# Data Structures and Algorithm
# Assignment-1(Report)

*Aditya Prasad*
*2022036*

**Q1a)**

```cpp
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char from_peg, char to_peg,
                  char temp_peg1, char temp_peg2)
{
    if (n == 0)
        return;
    if (n == 1)
    {
        cout << "T" << from_peg - 'A' + 1 << " -> T" << to_peg - 'A' + 1 << endl;
        return;
    }

    towerOfHanoi(n - 2, from_peg, temp_peg1, temp_peg2,
                                        to_peg);
    cout << "T" << from_peg - 'A' + 1 << " -> T" << temp_peg2 - 'A' + 1 << endl;

    cout << "T" << from_peg - 'A' + 1 << " -> T" << to_peg - 'A' + 1 << endl;

    cout << "T" << temp_peg2 - 'A' + 1 << " -> T" << to_peg - 'A' + 1 << endl;

    towerOfHanoi(n - 2, temp_peg1, to_peg, from_peg,
                                    temp_peg2);
}

int main()
{
    int n;
    cout << "Enter the number of disks: ";
    cin >> n;
    cout << "The sequence of steps are:-" << endl;
    towerOfHanoi(n, 'A', 'D', 'B', 'C');
    return 0;
}
```

**OUTPUT:**

```
Enter the number of disks: 3
The sequence of steps are:-
T1 -> T2
T1 -> T3
T1 -> T4
T3 -> T4
T2 -> T4
```

**Q1b)**

To move 8 disks from source to destination using the above Tower of Hanoi algorithm, we need to follow a recursive approach. Let's examine how the code works with 8 disks:

1. Initially, the towerOfHanoi function is called with n = 8, and the source peg (T1), destination peg (T4), and two temporary pegs (T2 and T3) are provided.
2. The function recursively divides the problem into subproblems:
   - It moves the top 6 disks (n-2) from the source peg (T1) to one of the temporary pegs (T2) using the destination peg (T4) as an auxiliary peg.
   - Then, it moves the bottom 2 disks to the destination peg (T4) directly.
   - Next, it moves the 6 disks (n-2) from the temporary peg (T2) to the destination peg (T4) using the source peg (T1) as an auxiliary peg.
3. This process continues recursively until there are only 2 disks left, which can be moved directly to the destination peg.

Explanation of each step:
- The function divides the problem into smaller subproblems by moving all but two disks to one of the temporary pegs (T2).
- Then, it moves the bottom two disks directly to the destination peg (T4).
- After that, it moves the rest of the disks from the temporary peg (T2) to the destination peg (T4) using the source peg (T1) as an auxiliary.
- This process continues recursively until all disks are moved to the destination peg.

Overall, the algorithm follows the rules of Tower of Hanoi recursively, exploiting the fact that moving n disks is equivalent to moving n-1 disks, with one additional disk movement. This approach ensures that the disks are moved efficiently while maintaining the rules of the puzzle.

**Enter the number of disks: 8**
**The sequence of steps are:-**
**T1 -> T3**
**T1 -> T4**
**T3 -> T4**
**T1 -> T2**
**T1 -> T3**
**T2 -> T3**

**T4 -> T2**
**T4 -> T3**
**T2 -> T3**
**T1 -> T4**
**T1 -> T2**
**T4 -> T2**
**T3 -> T2**
**T3 -> T1**
**T2 -> T1**
**T3 -> T4**
**T3 -> T2**
**T4 -> T2**
**T1 -> T4**
**T1 -> T2**
**T4 -> T2**
**T1 -> T3**
**T1 -> T4**
**T3 -> T4**
**T2 -> T1**
**T2 -> T3**
**T1 -> T3**
**T2 -> T4**
**T2 -> T1**
**T4 -> T1**
**T3 -> T4**
**T3 -> T1**
**T4 -> T1**
**T2 -> T3**
**T2 -> T4**
**T3 -> T4**
**T1 -> T4**
**T1 -> T2**
**T4 -> T2**
**T1 -> T3**
**T1 -> T4**
**T3 -> T4**
**T2 -> T3**
**T2 -> T4**
**T3 -> T4**

**Q1c)**
The time complexity of the Tower of Hanoi problem is the same whether you use three pegs or four pegs because the number of moves required to solve the problem for a given number of disks remains unchanged.

The time complexity of the Tower of Hanoi problem with $n$ disks is $O(2^n)$, regardless of the number of pegs used. This means that the number of moves required to solve the Tower of Hanoi problem doubles with each additional disk.

Therefore, whether you have three or four pegs, the time complexity remains exponential with respect to the number of disks. Adding more pegs doesn't affect the time complexity of the problem; it only changes the strategy used to solve it.

**Q2a)**

Recursive Approach

```cpp
// Recursive Merge Sort
void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    vector<int> left(n1);
    vector<int> right(n2);

    for (int i = 0; i < n1; i++)
        left[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        right[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }
}

void mergeSortRecursive(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSortRecursive(arr, l, m);
        mergeSortRecursive(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Iterative Approach

```cpp
void mergeSortIterative(vector<int>& arr) {
    int n = arr.size();
    for (int curr_size = 1; curr_size <= n - 1; curr_size *= 2) {
        for (int left_start = 0; left_start < n - 1; left_start += 2 * curr_size) {
            int mid = min(left_start + curr_size - 1, n - 1);
            int right_end = min(left_start + 2 * curr_size - 1, n - 1);
            merge(arr, left_start, mid, right_end);
        }
    }
}

int main() {
    cout << "Input the array: ";
    string input;
    getline(cin, input);

    vector<int> arr;
    size_t pos = 0;
    while ((pos = input.find(',')) != string::npos) {
        arr.push_back(stoi(input.substr(0, pos)));
        input.erase(0, pos + 1);
    }
    arr.push_back(stoi(input));

    // Recursive Merge Sort
    mergeSortRecursive(arr, 0, arr.size() - 1);

    cout << "Sorted array by Recursion: ";
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1)
            cout << ",";
    }
    cout << endl;

    // Iterative Merge Sort
    mergeSortIterative(arr);

    cout << "Sorted array by Iteration: ";
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i];
        if (i < arr.size() - 1)
            cout << ",";
    }
    cout << endl;

    return 0;
}
```

This C++ code implements both recursive and iterative versions of the Merge Sort algorithm to sort an array of integers.

1. The code includes necessary header files `<iostream>` and `<vector>` for input/output operations and to use the `std::vector` container, respectively.

2. The `merge` function is defined to merge two subarrays. It takes four parameters:
   - `arr`: A reference to the vector containing the array to be merged.
   - `l`: The left index of the subarray.
   - `m`: The middle index of the subarray.
   - `r`: The right index of the subarray.

This function divides the array into two subarrays, merges them, and updates the original array.

3. The `mergeSortRecursive` function implements the recursive merge sort algorithm. It takes three parameters:
- `arr`: A reference to the vector containing the array to be sorted.
- `l`: The left index of the subarray.
- `r`: The right index of the subarray.

This function recursively divides the array into smaller subarrays until each subarray has only one element, then merges them back together in sorted order using the `merge` function.

4. The `mergeSortIterative` function implements the iterative merge sort algorithm. It takes one parameter:
- `arr`: A reference to the vector containing the array to be sorted.

This function iteratively divides the array into subarrays of increasing size (doubling the size at each iteration) and merges them back together until the entire array is sorted.

5. In the `main` function:
- It prompts the user to input the array of integers.
- It reads the input string and parses it to populate the vector `arr`.
- It calls the `mergeSortRecursive` function to sort the array recursively and prints the sorted array.
- It then calls the `mergeSortIterative` function to sort the array iteratively and prints the sorted array again.

Overall, this code demonstrates the implementation of both recursive and iterative versions of the Merge Sort algorithm to sort an array of integers.

**OUTPUT:**

```
Input the array: 2,3,23,4,5,12,354,2,0,12
Sorted array by Recursion: 0,2,2,3,4,5,12,12,23,354
Sorted array by Iteration: 0,2,2,3,4,5,12,12,23,354
```

**Q2b)**
Suppose the input array is: 9 12 4 6 2 8 11 5

Iteration 1:
Subarrays of size 1: [9] [12] [4] [6] [2] [8] [11] [5]
Merging pairs: [9 12] [4 6] [2 8] [5 11]
Merging pairs: [4 6 9 12] [2 5 8 11]

Iteration 2:
Subarrays of size 2: [4 6 9 12] [2 5 8 11]
Merging pairs: [2 4 5 6 8 9 11 12]
The final sorted array is: 2 4 5 6 8 9 11 12

**Q3)** Sequence of the function:
(Going down in ascending order)

n^-1<
n^-(½)<
log(n)/n <
2^2^100 <
log n to base 10<
log n to base 2<
log(n!) <
2n <
3n <
n2^100 <
n log(n) ) <
C(n,64) <
n^64 <
n^65 <
2.1^(n^½)<
2^n<
2^(n+1)<
n2^n<
3^n <
4^n = 2^2n <
n! <
n^n <
2^2^n

**Q4a)**

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<int> maxMinSubarrays(const vector<int>& arr) {
    int n = arr.size();
    vector<int> result(n, 0);

    for (int size = 1; size <= n; size++) {
        int minValue = INT_MAX;
        for (int i = 0; i <= n - size; i++) {
            minValue = *min_element(arr.begin() + i, arr.begin() + i + size);
            result[size - 1] = max(result[size - 1], minValue);
        }
    }

    return result;
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
                        std::__1::vector<int> maxMinSubarrays(const std::__1::vector<int> &arr)
    vector<int> result = maxMinSubarrays(arr);

    cout << "Consider an array, Arr = [";
    for (int i = 0; i < n; i++) {
        cout << arr[i];
        if (i < n - 1) {
            cout << ", ";
        }
    }
    cout << "]." << endl;

    for (int size = 1; size <= n; size++) {
        cout << "For subarrays of size " << size << ": max(";
        for (int i = 0; i < size; i++) {
            cout << "min(" << arr[i];
            if (i < size - 1) {
                cout << "), ";
            } else {
                cout << ")";
            }
        }
        cout << ") = " << result[size - 1] << endl;
    }

    cout << "The resulting array is [";
    for (int i = 0; i < result.size(); i++) {
        cout << result[i];
        if (i < result.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]." << endl;

    return 0;
}
```

This C++ code finds the maximum of minimums of all subarrays of a given array.

1. Includes necessary header files `<iostream>`, `<vector>`, and `<algorithm>` for input/output operations, vector container, and the `min_element` function, respectively.

2. Defines a function `maxMinSubarrays` that takes a constant reference to a vector of integers `arr` as input and returns a vector of integers as output. This function computes the maximum of minimums of all subarrays of `arr`.

3. In the `main` function:
   - Prompts the user to enter the size of the array (`n`).
   - Reads the elements of the array from the user.
   - Calls the `maxMinSubarrays` function with the input array and stores the result.
   - Prints the original array.
   - Loops over the result vector to print the maximum of minimums for each subarray size.
   - Prints the resulting array containing the maximum of minimums for all subarray sizes.

Let's delve into the `maxMinSubarrays` function:

- It initializes a result vector `result` of size `n` with all elements initialized to 0.
- It iterates over all possible subarray sizes from 1 to `n`.
- For each subarray size, it initializes `minValue` to `INT_MAX`.
- It iterates over all possible starting indices of subarrays of the current size.
- For each starting index, it finds the minimum value in the subarray using the `min_element` function.
- It updates `minValue` with the minimum value found.
- It updates the corresponding element in the `result` vector with the maximum of the current value and the `minValue`.
- Finally, it returns the `result` vector containing the maximum of minimums for all subarray sizes.

In summary, the code allows the user to input an array of integers, computes the maximum of minimums of all subarrays of the array, and prints the results.

Time Complexity for first code: O(n^2) In the brute force approach, for each subarray size, we iterate over all possible subarrays, and for each subarray, we find the minimum element. This results in a time complexity of O(n^2), where n is the size of the array. Optimized Approach using Monotonic Stack

**OUTPUT:**

```
Enter the size of the array: 5
Enter the elements of the array: 1 2 3 4 5
Consider an array, Arr = [1, 2, 3, 4, 5].
For subarrays of size 1: max(min(1)) = 5
For subarrays of size 2: max(min(1), min(2)) = 4
For subarrays of size 3: max(min(1), min(2), min(3)) = 3
For subarrays of size 4: max(min(1), min(2), min(3), min(4)) = 2
For subarrays of size 5: max(min(1), min(2), min(3), min(4), min(5)) = 1
The resulting array is [5, 4, 3, 2, 1].
```

**Q4b)**

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

vector<int> maxMinSubarrays(const vector<int>& arr) {
    int n = arr.size();
    vector<int> result(n, 0);

    stack<int> indexStack;
    stack<int> maxValues;

    for (int i = 0; i < n; i++) {
        while (!indexStack.empty() && arr[i] < arr[indexStack.top()]) {
            indexStack.pop();
        }

        if (!indexStack.empty()) {
            result[i] = arr[indexStack.top()];
        } else {
            result[i] = arr[i];
        }

        indexStack.push(i);
    }

    return result;
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    vector<int> result = maxMinSubarrays(arr);

    cout << "Consider an array, Arr = [";
    for (int i = 0; i < n; i++) {
        cout << arr[i];
        if (i < n - 1) {
            cout << ", ";
        }
    }
    cout << "]." << endl;

    for (int i = 0; i < n; i++) {
        cout << "For subarrays of size " << i + 1 << ": max(" << result[i] << ") = " << result[i] << endl;
    }

    cout << "The resulting array is [";
    for (int i = 0; i < result.size(); i++) {
        cout << result[i];
        if (i < result.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]." << endl;

    return 0;
}
```

This C++ code finds the maximum of minimums of all subarrays of a given array using a stack-based approach.

1. Includes necessary header files `<iostream>`, `<vector>`, and `<stack>` for input/output operations, vector container, and stack data structure, respectively.
2. Defines a function `maxMinSubarrays` that takes a constant reference to a vector of integers `arr` as input and returns a vector of integers as output. This function computes the maximum of minimums of all subarrays of `arr` using a stack-based approach.
3. In the `main` function:
   - Prompts the user to enter the size of the array (`n`).
   - Reads the elements of the array from the user.
   - Calls the `maxMinSubarrays` function with the input array and stores the result.
   - Prints the original array.
   - Loops over the result vector to print the maximum of minimums for each subarray size.
   - Prints the resulting array containing the maximum of minimums for all subarray sizes.

Now, let's delve into the `maxMinSubarrays` function:
- It initializes a result vector `result` of size `n` with all elements initialized to 0.
- It declares two stacks: `indexStack` to store indices of elements and `maxValues` to store maximum values.
- It iterates over each element of the input array `arr`.
- Within the loop, it compares the current element with elements stored at the top of the `indexStack`.
- If the current element is smaller than the element at the top of the stack, it pops elements from both stacks until it finds an element greater than or equal to the current element.
- After the while loop, if the `indexStack` is not empty, it updates the result array with the element at the top of the `indexStack` (which represents the minimum element within the current subarray).
- If the `indexStack` is empty, it means the current element is the smallest so far for all previous elements, so it updates the result array with the current element.
- Finally, it pushes the current index onto the `indexStack`.

In summary, the code efficiently computes the maximum of minimums of all subarrays of a given array using a stack-based approach.

Time Complexity for second code: O(n) In the optimized approach using a monotonic stack, we iterate through the array once, maintaining a stack of indices. For each element, we pop elements from the stack until a smaller or equal element is encountered, updating the result array. This results in a linear time complexity of O(n), where n is the size of the array. Clearly, the optimized approach using a monotonic stack is more efficient with a time complexity of O(n), making it a better choice for larger arrays.

**OUTPUT:**

```
Enter the size of the array: 5
Enter the elements of the array: 1 2 3 4 5
Consider an array, Arr = [1, 2, 3, 4, 5].
For subarrays of size 1: max(1) = 1
For subarrays of size 2: max(1) = 1
For subarrays of size 3: max(2) = 2
For subarrays of size 4: max(3) = 3
For subarrays of size 5: max(4) = 4
The resulting array is [1, 1, 2, 3, 4].
```