Sum of Two Terms

Mac Radigan

Are The Individual Terms Of A Two-Term Sum Present In An Array?

Overview

Given an array of integers, $\mathbb{X}$, determine whether or not there exist two elements, m and n, in the array (at different positions) whose sum is equal to some target value, Σ.

Algorithm #1

Background

Algorithm 1 makes use of the fact that:

$$n = (n - k) + k$$

holds true for any integers n and k.

Therefor, given knowledge of the elements present in the sequence $\mathbb{X}$ between 0 and Σ, we can apply the above formula as a test for existence.

This algorithm makes an assumption about the maximal sum that will be encountered, M, and uses this size in allocating a histogram.

Implementation

Initially size a histogram of unit bin size based on the expected maximum sum to be supported (say, upper bound M).

Build a histogram of unit bin size from the input sequence.

$$h_k = |\{k : k \in \mathbb{X}\}|$$

Scan the histogram up to half the number of bins, applying the formula:

$$n = (n - k) + k$$

If the above equation holds for any element encountered, then two terms have been found that add to the given sum.

Note that in the implementation we are using the C++ behavior that of integral type having a truth value of T if and only if their register value is non-zero.

Algorithm 1 Has Two Sum Terms

---

    given set of terms $\mathbb{X}$, goal sum Σ
    for each $x \in \mathbb{X}$ **do**
        $h_x \leftarrow h_x + 1$ ⊬ build a histogram
    end for each
    for $k \leftarrow 0 \cdots |\mathbb{X}|$ **do**
        if $\Sigma \mathop{=\!=} h_{\Sigma-k} + h_k$ **then**
            return ⊤   ⊬ check sum of terms holds
        end if
    end for
    if $2 \mid \Sigma \wedge h_{\lceil \frac{k}{2} \rceil} \geq 2$ **then**
        return ⊤   ⊬ check two terms in histogram center
    end if
    return ⊥

Performance

For a sequence xs, having N elements, we have:

Measure	Performance
average time complexity	O(N)
worst case time complexity	N + 1/2 M
constant space complexity	M

Note that time complexity assumptions for the average case are not strictly valid without knowledge of the underlying statistical distribution of the input data.

Note that for large M, the space complexity of this algorithm may be substantial. Algorithm 2 provides better performance when M is large.

```
// ============================================================================
// has_two_sum_terms (Algorithm 1)
// ============================================================================
```

```
4    //
5    //   returns true if there exists elements m and n in sequence xs such that
6    //                 sum = m + n, where xs and sum are given
7    //
8    //   inputs:
9    //
10   //     xs : vector<T>   - a sequence of terms to consider
11   //
12   //     sum : T          - the specified target sum for testing terms
13   //
14   //   template paramters:
15   //
16   //     T : class        - the data type of the terms and target sum
17   //
18   //     M : size_t       - an upper bound on the expected sum
19   //
20   //   returns:
21   //
22   //     has_terms : bool - true  if the input sequence contains two elements
23   //                              equal to a given sum
24   //                       false otherwise
25   //
26   // ----------------------------------------------------------------------------
27   template<class T, std::size_t M>
28   class SequenceCheck
29   {
30    public:
31     inline bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
32     {
33       // build a histogram
34       hist_.fill(0);
35       for(auto x : xs) if(x<=sum) hist_[x]++;
36       // each element up to half of the histogram
37       for(auto k=0; k<std::floor(sum/2.0); ++k)
38       {
39         // test if n = (n-k) + k
40         if( hist_[sum-k] && hist_[k] ) return true;
41       } // k = 0...sum/2
42       // special case at ceil(N/2) when N odd, must have at least two terms
43       if( !(sum%2) && hist_[std::ceil(sum/2.0)]>1 ) return true;
44       return false; // otherwise no such two terms
45     } // has_two_terms
46    private:
47     std::array<T,M> hist_{};
48   }; // SequenceCheck
```

Algorithm #2

Note that Algorithm 2 currently does not enforce the selection of two terms in the sequence to be from two distinct different positions.

Background

Algorithm 2 makes use of the algebraic group property that every number has an inverse, and thus we may rewrite:

$$\Sigma = m + n \leftrightarrow n = \Sigma - m$$

Thus for each element m encountered in $\mathbb{X}$, we know uniquely of a corresponding n in $\mathbb{X}$ that we seek.

Therefor, we may scan xs once to identify its compliment with respect to Σ.

Now, with a set of compliments, say $\bar{\mathbb{X}}$, we may scan $\mathbb{X}$ again to determine if any element exists in $\bar{\mathbb{X}}$.

If we find that an element in $\mathbb{X}$ is found in the set of compliments, then we know the sum can be produced from two terms that exist in the sequence.

Since it is also required that the terms in the sum are at distictly different positions in the sequence, use an unordered map for the set of compliments, and use the position to track the position of the original term. When checking the compliment, verify that the position is also distinct from the original term.

Implementation

Initially an empty, unordered map (uses a hash map implementation). Call this the compliment map $\bar{\mathbb{X}}$.

Scan the input sequence, $\mathbb{X}$, for each x in $\mathbb{X}$. For each x, compute the compliment of the sum, Σ, and x, say: $\bar{c} = \Sigma - x$, and insert the compliment $\bar{c}$ and the ordinal position of x (say k) into $\bar{\mathbb{X}}$.

Scan the input sequence, $\mathbb{X}$, for each x in $\mathbb{X}$ again, checking the compliment map $\bar{\mathbb{X}}$ for x. If x exists in $\bar{\mathbb{X}}$, and the position is distinct from the map's value of k, then we have found two terms that produce the sum.

If the end of the sequence is reached without finding a matching x in the compliment map, $\bar{\mathbb{X}}$, then there are no two terms in
$mathbbX$ that will produce the sum.

Performance

For a sequence $\mathbb{X}$, having N elements, we have:

Algorithm 2 Has Two Sum Terms

---

    given set of terms $\mathbb{X}$, goal sum Σ
    for $k \leftarrow 0 \cdots |\mathbb{X}|$ **do**
        $\bar{x} \leftarrow \Sigma - \mathbb{X}_k$
        $\mathbb{F}_{\bar{x}} \leftarrow k$ ⊢ map of compliments to position
    end for
    for $k \leftarrow 0 \cdots |\mathbb{X}|$ **do**
        if $\mathbb{X}_k \in \bar{\mathbb{X}} \wedge \mathbb{F}_k \neq k$ **then**
            return $\top$ ⊢ check are compliments present and distinct
        end if
    end for
    return $\bot$

Measure	Performance
average time complexity	O(N)
best case time complexity	N + 1
worst case time complexity	2 * N
average case space complexity	O(N)
best case space complexity	1
worst case space complexity	N

Note that the space complexity is dependent only on the number of unique elements in the input sequence ($\mathbb{X}$).

Note that for small M, algorithm 1 can have better worst case time complexity as well as substantially better space complexity. These assumptions can be further bounded and refined given statistical knowledge of the input data.

```cpp
// ============================================================================
// has_two_sum_terms (Algorithm 2)
// ============================================================================
//
//   returns true if there exists elements m and n in sequence xs such that
//              sum = m + n, where xs and sum are given
//
//   inputs:
//
//      xs : vector<T>   - a sequence of terms to consider
//
//      sum : T          - the specified target sum for testing terms
//
//   template paramters:
//
//      T : class        - the data type of the terms and target sum
//
//   returns:
//
//      has_terms : bool - true  if the input sequence contains two elements
//                               equal to a given sum
//                         false otherwise
//
// ----------------------------------------------------------------------------
bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
{
```

```
30    {
31      const T x = xs[k];
32      const auto diff = sum - x; // caution: T must be a signed datatype
33      compliments[diff] = k;
34    } // foreach index k of x in xs
35    // scan the input sequence again to identify if any compliments are present
36    for(auto k=0; k<xs.size(); ++k)
37    {
38      const T x = xs[k];
39      const auto it = compliments.find(x);
40      // check that the compliment is at a distict position from the original term
41      if( (it!= compliments.end()) && (k != it->second) ) return true;
42    } // foreach x in xs
43    return false; // otherwise no such two terms
44  } // has_two_terms
```

Source Code

```
1   // sum-two-terms.cc
2   // Mac Radigan
3
4
5     #include <array>
6     #include <assert.h>
7     #include <cmath>
8     #include <cstdlib>
9     #include <iomanip>
10    #include <iostream>
11    #include <map>
12    #include <sys/types.h>
13    #include <unordered_map>
14    #include <vector>
15
16
17    // ============================================================================
18    // has_two_sum_terms (Algorithm 1)
19    // ============================================================================
20    //
21    //   returns true if there exists elements m and n in sequence xs such that
22    //                 sum = m + n, where xs and sum are given
23    //
24    //   inputs:
25    //
```

```
26      //     xs : vector<T>    - a sequence of terms to consider
27      //
28      //     sum : T           - the specified target sum for testing terms
29      //
30      //   template paramters:
31      //
32      //     T : class         - the data type of the terms and target sum
33      //
34      //     M : size_t        - an upper bound on the expected sum
35      //
36      //   returns:
37      //
38      //     has_terms : bool - true  if the input sequence contains two elements
39      //                               equal to a given sum
40      //                        false otherwise
41      //
42      // ----------------------------------------------------------------------
43      //
44      //
45      // Background:
46      //
47      //   Algorithm 1 makes use of the fact that:
48      //     n = (n-k) + k
49      //   holds true for any integers n and k.
50      //
51      //   Therefor, given knowledge of the elements present in the sequence xs
52      //     between 0 and sum, we can apply the above formula as a test for
53      //     existence.
54      //
55      //   This algorithm makes an assumption about the maximal sum that will
56      //     be encountered, M, and uses this size in allocating a histogram.
57      //
58      //
59      // Implementation:
60      //
61      //   Initially size a histogram of unit bin size based on the expected
62      //     maximum sum to be supported (say, upper bound M).
63      //
64      //   Build a histogram of unit bin size from the input sequence.
65      //
66      //   Scan the histogram up to half the number of bins, applying the formula:
67      //     n = (n-k) + k
68      //
69      //   If the above equation holds for any element encountered, then two terms
70      //     have been found that add to the given sum.
```

```
71    //
72    //   There is one additional case to consider, that is, when considering the
73    //      center element of the histogram when the sum is odd.  In this case,
74    //      since is it required that the terms in the sum are at different
75    //      positions, we must check that there are two terms in the sequence,
76    //      in other words, that the histogram count at this position is greater
77    //      than one.
78    //
79    //   Note that in the implementation we are using the C++ behavior that of
80    //      integral type having a truth value of T if and only if their register
81    //      value is non-zero.
82    //
83    //
84    // Performance:
85    //
86    //   For a sequence xs, having N elements, we have:
87    //
88    //      average time complexity:       O(N)
89    //      worst case time complexity:    N + 1/2 M
90    //
91    //      constant space complexity:     M
92    //
93    //
94    //   Note that time complexity assumptions for the average case are not strictly
95    //      valid without knowledge of the underlying statistical distribution of the
96    //      input data.
97    //
98    //
99    //   Note that for large M, the space complexity of this algorithm may be
100   //      substantial.  Algorithm 2 provides better performance when M is large.
101   //
102   //
103   namespace demo::algo1 {
104     template<class T, std::size_t M>
105     class SequenceCheck
106     {
107      public:
108       inline bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
109       {
110         // build a histogram
111         hist_.fill(0);
112         for(auto x : xs) if(x<=sum) hist_[x]++;
113         // each element up to half of the histogram
114         for(auto k=0; k<std::floor(sum/2.0); ++k)
115         {
```

```
116              // test if n = (n-k) + k
117              if( hist_[sum-k] && hist_[k] ) return true;
118          } // k = 0...sum/2
119          // special case at ceil(N/2) when N odd, must have at least two terms
120          if( !(sum%2) && hist_[std::ceil(sum/2.0)]>1 ) return true;
121          return false; // otherwise no such two terms
122        } // has_two_terms
123      private:
124        std::array<T,M> hist_{};
125    }; // SequenceCheck
126  } // demo::algo1
127
128
129  // ============================================================================
130  // has_two_sum_terms (Algorithm 2)
131  // ============================================================================
132  //
133  //   returns true if there exists elements m and n in sequence xs such that
134  //                  sum = m + n, where xs and sum are given
135  //
136  //   inputs:
137  //
138  //     xs : vector<T>   - a sequence of terms to consider
139  //
140  //     sum : T          - the specified target sum for testing terms
141  //
142  //   template paramters:
143  //
144  //     T : class        - the data type of the terms and target sum
145  //
146  //   returns:
147  //
148  //     has_terms : bool - true  if the input sequence contains two elements
149  //                              equal to a given sum
150  //                        false otherwise
151  //
152  // ----------------------------------------------------------------------------
153  //
154  //
155  // Background:
156  //
157  //   Algorithm 2 makes use of the algebraic group property that every number
158  //      has an inverse, and thus we may rewrite:
159  //
160  //      s = m + n   as   n = s - m
```

```
161    //
162    //    Thus for each element m encountered in xs, we know uniquely of a
163    //       corresponding n in xs that we seek.
164    //
165    //    Therefor, we may scan xs once to identify its compliment with respect
166    //       to s.
167    //
168    //    Now, with a set of compliments, say xs', we may scan xs again to
169    //       determine if any element exists in xs'.
170    //
171    //    If we find that an element in xs is found in the set of compliments,
172    //       then we know the sum can be produced from two terms that exist in
173    //       the sequence.
174    //
175    //    Since it is also required that the terms in the sum are at distictly
176    //       different positions in the sequence, use an unordered map for the
177    //       set of compliments, and use the position to track the position of
178    //       the original term.  When checking the compliment, verify that
179    //       the position is also distinct from the original term.
180    //
181    // Implementation:
182    //
183    //    Initially an empty, unordered map (uses a hash map implementation).
184    //       Call this the compliment map cs.
185    //
186    //    Scan the input sequence, xs, for each x in xs.  For each x, compute
187    //       the compliment of the sum, s, and x, say:  c = s - x, and insert
188    //       the compliment c and the ordinal position of x (say k) into cs.
189    //
190    //    Scan the input sequence, xs, for each x in xs again, checking the
191    //       compliment map cs for x.  If x exists in cs, and the position is
192    //       distinct from the map's value of k, then we have found two terms
193    //       that produce the sum.
194    //
195    //    If the end of the sequence is reached without finding a matching x in
196    //       the compliment map, cs, then there are no two terms in xs that will
197    //       produce the sum.
198    //
199    //
200    // Performance:
201    //
202    //    For a sequence xs, having N elements, we have:
203    //
204    //       average time complexity:        O(N)
205    //       best case time complexity:      N + 1
```

```
206    //      worst case time complexity:      2 * N
207    //
208    //      average case space complexity:   O(N)
209    //      best  case space complexity:     1
210    //      worst case space complexity:     N
211    //
212    //
213    //   Note that the space complexity is dependent only on the number of
214    //      unique elements in the input sequence (xs).
215    //
216    //
217    //   Note that for small M, algorithm 1 can have better worst case time complexity
218    //      as well as substantially better space complexity.  These assumptions can
219    //      be further bounded and refined given statistical knowledge of the input data.
220    //
221    //
222    namespace demo::algo2 {
223      template<class T>
224      bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
225      {
226        // hash map of compliments: CS := { c : sum-x=c forall x in xs }
227        std::unordered_map<T,std::size_t> compliments;
228        for(auto k=0; k<xs.size(); ++k)
229        {
230          const T x = xs[k];
231          const auto diff = sum - x; // caution: T must be a signed datatype
232          compliments[diff] = k;
233        } // foreach index k of x in xs
234        // scan the input sequence again to identify if any compliments are present
235        for(auto k=0; k<xs.size(); ++k)
236        {
237          const T x = xs[k];
238          const auto it = compliments.find(x);
239          // check that the compliment is at a distinct position from the original term
240          if( (it!= compliments.end()) && (k != it->second) ) return true;
241        } // foreach x in xs
242        return false; // otherwise no such two terms
243      } // has_two_terms
244    } // demo::algo2
245
246
247    //
248    // main test driver
249    //
250    int main(int argc, char *argv[])
```

```
251    {
252
253      // default element type (domain)
254      typedef int64_t element_t;
255
256      /*
257       *
258       *  Basic Fobonacci tests
259       *
260       */
261
262      // An array containing the first 20 terms of the Fibonacci sequence
263      //
264      //   F[n] := F[n-1] + F[n-2], with F[1]:=1 and F[2]:=2 for n = 0..20
265      //
266      //   Note that this subsequence (xs) contains the following pairs:
267      //
268      //      ( 3, 5) in xs,  and  8 =  3 +  5
269      //
270      //      (13,21) in xs,  and 34 = 13 + 21
271      //
272      //
273      //   And also that this subsequence (xs) does not contain pairs satisfying the following
274      //
275      //      19 = m + n for any m and n in xs
276      //
277      //      41 = m + n for any m and n in xs
278      //
279      //
280      std::vector<element_t> xs = {
281            1,    2,    3,    5,      8,
282           13,   21,   34,   55,     89,
283          144,  233,  377,  610,    987,
284         1597, 2584, 4181, 6765, 10946
285      }; // fibonnaci sequence xs
286
287      // unit test for both algorithms
288      auto my_assert = [](const std::vector<element_t> &xs, element_t x, bool expect) -> bool
289        // assumptions about the maximum expected target sum
290        constexpr std::size_t M = 1024;
291        // test algorithm 1
292        demo::algo1::SequenceCheck<element_t, M> check;
293        auto result_1 = check.has_two_sum_terms(xs, x);
294        assert(result_1 == expect);
295        // test algorithm 2
```

```
296        auto result_2 = demo::algo2::has_two_sum_terms<element_t>(xs, x);
297        assert(result_2 == expect);
298        std::cout << "test case for sum "
299                  << std::setw(3) << x
300                  << " passed"
301                  << std::endl << std::flush;
302     }; // my_assert
303
304     // list of test cases with expected results
305     std::map<element_t, bool> test_cases = {
306       { 8, true},
307       {34, true},
308       {19, false},
309       {41, false}
310     }; // test_cases
311
312     // run all tests
313     for(auto &test : test_cases) my_assert(xs, test.first, test.second);
314
315     /*
316      *
317      *  Some additional stress tests to handle special cases
318      *
319      */
320
321     std::vector<element_t> xs_2 = {
322         1,   1,
323         1,   1,
324         4,   4,
325         9,
326        12
327     }; // stress sequence x2_s
328
329     // list of test cases with expected results
330     std::map<element_t, bool> test_cases_2 = {
331       { 2, true},
332       { 8, true},
333       { 9, false},
334       {12, false}
335     }; // test_cases_2
336
337     // run all tests
338     for(auto &test : test_cases_2) my_assert(xs_2, test.first, test.second);
339
340     return EXIT_SUCCESS;
```

```
341      } // main
342
343   // *EOF
```

Test Data Generation

```
1    #!/usr/bin/env stack
2    -- Fibonacci.hs
3    -- Mac Radigan
4
5    fib :: Integer -> Integer
6    fib 0 = 1
7    fib 1 = 1
8    fib n = fib (n-1) + fib (n-2)
9
10   main :: IO ()
11   main = print $ map fib [1..20]
12
13   -- *EOF*
```

Unit Test Results

```
1    test case for sum    8 passed
2    test case for sum   19 passed
3    test case for sum   34 passed
4    test case for sum   41 passed
5    test case for sum    2 passed
6    test case for sum    8 passed
7    test case for sum    9 passed
8    test case for sum   12 passed
```