Algorithm Analysis

1 Why Analyze Algorithms?

To compare how efficiently different algorithms solve the same problem, we analyze their use of time and space, primarily as a function of input size n.

Motivating Example: Linear Search vs Binary Search

Suppose you are given a sorted list of numbers and want to check whether a target number x appears in the list.

- **Linear Search:** Check each element one by one.
- **Binary Search:** Repeatedly divide the list in half and search the relevant half.

Both algorithms give the same result but can have vastly different performance. Analyzing them helps us compare their behavior without having to run code.

2 Big-O Notation

Big-O notation describes how the runtime of an algorithm grows in the worst case as the input size increases. It captures the dominant term and ignores constant factors.

2.1 Example: Linear Search

```cpp
int linearSearch(const vector<int>& arr, int x) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == x) return i;
    }
    return -1;
}
```

Worst-case: examine all n elements.

$$\Rightarrow \text{Time Complexity: } O(n)$$

Example: Two Sum (Brute Force)

Given an array and a target k, check if any two distinct elements sum to k.

```cpp
bool twoSum(const vector<int>& arr, int k) {
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] + arr[j] == k) return true;
        }
    }
    return false;
}
```

Checks all pairs: $\approx \frac{n(n-1)}{2}$ comparisons.

$$\Rightarrow \text{Time Complexity: } O(n^2)$$

2.2 Example: Matrix Multiplication

```cpp
void multiplyMatrices(const vector<vector<int>>& A,
                      const vector<vector<int>>& B,
                      vector<vector<int>>& C) {
    int n = A.size();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

$$\Rightarrow \text{Time Complexity: } O(n^3)$$

3 The Master Method

Divide-and-conquer algorithms often satisfy recurrences of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \quad \text{where } f(n) = O(n^d),\ a \geq 1,\ b > 1$$

Let $c = \log_b a$. Then:

- **Case 1:** If $a > b^d$, then $T(n) = O(n^{\log_b a})$
- **Case 2:** If $a = b^d$, then $T(n) = O(n^d \log n)$
- **Case 3:** If $a < b^d$ and $f(n)$ dominates recursive cost, then $T(n) = O(n^d)$ (under regularity condition)

Example: Binary Search

```cpp
int binarySearch(const vector<int>& arr, int low, int high, int x) {
    if (low > high) return -1;
    int mid = (low + high) / 2;
    if (arr[mid] == x) return mid;
    else if (arr[mid] < x)
        return binarySearch(arr, mid + 1, high, x);
    else
        return binarySearch(arr, low, mid - 1, x);
}
```

Recurrence: $T(n) = T(n/2) + 1$

- $a = 1,\ b = 2,\ f(n) = 1 = O(n^0) \Rightarrow d = 0$
- $\log_b a = \log_2 1 = 0 \Rightarrow$ Case 2

$$\Rightarrow \text{Time Complexity: } O(\log n)$$

Recursion Tree Sketch:

$$T(n) \rightarrow T(n/2) \rightarrow T(n/4) \rightarrow T(n/8) \rightarrow \cdots \rightarrow T(1)$$

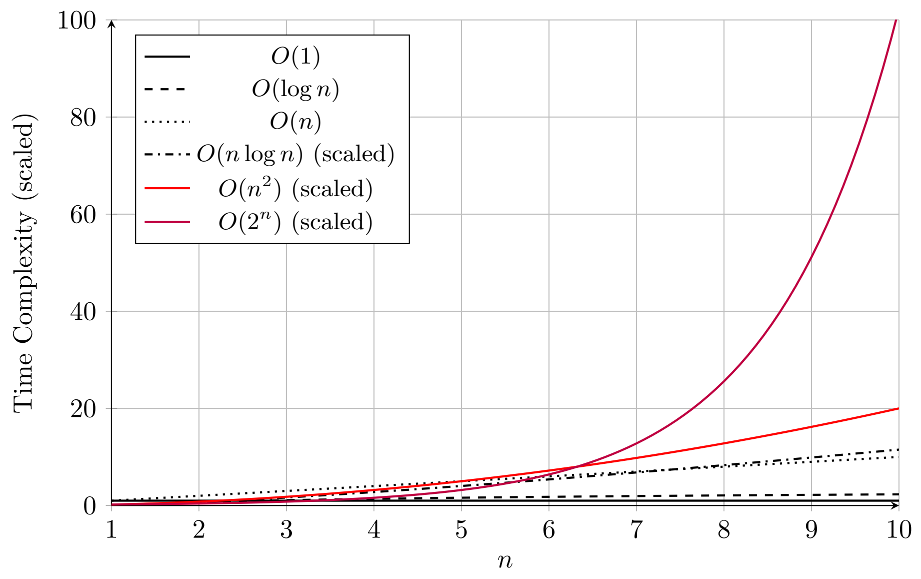

Figure 1: Comparison of Common Growth Rates (scaled for visualization)

Execution Time at $n = 100$ (1 Operation = 1 Second)

Assume that each basic operation takes 1 second. The table below shows the total runtime for different time complexities at $n = 100$:

Complexity	Expression at $n = 100$	Time (seconds)
$O(1)$	1	1 s
$O(\log n)$	$\log_2(100) \approx 7$	7 s
$O(n)$	100	100 s
$O(n \log n)$	$100 \cdot 7 \approx 700$	700 s
$O(n^2)$	$100^2 = 10{,}000$	2.8 hours
$O(2^n)$	$2^{100} \approx 1.27 \times 10^{30}$	$\sim 4 \times 10^{22}$ years

Note:

- $O(n^2)$ already becomes impractical beyond a few minutes.
- $O(2^n)$ is completely infeasible even for $n = 100$—illustrating the exponential explosion of brute-force algorithms.

4 Other Asymptotic Notations

Notation	Meaning
$O(f(n))$	Upper bound (at most proportional to $f(n)$)
$\Omega(f(n))$	Lower bound (at least proportional to $f(n)$)
$\Theta(f(n))$	Tight bound (exactly proportional to $f(n)$)
$o(f(n))$	Strictly smaller (grows slower than $f(n)$)
$\omega(f(n))$	Strictly larger (grows faster than $f(n)$)

5 Formal Proof Techniques

1. **Definition of O-notation:** Show $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$, with some constant $c > 0$.

Example: If $T(n) \leq 3n + 5$, we can choose $c = 4,\ n_0 = 5$ and prove:

$$3n + 5 \leq 4n \quad \text{for all } n \geq 5$$

2. Mathematical Induction: Assume $T(k) \leq c \cdot k$ for $k < n$, and show it holds for n.

3. Substitution Method: Assume a bound (e.g., $T(n) \leq c \cdot n$), substitute into the recurrence, and verify it holds.

4. Recursion Tree Method: Unroll the recurrence into a tree, compute work per level, and sum total work.

5. Proving Tight Bounds (Θ): Show both upper and lower bounds exist:

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \quad \text{for all } n \geq n_0$$