This piece of writing discusses different solutions to the problem posed by Codility under the name MinAbsSum found in the lesson 17 " Dynamic Programming" .

Once we have read the problem we could simply proceed with the algorithm that follows straightforward from the description of the problem. That is, to find the minimum value of $val(A, S)$ by generating and testing every permutation of the sequence S:

```python
def solution_1(A):
    import itertools
    min_val, N = abs(sum(A)), len(A)
    for S in list(itertools.product([1,-1], repeat=N)):
        val = 0
        for i in xrange(N):
            val += A[i]*S[i]
        val = abs(val)

        if val < min_val:
            min_val = val
    return min_val
```

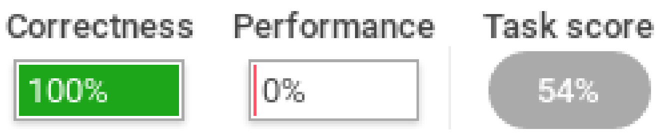
This exhaustive search implementation will produce correct results provided there is enough time and memory allowed.

We could also insert a halt signal to stop the search if the best plausible value has been found ($if\ min\_val == 0$) . That would improve the performance for most of the inputs, but it would not help in the worst case.

The first outer for-loop there is in the code will iterate the list of all the permutations of length N with repetition of 1 and -1. The number of permutations with repetition of two elements choosing N of them at a time is 2^N. So the length of the list that the first outer for-loop has to iterate is 2^N. The second or inner for-loop has to iterate N elements. If we neglect the time complexity of itertools.product, then, solution_1 has a time complexity of $O(N \cdot 2^N)$.

The memory requirement is mainly that of the list created for the outer for-loop to iterate. Knowing that the list is of length 2^N, and each element of the list is again a list of length N, we can say that solution_1 has a space complexity of $O(N \cdot 2^N)$.

Under the Codility test, this brute-force search will attain 100% Correctness, 0% Performance and an overall task score of 54%. The issues detected are runtime errors and timeout errors. In the performance tests sometimes the allowed time was exceeded sometimes not, but all of them ended with the Python interpreter raising "MemoryError".



A different approach to this problem could improve the performance of the solution using a different strategy, that is, changing the underlying algorithm.

A key observation when attempting to solve this problem is to note that if A is a set of numbers and we can find a subset of A with numbers that when summed together, the subset adds up to half the sum of all the elements of A, then, the rest of the elements of A must also add up to half the sum of A, and so, the optimal sequence S exists and our function should return 0. For example, S could be 1 for every element of the subset and -1 for the rest, effectively minimizing the value of $val(A, S)$ to 0.

In this manner, the problem of finding the minimum of $|\sum_{i=0}^{N-1} A[i] \cdot S[i]|$ for any S, converts to finding the closest number to half the sum of all the elements of A that can be reached by selecting and adding together elements of A. Once that closest number to half of the sum of A is found, we can calculate the difference with half the sum of A, multiply that difference by two and tell our function to return it if sum of A is an even number. If sum of A is an odd number and we are using the floor integer division, then the value to return will be twice the difference *plus one*.

Another good observation is to note that the given elements of A come as positives, negatives and zeros, but because $S[i]$ can be 1 or -1 to help us find the minimum, we can always select and change the sign of any element of A to match our needs. Knowing this, we can simplify the problem by making all the elements of A non-negative. For example, by applying abs to all the elements of A at the start of the program.

In the following, we will say that a number is "reachable" or it is a "reachable number" if there exists a subset of elements of A which sum equals the number.

To find the closest reachable number to half the sum of A we only need to search for numbers greater or smaller than half sum of A; we do not need to search on both sides. For a matter of ease, we are only going to consider all the numbers from 1 to half the sum of A (i.e., $sum(A)//2$) and check which ones can be reached.

To do this, we will first create an array called '$reachables$' of length $sum(A)//2 + 1$ filled with zeros to denote that a number represented with the same index in the array has not been found to be reachable yet. If a number is found to be reachable, the array element with the same index is changed to 1.

The process flow will be: Can number 1 be reached? Store the answer with 1 for yes or 0 for not. Can 2 be reached? ... Can number x be reached? The idea of dynamic programming is to consider subproblems that when solved help to solve the originally posed problem.

In more detail, the next algorithm is going to go through all the elements of A one by one in any other given, and it is going to mark as reachable all the numbers up to half the sum of all the elements of A. For every element e of A, we mark as reachable the numbers that are already reachable plus e, that is, the number that is already reachable (marked as reachable in a previous step) plus e gets now (in the current step) marked as reachable.

The number zero will be assumed to be reachable. And in the first instance of the loop, the first element e_1 of A marks the number $0 + e_1 = e_1$ as reachable. In the second instance, the second element e_2 of A marks both $0 + e_2$ and $e_1 + e_2$ as reachable, and so forth. Note that the order in which the elements of A arrive does not matter, for a basic addition is a commutative operation.

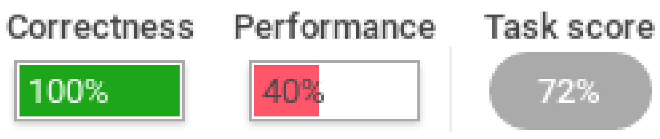
To mark all those reachable numbers we use a loop inside the one mentioned before, inside the one that runs through the elements of A. This second or inner loop checks all the indexes of the list *reachables* in search for already reachable numbers. However, we must note that a number that have just been marked as reachable on this pass through the reachable list, could mistakenly be taken as marked reachable on a previous pass. One way to remedy this is to search for reachable numbers from greatest to smallest, that is, to iterate the list *reachables* in reverse order.

```python
def solution_2(A):
    A = map(abs, A)
    total_sum = sum(A)
    half_sum = total_sum//2
    reachables = [1] + [0]*half_sum
    for elem in A:
       for num, reached in enumerate(reversed(reachables)):
          if reached and elem <= num:
             reachables[half_sum -num +elem] = 1

    for num, reached in enumerate(reversed(reachables)):
        if reached:
            if 2*half_sum == total_sum:
                return 2*num
            else:
                return 2*num +1
```

The outer for-loop iterates a list of length N, the inner for-loop iterates a list of maximum plausible length $N \cdot M$, where $M = max(abs(A))$. Therefore the worst-case time complexity of solution_2 is $O(M \cdot N^2)$.

With regards to the space complexity we can see that only the list *reachables* is created, so the worst-case space complexity is $O(N \cdot M)$.

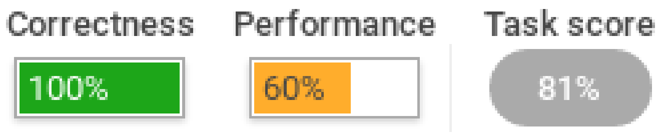
Under the Codility test the above implementation attains 100% Correctness, 40% Performance and a task score of 72%. The issues detected are timeout errors. The detected time complexity is $O(N^{**}2^{*}max(abs(A)))$.



The dynamic programming solution above can be improved by noting that the range of values of the elements of A is very small compared to the values N can take. Only between -100 and 100, and because we said that we do not need to worry about the sign, there are essentially only 101 different elements to deal with. If we count their occurrences before iterating we might improve the performance of the solution.

```
def solution_3(A):
    A = map(abs, A)
    counter = [0]*101 #max(A)
    for elem in A:
        counter[elem] += 1
    total_sum = sum(A)
    half_sum = total_sum//2
    reachables = [1] + [0]*half_sum
    for elem in xrange(1, 101):
        if not counter[elem]: continue
        for num in xrange(half_sum, -1, -1):
            if reachables[num]:
                i = 1
                while num +i*elem <= half_sum and i <= counter[elem]:
                    reachables[num +i*elem] = 1
                    i += 1

    return total_sum -2*(half_sum -list(reversed(reachables)).index(1))
```

Under the Codility test the above implementation attains 100% Correctness, 60% Performance and a task score of 81%. The issues detected are timeout errors and the detected time complexity is $O(N^{**}2^{*}max(abs(A)))$.



The score has improved slightly. However, there are three loops now. If we denote by D the number of different elements of A and by C the number of occurrences of each element of A, then, $D^{*}C == N$, and the worst-case time complexity of solution_3 is $O(D \cdot NM \cdot C) == O(N^2M)$.

With regards to the space complexity we can see that only the lists *counter* and *reachables* are created, so the worst-case space complexity is $O(M + N^{*}M)$.

To improve the time performance again we will need to remove the inner while loop and fuse it with the for-loop it is contained in. To do that, we will need to store more information in each step than just whether the next number (from 1 to $sum(A)//2$) can be reached. We will need to store how many occurrences remain in each step, that is, we will need to store the difference $counter[elem] - i$ in the previous solution_3. By storing correctly these values in each step we should be able to reduce the time complexity. That extra information could be stored in a new array or in the one we are already using.

We may observe that the list *reachables* only stores 0*s* and 1*s* to mean that the number represented by the index is reachable or not. We are clearly not making the most out of the resources being used. The list *reachables* could store any integer to denote more information, such as how many occurrences remain, at no extra memory expense.

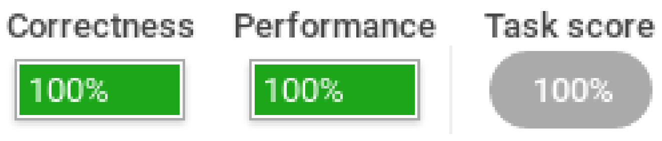
```
def solution_4(A):
    A = map(abs, A)
    counter = [0]*101 #max(A)
    for elem in A:
        counter[elem] += 1
    total_sum = sum(A)
    half_sum = total_sum//2
    reachables = [1] + [0]*half_sum
    for elem in xrange(1, 101):
        if not counter[elem]: continue
        for num in xrange(half_sum +1):
            if reachables[num]:
                reachables[num] = counter[elem] +1
            elif elem <= num and reachables[num -elem]:
                reachables[num] = reachables[num -elem] -1

    num = half_sum
    while not reachables[num]:
        num -= 1
    return total_sum - 2*num
```

The inner for-loop will have to iterate a list of worst-case length $(N^*M)//2$. And that inner loop will be executed as many times as different values there are in A (at most $M \leq 101$). So, the worst-case time complexity of solution_4 is $O(N^*M^{**}2)$.

The memory requirements are those of the '*counter*' and '*reachables*' lists. So the worst-case space complexity is $O(M + N^*M)$.

The announced expected worst-case space complexity in the problem description is O(N+sum(abs(A)), which might be a typo. In any case, according to the definition and properties of big-O, $O(N + N^*M) == O(N^*M) == O(M + N^*M)$ when n, $m \to \infty$.

Under the Codility test the above implementation attains 100% Correctness, 100% Performance and a task score of 100%. The solution obtained perfect score and the detected time complexity is O(N * max(abs(A))**2).



Correctness	Performance	Task score
100%	100%	100%

THE END

This text incorporates ideas from the Codility solution “Min Abs Sum with help of Peng Cao”.