Finding three elements in an array whose sum is closest to an given number

Given an array of integers, A_1, A_2, ..., A_n, including negatives and positives, and another integer S. Now we need to find three different integers in the array, whose sum is closest to the given integer S. If there exists more than one solution, any of them is ok.

You can assume all the integers are within int32_t range, and no arithmetic overflow will occur with calculating the sum. S is nothing special but a randomly picked number.

Is there any efficient algorithm other than brute force search to find the three integers?

arrays algorithm

edited Oct 5 '11 at 16:08
eeerahul
1,229 4 11 26

asked Jan 15 '10 at 8:46
ZelluX
22.4k 12 44 90

13 Answers

> Is there any efficient algorithm other than brute force search to find the three integers?

Yep; we can solve this in $O(n^2)$ time! First, consider that your problem `P` can be phrased equivalently in a slightly different way that eliminates the need for a "target value":

> **original problem `P` :** Given an array `A` of `n` integers and a target value `S`, does there exist a 3-tuple from `A` that sums to `S`?
>
> **modified problem `P'` :** Given an array `A` of `n` integers, does there exist a 3-tuple from `A` that sums to zero?

Notice that you can go from this version of the problem `P'` from `P` by subtracting your S/3 from each element in `A`, but now you don't need the target value anymore.

Clearly, if we simply test all possible 3-tuples, we'd solve the problem in $O(n^3)$ -- that's the brute-force baseline. Is it possible to do better? What if we pick the tuples in a somewhat smarter way?

First, we invest some time to sort the array, which costs us an initial penalty of O(n log n). Now we execute this algorithm:

```
for (i in 1..n-2) {
  j = i+1  // Start right after i.
  k = n    // Start at the end of the array.

  while (k >= j) {
    // We got a match! All done.
    if (A[i] + A[j] + A[k] == 0) return (A[i], A[j], A[k])

    // We didn't match. Let's try to get a little closer:
    //   If the sum was too big, decrement k.
    //   If the sum was too small, increment j.
    (A[i] + A[j] + A[k] > 0) ? k-- : j++
  }
  // When the while-loop finishes, j and k have passed each other and there's
  // no more useful combinations that we can try with this i.
}
```

This algorithm works by placing three pointers, `i`, `j`, and `k` at various points in the array. `i` starts off at the beginning and slowly works its way to the end. `k` points to the very last element. `j` points to where `i` has started at. We iteratively try to sum the elements at their respective indices, and each time one of the following happens:

- The sum is exactly right! We've found the answer.
- The sum was too small. Move `j` closer to the end to select the next biggest number.
- The sum was too big. Move `k` closer to the beginning to select the next smallest number.

For each `i`, the pointers of `j` and `k` will gradually get closer to each other. Eventually they will pass each other, and at that point we don't need to try anything else for that `i`, since we'd be summing the same elements, just in a different order. After that point, we try the next `i` and repeat.

Eventually, we'll either exhaust the useful possibilities, or we'll find the solution. You can see that this is $O(n^2)$ since we execute the outer loop O(n) times and we execute the inner loop O(n) times. It's possible to do this sub-quadratically if you get really fancy, by representing each integer as a bit vector and performing a fast Fourier transform, but that's beyond the scope of this answer.

Note: Because this is an interview question, I've cheated a little bit here: this algorithm allows the selection of the same element multiple times. That is, (-1, -1, 2) would be a valid solution, as would (0, 0, 0). It also finds only the *exact* answers, not the closest answer, as the title mentions. As an exercise to the reader, I'll let you figure out how to make it work with distinct elements only (but it's a very simple change) and exact answers (which is also a simple change).

edited Mar 18 '14 at 19:11

answered Jan 15 '10 at 9:23

John Feminella
147k 27 247 294

Thanks for fantastic, clear explanation. – xxx Jan 15 '10 at 9:36

4 It seems the algorithm can only find 3-tuple that *equals* to S, not *closest* to S. – ZelluX Jan 15 '10 at 10:37

2 ZelluX: As I mentioned in the note, I didn't want to give too much away since it's an interview problem. Hopefully you can see how to modify it so that it gets you the closest answer, though. (Hint: one way is to keep track of the closest answer so far and overwrite it if you find a better one.) – John Feminella Jan 15 '10 at 10:40

10 what if we don't modify the problem statement, instead we will search for aj and ak that sum to ai +S. – Boolean May 20 '11 at 14:59

1 There is an issue when subtracting S/3 from A[]. Please see details below. – galactica Jul 30 '13 at 3:23

certainly this is a better solution because it's easier to read and therefore less prone to errors. The only problem is, we need to add a few lines of code to avoid multiple selection of one element.

Another O(n^2) solution (by using a hashset).

```
// K is the sum that we are looking for
for i 1..n
    int s1 = K - A[i]
    for j 1..i
        int s2 = s1 - A[j]
        if (set.contains(s2))
            print the numbers
    set.add(A[i])
```

edited Feb 24 at 23:55

Tuxdude
12.7k 5 43 60

answered Feb 19 '12 at 4:25

Cem
316 3 4

awesome..solution – Ram Jul 12 '12 at 18:00

4 Downside is O(N) storage, rather than doing it in-place. – Charles Munger Nov 16 '12 at 21:10

2 Using a hashset isn't strict O(n^2) since the hash set can degenerate in rare occasions, resulting in up to linear lookup times. – Ext3h Jul 24 '14 at 17:24

How about something like this, which is O(n^2)

```
for(each ele in the sorted array)
{
    ele = arr[i] - YOUR_NUMBER;
    let front be the pointer to the front of the array;
    let rear be the pointer to the rear element of the array.;

    // till front is not greater than rear.
    while(front <= rear)
    {
        if(*front + *rear == ele)
        {
            print "Found triplet "<<*front<<","<<*rear<<","<<ele<<endl;
            break;
        }
        else
        {
```

```
            // sum is > ele, so we need to decrease the sum by decrementing rear
pointer.
            if((*front + *rear) > ele)
                decrement rear pointer.
            // sum is < ele, so we need to increase the sum by incrementing the
front pointer.
            else
                increment front pointer.
        }
    }
```

This finds if sum of 3 elements is exactly equal to your number. If you want closest, you can modify it to remember the smallest delta(difference between your number of current triplet) and at the end print the triplet corresponding to smallest delta.

edited Jan 15 '10 at 9:36

answered Jan 15 '10 at 9:26

codaddict

205k 39 325 412

if you want to find k elements to get the sum what is the complexity? How you deal with this? – coder_15 Feb 18 '12 at 6:30

Here k is any number like 2,3,5,6.... – coder_15 Feb 18 '12 at 6:31

With this approach, complexity for k elements is O(n^(k-1)) for k >= 2. You need to add an outer loop for every additional summand. – Ext3h Jul 24 '14 at 17:40

Here is the C++ code:

```
bool FindSumZero(int a[], int n, int& x, int& y, int& z)
{
    if (n < 3)
        return false;

    sort(a, a+n);

    for (int i = 0; i < n-2; ++i)
    {
        int j = i+1;
        int k = n-1;

        while (k >= j)
        {
            int s = a[i]+a[j]+a[k];

            if (s == 0 && i != j && j != k && k != i)
            {
                x = a[i], y = a[j], z = a[k];
                return true;
            }

            if (s > 0)
                --k;
            else
                ++j;
        }
    }

    return false;
}
```

answered Sep 22 '13 at 9:32

Peter Lee

3,775 4 36 79

John Feminella's solution has a bug.

At the line

```
if (A[i] + A[j] + A[k] == 0) return (A[i], A[j], A[k])
```

We need to check if i,j,k are all distinct. Otherwise, if my target element is `6` and if my input array contains `{3,2,1,7,9,0,-4,6}` . If i print out the tuples that sum to 6, then I would also get `0,0,6` as output . To avoid this, we need to modify the condition in this way.

```
if ((A[i] + A[j] + A[k] == 0) && (i!=j) && (i!=k) && (j!=k)) return (A[i], A[j],
A[k])
```

edited Aug 27 '13 at 16:29

answered Aug 27 '13 at 16:23

Rambo7

465 4 9

2 John Feminella solution is just to present algorithm to solve problem, he has also specified that his solution wouldn't work for distinct number condition and you have to modify above code little bit which he has left for reader. – EmptyData Dec 17 '13 at 12:22

Note that we have a sorted array. This solution is similar to John's solution only that it looks for the sum and does not repeat the same element

```
#include <stdio.h>


int checkForSum (int arr[], int len, int sum) { //arr is sorted


        int i;
        for (i = 0; i < len ; i++) {
                int left = i + 1;
                int right = len - 1;

                while (right > left) {

                        printf ("values are %d %d %d\n", arr[i], arr[left],
arr[right]);
                        if (arr[right] + arr[left] + arr[i] - sum == 0) {
                                printf ("final values are %d %d %d\n", arr[i],
arr[left], arr[right]);
                                return 1;
                        }
                        if (arr[right] + arr[left] + arr[i] - sum > 0)
                                right--;
                        else
                                left++;

                }

        }
        return -1;
}



int main (int argc, char **argv) {

        int arr[] = {-99, -45, -6, -5, 0, 9, 12, 16, 21, 29};
        int sum = 4;
        printf ("check for sum %d in arr is %d\n", sum, checkForSum(arr, 10, sum));


}
```

answered Oct 23 '10 at 3:13

Pasada
176 1 3

Very simple N^2*logN solution: sort the input array, then go through all pairs A_i, A_j (N^2 time), and for each pair check whether (S - A_i - A_j) is in array (logN time).

Another O(S*N) solution uses classical dynamic programming approach.

In short:

Create an 2-d array V[4][S + 1]. Fill it in such a way, that:

V[0][0] = 1, V[0][x] = 0;

V1[A_i]= 1 for any i, V1[x] = 0 for all other x

V[2][A_i + A_j]= 1, for any i, j. V[2][x] = 0 for all other x

V[3][sum of any 3 elements] = 1.

To fill it, iterate through A_i, for each A_i iterate through the array from right to left.

answered Jan 15 '10 at 9:17

Olexiy
954 4 11

slight change to the first algorithm.. if the element doesn't exist, then at the end of the binary search, we'd have to look at the element on the left, current, and right to see which one gives the closest result. – Anurag Jan 15 '10 at 9:25

There is a problem in John's algorithm above, even if it tries to find only exact answers. When subtracting S/3 from A[], there is a chance to introduce wrong answers:

input: [2, 3, -1, 1], target = 5 correct output: [2,3,-1] or [2,3,1] and difference is 1.

Based on John's algorithm above, it goes like the following: subtract S/3 = 1 from A[]:

```
A'[] = [1, 2, -2, 0].
```

If choose output = [2, -2, 0], then we've got an exact match. Retrieving these elements' indices in the original A[], we get output elements = [3, -1, 1], which is wrong.

answered Jul 30 '13 at 3:23

galactica
237 4 18

That is naturally not integer division he is talking about. – poroszd Mar 9 '14 at 17:53

1 @poroszd: but the definition of the modified array (in John Feminella's answer") still stipulates that the modified array is an array of integers, not an array of floating point values. – Jonathan Leffler Sep 2 '14 at 22:49

@JonathanLeffler, thanks for getting my 10 pts back! :-) – galactica Sep 3 '14 at 1:15

Modified problem: Given an array , find 2 numbers summing to closest to a given target s. I think, this can be done in O(n), Sorting may not be required.

1. First pass has the array using bitmap array. Two arrays one for negative one for positive.
2. Second pass for each element in the array `A[i]` , `k= S-A[i]` , if `k` or `-k` is found in the hash map gr8 we have the answer, print it.

So we see it is O(n).

edited Aug 1 '11 at 10:13

Alberto Zaccagni
14.5k 4 37 76

answered Aug 1 '11 at 5:52

Dr.Sai
1 1

1 @Albero Zaccagni if question is "in given array ,find two integer such that sum is near to s" then your solution could not work. – user948587 Sep 16 '11 at 10:14

Another solution that checks and fails early:

```
public boolean solution(int[] input) {
        int length = input.length;

        if (length < 3) {
            return false;
        }

        // x + y + z = 0  => -z = x + y
        final Set<Integer> z = new HashSet<>(length);
        int zeroCounter = 0, sum; // if they're more than 3 zeros we're done

        for (int element : input) {
            if (element < 0) {
                z.add(element);
            }

            if (element == 0) {
                ++zeroCounter;
                if (zeroCounter >= 3) {
                    return true;
                }
            }
        }

        if (z.isEmpty() || z.size() == length || (z.size() + zeroCounter ==
length)) {
            return false;
        } else {
            for (int x = 0; x < length; ++x) {
                for (int y = x + 1; y < length; ++y) {
                    sum = input[x] + input[y]; // will use it as inverse addition
                    if (sum < 0) {
                        continue;
                    }
                    if (z.contains(sum * -1)) {
                        return true;
                    }
                }
            }
        }
        return false;
    }
```

I added some unit tests here: GivenArrayReturnTrueIfThreeElementsSumZeroTest.

If the set is using too much space I can easily use a java.util.BitSet that will use O(n/w) space.

answered Aug 11 at 14:55

moxi
183 5

Reduction : I think @John Feminella solution O(n2) is most elegant. We can still reduce the A[n] in which to search for tuple. By observing A[k] such that all elements would be in A[0] - A[k], when our search array is huge and SUM (s) really small.

A[0] is minimum :- Ascending sorted array.

s = 2A[0] + A[k] : Given s and A[] we can find A[k] using binary search in log(n) time.

answered Oct 14 '11 at 13:40

d1val
166 3

I think it is better first to sort the array which costs nlogn. let A is the array.here x is the target element so to find the sum of two integers which gives x. (A[i], x-A[i]) this is pair so

for i=1 to n

Binarysearch(x-A[i])

T(n)= O(nlogn) + O(n) + logn

edited Jul 21 at 10:02

answered Jul 21 at 9:44

Habte
1 1

First Sort the number : O(n log n).

Modified problem: Given an array , find 2 numbers summing to closest to a given target.

I think, this can be done in O(n) (after sorting).

Pick each number n[i] and run the modified problem for S-n[i] as target. this takes O(n * n).

This overall takes O(n log n) + n*O(n).

How does it look?

answered Jun 22 '11 at 10:30

xyz
2,420 7 35 72
