Chapter 10: Recursion

CSE 2010 Week 14

Code and Figures from : “C++ programming: an object-oriented approach” - Forouzan, Behrouz A, Gilberg, Richard, 2020.

What is recursion..

- Recursion is a powerful programming technique that allows us to break up complex computational problems into smaller, more simple ones.
- Recursion even allows us to implement a solution in a way that mirrors our natural (human) way of thinking about a problem.
- We can accomplish recursion by implementing functions that call themselves to complete a task.

There are 2 key requirements to make sure that our recursive functions are successful:

1. Every recursive call must simplify the computation in some way (use smaller values with each call).
2. There must be special cases to handle the simplest computations directly.
    - Each recursive function has a **general case** and **base case**.
    - A base case is the case that will terminate the recursion, while a general case is related to calls that do something and continue the recursion.

Fibonacci Numbers

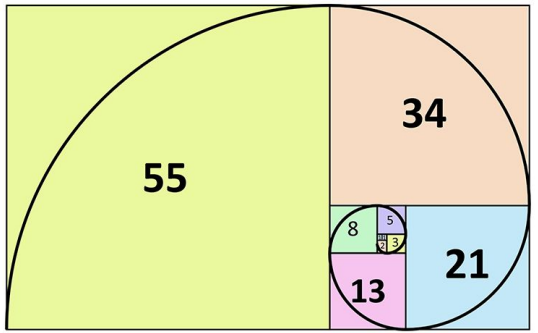
The Fibonacci sequence is a sequence of numbers in which each number is the sum of the previous two numbers.

The first 10 terms of a sequence are:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55

The Fibonacci numbers problem has two base cases and a general case

Base Case: F(0) = 0 & F(1) = 1

General Case: F(n) = F(n-1) + F(n-2)



Recursive Implementation to find the nth Fibonacci

The Fibonacci numbers problem has two base cases and a general case

Base Case: F(0) = 0 & F(1) = 1

General Case: F(n) = F(n-1) + F(n-2)

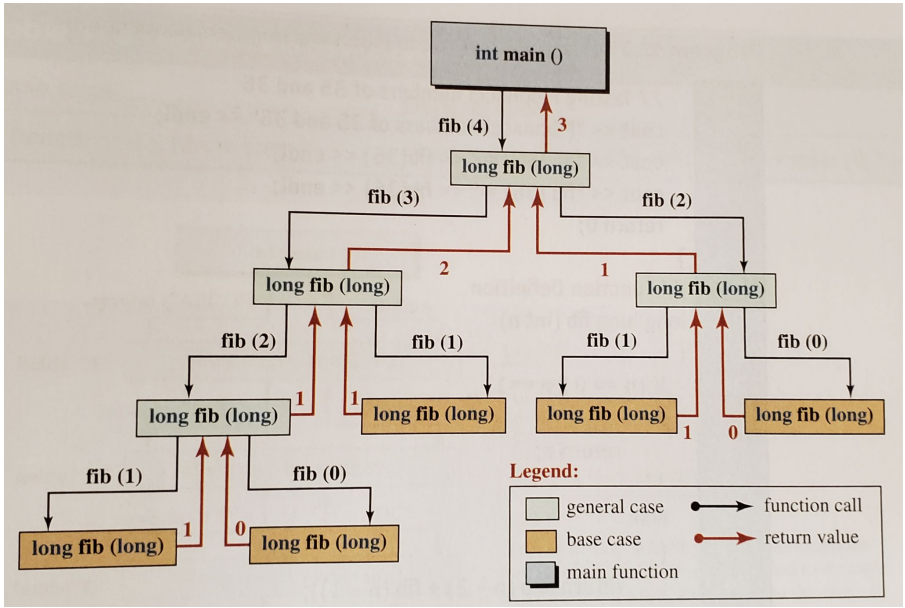
```
/*
 * Recusrive Implementation to find the nth
 * Fibonacci #
 */

#include <iostream>
using namespace std;
//long int = 8 bytes vs int = 4 bytes
long int fib(int n){
        if(n == 0 || n == 1){
                //base case
                return n;
        }
        else{
                //general case
                return (fib(n-2) + fib(n-1));
        }
}

int main()
{
        //Testing for the first 10 fib #s
        cout << "Fibonacci numbers from 0 to 10:\n";
        for(int i = 0;i <= 10; i++){
                cout << "F(" << i << ") = " << fib(i) << "\n";
        }
        cout << "\n\n";
        //testing larger fib numbers
        cout <<"F(35) = " << fib(35) << "\n";
        cout <<"F(36) = " << fib(36) << "\n";

        return 0;
}
```

Output

```
Fibonacci numbers from 0 to 10:
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55


F(35) = 9227465
F(36) = 14930352
```

Recursive Trace

```
/*
 * Recusrive Implementation to find the nth
 * Fibonacci #
 */

#include <iostream>
using namespace std;
//long int = 8 bytes vs int = 4 bytes
long int fib(int n){
        if(n == 0 || n == 1){
                //base case
                return n;
        }
        else{
                //general case
                return (fib(n-2) + fib(n-1));
        }
}

int main()
{
        //Testing for the first 10 fib #s
        cout << "Fibonacci numbers from 0 to 10:\n";
        for(int i = 0;i <= 10; i++){
                cout << "F(" << i << ") = " << fib(i) << "\n";
        }
        cout << "\n\n";
        //testing larger fib numbers
        cout <<"F(35) = " << fib(35) << "\n";
        cout <<"F(36) = " << fib(36) << "\n";

        return 0;
}
```



Let’s compare the recursive and iterative methods for finding fibonacci numbers

Fibonacci - Time and Space Complexity Comparison

Recursive:

- Time complexity: O(2^n) or exponential
- Space complexity: O(n)

Iterative:

- Time complexity: O(n)
- Space complexity: O(1) or constant

Greatest Common Divisor (GCD)

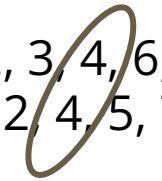
One function often needed in mathematics and computer science is one to find the greatest common divisor (GCD) of two positive integers.

Finding the GCD of two positive integers means finding the greatest integer that evenly divides into both integers.

Two positive integers may have many common divisors, but only one GCD.

Example: 12 & 140

Divisors of 12: 1, 2, 3, 4, 6, 12
Divisors of 140: 1, 2, 4, 5, 7, 10, 14, 20, 28, 35, 70, 140



GCD(12,140) = 4

Euclidean Algorithm for GCD

Thankfully this pretty smart guy named Euclid came up with a recursive algorithm to easily find the GCD of two positive integers.

Given two positive integers, A & B, the Euclidean Algorithm for finding GCD(A,B) is as follows:

- If B = 0 then GCD(A,B)=A, since the GCD(A,0)=A, and we can stop.
- Otherwise, write A in quotient remainder form (A = B * Q + R), and solve for R.
- We can then find GCD(B,R) using the Euclidean Algorithm since GCD(A,B) = GCD(B,R)

Recursive Implementation of Euclidean Algorithm for GCD

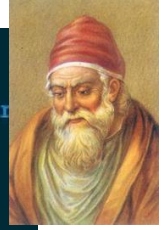
Base Case:

- If B = 0 then GCD(A,B)=A

General Case:

- Calculate R = A%B
- Find GCD(B,R)

```cpp
/*
 * Filename: gcd.cpp
 * Program to recursively find the GCD of some pairs of positive integer
 */
#include <iostream>
using namespace std;

int gcd(int first, int second){
        //base case
        if(second == 0)
                return first;
        else{
                //general case
                return gcd(second, first%second);
        }
}

int main()
{
        //testing gcd of 5 pairs
        cout << "gcd(12,140) = " << gcd(12,140) << "\n";
        cout << "gcd(21,35) = " << gcd(21,35) << "\n";
        cout << "gcd(13,0) = " << gcd(13,0) << "\n";
        cout << "gcd(0,45) = " << gcd(0,45) << "\n";
        cout << "gcd(154,200) = " << gcd(154,200) << "\n";

        return 0;
}
```



Euclid judging our recursive function

Output:

```
gcd(12,140) = 4
gcd(21,35) = 7
gcd(13,0) = 13
gcd(0,45) = 45
gcd(154,200) = 2
```

Palindrome Checker

A string is considered a palindrome if it reads the same forward and backward.

Examples: rotor, racecar, “Go hang a salami I’m a lasagna hog”

We want to write a function that checks whether a string is a palindrome. If it is, it returns true, if it’s not, it returns false.

Let’s think of how we can come up with the base and general cases.

Palindrome Checker - General case

Remember that the purpose of recursion is to simplify the problem with each function call, so how can we simplify a string to check if it's a palindrome?

Example: rotor

1. Remove the first character: otor
2. Remove the last character: roto
3. Remove both the first and last character: oto
4. Remove a character from the middle: roor
5. Cut the string into two halves: rot or

Which of these methods best follows our natural way of thinking checking whether a string is a palindrome?

- #3, so this will be our general case

Palindrome Checker - Base case(s)

Our base case lets us know when to stop the recursion. Let’s think of what conditions we will have met to determine whether the string is a palindrome or not once we have simplified the string enough with the general case.

The simplest strings for the palindrome test:

1. Strings with two characters - they should be equal
2. Strings with a single character - is a palindrome
3. The empty string - is a palindrome

These will be our base cases.

Recursive Implementation to check if a string is a palindrome

Base Case:

1. if(length <=1), return true
2. Check if first and last char are the same.

General Case:

- Split into smaller substring removing first and last characters

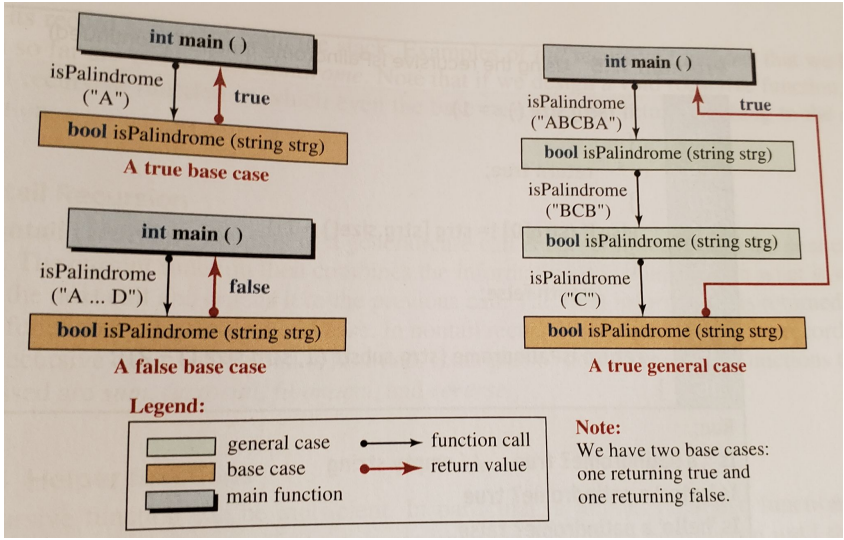
Output:

```
"" is a palindrome: true
"o" is a palindrome: true
"racecar" is a palindrome: true
"hello" is a palindrome: false
```

```cpp
/*
 * Filename:palindrome.cpp
 * Program that recusrively determines whether a string
 * is a palindrome or not
 */
#include <iostream>
using namespace std;

bool isPalindrome(string strg){
        //first base case if size is 1 or 0
        //then it is a palindrome
        if(strg.size() <= 1)
                return true;
        //second base case if first char and last char don't match
        //then it is NOT a palindrome
        if(strg[0] != strg[strg.size()-1])
                return false;

        //general case
        //creates a new substring removing 1st and last char
        return isPalindrome(strg.substr(1,strg.size()-2));
}

int main()
{
        //declare some strings to test out
        string str1("");//empty string
        string str2("o");//single char
        string str3("racecar");//palindrome
        string str4("hello");//not a palindrome

        //edit cout to display true or false instead of 1 or 0
        cout << boolalpha;
        //test strings
        cout << "\"\" is a palindrome: " << isPalindrome(str1) << "\n";
        cout << "\"o\" is a palindrome: " << isPalindrome(str2) << "\n";
        cout << "\"racecar\" is a palindrome: " << isPalindrome(str3) << "\n";
        cout << "\"hello\" is a palindrome: " << isPalindrome(str4) << "\n";

        return 0;
}
```

Recursive Trace

```cpp
/*
 * Filename:palindrome.cpp
 * Program that recusrively determines whether a string
 * is a palindrome or not
 */
#include <iostream>
using namespace std;

bool isPalindrome(string strg){
        //first base case if size is 1 or 0
        //then it is a palindrome
        if(strg.size() <= 1)
                return true;
        //second base case if first char and last char don't match
        //then it is NOT a palindrome
        if(strg[0] != strg[strg.size()-1])
                return false;

        //general case
        //creates a new substring removing 1st and last char
        return isPalindrome(strg.substr(1,strg.size()-2));
}

int main()
{
        //declare some strings to test out
        string str1("");//empty string
        string str2("o");//single char
        string str3("racecar");//palindrome
        string str4("hello");//not a palindrome

        //edit cout to display true or false instead of 1 or 0
        cout << boolalpha;
        //test strings
        cout << "\"\" is a palindrome: " << isPalindrome(str1) << "\n";
        cout << "\"o\" is a palindrome: " << isPalindrome(str2) << "\n";
        cout << "\"racecar\" is a palindrome: " << isPalindrome(str3) << "\n";
        cout << "\"hello\" is a palindrome: " << isPalindrome(str4) << "\n";

        return 0;
}
```



Recursive Sort & Search Algorithms

Sorting and searching through a list of elements is such a common task, that computer scientists have come up with several efficient algorithms to accomplish this.

For this class, we'll introduce some of the most popular ones.

- Quick Sort
- Merge Sort
- Binary Search

All 3 of these algorithms are examples of the "divide and conquer" approach of problem solving.

Divide and conquer algorithms accomplish their task by recursively breaking down a problem into two or more subproblems, until a solution to each subproblem is solved. Then, the solutions are all combined to give a solution to the original problem.

Quick Sort Algorithm

Given an array, *A*, of *n* elements : *A[0...n-1]*

- If array/subarray only has <=1 element, stop function (return)
- Else
  - Divide:
    - Pick one element in the array to use as a *pivot*.
    - Partition the elements into two sub-arrays
      - Elements less than or equal to pivot
      - Elements greater than pivot
      - The elements don't have to be in order in the subarrays, they just have to be in the right subarray.
      - This is all done within the original array.
  - Conquer:
    - Recursively call Quicksort on the two sub-arrays

Quick Sort - Partitioning

You can choose whichever element you want to be your pivot, but for this example we will choose the last element in the array to be our pivot value.

Given this pivot, our partitioning algorithm will rearrange an array around the pivot so that all elements larger than the pivot move after it and all elements smaller than pivot move before it.

Prototype for our partition function: `int partition(int arr[], int start, int end)`

Steps in our partition algorithm:

1. Initialize pivot to last element.
2. Initialize a temp variable to start index .
3. Initialize a variable i to traverse the array
4. Traverse the array while i < end
    - If we encounter an element at arr[temp] that is <= pivot value, swap elements
        - This will ensure that all elements smaller than pivot value are to the left.
        - Increase temp variable.
5. After traversal, the temp variable will be in the correct position of the pivot.
6. Swap the element at arr[pivot] with arr[temp variable]
7. Return the index of pivot

Partitioning

Steps in our partition algorithm:

1. Initialize pivot to last element.
2. Initialize a temp variable to start index .
3. Initialize a variable i to traverse the array
4. Traverse the array while i < end
    - If we encounter an element at arr[temp] that is <= pivot value, swap elements
        - This will ensure that all elements smaller than pivot value are to the left.
        - Increase temp variable.
5. After traversal, the temp variable will be in the correct position of the pivot.
6. Swap the element at arr[pivot] with arr[temp variable]
7. Return the index of pivot

```
/*
 * Function to partition an array.
 * After partitioning, all elements to the left are < pivot
 * and all elements to the right are > pivot
 */
int partition(int arr[], int start, int end){
        //initialize pivot value to last element
        int pivot = arr[end];
        //temporary value to serve as pivot placeholder
        int temp = start;
        //index to traverse array
        int i = start;
        //loop through the whole array
        while(i < end){
                //check if current element is <= pivot
                if(arr[i] <= pivot){
                        //if so, swap with temp pivot
                        swap(arr[i], arr[temp]);
                        temp++;//move temp over
                }
                i++;//say no to infinite loops
        }
        //final swap to place pivot in correct location
        swap(arr[temp], arr[end]);
        return temp;//index of final pivot location
}
```

Does this really work?

Let’s look at a walkthrough of it!

After Partitioning....

- Our partition() function is going to return the index of the pivot to us (*p*)
- All elements to the left of the pivot will be smaller than the pivot.
- All elements to the right of the pivot will be greater than or equal to the pivot.
- Therefore, we can now recursively call Quicksort with our subarrays:

```
quickSort(arr,beg,pivot-1);
quickSort(arr,pivot+1,end);
```

Recursive Implementation of Quicksort Algorithm

Given an array, *A*, of *n* elements : *A[0...n-1]*

- If array/subarray only has <=1 element, stop function (return)
- Else
  - Divide:
    - Pick one element in the array to use as a *pivot*.
    - Partition the elements into two sub-arrays
      - Elements less than or equal to pivot
      - Elements greater than pivot
      - The elements don't have to be in order in the subarrays, they just have to be in the right subarray.
  - Conquer:
    - Recursively call Quicksort on the two sub-arrays

```
/*
 * Recursive quicksort function
 */
void quickSort(int arr[], int beg, int end){
        //base case
        if(beg >= end || beg < 0)
                return;//we're done
        //general case
        //get index of pivot value
        int pivot = partition(arr,beg,end);
        //divide and conquer
        quickSort(arr,beg,pivot-1);//left subarray
        quickSort(arr,pivot+1,end);//right subarray
}

int main()
{
        int array[] = {7,2,1,6,8,5,3,4};
        cout << "Original array: ";
        print(array,8);
        //apply quicksort
        quickSort(array,0,7);//provide first and last index
        cout << "Sorted array after applying Quicksort: ";
        print(array,8);

        return 0;
}
```

Output:

```
Original array: 7 2 1 6 8 5 3 4
Sorted array after applying Quicksort: 1 2 3 4 5 6 7 8
```

Merge Sort Algorithm

The basic idea behind merge sort is to continuously divide a vector of elements into smaller and smaller subvectors, sorting each half and merging them back together.

Given a vector, *V*, of *n* elements : *V[0...n-1]*

- If the vector has fewer than two elements, return.
- Else
  - Divide:
    - Divide the vector subvectors at the midpoint.
  - Conquer:
    - Recursively call mergeSort on each of the subvectors.
  - Combine:
    - Merge the two “sorted” subvectors back into a single sorted vector by taking a new element from either the first or second subvector and choosing the smaller of the elements each time.

Unlike Quicksort where the divide step does a lot of work, for Merge Sort, the combine step does all the heavy lifting.

Note: Quick Sort does not have a “combine” step because everything is done in place. Merge sort utilizes temporary vectors.

Merging

The MVP of Merge Sort is the Merging function, which accepts a vector of the elements to merge, the beginning index first subvector, the end of the first subvector, and the end of the second subvector.

Steps in our Merging Algorithm:

1. Determine the size of the range to be merged and create a temporary vector to hold the merged subvector elements.
2. Determine beginning indices of the first and second subvector
3. While the indices are not past their ranges, move smaller element of both subvectors into the temporary vector.
4. Once one vector's elements have all been copied, copy the remaining elements into the temporary vector.
5. Finally, copy the sorted elements of the temporary vector into the original vector.

Merging

Steps in our Merging Algorithm:

1. Determine the size of the range to be merged and create a temporary vector to hold the merged subvector elements.

2. Determine beginning indices of the first and second subvector.

3. While the indices are not past their ranges, move smaller element of both subvectors into the temporary vector.

4. Once one vector's elements have all been copied, copy the remaining elements into the temporary vector.

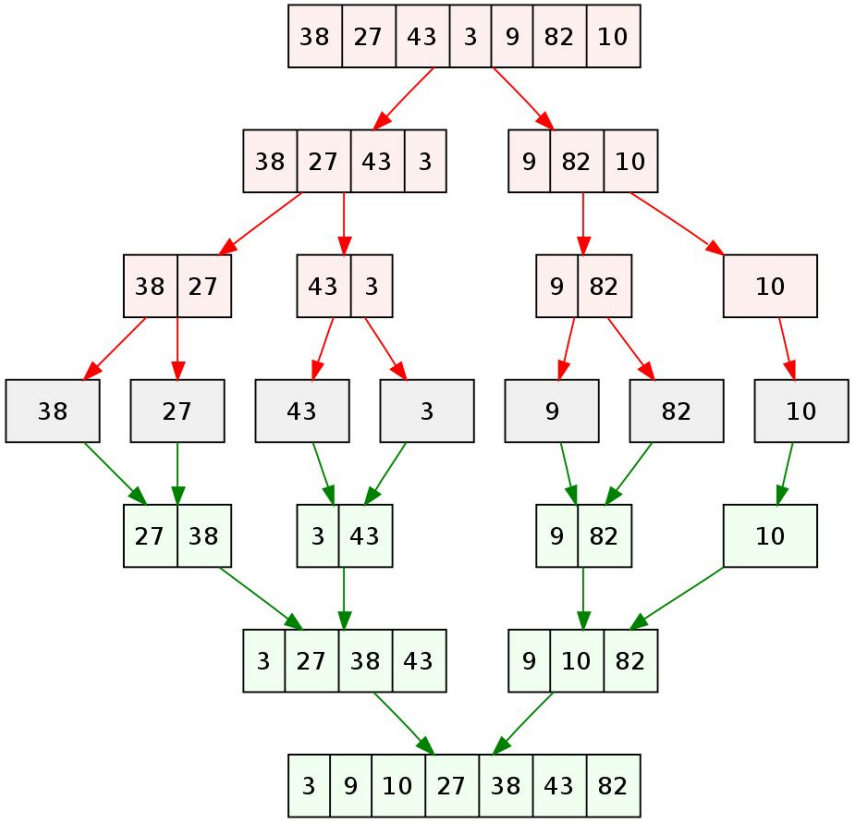
5. Finally, copy the sorted elements of the temporary vector into the original vector.

```
18 * Merge two adjacent ranges in a vector
19 * a - a vector with elements to merge
20 * from - the start of the first range
21 * mid - the end of the first range/start of second
22 * to - the end of the second range
23 */
24 void merge(vector<int> & a, int from, int mid, int to){
25         //Determine size of the range to be merged
26         int n = to - from + 1;
27         //create a temp vector
28         vector<int> b(n);
29         //determine indices of subvectors
30         int sub1 = from;
31         int sub2 = mid + 1;
32         int j = 0;//index of temp vector
33
34         //while loop to traverse through both subvectors
35         while(sub1 <= mid && sub2 <= to){
36                 //copy smallest element into temp vector
37                 if(a[sub1] < a[sub2]){
38                         b[j] = a[sub1];//copy element
39                         sub1++;//increase sub1 index
40                 }
41                 else{
42                         b[j] = a[sub2];//copy element
43                         sub2++;//increase sub2 index
44                 }
45                 j++;//increase temp vector index
46         }
47         //we need to finish copying remaining elements
48         //loop will end once all elements of a subvector
49         //are copied over
50         while(sub1 <= mid){
51                 b[j] = a[sub1];
52                 sub1++;
53                 j++;
54         }
55         //OR
56         while(sub2 <= to){
57                 b[j] = a[sub2];
58                 sub2++;
59                 j++;
60         }
61         //now copy the temp vector to the og vector
62         for(j = 0; j < n; j++)
63                 a[from+j] = b[j];
64 }
```

Merging

Steps in our Merging Algorithm:

1. Determine the size of the range to be merged and create a temporary vector to hold the merged subvector elements.
2. Determine beginning indices of the first and second subvector.
3. While the indices are not past their ranges, move smaller element of both subvectors into the temporary vector.
4. Once one vector's elements have all been copied, copy the remaining elements into the temporary vector.
5. Finally, copy the sorted elements of the temporary vector into the original vector.



https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg

Recursive Implementation of Merge Sort Algorithm

Given a vector, *V*, of *n* elements : *V[0...n-1]*

- If the vector has fewer than two elements, return.
- Else
  - Divide:
    - Divide the vector subvectors at the midpoint.
  - Conquer:
    - Recursively call mergeSort on each of the subvectors.
  - Combine:
    - Merge the two “sorted” subvectors back into a single sorted vector by taking a new element from either the first or second subvector and choosing the smaller of the elements each time.

```
69 void mergeSort(vector<int>& a, int first, int last){
70         //base case
71         if(first == last)
72                 return;//all done
73         //general case
74         //determine the mid point
75         int mid = (first+last)/2;
76         //recursive calls with both subvectors
77         mergeSort(a,first,mid);
78         mergeSort(a,mid+1,last);
79         //merge the subvectors
80         merge(a,first,mid,last);
81 }
82
83 int main()
84 {
85         vector<int> v{38,3,27,43,3,9,82,10};
86         cout << "Original vector: ";
87         print(v);
88         mergeSort(v,0,v.size()-1);
89         cout << "Sorted vector after applying Merge Sort: ";
90         print(v);
91
92         return 0;
93 }
```

Output:

```
Original vector: 38 3 27 43 3 9 82 10
Sorted vector after applying Merge Sort: 3 3 9 10 27 38 43 82
```

Just an FYI: Time Complexity of Sorting and Searching Algorithms

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)

https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/

Binary Search (again)

In Chapter 6, we saw how to implement the Binary Search algorithm on a vector or array.

We learned that Binary Search is WAY faster than Linear Search, and when the number of elements in a container is large enough, and you need to complete multiple searches, it is much more efficient to sort the container (you can use Quick Sort or Merge Sort), and then apply Binary Search!

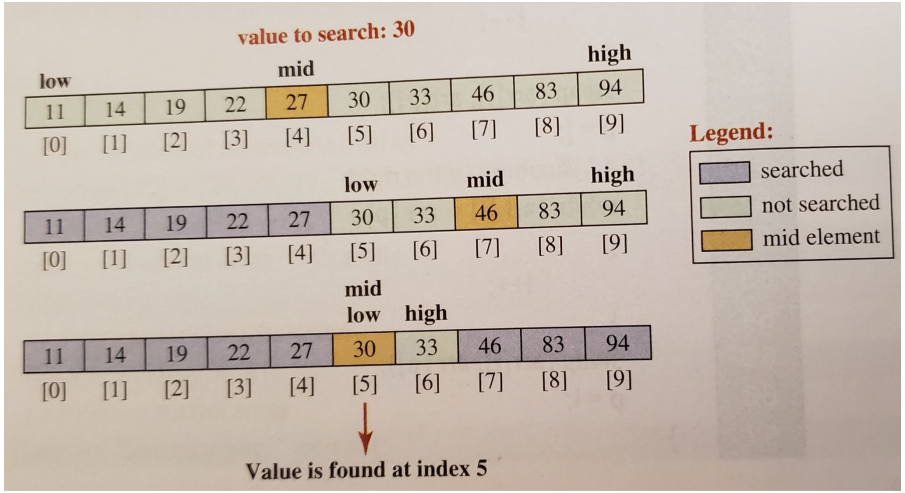
We learned how to do it iteratively, now lets see how we can do it recursively.

Binary Search - Recursive Algorithm

Algorithm:

Given an array, *A*, of *n* elements : *A[0...n-1]*

1. Determine the beginning and end of the search interval.
2. If there is no search interval, return -1 (value not in array)
3. Else
   - Calculate the middle point of search interval.
   - If middle element = value we are searching for, return index.
   - If value > middle element, search continues in right half of array.
   - If value < middle element, search continues in left half of array.
4. Recursively send the next search interval.



Recursive Implementation of Binary Search Algorithm

Given an array, *A*, of *n* elements :
A[0...n-1]

1. Determine the beginning and end of the search interval.
2. If there is not search interval, return -1 (value not in array)
3. Else
   - Calculate the middle point of search interval.
   - If middle element = value we are searching for, return index.
   - If value > middle element, search continues in right half of array.
   - If value < middle element, search continues in left half of array.
4. Recursively send the next search interval.

```cpp
/*
* Filename: binarySearch.cpp
* Recursive implementation of the binary search algorithm
*/
#include <iostream>
using namespace std;

int binarySearch(const int arr[], int lo, int hi, int value){
        //base case if there is no more search interval
        //and we have not found the value
        if(lo > hi)
                return -1;
        //another base case, if we find the value
        int mid = (lo + hi) / 2;
        if(arr[mid] == value)
                return mid;
        //general case, search one half of the array
        if(arr[mid] < value)
                return(binarySearch(arr,mid+1,hi,value));//right half
        else
                return(binarySearch(arr,lo,mid-1,value));//left half
}
int main(){
        int arr[10] = {11,14,19,22,27,30,33,46,83,94};
        int value = 0;//user input
        //prompt the user for a value
        cout << "Enter value to search for:";
        cin >> value;

        //call binarySearch
        int index = binarySearch(arr,0,9,value);
        //output whether the value was found
        if(index > -1)
                cout << "The value WAS found at index: " << index << "\n";
        else
                cout << "The value WAS NOT found :(\n";

        return 0;
}
```

So....what is better: recursion or iteration?

Both...sorta

The efficiency of the method used depends on lots of things: the algorithm, the programming language, the compiler, etc.

Everything that is implemented recursively can be implemented iteratively, BUT it might not be as “elegant” or easily understood.

Recursion generally uses more memory and takes longer since it continuously making calls to memory.

Examples of algorithms where ***iteration*** is more efficient than recursion:

- Fibonacci Numbers
- Factorial Calculation

But many times, the recursive and iterative methods are very similar in performance.

In conclusion:

- Use whichever method makes most sense to YOU.
- Recursion is strongly related to mathematical induction, which is used in proving the time complexity of algorithms (Remember this in CSE 4310).
- *“To iterate is human, to recurse is divine”* - L. Peter Deutsch (Computer Scientist)