JoSDC'24
المسابقة الوطنية لتصميم الشرائح الإلكترونية
Jordan National Semiconductors Design Competition

JoSDC'24 Final Benchmarks

Benchmark 2
Three Sum Problem Solution

Author: Abdalrahman Ebdah

1. Problem Description

Problem Statement:

Given an array of integers ***nums***, find all unique triplets (a, b, c) such that, **but for simplicity we will consider our result to be the number of templets only**:

$$a + b + c = 0$$

Input:

- An array of integers ***nums*** with size ***n***.

Output:

- The number of unique triplets that sum to zero.

Example:

Input: $[-1,0,1,2,-1,-4]$

Output: 2 where the two triplets are $[[-1,-1,2],[-1,0,1]]$

Constraints:

1. Each triplet must be unique.
2. The order of the triplets and numbers within them does not matter.
3. You cannot use the same array element more than once in a triplet.

2. Matrix Multiplication Algorithm

To compute the product of two matrices, we iterate through each row of the first matrix and each column of the second matrix, calculating the dot product for each element in the result matrix. The following pseudocode and C++ implementation illustrate this process.

Two-Pointers Technique

```python
def count_zero_sum_triplets(nums):
    n = len(nums)
    triplet_count = 0

    for i in range(n - 2):
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        left, right = i + 1, n - 1
        while left < right:
            total = nums[i] + nums[left] + nums[right]
            if total == 0:
                triplet_count += 1
                left += 1
                right -= 1
                while left < right and nums[left] == nums[left 1]:
                    left += 1
                while left < right and nums[right] == nums[right + 1]:
                    right -= 1
            elif total < 0:
                left += 1
            else:
                right -= 1

    return triplet_count
```

Sort

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2

        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        merge(arr, left_half, right_half)
```

Merge

```python
def merge(arr, left_half, right_half):
    i = j = k = 0

    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1

    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1
```

3. Specifications and Complexity Analysis

3.1 Solution Approach

To solve the **Three Sum Problem**, one efficient approach involves **sorting the array** followed by the **two-pointer technique**.

Merge Sort

First, we apply merge sort, a divide-and-conquer algorithm that recursively splits the array into halves, sorts each half, and merges them back together. *Merge sort* ensures a stable and efficient sorting with a time complexity of O ($nlogn$), which is crucial for simplifying the next steps.

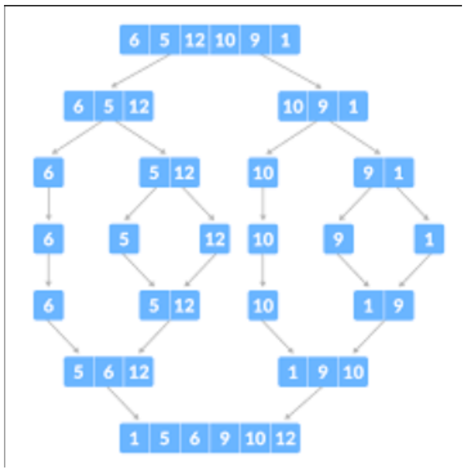

Figure 1 merge sort

Note: The purpose of using *merge sort* in solving the problem is to assess the processor's capability to efficiently handle recursive function calls.

Two-Pointers Technique

Once the array is sorted, the problem reduces to finding triplets (a, b, c) such that $a + b + c = 0$.

For each element **a**, we fix it and use two pointers: one starting from the next element (left pointer) and another from the end of the array (right pointer).

We calculate the sum **a**+***nums***[left]+***nums***[right] If the sum equals zero, we record the triplet and move both pointers inward while skipping duplicate values to ensure uniqueness. If the sum is less than zero, we move the left pointer to increase the sum, and if it's greater, we move the right pointer to decrease the sum.

Since each element is processed once after sorting, the two-pointer search runs O (n^2) time. Combining the sorting step of O ($nlogn$) with the two-pointer search O (n^2) the overall time complexity becomes O (n^2), which is efficient compared to the naive cubic approach O (n^3).

3.2 The Assembly Code and Instructions Memory

The assembly code contains main, sort and merge subroutines and many while loops in both the merge function and the two-pointers part.

The code consists of around 200 instructions, so the instruction memory depth should be at least 256 words.

3.3 Data Memory

Since merge sort is used in the solution, additional memory allocation is required for the auxiliary array used during the merging process. The size of this auxiliary array matches the size of the main array, ***nums***. Additionally, due to *merge sort's* recursive nature, the system must allocate stack space to store the activation record for each recursive call. Each activation record typically requires 4 words allocated for the return address and function parameters.
To determine the maximum stack expansion in memory, we first calculate the maximum recursion depth. In merge sort, the array is repeatedly split into two halves until each subarray contains a single element. Therefore, the maximum recursion depth corresponds to the base-2 logarithm of the array size, log2(n) To find the total stack space required, we multiply this recursion depth by the 4 words needed for each activation record.
In this benchmark, the ***nums*** array contains 1500 unique signed integers, requiring 1500 words for the main array and an additional 1500 words for the auxiliary array. The recursion depth is approximately $\log_2 1500 \approx 11.55$, rounded up to 12 levels. Multiplying this by 4 words per level results in 48 words of stack depth.
To prevent stack overflow and memory overlap with the arrays, the stack pointer (register **$29**) should be initialized to an address that accommodates the array and stack space. Since the stack grows downward toward lower memory addresses, initializing the stack pointer to 3050 ensures safe memory allocation and prevents conflicts.

4. Related Attached Files

The benchmark directory includes the following files, each serving a specific purpose in the project:

File Name	Functionality
ThreeSumProblem.asm	The assembly code for the problem solution using sort and two-pointers technique approach
three_sum_solution.py	The solution of the problem using python
Numbers.csv	Contains the numbers of the ***nums*** array in a comma-separated form
dataMemoryWORD.mif	A memory initialization file for those using word addressing mode, well-organized for easy use

Important Notes

- If you are using a different addressing mode, adjust your memory initialization file accordingly.
- Ensure you provide your machine code file with execution results, along with any other required files for benchmark processing and project evaluation.