Data Structure and Algorithms

Affefah Qureshi

Department of Computer Science

Iqra University, Islamabad Campus.

Complexity Analysis

- Complexity analysis is a computer science technique that measures the amount of resources required to run an algorithm, such as time or storage.
- It's used to understand how an algorithm behaves as the size of its input increases.
- Algorithms are often expressed in terms of order complexity, such as O(n) or O(n^2). The term with the highest degree is used to predict how long it will take for the algorithm to complete.

Comparing Algorithms

- Given two or more algorithms to solve the same problem, how do we select the best one?

- Some criteria for selecting an algorithm
  - Is it easy to implement, understand, modify?
  - How long does it take to run it to completion?
  - How much of computer memory does it use?

- Software engineering is primarily concerned with the first criteria

- In this course we are interested in the second and third criteria

Complexity Analysis

1. Complexity in terms of input size, N
2. Machine Independent
3. Basic Computer Steps
4. Time and Space

Comparing Algorithms

- Time complexity
  - The amount of time that an algorithm needs to run to completion
  - Better algorithm is the one which runs faster
    - ➢ Has smaller time complexity
- Space complexity
  - The amount of memory an algorithm needs to run
- In this lecture, we will focus on analysis of time complexity

How To Calculate Running Time

Most algorithms transform input objects into output objects

- The running time of an algorithm typically grows with input size
  - Idea: analyze running time as a function of input size

How To Calculate Running Time

- Most important factor affecting running time is usually the size of the input

```
int find_max( int *array, int n ) {
    int max = array[0];
    for (int i = 1; i < n; ++i ) {
        if ( array[i] > max ) { max =
            array[i];
        }
    }
    return max;
}
```

- Regardless of the size n of an array the cost will always be same
  - Every element in the array is checked one time

How To Calculate Running Time

- Even on inputs of the same size, running time can be very different

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

- Idea: Analyze running time for different cases
  - Best case
  - Worst case
  - Average case

Asymptotic Notations

Asymptotic Analysis is defined as the big idea that handles the above issues in analyzing algorithms.

In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time).

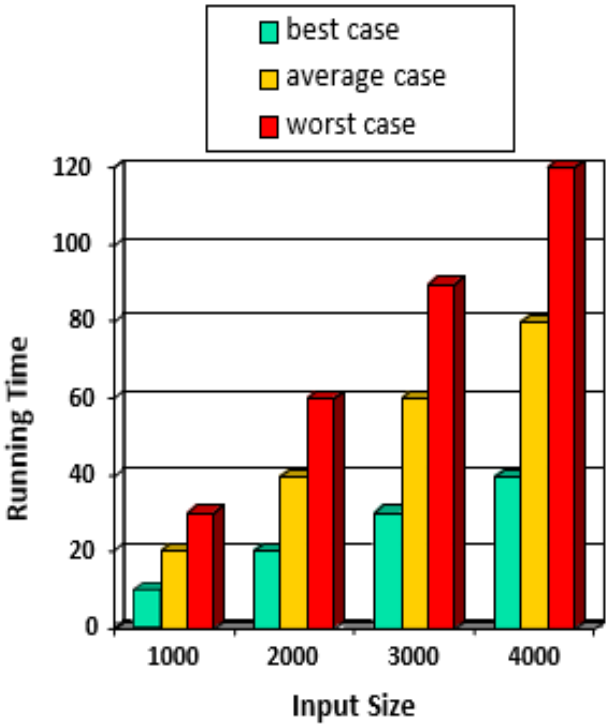
We calculate, how the time (or space) taken by an algorithm increases with the input size.

Asymptotic Notations

- Big Oh O
  - Worst Case
  - Upper Bound (At most)
- Big Omega Ω
  - Best Case
  - Lower Bound (At least)
- Big Theta θ
  - Average Case
  - Upper Lower Bound

How To Calculate Running Time

- Best case running time is usually not very useful
- Average case time is very useful but often hard to determine
- Worst case running time is easier to analyze
  - Crucial for real-time applications such as games, finance and robotics



Analyzing an Algorithm – Operations

- Each machine instruction is executed in a fixed number of cycles
  - We may assume each operation requires a fixed number of cycles

- Idea: Use abstract machine that uses steps of time instead of secs
  - Each elementary operation takes 1 steps
- Example of operations
  - Retrieving/storing variables from memory
  - Variable assignment =
  - Integer operations + - * / % ++ --
  - Logical operations && || !
  - Bitwise operations & | ^ ~
  - Relational operations == != < <= => >
  - Memory allocation and deallocation new delete

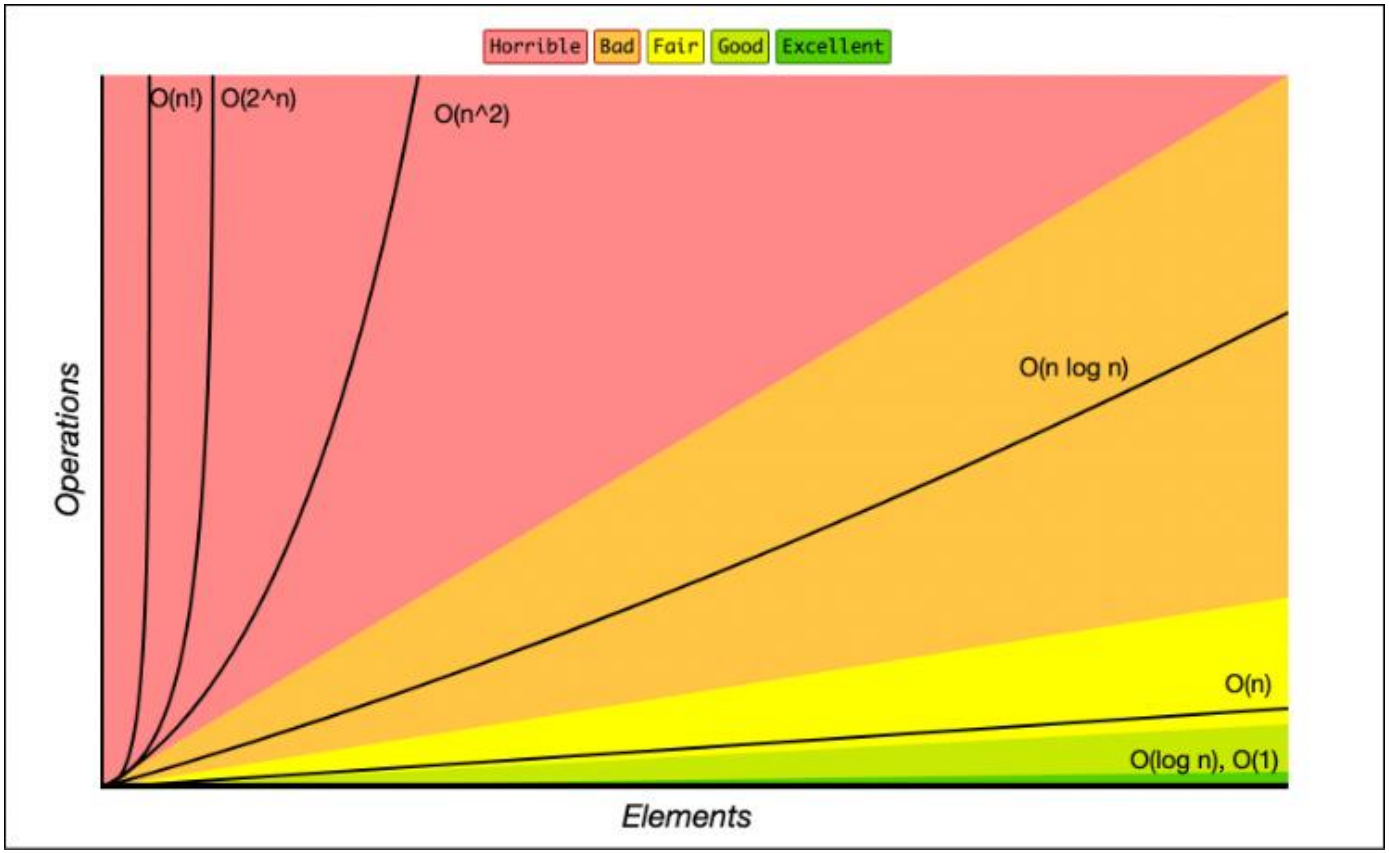
Types of complexities (time and space)

- Constant: O(1)
- Linear time: O(n)
- Logarithmic time: O(n log n)
- Quadratic time: $O(n^2)$
- Exponential time: $O(2^n)$
- Factorial time: O(n!)

Big Oh

- Indicates the upper or highest growth rate that the algorithm
- Ignore Constants
  - 5n → O(n)
- Certain Terns “dominate’ others
  - $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$
  - Ignore low order terms

Big Oh


Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Analyzing an Algorithm – Blocks of Operations

- Each operation runs in a step of 1 time unit
- Therefore any fixed number of operations also run in 1 time step
  - s1; s2; …. ; sk
  - As long as number of operations k is constant

```
//Swap variables a and b

int temp = a;
a=b;
b = temp;
```

Constant Time: O(1) Example

X = 5 + (15 * 20)

- Independent of input size , N
- O(1)

X = 5 + (15 * 20)

Y = 15 – 2

Print X + Y

- Total time = O(1) + O(1) + O(1) = O(1)

Linear Time: O(N) Example

for x in range (0,n): → n+1 times O(n)

print x; // O(1)

So, O(N) * O(1) = O(N)

Linear Time: O(N) Example

X = 5 + (15 * 20) // O(1)

for x in range (0,n): → n+1 times O(n)

print x; // O(1)

So, Total Time = O(1) + O(N) = O(N)

Quadratic Time: $O(N^2)$ Example

for x in range (0,n): → n+1 times O(n)

for y in range (0,n): → n+1 times O(n)

print x; // O(1)

So, $O(N) * O(N) * O(1) = O(N^2)$

Quadratic Time: $O(N^2)$ Example

```
X = 5 + (15 * 20) // O(1)
for x in range (0,n): → n+1 times O(n)
        print x; // O(1)
for x in range (0,n): → n+1 times O(n)
        for y in range (0,n): → n+1 times O(n)
                print x; // O(1)
```

So, $O(1) + O(N) + O(N^2) = O(N^2)$

Analyzing an Algorithm

```
//input: int A[N], array of N integers
//OutputL Summ of all nu,bers in array A

int sumArraay(int A[], int n) {
        int s=0;
        for(int i=0; i<n; i++)
                s = s + A[i];
        return s; }
```

- The complexity function of the algorithm is : O(n)

O(?) → $O(N^2)$

```
if x > 0:
  //O(1)
 else if x < 0:
  // O(log n)
 else:
   //O(n^2)
```

Any Question So Far?