Data Structures and Algorithms
(ES221)

Computational Complexity of Algorithms
(2)

Dr. Zubair Ahmad

- **Attendance?**

  - Active Attendance
  - **Dead Bodies.**
  - **Active Minds**
  - Mobiles in hands -> Mark as absent
  - 80% mandatory

Scenario!!

$$\lim_{x \to 0} \frac{Sin(x)}{x} = 1$$

Whats Wrong withabove equation?

L'Hôpital's Rule

L'Hôpital's Rule is a powerful tool for evaluating difficult limits, especially in cases of indeterminate forms..

By differentiating the numerator and denominator separately, we can often simplify complex expressions and find the limit efficiently

It should be used carefully and only when applicable.

L'Hôpital's Rule

L'Hôpital's Rule states that if we have a limit of the form:

$$\lim_{x \to c} \frac{f(x)}{g(x)}$$

and both **f(x)** and **g(x)** approach either **0** or **∞** as **x→c** then we can differentiate the numerator and denominator separately:

$$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f''(x)}{g''(x)}$$

f(x) and **g(x)** are both differentiable near x=c

.The limit of **f′(x)/g′(x)** exists or can be further evaluated

L'Hôpital's Rule

L'Hôpital's Rule is only applicable to limits that result in the following **indeterminate forms**:

$$\frac{0}{0} \qquad\qquad \frac{\infty}{\infty}$$

If the limit does **not** result in one of these forms, L'Hôpital Rule **cannot** be applied directly.

L'Hôpital's Rule

To evaluate a limit using L'Hôpital's Rule, follow these steps:

Check the indeterminate form: Compute $\lim_{x \to c} \frac{f(x)}{g(x)}$ If it results in 0/0 or ∞/∞, proceed

Differentiate separately: Compute the derivatives **f'(x)** and **g'(x)**

Evaluate the new limit: Compute $\lim_{x \to c} \frac{f''(x)}{g''(x)}$

Repeat if necessary: If the new limit is still indeterminate, apply L'Hôpital's Rule again

L'Hôpital's Rule

Examples

$$\lim_{x \to \infty} \frac{x}{e^x}$$

$$\lim_{x \to 0} \frac{1 - \cos(x)}{x^2}$$

Maximum Subsequence Problem

The **Maximum Subsequence Problem** (also known as the **Maximum Subarray Problem**) is a fundamental problem in computer science and algorithm design.

It involves finding a contiguous subsequence (subarray) within a given array of numbers that has the **maximum possible sum**

Given an array of integers:

A=[a1,a2,a3,...,an]

Find a contiguous subarray **A[i:j]** (where **1≤i≤j≤n** such that the sum of its elements is maximized.

Maximum Subsequence Problem

```
current_sum=max(A[i],current_sum+A[i])
```

Example
Input:
A=[−2,1,−3,4,−1,2,1,−5,4]

Output:
The maximum sum subarray is **[4, -1, 2, 1]** with sum **6**

A=[5,−2,−3,7,−1,2,−2,3,−5,6]

Goal: Find the **maximum sum** and the **subarray** that gives this sum.

Brute Force Approach

The **brute force approach** systematically evaluates all possible subarrays to determine which one has the maximum sum.

Steps:

1.Generate all possible subarrays: We use **two nested loops** to select every possible contiguous subarray.

2.Compute the sum of each subarray: A third loop iterates over the selected subarray to compute the sum.

3.Track the maximum sum: If the computed sum is greater than the current maximum, update the maximum sum.

Brute Force Approach ($O(n^3)$)

Algorithm Explanation

1.Outer loop picks the starting index **i** of the subarray.
2.Middle loop picks the ending index **j** of the subarray.
3.Inner loop sums the elements from **i** to **j**.
4.Store the **maximum sum** found

Complexity Analysis

- **Outer loop (i)** runs **O(n) times**.
- **Middle loop (j)** runs **O(n) times** for each **i**.
- **Inner loop (k)** computes the sum

Improved Brute Force Approach ($O(n^2)$)

The brute force approach (**$O(n^3)$**) recomputes the sum of every subarray in an inner loop. We can improve it by **avoiding redundant computations**.

Optimization Idea
Instead of recalculating the sum of subarrays from scratch in the innermost loop, we **use a cumulative sum approach**:

- Start from an index **i**
- Expand the subarray by including **j**
- Maintain a **running sum** instead of recalculating it.

This reduces one loop, making the complexity **$O(n^2)$ instead of $O(n^3)$**.

Improved Brute Force Approach ($O(n^2)$)

Algorithm Steps

1. **Outer loop** selects the starting index i.
2. **Inner loop** expands the subarray to the ending index j.
3. **Keep a running sum (currentSum)** instead of recalculating with another loop.
4. **Track the maximum sum** and update the subarray.

Complexity Analysis

- **Outer loop (i)** runs **O(n) times**.
- **Inner loop (j)** runs **O(n) times** for each i.
- **No extra third loop for summing elements**

Kadane Algorithm

Kadane Algorithm is an **efficient method** to find the **maximum sum subarray** in **O(n) time complexity**. Instead of checking all subarrays like brute force methods, it **maintains a running sum** and updates the maximum sum found so far.

Key Idea

1.Traverse the array once.

2.Maintain a currentSum:

1. If currentSum becomes negative, reset it to 0.
2. Otherwise, continue adding elements.

3.Track the maximum sum (maxSum) found so far.

4.If needed, store the subarray indices to get the actual subarray.

Kadane Algorithm

Algorithm Steps

1.Initialize maxSum = INT_MIN (to track the max subarray sum).

2.Initialize currentSum = 0 (to track the running sum).

3.Iterate through the array:

1. Add the current element to currentSum.
2. If currentSum > maxSum, update maxSum.
3. If currentSum < 0, reset currentSum = 0 (ignore negative sums).

4.The final maxSum is the answer

Space complexity

Space complexity refers to the amount of memory required by an algorithm to run, relative to the size of the input.

Its important because, even if an algorithm is time-efficient, excessive memory usage could make it impractical for large inputs

Fixed part: The space required for variables, constants, and program code, which does not depend on the input size.

Variable part: The space required for dynamically allocated memory such as data structures, arrays, recursion stacks, etc., which varies with the input size

Space complexity

Constant Space Complexity (O(1))

An algorithm has **O(1)** space complexity if the memory usage doesn’ t grow with the input size. The algorithm uses a fixed amount of memory regardless of the input size

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

int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 10;
    cout << "Sum: " << add(x, y) <<
endl;
    return 0;
}
```

Linear Space Complexity (O(n))

An algorithm has **O(n)** space complexity if the memory required grows linearly with the size of the input. This typically happens when the algorithm needs to store the entire input or a portion of it

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

vector<int> store_numbers(int n) {
    vector<int> arr;
    for (int i = 0; i < n; i++) {
        arr.push_back(i);
    }
    return arr;
}

int main() {
    int n = 5;
    vector<int> result = store_numbers(n);
    for (int i : result) {
        cout << i << " ";
    }
    cout << endl;
    return 0;
}
```

Space complexity

Quadratic Space Complexity ($O(n^2)$)

An algorithm has **$O(n^2)$** space complexity when the memory usage grows quadratically with the size of the input. This typically happens when there' s a 2D data structure like a matrix.

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

vector<vector<int>> create_matrix(int n) {
    vector<vector<int>> matrix(n, vector<int>(n, 0));
// Creates an n x n matrix filled with 0s
    return matrix;
}

int main() {
    int n = 3;
    vector<vector<int>> result = create_matrix(n);
    for (const auto& row : result) {
        for (int elem : row) {
            cout << elem << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Recursive Space Complexity (O(n))

When using recursion, the space complexity often depends on the depth of the recursion stack**. For example**, in problems where the function calls itself recursively, each recursive call uses some space.

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

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

int main() {
    int n = 5;
    cout << "Factorial: " <<
factorial(n) << endl;
    return 0;
}
```

Questions?

zahmaad.github.io