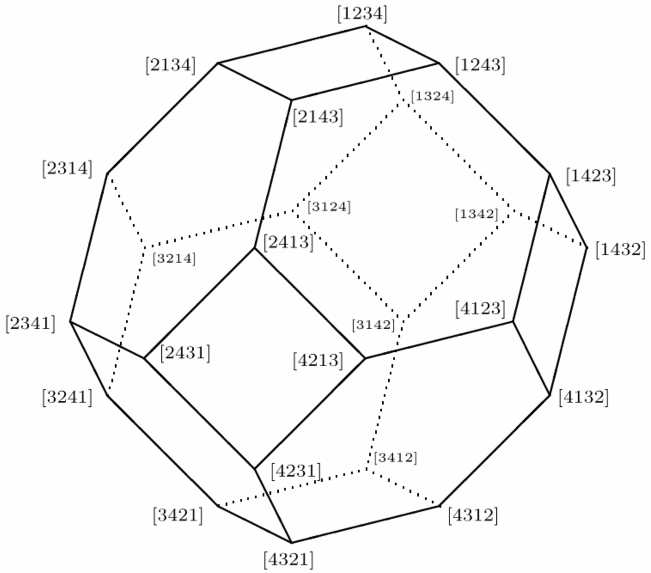
CS3334 Data Structures

Lecture 2: Recursive Functions & Binary Search



Chee Wei Tan

What is Recursive Function?

- A recursive function is a function that calls itself (i.e., a recursive call).
- A recursive function has base case(s), i.e., no more recursive calls.
- Given an input size *n*, recursive calls reduce *n* progressively until *n* reaches a base case.

Example: Fast Exponentiation (1/2)

- Compute x^n, given x and integer $n \geq 0$
- Assume multiplication between 2 values requires 1 step
- Straightforward implementation requires O(n) time
  - $x^8 = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$
  - 7 multiplications, O(n-1) = O(n)
- However, a recursive function can do it in O(log$n$) time
  - It can compute x^8 in 3 multiplications.
  - $(x) \cdot (x) = x^2$

    $(x \cdot x) \cdot (x \cdot x) = x^4$

    $(x \cdot x \cdot x \cdot x) \cdot (x \cdot x \cdot x \cdot x) = x^8$

Example: Fast Exponentiation (2/2)

```
double power(double x, int n)
{
   if (n==0) return 1; //base case
   if (n==1) return x; //base case
   if (n%2==0) //n is even
      return power(x*x,n/2); //recursive call
  else //n is odd
      return power(x*x,n/2)*x; //recursive call
/* n/2 is integer division */
}
```

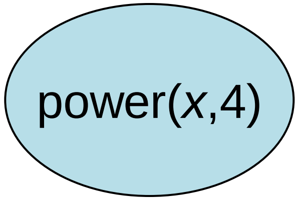
Tree of Recursive Calls

- Drawing the tree of recursive calls is a technique to trace the execution of a recursive function
- Try to trace the call power(x,4) which computes x^4 in the example, the tree of recursive calls is drawn in the next few slides

Recursive Calls of power(*x*,4) (1/5)

- The root node represents the initial call, i.e., power(*x*, 4)

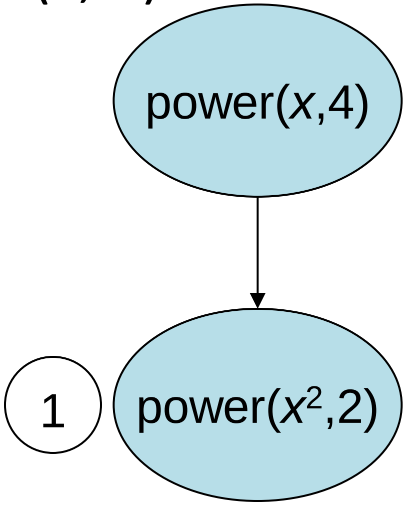
```
double power(double x, int n)
{
   if (n==0) return 1; //base case
   if (n==1) return x; //base case
   if (n%2==0) //n is even
      return power(x*x,n/2);
   else //n is odd
      return power(x*x,n/2)*x;
}
```



Recursive Calls of power(x,4) (2/5)

- Since 4 is even (i.e., $n > 1$), the function makes 1st recursive call power(x^2, 4/2=2). So, we draw a node representing power(x^2, 2) and link it as a child of power(x, 4).

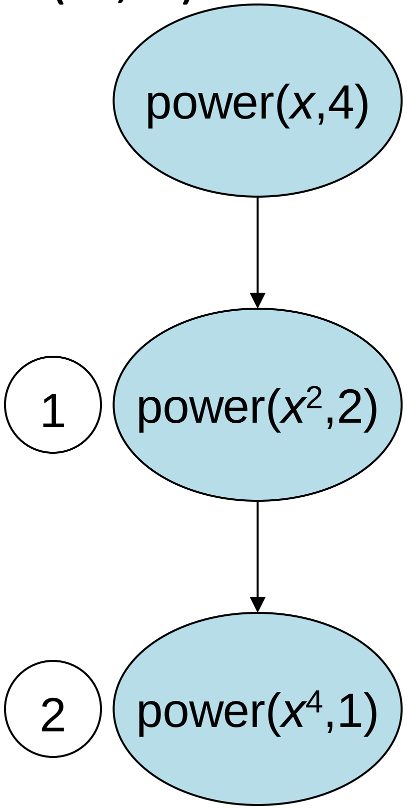
```
double power(double x, int n)
{
   if (n==0) return 1; //base case
   if (n==1) return x; //base case
   if (n%2==0) //n is even
      return power(x*x,n/2);
   else //n is odd
      return power(x*x,n/2)*x;
}
```



Recursive Calls of power(*x*,4) (3/5)

- Since 2 is even (i.e., *n* > 1), the function makes 2nd recursive call power($x^2x^2=x^4$, 2/2=1). So, we draw a node representing power(x^4, 1), and link it as a child of power(x^2, 2).

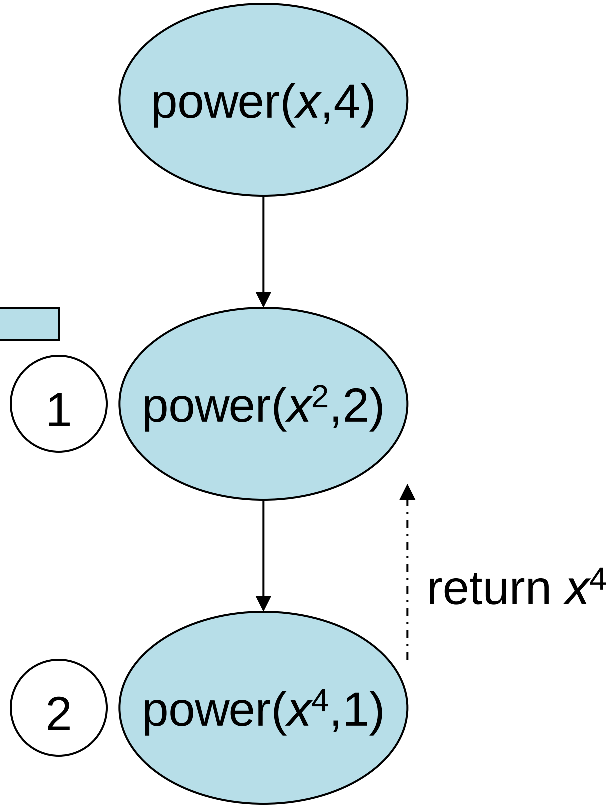
```
double power(double x, int n)
{
   if (n==0) return 1; //base case
   if (n==1) return x; //base case
   if (n%2==0) //n is even
      return power(x*x,n/2);
   else //n is odd
      return power(x*x,n/2)*x;
}
```



Recursive Calls of power(*x*,4) (4/5)

- Since *n*=1 (i.e., the second base case), 2nd recursive call power(x^4, 4) returns x^4 (Line 4)

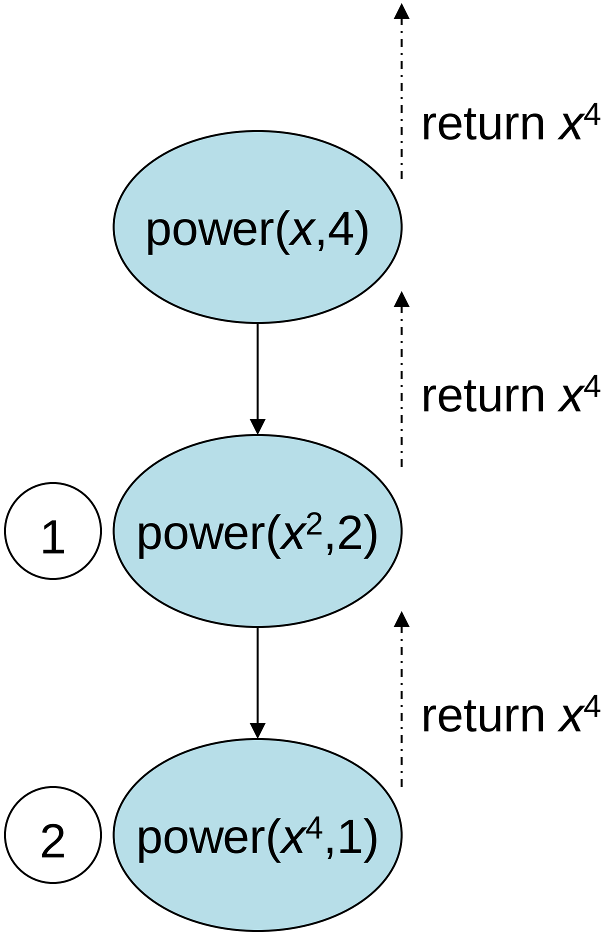
```
  double power(double x, int n)
  {
1    if (n==0) return 1; //base case
2    if (n==1) return x; //base case
3    if (n%2==0) //n is even
4  ① ② return power(x*x,n/2);
5    else //n is odd
6       return power(x*x,n/2)*x;
  }
```



Recursive Calls of power(*x*,4) (5/5)

- 1st recursive call power(x^2, 4) returns x^4 (Line 4), i.e., the initial power(x, 4) returns x^4.

```
  double power(double x, int n)
  {
1    if (n==0) return 1; //base case
2    if (n==1) return x; //base case
3    if (n%2==0) //n is even
4       return power(x*x,n/2);
5    else //n is odd
6       return power(x*x,n/2)*x;
  }
```



Worst-Case Time Complexity Analysis (General Framework)

- Let *T*(*n*) be the worst case time complexity of the given recursive function
- Find a recurrence formula for *T*(*n*)
- Solve the recurrence formula by unrolling the recurrence formula a few times to see the general pattern

Worst-Case Time Complexity (1/2)

- Let $T(n)$ be the worst case running time of the algorithm (where n is the problem size).
- Let c be the constant time for the local work (the running time of the local work performed in the call is at most some constant c).
- Since at most one recursive call is made, the recurrence formula is $T(n) \leq T(n/2) + c$.
- Assume $n=2^k$, where k is an integer.

Worst-Case Time Complexity (2/2)

$$
k\text{ equations}\left\{\begin{array}{r}
T(n) \le T(n/2) + c \longleftarrow \\
\left.\begin{array}{r}
T(n/2) \le T(n/4) + c \\
T(n/4) \le T(n/8) + c \\
\dots \\
+)\ \ T(2) \le T(1) + c
\end{array}\right\}
\end{array}\right.
$$

Find a recurrence formula for T(*n*)

Solve the recurrence formula by unrolling the recurrence formula a few times to see the general pattern

$$
\begin{array}{c}
\hline
T(n) \le T(1) + kc
\end{array}
$$

- Therefore, $T(n) \le T(1) + c\log_2 n = O(\log n)$ (since $n=2^k$, $k=\log_2 n$)

Worst-Case Space Complexity (1/3)

- For each recursive call, a separate copy of the variables declared in the function is created and stored in a stack. The storage is released when the recursive call finishes (e.g., a return statement is reached)
- Thus, the amount of space needed to implement a recursive function depends on the *depth* of recursion, not on the *number* of times the function is invoked.

Worst-Case Space Complexity (2/3)

- Let $S(n)$ be the worst case space requirement of the algorithm (where n is the problem size).
- Let c be the constant space for the local storage (the local storage required in the call is at most some constant c).
- Since at most one recursive call is made, the recurrence formula is $S(n) \leq S(n/2) + c$.
- Assume $n=2^k$, where k is an integer.

Worst-Case Space Complexity (3/3)

$$
\begin{array}{r}
k\text{ equations}\left\{
\begin{array}{rl}
S(n) \le S(n/2) + c & \longleftarrow \text{Find a recurrence formula for } S(n) \\
\left.\begin{array}{r}
S(n/2) \le S(n/4) + c \\
S(n/4) \le S(n/8) + c \\
\ldots \\
+)\ \ S(2) \le S(1) + c
\end{array}\right\} & \begin{array}{l}\text{Solve the recurrence formula by unrolling the} \\ \text{recurrence formula a few times to see the} \\ \text{general pattern}\end{array}
\end{array}\right. \\
\hline
S(n) \le S(1) + kc
\end{array}
$$

- Therefore, $S(n) \le S(1) + c\log_2 n = \mathrm{O}(\log n)$ (since $n=2^k$, $k=\log_2 n$)

Worst-Case Time and Space Complexity

- The tree of recursive calls can visualize the time and space complexity
- Time complexity is proportional to the number of nodes in the tree.
- Space complexity is proportional to the length of longest root-to-leaf path.

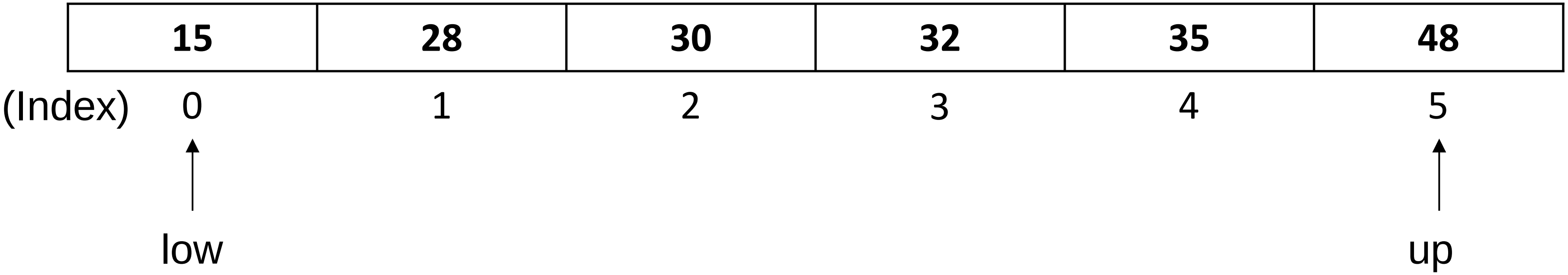
Recursive Binary Search

```
int binarySearch(int A[], int low, int up, int x)
{
   if (low>up) return -1; //cannot find x
   int mid = (low+up)/2;
   if (A[mid]==x) return mid; //find x
   else if (A[mid]<x)
      return binarySearch(A,mid+1,up,x);
   else
      return binarySearch(A,low,mid-1,x);
}
```

- Elements in A[] are sorted in increasing order.

binarySearch(A, 0, 5, 28)

Sorted Array A

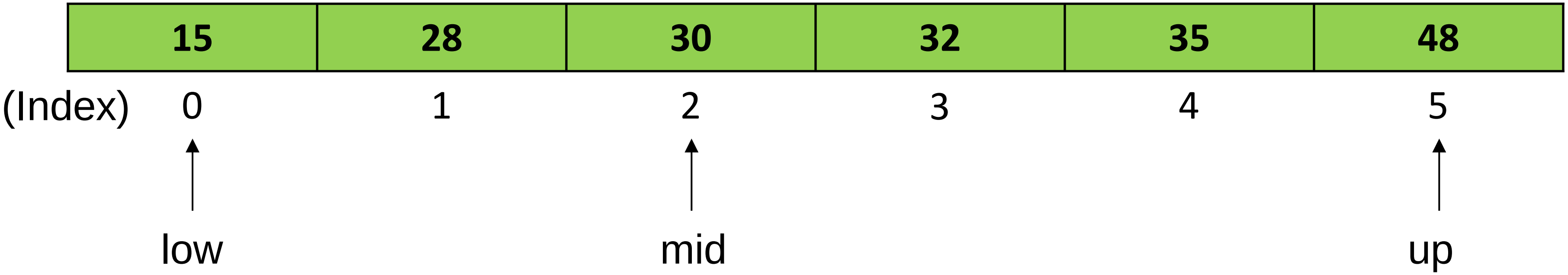


	15	**28**	**30**	**32**	**35**	**48**
(Index)	0	1	2	3	4	5

x = 28, low = 0, up = 5

binarySearch(A, 0, 5, 28)

Sorted Array A



(Index)	0	1	2	3	4	5
	15	**28**	**30**	**32**	**35**	**48**

low

mid

up

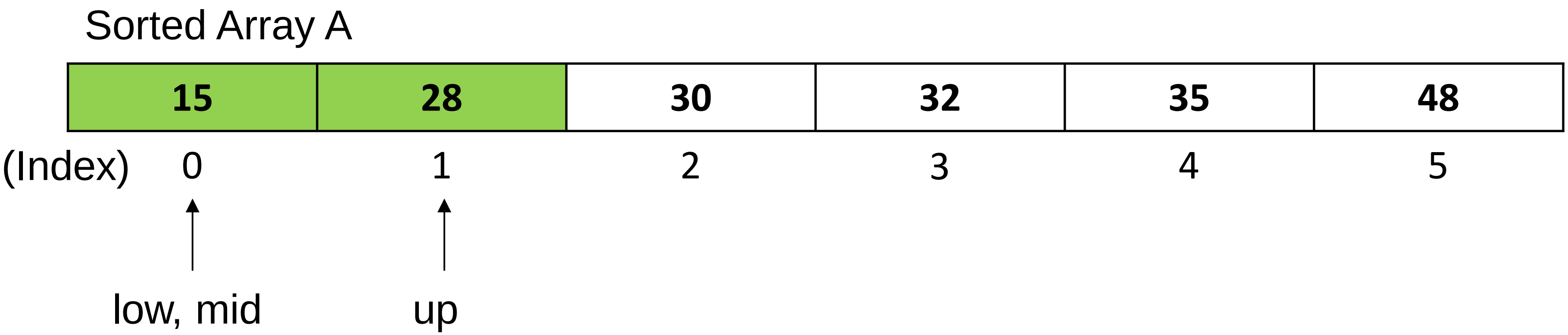
x = 28, low = 0, up = 5

1st recursion: mid = $\lfloor (0 + 5) / 2 \rfloor$ = 2

Since 28 < A[2], up = mid - 1 = 2 - 1 = 1

```
int binarySearch(int A[], int low, int up, int x)
{
   if (low>up) return -1;
   int mid = (low+up)/2;
   if (A[mid]==x) return mid;
   else if (A[mid]<x)
      return binarySearch(A,mid+1,up,x);
   else
      return binarySearch(A,low,mid-1,x);
}
```

binarySearch(A, 0, 1, 28)



Sorted Array A

(Index)	0	1	2	3	4	5
	🟩 **15**	🟩 **28**	**30**	**32**	**35**	**48**

x = 28, low = 0, up = 1

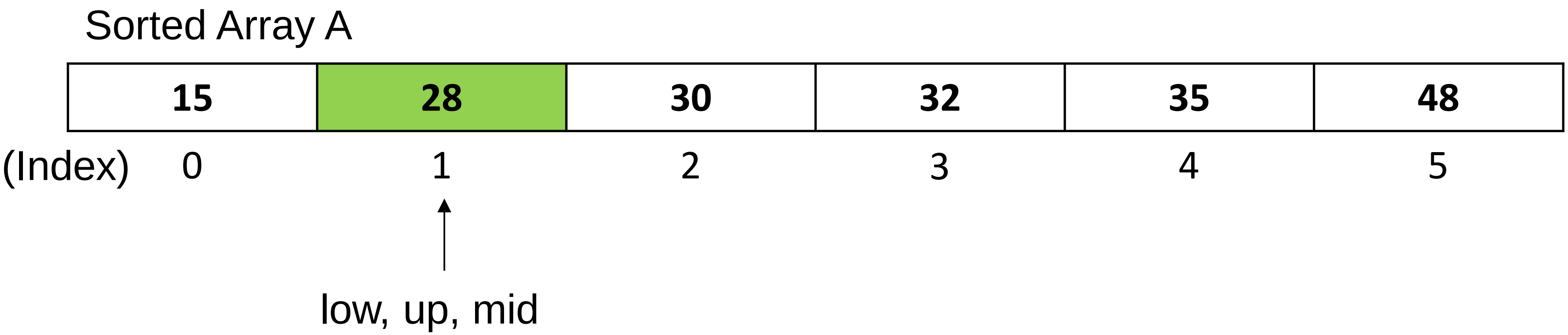
2nd recursion: mid = $\lfloor (0 + 1) / 2 \rfloor = 0$

Since 28 > A[0],

low = mid + 1 = 0 + 1 = 1

```
int binarySearch(int A[], int low, int up, int x)
{
   if (low>up) return -1; //cannot find x
   int mid = (low+up)/2;
   if (A[mid]==x) return mid; //find x
   else if (A[mid]<x)
      return binarySearch(A,mid+1,up,x);
   else
      return binarySearch(A,low,mid-1,x);
}
```

binarySearch(A, 1, 1, 28)



Sorted Array A

(Index)	0	1	2	3	4	5
	15	**28**	**30**	**32**	**35**	**48**

low, up, mid

x = 28, low = 1, up = 1

3rd recursion: mid= $\lfloor (1 + 1) / 2 \rfloor$ = 1

Since 28 = A[1], return 1

```
int binarySearch(int A[], int low, int up, int x)
{
   if (low>up) return -1; //cannot find x
   int mid = (low+up)/2;
   if (A[mid]==x) return mid; //find x
   else if (A[mid]<x)
      return binarySearch(A,mid+1,up,x);
   else
      return binarySearch(A,low,mid-1,x);
}
```

Worst-Case Time Complexity

- Let $T(n)$ be the worst case running time of the algorithm (where n is the problem size).
- Let c be the constant time for the local work (the local work performed in the call is at most some constant c).
- Since at most one recursive call is made, the recurrence formula is $T(n) \leq T(n/2) + c$.
- Assume $n=2^k$, where k is an integer.

Worst-Case Space Complexity

- Let $S(n)$ be the worst case space requirement of the algorithm (where n is the problem size).
- Let c be the constant time for the local storage (the local storage required in the call is at most some constant c).
- Since at most one recursive call is made, the recurrence formula is $S(n) \leq S(n/2) + c$.
- Assume $n=2^k$, where k is an integer.

Fibonacci Sequence

- The Fibonacci sequence is defined as:

$f_0 = 0,\qquad f_1 = 1$

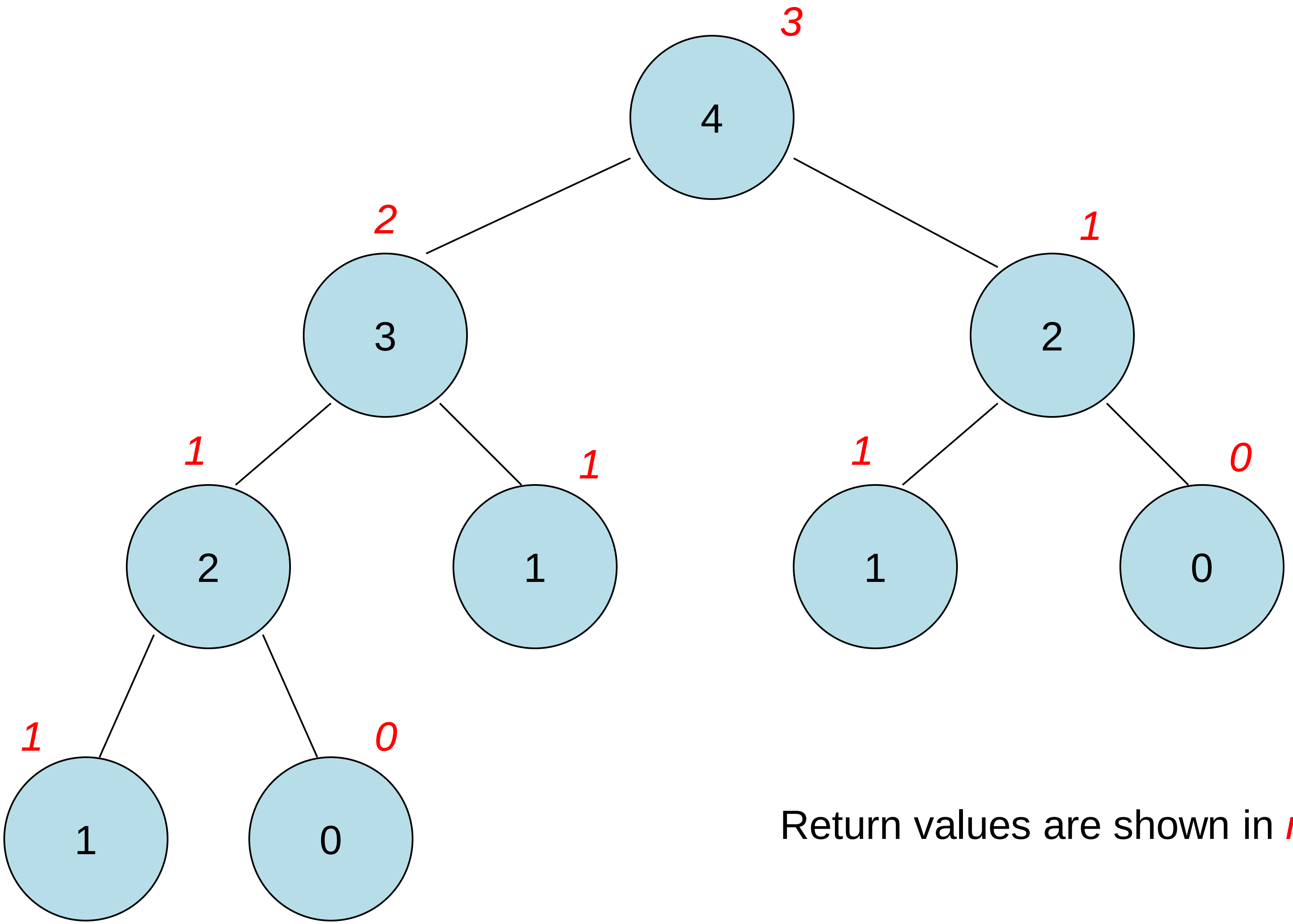
$f_i = f_{i-1} + f_{i-2}\quad \text{for } i \geq 2$

```
// to compute f_n
int fib(int n)
{
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Tracing fib(4)

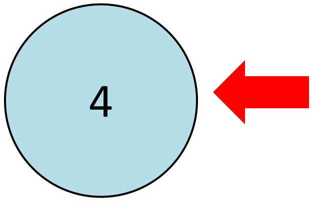
- To trace the execution for the call fib(4), the tree of recursive calls (next slide) is drawn as follows:
  - The root node represents the initial call, i.e., fib(4)
  - The call fib(4) makes two recursive calls, fib(3) and fib(2). So, we draw two nodes representing fib(3) and fib(2), and link them as children of fib(4).
  - We repeat this until all the recursive calls are drawn
  - Then, we determine the return value of each call from bottom to top

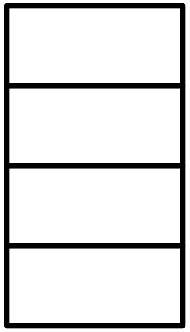
Tree of Recursive Calls for fib(4)



Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)

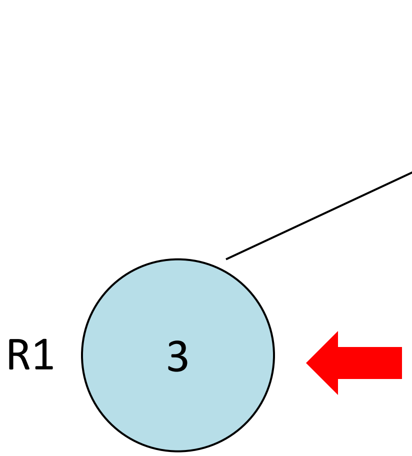


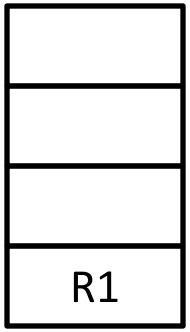


Storage for
recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Tree of Recursive Calls for fib(4)

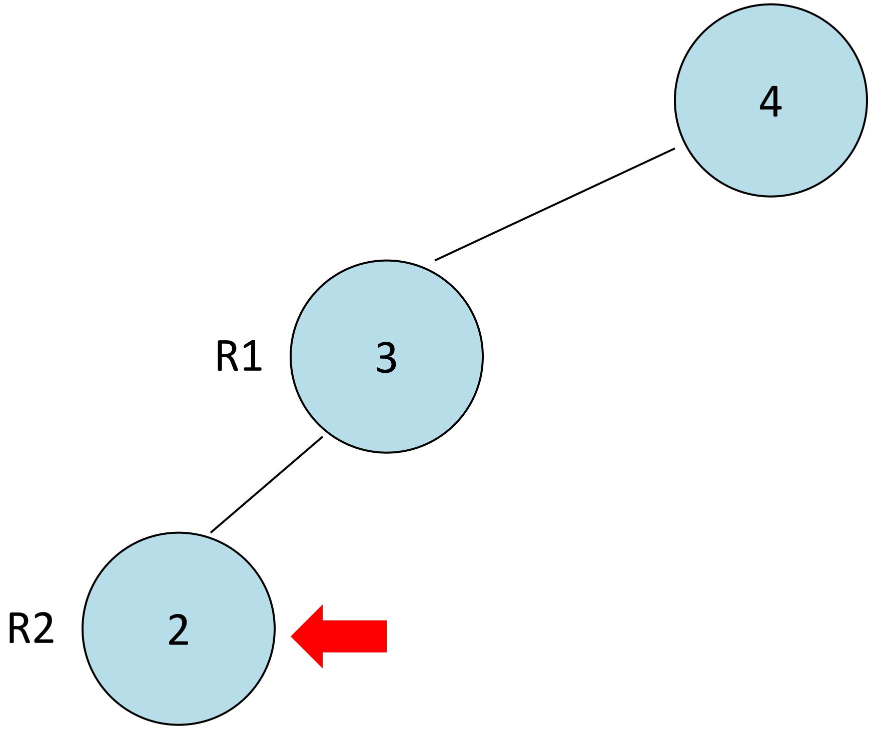


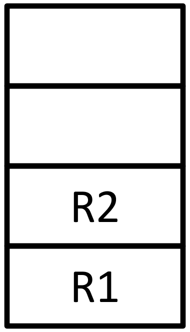


Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Tree of Recursive Calls for fib(4)

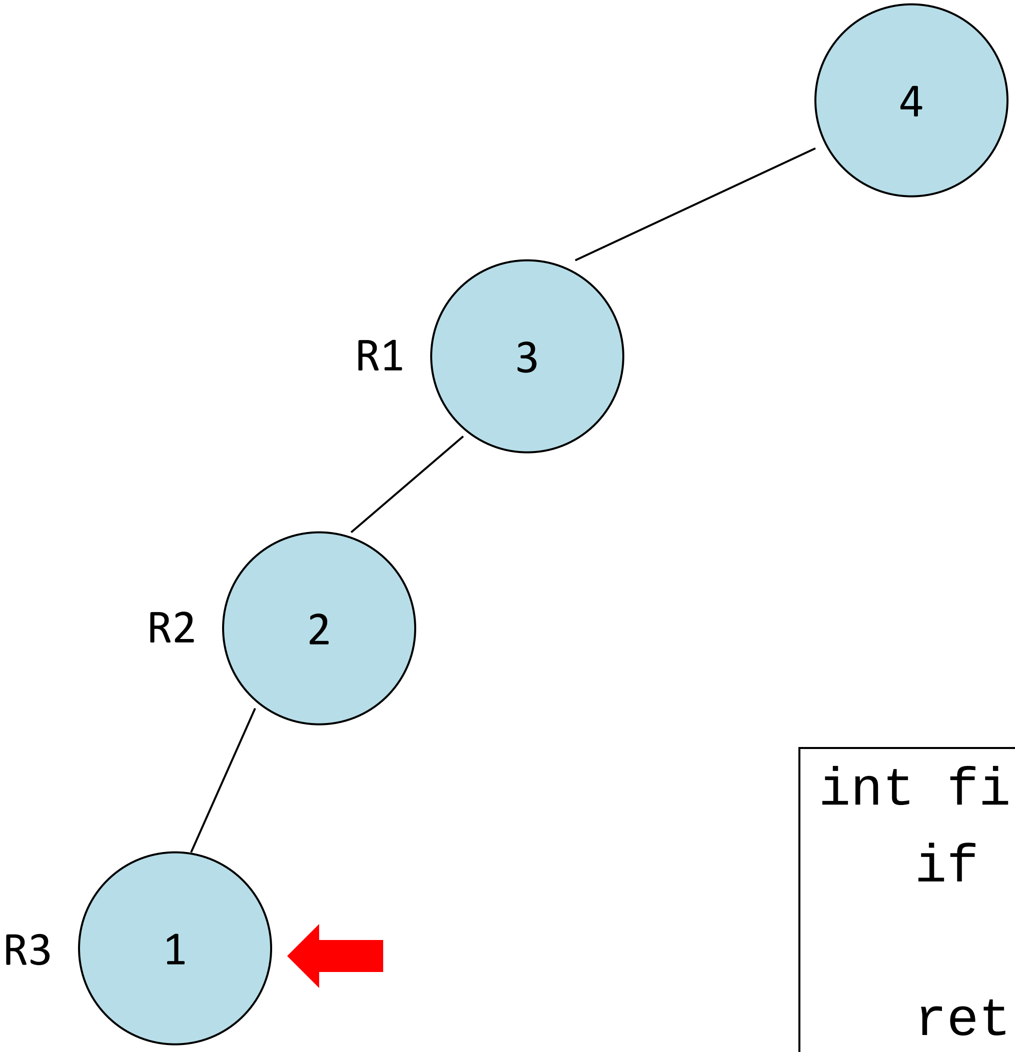


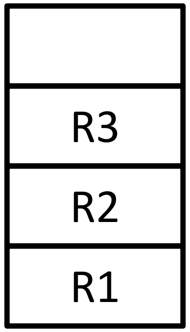


Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Tree of Recursive Calls for fib(4)

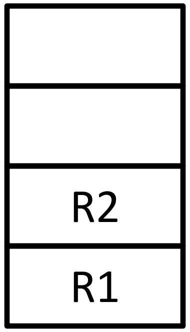




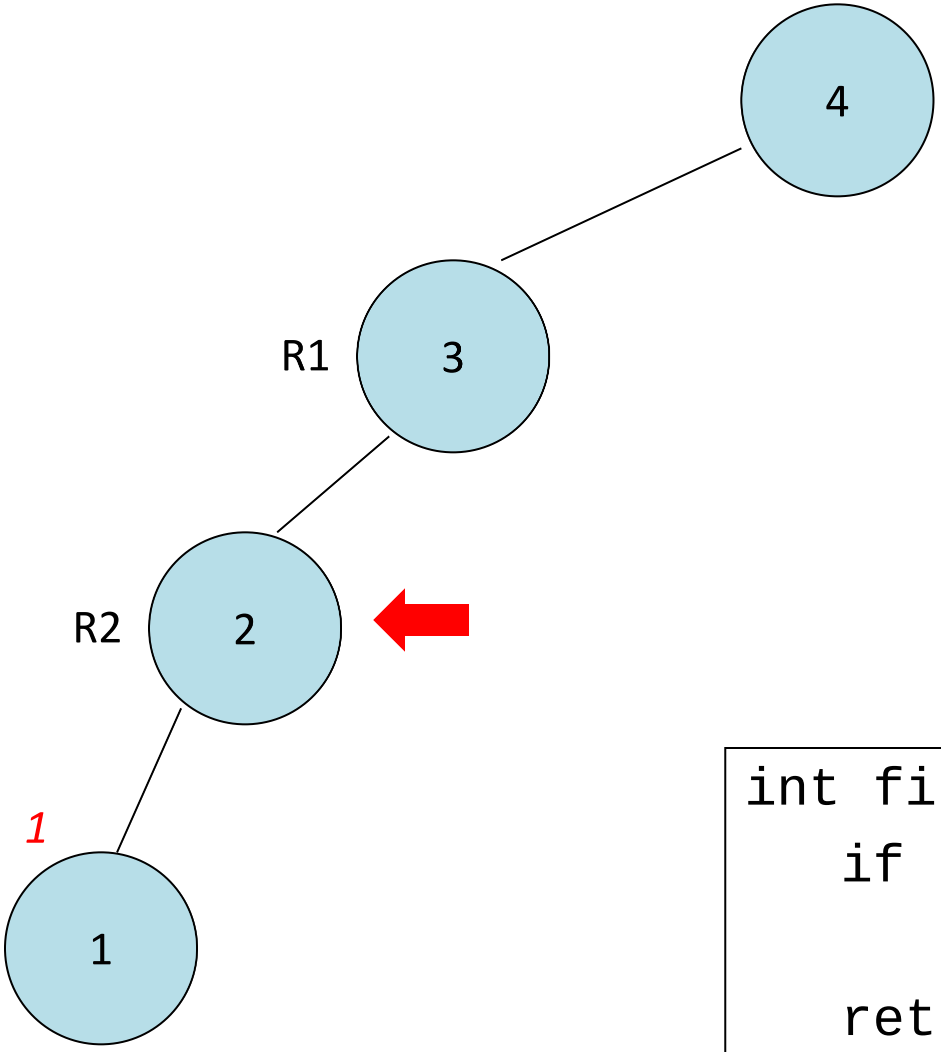
Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Tree of Recursive Calls for fib(4)



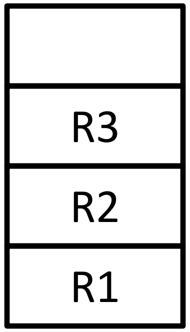
Storage for recursive calls



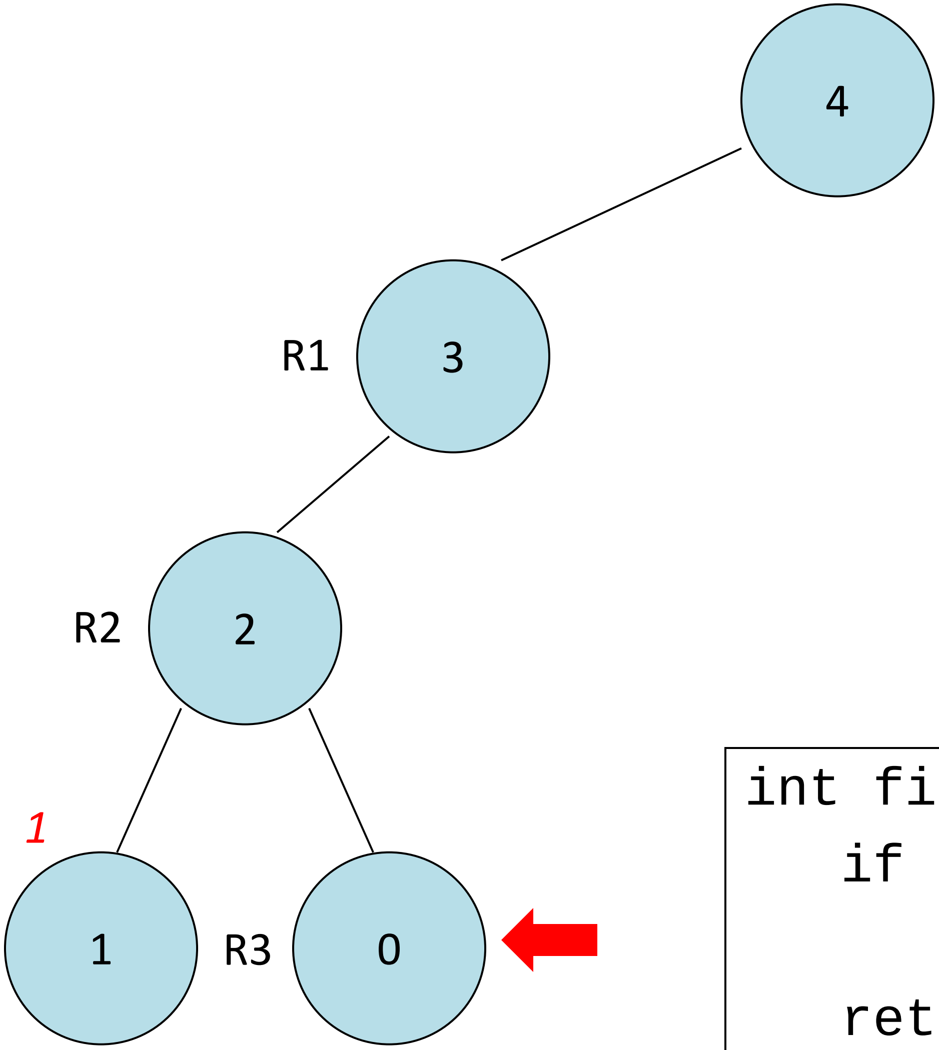
```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



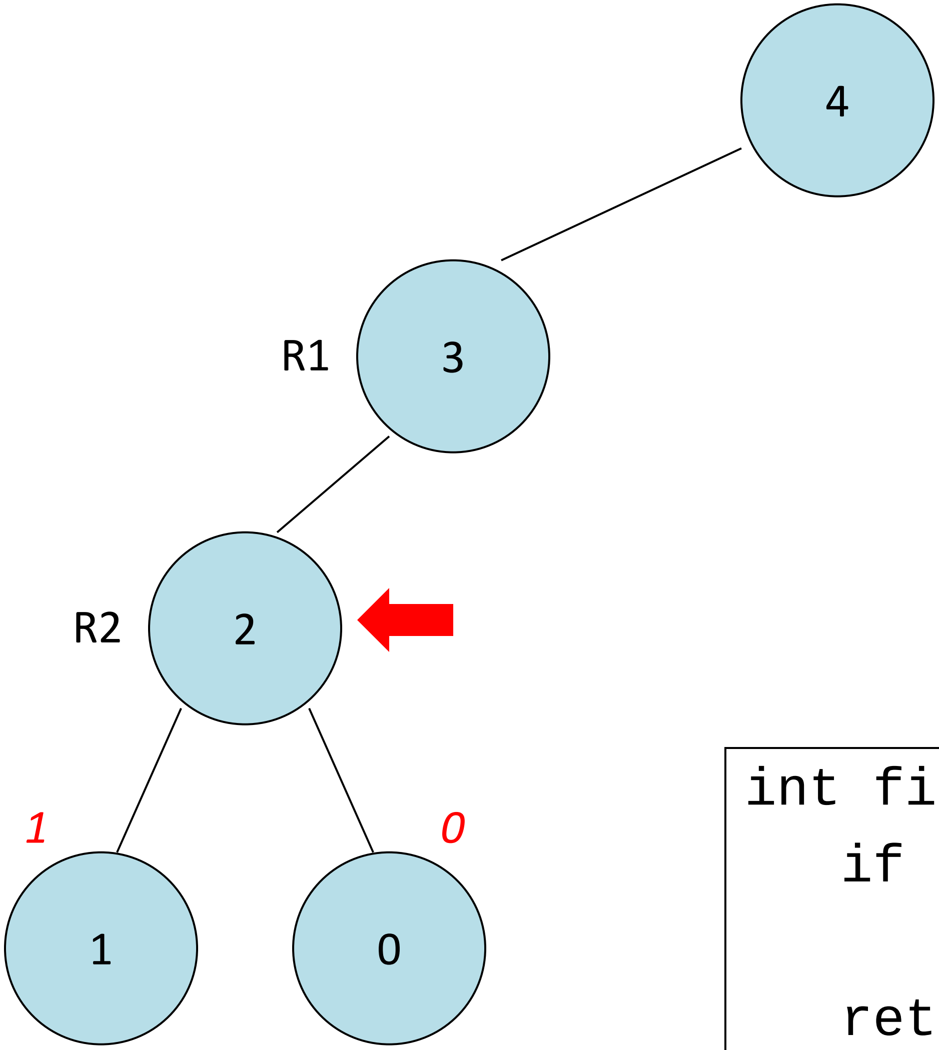
Storage for recursive calls

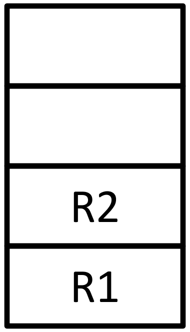


```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



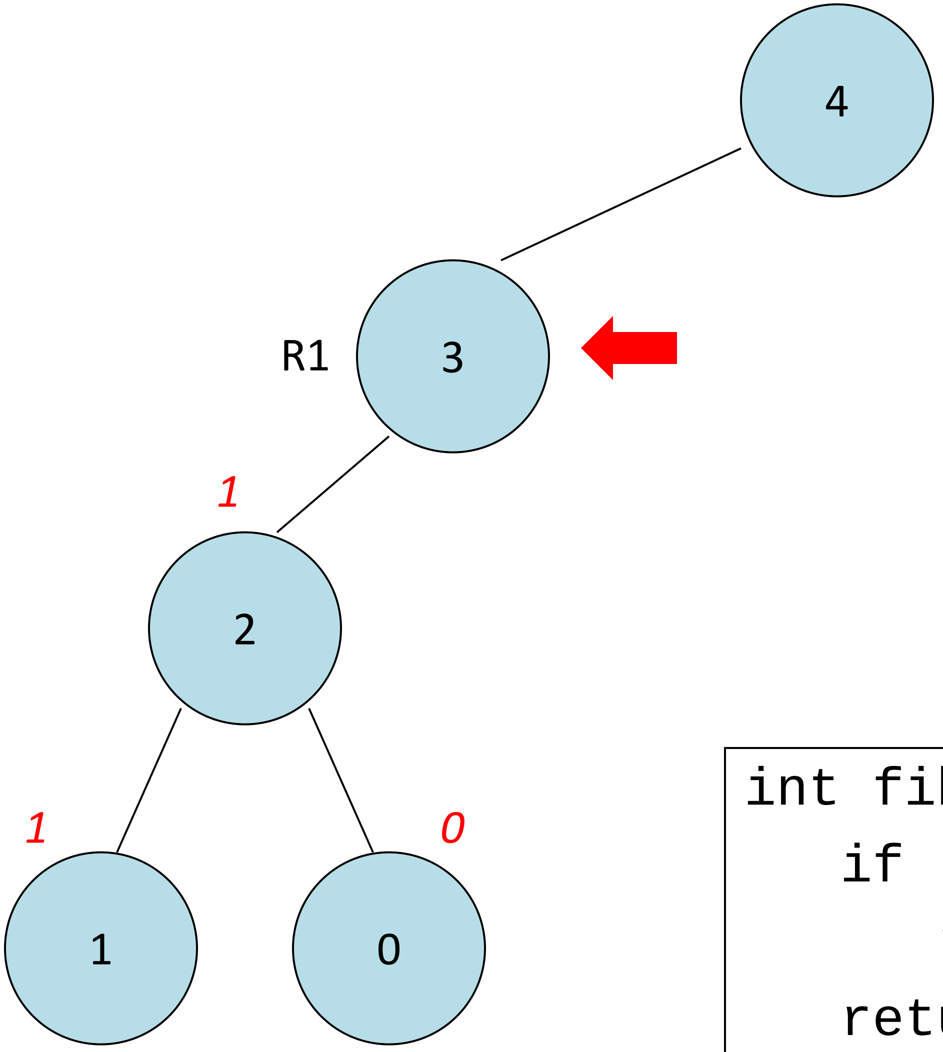


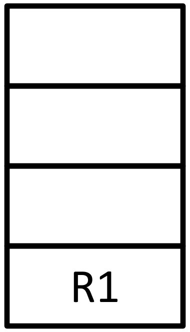
Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



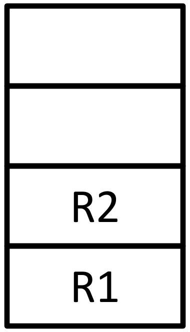


Storage for recursive calls

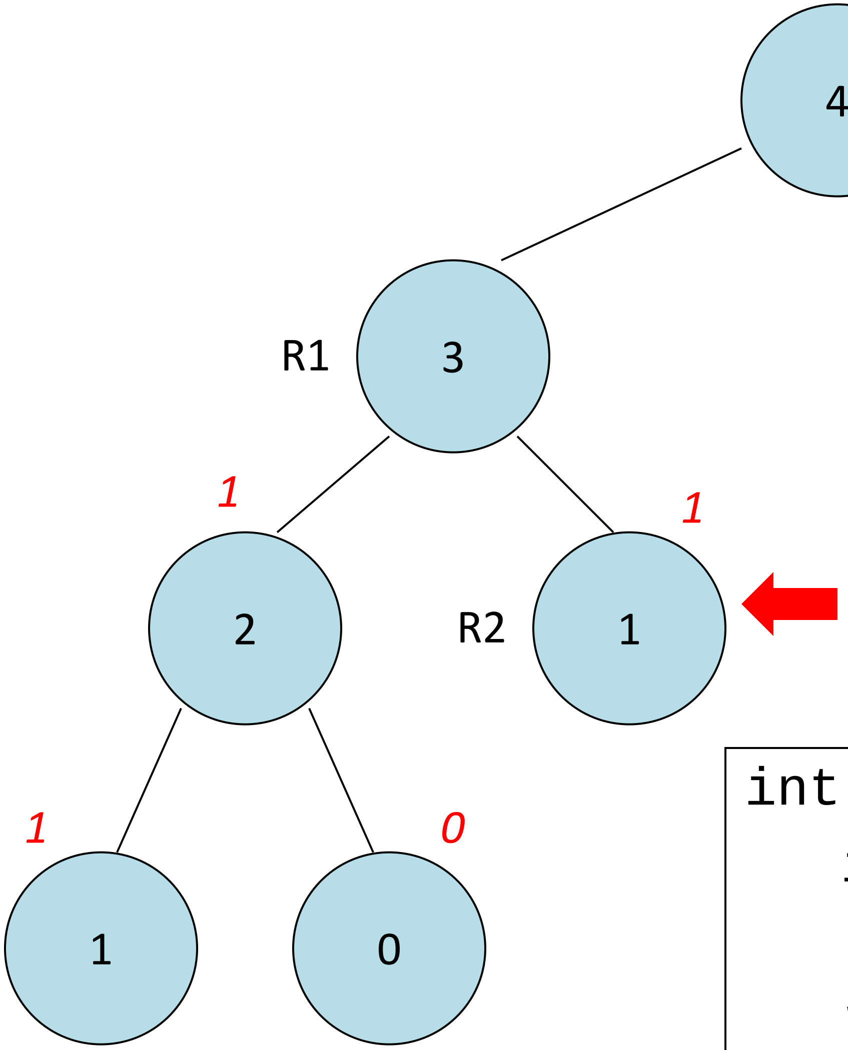
```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



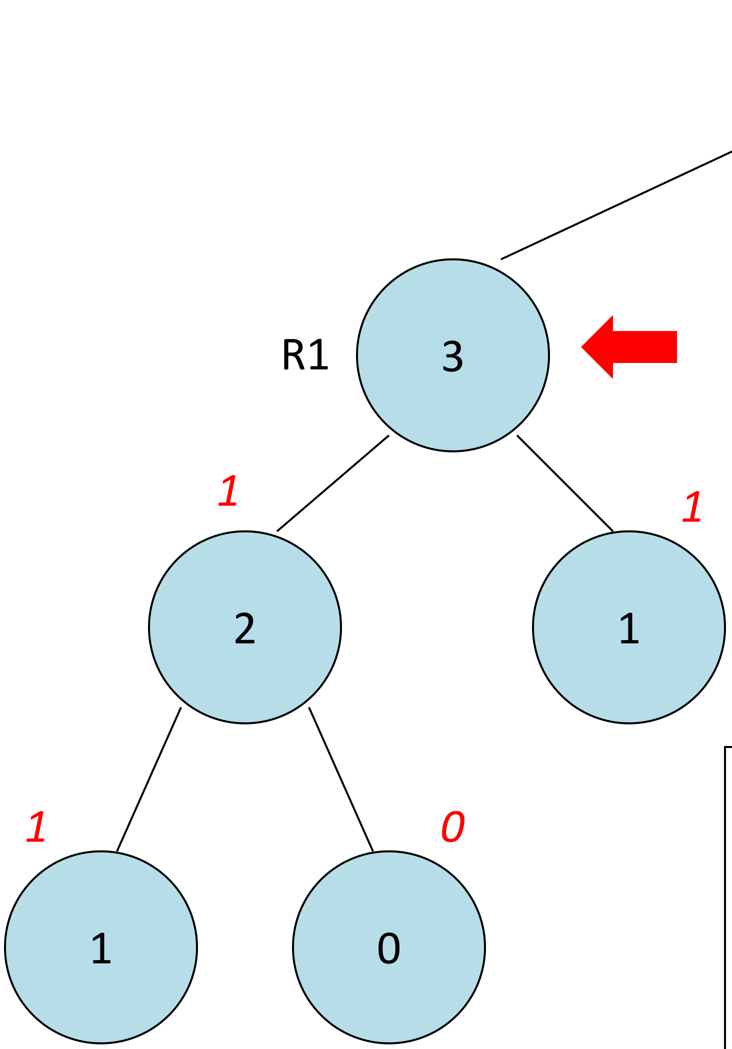
Storage for recursive calls

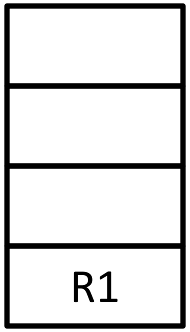


```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



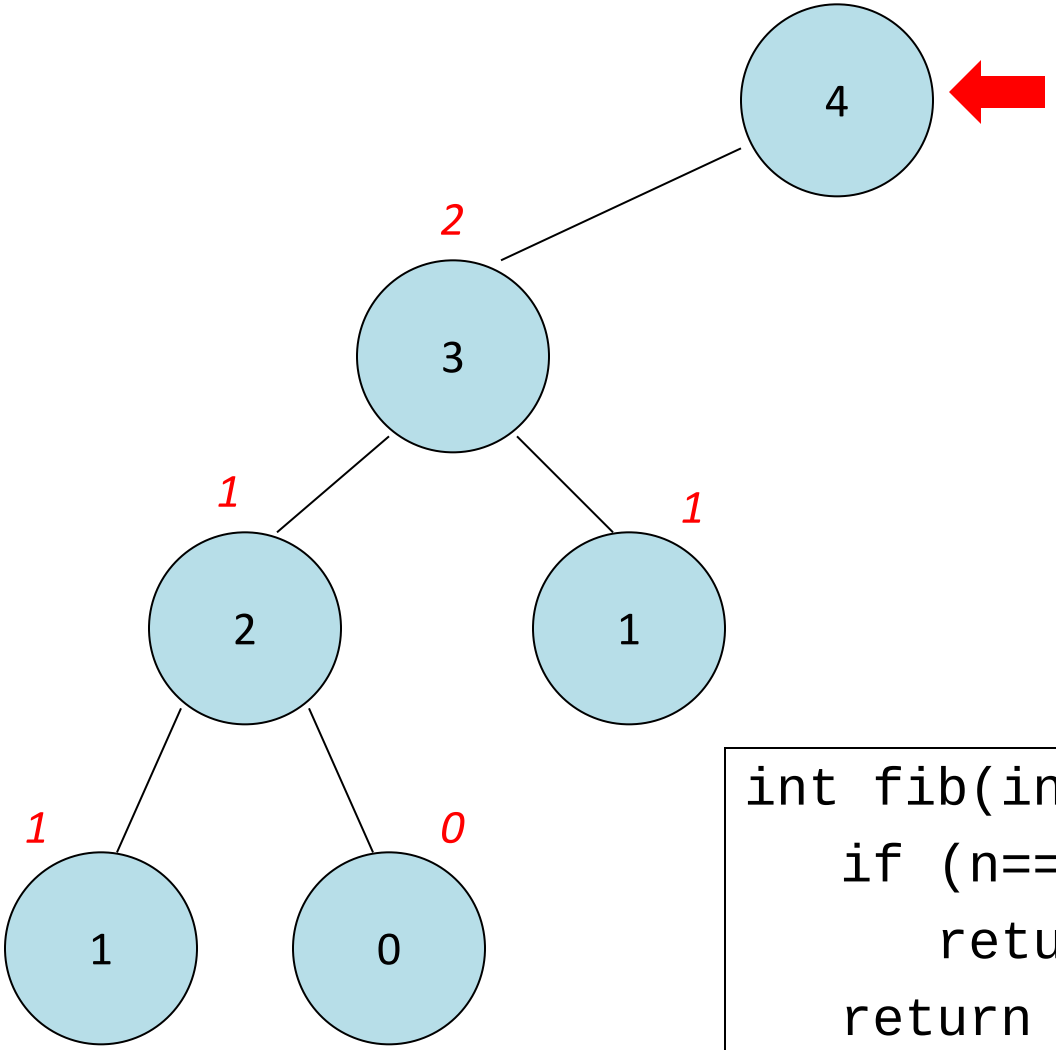


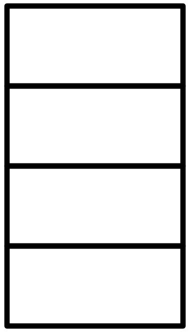
Storage for
recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



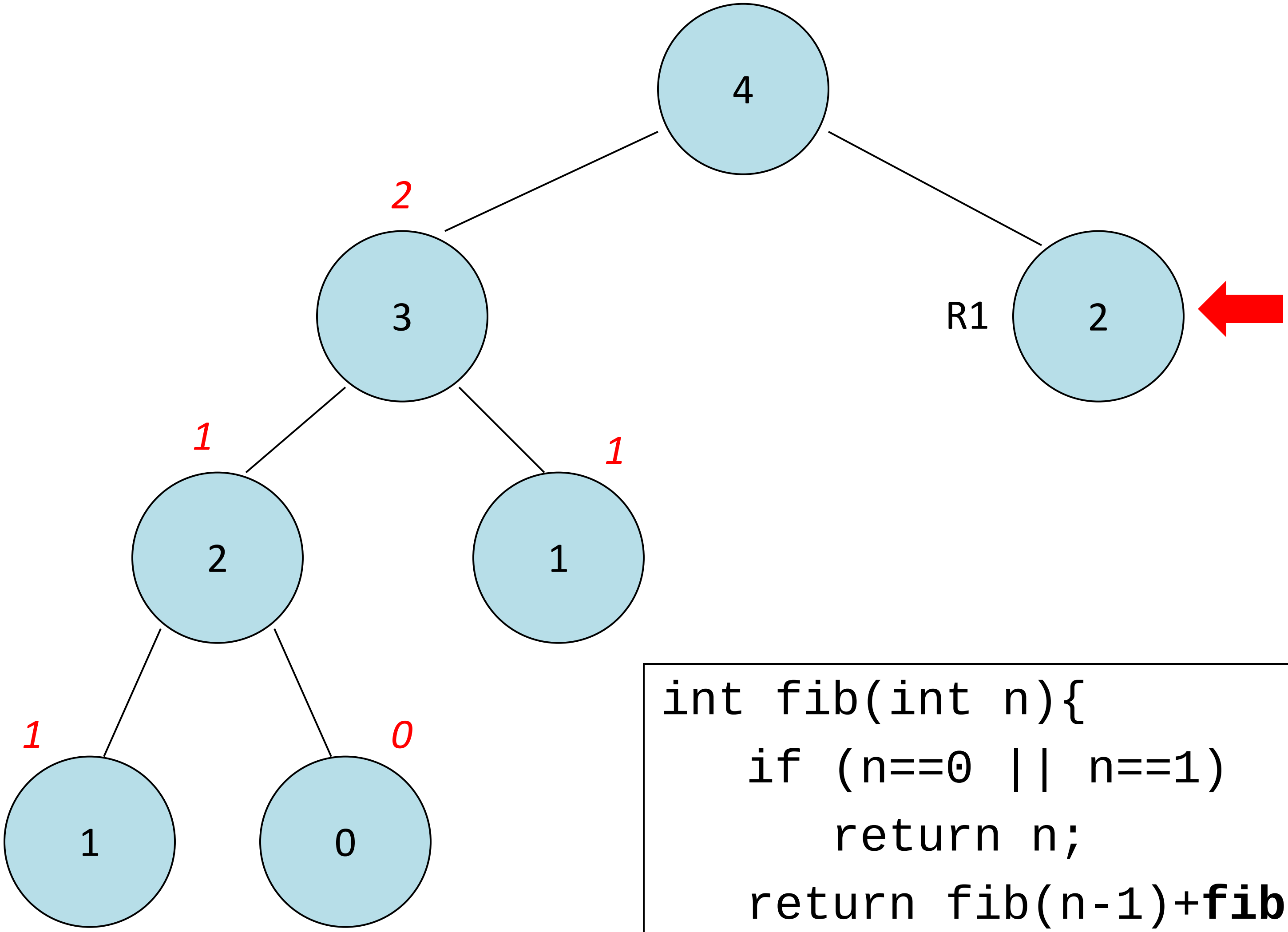


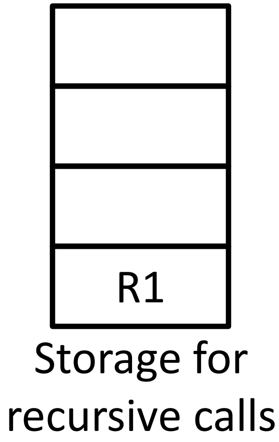
Storage for
recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



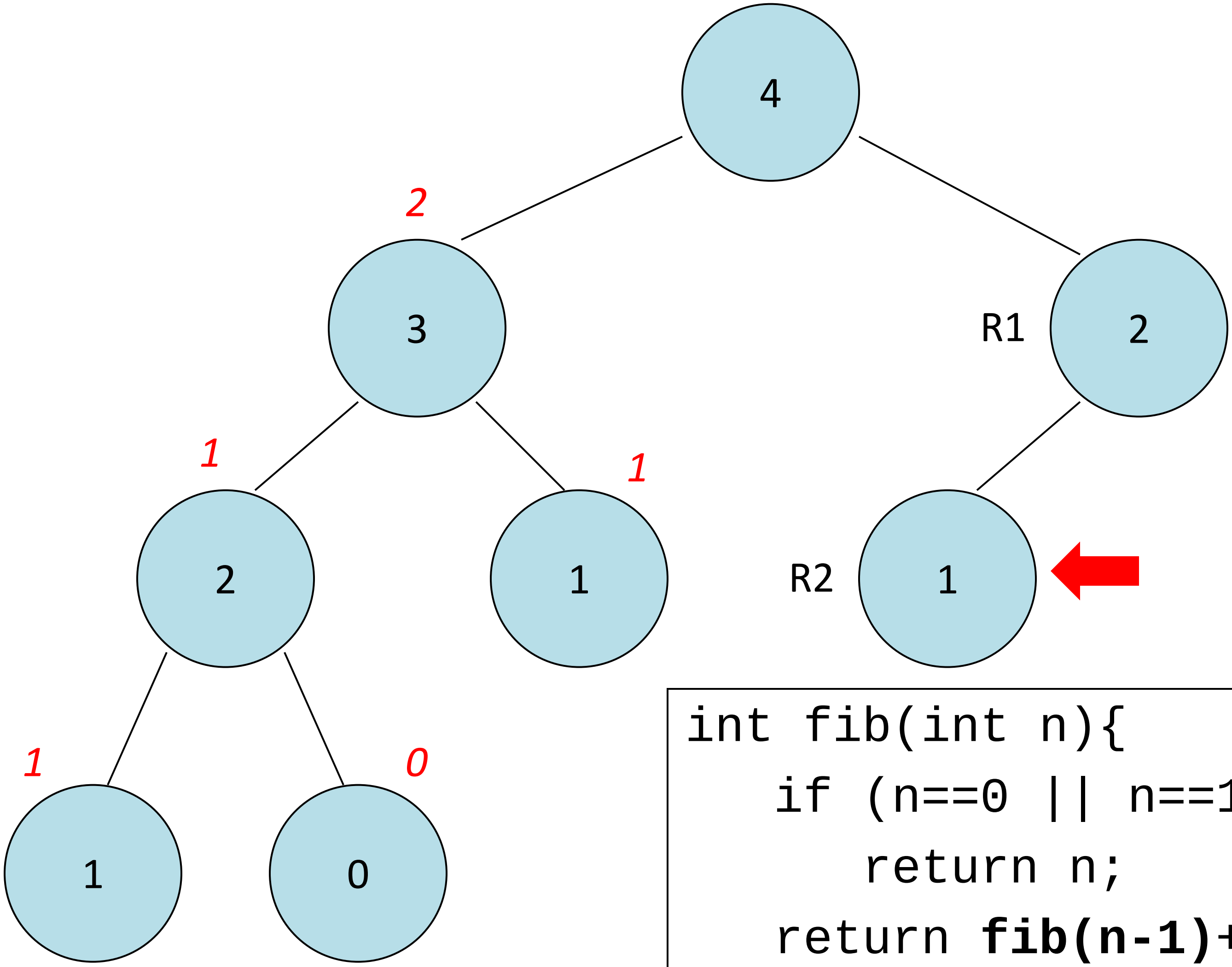


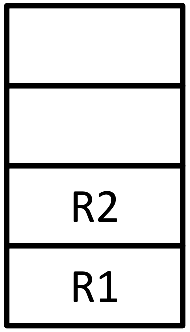
Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



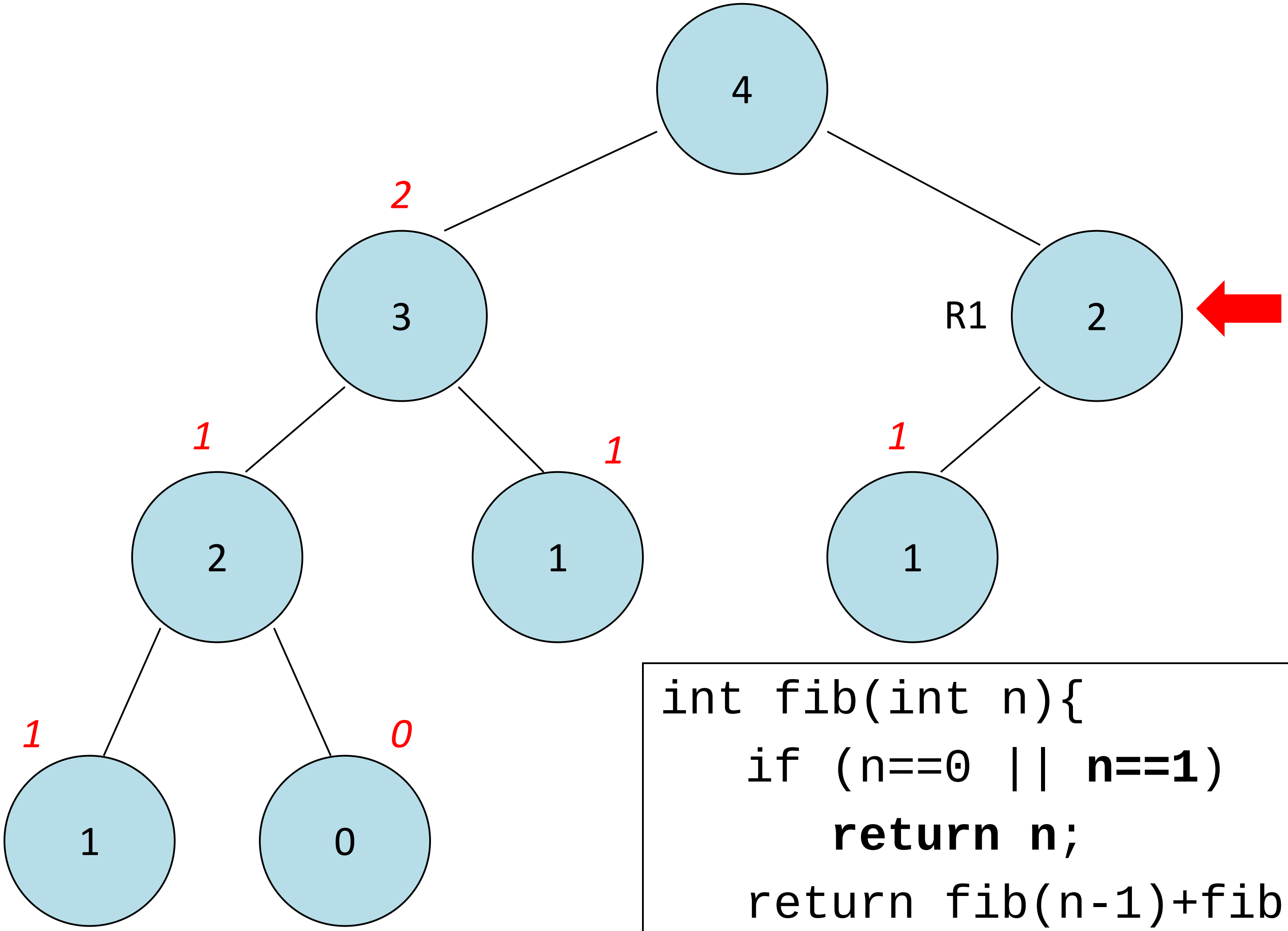


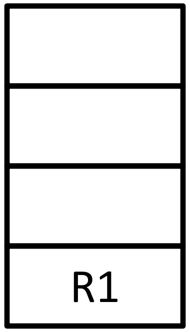
Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



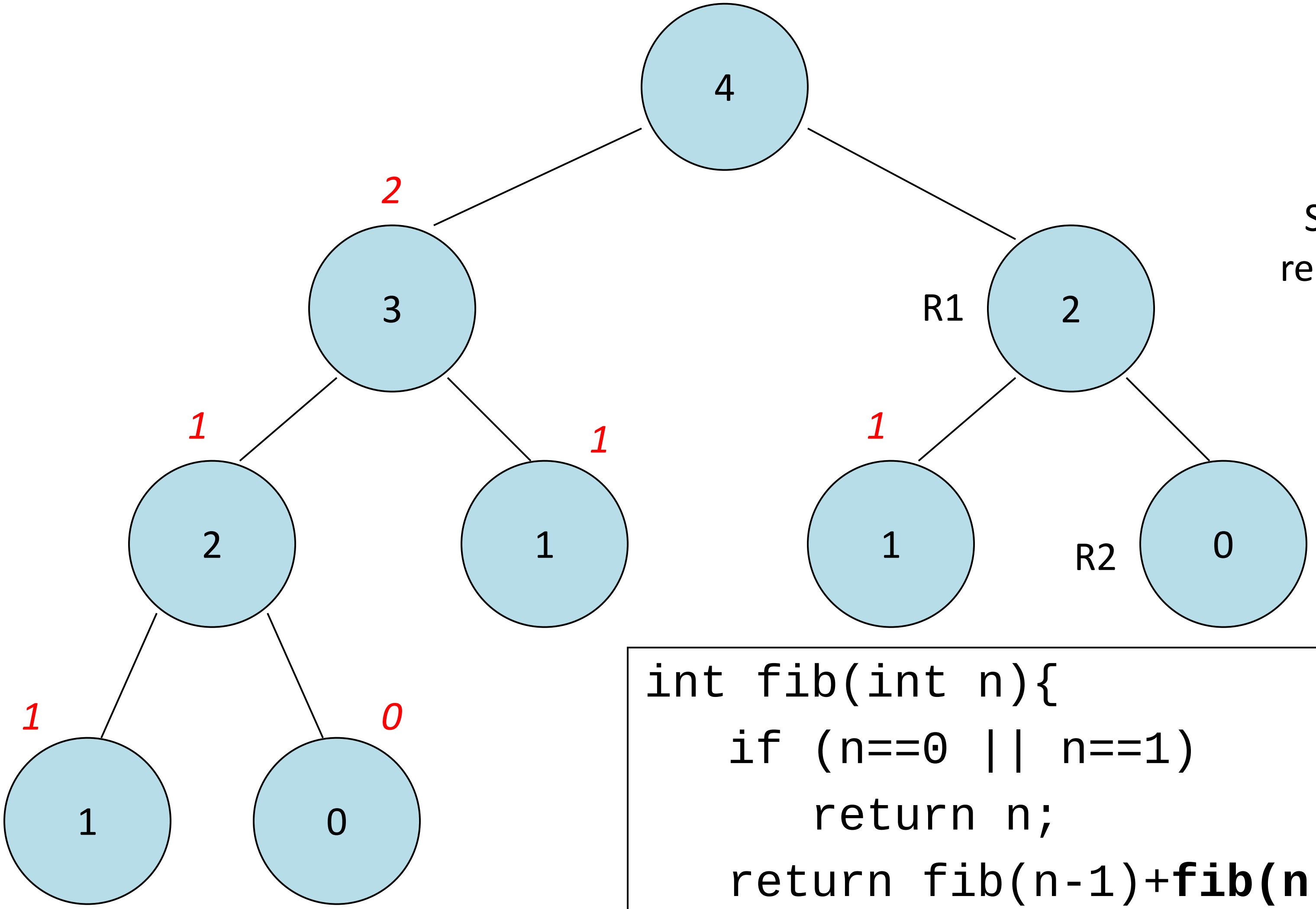


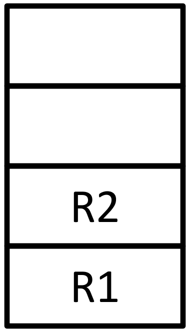
Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



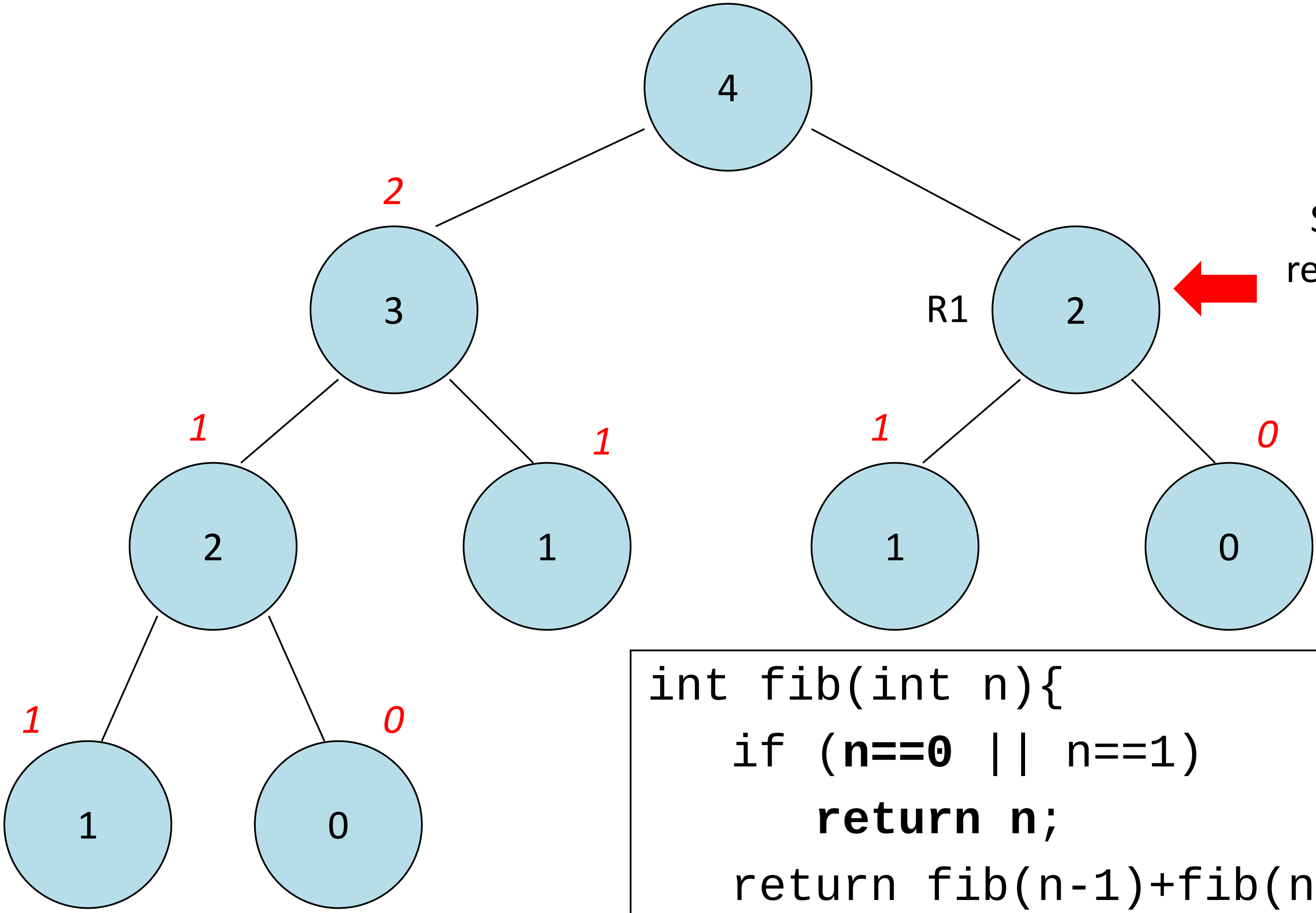


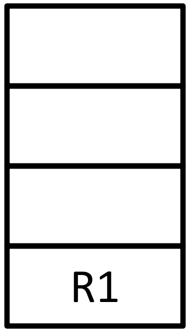
Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



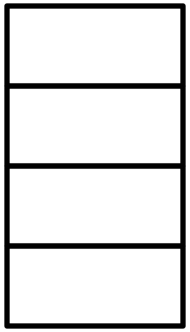


Storage for recursive calls

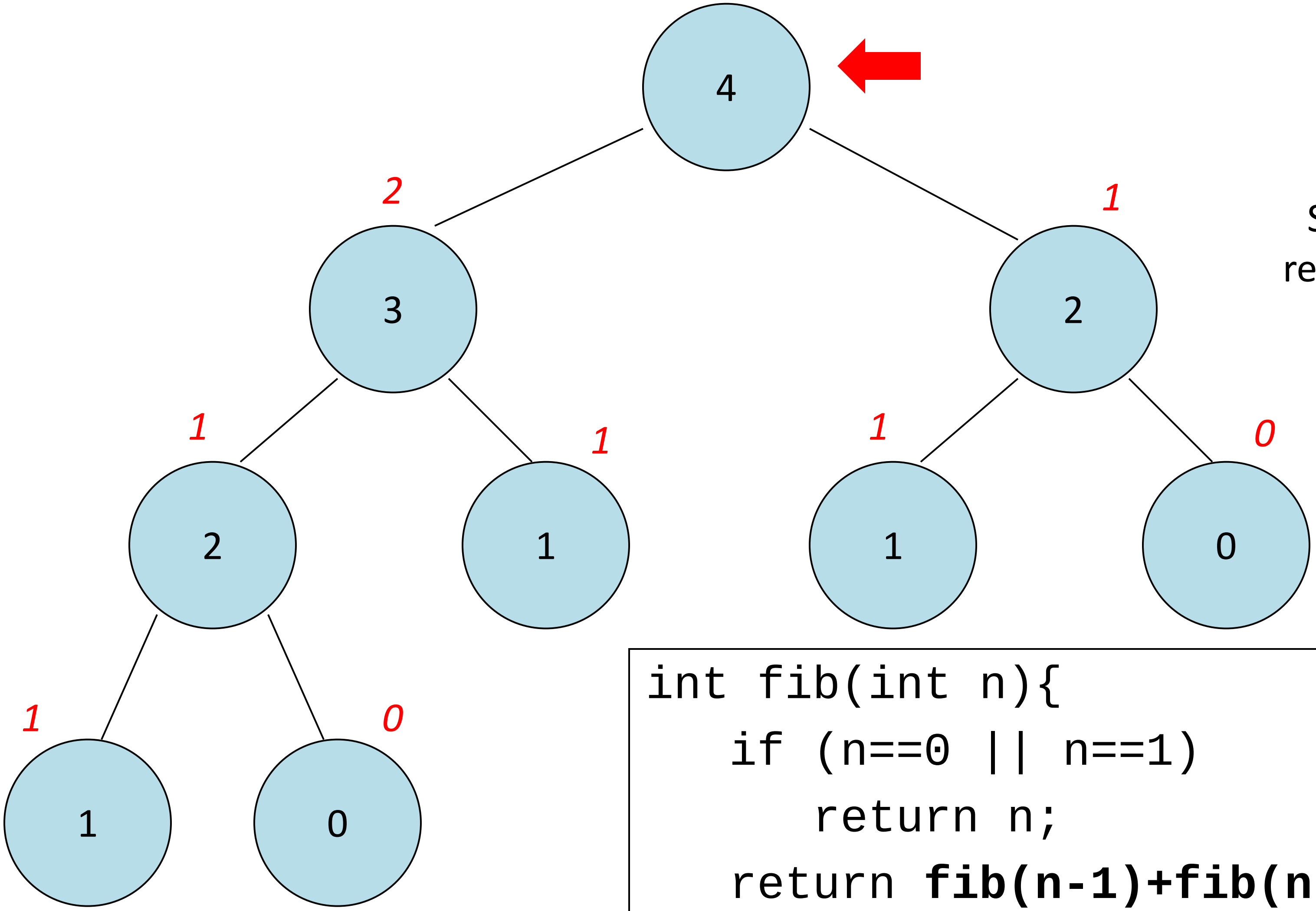
```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)



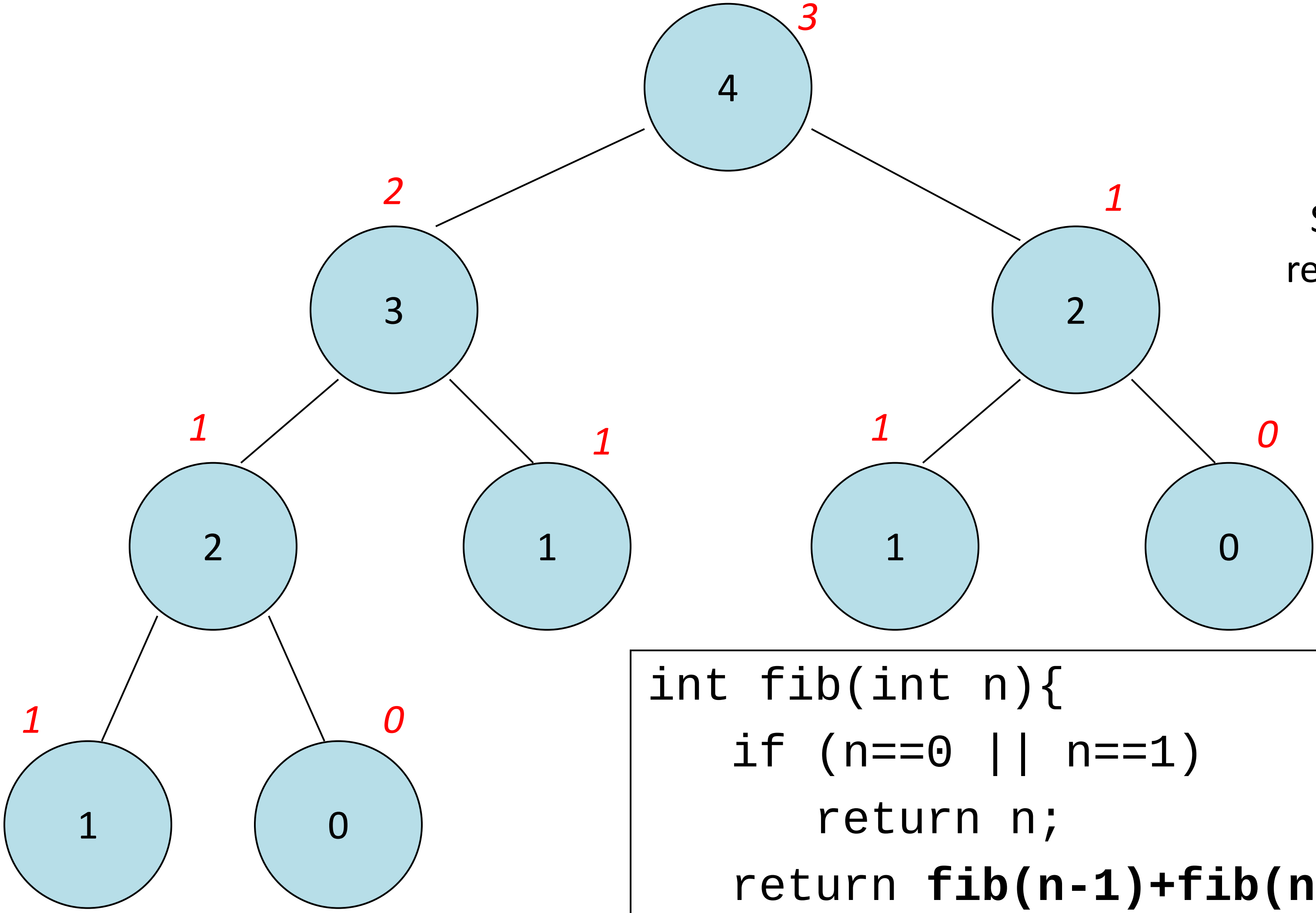
Storage for recursive calls

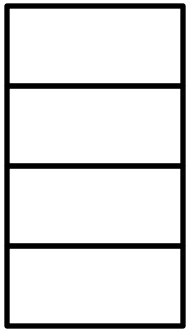


```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Tree of Recursive Calls for fib(4)





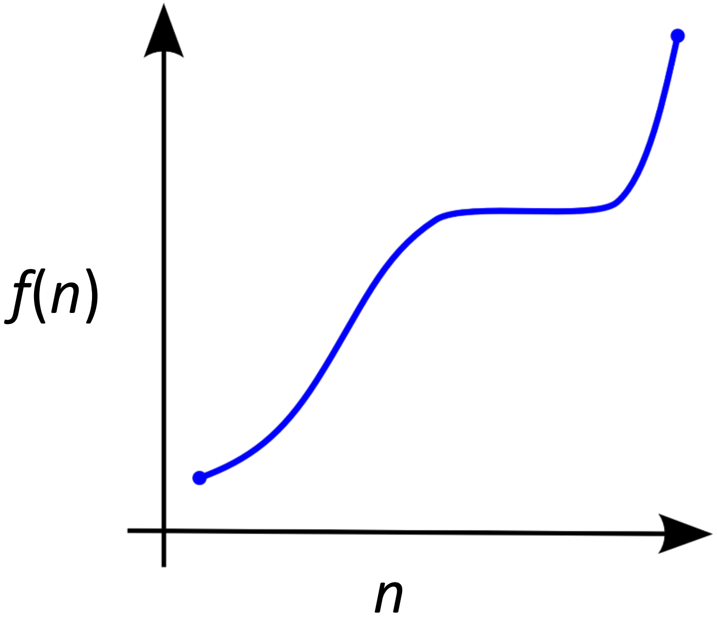
Storage for recursive calls

```
int fib(int n){
   if (n==0 || n==1)
      return n;
   return fib(n-1)+fib(n-2);
}
```

Return values are shown in *red italic*

Note: Increasing Functions

- A function $f(n)$ is called increasing, if for all x and y such that $x \leq y$, $f(x) \leq f(y)$, so $f(n)$ preserves the order.



Worst-Case Space Complexity (1/2)

- Let S(*n*) be the worst-case space required by fib(*n*).
- Let *c* be the constant space for the local storage (the local storage required in the call is at most some constant *c*).
- The calls fib(*n*-1) and fib(*n*-2) execute one after the other; hence, storage can be reused.
- Therefore, we can set up the following recurrence formula for S(*n*):

$$S(n) = \max\{\, S(n-1), S(n-2) \,\} + c$$

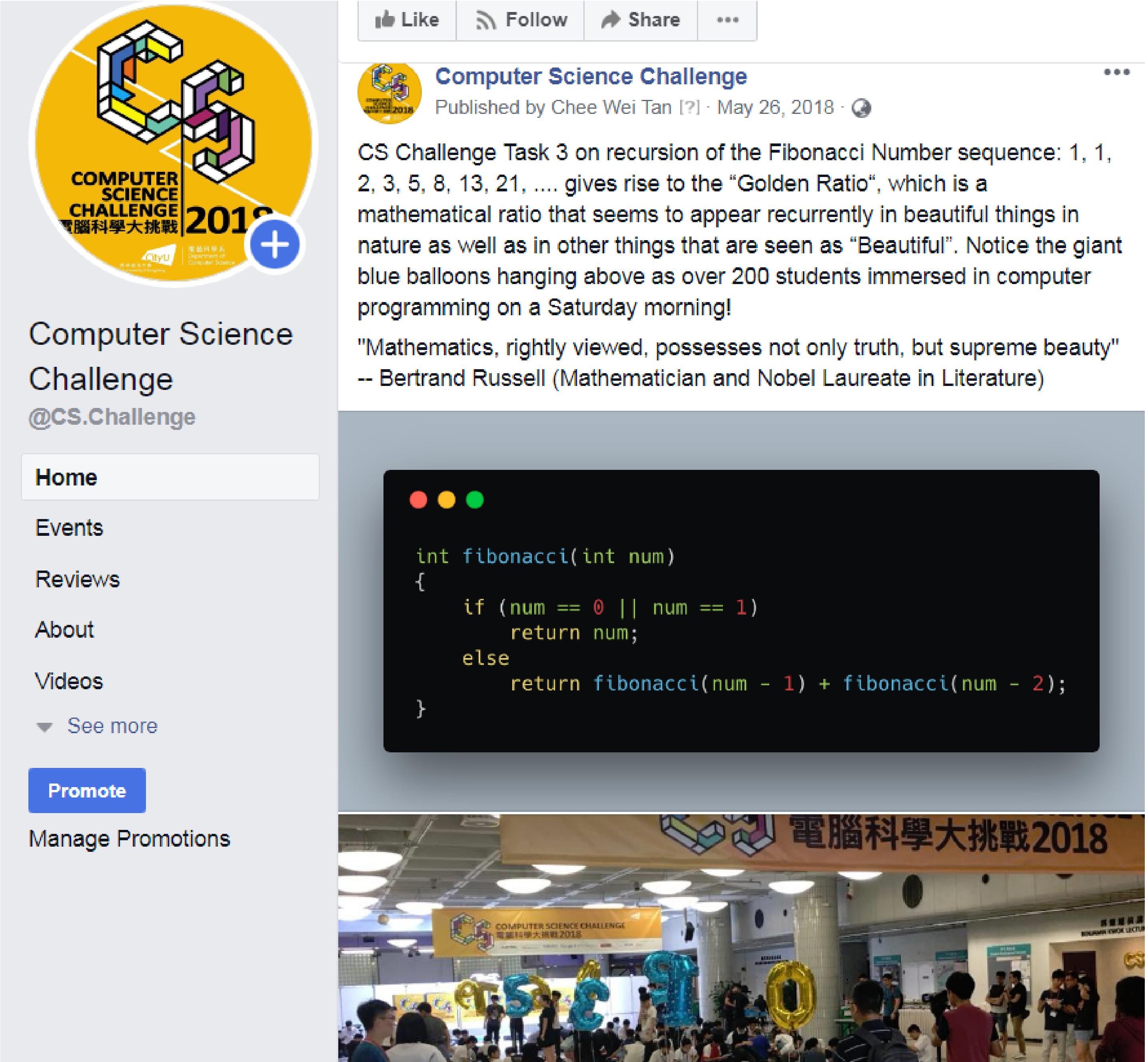
- Assuming $S(n)$ is an increasing function (i.e., $S(\mathrm{n}-1) \geq S(\mathrm{n}-2)$), $\boldsymbol{S(n) = S(n-1) + c}$

Worst-Case Space Complexity (2/2)

- Solving the recurrence formula:

$$
\begin{array}{rl}
S(n) = & S(n\text{-}1) + c \\
S(n\text{-}1) = & S(n\text{-}2) + c \\
S(n\text{-}2) = & S(n\text{-}3) + c \\
 & \dots \\
+)\ \ S(2) = & S(1) + c \\ \hline
S(n) = & S(1) + (n\text{-}1)c \\
 = & \mathrm{O}(n)
\end{array}
$$

The Beauty of Data Structures

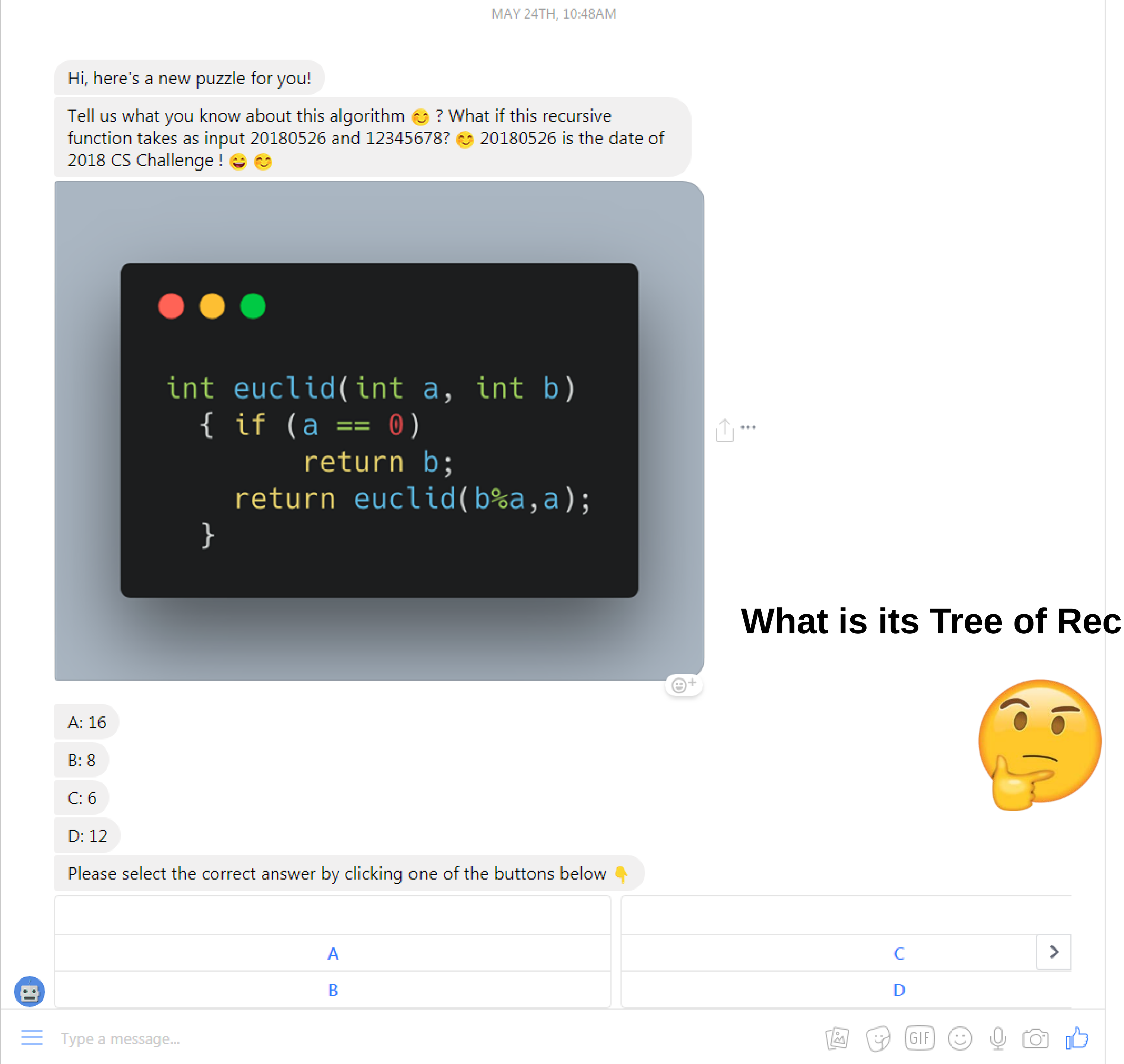


Computer Science Challenge
Published by Chee Wei Tan [?] · May 26, 2018 ·

CS Challenge Task 3 on recursion of the Fibonacci Number sequence: 1, 1, 2, 3, 5, 8, 13, 21, ... gives rise to the "Golden Ratio", which is a mathematical ratio that seems to appear recurrently in beautiful things in nature as well as in other things that are seen as "Beautiful". Notice the giant blue balloons hanging above as over 200 students immersed in computer programming on a Saturday morning!

"Mathematics, rightly viewed, possesses not only truth, but supreme beauty" -- Bertrand Russell (Mathematician and Nobel Laureate in Literature)

```
int fibonacci(int num)
{
    if (num == 0 || num == 1)
        return num;
    else
        return fibonacci(num - 1) + fibonacci(num - 2);
}
```



MAY 24TH, 10:48AM

Hi, here's a new puzzle for you!

Tell us what you know about this algorithm 😊 ? What if this recursive function takes as input 20180526 and 12345678? 😊 20180526 is the date of 2018 CS Challenge ! 😄 😊

```
int euclid(int a, int b)
  { if (a == 0)
          return b;
      return euclid(b%a,a);
  }
```

A: 16

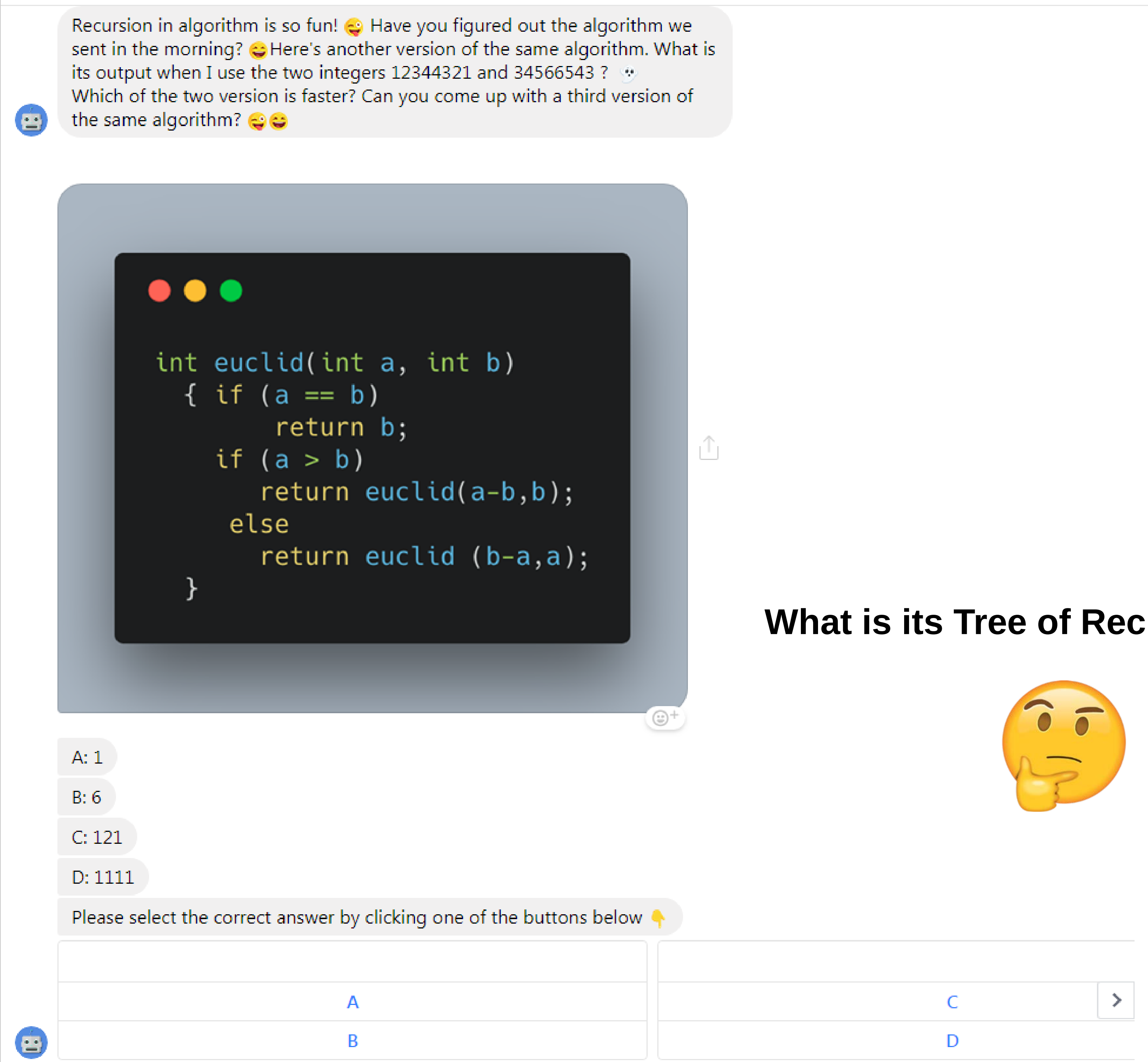
B: 8

C: 6

D: 12

Please select the correct answer by clicking one of the buttons below 👇

What is its Tree of Recursion?



Recursion in algorithm is so fun! 😜 Have you figured out the algorithm we sent in the morning? 😄Here's another version of the same algorithm. What is its output when I use the two integers 12344321 and 34566543 ? 💀
Which of the two version is faster? Can you come up with a third version of the same algorithm? 😜😄

```
int euclid(int a, int b)
  { if (a == b)
          return b;
    if (a > b)
         return euclid(a-b,b);
     else
         return euclid (b-a,a);
  }
```

A: 1

B: 6

C: 121

D: 1111

Please select the correct answer by clicking one of the buttons below 👇

What is its Tree of Recursion?