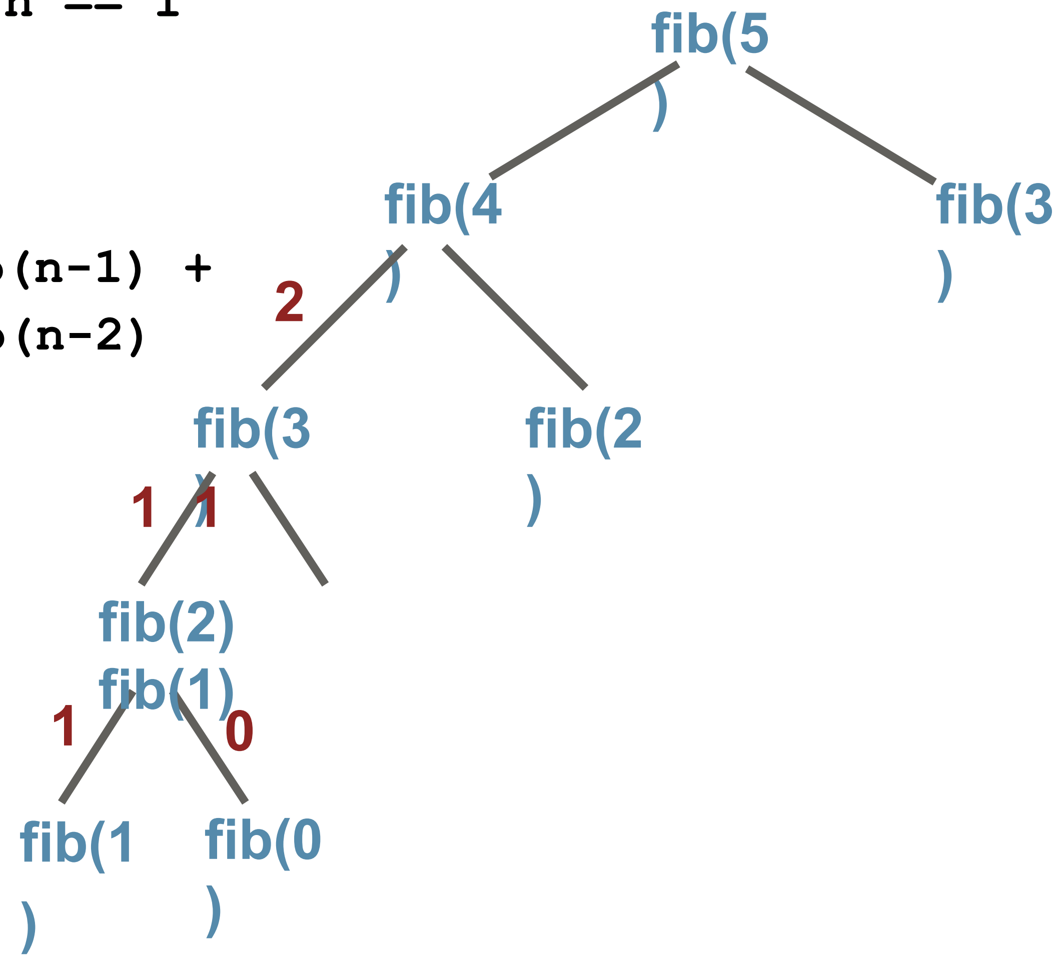
Computing fib(5)

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



Computing fib(5)

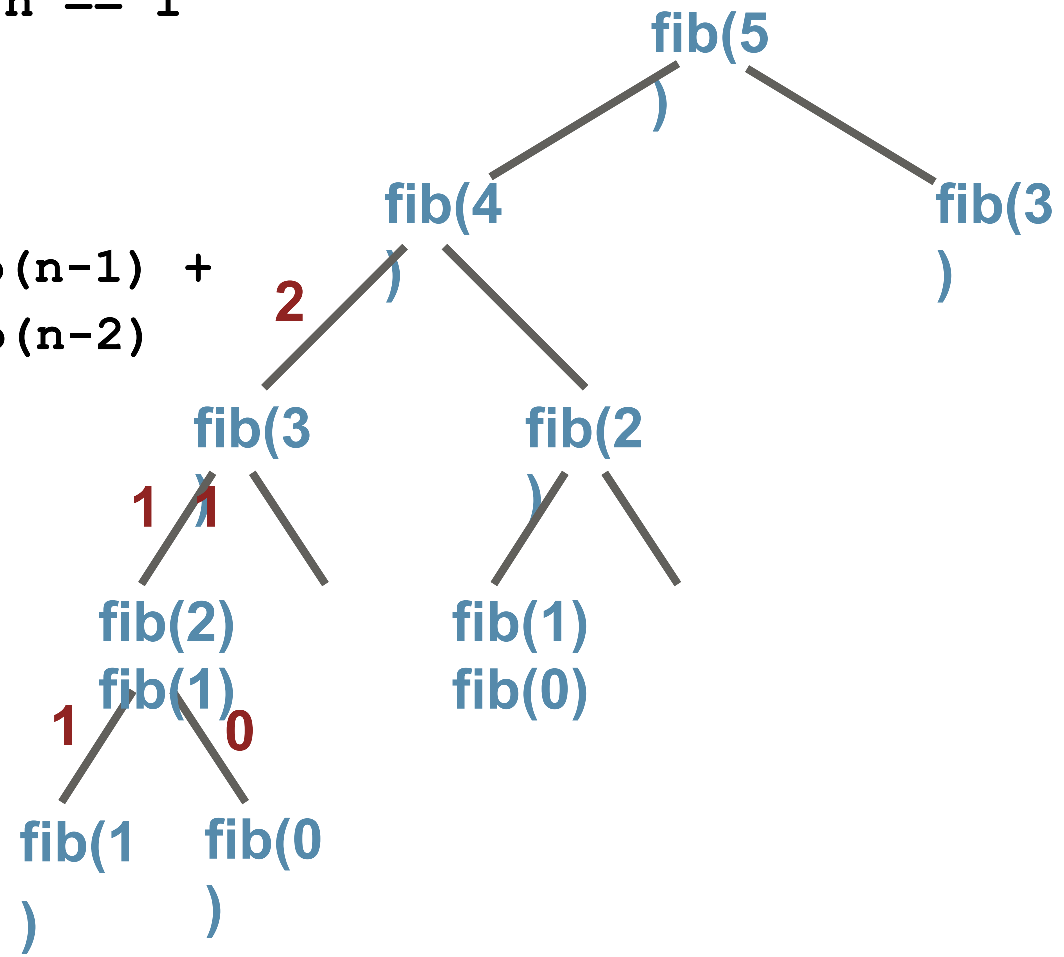
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

Computing fib(5)

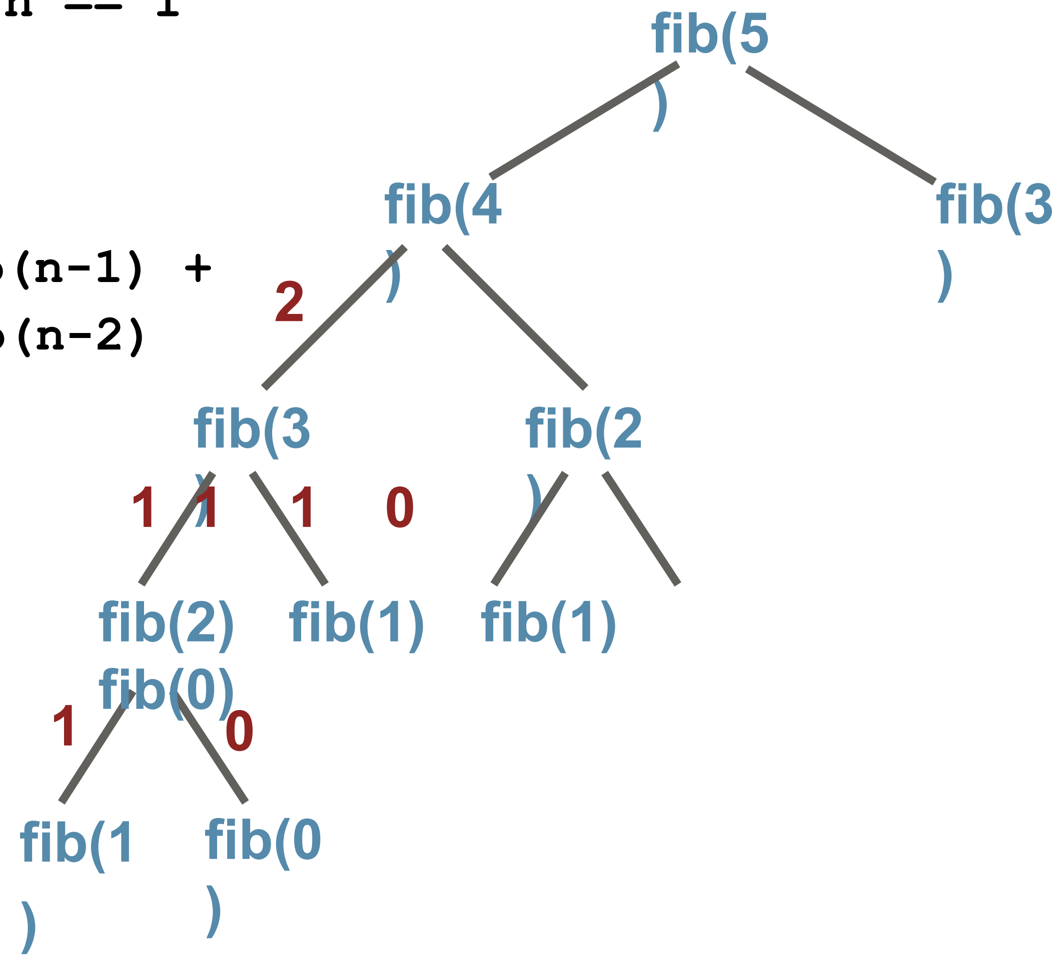
```
function fib(n):
  if n == 0 or n == 1
    value = n

  else
    value = fib(n-1) +
            fib(n-2)

  return(value)
```



Prepared by Pawan R.

Computing fib(5)

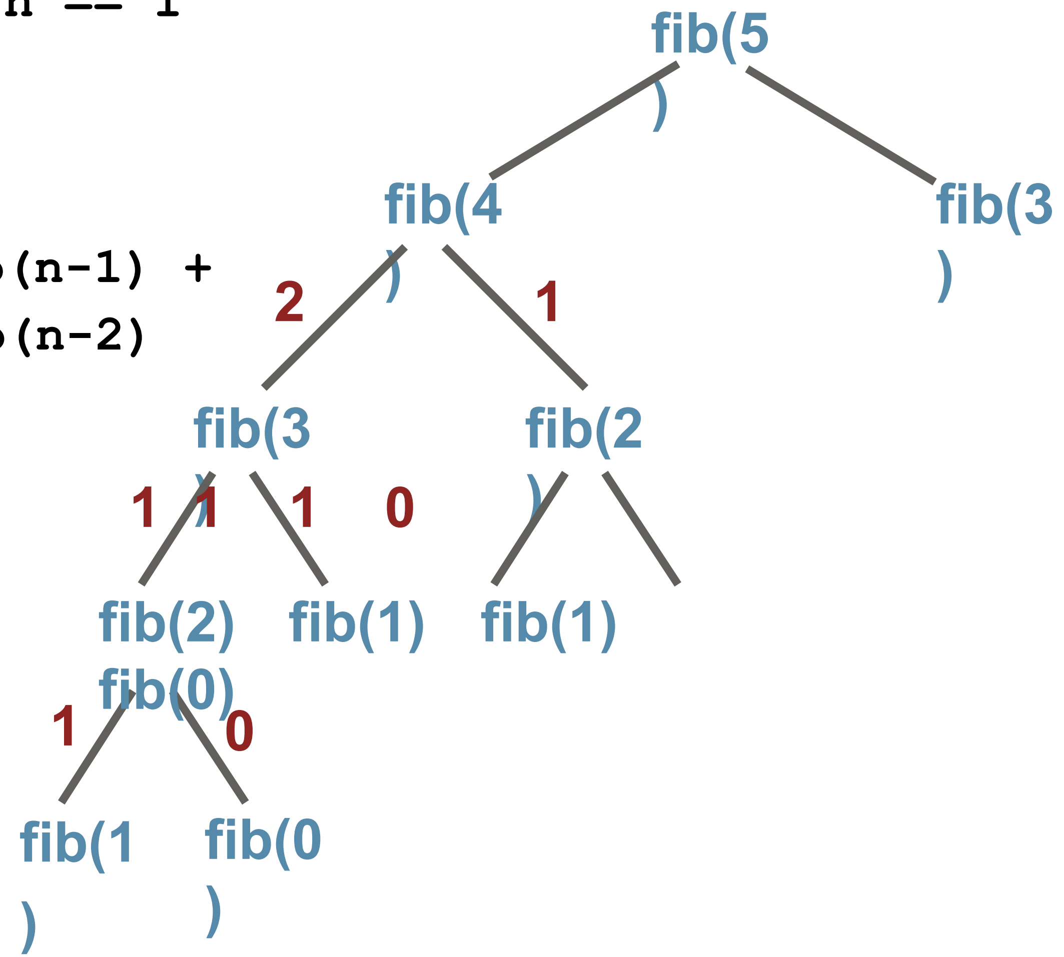
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

Computing fib(5)

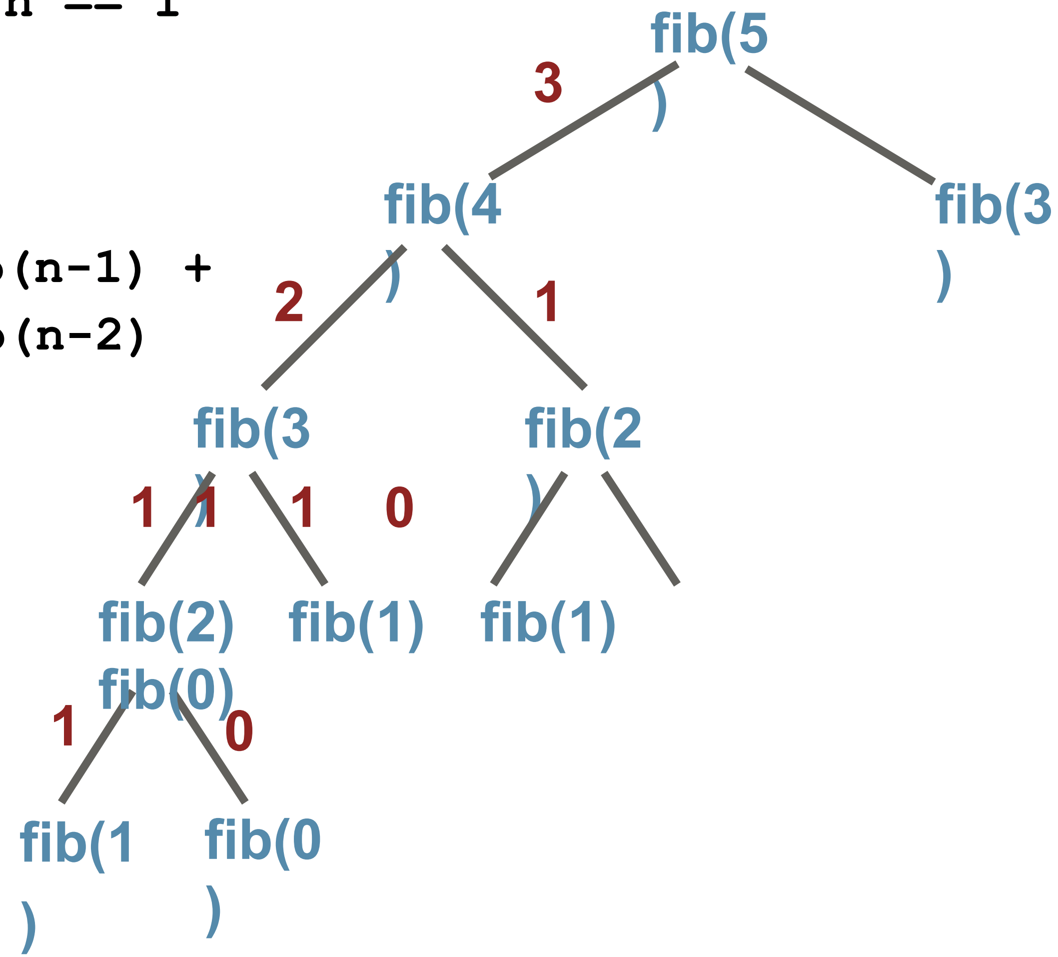
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

Computing fib(5)

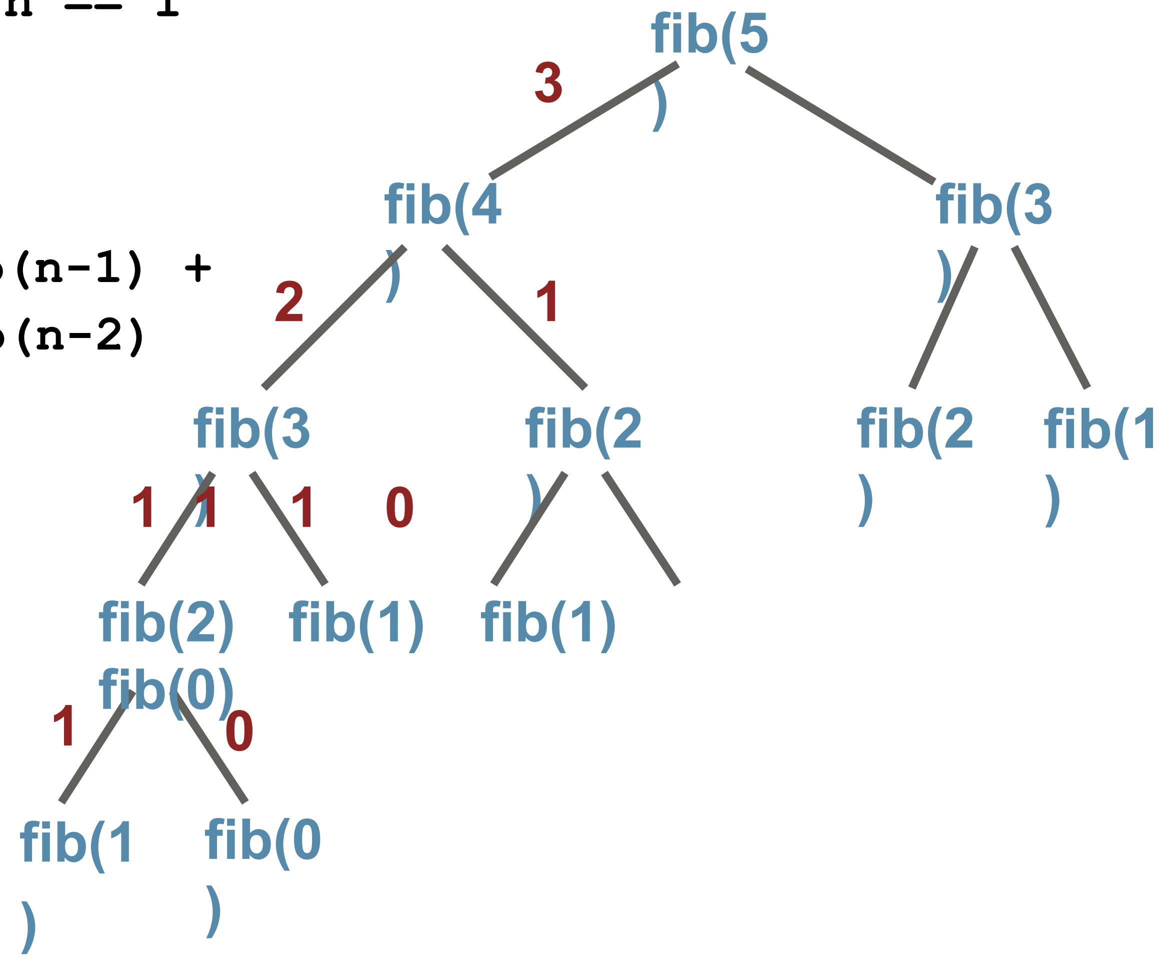
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

Computing fib(5)

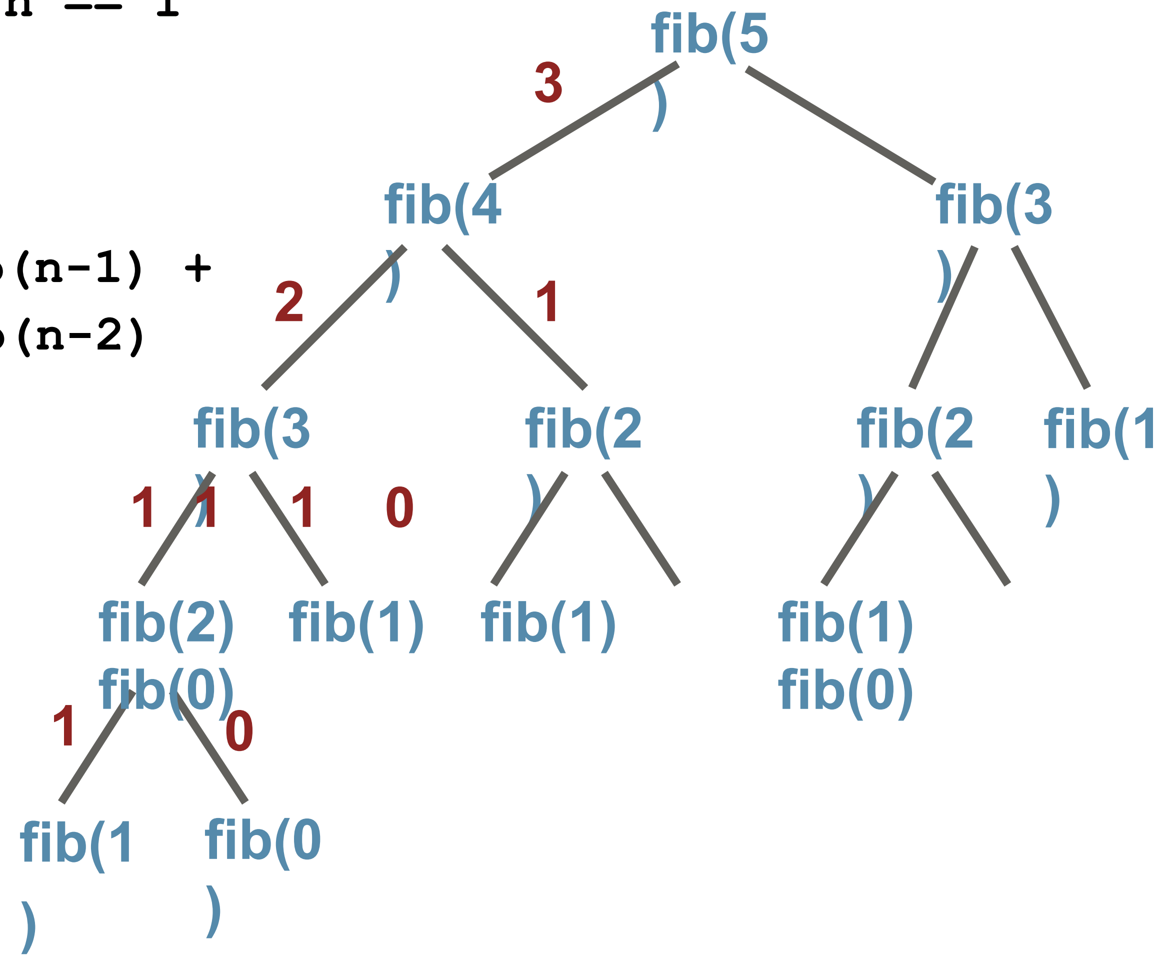
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

Computing fib(5)

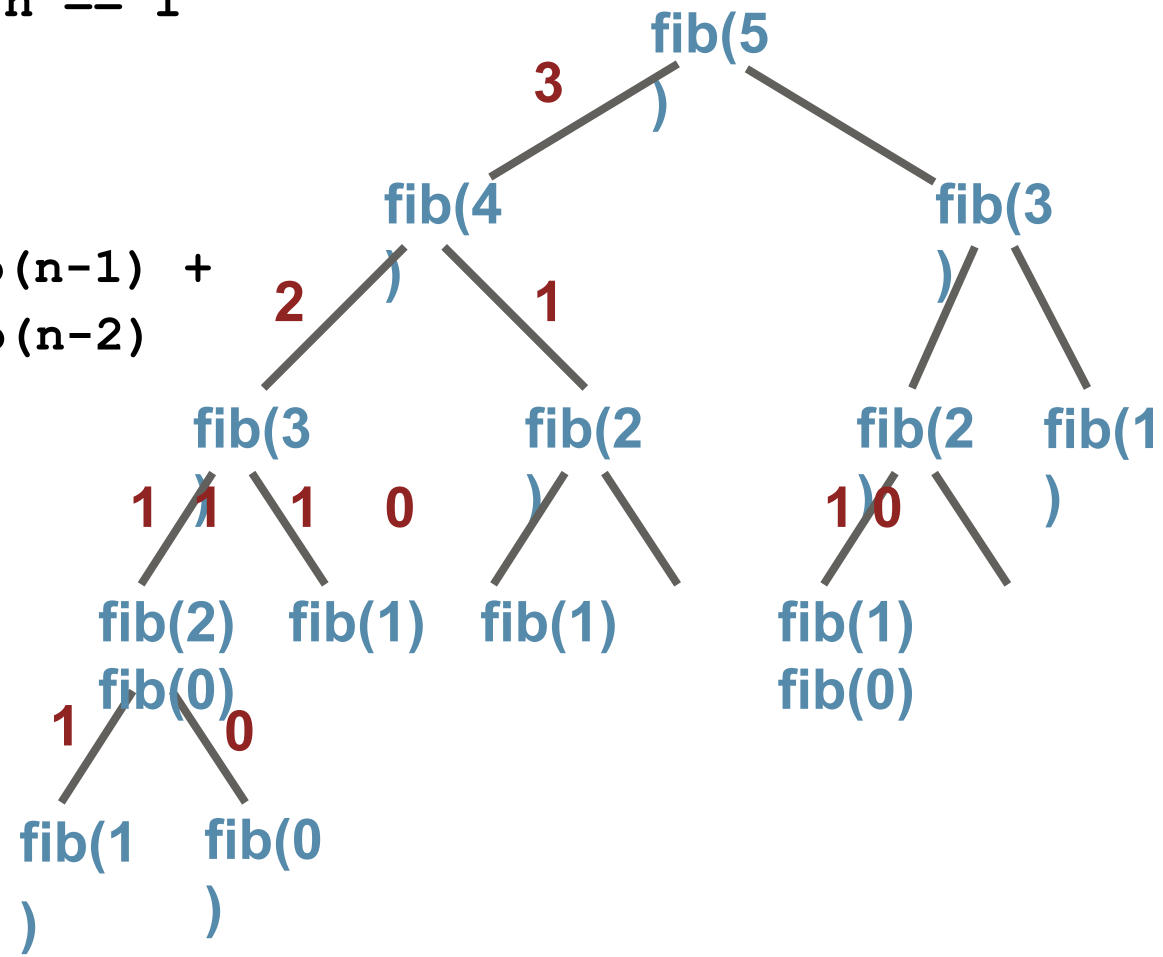
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

Computing fib(5)

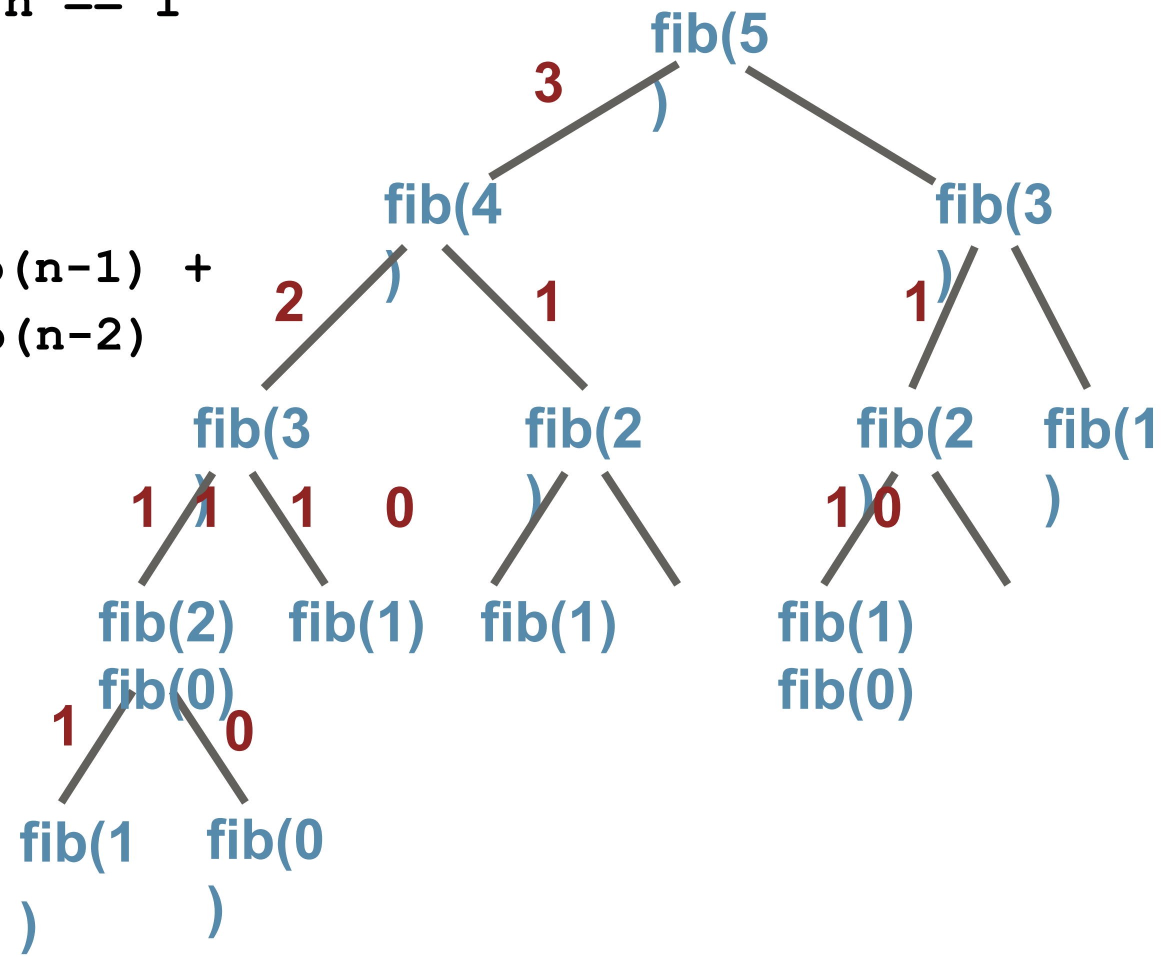
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

Computing fib(5)

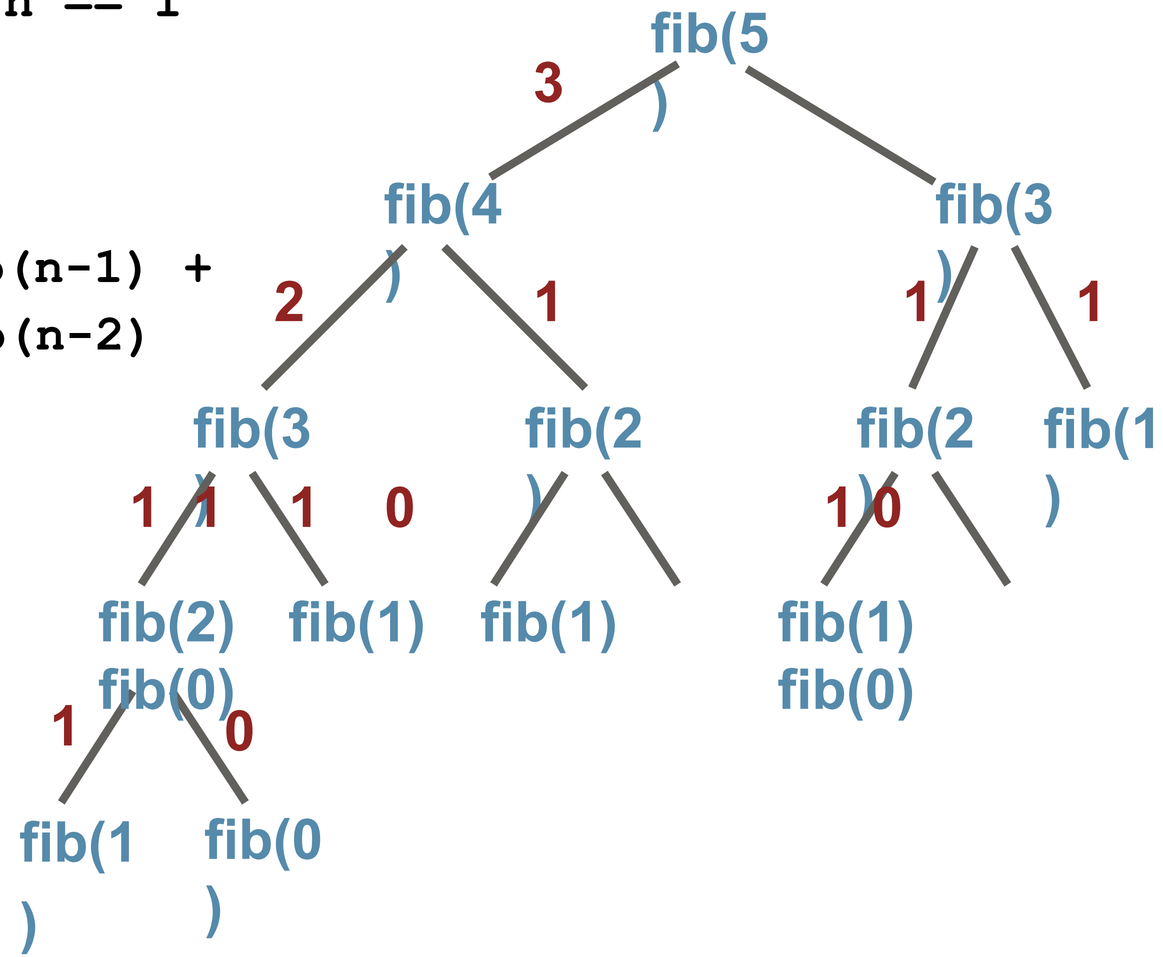
```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



Prepared by *Pawan R.*

Computing fib(5)

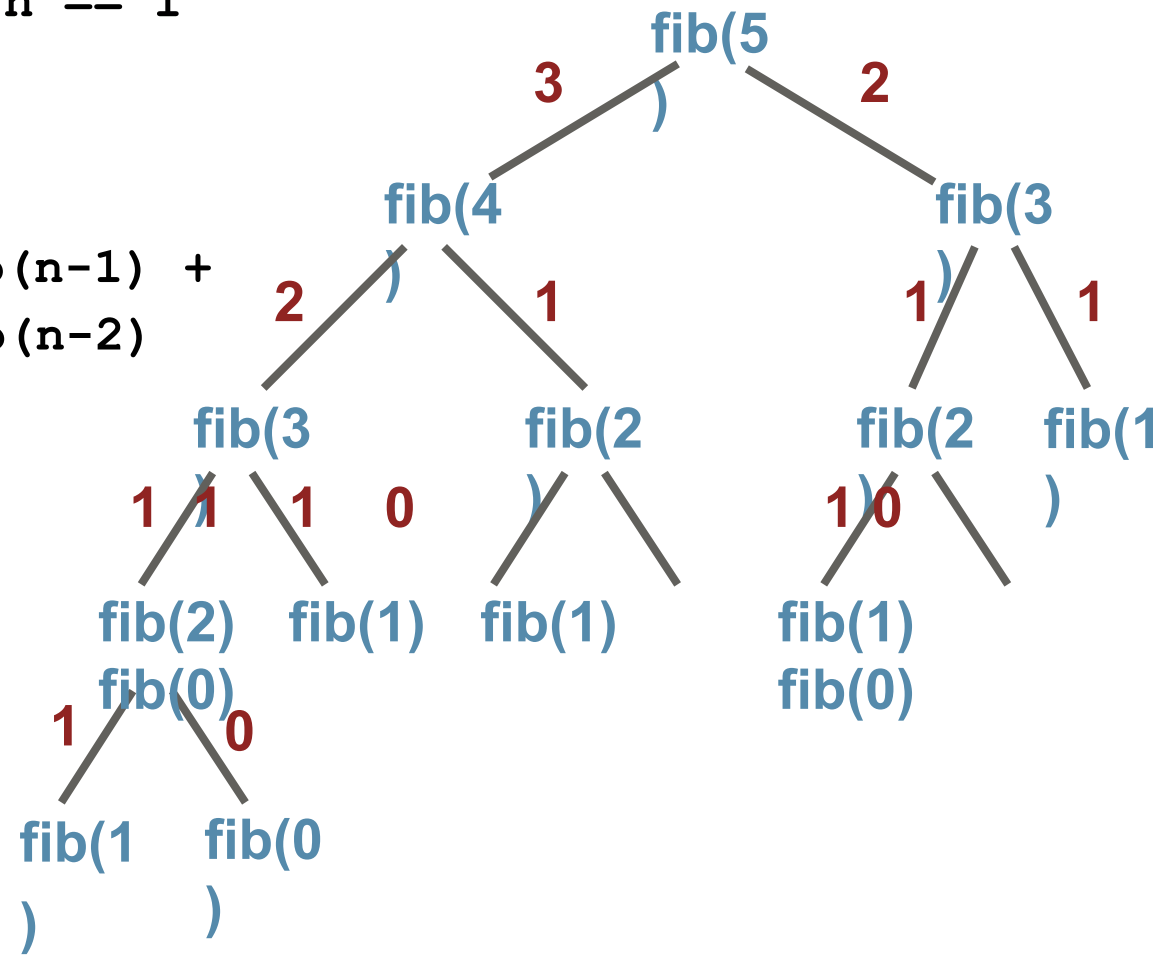
```
function fib(n):
   if n == 0 or n == 1
     value = n
   else
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



Prepared by *Pawan R.*

Computing fib(5)

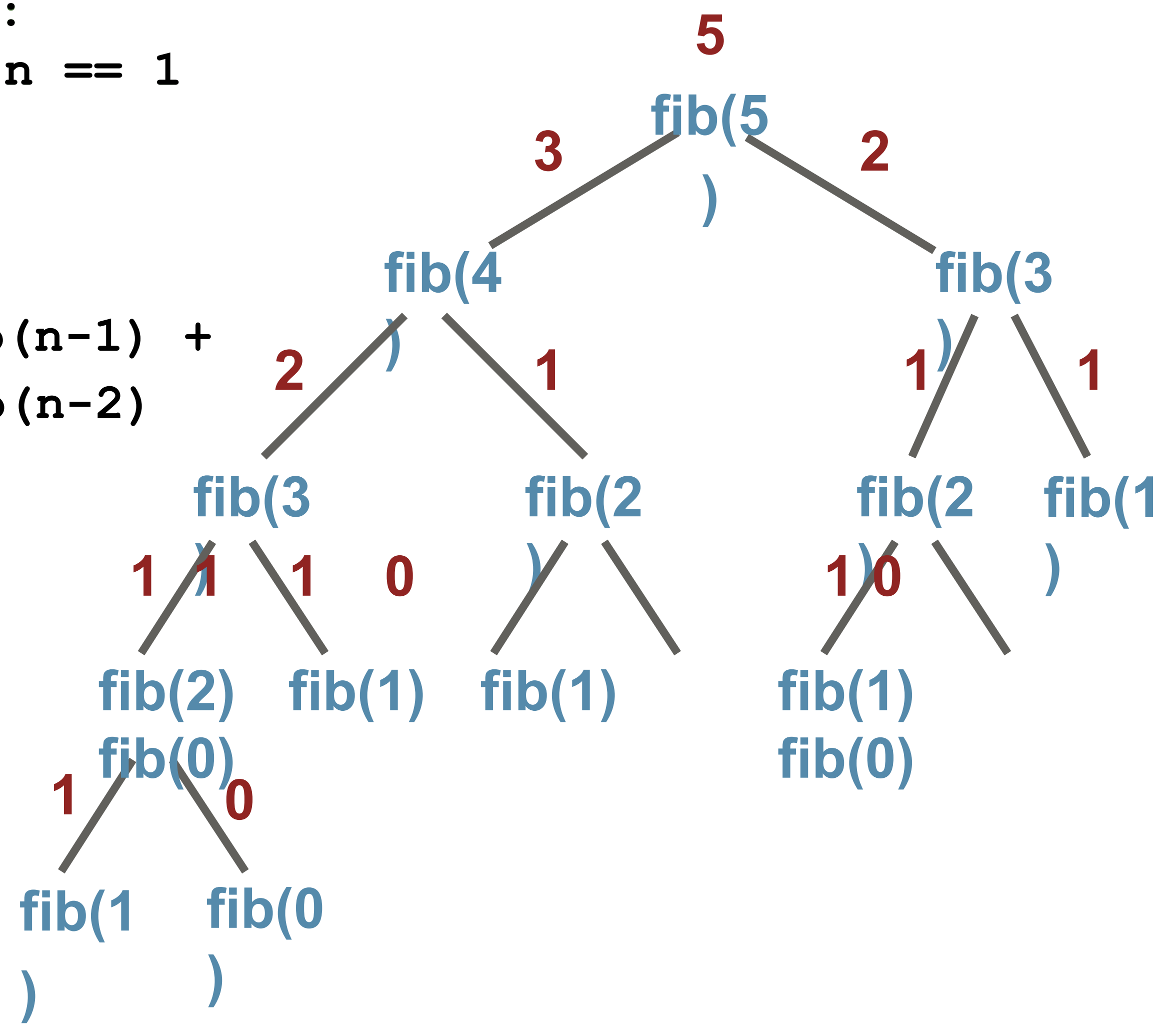
```
function fib(n):
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Prepared by Pawan R.

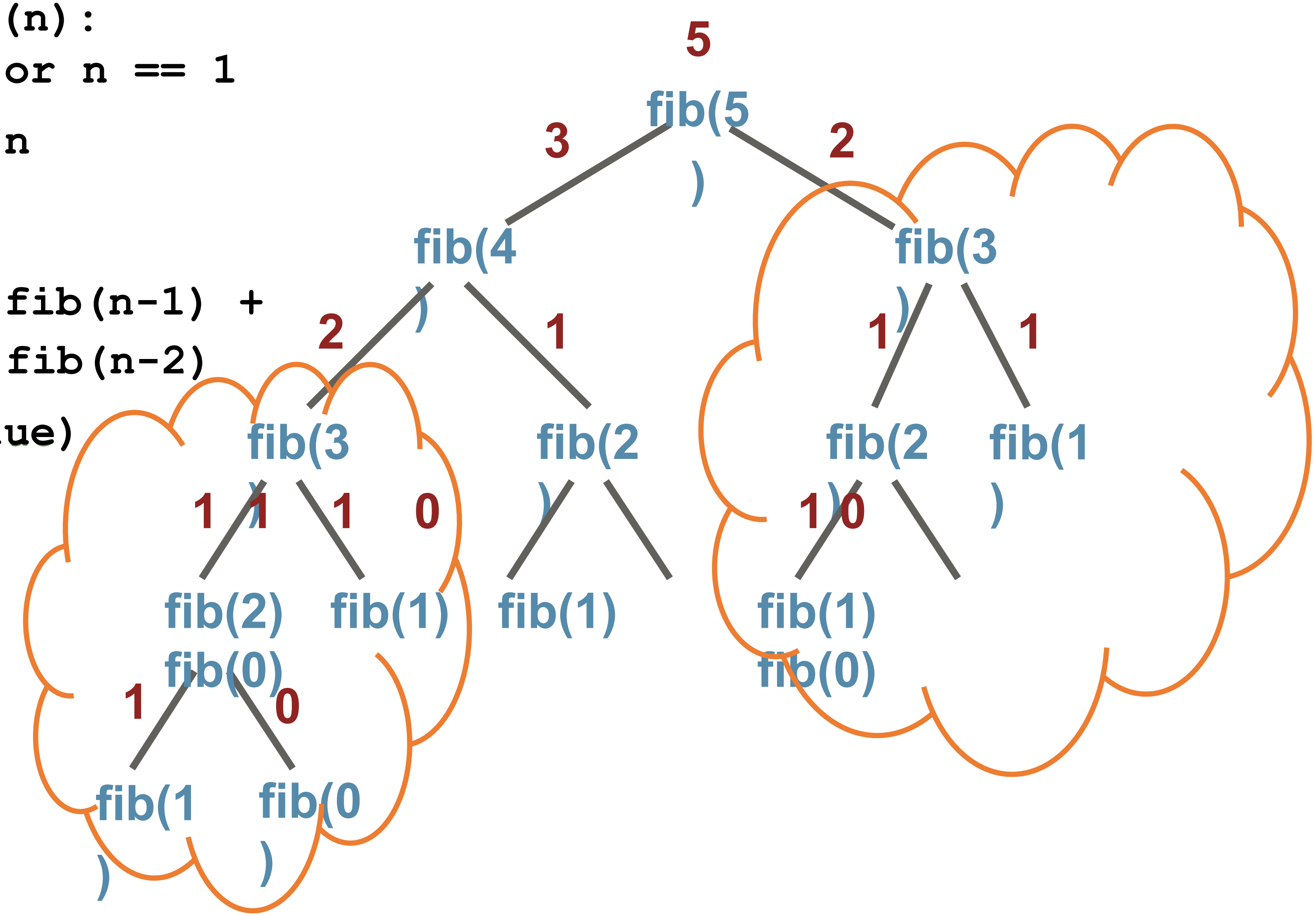
Computing fib(5)

```
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)
```

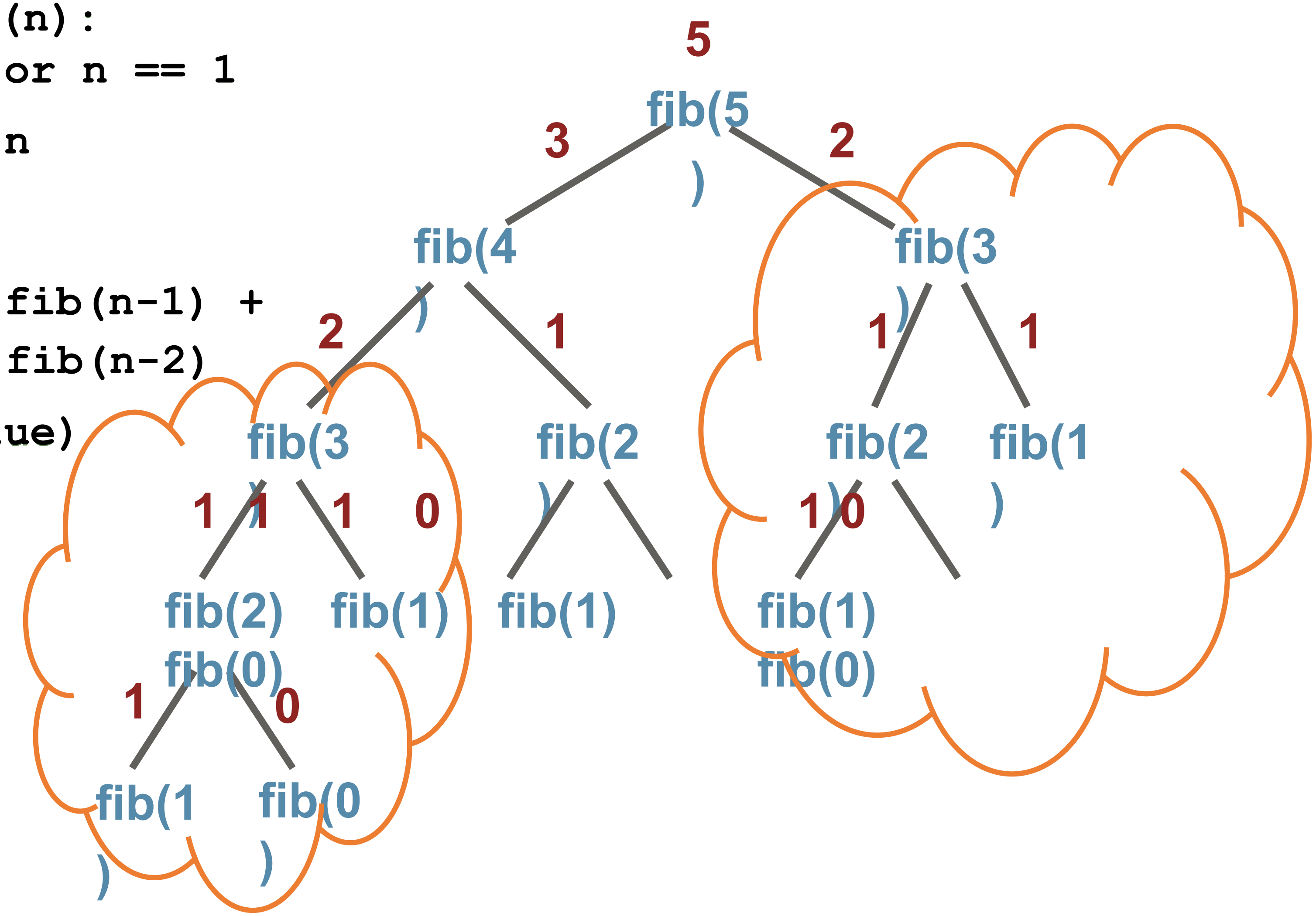


Prepared by Pawan R.

Computing fib(5)

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

Time complexity ??

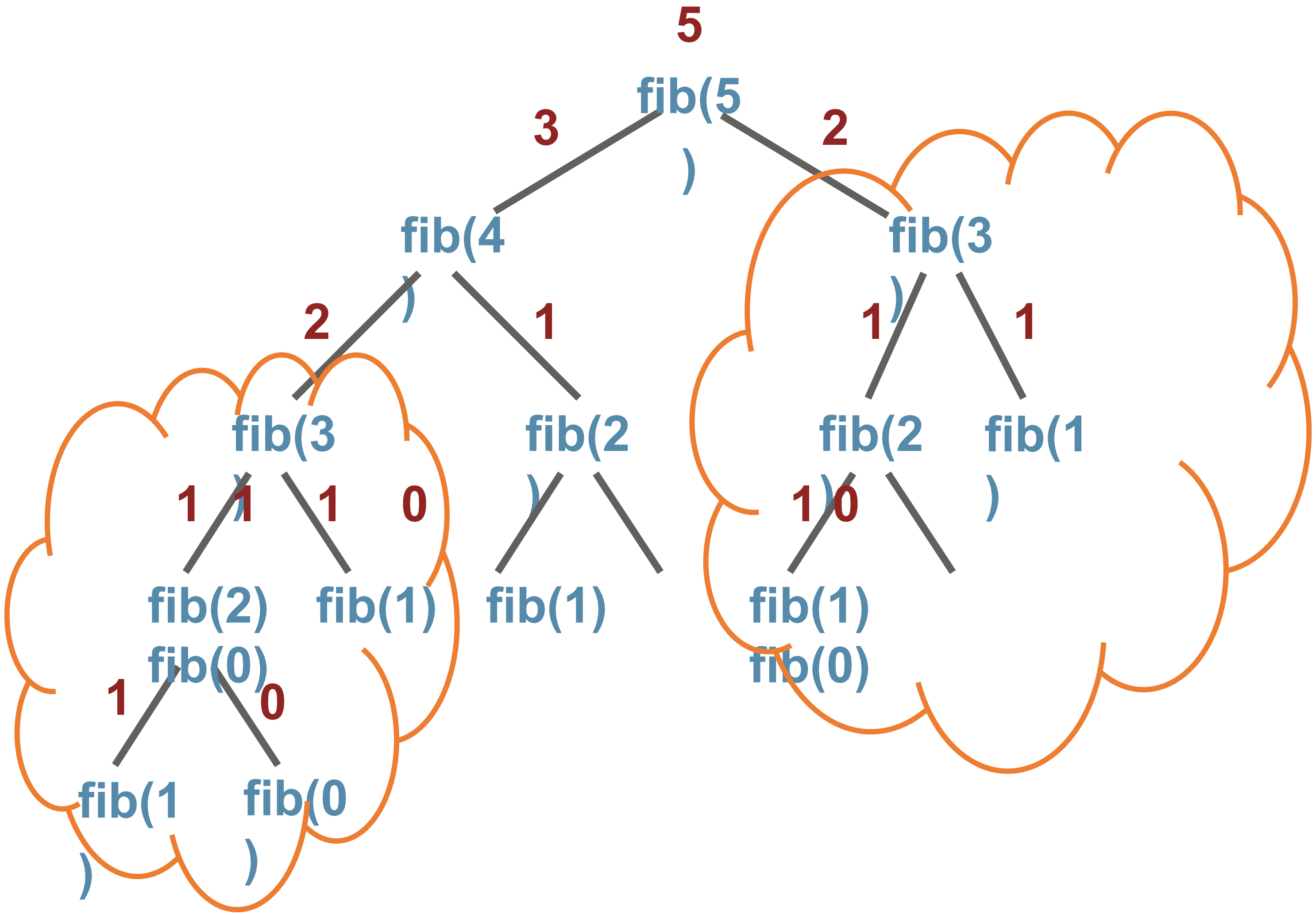


OBSERVATIONS

Overlapping subproblems

Wasteful recomputation

Computation tree grows exponentially



Prepared by Pawan R.

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)

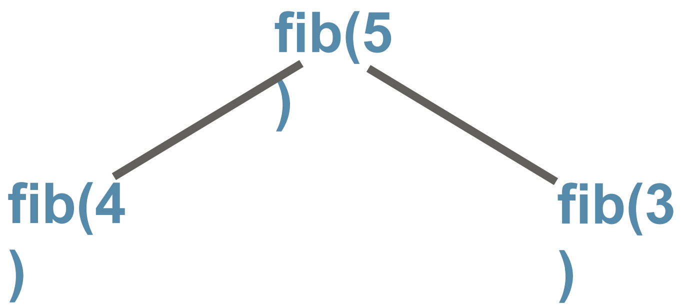
Memoized fib(5)

Memoization

-Store each newly computed value in a table

-Look up table before starting a recursive computation

-Computation tree is linear



K	Fib(k)

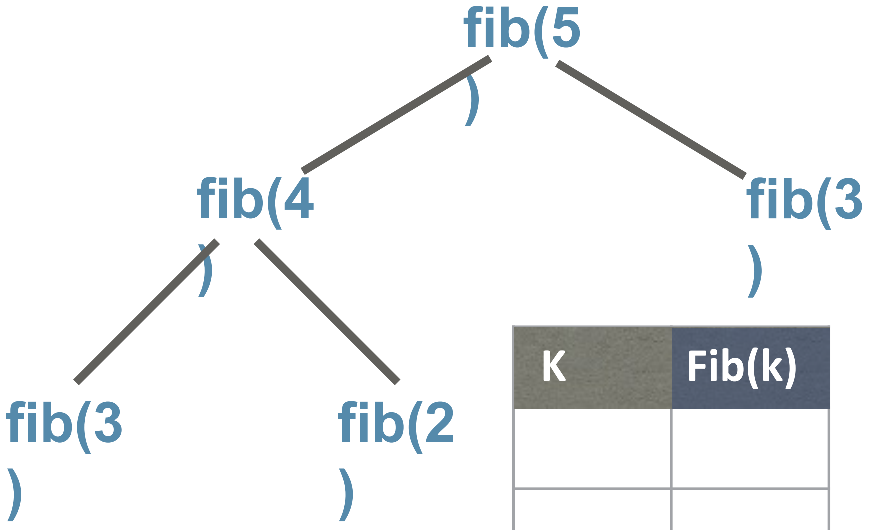
Memoized fib(5)

Memoization

-Store each newly computed value in a table

-Look up table before starting a recursive computation

-Computation tree is linear



K	Fib(k)

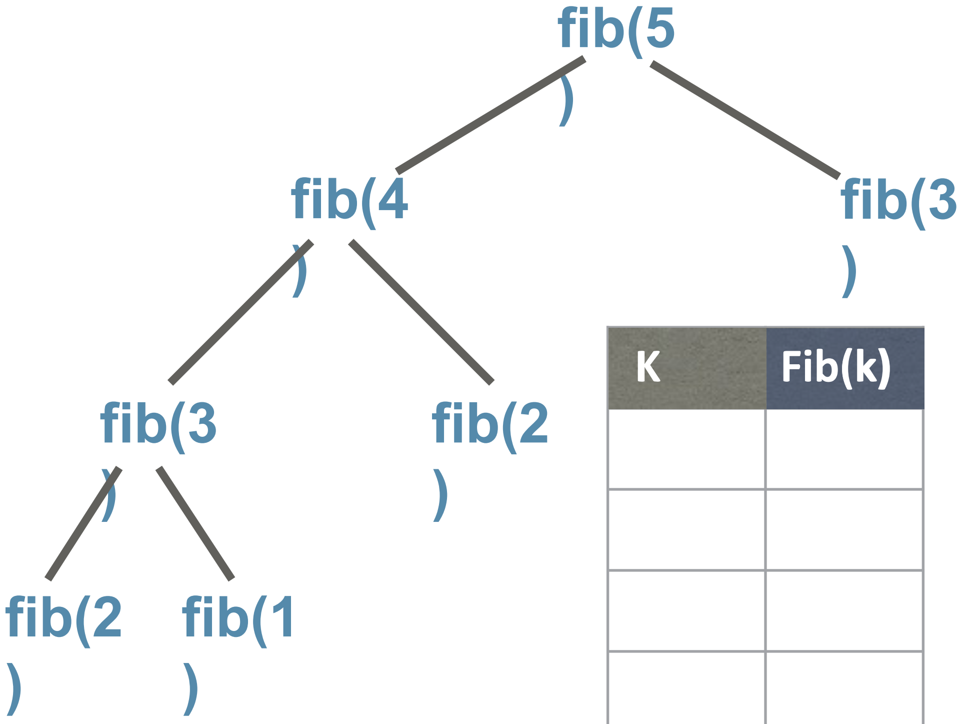
Memoized fib(5)

Memoization

-Store each newly computed value in a table

-Look up table before starting a recursive computation

-Computation tree is linear



K	Fib(k)

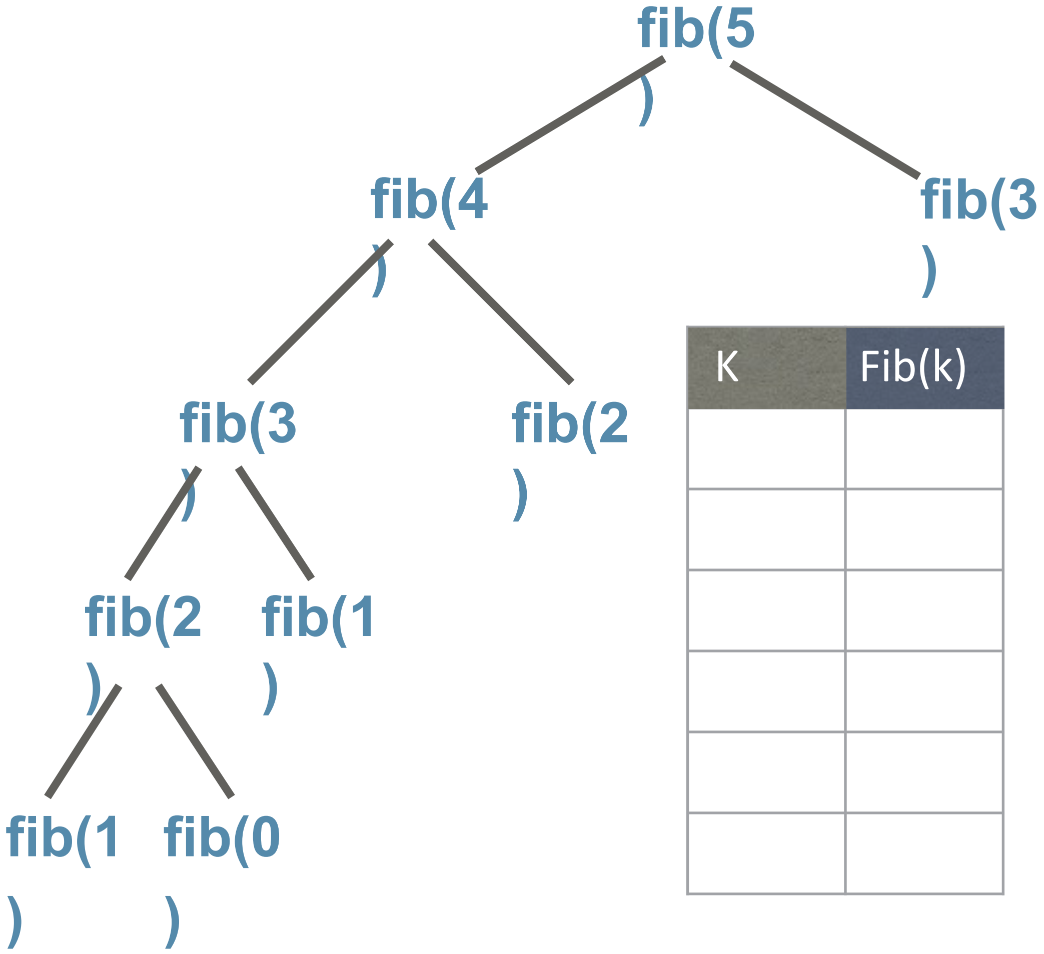
Memoized fib(5)

Memoization

-Store each newly computed value in a table

-Look up table before starting a recursive computation

-Computation tree is linear



K	Fib(k)

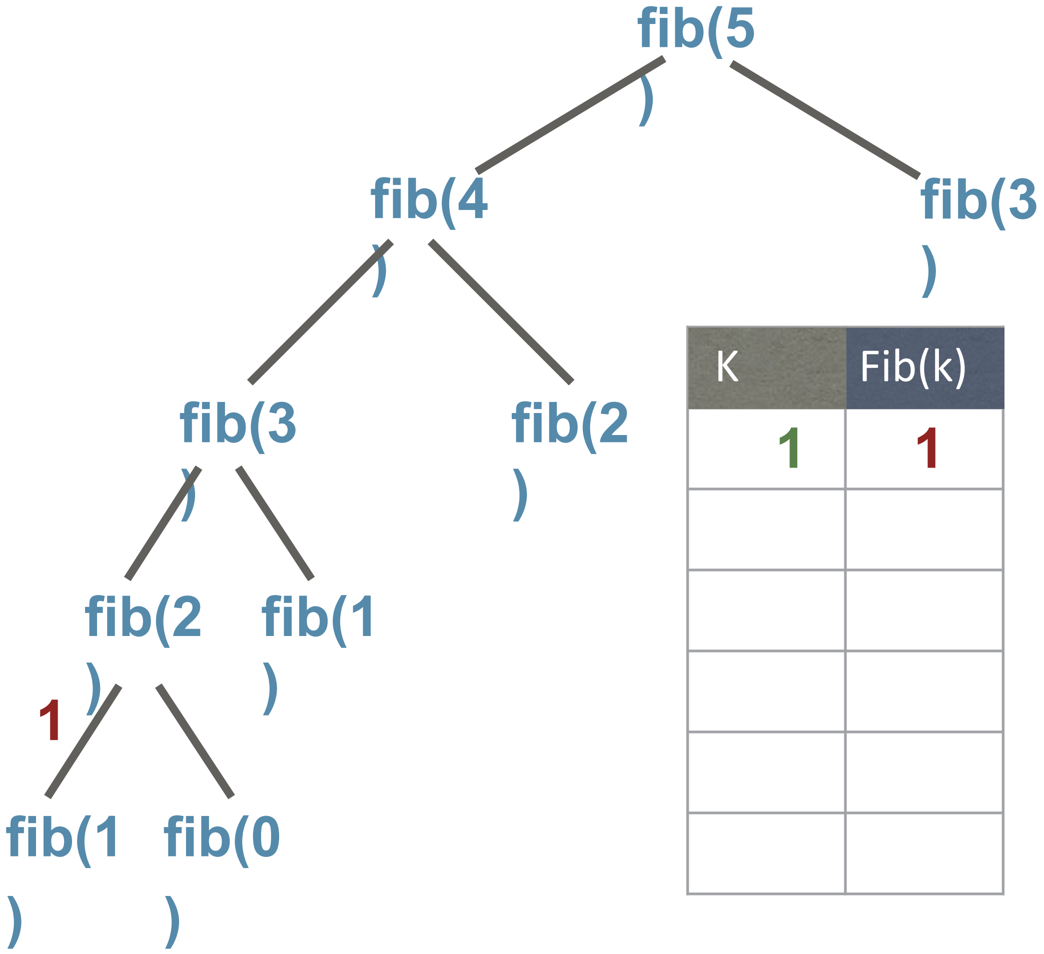
Memoized fib(5)

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

Prepared by Pawan R.

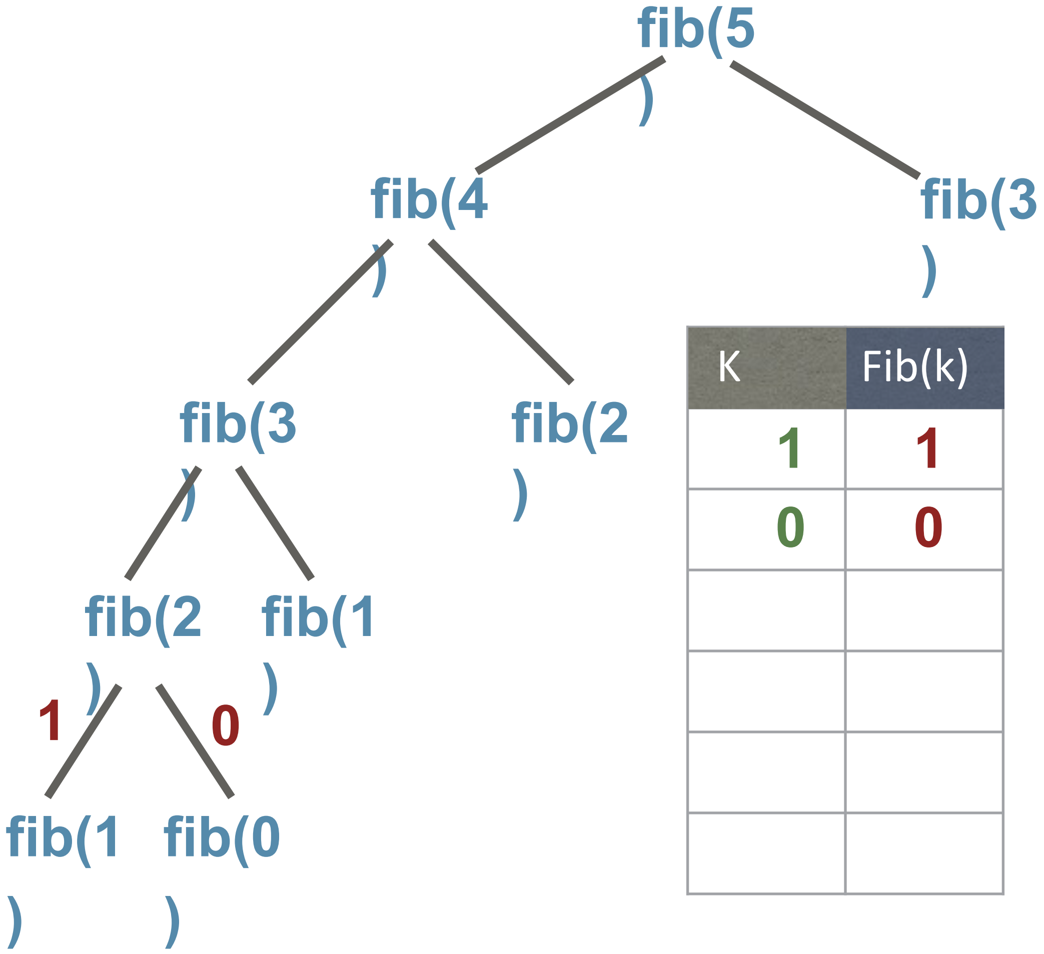
Memoized fib(5)

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

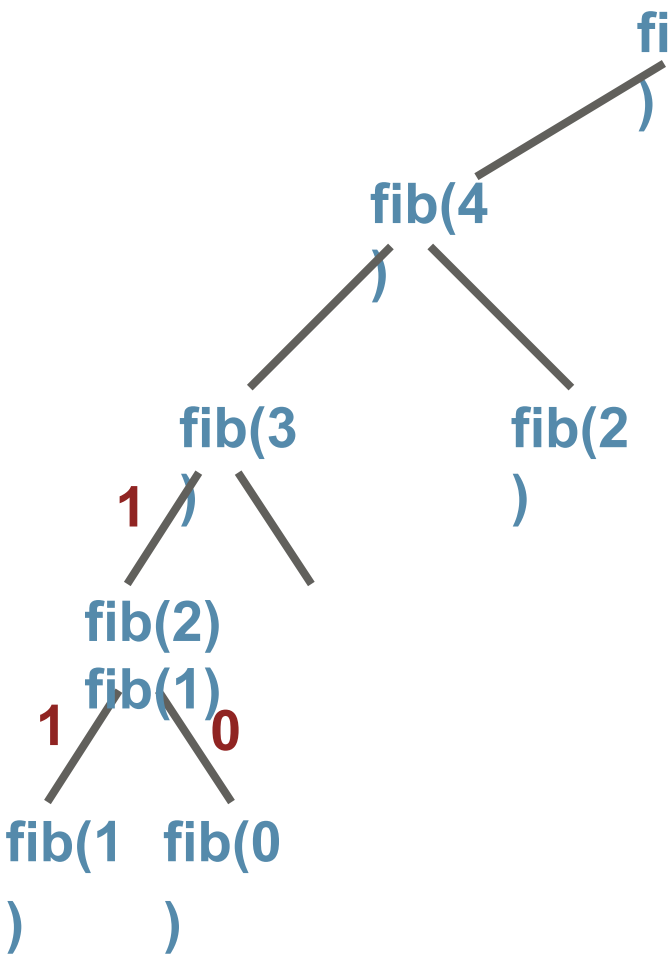
Memoized fib(5)

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

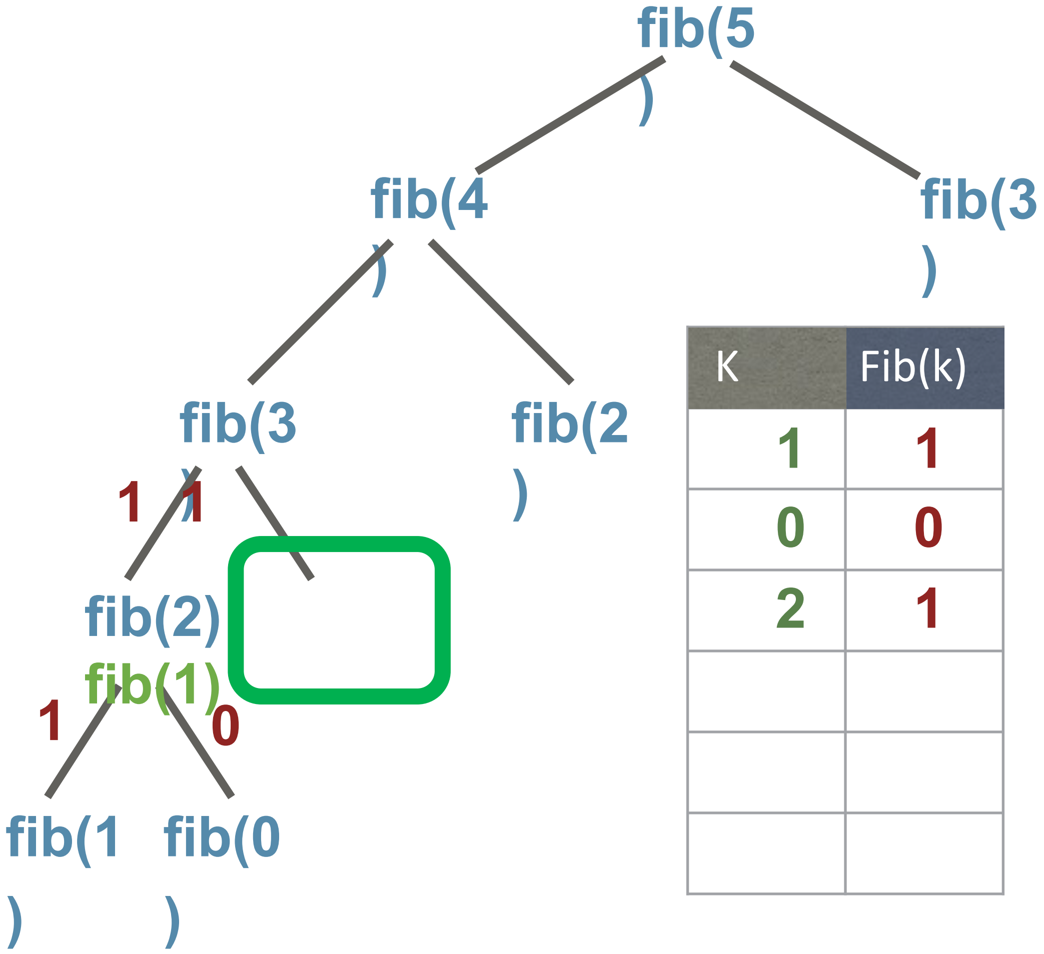
Memoized fib(5)

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

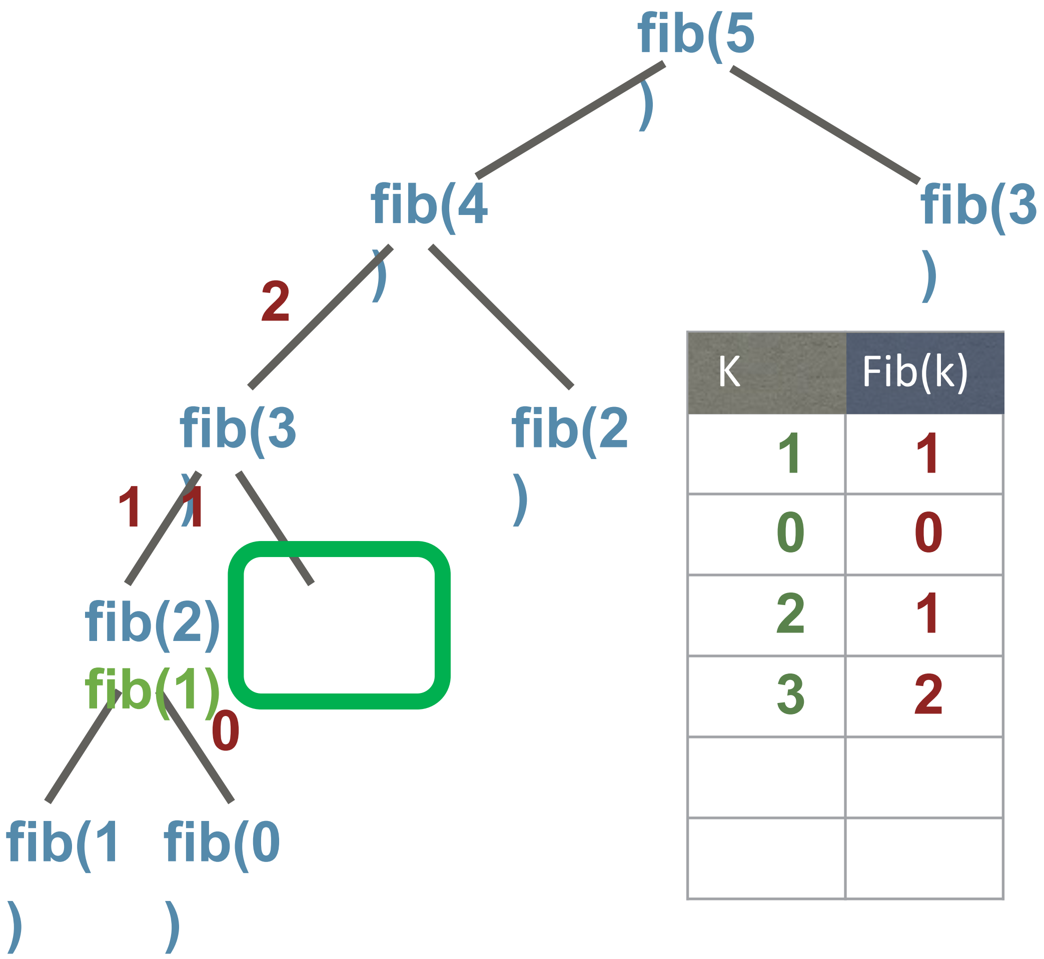
Memoized fib(5)

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

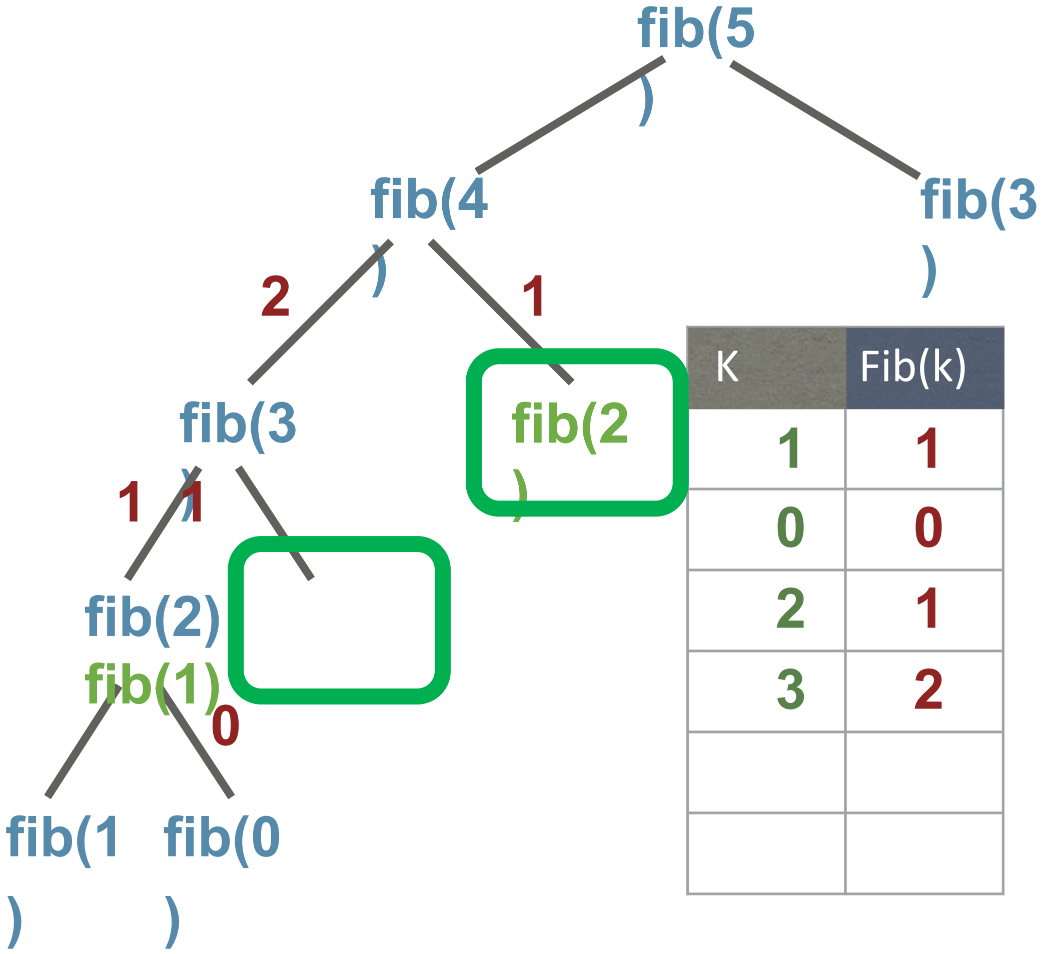
Memoized fib(5)

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

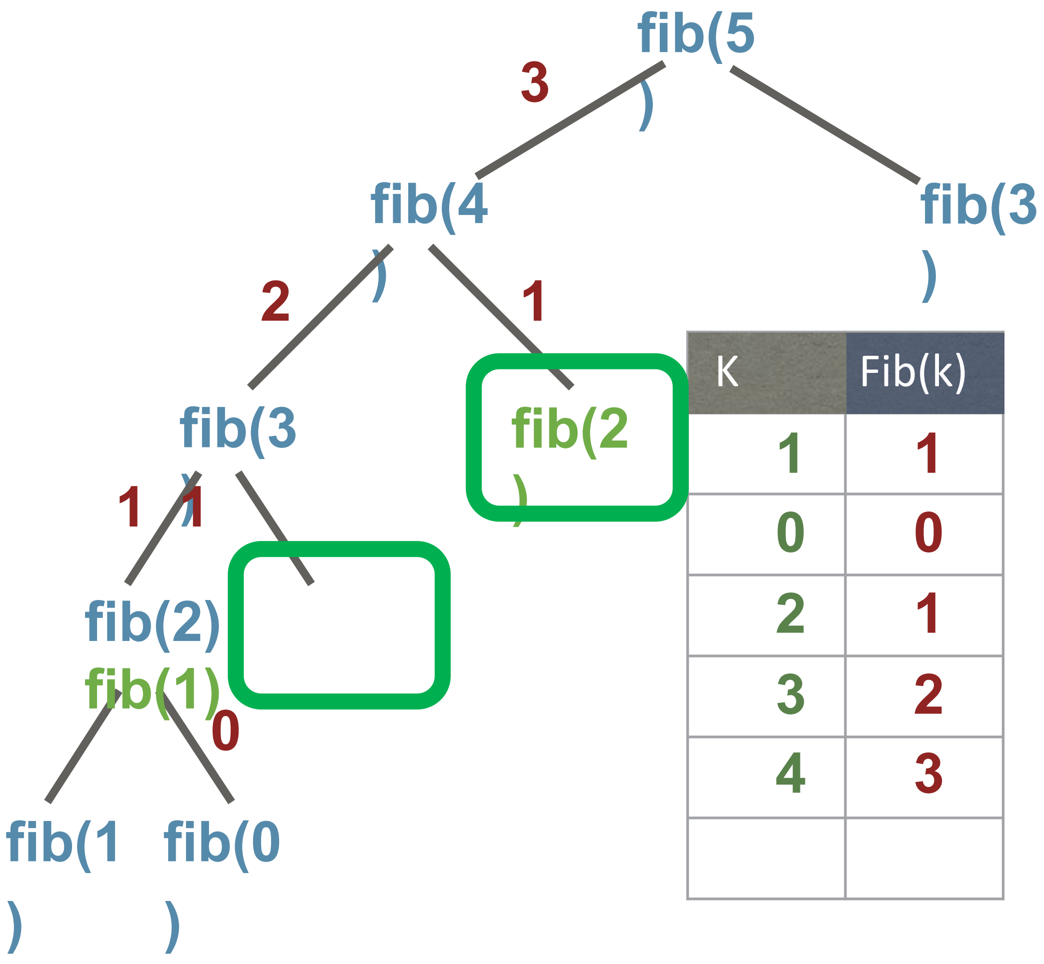
Memoized fib(5)

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

Prepared by Pawan R.

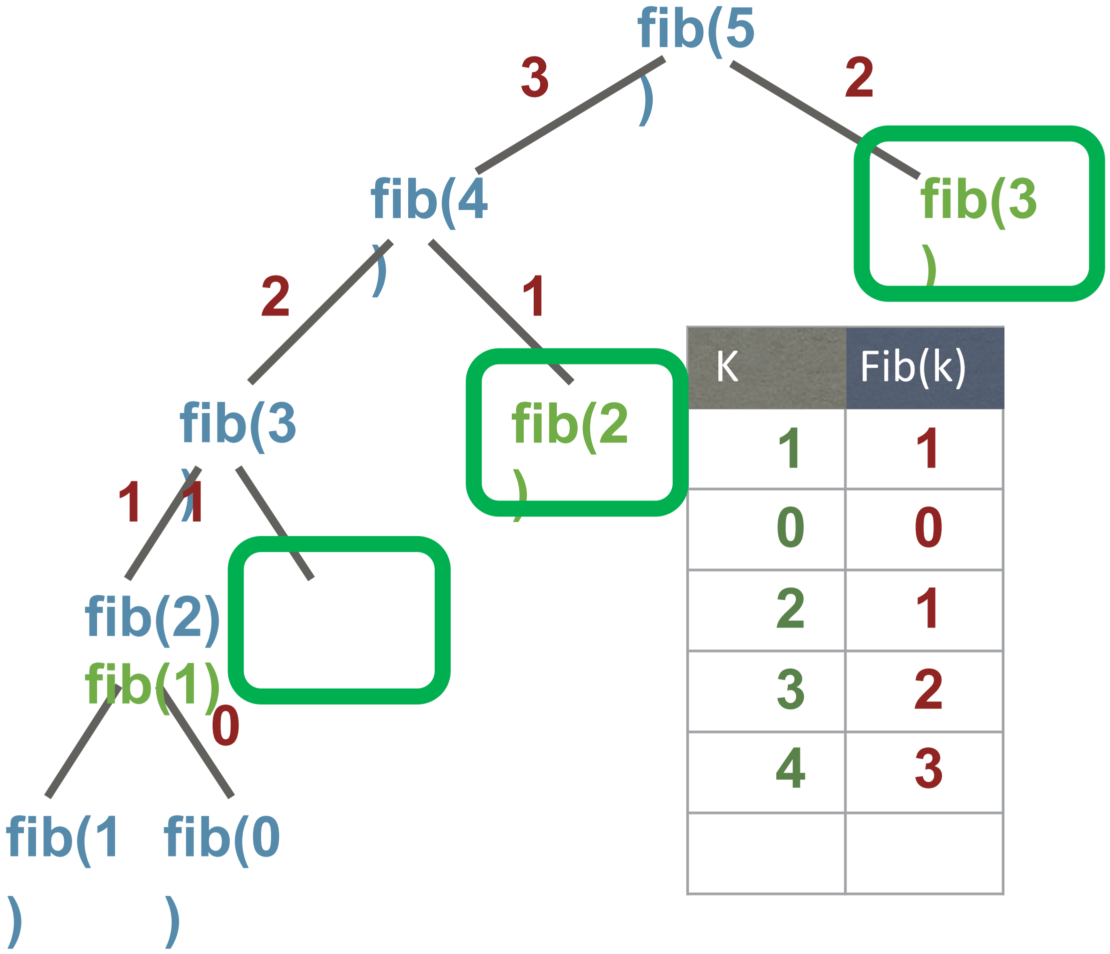
Memoized fib(5)

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

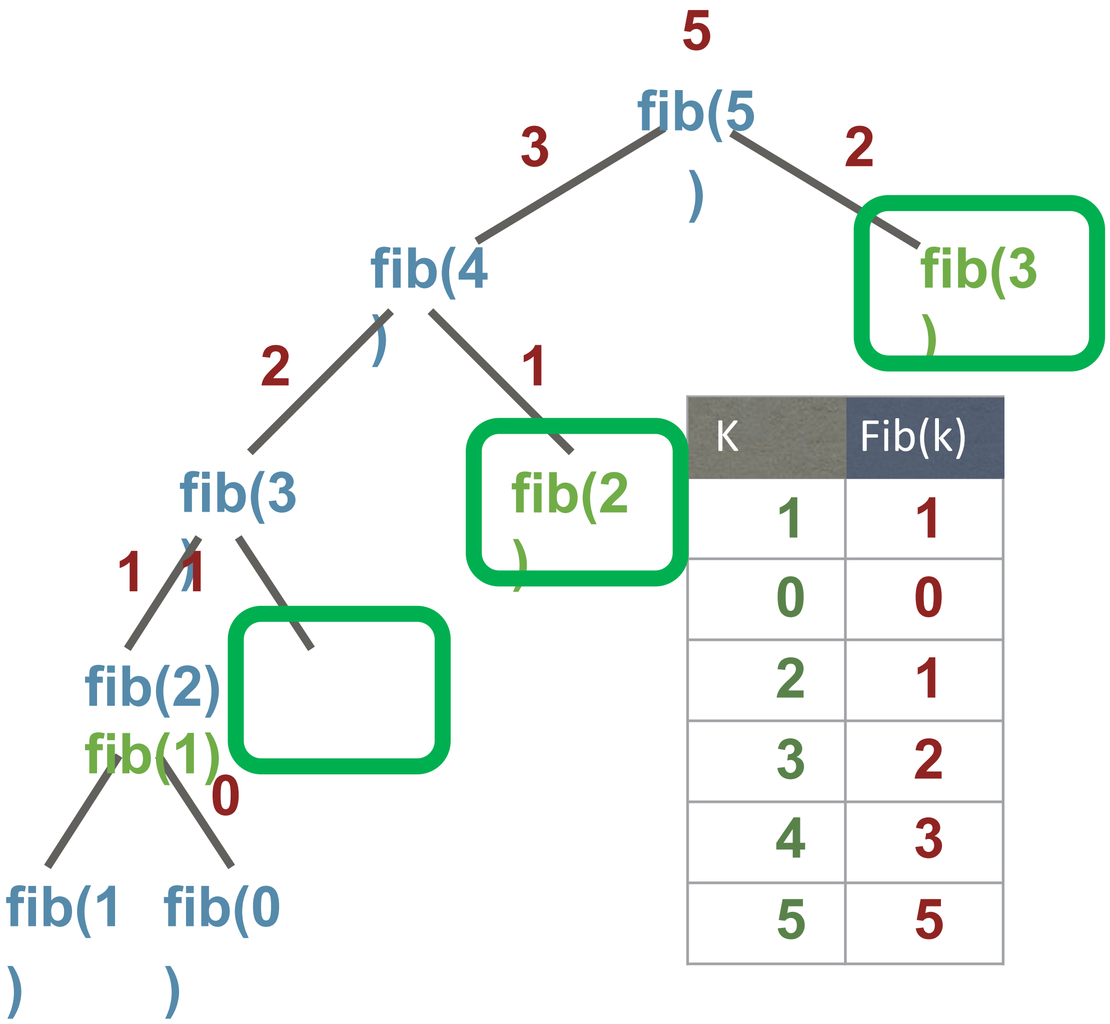
Memoized fib(5)

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)
```

Extra

Extra

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)
```

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)
```

Need to solve recursively exploiting the inductive definition.

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

fib(5)

fib(4)

fib(3)

fib(2)

fib(1)

fib(0)

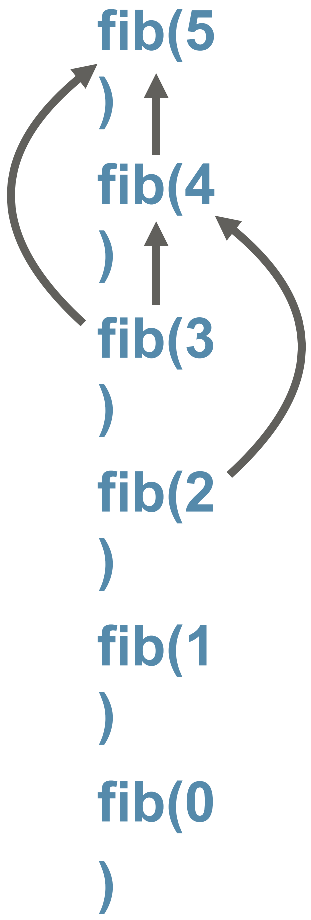
Dynamic programming

Anticipate what the memory table looks like

Subproblems are known from problem structure

Dependencies form a dag

Solve subproblems in topological order



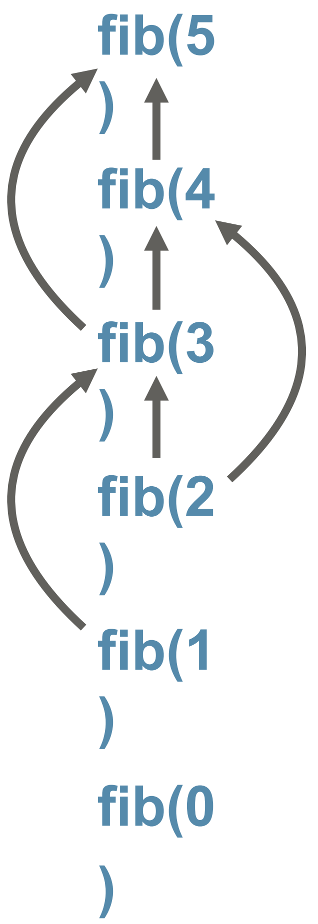
Dynamic programming

Anticipate what the memory table looks like

Subproblems are known from problem structure

Dependencies form a dag

Solve subproblems in topological order

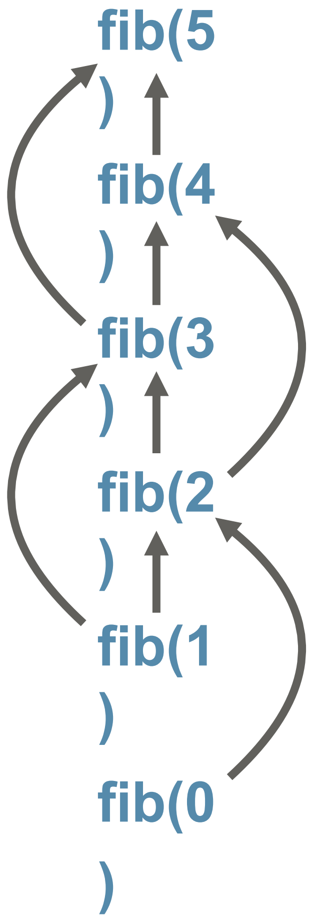


Dynamic programming

Anticipate what the memory table looks like

- Subproblems are known from problem structure
- Dependencies form a dag

Solve subproblems in topological order



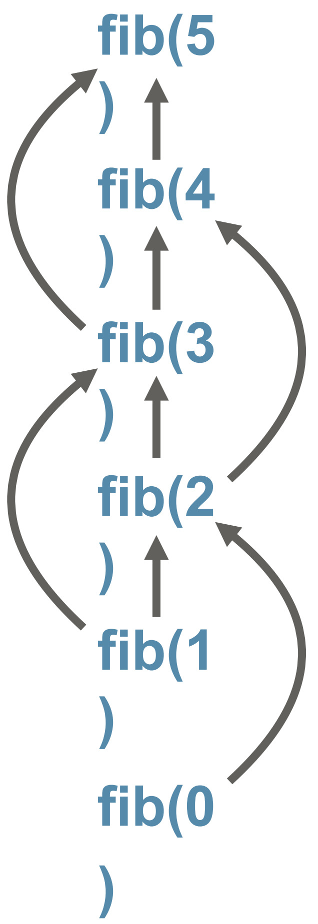
Dynamic programming

Anticipate what the memory table looks like

- Subproblems are known from problem structure
- Dependencies form a dag

Solve subproblems in topological order

K	0	1	2	3	4	5
fib(k)						



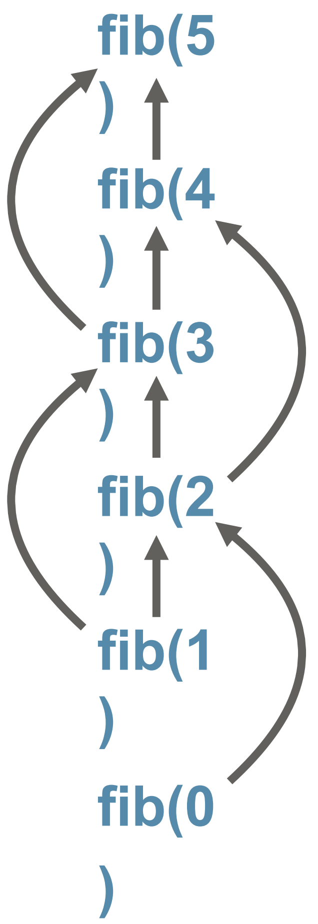
Dynamic programming

Anticipate what the memory table looks like

Subproblems are known from problem structure

Dependencies form a dag

Solve subproblems in topological order



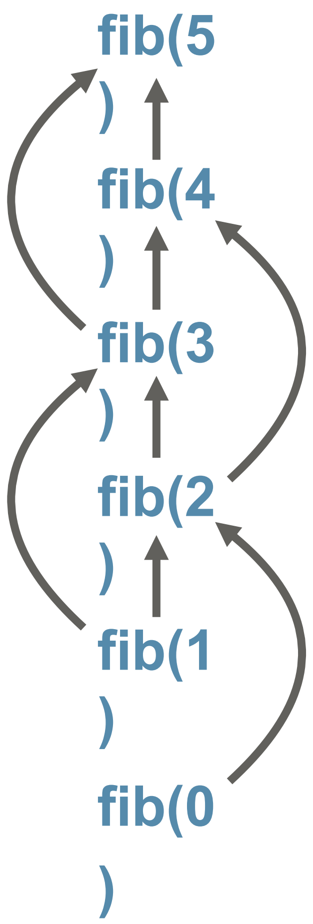
K	0	1	2	3	4	5
fib(k)	0					

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**				

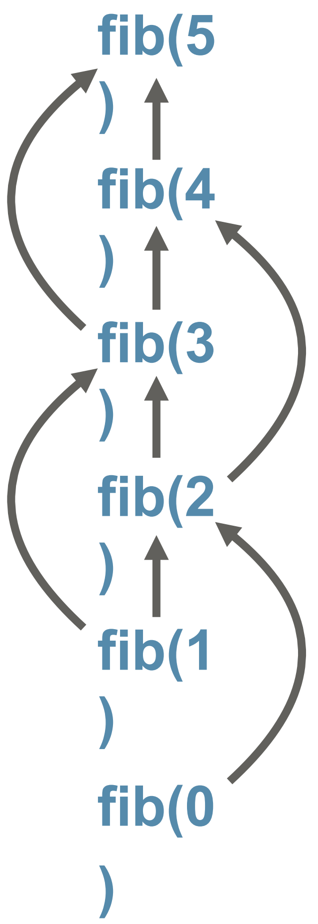
Dynamic programming

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			



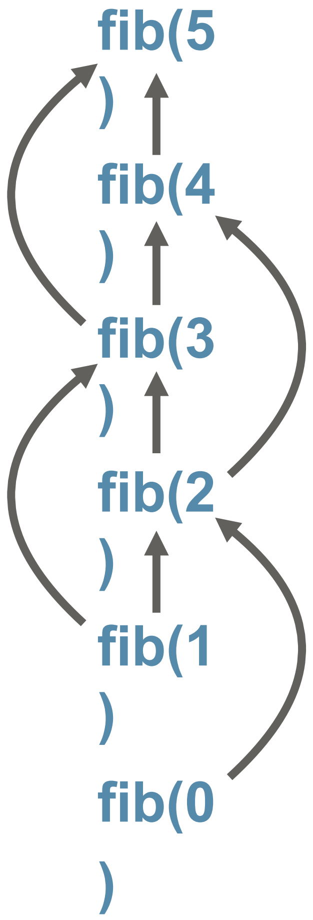
Dynamic programming

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

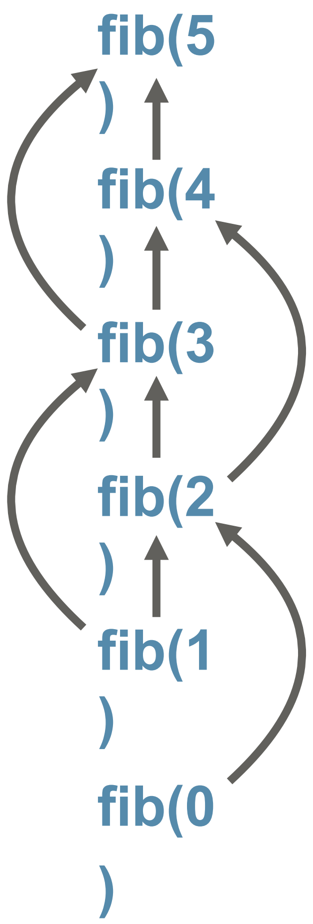
Anticipate what the memory table looks like

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

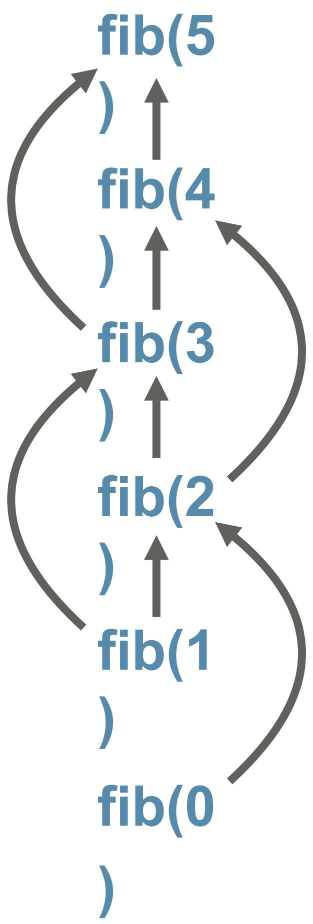
Anticipate what the memory table looks like

- 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])
```

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

Takeaway

Overlapping sub problem

Tabular Method

Recursive formulation– Don’t solve recursively

Compute opt for smaller sub problem

Optimal substructure

Computing Optimal value vs Optimal solution

P1. Knapsack-without profit

Input

n objects $o_1, o_2, \dots, o_n$

Size $s_1, s_2, \dots, s_n$

Bag with capacity W

Find a subset of objects with largest total size $\leq$ W

Prepared by Pawan R.

P1. Knapsack-without profit

Input

n objects $o_1, o_2, \ldots, o_n$

Size $s_1, s_2, \ldots, s_n$

Bag with capacity W

Find a subset of objects with largest total size $\leq$ W

Brute force

Check all subsets --- 2^n

Intuition to build DP solution

Subproblems ??? ------Toughest part

Once you decide what subproblems are, everything will fall in place

Subproblems are guided by recursive formulation....

But how to write recursive formulation....

Suppose object o_1 is in the optimal solution, what we need to solve now ??

Suppose object o_1 is in the optimal solution, what we need to solve now ??

Then the remaining subset of objects $\{o_2, o_3, \ldots, o_n\}$ of largest size whose total size is atmost $W - s_1$.

Suppose object o_1 is not in the optimal solution, what we need to solve now ??

Suppose object o_1 is not in the optimal solution, what we need to solve now ??

Then the remaining subset of objects $\{o_2, o_3, \ldots, o_n\}$ of largest size whose total size is atmost W.

OPTIMAL SUBSTRUCTURE PROPERTY

Suppose object o_1 is in the optimal solution, what we need to solve now ??

Then the remaining subset of objects $\{o_2, o_3, \ldots, o_n\}$ of largest size whose total size is atmost $W - s_1$.

Suppose object o_1 is not in the optimal solution, what we need to solve now ??

Then the remaining subset of objects $\{o_2, o_3, \ldots, o_n\}$ of largest size whose total size is atmost W.

OPTIMAL SUBSTRUCTURE PROPERTY

If someone tell you that the object O_1 is there in an optimal solution, so now you can claim that remaining the remaining subset in that particular optimal solution is the largest size of the capacity at most $W - s_1$.

OPTIMAL SUBSTRUCTURE PROPERTY

If someone tell you that the object O_1 is there in an optimal solution, so now you can claim that remaining the remaining subset in that particular optimal solution is the largest size of the capacity at most $W - s_1$.

Proof…(By contradiction) Cut and Paste argument…

Recusrive Formulation

$$OPT(\{o_1, o_2, \dots, o_n\}, W) = \max \begin{cases} OPT(\{o_2, \dots, o_n\}, W) \\ OPT(\{o_1, o_2, \dots, o_n\}, W - s_1) + s_1 \end{cases}$$

Recusrive Formulation

o1 sholudn't be part of OPT

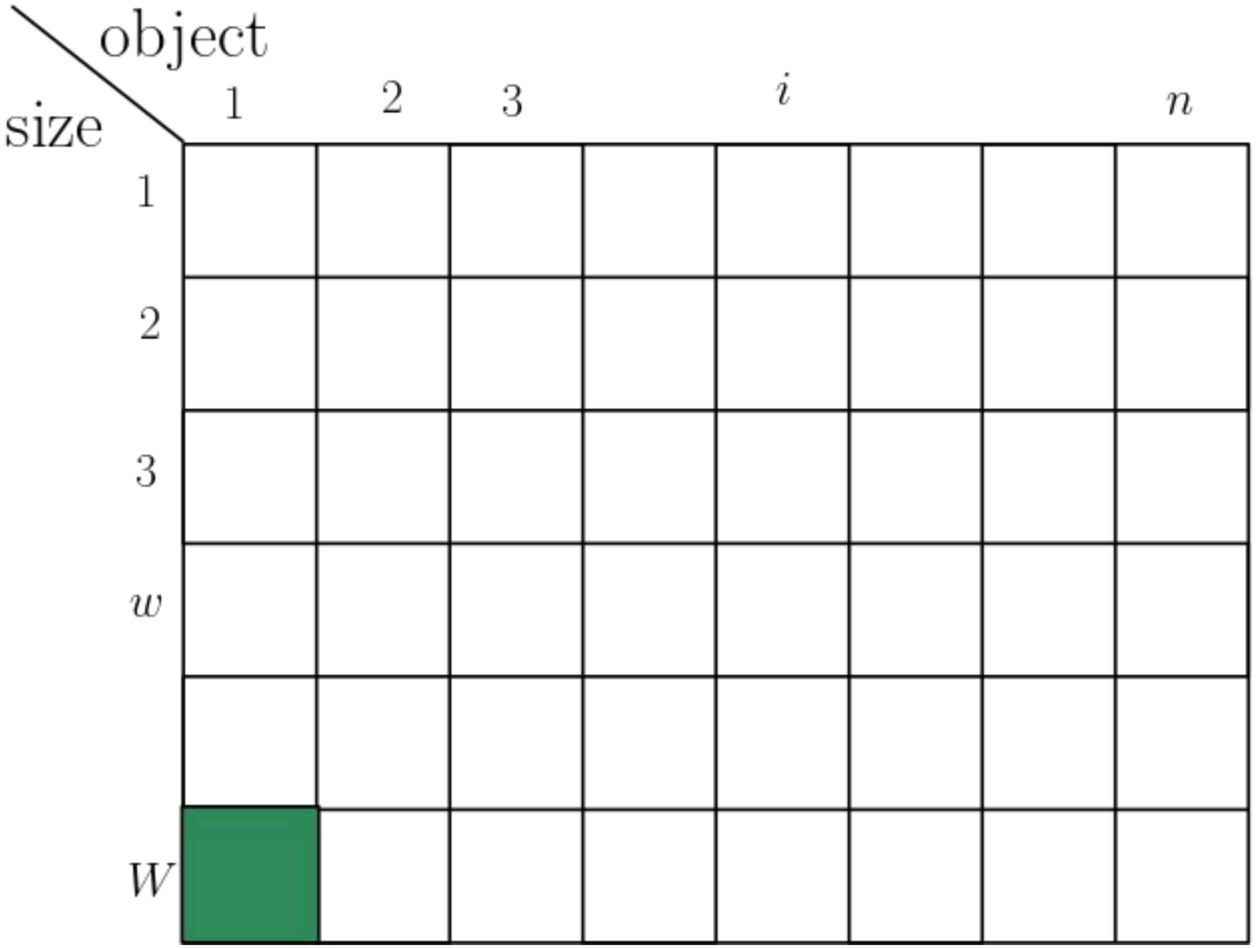
$$OPT(\{o_1, o_2, \ldots, o_n\}, W) = \max \begin{cases} OPT(\{o_2, \ldots, o_n\}, W) \\ OPT(\{o_1, o_2, \ldots, o_n\}, W - s_1) + s_1 \end{cases}$$

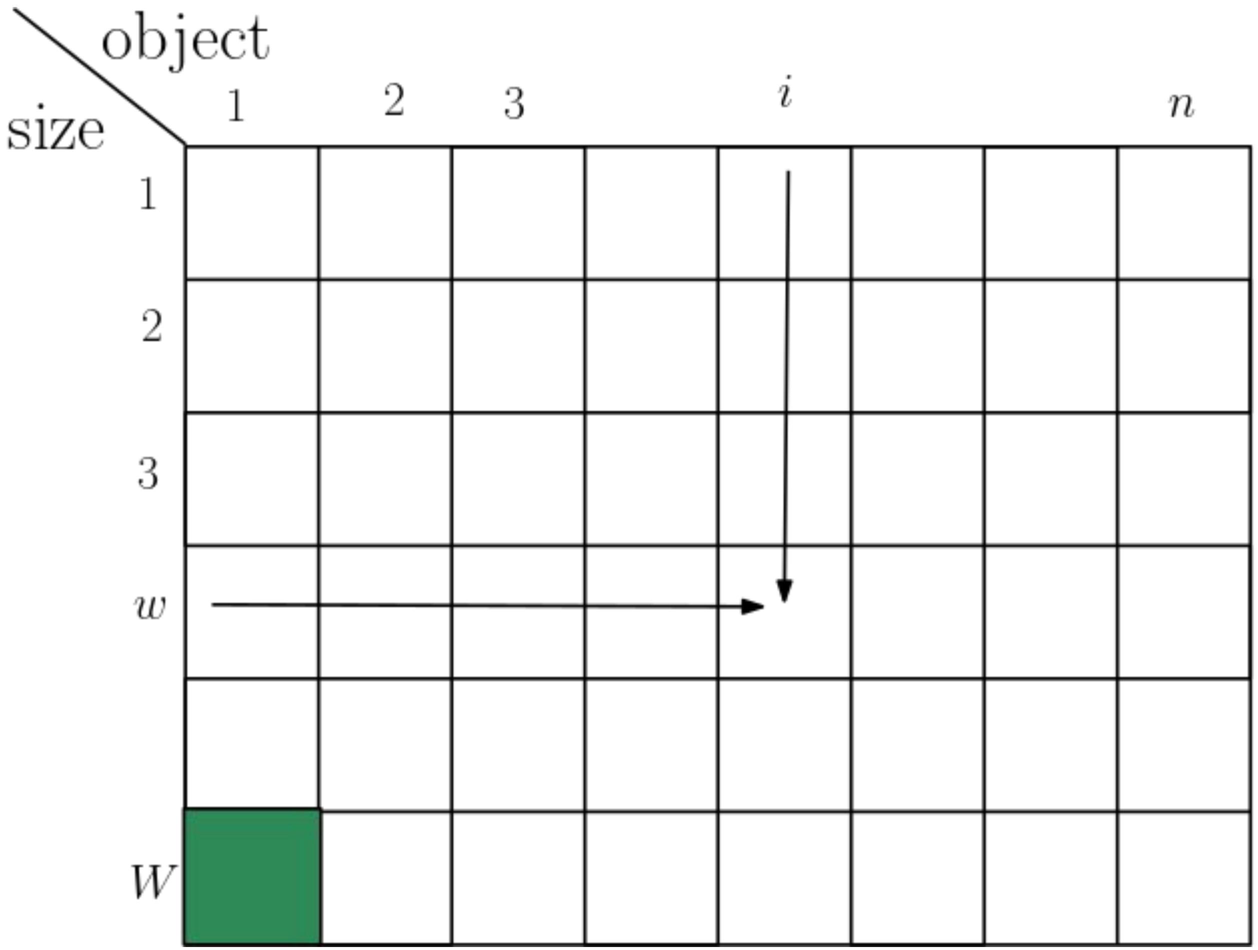
$$OPT(i, w) = \max \begin{cases} OPT(i+1, w) \\ OPT(i+1, w - s_i) + s_i \end{cases}$$

SEMANTICS

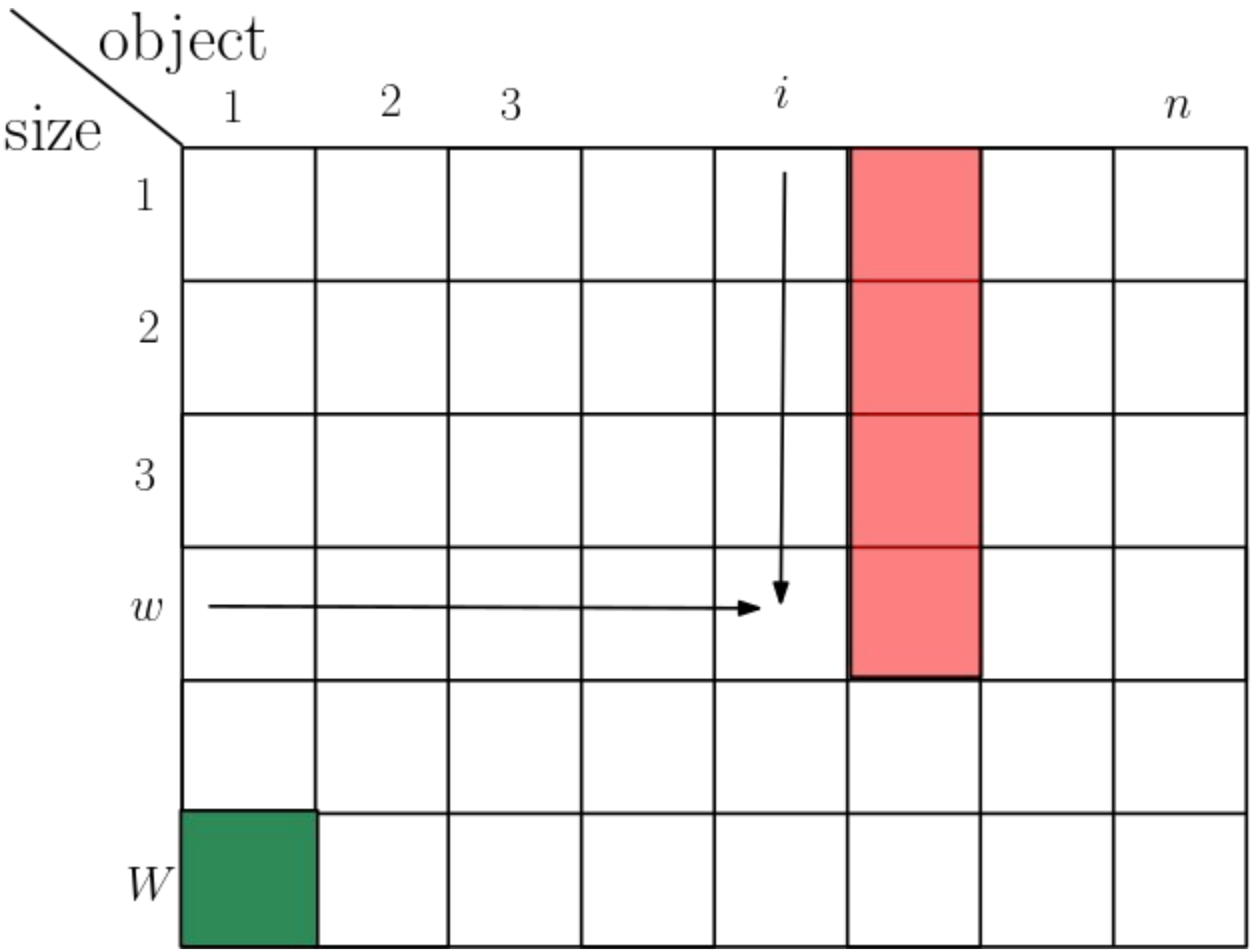
Largest sized subset {i,....,n} with total size at most w

size \ object	1	2	3					n
1								
2								
3								
W								

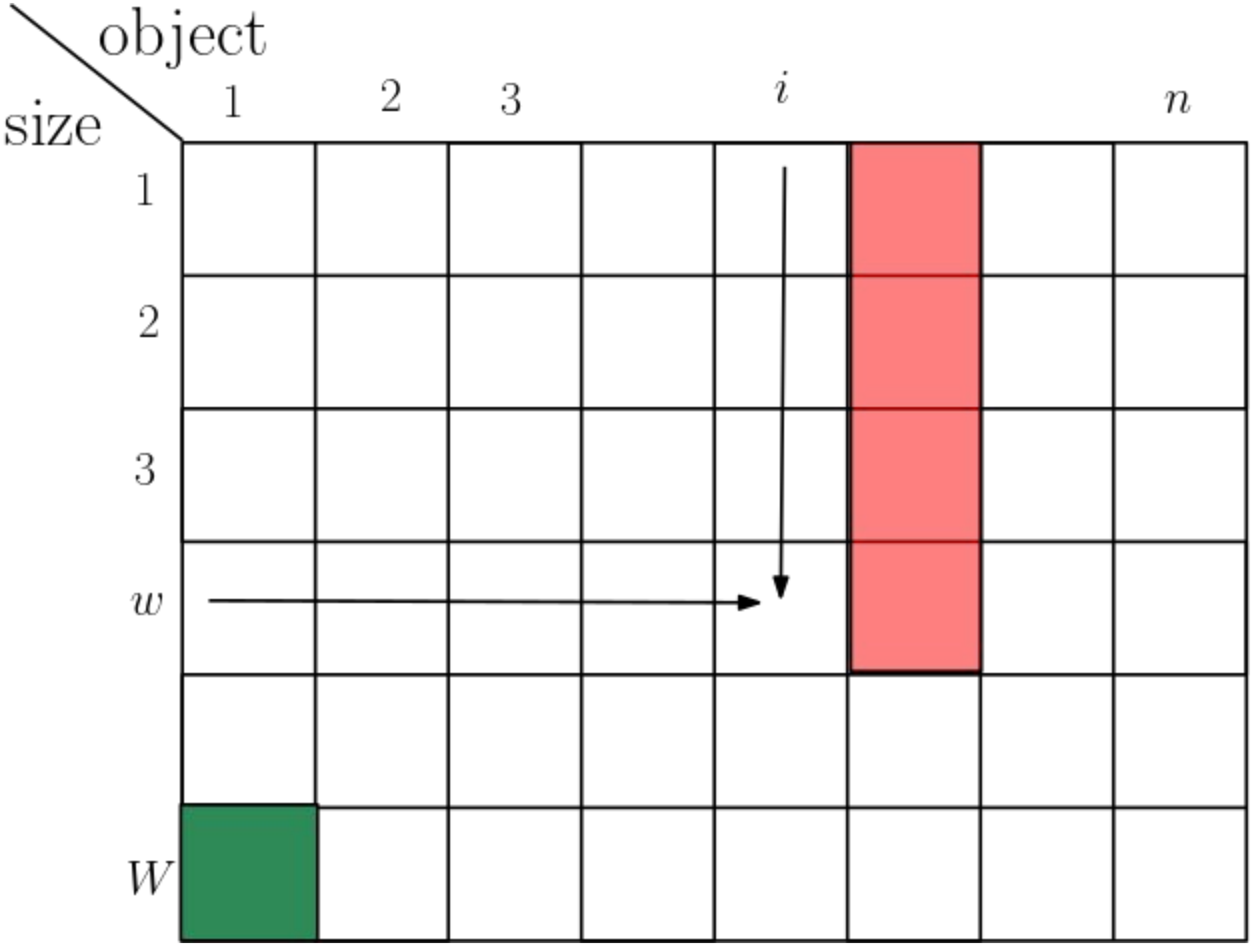

object
size
1
2
3
i
n
1
2
3
w
W



Prepared by Pawan R.



Prepared by Pawan R.

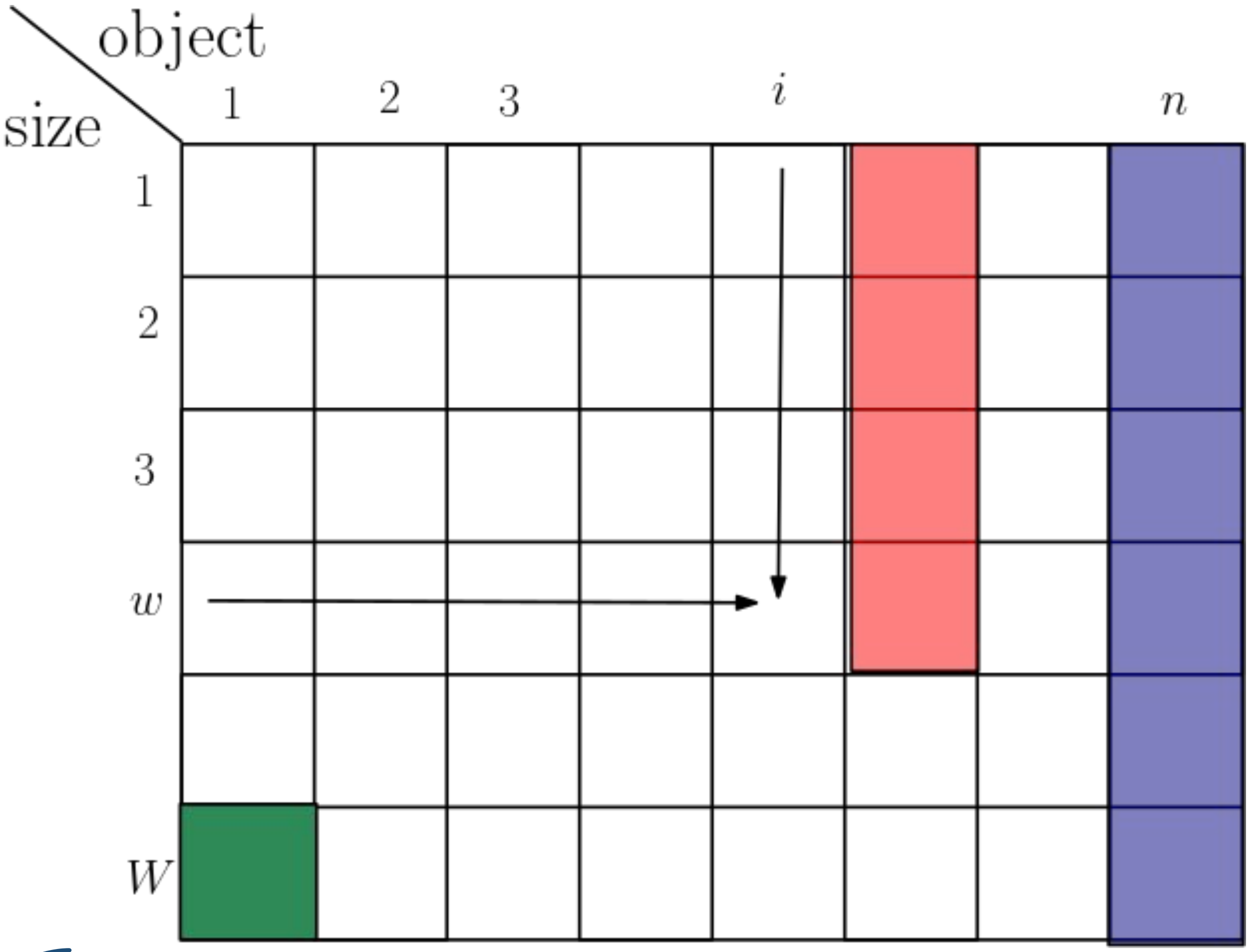


This basically tells you the direction via which you fill the table, thus we get the base case too

$$OPT(n, w) = \begin{cases} 0 & s_n > w \\ s_n & \text{otherwise} \end{cases}$$

For all w \in {1,2,…,W}

Prepared by *Pawan R.*



This basically tells you the direction via which you fill the table, thus we get the base case too

$OPT(n, w) = 0 \qquad s_n > w$
$\qquad\qquad\quad = s_n \qquad \text{otherwise}$

For all w \in {1,2,…,W}

Prepared by Pawan R.

Total Time

How many entries we are computing ?? nW

Time to compute each entry - Constant time..
By using max of two entries..

$O(nW)$

Cut-and-Paste Arguments

To show optimal substructure, assume that some piece of the optimal solution S* is not an optimal solution to a smaller subproblem

Show that replacing that piece with the optimal solution to the smaller subproblem improves the allegedly optimal solution S*.

Conclude, therefore, that S* must include an optimal solution to a smaller subproblem.

P2. Coin Change Problem

If we want to make change for Rs. T, and we have infinite supply of each coin in the set

Coins = $\{v_1, v_2, \ldots, v_n\}$, where v_i is the value of the i-th coin.

What is the minimum number of coins required to reach the value S?

Greedy Algorithm

Greedy Algorithm

Coins = {6, 4, 1} and T = 8

Counter Example

Greedy Algorithm

Coins = {6, 4, 1} and T = 8

Coins={1,5,10,25}-USA

Coins={1,2,5,10} -Indian

Dynamic Programming

$$OPT(T,n) = \min \begin{cases} OPT(T,\ n-1) & \text{First coin with value } v_1 \text{ is not used} \\ OPT(T-v_1,\ n)+1 & \text{First coin with value } v_1 \text{ is used} \end{cases}$$

Dynamic Programming

$$OPT(T,n) = \min \begin{cases} OPT(T,\ n-1) & \text{First coin with value } v_1 \text{ is not used} \\ OPT(T - v_1,\ n) + 1 & \text{First coin with value } v_1 \text{ is used} \end{cases}$$

$$OPT(S,i) = \min \begin{cases} OPT(S,\ i-1) & i\text{-th coin with value } v_i \text{ is not used} \\ OPT(S - v_i,\ i) + 1 & i\text{-th coin with value } v_i \text{ is used} \end{cases}$$

SEMANTICS

the minimum number of coins required to reach sum $S \leq T$ with the first i coins, i.e., coins selected from the subset $\{v_1, v_2, \ldots, v_i\}$ (where $0 \leq i \leq n$).

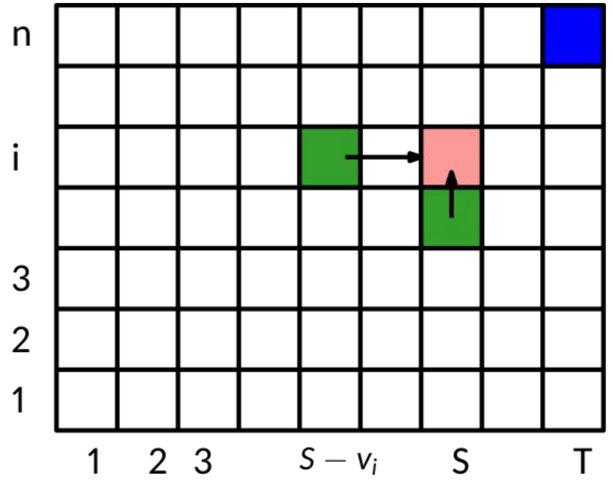
Prepared by Pawan R.

Dynamic Programming

$$OPT(S,i) = \min \begin{cases} OPT(S,\ i-1) & \text{i-th coin with value } v_i \text{ is not used} \\ OPT(S-v_i,\ i) + 1 & \text{i-th coin with value } v_i \text{ is used} \end{cases}$$

SEMANTICS

the minimum number of coins required to reach sum $S \leq T$ with the first i coins, i.e., coins selected from the subset $\{v_1, v_2, \ldots, v_i\}$ (where $0 \leq i \leq n$).



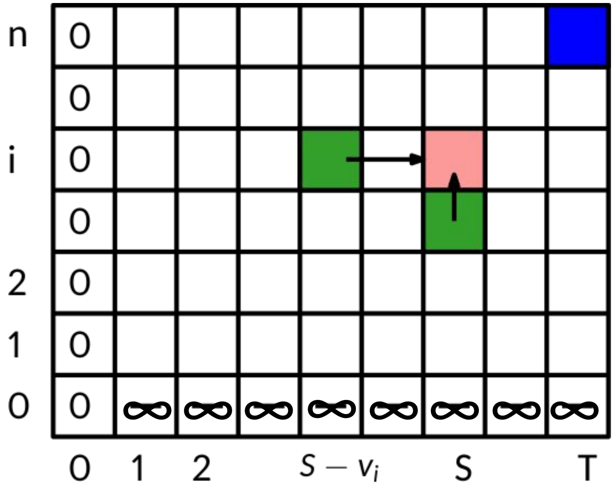
Prepared by Pawan R.

Dynamic Programming

$$OPT(S,i) = \min \begin{cases} OPT(S,\ i-1) & \text{i-th coin with value } v_i \text{ is not used} \\ OPT(S-v_i,\ i) + 1 & \text{i-th coin with value } v_i \text{ is used} \end{cases}$$

SEMANTICS

the minimum number of coins required to reach sum $S \leq T$ with the first i coins, i.e., coins selected from the subset $\{v_1,\ v_2, \ldots, v_i\}$ (where $0 \leq i \leq n$).



P3. Set Cover Problem

Given a universe $U=\{e_1, e_2, e_3, \ldots, e_n\}$ of n elements,

And a family of subsets $F = \{S_1, S_2, S_3, \ldots, S_m\}$.

find a minimum number of subcollection *C* of *F* such that *C* covers all elements of *U*.

$U = \{1, 2, 3, 4, 5\}$

$F = \{ S_1 = \{1, 2, 3\}, \quad S_2 = \{2,3\}, \quad S_3 = \{4, 5\}, \quad S_4 = \{1, 2, 4\} \}$

$C = \{S_1, S_3\}$

P3. Set Cover Problem

Given a universe $U=\{e_1, e_2, e_3, \ldots, e_n\}$ of n elements,

And a family of subsets $F = \{S_1, S_2, S_3, \ldots, S_m\}$.

find a minimum number of subcollection *C* of *F* such that *C* covers all elements of *U*.

Brute Force = $O(2^m \text{poly}(n))$

$U = \{1, 2, 3, 4, 5\}$

$F = \{ S_1 = \{1, 2, 3\}, \quad S_2 = \{2,3\}, \quad S_3 = \{4, 5\}, \quad S_4 = \{1, 2, 4\} \}$

$C = \{S_1, S_3\}$

Greedy Algorithm ??

Dynamic Programming

Can we improve to $O^*(2^n)$??

Dynamic Programming

Can we improve to $O^*(2^n)$??

Given : $F = \{S_1, S_2, S_3,\dots, S_m\}$, U

To find subproblem, but how ?--- need recurrence formulation …

Dynamic Programming

Can we improve to $O^*(2^n)$??

Given : $F = \{S_1, S_2, S_3, \ldots, S_m\}$, U

To find subproblem, but how ?--- need recurrence formulation …

How to proceed?

Is S_1 in optimal solution, then what to cover and by what ?

Is S_1 in not in optimal solution, then what to cover and by what ?

Dynamic Programming

Can we improve to $O^*(2^n)$??

Given : $F = \{S_1, S_2, S_3, \ldots, S_m\}$, U

To find subproblem, but how ?--- need recurrence formulation …

How to proceed?

Is S_1 in optimal solution, then what to cover and by what ? → $U \setminus S_1$ $\{S_2, S_3, \ldots, S_m\}$

Is S_1 in not in optimal solution, then what to cover and by what ? → U $\{S_2, S_3, \ldots, S_m\}$

Dynamic Programming

$$OPT(\{S_1, S_2, S_3, \ldots, S_n\}, U) = \min \begin{cases} OPT(\{S_2, S_3, \ldots, S_n\}, U) & S_1 \text{ is not used} \\ OPT(\{S_2, S_3, \ldots, S_n\}, U \setminus S_1) + 1 & S_1 \text{ is used} \end{cases}$$

Dynamic Programming

$$OPT(\{S_1, S_2, S_3, \dots, S_n\}, U) = \min \begin{cases} OPT(\{S_2, S_3, \dots, S_n\}, U) & S_1 \text{ is not used} \\ OPT(\{S_2, S_3, \dots, S_n\}, U \setminus S_1) + 1 & S_1 \text{ is used} \end{cases}$$

$$OPT(i, A) = \min \begin{cases} OPT(S_{i+1}, A) & S_i \text{ is not used} \\ OPT(S_{i+1}, A \setminus S_i) + 1 & S_i \text{ is used} \end{cases}$$

SEMANTICS

To cover a subset A of U, the minimum number of subset required of F from S_i to S_m.

Prepared by Pawan R.

Dynamic Programming

$$OPT(i,A) = \min \begin{cases} OPT(S_{i+1},\ A) & S_i \text{ is not used} \\ OPT(S_{i+1},\ A \setminus S_i) + 1 & S_i \text{ is used} \end{cases}$$

SEMANTICS

To cover a subset A of U, the minimum number of subset required of F from S_i to S_m.

Goal : OPT(1, U)

Prepared by Pawan K.

Dynamic Programming

$$OPT(i,A) = \min \begin{cases} OPT(S_{i+1},\ A) & S_i \text{ is not used} \\ OPT(S_{i+1},\ A \setminus S_i) + 1 & S_i \text{ is used} \end{cases}$$

SEMANTICS

To cover a subset A of U, the minimum number of subset required of F from S_i to S_m.

Base Case: We need i=1 as a goal, so basically we need in decreasing order, so m will play the role in defining the base case.

$$OPT(m,A) = \min \begin{cases} 1 & \text{If } A \text{ is subset of } S_m \\ \infty & \text{Otherwise} \end{cases}$$

Goal : OPT(1, U)

Dynamic Programming

$$OPT(i, A) = \min \begin{cases} OPT(S_{i+1}, A) & S_i \text{ is not used} \\ OPT(S_{i+1}, A \setminus S_i) + 1 & S_i \text{ is used} \end{cases}$$

SEMANTICS

To cover a subset A of U, the minimum number of subset required of F from S_i to S_m.

Base Case: We need i=1 as a goal, so basically we need in decreasing order, so m will play the role in defining the base case.

$$OPT(m, A) = \min \begin{cases} 1 & \text{If } A \text{ is subset of } S_m \\ \infty & \text{Otherwise} \end{cases}$$

Time Complexity=$O(2^n m)$

Goal : OPT(1, U)

Prepared by *Pawan R.*

P4. Maximum Contiguous Subsequence in Array

IP: Sequence $S=\{e_1, e_2, e_3, \ldots, e_n\}$ of n integers

OP: pair (i,j) $e_i + e_{i+1} + e_{i+2} + \ldots + e_j$ is maximum

P4. Maximum Contiguous Subsequence in Array

IP: Sequence $S=\{e_1, e_2, e_3, \ldots, e_n\}$ of n integers

OP: pair (i,j) $e_i + e_{i+1} + e_{i+2} + \ldots + e_j$ is maximum

Brute Force: For every pair, find sum: $O(n^3)$ or $O(n^2)$

DnC: O(nlogn), Tut-2,Q 8

Dynamic Programming

How to proceed?

Optimal solution contains element e_i ----

$$\begin{cases} 1.\ OPT(i-1) + e_i \\ 2.\ e_i \end{cases}$$

Optimal element does not contain element e_i.

This means that ——— e_i ———

$$OPT(i) = \max \begin{cases} OPT(i-1) + e_i \\ e_i \end{cases}$$

SEMANTICS

Optimal value if sub sequence ends at e_i.

P5. Longest Common Subword

Given two strings, find the (length of the) longest common subword

“secret”, “secretary” — “secret”, length 6

“bisect”, “trisect” — “sect”, length 4

More formally …

Let $u = a_0\, a_1 \ldots a_m$ and $v = b_0\, b_1 \ldots b_n$ be two sequences

If we can find i, j such that

$a_i\, a_{i+1} \ldots a_{i+k-1} = b_j\, b_{j+1} \ldots b_{j+k-1}$,

u and v have a common subword of length k

Aim is to find the length of the longest common subword of u and v

Brute force

Let $u = a_0\, a_1 \ldots\, a_m$ and $v = b_0\, b_1 \ldots\, b_n$

Try every pair of starting positions i in u, j in v

Match $(a_i, b_i), (a_{i+1}, b_{i+1}), \ldots$ as far as possible

Keep track of the length of the longest match

Assuming $m > n$, this is $O(mn^2)$

mn pairs of positions

From each starting point, scan can be $O(n)$

Inductive structure

Let $u = a_0\, a_1 \dots a_m$ and $v = b_0\, b_1 \dots b_n$

$a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$ is a common subword of length k at (i,j),

iff $a_{i+1} \dots a_{i+k-1} = b_{j+1} \dots b_{j+k-1}$ is a common sub word of length k-1 at (i+1,j+1).

LCW(i,j): length of the longest common subword starting at a_i and b_j

If $a_i \neq b_j$, **LCW(i,j)** is 0,

otherwise **1+LCW(i+1,j+1)**

Boundary condition: when we have reached the end of one of the words

Inductive structure

Consider positions 0 to m+1 in u, 0 to n+1 in v

To know the end of the sequence

m+1, n+1 means we have reached the end of the word

LCW(m+1,j) = 0 for all j

LCW(i,n+1) = 0 for all i

LCW(i,j) = 0, if $a_i \neq$

b_j

= 1+ **LCW(i+1,j+1)**,

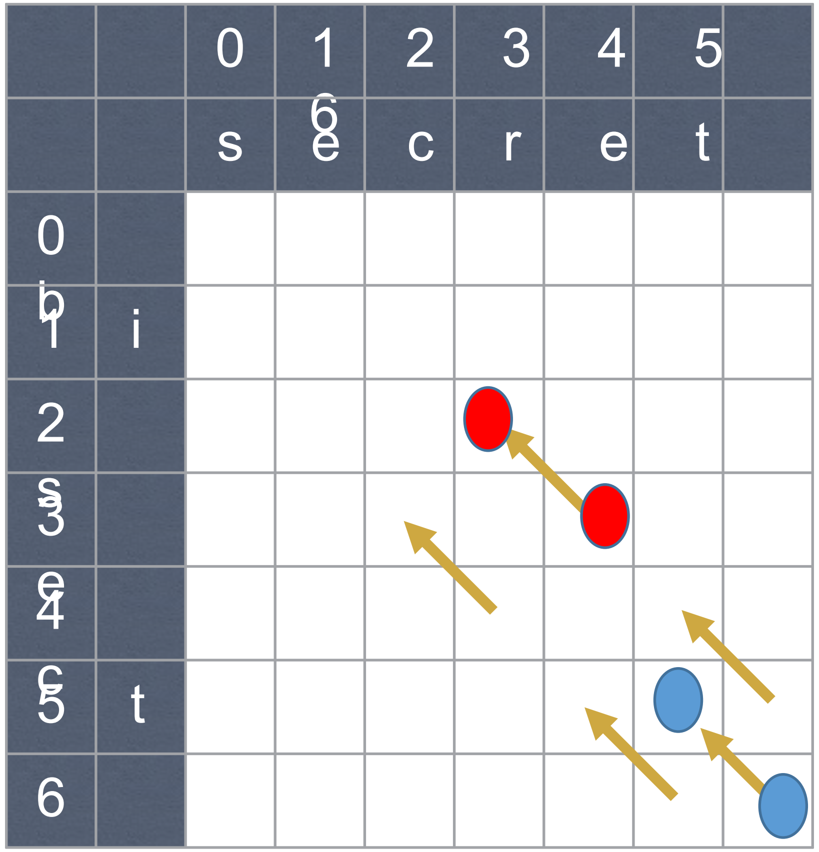
Prepared by Pawan R.

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column



Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6								

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							0
1	i							0
2	s							0
3	e							0
4	c							0
5	t							0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					0	0	0
1	i					0	0	0
2	s					0	0	0
3	e					1	0	0
4	c					0	0	0
5	t					0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				0	0	0	0
1	i				0	0	0	0
2	s				0	0	0	0
3	e				0	1	0	0
4	c				0	0	0	0
5	t				0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b			0	0	0	0	0
1	i			0	0	0	0	0
2	s			0	0	0	0	0
3	e			0	0	1	0	0
4	c			1	0	0	0	0
5	t			0	0	0	1	0
6	•	0	0	0	0	0	0	0

Prepared by Pawan R.

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b		0	0	0	0	0	0
1	i		0	0	0	0	0	0
2	s		0	0	0	0	0	0
3	e		2	0	0	1	0	0
4	c		0	1	0	0	0	0
5	t		0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCW(i,j) depends on LCW(i+1,j+1)

Last row and column have no dependencies

Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Reading off the solution

Find (i,j) with largest entry

LCW(2,0) = 3

Read off the actual subword diagonally

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Complexity

Recall that the brute force approach was $O(mn^2)$

The inductive solution is $O(mn)$ if we use dynamic programming (or memoization)

Need to fill an $O(mn)$ size table

Each table entry takes constant time to compute

P6. Longest common subsequence

Subsequence: can drop some letters in between

Given two strings, find the (length of the) longest common subsequence

"secret", "secretary" — "secret", length 6

"bisect", "trisect" — "isect", length 5

"bisect", "secret" — "sect", length 4

"director", "secretary" —"ectr", "retr", length 4

Prepared by Pawan R.

Applications

Analyzing genes of two species (in bionforamtics..)

DNA is a long string over 4 proteins A,T,G,C

Two species are closer if their DNA has longer common subsequence

Unix Command diff

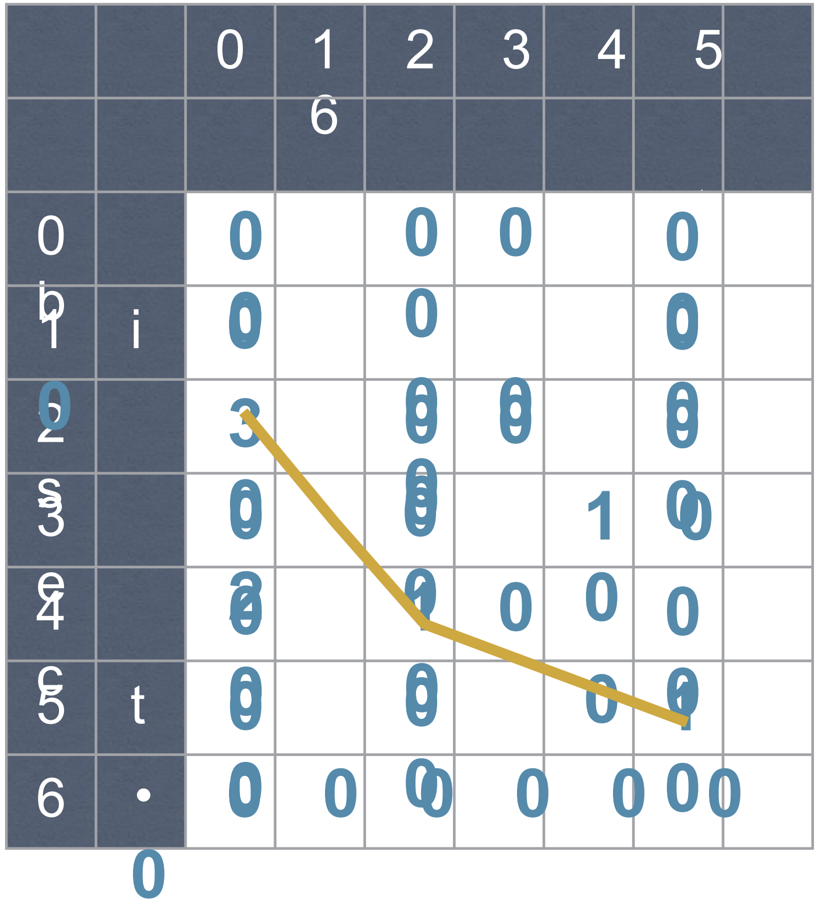
Compares text files

Find longest matching subsequence of lines

LCS

LCS is longest path we can find between non-zero LCW entries, moving right and down

“bisect”, “secret” — “sect”, length 4



Inductive structure

a_0	a_1	a_2	…	…	a_{m-1}	a_m
b_0	b_1	b_2	…	b_{n-1}	b_n	

If $a_0 = b_0$,

$LCS(a_0a_1 \ldots a_m, b_0b_1 \ldots b_n) = 1 + LCS(a_1a_2 \ldots a_m, b_1b_2 \ldots b_n)$

Can force (a_0, b_0) to be part of LCS

If not, a_0 and b_0 cannot both be part of LCS

Not sure which one to drop

Solve both subproblems

$LCS(a_1 a_2 \ldots a_m, b_0 b_1 \ldots b_n)$ and $LCS(a_0 a_1 \ldots a_m, b_1 b_2 \ldots b_n)$, and take the maximum

Inductive structure

a_0	a_1	a_2	…	…	a_{m-1}	a_m
b_0	b_1	b_2	…	b_{n-1}	b_n	

LCS(i,j) stands for LCS($a_i\, a_{i+1} \ldots a_m$, $b_j\, b_{j+1} \ldots b_n$)

If $a_i = b_j$, LCS(i,j) = 1 + LCS(i+1,j+1)
If $a_i \neq b_j$, LCS(i,j) = max(LCS(i+1,j), LCS(i,j+1))

As with LCW, extend positions to m+1, n+1

LCS(m+1,j) = 0 for all j

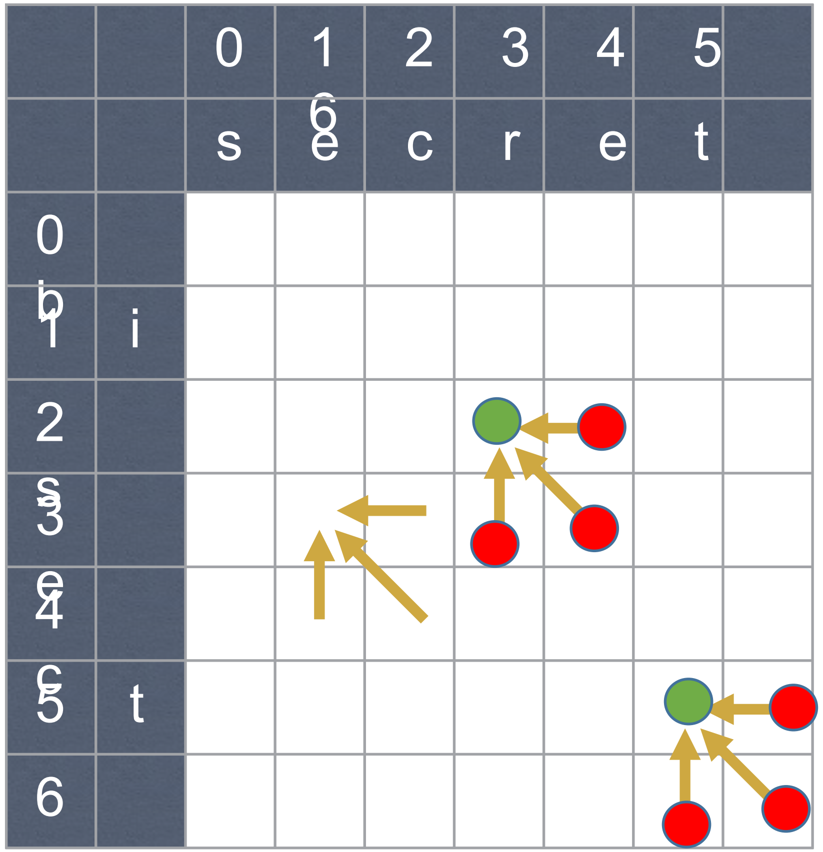
LCS(i,n+1) = 0 for all i

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal



Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1 6	2	3	4	5	
		s	e	c	r	e	t	
0								
b 1	i							
2								
s 3								
e 4								
c 5	t							
6								

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							0
1	i							0
2	s							0
3	e							0
4	c							0
5	t							0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					1	0	0
1	i					1	0	0
2	s					1	0	0
3	e					1	0	0
4	c					1	0	0
5	t					1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				1	1	0	0
1	i				1	1	0	0
2	s				1	1	0	0
3	e				1	1	0	0
4	c				1	1	0	0
5	t				1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b			2	1	1	0	0
1	i			2	1	1	0	0
2	s			2	1	1	0	0
3	e			2	1	1	0	0
4	c			2	1	1	0	0
5	t			1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b		3	2	1	1	0	0
1	i		3	2	1	1	0	0
2	s		3	2	1	1	0	0
3	e		3	2	1	1	0	0
4	c		2	2	1	1	0	0
5	t		1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)

Dependencies for LCS(m,n) are known

Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Prepared by Pawan R.

Recovering the sequence

Trace back the path by which each entry was filled

Each diagonal step is an element of the LCS

"sect"

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Prepared by Pawan R.

Recovering the sequence

Trace back the path by which each entry was filled

Each diagonal step is an element of the LCS

"sect"

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Complexity

Again $O(mn)$ using dynamic programming
(or memoization)

Need to fill an $O(mn)$ size table

Each table entry takes constant time to compute

P7. Multiplying matrices

To multiply matrices A and B, need compatible dimensions

A of dimension m x n, B of dimension n x p AB

has dimension mp

Each entry in AB take O(n) steps to compute AB[i,j]

is A[i,1]B[1,j] + A[i,2]B[2,j] +…+ A[i,n]B[n,j]

Overall, computing AB is O(mnp)

Multiplying matrices

Matrix multiplication is associative

$ABC = (AB)C = A(BC)$

Bracketing does not change the answer …

… but can affect the complexity of computing it!

Multiplying matrices

Suppose dimensions are A[1,100], B[100,1], C[1,100]

- Computing A(BC)
  - BC is [100,100], 100 x 1 x 100 = 10000 steps
  - A(BC) is [1,100],1 x 100 x 100 = 10000 steps
- Computing (AB)C
  - AB is [1,1], 1 x 100 x 1 = 100 steps
  - (AB)C is [1,100], 1 x 1 x 100 = 100 steps

A(BC) takes 20000 steps, (AB)C takes 200 steps!

Multiplying matrices

Given matrices M_1, M_2,…, M_n of dimensions $[r_1,c_1]$, $[r_2,c_2]$, …, $[r_n,c_n]$

Dimensions match, so $M_1 \times M_2 \times \ldots \times M_n$ can be computed

$c_i = r_{i+1}$ for $1 \leq i < n$

Find an optimal order to compute the product

That is, bracket the expression optimally

Inductive structure

Product to be computed: $M_1 \times M_2 \times \ldots \times M_n$

Final step would have combined two subproducts

$(M_1 \times M_2 \times \ldots \times M_k) \times (M_{k+1} \times M_{k+2} \times \ldots \times M_n)$, for some $1 \leq k < n$

First factor has dimension (r_1, c_k), second (r_{k+1}, c_n)

Final multiplication step costs $= r_1 c_k c_n$

Add cost of computing the two factors

Subproblems

Final step is

$(M_1 \times M_2 \times \ldots \times M_k) \times (M_{k+1} \times M_{k+2} \times \ldots \times M_n)$

Subproblems are $(M_1 \times M_2 \times \ldots \times M_k)$ and $(M_{k+1} \times M_{k+2} \times \ldots \times M_n)$

Total cost is $\text{Cost}(M_1 \times M_2 \times \ldots \times M_k) + \text{Cost}(M_{k+1} \times M_{k+2} \times \ldots \times M_n)$
$+ r_1 c_k c_n$

Which k should we choose?

No idea! Try them all and choose the minimum!

Inductive formulation

$Cost(M_1 \times M_2 \times \ldots \times M_n) =$

minimum value, for $1 \leq k < n$, of

$Cost(M_1 \times M_2 \times \ldots \times M_k) +$

$Cost(M_{k+1} \times M_{k+2} \times \ldots \times M_n) +$

$r_1 c_k c_n$

When we compute $Cost(M_1 \times M_2 \times \ldots \times M_k)$ we will get subproblems of the form $M_j \times M_{j+1} \times \ldots \times M_k$

In general …

$Cost(M_i \times M_{i+1} \times \ldots \times M_j) =$

minimum value, for $i \le k < j$, of

$$Cost(M_i \times M_{i+1} \times \ldots \times M_k) + Cost(M_{k+1} \times M_{k+2} \times \ldots \times M_j) + r_i c_k c_j$$

Write $Cost(i,j)$ to denote $Cost(M_i \times M_{i+1} \times \ldots \times M_j)$

Final equation

$Cost(i,i) = 0$ — No multiplication to be done

$Cost(i,j) = \min$ over $i \leq k < j$

$$[\ Cost(i,k) + Cost(k+1,j) + r_i c_k c_j\ ]$$

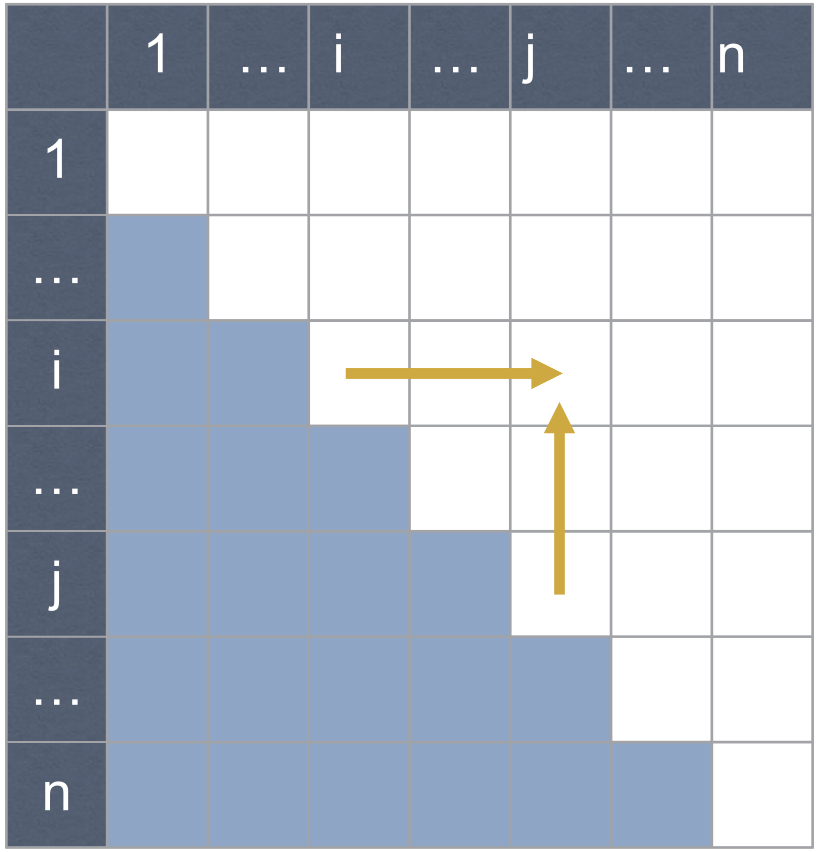
Note that we only require $Cost(i,j)$ when $i \leq j$

Subproblem dependency

Cost(i,j) depends on
Cost(i,k), Cost(k+1,j) for all $i \leq k < j$

Can have O(n) dependent values, unlike LCS, LCW, ED

Start with main diagonal and fill matrix by columns, bottom to top, left to right

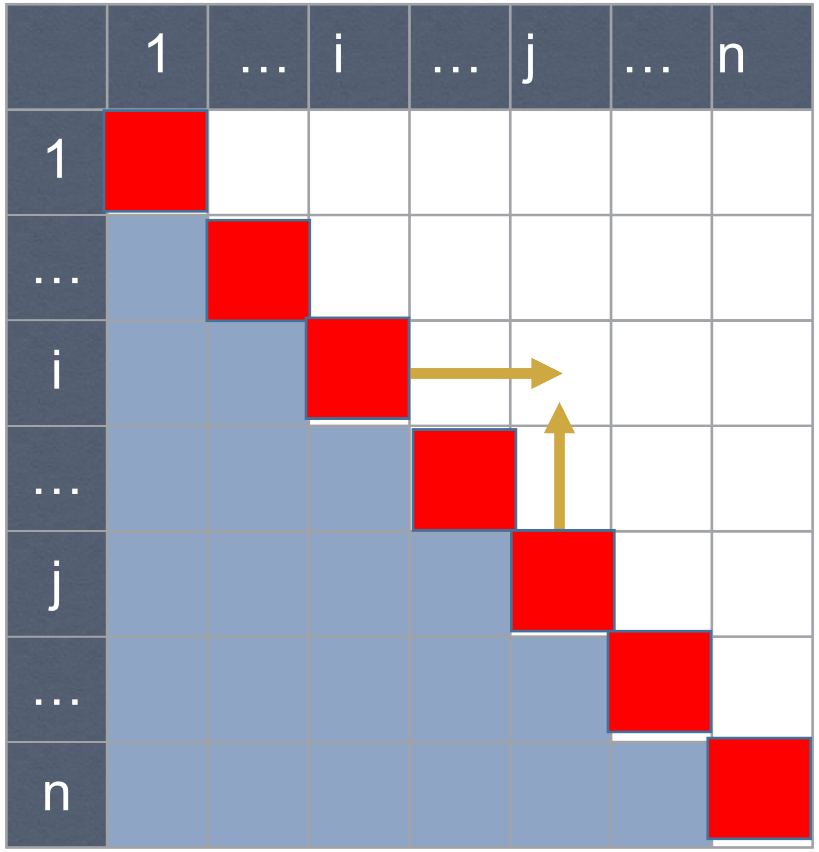


Subproblem dependency

Cost(i,j) depends on
Cost(i,k), Cost(k+1,j) for all i ≤ k < j

Can have O(n) dependent values, unlike LCS, LCW, ED

Start with main diagonal and fill matrix by columns, bottom to top, left to right

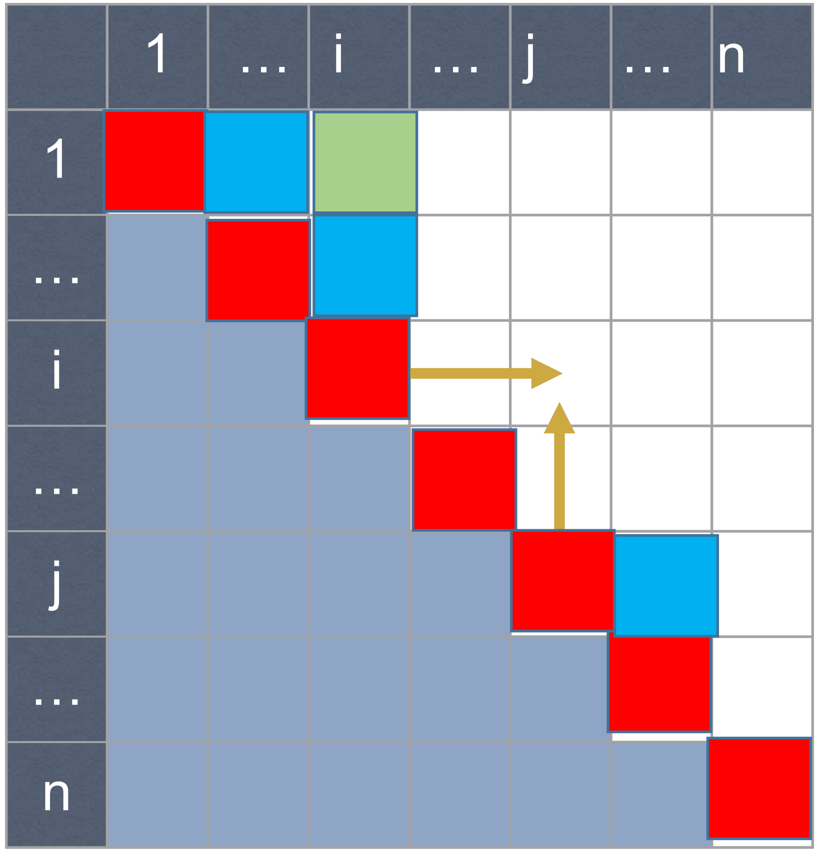


Subproblem dependency

Cost(i,j) depends on
Cost(i,k), Cost(k+1,j) for all $i \leq k < j$

Can have O(n) dependent values, unlike LCS, LCW, ED

Start with main diagonal and fill matrix by columns, bottom to top, left to right

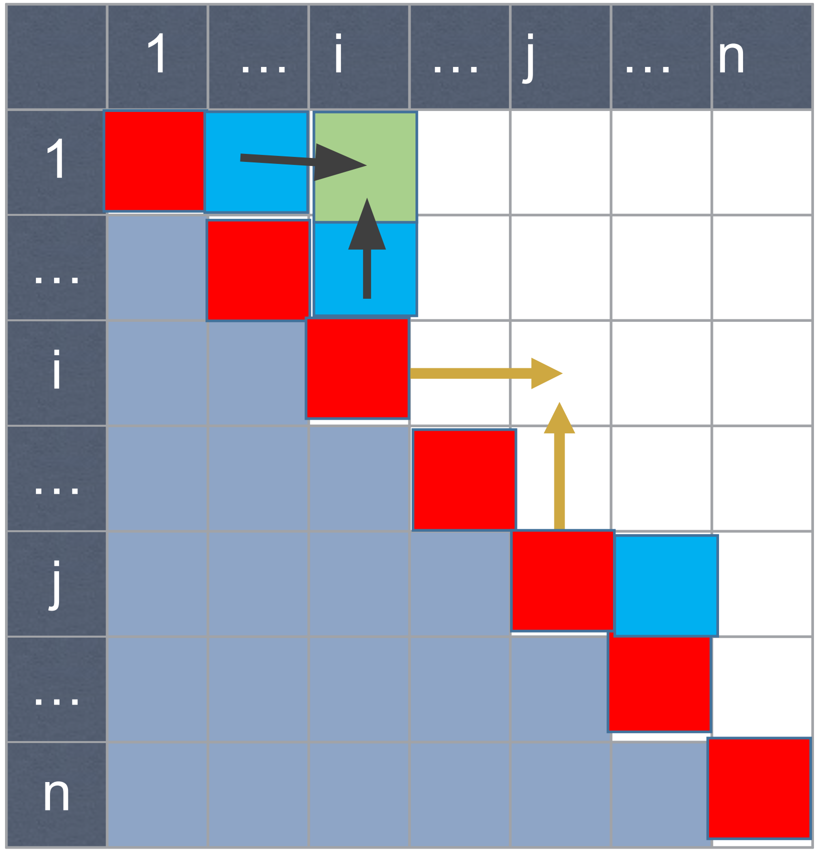


Subproblem dependency

Cost(i,j) depends on
Cost(i,k), Cost(k+1,j) for all $i \leq k < j$

Can have O(n) dependent values, unlike LCS, LCW, ED

Start with main diagonal and fill matrix by columns, bottom to top, left to right



Complexity

As with LCS, we to fill an $O(n^2)$ size table

However, filling MMC[i][j] could require examining $O(n)$ intermediate values

Hence, overall complexity is $O(n^3)$

P8. Maximum Independent Set on tree

P8. Minimum Vertex Cover on tree

Exercise : Based on the previous idea

P9. Diameter of a rooted tree

P10. Weighted tree, find a maximum sum path between any two nodes

Excercise

P11. Colouring in tree

I/P : given a tree,
O/P: color nodes black as many as possible without coloring two adjacent nodes

P11. Colouring in tree

I/P : given a tree,
O/P: color nodes black as many as possible without coloring two adjacent nodes

Minimum number of colors need to color nodes of a tree ??

Prepared by *Pawan R.*