Memoization

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 9

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- Insertion sort
  - $isort([\,]) = [\,]$ p
  - $isort([x_0, x_1, \ldots, x_n]) =$
    $insert(isort([x_0, x_2, \ldots, x_{n-1}]), x_n)$

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- Insertion sort
  - $isort([\,]) = [\,]$ p
  - $isort([x_0, x_1, \ldots, x_n]) =$
    $insert(isort([x_0, x_2, \ldots, x_{n-1}]), x_n)$

```
def fact(n):
  if n <= 0:
    return(1)
  else:
    return(n * fact(n-1))
```

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- Insertion sort
  - $isort([\,]) = [\,]$ p
  - $isort([x_0, x_1, \ldots, x_n]) =$
    $insert(isort([x_0, x_2, \ldots, x_{n-1}]), x_n)$

```
def fact(n):
  if n <= 0:
    return(1)
  else:
    return(n * fact(n-1))
```

- $fact(n-1)$ is a subproblem of $fact(n)$
  - So are $fact(n-2)$, $fact(n-3)$, ..., $fact(0)$
- $isort([x0, x_1, \ldots, x_{n-1}])$ is a subproblem of $isort([x_0, x_2, \ldots, x_n])$
  - So is $isort([x_i, \ldots, x_j])$ for any $0 \leq i < j \leq n$
- Solution to original problem can be derived by combining solutions to subproblems

Evaluating subproblems

- Fibonacci numbers
  - $\mathit{fib}(0) = 0$
  - $\mathit{fib}(1) = 1$
  - $\mathit{fib}(n) = \mathit{fib}(n-1) + \mathit{fib}(n-2)$

Evaluating subproblems

- Fibonacci numbers
  - $fib(0) = 0$
  - $fib(1) = 1$
  - $fib(n) = fib(n-1) + fib(n-2)$

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  return(value)
```

Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

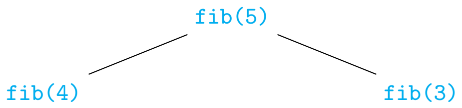
Evaluating `fib(5)`

`fib(5)`

Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

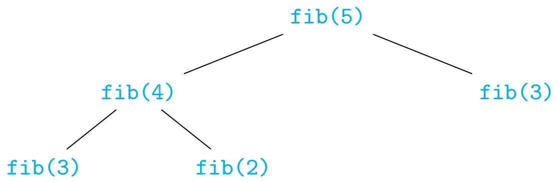
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

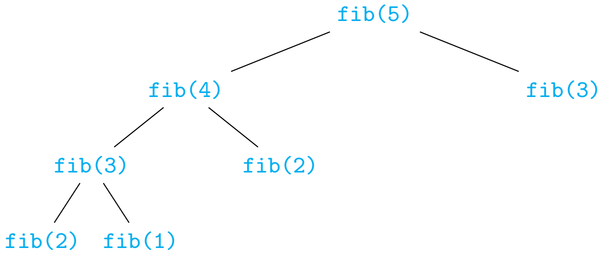
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

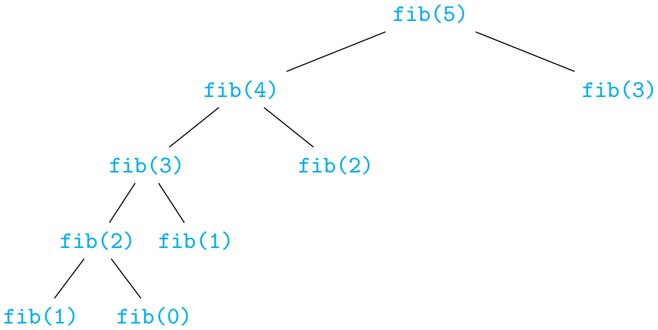
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

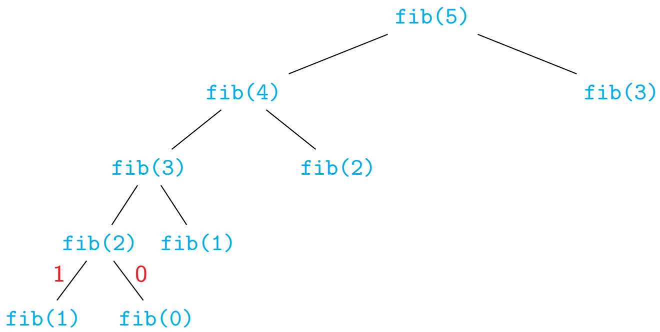
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

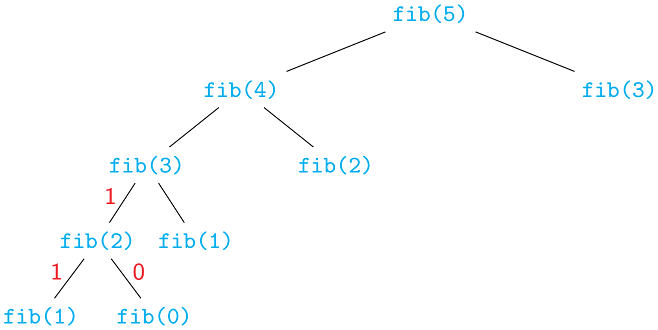
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

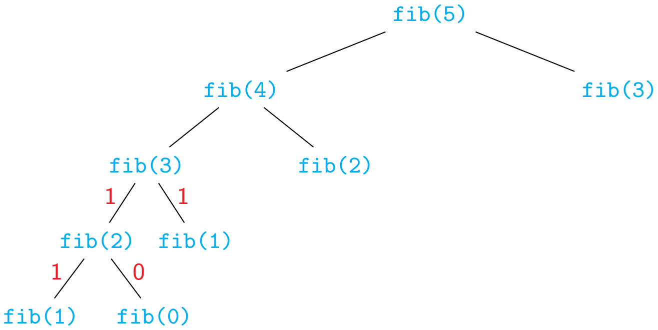
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

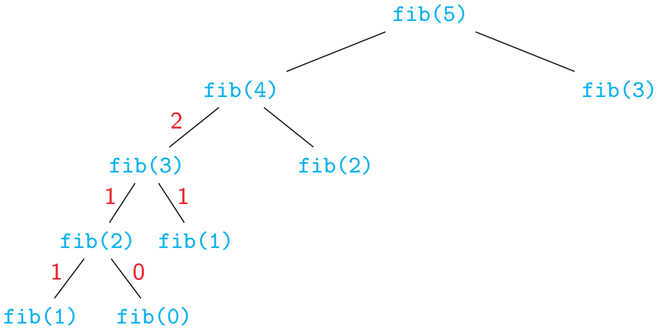
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

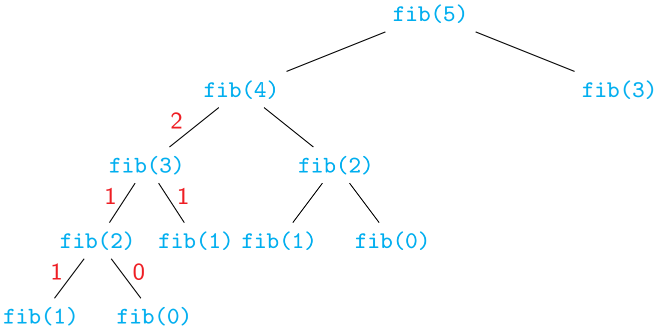
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

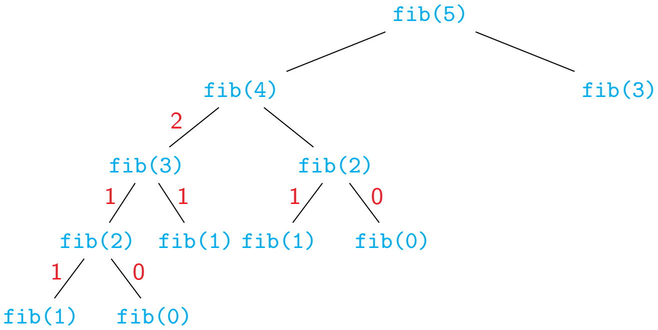
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

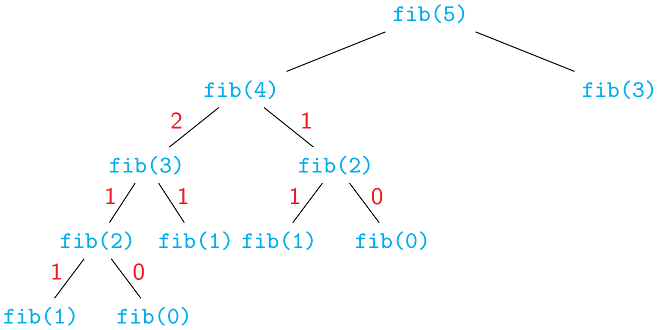
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

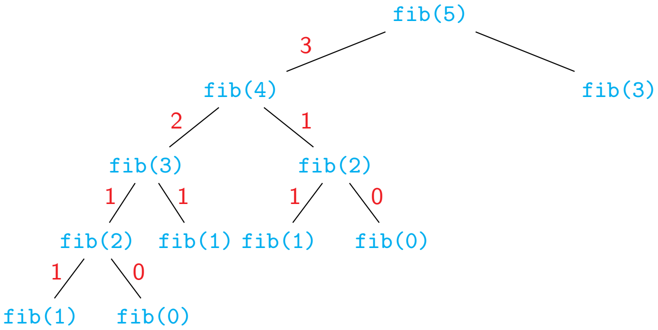
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

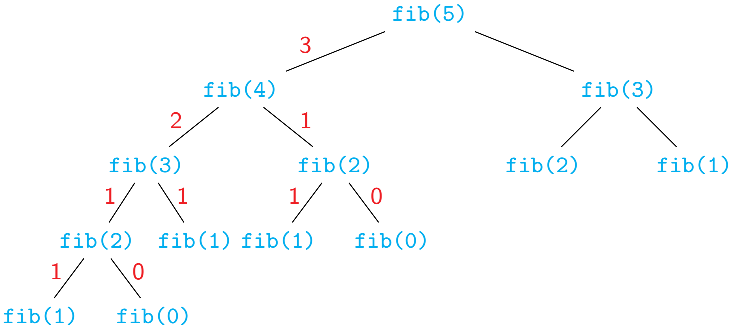
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

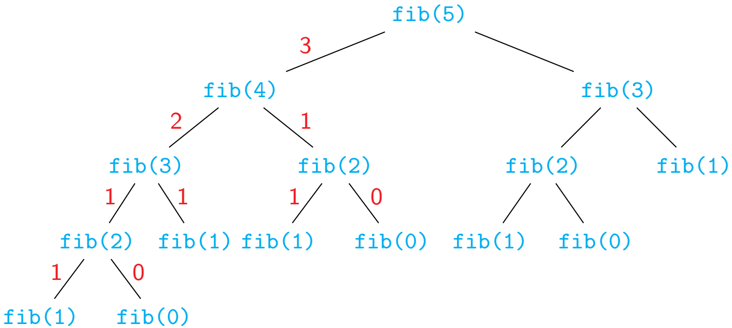
Evaluating fib(5)



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

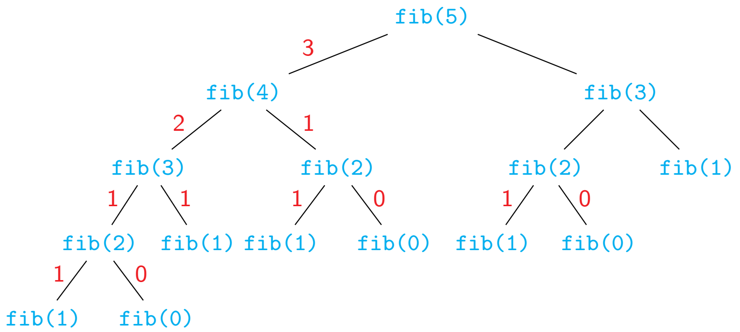
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

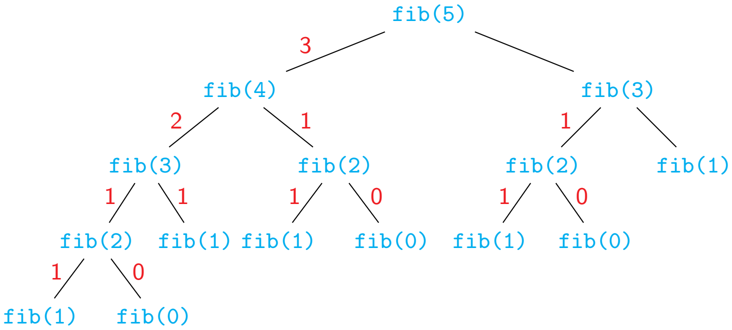
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

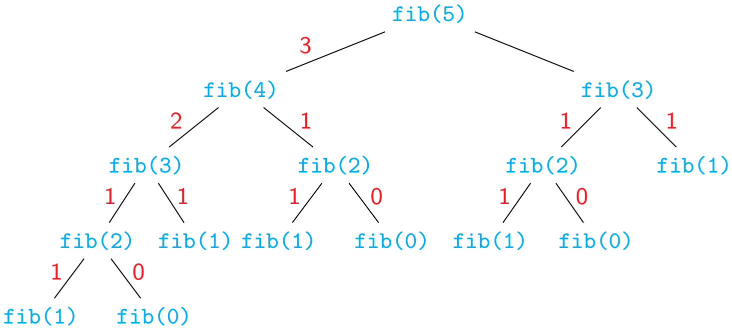
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

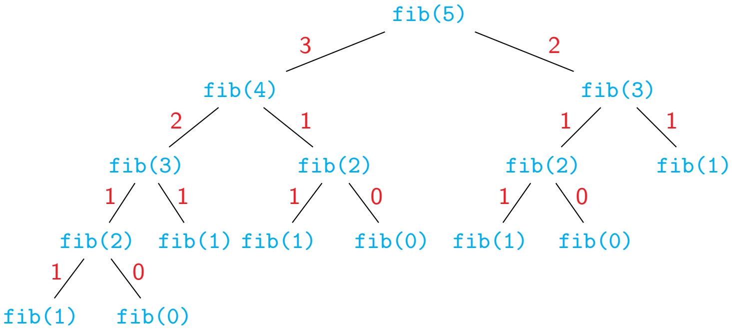
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

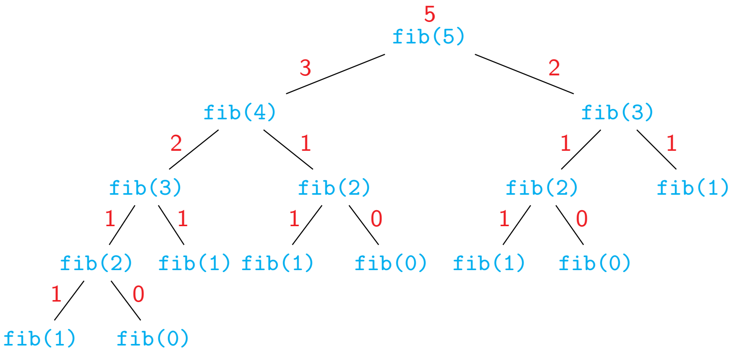
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

Evaluating `fib(5)`

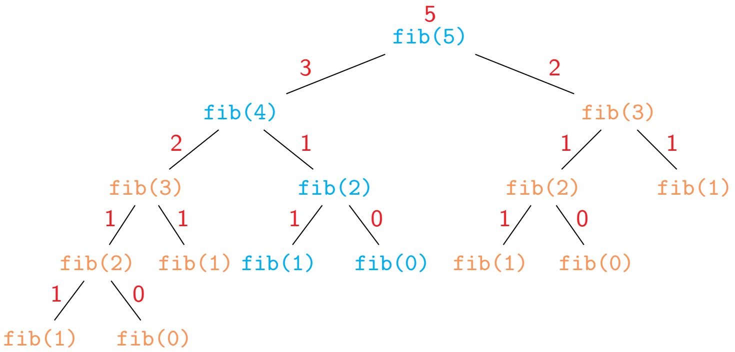


Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

- Wasteful recomputation
- Computation tree grows exponentially

Evaluating `fib(5)`

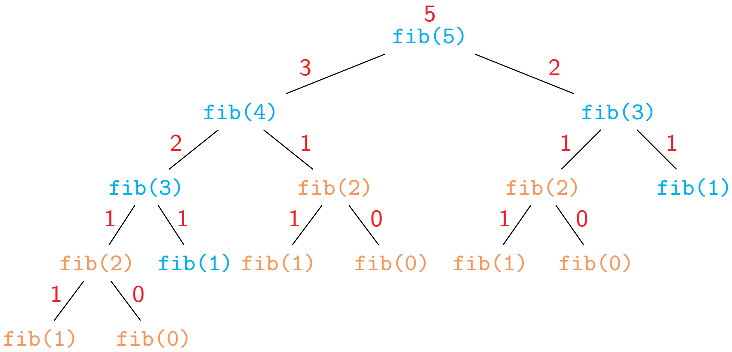


Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

- Wasteful recomputation
- Computation tree grows exponentially

Evaluating `fib(5)`



Evaluating subproblems

- Build a table of values already computed
  - Memory table

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear

Evaluating subproblems

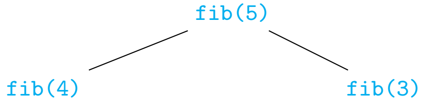
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear

`fib(5)`

k						
fib(k)						

Evaluating subproblems

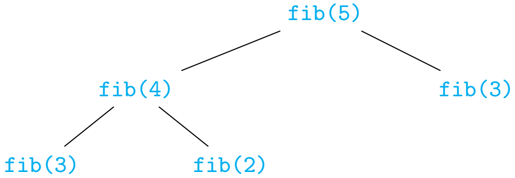
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

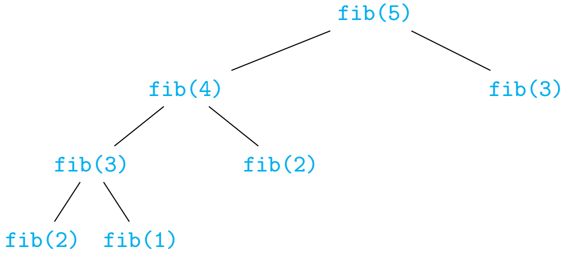
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

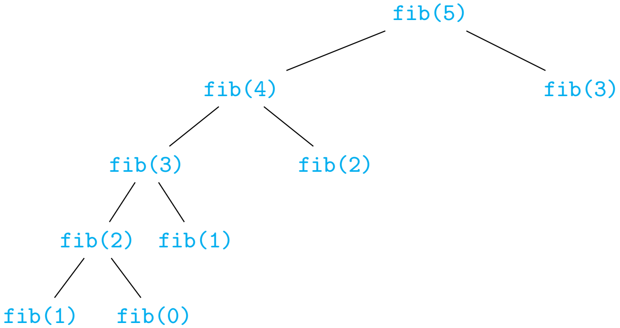
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

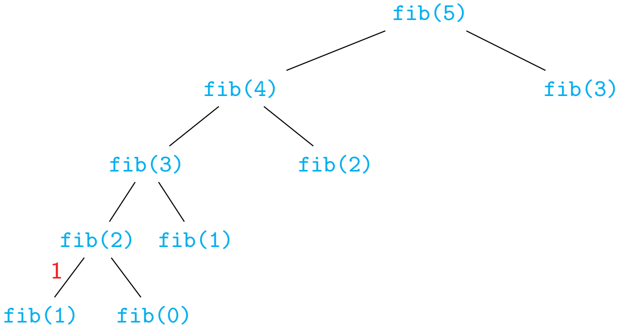
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

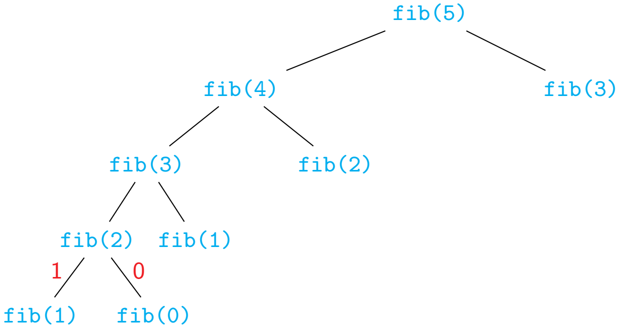
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1					
fib(k)	1					

Evaluating subproblems

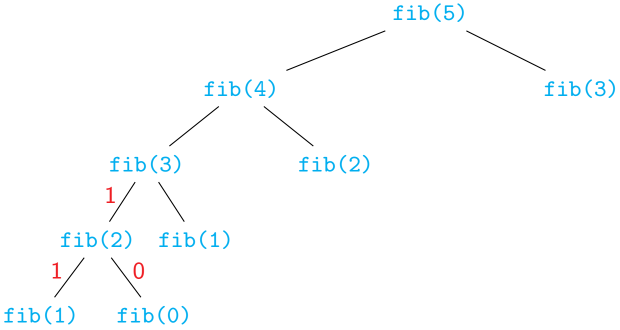
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0				
fib(k)	1	0				

Evaluating subproblems

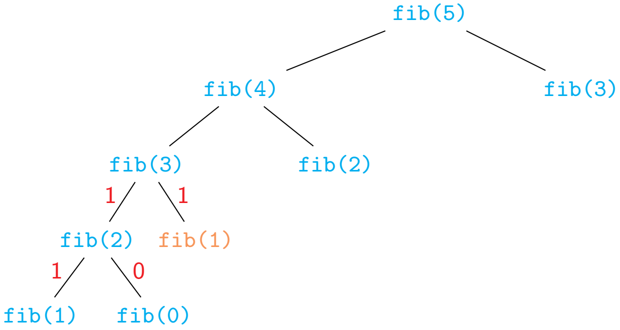
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2			
fib(k)	1	0	1			

Evaluating subproblems

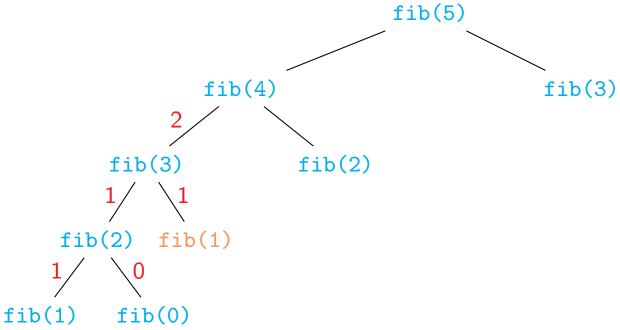
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2			
fib(k)	1	0	1			

Evaluating subproblems

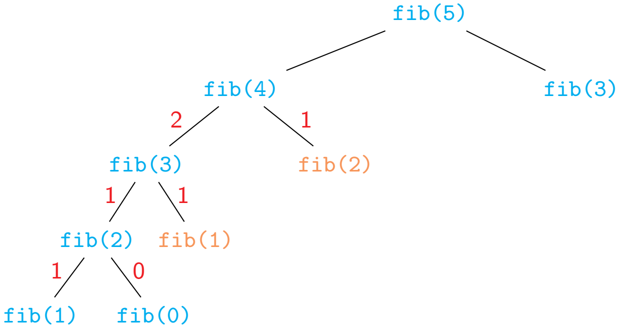
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



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

Evaluating subproblems

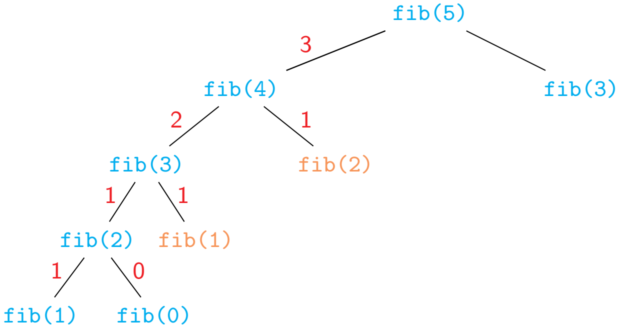
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



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

Evaluating subproblems

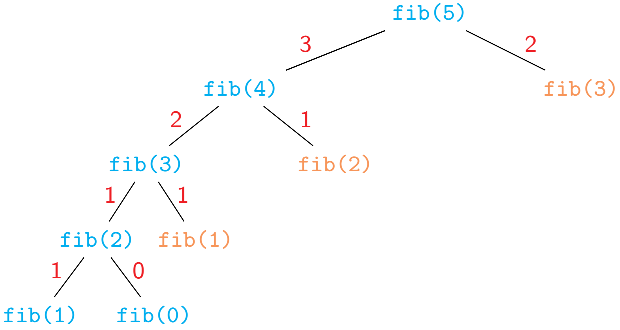
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



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

Evaluating subproblems

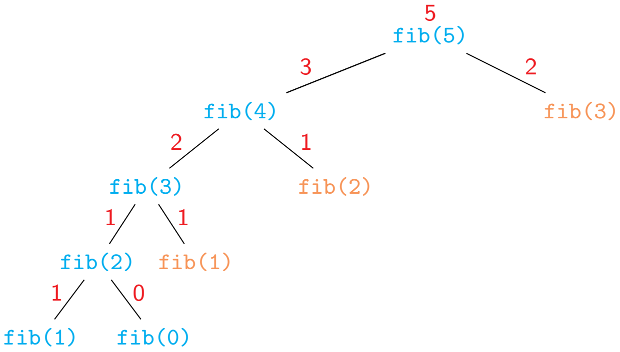
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



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

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



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

Memoizing recursive implmentations

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  return(value)
```

Memoizing recursive implmentations

```
def fib(n):
  if n in fibtable.keys():
    return(fibtable[n])
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

Memoizing recursive implmentations

```
def fib(n):
  if n in fibtable.keys():
    return(fibtable[n])
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

In general

```
def f(x,y,z):
  if (x,y,z) in ftable.keys():
    return(ftable[(x,y,z)])
  recursively compute value
    from subproblems
  ftable[(x,y,z)] = value
  return(value)
```

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a dag

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a dag
- Solve subproblems in topological order
  - Never need to make a recursive call

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a dag
- Solve subproblems in topological order
  - Never need to make a recursive call

Evaluating `fib(5)`

`fib(5)`

`fib(4)`

`fib(3)`

`fib(2)`

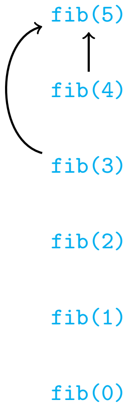
`fib(1)`

`fib(0)`

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a dag
- Solve subproblems in topological order
  - Never need to make a recursive call

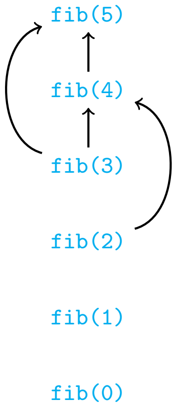
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a dag
- Solve subproblems in topological order
  - Never need to make a recursive call

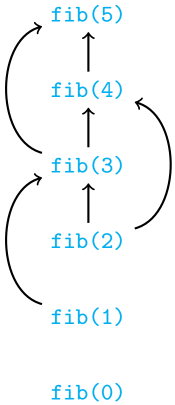
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a dag
- Solve subproblems in topological order
  - Never need to make a recursive call

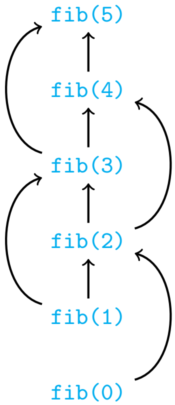
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a dag
- Solve subproblems in topological order
  - Never need to make a recursive call

Evaluating `fib(5)`



Summary

Memoization

- Store subproblem values in a table
- Look up the table before making a recursive call

Summary

Memoization

- Store subproblem values 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
- Iterative evaluation of subproblems, no recursion