CS 218 Design and Analysis of Algorithms

Nutan Limaye

Indian Institute of Technology, Bombay
nutan@cse.iitb.ac.in

Module 1: Basics of algorithms

Fibonacci numbers

Computing Fibonacci numbers

Input: Given 1^k

Output: kth Fibonacci number.

Recurrence $f(k) = f(k-1) + f(k-2)$.

$\texttt{Fib}(k)$

if $k = 0$ **then**
    return 0
end if
if $k = 1$ **then**
    return 1
else if **then**
    return $\texttt{Fib}(k-1) + \texttt{Fib}(k-2)$.
end if

Running time: $T(k) = T(k-1) + T(k-2) + 1 =? = O(2^k)$.

Fibonacci numbers

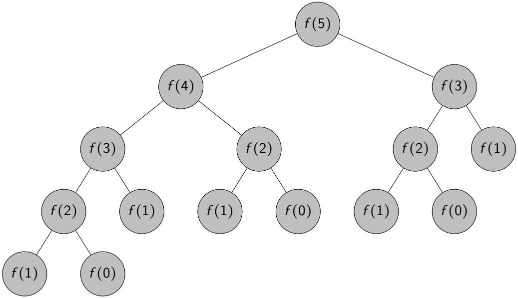
Computing Fibonacci numbers

Input: Given 1^k

Output: kth Fibonacci number.

Recurrence $f(k) = f(k-1) + f(k-2)$.

Example: $f(5)$.



Fibonacci numbers

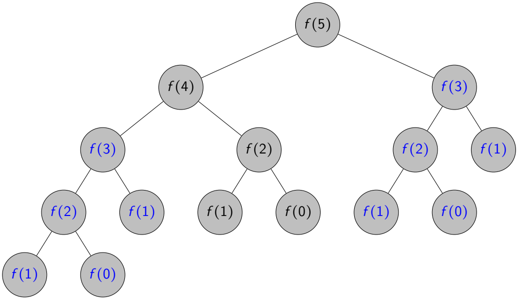
Computing Fibonacci numbers

Input: Given 1^k

Output: kth Fibonacci number.

Recurrence $f(k) = f(k-1) + f(k-2)$.

Example: $f(5)$.



Fibonacci numbers

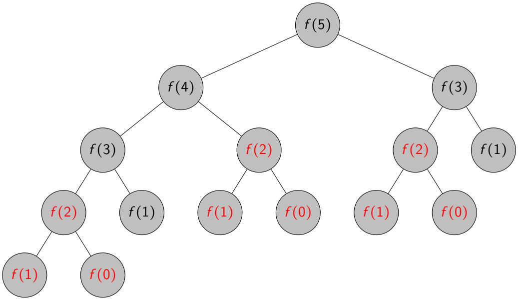
Computing Fibonacci numbers

Input: Given 1^k

Output: kth Fibonacci number.

Recurrence $f(k) = f(k-1) + f(k-2)$.

Example: $f(5)$.



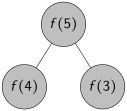
We are recomputing too much!

Those who don't remember history are condemned to repeat it.
– George Santayana

Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

```
if Table contains f(k) then
    Return the table value.
end if
if k = 0 or k = 1 then
    val ← k
else
    val ← Fib(k − 1) + Fib(k − 2)
    Store val as the kth entry in the table
    return val
end if
```

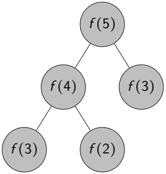


k	$f(k)$
0	
1	
2	
3	
4	
5	

Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

if Table contains $f(k)$ **then**
    Return the table value.
end if
if $k = 0$ or $k = 1$ **then**
    val $\leftarrow k$
else
    val $\leftarrow \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$
    Store val as the kth entry in the table
    return val
end if

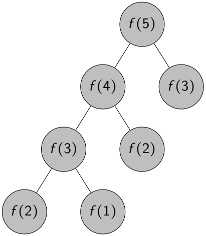


k	$f(k)$
0	
1	
2	
3	
4	
5	

Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

if Table contains $f(k)$ **then**
  Return the table value.
end if
if $k = 0$ or $k = 1$ **then**
  val $\leftarrow k$
else
  val $\leftarrow \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$
  Store val as the kth entry in the table
  return val
end if



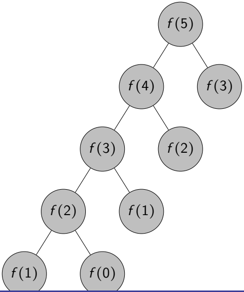
k	$f(k)$
0	
1	1
2	
3	
4	
5	

Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

if Table contains $f(k)$ **then**
  Return the table value.
end if
if $k = 0$ or $k = 1$ **then**
  val $\leftarrow k$
else
  val $\leftarrow \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$
  Store val as the kth entry in the table
  return val
end if

k	$f(k)$
0	0
1	1
2	
3	
4	
5	

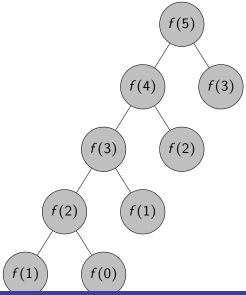


Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

```
if Table contains f(k) then
    Return the table value.
end if
if k = 0 or k = 1 then
    val ← k
else
    val ← Fib(k − 1) + Fib(k − 2)
    Store val as the kth entry in the table
    return val
end if
```

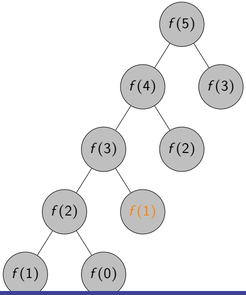
k	$f(k)$
0	0
1	1
2	1
3	
4	
5	



Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

if Table contains $f(k)$ **then**
  Return the table value.
end if
if $k = 0$ or $k = 1$ **then**
  val $\leftarrow k$
else
  val $\leftarrow \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$
  Store val as the kth entry in the table
  return val
end if



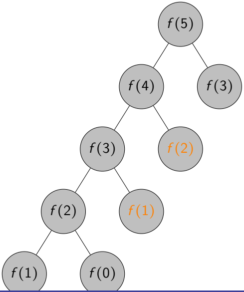
k	$f(k)$
0	0
1	1
2	1
3	2
4	
5	

Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

```
if Table contains f(k) then
    Return the table value.
end if
if k = 0 or k = 1 then
    val ← k
else
    val ← Fib(k − 1) + Fib(k − 2)
    Store val as the kth entry in the table
    return val
end if
```

k	$f(k)$
0	0
1	1
2	1
3	2
4	
5	

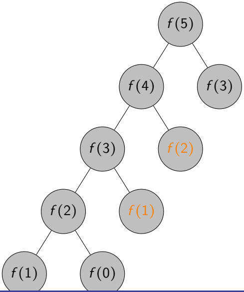


Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

if Table contains $f(k)$ **then**
    Return the table value.
end if
if $k = 0$ or $k = 1$ **then**
    val $\leftarrow k$
else
    val $\leftarrow \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$
    Store val as the kth entry in the table
    return val
end if

k	$f(k)$
0	0
1	1
2	1
3	2
4	3
5	

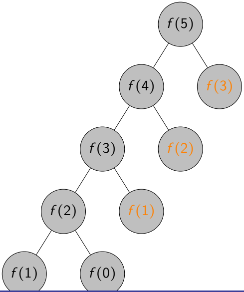


Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

```
if Table contains f(k) then
    Return the table value.
end if
if k = 0 or k = 1 then
    val ← k
else
    val ← Fib(k − 1) + Fib(k − 2)
    Store val as the kth entry in the table
    return val
end if
```

k	$f(k)$
0	0
1	1
2	1
3	2
4	3
5	

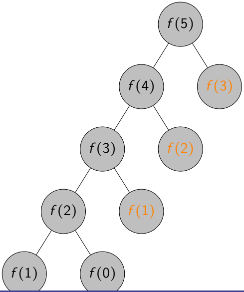


Memoization

Remember the values computed by the recursive calls for future reference. (Keep a memo.)

if Table contains $f(k)$ **then**
    Return the table value.
end if
if $k = 0$ or $k = 1$ **then**
    val $\leftarrow k$
else
    val $\leftarrow \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$
    Store val as the kth entry in the table
    return val
end if

k	$f(k)$
0	0
1	1
2	1
3	2
4	3
5	5



Running time analysis

if Table contains $f(k)$ **then**
  Return the table value.
end if
if $k = 0$ or $k = 1$ **then**
  val $\leftarrow k$
else
  val $\leftarrow \texttt{Fib}(k-1) + \texttt{Fib}(k-2)$
  Store val as the kth entry in the table
  return val
end if

The table gets filled from smaller values to larger values.

The recursion for each index i is evaluated at most once.

Look up into the table takes $O(1)$ time.

Overall running time $O(k)$.

Weighted Interval Scheduling

Problem Description

Input: For each $i \in [n]$, start and finish times $(s(i), f(i))$
and a weight for each interval $w(i) \in \mathbb{N}$

Output: Maximum weight non-conflicting jobs

Seen this problem in Problem Sheet 1.

Does not have good greedy heuristic.

What about a recursion-like algorithm for it?

Can we design that?

Weighted Interval Scheduling

Problem Description

Input:	For each $i \in [n]$, start and finish times $(s(i), f(i))$ and a weight for each interval $w(i) \in \mathbb{N}$
Output:	Maximum weight non-conflicting jobs

A possible recursive strategy

Let us sort the jobs based on their finish time. $f(1) \leq f(2) \ldots \leq f(n)$.

Let $p(i)$ be j if j is the last job before ; (in the above ordering) that is non-conflicting with i.

If the nth job belongs to OPT

for all $i > p(n)$, ith job does not belong to OPT.
Recurse on $\{1, \ldots, p(n)\}$ jobs.

If the nth job does not belong to OPT

recurse on $\{1, \ldots, n-1\}$ jobs.

Weighted Interval Scheduling

Problem Description

Input: For each $i \in [n]$, start and finish times $(s(i), f(i))$ and a weight for each interval $w(i) \in \mathbb{N}$

Output: Maximum weight non-conflicting jobs

Assume that we have sorted the jobs in non-decreasing order of their finish times.

We have also computed $p(i)$ for each $i \in [n]$.

`WtIntSc`(i)

if $i = 0$ **then**
  return i
else if Table(i) is non-empty **then**
  return Table(i)
else
  Table$(i) \leftarrow \max\{w(i) + \texttt{WtIntSc(p(i)), WtIntSc(i-1)}\}$
end if

Correctness of the recursive algorithm

Lemma

For every $i \in [n]$, `WtIntSc(i)` computes the the optimal solution for the sub-problem $\{1, \ldots i\}$, where the jobs are ordered according to their finish times.

Let $\mathtt{Opt}(i)$ denote the value of the optimal solution for the sub-problem $\{0, \ldots, i\}$.

Trivially $\mathtt{Opt}(0) = 0$. Assume that the induction hypothesis holds for $\forall j < i$.

Proof

Due to this, we know that $\mathtt{WtIntSc}(p(i)) = \mathtt{Opt}(p(i))$ and $\mathtt{WtIntSc}(i-1) = \mathtt{Opt}(i-1)$.

From our previous reasoning, the correct answer for the subproblem $\{1, \ldots, i\}$ is max of $w(i) + \mathtt{Opt}(p(i))$ and $\mathtt{Opt}(i-1)$.

Therefore the algorithm computes it correctly.

Running time analysis

Running time analysis of the dynamic programming algorithm.

The sorting of the jobs according to their finish time takes time $O(n \log n)$.

Assuming writing the values into the table (array) and reading from it takes time $O(1)$the time taken by the algorithm is $O(\text{the number of calls made to } \texttt{WtIntSc}) = O(n)$.

So the overall time taken is $O(n \log n)$.