13.1.2
Automatic/implicit memoization

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure)

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure)

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure)

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)

Automatic implicit memoization

Initialize a (dynamic) dictionary data structure D to empty

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (n is already in D)
            return value stored with n in D
        val ⇐ Fib(n − 1) + Fib(n − 2)
        Store (n, val) in D
        return val
```

Use hash-table or a map to remember which values were already computed.

Explicit memoization (not automatic)

1. Initialize table/array M of size n: $M[i] = -1$ for $i = 0, \ldots, n$.
2. Resulting code:

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (M[n] ≠ −1) // M[n]: stored value of Fib(n)
        return M[n]
    M[n] ⇐ Fib(n − 1) + Fib(n − 2)
    return M[n]
```

3. Need to know upfront the number of subproblems to allocate memory.

Explicit memoization (not automatic)

1. Initialize table/array M of size n: $M[i] = -1$ for $i = 0, \ldots, n$.
2. Resulting code:

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (M[n] ≠ -1) // M[n]: stored value of Fib(n)
            return M[n]
        M[n] ⇐ Fib(n - 1) + Fib(n - 2)
        return M[n]
```

3. Need to know upfront the number of subproblems to allocate memory.

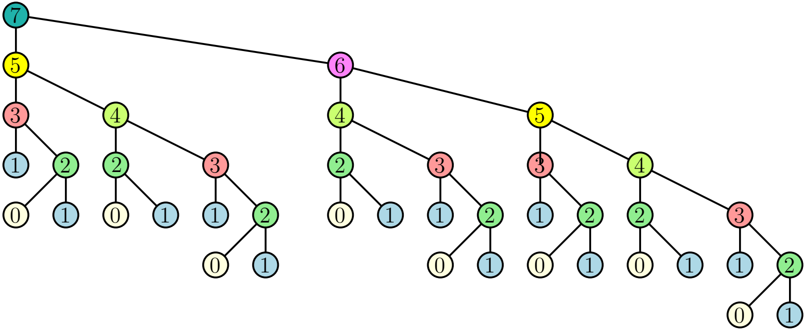
Explicit memoization (not automatic)

1. Initialize table/array M of size n: $M[i] = -1$ for $i = 0, \ldots, n$.
2. Resulting code:

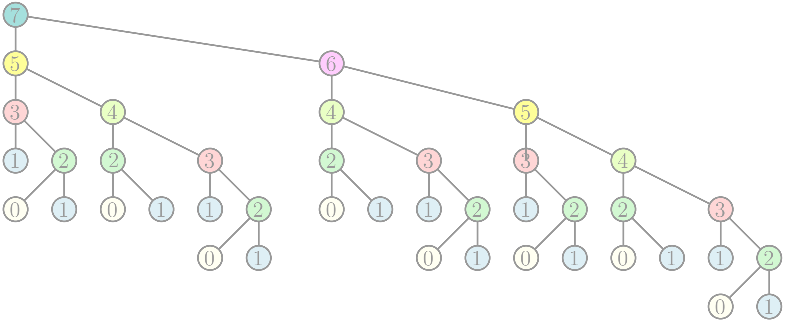
```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (M[n] ≠ -1) // M[n]: stored value of Fib(n)
            return M[n]
        M[n] ⇐ Fib(n − 1) + Fib(n − 2)
        return M[n]
```

3. Need to know upfront the number of subproblems to allocate memory.

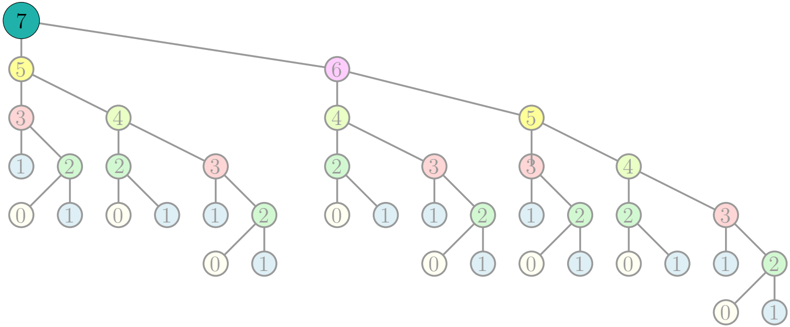
Recursion tree for the memoized Fib...



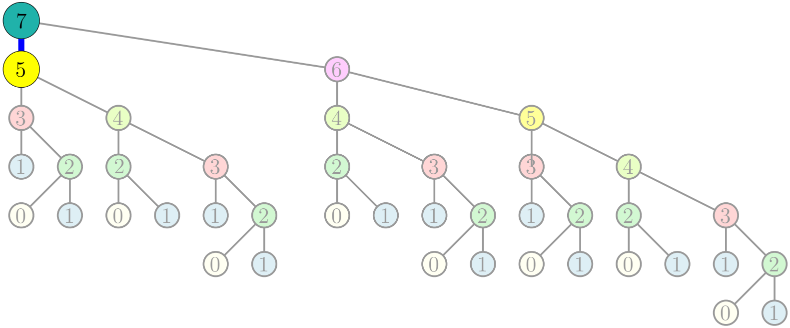
Recursion tree for the memoized Fib...



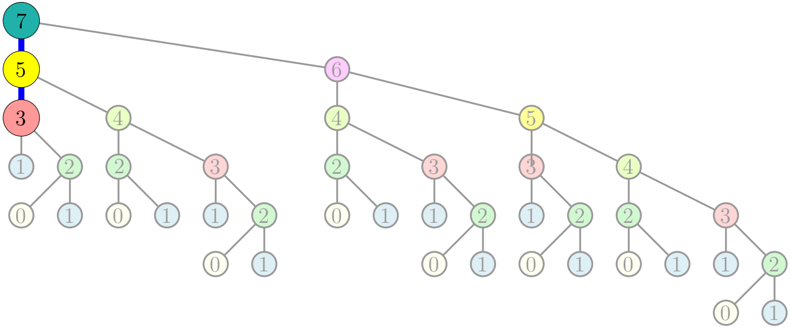
Recursion tree for the memoized Fib...



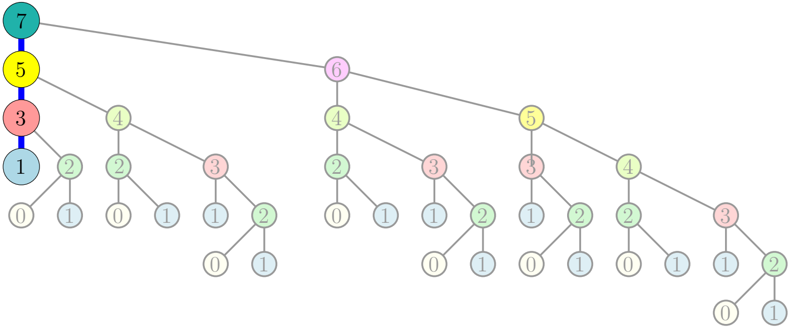
Recursion tree for the memoized Fib...



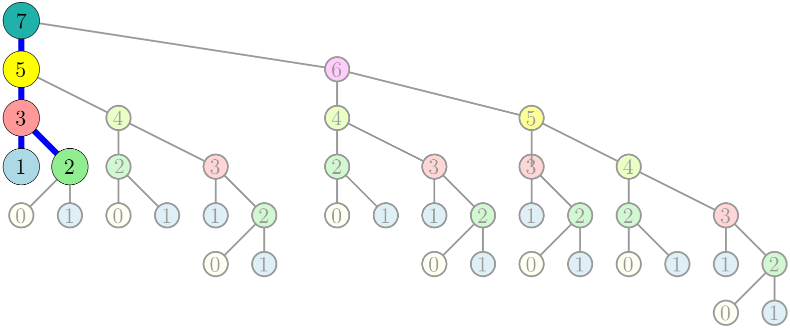
Recursion tree for the memoized Fib...



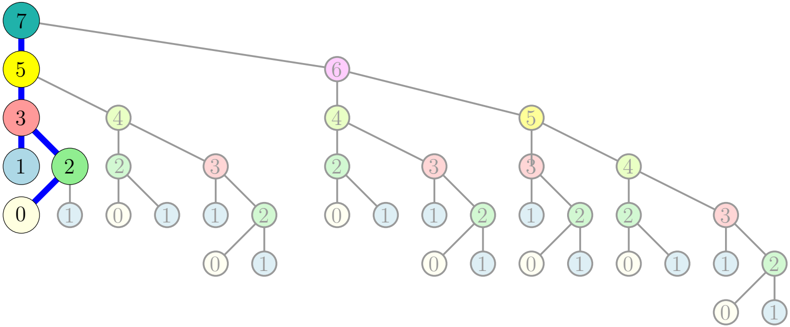
Recursion tree for the memoized Fib...



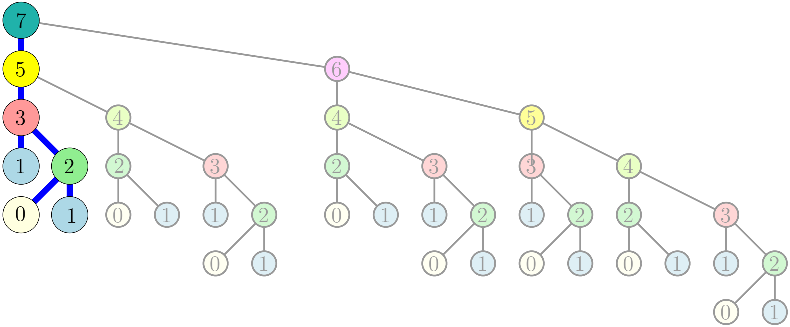
Recursion tree for the memoized Fib...



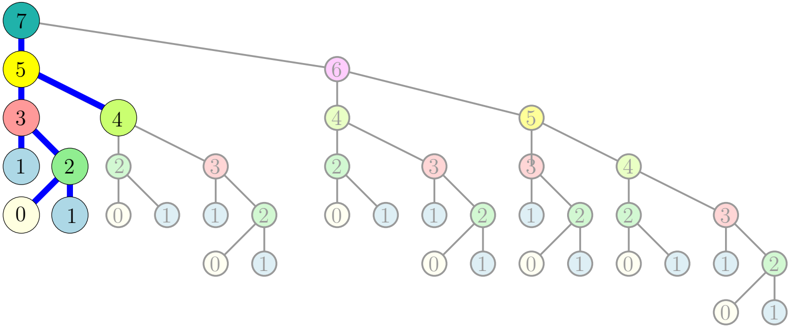
Recursion tree for the memoized Fib...



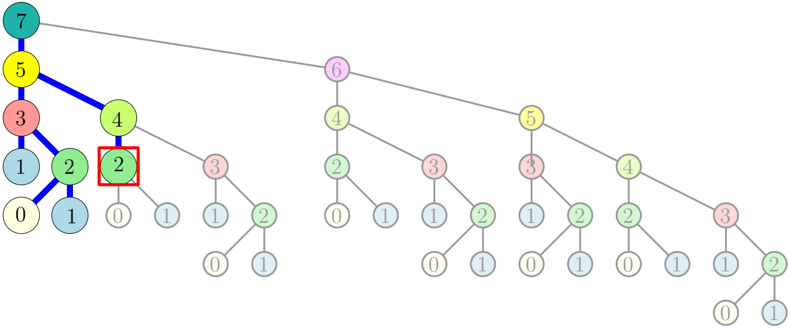
Recursion tree for the memoized Fib...



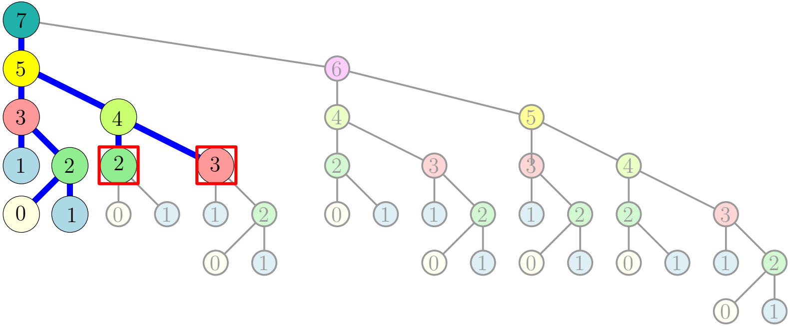
Recursion tree for the memoized Fib...



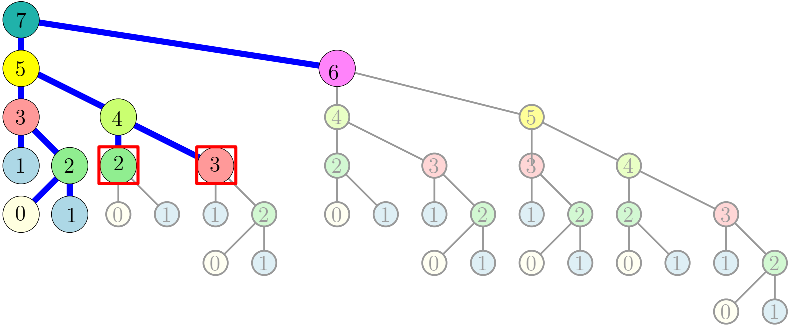
Recursion tree for the memoized Fib...



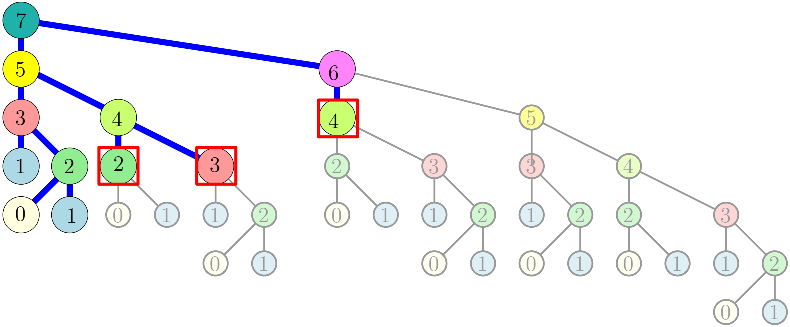
Recursion tree for the memoized Fib...



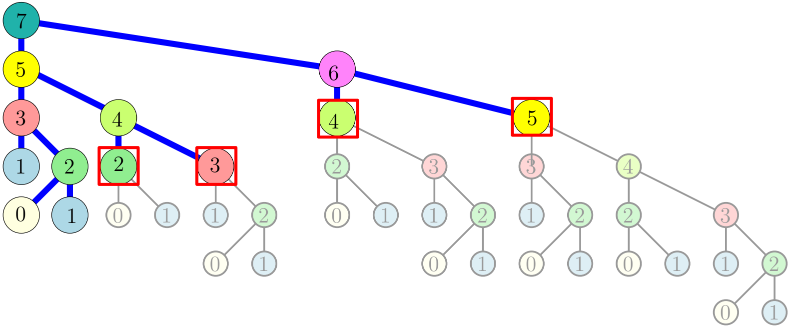
Recursion tree for the memoized Fib...



Recursion tree for the memoized Fib...



Recursion tree for the memoized Fib...



Automatic Memoization

1. Recursive version:

   $f(x_1, x_2, \ldots, x_d)$:
   CODE

2. Recursive version with memoization:

   $g(x_1, x_2, \ldots, x_d)$:
   if f already computed for $(x_1, x_2, \ldots, x_d)$ **then**
   return value already computed
   NEW_CODE

3. NEW_CODE:
   1. Replaces any "**return** α" with
   2. Remember "$f(x_1, \ldots, x_d) = \alpha$"; **return** α.

Automatic Memoization

1. Recursive version:

```
f(x_1, x_2, ..., x_d):
        CODE
```

2. Recursive version with memoization:

```
g(x_1, x_2, ..., x_d):
        if f already computed for (x_1, x_2, ..., x_d) then
            return value already computed
        NEW_CODE
```

3. NEW_CODE:
   1. Replaces any "**return** α" with
   2. Remember "$f(x_1, \ldots, x_d) = \alpha$"; **return** α.

Automatic Memoization

1. Recursive version:

```
f(x_1, x_2, ..., x_d):
        CODE
```

2. Recursive version with memoization:

```
g(x_1, x_2, ..., x_d):
        if f already computed for (x_1, x_2, ..., x_d) then
            return value already computed
        NEW_CODE
```

3. NEW_CODE:
   1. Replaces any "**return** α" with
   2. Remember "$f(x_1, \ldots, x_d) = \alpha$"; **return** α.

Explicit vs Implicit Memoization

1. Explicit memoization (on the way to iterative algorithm) preferred:
   1. analyze problem ahead of time
   2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
   1. problem structure or algorithm is not well understood.
   2. Need to pay overhead of data-structure.
   3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (on the way to iterative algorithm) preferred:
    1. analyze problem ahead of time
    2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
    1. problem structure or algorithm is not well understood.
    2. Need to pay overhead of data-structure.
    3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (on the way to iterative algorithm) preferred:
    1. analyze problem ahead of time
    2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
    1. problem structure or algorithm is not well understood.
    2. Need to pay overhead of data-structure.
    3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (on the way to iterative algorithm) preferred:
    1. analyze problem ahead of time
    2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
    1. problem structure or algorithm is not well understood.
    2. Need to pay overhead of data-structure.
    3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (on the way to iterative algorithm) preferred:
    1. analyze problem ahead of time
    2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
    1. problem structure or algorithm is not well understood.
    2. Need to pay overhead of data-structure.
    3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit/implicit memoization for Fibonacci

```
Init:   M[i] = −1, i = 0,...,n.

Fib(k):
    if (k = 0)
        return 0
    if (k = 1)
        return 1
    if (M[k] ≠ −1)
        return M[n]
    M[k] ⇐ Fib(k − 1) + Fib(k − 2)
    return M[k]
```

Explicit memoization

```
Init:   Init dictionary D

Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (n is already in D)
        return value stored with n in D
        val ⇐ Fib(n − 1) + Fib(n − 2)
    Store (n, val) in D
    return val
```

Implicit memoization

How many distinct calls?

```
binom(t, b)    // computes (t choose b)
    if t = 0 then return 0
    if b = t or b = 0 then return 1
    return binom(t − 1, b − 1) + binom(t − 1, b).
```

How many distinct calls does **binom**$(n, \lfloor n/2 \rfloor)$ makes during its recursive execution?

- $\Theta(1)$.
- $\Theta(n)$.
- $\Theta(n \log n)$.
- $\Theta(n^2)$.
- $\Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right)$.

That is, if the algorithm calls recursively **binom**$(17, 5)$ about 5000 times during the computation, we count this is a single distinct call.

Running time of memoized binom?

```
D: Initially an empty dictionary.
binomM(t, b)    // computes (t choose b)
    if b = t then return 1
    if b = 0 then return 0
    if D[t, b] is defined then return D[t, b]
    D[t, b] ⇐ binomM(t − 1, b − 1) + binomM(t − 1, b).
    return D[t, b]
```

Assuming that every arithmetic operation takes $O(1)$ time, What is the running time of **binomM**$(n, \lfloor n/2 \rfloor)$?

- $\Theta(1)$.
- $\Theta(n)$.
- $\Theta(n^2)$.
- $\Theta(n^3)$.
- $\Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right)$.

THE END

...

(for now)