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.

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 $\Sigma == h_{\Sigma - k} + h_k$ **then**
        return $\top$ ⍝ check sum of terms holds
    end if
end for
if $2 \mid \Sigma \wedge h_{\lceil \frac{k}{2} \rceil} \geq 2$ **then**
    return $\top$ ⍝ check two terms in histogram center
end if
return $\perp$

Performance

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

Measure	Performance
average time complexity	O(N)
worst case time complexity	O(N + 1/2 M)
constant space complexity	O(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)
// ============================================================================
//
//   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
//
//     M : size_t       - an upper bound on the expected sum
//
//   returns:
//
//     has_terms : bool - true  if the input sequence contains two elements
//                              equal to a given sum
//                        false otherwise
//
// ---------------------------------------------------------------------------
template<class T, std::size_t M>
class SequenceCheck
{
 public:
  inline bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
  {
    // build a histogram
    hist_.fill(0);
    for(auto x : xs) if(x<=sum) hist_[x]++;
    // each element up to half of the histogram
    for(auto k=0; k<std::floor(sum/2.0); ++k)
    {
      // test if n = (n-k) + k
      if( hist_[sum-k] && hist_[k] ) return true;
    } // k = 0...sum/2
    // special case at ceil(N/2) when N odd, must have at least two terms
    if( !(sum%2) && hist_[std::ceil(sum/2.0)]>1 ) return true;
    return false; // otherwise no such two terms
  } // has_two_terms
 private:
  std::array<T,M> hist_{};
}; // SequenceCheck
```

Algorithm #2

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:

Measure	Performance
average time complexity	O(N)

Measure	Performance
best case time complexity	O(1)
worst case time complexity	O(N)
average case space complexity	O(N)
best case space complexity	O(1)
worst case space complexity	O(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.

```
// ============================================================================
// 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)
{
  // hash map of xs_bar: CS := { c : sum-x=c forall x in xs }
  std::unordered_map<T,std::size_t> xs_bar;
  for(auto k=0; k<xs.size(); ++k)
  {
```

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
  if $\mathbb{X}_k \in \overline{\mathbb{X}} \wedge \mathbb{F}_k \neq k$ **then**
    return $\top$ ⍝ check are compliments present and distinct
  end if
end for
return $\perp$

```
    const T x = xs[k];
    const auto diff = sum - x; // caution: T must be a signed datatype
    xs_bar[diff] = k;
    // scan the input sequence again to identify if any compliments are present
    const auto x_bar = xs_bar.find(x);
    // check that the compliment is at a distinct position from the original term
    if( (x_bar!= xs_bar.end()) && (k != x_bar->second) ) return true;
  } // foreach index k of x in xs
  return false; // otherwise no such two terms
} // has_two_sum_terms
```

Source Code

```
// sum-two-terms.cc
// Mac Radigan


  #include <array>
  #include <assert.h>
  #include <cmath>
  #include <cstdlib>
  #include <iomanip>
  #include <iostream>
  #include <map>
  #include <sys/types.h>
  #include <unordered_map>
  #include <vector>


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

```
// ==========================================================================
//
//   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
//
//     M : size_t       - an upper bound on the expected sum
//
//   returns:
//
//     has_terms : bool - true  if the input sequence contains two elements
//                              equal to a given sum
//                        false otherwise
//
// --------------------------------------------------------------------------
//
//
// 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 xs
//     between 0 and sum, 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.
//
//   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.
//
//   There is one additional case to consider, that is, when considering the
//     center element of the histogram when the sum is odd.  In this case,
//     since is it required that the terms in the sum are at different
//     positions, we must check that there are two terms in the sequence,
//     in other words, that the histogram count at this position is greater
//     than one.
//
//   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:
//
//     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.
//
//
namespace demo::algo1 {
  template<class T, std::size_t M>
  class SequenceCheck
  {
   public:
    inline bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
```

```
    {
      // build a histogram
      hist_.fill(0);
      for(auto x : xs) if(x<=sum) hist_[x]++;
      // each element up to half of the histogram
      for(auto k=0; k<std::floor(sum/2.0); ++k)
      {
        // test if n = (n-k) + k
        if( hist_[sum-k] && hist_[k] ) return true;
      } // k = 0...sum/2
      // special case at ceil(N/2) when N odd, must have at least two terms
      if( !(sum%2) && hist_[std::ceil(sum/2.0)]>1 ) return true;
      return false; // otherwise no such two terms
    } // has_two_terms
   private:
    std::array<T,M> hist_{};
  }; // SequenceCheck
} // demo::algo1


// ==========================================================================
// 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
//
// --------------------------------------------------------------------------
//
```

```
//
// Background:
//
//   Algorithm 2 makes use of the algebraic group property that every number
//     has an inverse, and thus we may rewrite:
//
//     s = m + n  as  n = s - m
//
//   Thus for each element m encountered in xs, we know uniquely of a
//     corresponding n in xs that we seek.
//
//   Therefor, we may scan xs once to identify its compliment with respect
//     to s.
//
//   Now, with a set of compliments, say xs', we may scan xs again to
//     determine if any element exists in xs'.
//
//   If we find that an element in xs 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 cs.
//
//   Scan the input sequence, xs, for each x in xs.  For each x, compute
//     the compliment of the sum, s, and x, say:  c = s - x, and insert
//     the compliment c and the ordinal position of x (say k) into cs.
//
//   Scan the input sequence, xs, for each x in xs again, checking the
//     compliment map cs for x.  If x exists in cs, 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, cs, then there are no two terms in xs that will
//     produce the sum.
//
```

```
//
// Performance:
//
//   For a sequence xs, having N elements, we have:
//
//     average time complexity:         O(N)
//     best case time complexity:       O(1)
//     worst case time complexity:      O(N)
//
//     average case space complexity:   O(N)
//     best  case space complexity:     O(1)
//     worst case space complexity:     O(N)
//
//
//   Note that the space complexity is dependent only on the number of
//     unique elements in the input sequence (xs).
//
//
//   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.
//
//
namespace demo::algo2 {
  template<class T>
  bool has_two_sum_terms(const std::vector<T> &xs, const T sum)
  {
    // hash map of compliments: CS := { c : sum-x=c forall x in xs }
    std::unordered_map<T,std::size_t> compliments;
    for(auto k=0; k<xs.size(); ++k)
    {
      const T x = xs[k];
      const auto diff = sum - x; // caution: T must be a signed datatype
      compliments[diff] = k;
      // scan the input sequence again to identify if any compliments are present
      const auto x_bar = compliments.find(x);
      // check that the compliment is at a distinct position from the original term
      if( (x_bar!= compliments.end()) && (k != x_bar->second) ) return true;
    } // foreach index k of x in xs
    return false; // otherwise no such two terms
  } // has_two_sum_terms
} // demo::algo2


//
```

```
// main test driver
//
int main(int argc, char *argv[])
{

  // default element type (domain)
  typedef int64_t element_t;

  /*
   *
   *  Basic Fobonacci tests
   *
   */

  // An array containing the first 20 terms of the Fibonacci sequence
  //
  //   F[n] := F[n-1] + F[n-2], with F[1]:=1 and F[2]:=2 for n = 0..20
  //
  //   Note that this subsequence (xs) contains the following pairs:
  //
  //     ( 3, 5) in xs,  and  8 =  3 +  5
  //
  //     (13,21) in xs,  and 34 = 13 + 21
  //
  //
  //   And also that this subsequence (xs) does not contain pairs satisfying the followin
  //
  //     19 = m + n for any m and n in xs
  //
  //     41 = m + n for any m and n in xs
  //
  //
  std::vector<element_t> xs = {
       1,    2,    3,    5,     8,
      13,   21,   34,   55,    89,
     144,  233,  377,  610,   987,
    1597, 2584, 4181, 6765, 10946
  }; // fibonnaci sequence xs

  // unit test for both algorithms
  auto my_assert = [](const std::vector<element_t> &xs, element_t x, bool expect) -> bool
    // assumptions about the maximum expected target sum
    constexpr std::size_t M = 1024;
    // test algorithm 1
    demo::algo1::SequenceCheck<element_t, M> check;
```

```
      auto result_1 = check.has_two_sum_terms(xs, x);
      assert(result_1 == expect);
      // test algorithm 2
      auto result_2 = demo::algo2::has_two_sum_terms<element_t>(xs, x);
      assert(result_2 == expect);
      std::cout << "test case for sum "
                << std::setw(3) << x
                << " passed"
                << std::endl << std::flush;
    }; // my_assert

    // list of test cases with expected results
    std::map<element_t, bool> test_cases = {
      { 8, true},
      {34, true},
      {19, false},
      {41, false}
    }; // test_cases

    // run all tests
    for(auto &test : test_cases) my_assert(xs, test.first, test.second);

    /*
     *
     *  Some additional stress tests to handle special cases
     *
     */

    std::vector<element_t> xs_2 = {
        1,  1,
        1,  1,
        4,  4,
        9,
       12
    }; // stress sequence x2_s

    // list of test cases with expected results
    std::map<element_t, bool> test_cases_2 = {
      { 2, true},
      { 8, true},
      { 9, false},
      {12, false}
    }; // test_cases_2

    // run all tests
```

```
    for(auto &test : test_cases_2) my_assert(xs_2, test.first, test.second);

    return EXIT_SUCCESS;
  } // main

// *EOF*
```

Test Data Generation

```
#!/usr/bin/env stack
-- Fibonacci.hs
-- Mac Radigan

fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main :: IO ()
main = print $ map fib [1..20]

-- *EOF*
```

Unit Test Results

```
test case for sum   8 passed
test case for sum  19 passed
test case for sum  34 passed
test case for sum  41 passed
```