Algorithm Analysis Exercises

RES. ASST. KEREM KESKİN

CEN215 - DATA STRUCTURE LAB

Ex1 - Constant Time

```
int get_first_element(int arr[]) {
    return arr[0];
}
```

Ex1 - Constant Time

```
int get_first_element(int arr[]) {
    return arr[0];
}
```

- This code returns the first element of an array.
- Best Case and Worst Case: O(1),O(1)
- Description: This code reads and returns only the first element of the array. No matter the size of the array, the number of operations does not change, there is always a single operation. It runs in constant time because it directly accesses the first element of the array.

Ex2-Find Max and Min in Array

```
void find_max_min(int arr[], int size, int *max, int
*min) {
    *max = arr[0];
    *min = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > *max) {
            *max = arr[i];
        }
        if (arr[i] < *min) {
            *min = arr[i];
        }
    }
}
```

Ex2-Find Max and Min in Array

```
void find_max_min(int arr[], int size, int *max,
int *min) {
    *max = arr[0];
    *min = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > *max) {
            *max = arr[i];
        }
        if (arr[i] < *min) {
            *min = arr[i];
        }
    }
}
```

- This code finds the largest and smallest elements in an array.
- Best Case: O(n) - If all elements in the array are equal, only one comparison is made with each element.
- Worst Case: O(n) - If elements are mixed in the array, each element is compared twice.
- Description: The code takes each element in turn and updates the maximum or minimum value. If all the elements are the same, only one comparison is made, but if the elements are at random values, the whole array is scanned and two comparisons are made., finds the first repeated element in an array.

Ex3 - Linear Search

```
int linear_search(int arr[], int size, int target)
{
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return 1; // target found
        }
    }
    return 0; // target not found
}
```

Ex3 - Linear Search

```
int linear_search(int arr[], int size, int target)
{
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return 1; // target found
        }
    }
    return 0; // target not found
}
```

- This code scans through an array to find the target element.
- Best Case: O(1) - If the element sought is in the first row, it is found immediately.
- Worst Case: O(n) - If the searched element is last or not in the array, all elements are checked.
- Description: The code checks each element in the array in turn until it finds the target element. If the employee we are looking for is at the beginning, the process ends immediately; but we need to look at all elements towards the end of the array or if they are not in the array. Therefore, the number of operations increases with the array size.

Ex4 - Recursive Factorial

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

Ex4 - Recursive Factorial

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

- This code calculates the factorial of a given number.
- Best Case : O(1)
- Worst Case: O(n)
- Explanation: The factorial function multiplies all numbers up to the given number. Therefore 5! It multiplies all numbers backwards as 5, 4, 3... Therefore, a total of n operations are performed.

Ex5 - Nested Loop

```
int find_in_matrix(int matrix[][3], int rows, int target) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            if (matrix[i][j] == target) {
                return 1; // target found
            }
        }
    }
    return 0; // target not found
}
```

Ex5 - Nested Loop

```
int find_in_matrix(int matrix[][3], int rows, int
target) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 3; j++) {
            if (matrix[i][j] == target) {
                return 1; // target found
            }
        }
    }
    return 0; // target not found
}
```

- This code tries to find the target value by searching all elements of a two-dimensional matrix.
- Best Case: O(1) - If the target element is in the first row, it is found immediately.
- Worst Case: O(n×m) — If the target element is last or does not exist at all, all elements are looked at.(Correction: In general, if matrix size not known Worst case: O(n×m) but in this case O(n) □ n=rows because column known and its 3.)
- Description: This code checks each element row by row and column by column. If the employee we are looking for is at the beginning, we will find it immediately. If it is at the end or not at all, we have to look through the entire matrix, so processing time increases with the number of elements of the matrix.

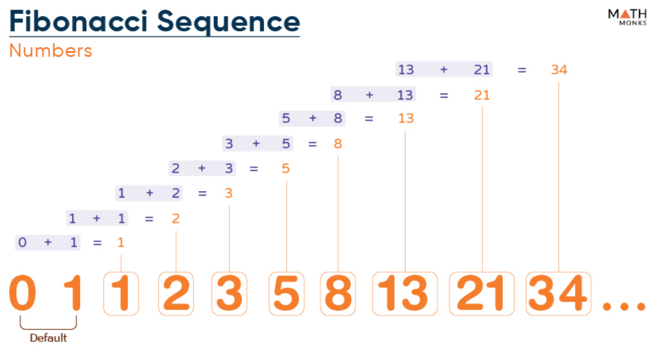
Ex6 - Recursive Fibonacci

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Ex6 - Recursive Fibonacci

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- This code is the nth part of the Fibonacci series. calculates the term. For example, for n=5 the output will be 5, which is the 5th term.
- Best Case: O(1) - If n=0 or n=1, returns in a single operation.
- Worst Case: $O(2^n)$ - For a larger value of n, two new Fibonacci calls are made with each call, which means exponential growth.
- Explanation: Each Fibonacci number is the sum of the previous two Fibonacci numbers. However, the code performs two new calculations with each call, resulting in a large number of repetitive calculations and the processing time increases exponentially.



Ex7 – Find Duplicate

```
int find_duplicate(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                return arr[i];
            }
        }
    }
    return -1; // return -1 if there is no duplicate
}
```

Ex7 – Find Duplicate

```
int find_duplicate(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                return arr[i];
            }
        }
    }
    return -1; // return -1 if there is no
duplicate
}
```

- This code finds the first repeated element in an array.
- Best Case: O(1) - If the first two elements are the same, the loop stops immediately and finds the repeated element.
- Worst Case: O(n^2) - If the array consists of completely unique elements or only the last two elements are the same, all elements are compared.
- Description: This code tries to find the first repeating element by comparing each element with all other elements. If a repetitive element is at the beginning, it can be found immediately. Otherwise, each element has to be looked at each other, which makes O(n^2) .

Ex8 - Sum of Cubes

```
#include <stdio.h>
int sum_of_cubes(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 1; k <= n; k++) {
                total += i * j * k;
            }
        }
    }
    return total;
}
```

Ex8 - Sum of Cubes

```
#include <stdio.h>
int sum_of_cubes(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 1; k <= n; k++) {
                total += i * j * k;
            }
        }
    }
    return total;
}
```

- This code calculates the sum of the cubes of the numbers 1 to n.
- Best Case: O(n^3) - All loops work in all cases.
- Worst Case: O(n^3) - All loops run regardless of the array elements.
- Description: This code checks all possibilities using three nested loops. Therefore, in the best and worst cases the complexity is O(n^3) .

Ex9 – Bubble Sort

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Eleman takası
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }  }  }
```

Ex9 – Bubble Sort

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Eleman takası
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }  }  }
```

- This code sorts the array using the bubble sort algorithm.
- Best Case: O(n) - If the array is already sorted, the loop runs only once.
- Worst Case: O(n^2) - When the array is in reverse order, all comparisons must be made.
- Description: Balloon sort compares consecutive elements each time. If the array is already sorted, at best a single pass is sufficient, but at worst the comparison is made for all elements.

Ex10 - Binary Search

```
#include <stdio.h>
int binary_search(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;          }
```

Ex10 - Binary Search

```
#include <stdio.h>
int binary_search(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;          }
```

- This code implements the binary search algorithm to find a specific element in a sorted array. When the element is found, its index is returned, when it is not found, it returns -1.
- Best Case: O(1) - If the target element is at the midpoint, it is found immediately.
- Worst Case: O(logn) - As the array grows, the search space is halved at each step.
- Description: Binary search is an effective method to find element in a sorted array. Since it reduces the search space by half each time, its complexity becomes O(logn).