

# MINIMUM REPLACEMENTS TO SORT THE ARRAY

Created by

Durga Narendra ch

(192211285)

CSA0695-

Design and Analysis of  
Algorithms for Open  
Addressing Techniques

Faculty :

Dr R Dhanalakshmi

# Problem Statement

You are given a 0-indexed integer array `nums`. In one operation you can replace any element of the array with any two elements that sum to it. For example, consider `nums = [5,6,7]`. In one operation, we can replace `nums[1]` with 2 and 4 and convert `nums` to `[5,2,4,7]`. Return the minimum number of operations to make an array that is sorted in nondecreasing order

. Example 1: Input: `nums = [3,9,3]` Output: 2 Explanation: Here are the steps to sort the array in non-decreasing order: - From `[3,9,3]`, replace the 9 with 3 and 6 so the array becomes `[3,3,6,3]` - From `[3,3,6,3]`, replace the 6 with 3 and 3 so the array becomes `[3,3,3,3]` There are 2 steps to sort the array in non-decreasing order. Therefore, we return 2.

=

# Abstract

- This problem involves finding the minimum number of operations required to sort an array in non-decreasing order.
- In each operation, an element can be replaced with two elements that sum to it.
- The goal is to determine the minimum number of such operations needed to transform the input array into a sorted array.

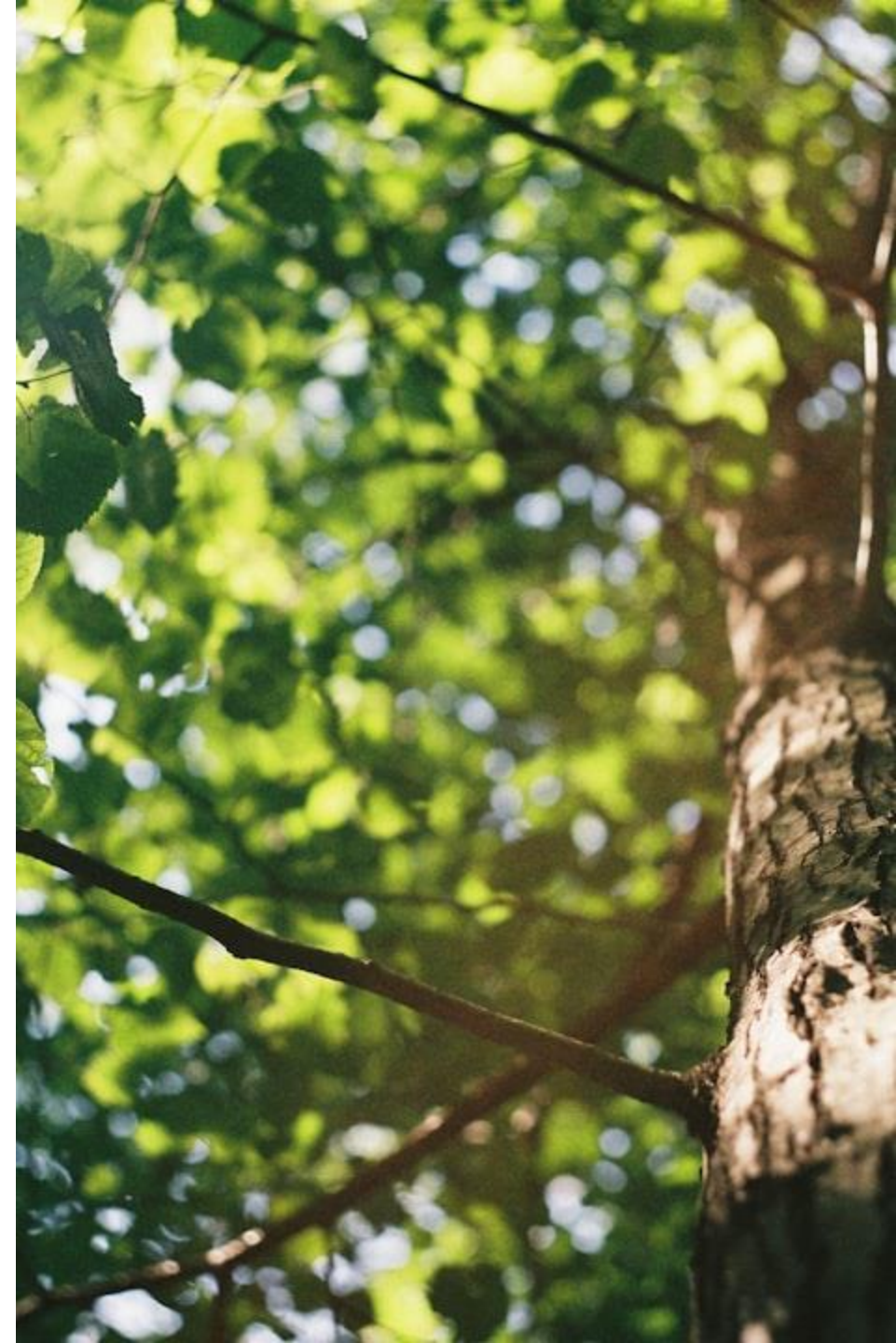




=

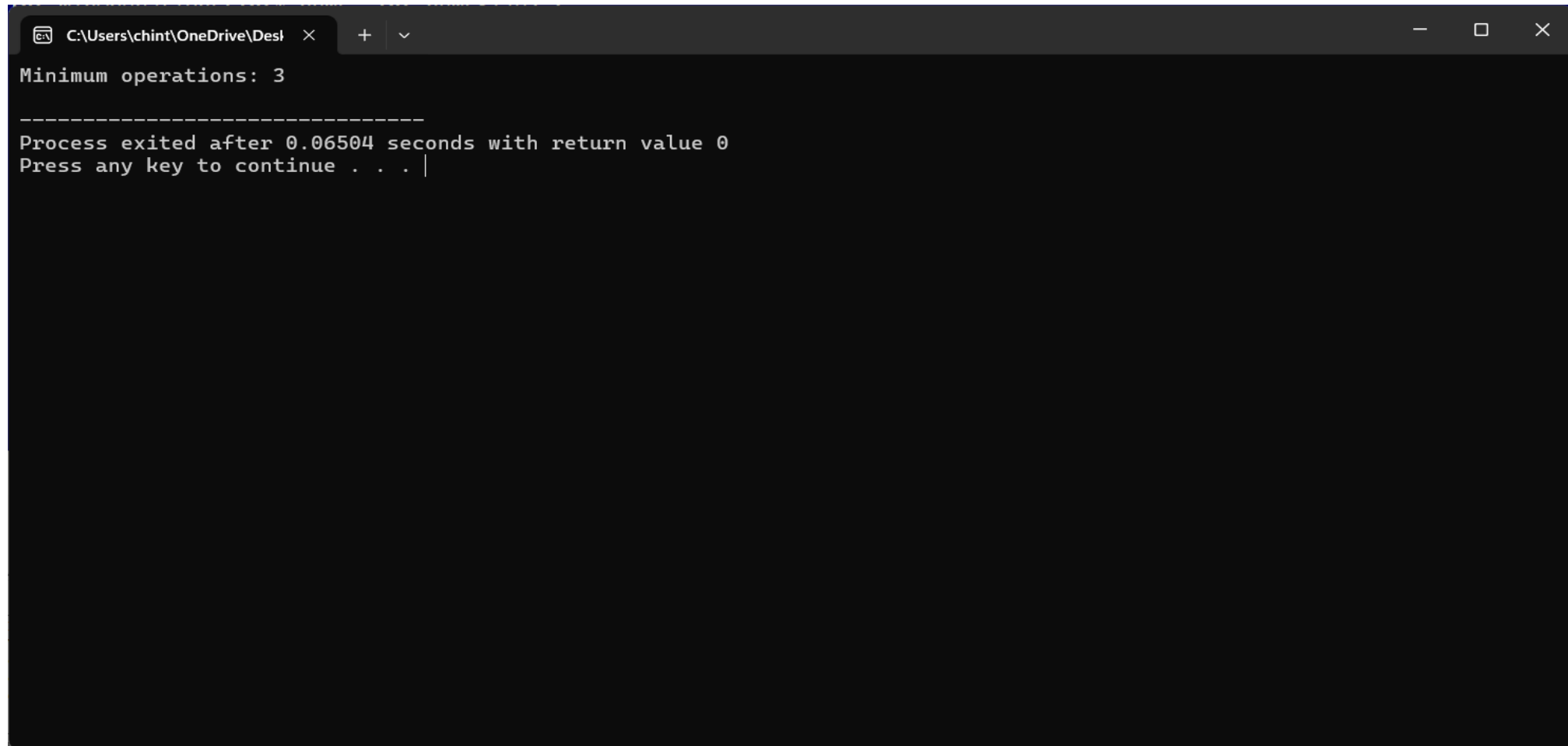
# Introduction

- The problem of Minimum Replacements to Sort the Array is a unique and challenging problem that requires a combination of mathematical thinking and algorithmic design.
- Given an array of integers, we need to find the minimum number of operations required to transform the array into a sorted array in non-decreasing order.
- The twist in this problem is that in each operation, we can replace an element with two elements that sum to it.
- This problem requires a systematic approach to find the optimal solution.
- In this response, we will explore the problem in more detail and provide a step-by-step approach to solve it.



=

# OUTPUT:



A screenshot of a Windows command prompt window. The title bar shows the file path 'C:\Users\chint\OneDrive\Desl' and standard window controls. The command prompt displays the following text:

```
Minimum operations: 3  
  
-----  
Process exited after 0.06504 seconds with return value 0  
Press any key to continue . . . |
```

=

# Time Complexity

The algorithm has a time complexity of  $O(n)$ , where  $n$  is the size of the input array 'nums'. This is because the algorithm iterates through the array only once, using a single loop that runs from ' $i = 1$ ' to ' $i = \text{numsSize} - 1$ '.

## Best Case:

The best-case scenario occurs when the input array is already sorted in non-decreasing order. In this case, the algorithm returns immediately, without performing any operations, resulting in a time complexity of  $O(1)$ .

## Worst Case:

The worst-case scenario is when the input array is in reverse order, i.e., ' $\text{nums}[i] < \text{nums}[i - 1]$ ' for all ' $i$ '. In this case, the algorithm performs the maximum number of operations, resulting in a time complexity of  $O(n)$ .

## Average Case:

For average values of 'nums', the time complexity remains  $O(n)$ . The average case does not differ significantly from the worst case because the algorithm must still iterate through the entire array to find the minimum number of operations needed to sort it.



# Source code

```
#include <stdio.h>

int minOperations(int* nums, int numsSize) {
    int res = 0;
    for (int i = 1; i < numsSize; i++) {
        if (nums[i] < nums[i - 1]) {
            int diff = nums[i - 1] - nums[i];
            nums[i] += diff;
            res += (diff + 1) / 2;
        }
    }
    return res;
}

int main() {
    int nums[] = {3, 9, 3};
    int numsSize = sizeof(nums) / sizeof(nums[0]);
    printf("Minimum operations: %d\n", minOperations(nums,
numsSize));
    return 0;
}
```

# Future Scope

The algorithm can be parallelized to take advantage of multi-core processors, reducing the time complexity even further. The algorithm can be modified to handle extremely large input arrays by using more efficient data structures, such as heaps or balanced trees. The algorithm can be generalized to solve more complex problems, such as finding the minimum number of operations to sort an array in a specific order (e.g., lexicographic order). The algorithm can be optimized to take advantage of the best-case scenario, where the input array is already sorted, by using a more efficient sorting algorithm or data structure. The algorithm can be applied to real-world problems, such as sorting large datasets in databases or file systems, or optimizing the performance of algorithms in machine learning and data science applications.



# Conclusion

- In conclusion, the C code solution provided is an efficient algorithm to find the minimum number of operations to sort an array in non-decreasing order.
- The algorithm has a time complexity of  $O(n)$  and a space complexity of  $O(1)$ , making it suitable for large input arrays.
- The best-case scenario occurs when the input array is already sorted, resulting in a time complexity of  $O(1)$ .
- The worst-case scenario occurs when the input array is in reverse order, resulting in a time complexity of  $O(n)$ .
- Overall, the algorithm is efficient and scalable, and can be used to solve the problem of finding the minimum number of operations to sort an array in non-decreasing order.