

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.

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.

```

1 // =====
2 // has_two_sum_terms (Algorithm 1)
3 // =====
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         hist_.fill(0);
34         // build a histogram
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 N/2 when N even, must have at least two terms
43         if( !(sum%2) && hist_[std::floor(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 \mathbb{X} once to identify its compliment with respect to Σ .

Now, with a set of compliments, say $\bar{\mathbb{X}} = \Sigma - \mathbb{X}$, 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.

Implementation

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

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: $\bar{x} = \Sigma - x$, and insert the compliment \bar{x} into $\bar{\mathbb{X}}$.

Scan the input sequence, \mathbb{X} , for each x in \mathbb{X} again, checking the compliment set $\bar{\mathbb{X}}$ for x . If x exists in $\bar{\mathbb{X}}$, 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 set, $\bar{\mathbb{X}}$, then there are no two terms in \mathbb{X} that will produce the sum.

Performance

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

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.

```

1 // =====
2 // has_two_sum_terms (Algorithm 2)
3 // =====
4 //
5 // IMPORTANT: Note that Algorithm 2 currently does not enforce the selection

```

```

6  //          of two terms in the sequence to be from two distinct different
7  //          positions.
8  //
9  //  returns true if there exists elements m and n in sequence xs such that
10 //          sum = m + n, where xs and sum are given
11 //
12 //  inputs:
13 //
14 //      xs : vector<T>    - a sequence of terms to consider
15 //
16 //      sum : T           - the specified target sum for testing terms
17 //
18 //  template paramters:
19 //
20 //      T : class         - the data type of the terms and target sum
21 //
22 //  returns:
23 //
24 //      has_terms : bool - true  if the input sequence contains two elements
25 //                          equal to a given sum
26 //                          false otherwise
27 //
28 // -----
29 template<class T>
30 bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
31 {
32     // set of compliments: CS := { c : sum-x=c forall x in xs }
33     std::unordered_set<T> compliments;
34     for(const auto &x : xs)
35     {
36         const auto diff = sum - x; // caution: T must be a signed datatype
37         compliments.insert(diff);
38     } // foreach x in xs
39     // scan the input sequence again to identify if any compliments are present
40     for(const auto &x : xs)
41     {
42         if(compliments.find(x) != compliments.end()) return true;
43     } // foreach x in xs
44     return false;
45 } // 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_set>
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 euqation holds for any element encountered, then two terms
70 // have been found that add to the given sum.
71 //
72 // Note that in the implementation we are using the C++ behavior that of
73 // integral type having a truth value of  $T$  if and only if their register
74 // value is non-zero.
75 //
76 //
77 // Performance:
78 //
79 // For a sequence  $xs$ , having  $N$  elements, we have:
80 //
81 // average time complexity:  $O(N)$ 
82 // worst case time complexity:  $N + 1/2 M$ 
83 //
84 // constant space complexity:  $M$ 
85 //
86 //
87 // Note that time complexity assumptions for the average case are not strictly
88 // valid without knowledge of the underlying statistical distribution of the
```

```
89  //      input data.
90  //
91  //
92  //      Note that for large M, the space complexity of this algorithm may be
93  //      substantial. Algorithm 2 provides better performance when M is large.
94  //
95  //
96  namespace demo::algo1 {
97      template<class T, std::size_t M>
98      class SequenceCheck
99      {
100      public:
101          inline bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
102          {
103              hist_.fill(0);
104              // build a histogram
105              for(auto x : xs) if(x<=sum) hist_[x]++;
106              // each element up to half of the histogram
107              for(auto k=0; k<std::floor(sum/2.0); ++k)
108              {
109                  // test if  $n = (n-k) + k$ 
110                  if( hist_[sum-k] && hist_[k] ) return true;
111              } // k = 0...sum/2
112              // special case at N/2 when N even, must have at least two terms
113              if( !(sum%2) && hist_[std::floor(sum/2.0)]>1 ) return true;
114              return false; // otherwise no such two terms
115          } // has_two_terms
116      private:
117          std::array<T,M> hist_{};
118      }; // SequenceCheck
119  } // demo::algo1
120
121
122  // =====
123  // has_two_sum_terms (Algorithm 2)
124  // =====
125  //
126  // IMPORTANT: Note that Algorithm 2 currently does not enforce the selection
127  //            of two terms in the sequence to be from two distinct different
128  //            positions.
129  //
130  //
131  //      returns true if there exists elements m and n in sequence xs such that
132  //      sum = m + n, where xs and sum are given
133  //
```



```

134 //   inputs:
135 //
136 //   xs : vector<T>   - a sequence of terms to consider
137 //
138 //   sum : T          - the specified target sum for testing terms
139 //
140 //   template paramters:
141 //
142 //   T : class        - the data type of the terms and target sum
143 //
144 //   returns:
145 //
146 //   has_terms : bool - true   if the input sequence contains two elements
147 //                       equal to a given sum
148 //                       false otherwise
149 //
150 // -----
151 //
152 //
153 // Background:
154 //
155 //   Algorithm 2 makes use of the algebraic group property that every number
156 //   has an inverse, and thus we may rewrite:
157 //
158 //    $s = m + n$  as  $n = s - m$ 
159 //
160 //   Thus for each element  $m$  encountered in  $xs$ , we know uniquely of a
161 //   corresponding  $n$  in  $xs$  that we seek.
162 //
163 //   Therefor, we may scan  $xs$  once to identify its compliment with respect
164 //   to  $s$ .
165 //
166 //   Now, with a set of compliments, say  $xs'$ , we may scan  $xs$  again to
167 //   determine if any element exists in  $xs'$ .
168 //
169 //   If we find that an element in  $xs$  is found in the set of compliments,
170 //   then we know the sum can be produced from two terms that exist in
171 //   the sequence.
172 //
173 //
174 // Implementation:
175 //
176 //   Initially an empty, unordered set (uses a hash map implementation).
177 //   Call this the compliment set  $cs$ .
178 //

```

```

179 // Scan the input sequence, xs, for each x in xs. For each x, compute
180 // the compliment of the sum, s, and x, say: c = s - x, and insert
181 // the compliment c into cs.
182 //
183 // Scan the input sequence, xs, for each x in xs again, checking the
184 // compliment set cs for x. If x exists in cs, then we have found
185 // two terms that produce the sum.
186 //
187 // If the end of the sequence is reached without finding a matching x in
188 // the compliment set, cs, then there are no two terms in xs that will
189 // produce the sum.
190 //
191 //
192 // Performance:
193 //
194 // For a sequence xs, having N elements, we have:
195 //
196 // average time complexity: O(N)
197 // best case time complexity: N + 1
198 // worst case time complexity: 2 * N
199 //
200 // average case space complexity: O(N)
201 // best case space complexity: 1
202 // worst case space complexity: N
203 //
204 //
205 // Note that the space complexity is dependent only on the number of
206 // unique elements in the input sequence (xs).
207 //
208 //
209 // Note that for small M, algorithm 1 can have better worst case time complexity
210 // as well as substantially better space complexity. These assumptions can
211 // be further bounded and refined given statistical knowledge of the input data.
212 //
213 //
214 namespace demo::algo2 {
215     template<class T>
216     bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
217     {
218         // set of compliments: CS := { c : sum-x=c forall x in xs }
219         std::unordered_set<T> compliments;
220         for(const auto &x : xs)
221         {
222             const auto diff = sum - x; // caution: T must be a signed datatype
223             compliments.insert(diff);

```

```

224     } // foreach x in xs
225     // scan the input sequence again to identify if any compliments are present
226     for(const auto &x : xs)
227     {
228         if(compliments.find(x) != compliments.end()) return true;
229     } // foreach x in xs
230     return false; // otherwise no such two terms
231 } // has_two_terms
232 } // demo::algo2
233
234
235 //
236 // main test driver
237 //
238 int main(int argc, char *argv[])
239 {
240
241     // default element type (domain)
242     typedef int64_t element_t;
243
244     /*
245     *
246     * Basic Fobonacci tests
247     *
248     */
249
250     // An array containing the first 20 terms of the Fibonacci sequence
251     //
252     //  $F[n] := F[n-1] + F[n-2]$ , with  $F[1] := 1$  and  $F[2] := 2$  for  $n = 0..20$ 
253     //
254     // Note that this subsequence (xs) contains the following pairs:
255     //
256     // ( 3, 5) in xs, and  $8 = 3 + 5$ 
257     //
258     // (13,21) in xs, and  $34 = 13 + 21$ 
259     //
260     //
261     // And also that this subsequence (xs) does not contain pairs satisfying the following
262     //
263     //  $19 = m + n$  for any  $m$  and  $n$  in xs
264     //
265     //  $41 = m + n$  for any  $m$  and  $n$  in xs
266     //
267     //
268     std::vector<element_t> xs = {

```

```
269         1,   2,   3,   5,   8,
270         13,  21,  34,  55,  89,
271         144, 233, 377, 610, 987,
272         1597, 2584, 4181, 6765, 10946
273     }; // fibonacci sequence xs
274
275     // unit test for both algorithms
276     auto my_assert = [&](element_t x, bool expect) -> bool {
277         // assumptions about the maximum expected target sum
278         constexpr std::size_t M = 1024;
279         // test algorithm 1
280         demo::algo1::SequenceCheck<element_t, M> check;
281         auto result_1 = check.has_two_sum_terms(xs, x);
282         assert(result_1 == expect);
283         // test algorithm 2
284         auto result_2 = demo::algo2::has_two_sum_terms<element_t>(xs, x);
285         assert(result_2 == expect);
286         std::cout << "test case for sum "
287                     << std::setw(3) << x
288                     << " passed"
289                     << std::endl << std::flush;
290     }; // my_assert
291
292     // list of test cases with expected results
293     std::map<element_t, bool> test_cases = {
294         { 8, true},
295         {34, true},
296         {19, false},
297         {41, false}
298     }; // test_cases
299
300     // run all tests
301     for(auto &test : test_cases) my_assert(test.first, test.second);
302
303     /*
304     *
305     * Some additional stress tests to handle special cases
306     *
307     */
308
309     std::vector<element_t> xs_2 = {
310         1,  1,
311         1,  1,
312         4,  4,
313         9,
```

```
314         12
315     }; // stress sequence x2_s
316
317     // list of test cases with expected results
318     std::map<element_t, bool> test_cases_2 = {
319         { 2, true},
320         { 8, true},
321         { 9, false},
322         {12, false}
323     }; // test_cases_2
324
325     // run all tests
326     for(auto &test : test_cases_2) my_assert(test.first, test.second);
327
328     return EXIT_SUCCESS;
329 } // main
330
331 // *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
```