Recursion

Introduction

- Recursion is where a function calls itself.
- Example:

```
int factorial(int n)
{
    if (n == 0)
        return(1);
    else
        return(n * factorial(n - 1));
}
```

recursive call

- A recursive function will have:
  - One or more recursive calls.
  - One or more *stopping cases*.
    - Also known as *simple cases* or *terminal conditions*.

Structure of Recursive Functions (continued)

```
int multiply(int m, int n)
{
    int ans;

    if (n == 1)
        ans = m;
    else
        ans = m + multiply(m, n - 1);

    return(ans);
}
```

stopping case (base case)

recursive call

Operation of Recursive Functions

- Recursion works because the system maintains stack memory for you.
  - Each time a function is called, a stack frame is *pushed* onto the stack.
  - Each time a function returns, the stack frame is *popped* from the stack.
- Example: call the multiply() function from main.

```
void main(void)
{
    int result;

    result = multiply(5, 3);
}
```

Operation of Recursive Functions

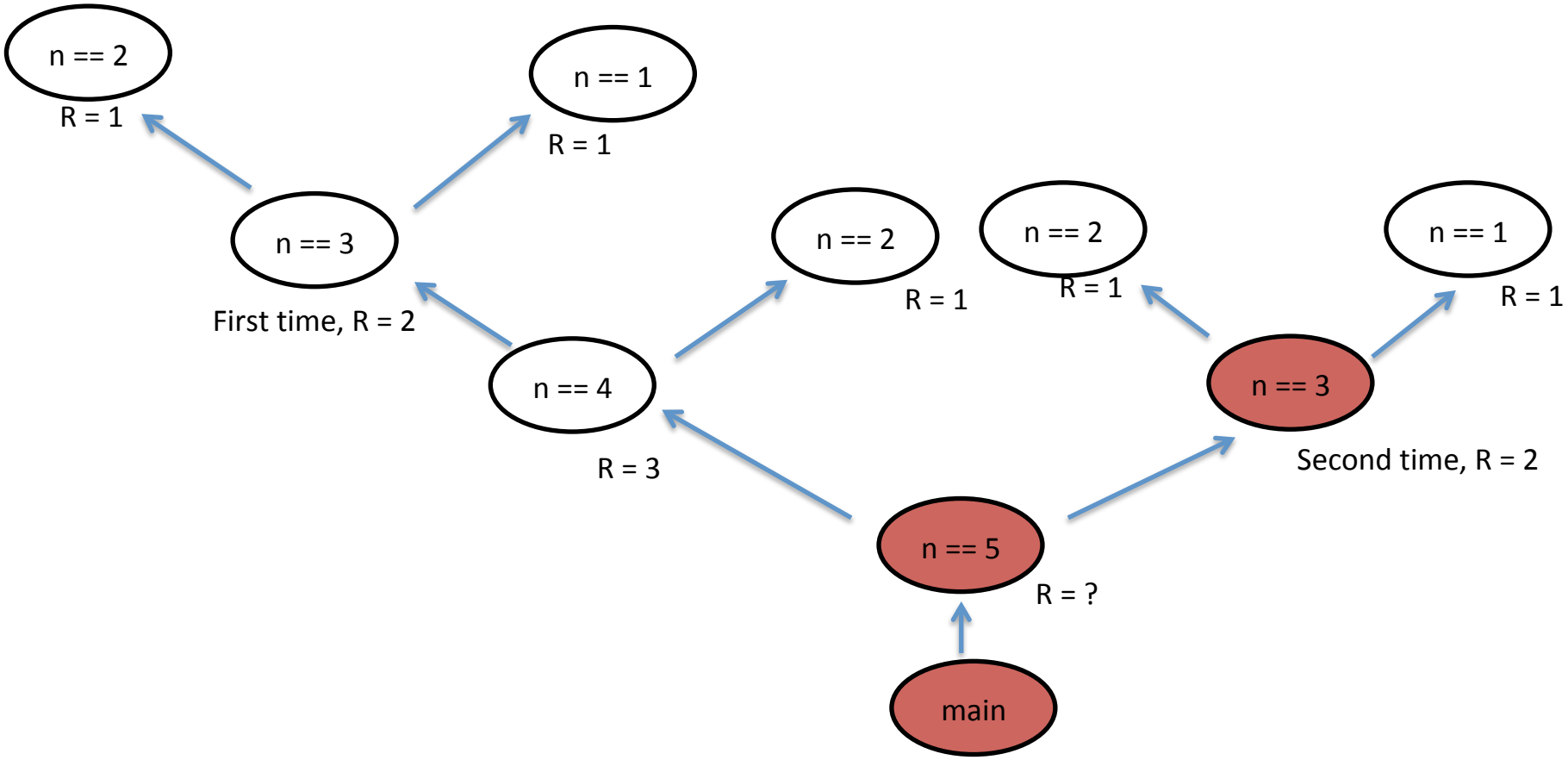
- A recursive function may call, itself more than once.
- Example: Consider calculation of n^{th} term of a Fibonacci series:

```
if n is 1 or 2 fib = 1
Otherwise fib = fib(n-1)+ fib(n-2)
```

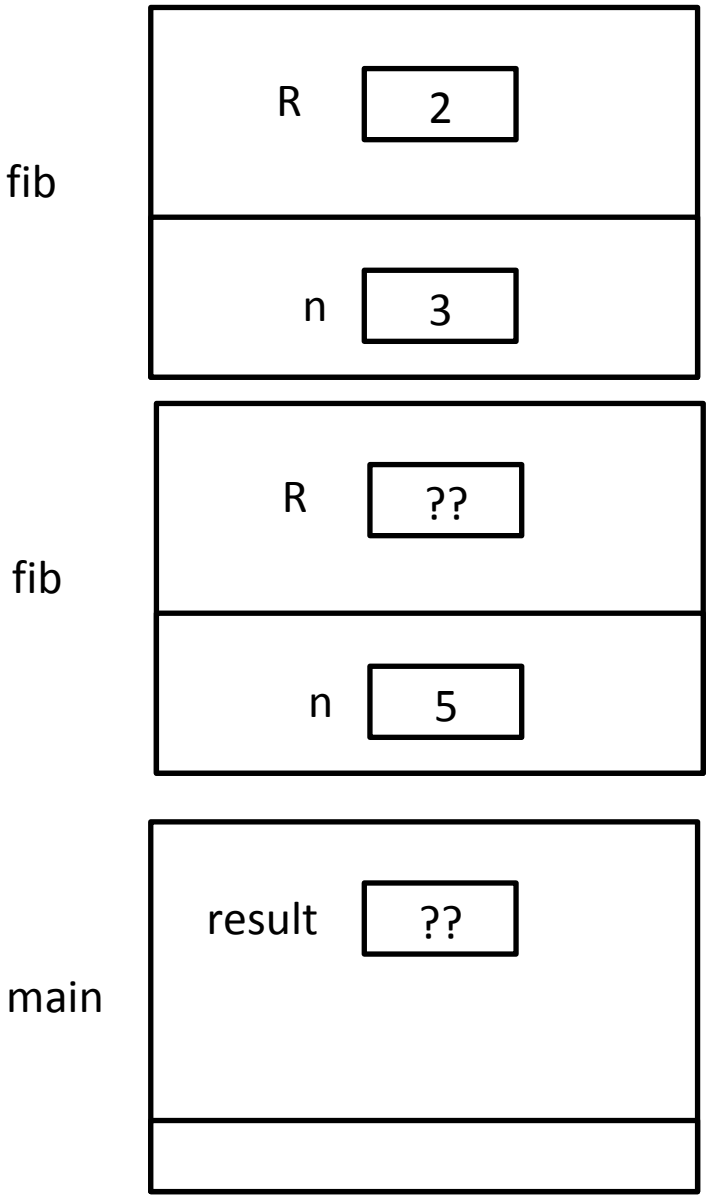
```
int fib(int n){
     int R;
    if(n == 1 || n ==2) R = 1;
    else
    R = fib(n-1) + fib(n-2);
    // Point one - when R is equal 2 for 2nd time
    return R;
}
```

```
void main(void) {
    int result;
    result = fiboncci(5);
}
```

AR at Point One



AR at Point One:



Another Example

- Writing a recursive solution for the following search function

```
const char* search_recursively(const char**a, int n, const char* target);
/*  REQUIRES: n > 0, a[0], a[1], a[2]..., a[n-1], exists.
 *   PROMISES: if array a contains a value that matches argument target, returns a
 *                  pointer to that element. Otherwise returns NULL .
 */
```

Solution

```
const char* search_recursively(const char**a, int n, const char* target)
{
   assert (n >= 0);
   if(n==0)
      return NULL;
   else if(strcmp(*a, target)==0)
      return *a;
   else
    return  search_recursively(a+1, n-1 , target);
}
```

Binary Search

Solution

```
int Binary_Search(int key, const int *a, int low, int hi)
{
    int result, mid;
    if (low == hi){
        if(key == a[low])
            return low;
        else
            return -1;
    }
    mid = (low + hi)/2;
    if (key <= a[mid])
        result = Binary_Search(key, a, low, mid);
    else
        result = Binary_Search(key, a, mid+1, hi);
    return result;
}
```

Sorting Techniques

Introduction

- Sorting data is one of the most common tasks in computing.
- There are dozens of sorting algorithms, each with advantages and disadvantages.
- Your choice depends on the following factors:
  - The type and amount of data to be sorted.
  - The speed of the sort.
  - The amount of memory needed by the sort.
  - Whether the data to be sorted is in random order, already partially ordered, or in reverse order.
  - The complexity of the sort algorithm, and the ease with which it can be programmed.
  - Whether the data can be sorted in RAM, or must be sorted in a file.

Introduction (continued)

- Sophisticated sort algorithms try to:
  - maximize speed
  - minimize memory usage
- Sometimes simple, inefficient algorithms like the bubble sort are OK for small data sets:
  - The algorithm is easy to implement and debug.
  - Its speed is not much worse than other algorithms when sorting small data sets.
- The efficiency of a sort depends on:
  - The number of *comparisons* it makes.
  - The number of *swaps* it makes.
- Computer scientists have analyzed most algorithms, and have classified them according to their efficiency.
  - They use "order" (or "big-Oh") notation to do this.

Sorting Efficiency and Analysis (continued)

- Some possible classifications are:
  - $O(\log_2 N) = O(\log_2 8) = 3$
  - $O(N) = O(8) = 8$
  - $O(N\log_2 N) = O(8\log_2 8) = 24$
  - $O(N^2) = O(8^2) = 64$
- N is the number of items to be sorted.
- O( ) is the number of operations (comparisons & swaps) needed to sort N items.

Sorting Efficiency and Analysis (continued)

N	O(logN)	O(N)	O(NlogN)	$O(N^2)$
8	3	8	24	64
16	4	16	64	256
32	5	32	160	1024
64	6	64	384	4096
128	7	128	896	16384
256	8	256	2048	262144

Sorting Techniques

- Some of the sorting methods are listed below:
  - Insertion sort
  - Quick sort
  - Merge sort
  - Selection sort
  - Bubble sort
  - Heap sort
- Bubble Sort:
  - This the simplest method of sorting.
  - Keeps passing though the list (array or file), exchanging the adjacent elements that are out of order, continuing until the list is sorted.
  - This is the slowest method of sorting

Selection Sort

- Selection Sort:
  - Another simple sorting algorithm, called selection sort works as follow:
    - First, find the smallest element in the array and exchange it with element in the first position.
    - Then, find the second smallest element and exchange it with the second element.
    - Continue in this way until the entire array is sorted.
- Insertion Sort:
  - This sort algorithm is similar that people often use to sort the bridge hands.
    - Consider the element one at a time
    - Insert each element into its proper place among those already sorted.
  - The computer implementation of this method, requires making space for the element being inserted, by moving larger elements to the right.
  - The elements to the left of the current index are in sorted order.

A Few Internet Sites:

- https://www.bluffton.edu/homepages/facstaff/nesterd/java/SortingDemo.html
- https://www.toptal.com/developers/sorting-algorithms
- http://sorting.at/

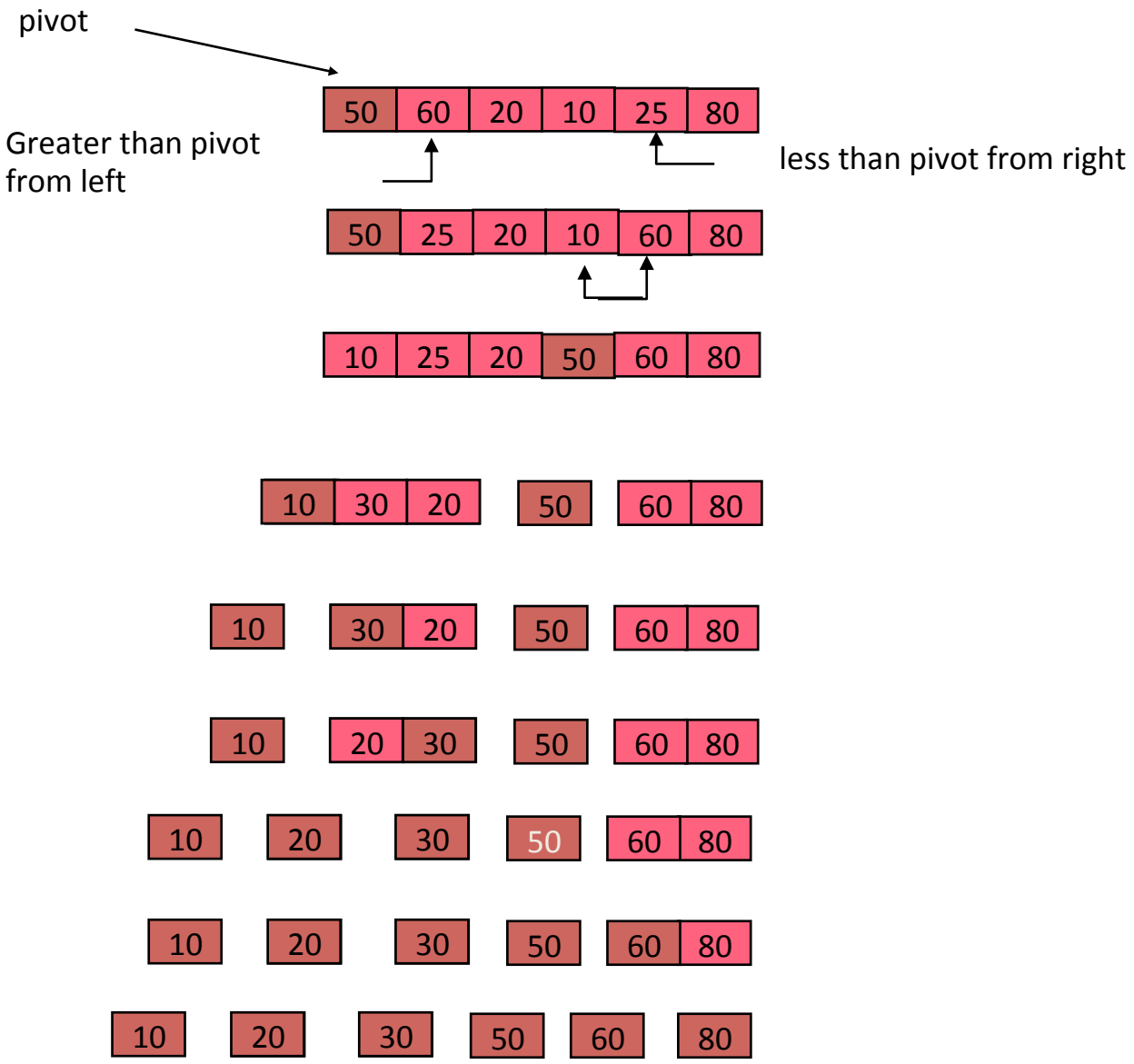
Quicksort

Quicksort

- Quicksort is one of the more widely used sort methods.
- Was invented in 1960, by C. A. R. Hoare.
- Works for variety of input data, and consumes fewer resources than any other sorting method in many situation.
- Standard C/C++ library, called qsort, uses this method of sorting.
- Quicksort is a divide and conquer method for sorting.
- It works by partitioning an array into two parts, then sorting the parts independently.

```
void quicksort (int a[ ], int left, int right)
{
    if (right <= left) return;
    int partitionPosition = partition(a, left, right);
    quicksort(a, left, partitionPosition-1);
     quicksort(a, partitionPosition+1, right);
}
```

Basic Algorithm



Standard C/C++ library Sort (qsort)

Introduction

- qsort() is a system-supplied function that implements the quick sort algorithm.
- Is available in C/C++ libraries that follow the ANSI standard.
- Is highly optimized.

```
void qsort(void *base, size_t nmemb, size_t size,
                int (* compar)(const void *, const void *));
```

  - `base:` pointer to the first element of the array
  - `nmemb:` number of elements to sort
  - `size:` size of each element, in bytes
  - `compar:` function called by qsort to compare the 2 elements pointed to by the 2 arguments.
    - if element1 < element2, returns -1
    - if element1 == element2, returns 0
    - if element1 > element2, returns +1

Example (continued)

```
int mycompare(const int *a,
                  const int *b)
{
    if (*a < *b)
        return(-1);
    else if (*a > *b)
        return(1);
    else
        return(0);
}
```

```
#include <string.h>

int mystrcompare(const char *a,
                         const char *b)
{
    if (strcmp(a,b)<0)
        return(-1);
    else if(strcmp(a,b)>0)
        return(1);
    else
        return(0);
}
```

```
void main(void)
{
    int array[SIZE];

    fill(array, SIZE);
    qsort(array, SIZE, sizeof(int), mycompare);
}
```