The Art and Science of Algorithm Evaluation: Understanding Time Complexity and Asymptotic Analysis

Minor In AI, IIT Ropar

12th March, 2025

Contents

1 Your Favorite Ice Cream and the Soul of an Algorithm **2**

2 Losing Your Keys and the Linear Search Algorithm **2**

3 Why Counting Steps Isn't Enough **3**

4 Finding the Basic Operation: The Heart of Algorithm Analysis **4**

5 Best Case, Worst Case, and Average Case Analysis **4**

6 Asymptotic Notation: Big O, Omega, and Theta **4**

7 What Does 'n' Really Mean? The Order of Growth **5**

1 Your Favorite Ice Cream and the Soul of an Algorithm

Imagine you have a favorite ice cream shop, or a juice center, or maybe a place that sells your favorite evening snacks. What makes you like it so much? Is it the taste? The ambiance? The friendly staff? You might have a feeling, a sense of *why* you love it. You might even rank it in your mind. "On a scale of 1 to 10, this place is a 9!"

Now, let's switch gears. Imagine you've written a computer program—an algorithm—to solve a problem. Can you simply say, "This algorithm is good because it looks nice, has beautiful loops, and feels right"? Unfortunately, no. The computing world demands more. It asks for *quantifiable* proof. It wants to know *how well* your algorithm performs. This is where the art and science of algorithm evaluation comes in.

Just like you might try to understand why you love your favorite ice cream, we need tools to understand the qualities of our algorithms. We need ways to measure them, to compare them, and to improve them. That's what this book is all about. We'll learn how to analyze algorithms, not based on subjective feelings, but on mathematical principles.

Figure 1: Happily eating ice cream vs. Coding lol

2 Losing Your Keys and the Linear Search Algorithm

Let's say you've lost your keys. You decide to search for them. You might check your coat pocket, then the shelf, then the drawer, and so on. You have a sequence of places you check. This methodical approach is similar to a simple search algorithm called a *linear search*.

A linear search is a straightforward way to find a specific item (like your keys) within a list of items (like possible hiding places). It works by checking each item in the list one by one, until you find the item you're looking for.

Here's how we can represent a linear search in Python:

```
def linear_search(arr, key):
  """
  Searches for a key in an array using a linear search.
  Args:
    arr: The array to search.
    key: The key to search for.
  Returns:
```

```
    The index of the key if found, otherwise -1.
  """
  i = 0 #iterator
  n = len(arr) #length of the array
  while i < n: #iterating through the array
    if arr[i] != key: # comparing each element with the key
      i = i + 1
    else:
      return i
  return -1
```

Listing 1: Linear Search Algorithm Implementation

Explanation:

- The `linear_search` function takes two arguments: the array `arr` to search and the `key` you're looking for.
- It initializes an index `i` to 0 and gets the length of the array `n`.
- It then enters a `while` loop that continues as long as `i` is less than `n` (meaning we haven't reached the end of the array).
- Inside the loop, it compares the element at the current index `arr[i]` with the `key`.
- If they are not equal, it increments `i` to check the next element.
- If they are equal, it means we've found the `key`, so the function returns the index `i`.
- If the loop completes without finding the `key`, it means the `key` is not in the array, so the function returns -1.

Example:

```
my_array = [10, 15, 34, 78, 22, 91]
search_key = 78
index = linear_search(my_array, search_key)
print(f"The element {search_key} is found at index: {index}")  # Output: 3
search_key = 100
index = linear_search(my_array, search_key)
print(f"The element {search_key} is found at index: {index}")  # Output: -1
```

Listing 2: Example Usage of Linear Search

3 Why Counting Steps Isn't Enough

Now that we have a linear search algorithm, how do we determine if it's any good? One initial thought might be to count the *number of steps* the algorithm takes. We could assign a "cost" to each line of code and add them up.

However, this approach, called *step counting*, has some serious flaws:

- **Hardware Dependency:** The time it takes to execute a single line of code can vary greatly depending on the computer's processor, memory, and other hardware components.
- **Compiler Optimizations:** The compiler (the program that translates your code into machine instructions) can optimize the code in various ways, making step counting inaccurate.
- **Input Sensitivity:** The number of steps can depend heavily on the specific input to the algorithm. In our `linear_search` example, if the key is the first element in the array, the algorithm finds it quickly. If the key is the last element or not in the array at all, it takes much longer.

Because of these issues, step counting is not a reliable way to compare the efficiency of algorithms. We need a more abstract and hardware-independent measure.

4 Finding the Basic Operation: The Heart of Algorithm Analysis

Instead of counting every single step, we focus on the *most important* step. This is called the *basic operation*. The basic operation is the operation that contributes the most to the algorithm's running time. It's the operation that is executed the most frequently.

In our `linear_search` example, the basic operation is the *comparison* between the array element `arr[i]` and the `key`: `arr[i] != key`. This comparison is performed in each iteration of the `while` loop.

By focusing on the basic operation, we can analyze the algorithm's efficiency in a more meaningful way. We're not concerned with the exact time each line of code takes to execute, but rather with *how many times* the most crucial operation is performed.

5 Best Case, Worst Case, and Average Case Analysis

The number of times the basic operation is executed can vary depending on the input. This leads to three important scenarios:

- **Best Case:** The best case occurs when the `key` is the first element in the array. In this case, the basic operation (the comparison) is executed only *once*.
- **Worst Case:** The worst case occurs when the `key` is the last element in the array or not in the array at all. In this case, the basic operation is executed n times, where n is the number of elements in the array.
- **Average Case:** The average case assumes that the `key` is equally likely to be at any position in the array. On average, the basic operation will be executed $n/2$ times.

```
# Code from Chapter 2 repeated to illustrate the best case, worst case & average case
def linear_search(arr, key):
  """
  Searches for a key in an array using a linear search.
  Args:
    arr: The array to search.
    key: The key to search for.
  Returns:
    The index of the key if found, otherwise -1.
  """
  i = 0 #iterator
  n = len(arr) #length of the array
  while i < n: #iterating through the array
    if arr[i] != key: # comparing each element with the key
      i = i + 1
    else:
      return i
  return -1
```

Listing 3: Linear Search Algorithm (repeated for analysis)

6 Asymptotic Notation: Big O, Omega, and Theta

To express the efficiency of algorithms in a standard way, we use *asymptotic notation*. Asymptotic notation describes how the running time of an algorithm grows as the input size increases.

There are three main types of asymptotic notation:

- **Big O Notation (O):** Big O notation describes the *worst-case* running time of an algorithm. It provides an *upper bound* on the growth rate. In our `linear_search` example, the worst-case running time is O(n), which means that the running time grows linearly with the input size.
- **Omega Notation (Ω):** Omega notation describes the *best-case* running time of an algorithm. It provides a *lower bound* on the growth rate. In our `linear_search` example, the best-case running time is $\Omega(1)$, which means that the running time is constant regardless of the input size.

- **Theta Notation (Θ):** Theta notation describes the *average-case* running time of an algorithm when the best and worst case are the same. It provides a *tight bound* on the growth rate.

Think of it this way:

- **Big O (O):** "The algorithm will *never* take longer than this." (Worst-case scenario)
- **Omega (Ω):** "The algorithm will *always* take at least this long." (Best-case scenario)
- **Theta (Θ):** "The algorithm will *usually* take this long." (Average-case scenario)

Back to Linear Search
Therefore we can represent our Linear search with all three notations:

- **Big O:** O(n)
- **Omega:** Ω(1)
- **Theta:** Can't use Theta notation, because the Best and Worst case are different.

7 What Does 'n' Really Mean? The Order of Growth

The 'n' in O(n), Ω(n), or Θ(n) represents the *size of the input*. But what does that *really* mean? It's about understanding the *order of growth*.

Let's revisit our `linear_search` example. Suppose we perform the search with a varying number of keys: 10, 20, 50, 100, 1000, and 10000.

In the best case, the number of comparisons is always 1. If we plot this on a graph, we get a horizontal line. The time taken is constant, regardless of the input size.

In the worst case, the number of comparisons is equal to the number of keys. If we plot this on a graph, we get a straight line sloping upwards. The time taken grows linearly with the input size.

That's what O(n) means: as the input size increases, the running time increases linearly.

If an algorithm had a running time of $O(n^2)$ (quadratic), the time taken would grow much faster as the input size increased.

The order of growth helps us compare algorithms and choose the most efficient one for a particular task.

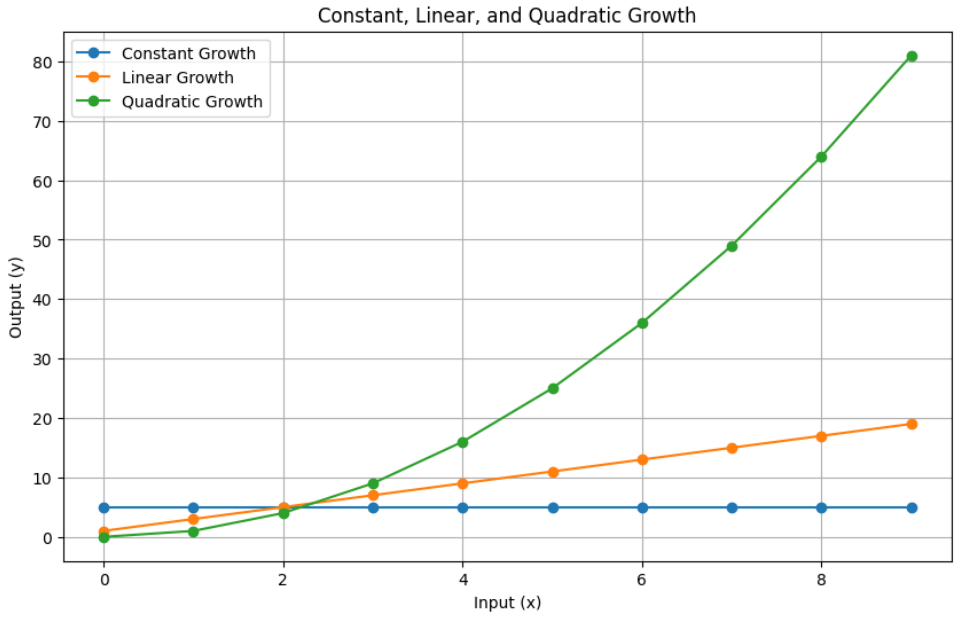

Figure 2: Graphs illustrating constant, linear, and quadratic growth.

Understanding time complexity and asymptotic analysis is crucial for becoming a skilled programmer and AI developer. By learning to evaluate algorithms, you can write code that is not only correct but also efficient and scalable. This knowledge will empower you to tackle complex problems and build innovative solutions.