Data Structures & Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 1

Complexity Analysis & Recursion

Course Roadmap

Part 1: Linear Data Structures

Lecture 1: Complexity Analysis & Recursion

Lecture 2: Arrays

Lecture 3: Linked List

Lecture 4: Stack

Lecture 5: Queue

Lecture 6: Deque

Lecture 7: STL in C++ (Linear Data Structures)

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Data Structures

2- Execution Time Cases

3- Complexity Analysis Examples

4- Recursion

5- Iteration vs. Recursion Examples


Let's
START UP

Lecture Agenda

Section 1: Introduction to Data Structures

Section 2: Execution Time Cases

Section 3: Complexity Analysis Examples

Section 4: Recursion

Section 5: Iteration vs. Recursion Examples

Introduction to Data Structures

- Data structure is a way to store and organize data in order to support efficient insertions, queries, searches, updates, and deletions. Although a data structure in itself does not solve the given programming problem, the algorithm operating on it does, using the most efficient data structure for the given problem may be a difference between passing or exceeding the problem's time limit. There are many ways to organize the same data and sometimes one way is better than the other on different context.
- Characteristics of Data Structure:

1 - Correctness: Data structure implementation should implement its interface correctly.

2 - Time Complexity: Running time or the execution time of operations of data structure must be as small as possible.

3 - Space Complexity: Memory usage of a data structure operation should be as little as possible.

- The foundation terms of a data structure:

1- Interface: It represents the set of operations that a data structure supports.

2- Implementation: It provides the internal representation of a data structure.

Introduction to Data Structures

- **Why we need data structures?** there are several advantages of using them, few of them are as follows:

1. Data Organization: We need a proper way of organizing the data so that it can accessed efficiently when we need that particular data. DS provides different ways of data organization so we have options to store the data in different data structures based on the requirement.

2. Efficiency: The main reason we organize the data is to improve the efficiency. We can store the data in arrays then why do we need linked lists and other data structures? because when we need to perform several operation such as add, delete update and search on arrays , it takes more time in arrays than some of the other data structures. So the fact that we are interested in other data structures is because of the efficiency.

- **Time Complexity:** It is a way to represent the amount of time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible. We will study about Time Complexity in details in later sections.
- **Space Complexity:** Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Introduction to Data Structures

Introduction to Data Structures

Lecture Agenda

- ✓ Section 1: Introduction to Data Structures
- **Section 2: Execution Time Cases**
- Section 3: Complexity Analysis Examples
- Section 4: Recursion
- Section 5: Iteration vs. Recursion Examples

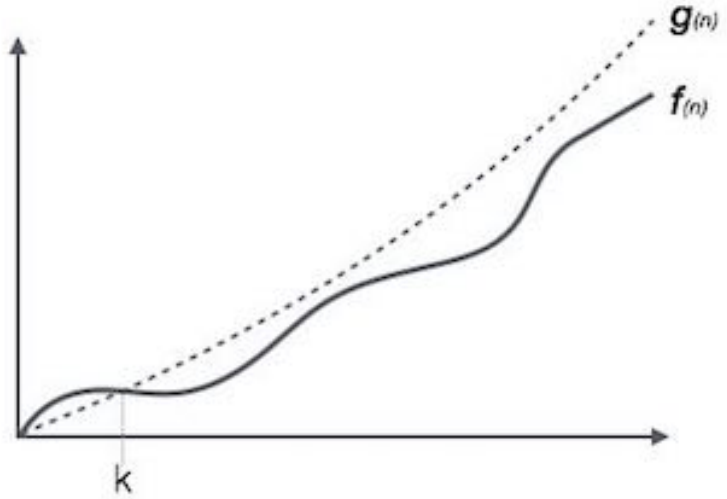
Execution Time Cases

- The running time of an algorithm depends on how long it takes a computer to run the statements of code of the algorithm and that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, among other factors.
- We can use a combination of two ideas. First, we need to determine how long the algorithm takes, in terms of the size of its input. This idea makes intuitive sense, doesn't it? We've already seen that the maximum number of guesses in two algorithms increases as the length of the array increases. Or think about a GPS. If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right? So we think about the running time of the algorithm as a function of the size of its input.
- There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- Best Case: Minimum time required for program execution. (Ω Notation)
- Average Case: Average time required for program execution. (Θ Notation)
- Worst Case: Maximum time required for program execution. (Big O Notation)

Worst Case (Big O Notation)

- The notation **O**(n) is the formal way to express the upper bound of an algorithm's running time.
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
- The Big O notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation.
- For a given function g(n), we denote by O(g(n)) (pronounced "big O of g of n" or sometimes just "O of g of n") the set of functions.

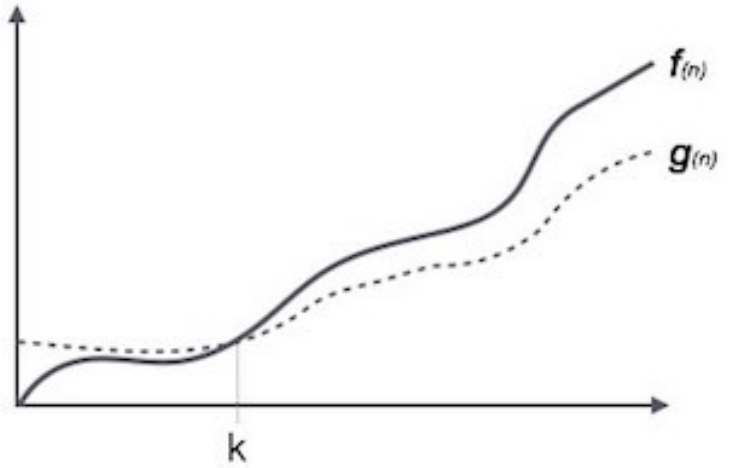


- For example: for a function f(n)
- **O**(f(n)) = { g(n) : there exists C > 0 and k such that

  $g(n) \leq c * f(n)$ for all n > k }

Best Case (Ω Notation)

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
- Ω notation provides an asymptotic lower bound. For a given function g(n), we denote by Ω (g(n))
- For a given function g(n), we denote by O(g(n)) (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions.

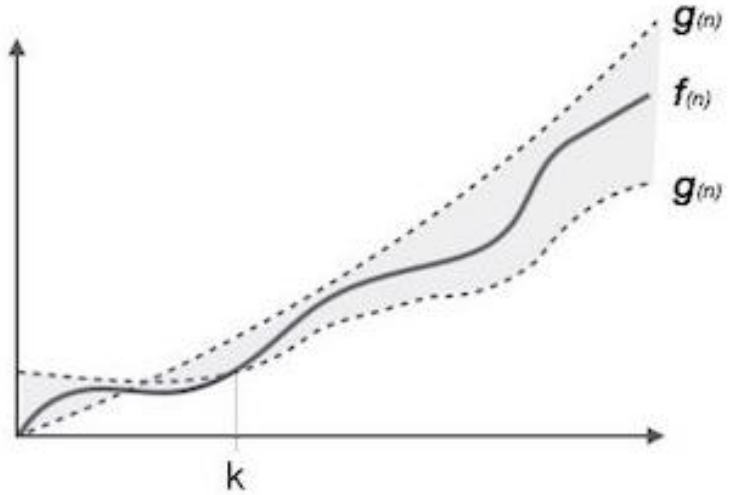


- For example: for a function f(n)
- Ω(f(n)) ≥ { g(n) : there exists c › 0 and k such that

  g(n) ≤ c * f(n) for all n › k }

Average Case (θ Notation)

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.



- For example: for a function f(n)
- θ(f(n)) = { g(n)

  if and only if g(n) = O(f(n)) and g(n) = Ω(f(n))

  for all n > k }

Common Asymptotic Notation

- Sort the following functions in ascending order:

  $O(n^3)$

  $O(\log n)$

  $O(n)$

  $n^{O(1)}$

  $O(1)$

  $O(n^2)$

  $O(n \log n)$

  $O(2^n)$

  $O(\sqrt{n})$

Common Asymptotic Notation

- Sort the following functions in ascending order: Assume $n = 10^3$

$O(n^3)$	10^9	
$O(\log n)$	10^1	
$O(n)$	10^3	
$n^{O(1)}$	$(10^3)^k$	such that k › 1
$O(1)$	1	
$O(n^2)$	10^6	
$O(n \log n)$	10^4	
$O(2^n)$	2^{1000}	
$O(\sqrt{n})$	32	

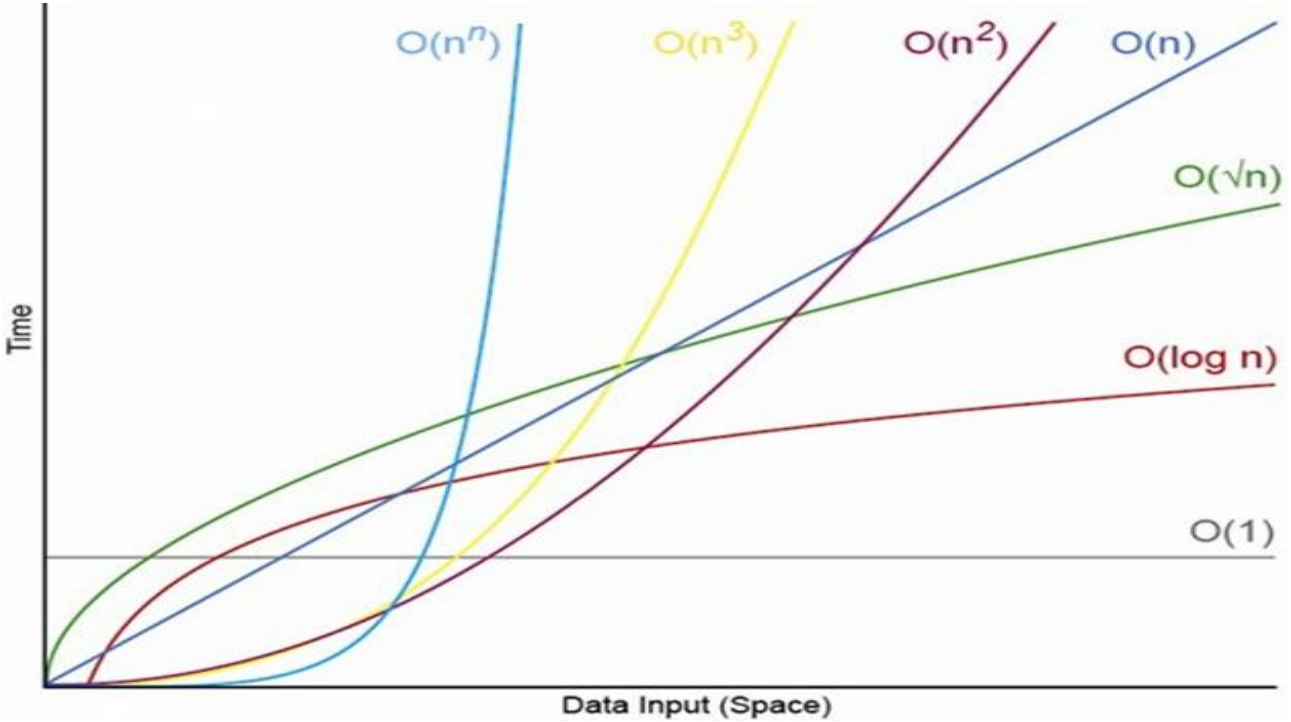
Common Asymptotic Notation

- Following is a list of some common asymptotic notations:

Constant	=>	$O(1)$
Logarithmic	=>	$O(\log n)$
Square root	=>	$O(\sqrt{n})$
Linear	=>	$O(n)$
$n * \log n$	=>	$O(n \log n)$
Quadratic	=>	$O(n^2)$
Cubic	=>	$O(n^3)$
Polynomial	=>	$n^{O(1)}$
Exponential	=>	$O(2^n)$

Common Asymptotic Notation

- This graph show behavior of each function



Lecture Agenda

- ✔ Section 1: Introduction to Data Structures
- ✔ Section 2: Execution Time Cases
- **Section 3: Complexity Analysis Examples**
- Section 4: Recursion
- Section 5: Iteration vs. Recursion Examples

Complexity Analysis Examples

Example 1

```
void func() {
    int i = 1;
    while (i <= 100) {
        int j = 1;
        while (j <= 100) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Example 2

```
void func() {
    int i = 1;
    while (i <= 100) {
        cout << "*";
        i += 1;
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 1

```
void func() {
    int i = 1;
    while (i <= 100) {
        int j = 1;
        while (j <= 100) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O(1)}$

Example 2

```
void func() {
    int i = 1;
    while (i <= 100) {
        cout << "*";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O(1)}$

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 3

```
void func(int n) {
    int i = 1;
    while (i * i <= n) {
        cout << "*";
        i += 1;
    }
}
```

Time Complexity: ??

Example 4

```
void func(int n) {
    int i = 1, s = 1;
    while (s <= n) {
        i += 1;
        s += i;
        cout << "*";
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 3

```
void func(int n) {
    int i = 1;
    while (i * i <= n) {
        cout << "*";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(\sqrt{n})$

Example 4

```
void func(int n) {
    int i = 1, s = 1;
    while (s <= n) {
        i += 1;
        s += i;
        cout << "*";
    }
}
```

Time Complexity: $\mathbf{O}(\sqrt{n})$

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 5

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        cout << "*";
        i *= 2;
    }
}
```

Time Complexity: ??

Example 6

```
void func(int n) {
    int i = 1;
    while (i * i <= n) {
        int j = 1;
        while (j <= n) {
            cout << "*";
            j *= 2;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 5

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        cout << "*";
        i *= 2;
    }
}
```

Time Complexity: $\mathbf{O}(log\ n)$

Example 6

```
void func(int n) {
    int i = 1;
    while (i * i <= n) {
        int j = 1;
        while (j <= n) {
            cout << "*";
            j *= 2;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(\sqrt{n} * log\ n)$

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 7

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        cout << "*";
        i += 1;
    }
}
```

Time Complexity: ??

Example 8

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= 100) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 7

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        cout << "*";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Example 8

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= 100) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 9

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            cout << "*";
            j *= 2;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Example 10

```
void func(int n) {
    int i = n / 2;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            int k = 1;
            while (k <= n) {
                cout << "*";
                k *= 2;
            }
            j *= 2;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 9

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            cout << "*";
            j *= 2;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $O(n * log\ n)$

Example 10

```
void func(int n) {
    int i = n / 2;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            int k = 1;
            while (k <= n) {
                cout << "*";
                k *= 2;
            }
            j *= 2;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $O(n * log^2\ n)$

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 11

```cpp
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Example 12

```cpp
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= i) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 11

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n^2)$

Example 12

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= i) {
            cout << "*";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n^2)$

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 13

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= i) {
            int k = 1;
            while (k <= 100) {
                cout << "*";
                k += 1;
            }
            cout << "\n";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Example 14

```
void func(int n) {
    int i = n / 2;
    while (i <= n) {
        int j = 1;
        while (j <= n / 2) {
            int k = 1;
            while (k <= n) {
                cout << "*";
                k *= 2;
            }
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 13

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= i) {
            int k = 1;
            while (k <= 100) {
                cout << "*";
                k += 1;
            }
            cout << "\n";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n^2)$

Example 14

```
void func(int n) {
    int i = n / 2;
    while (i <= n) {
        int j = 1;
        while (j <= n / 2) {
            int k = 1;
            while (k <= n) {
                cout << "*";
                k *= 2;
            }
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n^2 * log\ n)$

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 15

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            int k = 1;
            while (k <= n) {
                cout << "*";
                k += 1;
            }
            cout << "\n";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Example 16

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= i) {
            int k = 1;
            while (k <= j) {
                cout << "*";
                k += 1;
            }
            cout << "\n";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: ??

Complexity-Analysis.cpp

Complexity Analysis Examples

Example 15

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= n) {
            int k = 1;
            while (k <= n) {
                cout << "*";
                k += 1;
            }
            cout << "\n";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n^3)$

Example 16

```
void func(int n) {
    int i = 1;
    while (i <= n) {
        int j = 1;
        while (j <= i) {
            int k = 1;
            while (k <= j) {
                cout << "*";
                k += 1;
            }
            cout << "\n";
            j += 1;
        }
        cout << "\n";
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n^3)$

Complexity-Analysis.cpp

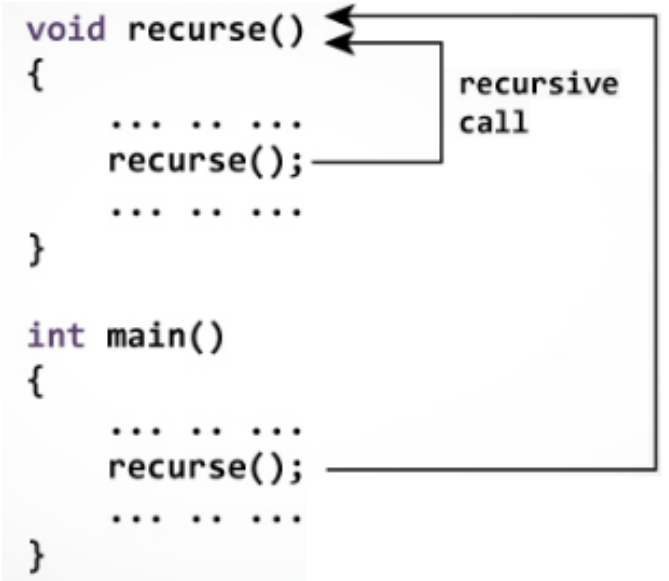
Lecture Agenda

- ✔ Section 1: Introduction to Data Structures
- ✔ Section 2: Execution Time Cases
- ✔ Section 3: Complexity Analysis Examples
- **Section 4: Recursion**
- Section 5: Iteration vs. Recursion Examples

Recursion

- **What is Recursion?** It is the process of repeating items in a self-similar way. The process in which a function calls itself is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.
- **What is base condition in recursion?** In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems. The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion.
- **How memory is allocated to different function calls in recursion?**

• When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.



Recursion

- What are the disadvantages of recursive programming over iterative programming?

- Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

- What are the advantages of recursive programming over iterative programming?

- Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi.

- Advantages of Recursion:

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.

- Disadvantages of Recursion:

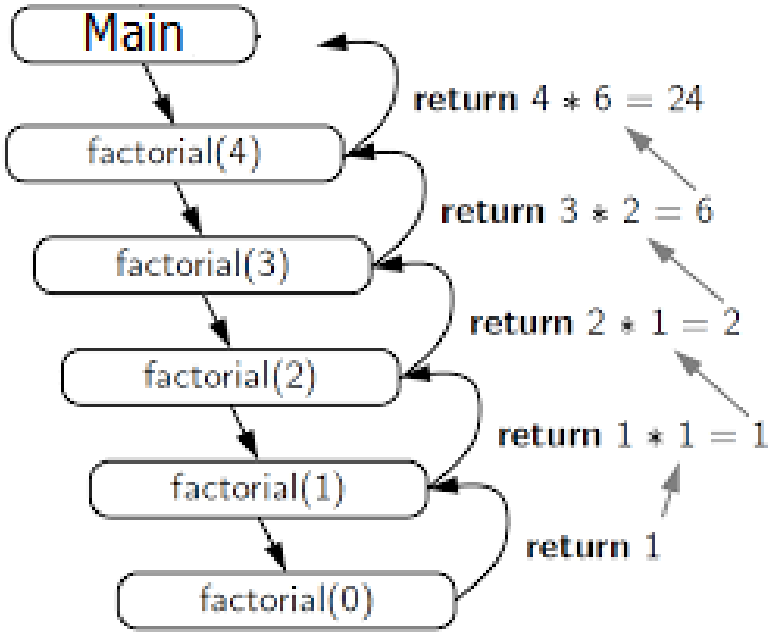
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Recursion Examples

- Implement functions which calculate the factorial of n.

- Recursive Solution

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



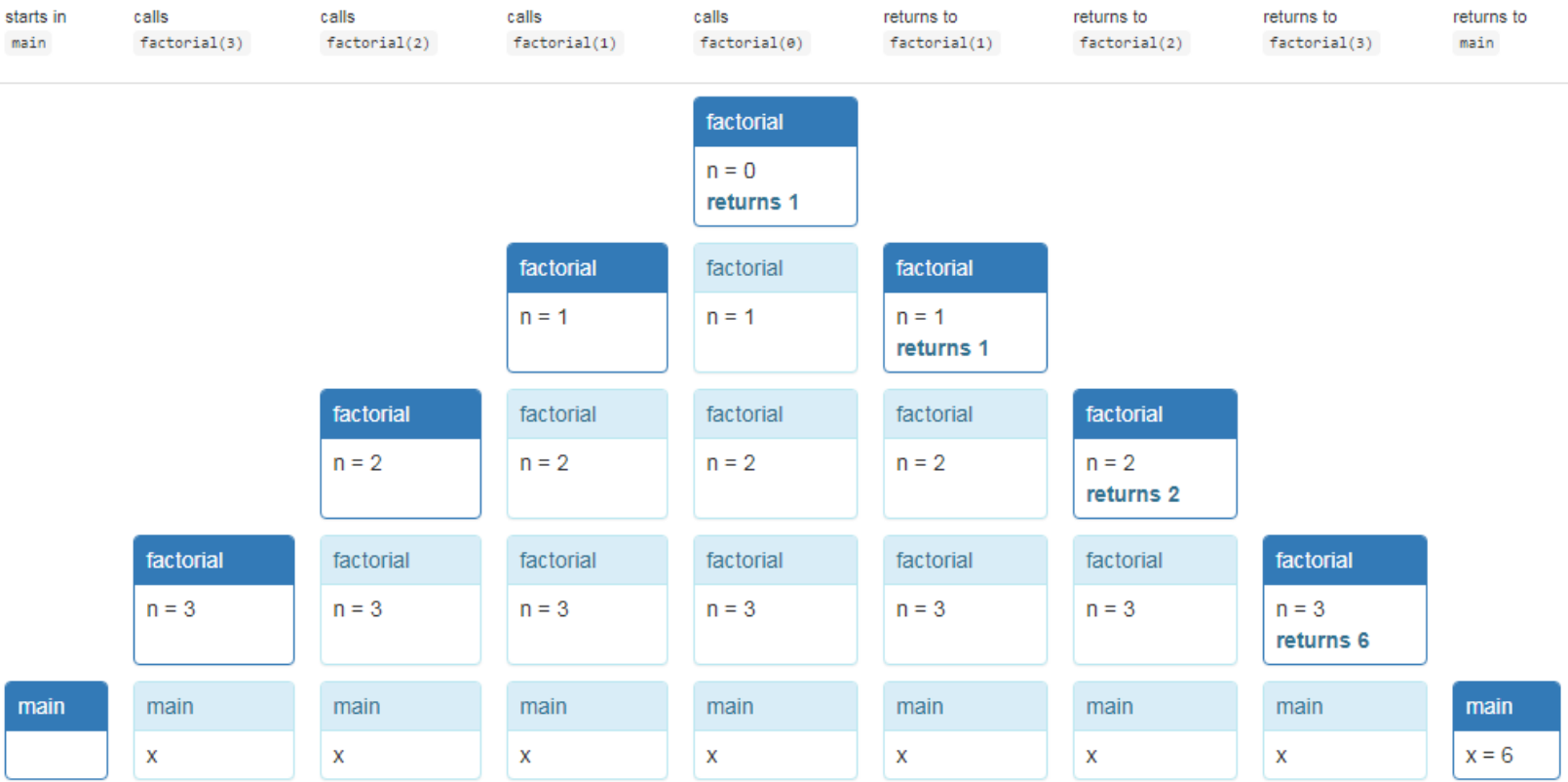
Recursive Definition: fact(n) = fact(n-1) * n

Base Case: fact(0) = 1

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp

Recursion Examples in Memory

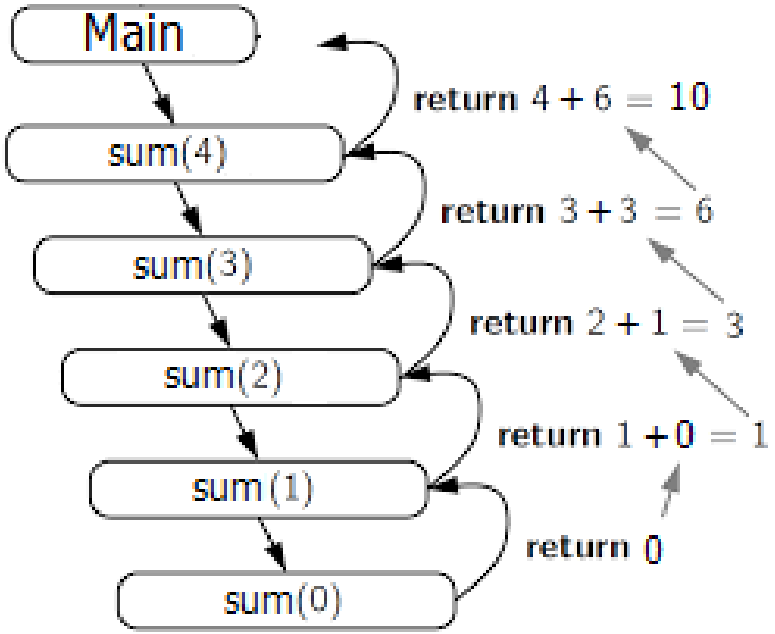


Recursion Examples

- Implement functions which calculate the summation of numbers form 1 to n.

- Recursive Solution

```
int calc_sum(int i) {
    if (i == 0)
        return 0;
    return calc_sum(i-1) + i;
}
```



Recursive Definition: sum(n) = sum(n-1) + n

Base Case: sum(0) = 0

Time Complexity: $\mathbf{O}(n)$

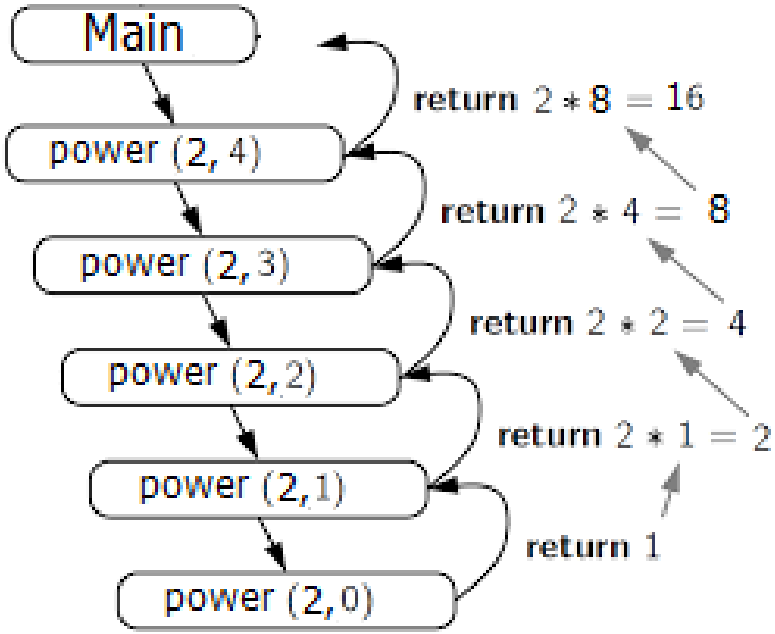
Recursion.cpp

Recursion Examples

- Implement functions which calculate the power operation of b^e.

- Recursive Solution

```
float power(float b, int e) {
    if (e == 0)
        return 1.0;
    return power(b, e-1) * b;
}
```



Recursive Definition:

power(b, e) = power(b, e-1) * b

Base Case: power(b, 0) = 1

Time Complexity: $O(n)$

Recursion.cpp

Recursion Examples

- Implement functions which check if the given number is a prime.

- Recursive Solution

```
bool is_prime(int i, int x) {
    if (i == x)
        return true;
    if (x % i == 0)
        return false;
    return is_prime(i+1, x);
}
```

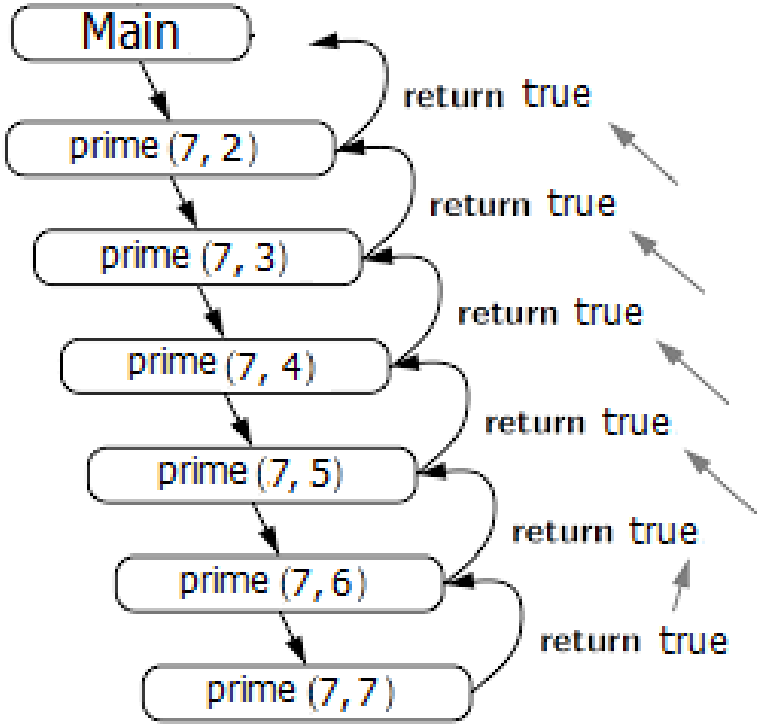
Number 7 is Prime

1 2 3 4 5 6 7

Number 10 is not Prime

1 2 3 4 5 6 7 8 9 10

Time Complexity: $\mathbf{O}(n)$



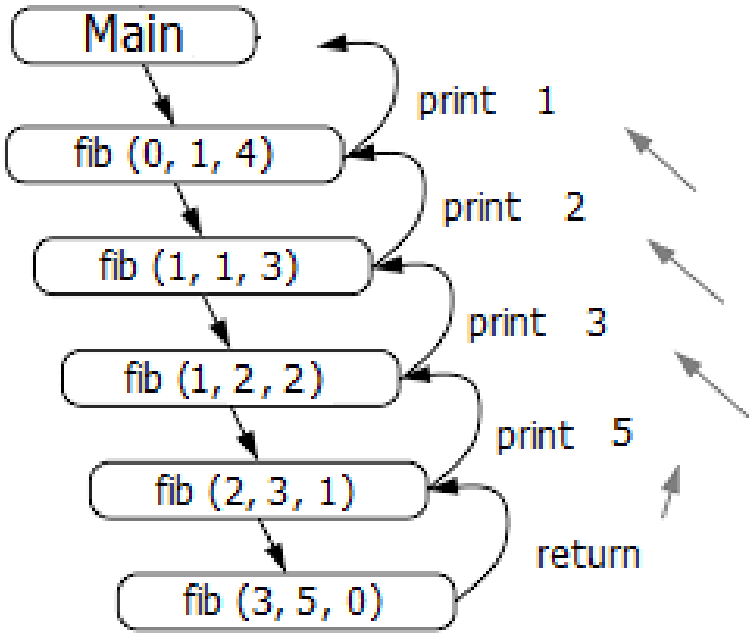
Recursion.cpp

Recursion Examples

- Implement functions which calculate the first n numbers of Fibonacci series.

- Recursive Solution

```
void fib(int x, int y, int i) {
    if (i == 0)
        return;
    cout << x + y << ' ';
    fib(y, x+y, i-1);
}
```



Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

Recursive Definition: fib(x, y) = fib(y, x + y)

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp

Functionality Testing

- In the Main function:

```
cout << "sum 1 to 4   : " << calc_sum(4) << '\n';
cout << "power 2^10   : " << power(2, 10) << '\n';
cout << "factorial 6 : " << factorial(6) << '\n';
cout << "fibonacci 6 : ";
fib(0, 1, 6);
cout << '\n';
cout << "15 is prime : " << is_prime(2, 15) << '\n';
cout << "17 is prime : " << is_prime(2, 17) << '\n';
```

- Expected Output:

```
sum 1 to 4   : 10
power 2^10   : 1024
factorial 6 : 720
fibonacci 6 : 1 2 3 5 8 13
15 is prime : 0
17 is prime : 1
```

Lecture Agenda

- ✓ Section 1: Introduction to Data Structures
- ✓ Section 2: Execution Time Cases
- ✓ Section 3: Complexity Analysis Examples
- ✓ Section 4: Recursion
- Section 5: Iteration vs. Recursion Examples

Iteration vs. Recursion Examples

- Implement functions which calculate the factorial of n.

- Recursive Solution

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

- Iterative Solution

```
int factorial(int n) {
    int res = 1;
    while (n > 0) {
        res *= n;
        n -= 1;
    }
    return res;
}
```

Time Complexity: $O(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which calculate the summation of numbers form 1 to n.

- Recursive Solution

```
int calc_sum(int i) {
    if (i == 0)
        return 0;
    return calc_sum(i-1) + i;
}
```

- Iterative Solution

```
int calc_sum(int n) {
    int res = 0;
    while (n > 0) {
        res += n;
        n -= 1;
    }
    return res;
}
```

Recursion.cpp
Iterative.cpp

Time Complexity: $\mathbf{O}(n)$

Iteration vs. Recursion Examples

- Implement functions which calculate the power operation of b^e.

- Recursive Solution

```
float power(float b, int e) {
    if (e == 0)
        return 1.0;
    return power(b, e-1) * b;
}
```

- Iterative Solution

```
float power(float b, int e) {
    int res = 1;
    while (e > 0) {
        res *= b;
        e -= 1;
    }
    return res;
}
```

Time Complexity: $O(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which check if the given number is a prime.

- Recursive Solution

```
bool is_prime(int i, int x) {
    if (i == x)
        return true;
    if (x % i == 0)
        return false;
    return is_prime(i+1, x);
}
```

- Iterative Solution

```
bool is_prime(int i, int n) {
    while (i < n) {
        if (n % i == 0)
            return false;
        i += 1;
    }
    return true;
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which calculate the first n numbers of Fibonacci series.

- Recursive Solution

```
void fib(int x, int y, int i) {
    if (i == 0)
        return;
    cout << x + y << ' ';
    fib(y, x+y, i-1);
}
```

- Iterative Solution

```
void fib(int x, int y, int n) {
    while (n > 0) {
        cout << x + y << ' ';
        int z = x;
        x = y;
        y = y + z;
        n -= 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Functionality Testing

- In the Main function:

```
cout << "sum 1 to 4   : " << calc_sum(4) << '\n';
cout << "power 2^10   : " << power(2, 10) << '\n';
cout << "factorial 6 : " << factorial(6) << '\n';
cout << "fibonacci 6 : ";
fib(0, 1, 6);
cout << '\n';
cout << "15 is prime : " << is_prime(2, 15) << '\n';
cout << "17 is prime : " << is_prime(2, 17) << '\n';
```

- Expected Output:

```
sum 1 to 4   : 10
power 2^10   : 1024
factorial 6 : 720
fibonacci 6 : 1 2 3 5 8 13
15 is prime : 0
17 is prime : 1
```

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which print the global array items in forward and backward directions.

- Recursive Solution

```
void print_array(int i) {
    if (i == n)
        return;
    cout << arr[i] << ' ';
    print_array(i+1);
}
```

```
void print_array_reverse(int i) {
    if (i == -1)
        return;
    cout << arr[i] << ' ';
    print_array_reverse(i-1);
}
```

- Iterative Solution

```
void print_array(int i) {
    while (i < n) {
        cout << arr[i] << ' ';
        i += 1;
    }
}
```

```
void print_array_reverse(int i) {
    while (i >= 0) {
        cout << arr[i] << ' ';
        i -= 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which print even numbers and odd numbers in a global array.

- Recursive Solution

```
void print_evens(int i, int n) {
    if (i == n)
        return;
    if (arr[i] % 2 == 0)
        cout << arr[i] << ' ';
    print_evens(i+1, n);
}

void print_odds(int i, int n) {
    if (i == n)
        return;
    if (arr[i] % 2 != 0)
        cout << arr[i] << ' ';
    print_odds(i+1, n);
}
```

- Iterative Solution

```
void print_evens(int i, int n) {
    while (i < n) {
        if (arr[i] % 2 == 0)
            cout << arr[i] << ' ';
        i += 1;
    }
}

void print_odds(int i, int n) {
    while (i < n) {
        if (arr[i] % 2 != 0)
            cout << arr[i] << ' ';
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which count even numbers and odd numbers in a global array.

- Recursive Solution

```
int count_even(int i, int n) {
    if (i == n)
        return 0;
    return count_even(i+1, n) + (arr[i] % 2 == 0);
}

int count_odd(int i, int n) {
    if (i == n)
        return 0;
    return count_odd(i+1, n) + (arr[i] % 2 != 0);
}
```

- Iterative Solution

```
int count_even(int i, int n) {
    int res = 0;
    while (i < n) {
        res += (arr[i] % 2 == 0);
        i += 1;
    }
    return res;
}

int count_odd(int i, int n) {
    int res = 0;
    while (i < n) {
        res += (arr[i] % 2 != 0);
        i += 1;
    }
    return res;
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which print positive numbers and negative numbers in a global array.

- Recursive Solution

```
void print_positive(int i, int n) {
    if (i == n)
        return;
    if (arr[i] > 0)
        cout << arr[i] << ' ';
    print_positive(i+1, n);
}

void print_negative(int i, int n) {
    if (i == n)
        return;
    if (arr[i] < 0)
        cout << arr[i] << ' ';
    print_negative(i+1, n);
}
```

- Iterative Solution

```
void print_positive(int i, int n) {
    while (i < n) {
        if (arr[i] > 0)
            cout << arr[i] << ' ';
        i += 1;
    }
}

void print_negative(int i, int n) {
    while (i < n) {
        if (arr[i] < 0)
            cout << arr[i] << ' ';
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which count positive numbers and negative numbers in a global array.

- Recursive Solution

```
int count_positive(int i, int n) {
    if (i == n)
        return 0;
    return count_positive(i+1, n) + (arr[i] > 0);
}
```

```
int count_negative(int i, int n) {
    if (i == n)
        return 0;
    return count_negative(i+1, n) + (arr[i] < 0);
}
```

Time Complexity: $\mathbf{O}(n)$

- Iterative Solution

```
int count_positive(int i, int n) {
    int res = 0;
    while (i < n) {
        res += (arr[i] > 0);
        i += 1;
    }
    return res;
}
```

```
int count_negative(int i, int n) {
    int res = 0;
    while (i < n) {
        res += (arr[i] < 0);
        i += 1;
    }
    return res;
}
```

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which find the min value and the max value in a global array.

- Recursive Solution

```
int find_min_array(int i, int n) {
    if (i == n)
        return 2e9;
    return min(find_min_array(i+1, n), arr[i]);
}

int find_max_array(int i, int n) {
    if (i == n)
        return -2e9;
    return max(find_max_array(i+1, n), arr[i]);
}
```

- Iterative Solution

```
int find_min_array(int i, int n) {
    int res = 2e9;
    while (i < n) {
        res = min(res, arr[i]);
        i += 1;
    }
    return res;
}
int find_max_array(int i, int n) {
    int res = -2e9;
    while (i < n) {
        res = max(res, arr[i]);
        i += 1;
    }
    return res;
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which find a given item in a global array.
- Implement functions which reverse a global array.

- Recursive Solution

```
bool find_item_array(int i, int n, int k) {
    if (i == n)
        return false;
    if (arr[i] == k)
        return true;
    return find_item_array(i+1, n, k);
}

void reverse_array(int i, int j) {
    if (i > j)
        return;
    swap(arr[i], arr[j]);
    reverse_array(i+1, j-1);
}
```

Time Complexity: $\mathbf{O}(n)$

- Iterative Solution

```
bool find_item_array(int i, int n, int k) {
    while (i < n) {
        if (arr[i] == k)
            return true;
        i += 1;
    }
    return false;
}
void reverse_array(int i, int j) {
    while (i < j) {
        swap(arr[i], arr[j]);
        i += 1;
        j -= 1;
    }
}
```

Recursion.cpp
Iterative.cpp

Functionality Testing

- Initialize a global array

```
const int n = 10;
int arr[n] = {7, 4, -5, 2, -10, 3, -12, -17, 6, 13};
```

- In the Main function:

```
cout << "array items : \n";
print_array(0);
cout << '\n';
print_array_reverse(n-1);
cout << '\n';
cout << "array reversing ... \n";
reverse_array(0, n-1);
cout << "array items : \n";
print_array(0);
cout << '\n';
print_array_reverse(n-1);
cout << '\n';
```

- Expected Output:

```
array items :
7 4 -5 2 -10 3 -12 -17 6 13
13 6 -17 -12 3 -10 2 -5 4 7
array reversing ...
array items :
13 6 -17 -12 3 -10 2 -5 4 7
7 4 -5 2 -10 3 -12 -17 6 13
```

Recursion.cpp
Iterative.cpp

Functionality Testing

- In the Main function:

```
cout << "count evens : " << count_even(0, n) << '\n';
cout << "count odds  : " << count_odd(0, n) << '\n';
cout << "array even items : ";
print_evens(0, n);
cout << '\n';
cout << "array odd items  : ";
print_odds(0, n);
cout << '\n';
```

- Expected Output:

```
count evens : 5
count odds  : 5
array even items : 6 -12 -10 2 4
array odd items  : 13 -17 3 -5 7
```

Recursion.cpp
Iterative.cpp

Functionality Testing

- In the Main function:

```
cout << "count positive : " << count_positive(0, n) << '\n';
cout << "count negative : " << count_negative(0, n) << '\n';
cout << "array positive items : ";
print_positive(0, n);
cout << '\n';
cout << "array negative items : ";
print_negative(0, n);
cout << '\n';
```

- Expected Output:

```
count positive : 6
count negative : 4
array positive items : 13 6 3 2 4 7
array negative items : -17 -12 -10 -5
```

Recursion.cpp
Iterative.cpp

Functionality Testing

- In the Main function:

```
cout << "find max array : " << find_max_array(0, n) << '\n';
cout << "find min array : " << find_min_array(0, n) << '\n';
cout << "find 4 in array  : " << find_item_array(0, n, 4) << '\n';
cout << "find 5 in array  : " << find_item_array(0, n, 5) << '\n';
cout << "find -4 in array : " << find_item_array(0, n, -4) << '\n';
cout << "find -5 in array : " << find_item_array(0, n, -5) << '\n';
```

- Expected Output:

```
find max array : 13
find min array : -17
find 4 in array  : 1
find 5 in array  : 0
find -4 in array : 0
find -5 in array : 1
```

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which print the global string chars in forward and backward directions.

- Recursive Solution

```
void print_str(int i) {
    if (i == str.size())
        return;
    cout << str[i] << ' ';
    print_str(i+1);
}

void print_str_reverse(int i) {
    if (i == -1)
        return;
    cout << str[i] << ' ';
    print_str_reverse(i-1);
}
```

- Iterative Solution

```
void print_str(int i) {
    while (i < str.size()) {
        cout << str[i] << ' ';
        i += 1;
    }
}

void print_str_reverse(int i) {
    while (i >= 0) {
        cout << str[i] << ' ';
        i -= 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which print small chars and capital chars in a global string.

- Recursive Solution

```
void print_smalls(int i) {
    if (i == str.size())
        return;
    if ('a' <= str[i] && str[i] <= 'z')
        cout << str[i] << ' ';
    print_smalls(i+1);
}
void print_capitals(int i) {
    if (i == str.size())
        return;
    if ('A' <= str[i] && str[i] <= 'Z')
        cout << str[i] << ' ';
    print_capitals(i+1);
}
```

- Iterative Solution

```
void print_smalls(int i) {
    while (i < str.size()) {
        if ('a' <= str[i] && str[i] <= 'z')
            cout << str[i] << ' ';
        i += 1;
    }
}

void print_capitals(int i) {
    while (i < str.size()) {
        if ('A' <= str[i] && str[i] <= 'Z')
            cout << str[i] << ' ';
        i += 1;
    }
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which count small chars and capital chars in a global string.

- Recursive Solution

```
int count_smalls(int i) {
    if (i == str.size())
        return 0;
    return count_smalls(i+1) +
           ('a' <= str[i] && str[i] <= 'z');
}

int count_capitals(int i) {
    if (i == str.size())
        return 0;
    return count_capitals(i+1) +
           ('A' <= str[i] && str[i] <= 'Z');
}
```

Time Complexity: $\mathbf{O}(n)$

- Iterative Solution

```
int count_smalls(int i) {
    int res = 0;
    while (i < str.size()) {
        if ('a' <= str[i] && str[i] <= 'z')
            res += 1;
        i += 1;
    }
    return res;
}

int count_capitals(int i) {
    int res = 0;
    while (i < str.size()) {
        if ('A' <= str[i] && str[i] <= 'Z')
            res += 1;
        i += 1;
    }
    return res;
}
```

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which find the min value and the max value in a global string.

- Recursive Solution

```
char find_min_str(int i) {
    if (i == n-1)
        return str[i];
    return min(find_min_str(i+1), str[i]);
}

char find_max_str(int i) {
    if (i == n-1)
        return str[i];
    return max(find_max_str(i+1), str[i]);
}
```

- Iterative Solution

```
char find_min_str(int i) {
    char res = str[n-1];
    while (i < n-1) {
        res = min(res, str[i]);
        i += 1;
    }
    return res;
}
char find_max_str(int i) {
    char res = str[n-1];
    while (i < n-1) {
        res = max(res, str[i]);
        i += 1;
    }
    return res;
}
```

Time Complexity: $\mathbf{O}(n)$

Recursion.cpp
Iterative.cpp

Iteration vs. Recursion Examples

- Implement functions which find a given item in a global string.
- Implement functions which reverse a global string.

- Recursive Solution

```
bool find_item_string(int i, char k) {
    if (i == str.size())
        return false;
    if (str[i] == k)
        return true;
    return find_item_string(i+1, k);
}

void reverse_str(int i, int j) {
    if (i > j)
        return;
    swap(str[i], str[j]);
    reverse_str(i+1, j-1);
}
```

Time Complexity: $\mathbf{O}(n)$

- Iterative Solution

```
bool find_item_string(int i, char k) {
    while (i < n) {
        if (str[i] == k)
            return true;
        i += 1;
    }
    return false;
}
void reverse_str(int i, int j) {
    while (i < j) {
        swap(str[i], str[j]);
        i += 1;
        j -= 1;
    }
}
```

Recursion.cpp
Iterative.cpp

Functionality Testing

- Initialize a global string

```
string str = "abCdeFghIjkL";
```

- In the Main function:

```
cout << "string items : \n";
print_str(0);
cout << '\n';
print_str_reverse(n-1);
cout << '\n';
cout << "string reversing ... \n";
reverse_str(0, n-1);
cout << "string items : \n";
print_str(0);
cout << '\n';
print_str_reverse(n-1);
cout << '\n';
```

- Expected Output:

```
string items :
a b C d e F g h I j k L
j I h g F e d C b a
string reversing ...
string items :
j I h g F e d C b a k L
a b C d e F g h I j
```

Recursion.cpp
Iterative.cpp

Functionality Testing

- In the Main function:

```
cout << "count smalls   : " << count_smalls(0) << '\n';
cout << "count capitals : " << count_capitals(0) << '\n';
cout << "string small items   : ";
print_smalls(0);
cout << '\n';
cout << "string capital items : ";
print_capitals(0);
cout << '\n';
```

- Expected Output:

```
count smalls   : 8
count capitals : 4
string small items   : j h g e d b a k
string capital items : I F C L
```

Recursion.cpp
Iterative.cpp

Functionality Testing

- In the Main function:

```
cout << "find max string : "  << find_max_str(0) << '\n';
cout << "find max string : "  << find_min_str(0) << '\n';
cout << "find a in string : " << find_item_string(0, 'a') << '\n';
cout << "find f in string : " << find_item_string(0, 'f') << '\n';
cout << "find A in string : " << find_item_string(0, 'A') << '\n';
cout << "find F in string : " << find_item_string(0, 'F') << '\n';
```

- Expected Output:

```
find max string : j
find max string : C
find a in string : 1
find f in string : 0
find A in string : 0
find F in string : 1
```

Recursion.cpp
Iterative.cpp

Lecture Agenda

- ✓ Section 1: Introduction to Data Structures
- ✓ Section 2: Execution Time Cases
- ✓ Section 3: Complexity Analysis Examples
- ✓ Section 4: Recursion
- ✓ Section 5: Iteration vs. Recursion Examples

Assignment

HackerRank - Recursion

[01] https://www.hackerrank.com/challenges/functional-programming-warmups-in-recursion---gcd/problem
[02] https://www.hackerrank.com/challenges/functional-programming-warmups-in-recursion---fibonacci-numbers/problem
[03] https://www.hackerrank.com/challenges/pascals-triangle/problem
[04] https://www.hackerrank.com/challenges/string-mingling/problem
[05] https://www.hackerrank.com/challenges/string-o-permute/problem
[06] https://www.hackerrank.com/challenges/string-compression/problem
[07] https://www.hackerrank.com/challenges/prefix-compression/problem
[08] https://www.hackerrank.com/challenges/string-reductions/problem
[09] https://www.hackerrank.com/challenges/functional-programming-the-sums-of-powers/problem
[10] https://www.hackerrank.com/challenges/sequence-full-of-colors/problem
[11] https://www.hackerrank.com/challenges/filter-elements/problem
[12] https://www.hackerrank.com/challenges/convex-hull-fp/problem
[13] https://www.hackerrank.com/challenges/super-digit/problem
[14] https://www.hackerrank.com/challenges/lambda-march-concave-polygon/problem
[15] https://www.hackerrank.com/challenges/functions-and-fractals-sierpinski-triangles/problem
[16] https://www.hackerrank.com/challenges/fractal-trees/problem
[17] https://www.hackerrank.com/challenges/crosswords-101/problem

HackerEarth - Recursion

[01] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/gcd-strings/

[02] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/lockdown-game/

[03] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/hack-the-money/

[04] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/a-tryst-with-chess/

[05] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/its-confidential-f006e2c4/

[06] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/n-queensrecursion-tutorial/

[07] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/simran-and-stairs/

[08] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/biggest-forest-700592dd/

[09] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/question-2-38-cf73c1b4/

[10] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/divide-number-a410603f/

[11] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/encrypted-love-2o/

[12] https://www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/practice-problems/algorithm/jumpingjack-488ce744/


DO
MORE.