Recursion and Searching Algorithms - MAD 102 Week 13 Notes

Hia Al Saleh

November 27th, 2024

Contents

1 Recursion

1.1 Definition

Recursion occurs when a function calls itself. It is a powerful technique used in algorithms, but improper usage can result in infinite loops or stack overflows. To prevent this, a base case is always required.

1.2 Structure of a Recursive Function

A recursive function typically contains:

1. **Base Case:** This is the condition that stops the recursion.
2. **Recursive Case:** The part where the function calls itself.

1.3 Example: Sum of Natural Numbers

The following function calculates the sum of natural numbers up to n:

```python
def sum_up_to(n):
    if n == 1:  # Base case
        return 1
    else:  # Recursive case
        return n + sum_up_to(n - 1)
```

1.4 Explanation of Execution

Starting with $n = 5$, the function calls itself multiple times until it reaches the base case:

$$\begin{aligned}\text{sum\_up\_to}(5) &= 5 + \text{sum\_up\_to}(4)\\ \text{sum\_up\_to}(4) &= 4 + \text{sum\_up\_to}(3)\\ &\dots\\ \text{sum\_up\_to}(1) &= 1\end{aligned}$$

2 Program Execution and the Stack

2.1 Stack Overview

In recursive functions, each function call is stored on the stack until it reaches the base case. This is known as ”pushing onto the stack.” Once the base case is reached, the results are returned, ”popping off the stack.”

2.2 Stack Overflow

A stack overflow occurs if too many recursive calls are made without reaching the base case. For example, the following faulty function causes infinite recursion:

```python
def infinite_recursion():
    return infinite_recursion()  # No base case
```

3 Searching Algorithms

3.1 Linear Search

Linear search examines each element in the list sequentially.

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index
    return -1  # Not found
```

Example:

- Input: `arr = [3, 5, 2, 8, 6], target = 8`
- Output: `3` (index of 8)

3.2 Binary Search

Binary search is efficient for sorted lists. It divides the list into halves:

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1  # Not found
```

4 Sorting Algorithms

4.1 Selection Sort

Selection sort repeatedly selects the smallest element from the unsorted part.

```python
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

4.2 Insertion Sort

Insertion sort inserts each element into its correct position in the sorted part:

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

5 Algorithm Efficiency

5.1 Big O Notation

- **Constant Time:** $O(1)$ - Example: Accessing an array element.
- **Linear Time:** $O(N)$ - Example: Linear search.
- **Logarithmic Time:** $O(\log N)$ - Example: Binary search.
- **Quadratic Time:** $O(N^2)$ - Example: Selection sort.