Partition Array Into Two Arrays to Minimize Sum Difference

Comments 1
Favorited 0
Views 235

Java | ELI5 Explanation | Meet in the Middle & Two Pointer

Abedkp 64 Feb 21, 2023

This Question took me forever, and I had to gather bits and pieces from a bunch of solutions to get it, so I thought I would break this down and explain it in the best way possible.

Quick Subset Recap:
**
A subset is a set of which all the elements are contained in another set. Lets say the "other set" is this array [1,2,3,4]*.
All possible Subsets of this array will be the following:**

1 - {1}
2 - {2}
3 - {3}
4 - {4}
5 - {1,2}
6 - {1,3}
7 - {1,4}
8 - {2,3}
9 - {2,4}
10 - {3,4}
11 - {1,2,3}
12 - {1,2,4}
13 - {1,3,4}
14 - {2,3,4}
15 - {1,2,3,4}

Did you notice Something? The number of possible subsets of a set is the **2 ^ (Length of the original set) - 1.**

So our original set was of length **4**, and **2 ^ 4 - 1 = 15**.

One way we can generate a set is to count from 1 to^2 ^ (Length of set) - 1, but we have to do this IN BINARY

So we start with 0001, 0010, 0011, 0100..... until 1111.

If we choose to take only the positions where there is a '1' at each step, we can generate every subset by the time we reach 1111.

Notice now, that the time complexity of this operation is 2^N where N is the length of the set.

Part 1 - The Most Brute Force way to solve this problem

We have an array such as **[3,9,7,3]** . We can solve this by first getting the totalsum of the array, which in this case is
22. Then, we can generate every possible subset of this set, and everytime we run into a situation where **exactly half** of the
bits are '1', then we have ran into a valid partion of the array. Then if we can get the value of the remaining partion by taking
total and subtracting it by the sum of all the bit positions that were 1.

Example:
When we see 0011, we take, we take position 2 and 3. So 7 and 3. The sum is 10. Now total, which is 22 - 10, gives us the
sum of the remaining portion of the array, which is 12.

Now we can find the difference between 12 and 10 now, and record it.

We keep doing this till we reach 1111, adn we record every difference, and keep track of the minimum.

Now, you can notice that this algorithm will take 2^n. So lets look at what we can do to improve this.

Part 2 - What the heck is Meet in the Middle?

The Brute force way is very wasteful, because we are travelling through many binary strings that do not have exactly
N / 2 bits.

Instead of doing this for the full array, and generating every subset for the array of length N, we will do this for two seperate arrays of length N / 2. If we generate every subset for an array of Length N, there will be approximately an 2^N - 1 Operations.

However, if we do it for 2 arrays of length N / 2. There will be 2(N/2) - 1 + 2(N/2) - 1 operations, which is much less!

Lets prove it! Say an array is of length 8.

Generating every subset for an array of length 8 = 2^8 - 1 = **255 operations:**
Generating every subset for 2 arrays of length 4 = 2^4 -1 + 2^4 - 1 = **30 operations**

By doing this, we managed to increase the performance of the code by almost **901%!!!**

Now lets see how we can use this concept and apply it to solve this problem.

Part 3 - How to use Meet in the Middle to solve this problem

Now, lets say I were to split this array in two halfs, right in the middle [1,9,2,0,-5,7,3,6]:

So I will have **[1,9,2,0]** and **[-5,7,3,6]**.

How Can I try to get every possible partition from here?

One logical way to go about this is to think like this:

1 - I need 4 elements to get half the sum of the array, then I can just subtract that from total to get the second half.
2 - I can try every combination of 1 elements in the left array, and match each combination with every combination
of 3 elements from the right side.
3 - I can try every combination of 2 elements in the left array, and match each combination with every combination
of 2 elements from the right side.
4 - I can try every combination of 3 elements in the left array, and match each combination with every combination
of 1 elements from the right side

And so on and so on: The general formula is when we are taking **X** elements from left, we need **N/2 - x elements** from the right.

So now, all we have to do is this:

1 - Try every possible subset for left.
2 - Every time a bit is turned on (equal 1), we take that position of the bit, and add it to a sum
3 - We mainatin a HashMap<count of numbers, the sum of the numbers>
4 - We do this for both left and right.

Finally, we can check our left HashMap, and look at all the sums that can be generated by each X amount of numbers chosen from
left, then we look at take the complement of that to get half, which again, is N/2 - X from the right, and we can calculate each
possible partition this way.

Example with [1,5,3,7,9,4]:

Remember, the structure of the HashMap is Map<Amount of numbers chosen, The sum of the numbers>

So after constructing the HashMaps they will look like this in this example:

HashMap Left =
{
1: [1,5,3]
2: [6,4,8]
3: [9]
}

HashMap right =
{
1: [7,9,4]
2: [16,11,13]
3: [20]
}

So when we look at a sum where we only took 1 number from the left side, we have to look at sums where we took
2 from the right side to make 3 Numbers.

Then we take total and subtract by the sum of the numbers from left and right), and that gives us the value of the other half
of the array that are unused. Now, we can calculate the difference between both portions and keep track of the min.

One last tidbit to point out, is that we will also have to include 0 as a key in this map. And 0 will mean that we are not taking
any elemetns of the specific side(left or right), but when we apply our formula, this will allow us to look for 3 elements from
the other side, and it will play out the same from there. You can try commenting out the part of the code where we add the 0 key and see how some test cases fail to understand this if its not clear.

Technically, we have already improved the time complexity so much at this point, but we can do a bit better. This still wont pass on leetcode because this is a heavy optimization problem with tight time constraints.

So lets look at the final part and see how we can use two pointers to make it work.

Part 4 - Two Pointer

Finally, lets make the observation that the best possible case is where a partition is equal to the total sum of the array / 2. If this happens, then the absolute difference will be 0. **So we want this to happen!**

Therefore, we will sort all the lists of sums in our HashMap to beable to apply a two pointer solution.

So now imagine this, we have

List of Sums from the left part:
List of Sums from the right part:

Lets have a pointer starting from the begining for the leftSide, and a portion starting from the end of the right list.

We sum up left[0] and right[end -1]. Now we have a sum from some partition of half the array. We calculate the difference between
this sum adn the sum of the other half.

Finally, since we want the sum to gravitate towards total / 2 and not away from it, we check if the current sum is greater than
total / 2, if it is, we want to make this number sum smaller, so we decrease the pointer from right, else if we want to make it bigger
we increase the pointer from left. We can similarly apply a binary search here, but I chose to skip it since this problem is already so involved.

Thats it: I dont have the energy to proofread this after typing it lol, so I apologize in advance for any grammar and spelling mistakes! Let me know if you'd like some clarifications in the comments.

```
class Solution {

    public int minimumDifference(int[] nums) {
        int n = nums.length / 2;
        int total = Arrays.stream(nums).sum();
        HashMap<Integer,List<Integer>> leftMap = new HashMap();
        HashMap<Integer,List<Integer>> rightMap = new HashMap();

        createSumMappings(0, leftMap, nums, n);
        createSumMappings(n, rightMap, nums, n);
        leftMap.put(0,Arrays.asList(0));
        rightMap.put(0,Arrays.asList(0));

        int min = Integer.MAX_VALUE;
        for(int i = 0; i <= n; i++){
            List<Integer> left = leftMap.get(i);
            List<Integer> right = rightMap.get(n - i);
            Collections.sort(left);
            Collections.sort(right);

            int p1 = 0, p2 = right.size() - 1;
            while(p1 < left.size() && p2 >= 0){
                int sum = left.get(p1) + right.get(p2);
                int remaining = total - sum;
                int diff =  Math.abs(remaining - sum);
                min = Math.min(min,diff);
                if(sum > total / 2) p2--;
                else p1++;

            }
        }
        return min;
    }
    public void createSumMappings(int offSet, HashMap < Integer, List < Integer >
        for (int i = 1; i <= Math.pow(2, n) - 1; i++) {
            String binary = Integer.toBinaryString(i);
            binary = addLeadingZeros(n - binary.length()) + binary;
            int sum = 0;
            setBits = 0;
            for (int j = 0; j < binary.length(); j++) {
                if (binary.charAt(j) == '1') {
                    setBits++;
                    sum += nums[j + offSet];
                }
            }
            if (!map.containsKey(setBits)) map.put(setBits, new ArrayList());
            map.get(setBits).add(sum);
        }
    }
    public String addLeadingZeros(int zeroCount) {
        StringBuilder sb = new StringBuilder();
        while (zeroCount > 0) {
            sb.append('0');
            zeroCount--;
        }
        return sb.toString();
    }
}
```

2 Share Favorite

Comments (1)

Sort by: Best

Type comment here... (Markdown supported)

Preview Comment

54k 7 minutes ago

Take my knees, bro. It's awesome, helped me a lot

0 Reply