Reading Material on Analysis of Algorithms

Introduction

Analyzing an algorithm involves measuring how its resource usage grows with the size of the input. The two main resources of interest are **time** (how long the algorithm takes to run) and **space** (how much memory it uses). In this guide, we explore how to perform such analyses, understand asymptotic notations, and classify algorithms by their complexity under different scenarios.

---

1. Calculating Time Complexity

What Is Time Complexity?

Time complexity provides an estimate of the amount of time an algorithm requires relative to the size of its input. It is commonly expressed using **Big O notation** (e.g., O(n), O(n²), etc.), which describes an upper bound on the running time as the input size approaches infinity

How to Calculate Time Complexity

1. **Identify the Basic Operations:**
   Determine the most frequent operations (e.g., comparisons, assignments, arithmetic operations) that contribute significantly to the running time.

2. **Analyze Loops and Recursions:**

   - **Loops:** Count how many times each loop runs. For example, a single loop over *n* elements generally contributes O(n) time.
   - **Nested Loops:** Multiply the sizes. For instance, two nested loops, each iterating *n* times, result in O(n²).
   - **Recursion:** Write a recurrence relation representing the number of operations, then solve it using techniques like the Master Theorem.

3. **Combine the Contributions:**
   Sum the complexities of different parts of the algorithm, and then take the highest-order term (the one that grows fastest) as the overall time complexity.

Example

Consider a simple algorithm:

```
// Pseudocode

for (i = 0; i < n; i++) {

    for (j = 0; j < n; j++) {

        // Constant time operation

    }

}
```

- The outer loop runs *n* times.
- The inner loop also runs *n* times for each iteration of the outer loop.
- Total operations ≈ n × n = n², so the time complexity is **O(n²)**.

---

2. Tilde Calculations and Order of Growth

Asymptotic Notations

In addition to Big O notation, several other notations help describe the behavior of functions as inputs grow large:

- **Big O (O):** An upper bound. It tells you that a function grows no faster than a certain rate.
- **Big Omega (Ω):** A lower bound. It indicates that a function grows at least as fast as a certain rate.
- **Big Theta (Θ):** A tight bound. When an algorithm's running time is both O(f(n)) and Ω(f(n)), it is Θ(f(n)).

Tilde (~) Notation

We use tilde approximations, where we throw away low-order terms that complicate formulas. We write ~ f(N) to represent any function that when divided by f(N) approaches 1 as N grows. We write g(N) ~ f(N) to indicate that g(N) / f(N) approaches 1 as N grows.

function	tilde approximation	order of growth
$N^3/6 - N^2/2 + N/3$	$\sim N^3/6$	N^3
$N^2/2 - N/2$	$\sim N^2/2$	N^2
$\lg N + 1$	$\sim \lg N$	$\lg N$
3	~ 3	1

Order of Growth

The **order of growth** describes how quickly a function (or algorithm's running time) increases as the input size increases. For example:

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

Understanding the order of growth helps in comparing the efficiency of different algorithms, especially for large inputs.

description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$	[*see page 47*]	*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`  if (a[i] > max) max = a[i];`
			loop	*find the maximum*

linearithmic	$N \log N$	[*see* ALGORITHM 2.4]	*divide and conquer*	*mergesort*

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  if (a[i] + a[j] == 0)`
		`  cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  for (int k = j+1; k < N; k++)`
		`  if (a[i] + a[j] + a[k] == 0)`
		`  cnt++;`
			triple loop	*check all triples*

exponential	2^N	[*see* CHAPTER 6]	*exhasutive search*	*check all subsets*

3. Complexity Classes: Best, Average, and Worst Cases

Definitions

- **Best Case:**
  The scenario where the algorithm performs the minimum number of operations. Although it provides an optimistic view, it is usually less important in worst-case guarantees.

- **Average Case:**
  The expected performance under a probabilistic distribution of all possible inputs. This analysis can be more involved since it often requires assumptions about input distribution.

- **Worst Case:**
  The scenario where the algorithm performs the maximum number of operations. This is most commonly used to ensure that the algorithm will not exceed a certain running time, regardless of input.

Examples

- **Linear Search:**

    - Best case: O(1) (if the target element is at the beginning)
    - Worst case: O(n) (if the target is not found or is at the end)
    - Average case: O(n) (assuming each position is equally likely)

- **Quick Sort:**

    - Best/Average case: O(n log n) (when the pivot divides the array evenly)
    - Worst case: O(n^2) (when the pivot is the smallest or largest element in a highly unbalanced partition)

Understanding these classes helps in selecting the appropriate algorithm for the problem at hand and preparing for the worst-case performance in critical applications.

---

4. Calculating Memory Usage

What Is Space Complexity?

Space complexity measures the total memory space that an algorithm requires relative to the input size. It considers both:

type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

Typical memory requirements for primitive types

- **Auxiliary Space:** Extra space or temporary space used by the algorithm.
- **Input Space:** The space taken up by the inputs.

How to Calculate Memory Usage

1. **Identify Variables and Data Structures:**
   Count the space required by variables, arrays, lists, trees, etc. For example, an array of size *n* typically uses O(n) space.

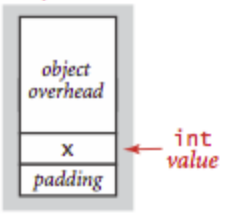
2. **Consider Recursive Calls:**
   Each recursive call may add a new frame to the call stack. Analyze the maximum depth of recursion to determine additional space usage.

3. **Combine the Contributions:**
   Add up the space used by all components. Often, we express space complexity using Big O notation, focusing on the dominant term.

integer wrapper object

```
public class Integer
{
   private int x;
...
}
```

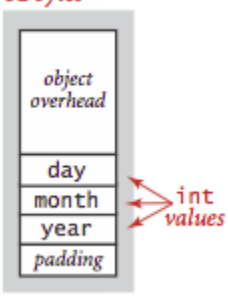
24 bytes



date object

```
public class Date
{
   private int day;
   private int month;
   private int year;
...
}
```

32 bytes



Example

If an algorithm uses:

- An input array of size *n* (O(n))
- A fixed number of extra variables (O(1))
- Recursive calls that add up to O(n) additional space

The total space complexity would be O(n).