Lecture Notes
Data Structures and Algorithms

Algorithm Analysis

Overview

1. Introduction to algorithm analysis:
    - What is algorithm analysis?
    - Why is algorithm analysis important?
    - Basic definitions (e.g. input size, running time)
2. Asymptotic notation:
    - Big-O notation
    - Other asymptotic notations (e.g. Big-Theta, Big-Omega)
3. Analyzing the running time of algorithms:
    - Counting operations
    - Best-case, worst-case, and average-case analysis
    - Amortized analysis
4. Examples of algorithm analysis
5. Counting atomic operations in source code
    - Identifying atomic operations
    - Measuring the number of atomic operations in a given piece of code
    - Using tools to automate the counting process
6. Conclusion:
    - Recap of key concepts
    - Future directions for algorithm analysis

Introduction to algorithm analysis

Algorithm analysis is the process of evaluating the efficiency of an algorithm, or a step-by-step procedure for solving a problem. It is an important area of study in computer science, as the efficiency of an algorithm can have a significant impact on the performance of a computer program.

There are several factors that can affect the efficiency of an algorithm, including the size of the input, the number of steps required to solve the problem, and the resources (e.g. memory, time) used by the algorithm. In algorithm analysis, we are primarily concerned with the running time of an algorithm, or the amount of time it takes to execute on a particular input.

To analyze the running time of an algorithm, we typically express the input size as a variable (e.g. n for a list of n items) and measure the number of basic operations (e.g. comparisons, assignments) performed by the algorithm as a function of the input size. For example, the running time of a linear search algorithm for a list of n items might be expressed as T(n) = 3n + 2, where T(n) is the running time and the 3n term represents the number of comparisons performed.

In Java, we can measure the running time of an algorithm by using the **System.currentTimeMillis()** method to record the start and end times of the algorithm, and then calculating the elapsed time in milliseconds. For example:

```
long startTime = System.currentTimeMillis();
// algorithm goes here
long endTime = System.currentTimeMillis();
long elapsedTime = endTime - startTime;
```

Asymptotic Notation

Measuring the runtime of an algorithm involves running the algorithm on a specific input and measuring the time it takes to complete. This is often done using a stopwatch or a timer function in a programming language. The goal of measuring the runtime of an algorithm is to get a specific, numerical value for the time it takes to run on a particular input.

Asymptotic analysis, on the other hand, is a way of studying the behavior of an algorithm as the input size gets very large. It is used to compare the efficiency of different algorithms, even if they have different runtimes for small inputs. In asymptotic analysis, we use asymptotic notation (e.g. Big-O, Big-Omega, Big-Theta) to describe the general trend in the running time of an algorithm as the input size increases.

Asymptotic notation is a system of mathematical notation used to describe the behavior of an algorithm as the input size grows indefinitely. It allows us to compare the efficiency of different algorithms, even if they have different running times for small inputs.

What is an Asymptote?

A line such that the distance between the curve and the line approaches zero as one or both of the ***x*** or ***y*** coordinates tends to infinity.

There are several different asymptotic notations that are commonly used in algorithm analysis, including Big-O, Big-Omega, and Big-Theta.

- **Big-O** notation is used to describe an upper bound on the running time of an algorithm. It represents the worst-case scenario, in which the algorithm takes the longest time to complete. For example, if the running time of an algorithm is $T(n) = 5n^2 + 3n + 1$, we might say that the

algorithm has a running time of $O(n^2)$, because the n^2 term grows faster than the other terms as n gets larger.

Big-O (**O**) notation is a mathematical notation used to describe the upper bound on the running time of an algorithm. It is commonly used in algorithm analysis to compare the efficiency of different algorithms.

Formally, we say that a function f(n) is O(g(n)) if and only if there exist positive constants c and n_0 such that f(n) <= c·g(n) for all n >= n0. In other words, the function f(n) is bounded above by a constant multiple of the function g(n) as n gets larger and larger.

Big-Omega (Ω) notation is used to describe a lower bound on the running time of an algorithm. It represents the best-case scenario, in which the algorithm takes the shortest time to complete. For example, if the running time of an algorithm is $T(n) = 5n^2 + 3n + 1$, we might say that the algorithm has a running time of **Omega**(n), because the n^2 term grows faster than the other terms as n gets larger.

- **Big-Theta (Θ)** notation is used to describe ***both*** an upper and a lower bound on the running time of an algorithm.

So What's the Difference Between Big O, Big Omega, and Big Theta?

We can think of Big O, Big Omega, and Big Theta like conditional operators:

- Big O is like **<=**, meaning the rate of growth of an algorithm is less than or equal to a specific value, e.g: $f(x) <= O(n^2)$
- Big Omega is like **>=**, meaning the rate of growth is greater than or equal to a specified value, e.g: $f(x) >= \Omega(n)$.
- Big Theta is like **==**, meaning the rate of growth is equal to a specified value, e.g: $f(x) = \Theta(n^2)$.

Intuition

Big-O notation is a way of describing how the running time of an algorithm grows as the input size gets larger. It's like a "zoomed out" view of the running time, where we ignore the details and just look at the overall trend.

For example, imagine that you have a list of numbers and you want to find the largest number in the list. One way to do this is to simply go through the list one number at a time and keep track of the largest number that you've seen so far. This algorithm is called a linear search, and it has a running time of O(n), where n is the size of the list.

Another way to find the largest number in the list is to sort the list in ascending order and then pick the last number. This algorithm is called a sorting algorithm, and it has a running time of O(n log n).

If you compare the running times of these two algorithms, you can see that the sorting algorithm takes longer to run, but it's not as simple as just saying "the sorting algorithm is twice as slow as the linear

search algorithm." Instead, you can use Big-O notation to describe how the running time of each algorithm grows as the input size gets larger.

In this case, you can say that the linear search algorithm has a running time of O(n), which means that it takes about the same amount of time to run no matter how large the list is (as long as it's not empty). On the other hand, you can say that the sorting algorithm has a running time of O(n log n), which means that it takes longer to run as the list gets larger.

Complexity Classes

Here is a list of common complexity classes, along with intuition and example algorithms:

- O(1) ("constant time"): An algorithm is said to run in constant time if the running time is independent of the input size. This means that the algorithm will always take the same amount of time to run, regardless of how big the input is. Examples of O(1) algorithms include accessing an element in an array (e.g. arr[i]) and looking up a value in a hash table.
- O(log n): An algorithm is said to run in logarithmic time if the running time increases logarithmically with the input size. This means that the algorithm becomes more efficient as the input size gets larger. Examples of O(log n) algorithms include binary search and the divide-and-conquer algorithm for finding the maximum subarray sum.
- O(n) ("linear time"): An algorithm is said to run in linear time if the running time increases linearly with the input size. This means that the algorithm takes longer to run as the input size gets larger, but the increase is always proportional. Examples of O(n) algorithms include linear search and iterating through an array.
- O(n log n): An algorithm is said to run in n log n time if the running time is the product of the input size and the logarithm of the input size. This is generally considered to be very efficient, because the logarithmic factor helps to offset the linear increase in running time. Examples of O(n log n) algorithms include quicksort and merge sort.
- O(n^2): An algorithm is said to run in quadratic time if the running time is the square of the input size. This is generally considered to be inefficient, because the running time increases very quickly with the input size. Examples of O(n^2) algorithms include bubble sort and the brute-force algorithm for finding the maximum subarray sum.
- O(n^3) (cubic time): The running time of an algorithm is O(n^3) if it grows as the cube of the input size. An algorithm with a running time of O(n^3) will become extremely slow as the input size increases. Example: Matrix multiplication.

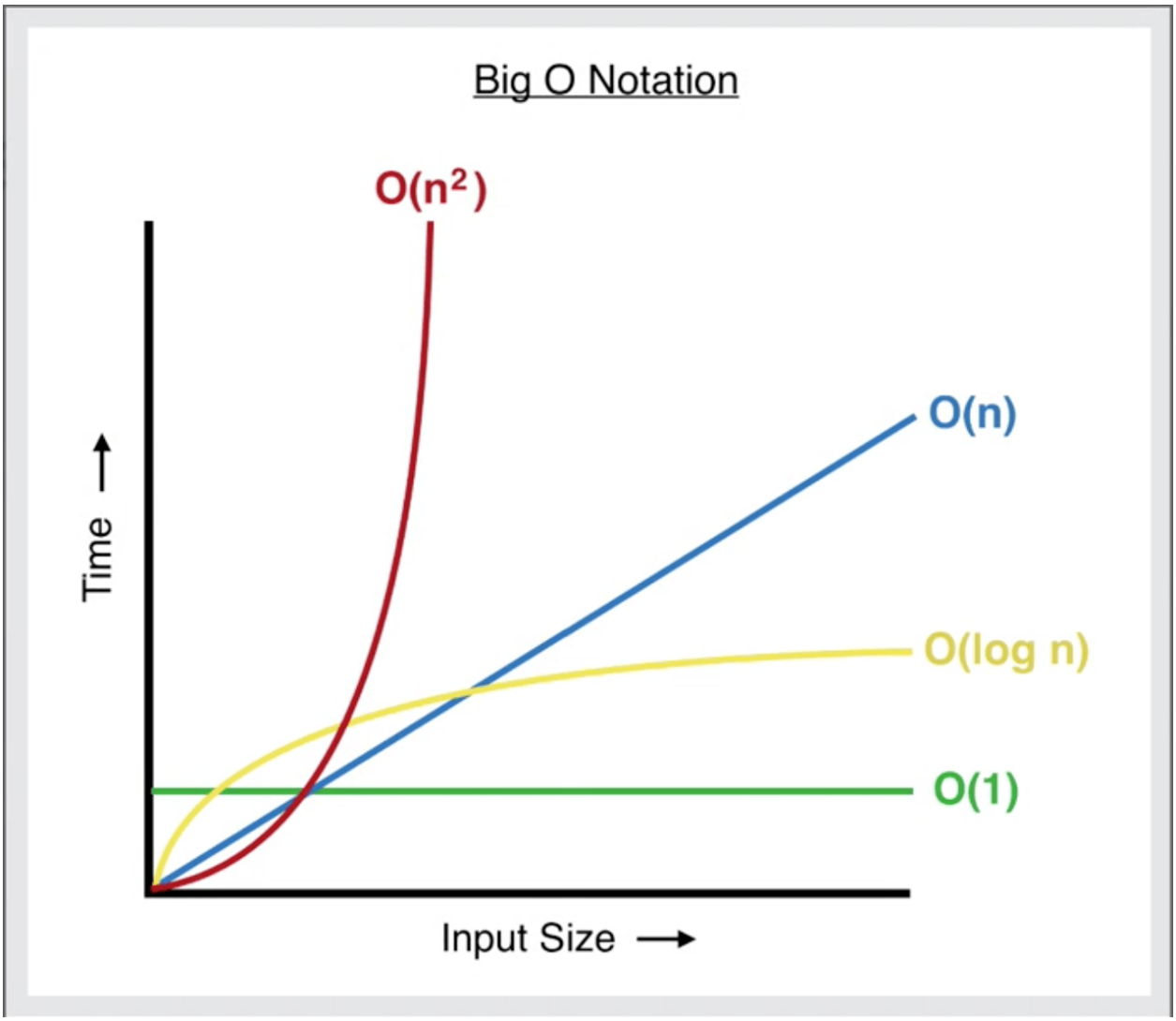

Figure 1. Source miro.medium.com

Analyzing the running time of algorithms

There are several ways to analyze the running time of an algorithm, including counting operations, best-case, worst-case, and average-case analysis, and amortized analysis.

- **Counting operations**: One way to analyze the running time of an algorithm is to count the number of basic operations (e.g. assignments, comparisons) that the algorithm performs. For example, the following Java code implements a linear search algorithm that has a running time of O(n), because the number of comparisons grows linearly with the size of the input:

```java
public int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {  // 3n + 2
        if (arr[i] == target) {             // n * 3
            return i;                       // 1
        }
    }
    return -1;                              // 1
}
```

  Going through this algorithm line by line we can identify each operation and how often it is run. Because, in the end we are only interested in how the runtime of the algorithm relates to the input size, we are specifically interested in loops and recursion. Counting the number of times each loop runs in the **linearSearch** algorithm above immediately tells us that the algorithm's complexity is O(n) for example.

- **Best-case, worst-case, and average-case analysis**: Another way to analyze the running time of an algorithm is to consider the best-case, worst-case, and average-case scenarios. The best-case scenario is the input that results in the shortest running time, the worst-case scenario is the input that results in the longest running time, and the average-case scenario is the input that results in an average running time. For example, the following Java code implements a binary search algorithm that has a best-case running time of O(1), a worst-case running time of O(log n), and an average-case running time of O(log n):

```java
public int binarySearch(int[] arr, int target) {
    int low = 0;
    int high = arr.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] < target) {
            low = mid + 1;
        } else if (arr[mid] > target) {
            high = mid - 1;
```

```
        } else {
            return mid;
        }
    }
    return -1;
```

To perform worst-case analysis on the linear search algorithm from the first example, we need to consider the input that will take the longest time to search. In this case, the worst-case input is the one where the target is not present in the array, because the algorithm will have to go through the entire array without finding a match.

If the array has n elements, then the worst-case running time of the linear search algorithm is O(n), because the algorithm will have to perform ***n*** comparisons (one for each element in the array). This means that the running time of the algorithm grows linearly with the size of the input.

Here is an example of an algorithm for which the worst-case and best-case complexity is quite different:

Bubble sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements that are out of order. It has a running time of $O(n^2)$ in the worst case, but a running time of O(n) in the best case.

Here is an implementation of bubble sort in Java:

```
public void bubbleSort(int[] arr) {
    boolean sorted = false;
    while (!sorted) {
        sorted = true;
        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                int temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
                sorted = false;
            }
        }
    }
}
```

The worst-case complexity of bubble sort occurs when the array is already sorted in the reverse order (i.e. the largest element is at the beginning and the smallest element is at the end). In this case, the inner loop (**for (int i = 0; i < arr.length - 1; i++)**) will have to run n - 1 times for each pass through the

outer loop (**while (!sorted)**), resulting in a total of ***n(n - 1) / 2*** comparisons. This is $O(n^2)$, because the number of comparisons grows as the square of the input size.

On the other hand, the best-case complexity of bubble sort occurs when the array is already sorted in the correct order (i.e. the smallest element is at the beginning and the largest element is at the end). In this case, the inner loop will never have to run, because the array is already sorted. This means that the running time of the algorithm will be *O(n)*, because the number of comparisons grows linearly with the size of the input.

Example: Maximum Subarray Sum

The maximum subarray sum is the maximum sum of a contiguous subsequence of an array. For example, given the array [1, -3, 2, 4, -1, 3], the maximum subarray sum is 7, which is obtained by selecting the subarray [2, 4].

Here is an implementation of the brute-force algorithm in Java:

```java
public int maxSubarraySum(int[] arr) {
    int maxSum = Integer.MIN_VALUE;
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j < arr.length; j++) {
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += arr[k];
            }
            maxSum = Math.max(maxSum, sum);
        }
    }
    return maxSum;
}
```

The inner loop (**for (int k = i; k <= j; k++)**) sums the elements of the subarray from index **i** to index **j**, and the outer two loops (**for (int i = 0; i < arr.length; i++)** and **for (int j = i; j < arr.length; j++)**) iterate over all possible start and end indices of the subarray. The variable **maxSum** is updated to the maximum value of the subarray sum at each iteration.

This algorithm has a running time of $O(n^3)$, because there are three nested loops. The outer two loops each have a running time of O(n), and the inner loop has a running time of O(n), resulting in a total running time of $O(n^3)$.

Although the brute-force algorithm is simple to implement and easy to understand, it is not very efficient for large inputs. There are more efficient algorithms for finding the maximum subarray sum, such as the divide-and-conquer algorithm, which has a running time of O(n log n).