Understanding Sedgewick's Formula 2.1 and Linear Recursion

1 Introduction

This document reviews the discussions on Sedgewick's formula 2.1 from the 3rd edition of *Algorithms*. The focus is on understanding the concept of linear recursion as described by the formula and how it contrasts with other recursive techniques.

2 Sedgewick's Formula 2.1

Formula 2.1 in Sedgewick's book describes a recurrence relation:

$$C(N) = C(N-1) + N$$

for $N \geq 2$ with $C(1) = 1$. Here, $C(N)$ represents the cost or time complexity of solving a problem of size N.

2.1 Explanation of the Terms

- **C(N)**: The cost or time complexity of solving the problem for size N.
- **C(N-1)**: The cost of solving a subproblem of size $N-1$.
- **N**: Represents the additional work done at each level of the recursion, typically proportional to the size of the problem.
- $\mathbf{C(1) = 1}$: The base case, which states that the cost of solving the problem when $N = 1$ is 1 unit of time.

The recurrence relation describes a **linear recursive process** where each recursive call reduces the problem size by 1, and the total cost is the sum of the costs of these recursive calls.

3 Linear Recursion

3.1 Definition

Linear recursion refers to a type of recursion where each function call makes exactly one recursive call to solve a subproblem, and the problem size is reduced by a constant amount (typically by 1) with each call. The process is linear because the depth of the recursion tree is proportional to the size of the input.

3.2 Example: Summing an Array

Consider a simple example where we recursively sum the elements of an array:

Listing 1: Summing an Array using Linear Recursion

```
int sum(int arr[], int N) {
    if (N == 0) {
        return 0;  // Base case: empty array has a sum of 0
    } else {
        return arr[N-1] + sum(arr, N-1);  // Recursively sum the rest of the arr
    }
}
```

- **Base Case**: When $N = 0$, the function returns 0.
- **Recursive Step**: The function processes the last element, adds it to the sum of the remaining elements (which is handled by the recursive call).
- **Cost**: The cost at each level is constant, and there are N levels, leading to a total time complexity of $O(N)$.

4 Recursive Descent

4.1 Definition

Recursive descent refers to the process of progressively breaking down a problem into smaller subproblems through recursion. Each recursive call typically reduces the problem size, leading to a "descent" through the levels of recursion.

4.2 Example: Removing an Element from an Array

Consider a recursive function that removes the first element of an array and processes the rest:

Listing 2: Removing the First Element using Linear Recursion

```
void removeFirstElement(int arr[], int N) {
    if (N == 0) {
        return;  // Base case: empty array
```

```
    }

    // Process the first element (e.g., just printing it)
    std::cout << arr[0] << std::endl;

    // Recursive call to remove the first element
    removeFirstElement(arr + 1, N - 1);
}
```

- **Base Case**: When $N = 0$, the recursion stops.
- **Recursive Step**: The function processes the first element and recursively calls itself to handle the rest of the array.
- **Recursive Breakdown**: The problem size decreases by 1 with each recursive call, leading to a linear sequence of operations.

5 Key Insights

- Sedgewick's formula describes the **recursive breakdown** of a problem, where each step of the recursion performs a small amount of work and then calls itself on a smaller subproblem.
- The recursion described by $C(N) = C(N-1) + N$ is linear in nature because the problem size decreases by 1 with each step, leading to a total time complexity of $O(N^2)$ when unrolled.
- Understanding the concept of **recursive descent** is crucial for analyzing recursive algorithms, as it highlights how the problem is progressively simplified until it reaches the base case.