Computing Quantiles

Larry Bush

October 17, 2002

Abstract

This paper presents a generic function to compute the m-th quantiles of a multiset of numbers. It is implemented as a C++ template function. A test program is also presented. This presentation is an adaptation and improvement of the program presented in [1] and [2].

Contents

1 Introduction

In the referenced paper [2] a program is presented for calculating the quantiles of a seqence.

The program used a function which calculates the interior m-tile of any quantity. An m-tile, represents the equal division of a sequence into parts. This is much like a quartile, but m-tile of a sequence may be calculated for any scalar value. A precise definition of an m-tile along with the assumptions used in [2] is given in the following section.

The quantile function presented in this paper improves upon that algorithm in qenericity and efficiency. These improvements are discussed in 3.2 below.

The Main program presented in this paper uses the new function to compute quartiles as well as user defined m-tiles. It also tests the function using several new tests. These changes are explained in 3.1.

Instructions for compiling and running this program are included in 4.

2 Definition of m-tiles

The definition of an *m-tile* used in this paper is the same as that used in [2, p. 2] which was adapted from [3, p. 361].

> "The following definition is adapted from [3, p. 361].
>
> For a given multiset of values S with N elements (counting repetitions), V is called the *r-th m-tile* (or *r-th m-tile mark*) ($r = 0, 1, \ldots, m$) if simultaneously,
>
> $$|\{x \in S : x < V\}|/N \leq r/m,$$
>
> and
>
> $$|\{x \in S : V < x\}|/N \leq (m - r)/m.$$
>
> In particular, the zeroth m-tile is the minimum of the values and the m-th m-tile is the maximum.[1]
>
> The case $m = 2$ and $r = 1$ defines the median value. In this case the definition says the median is any value that satisfies both
>
> $$|\{x \in S : x < V\}|/N \leq 1/2,$$
>
> and
>
> $$|\{x \in S : V < x\}|/N \leq 1/2.$$
>
> Suppose now the values of the multiset S are written as a sequence in ascending order, $x_0 \leq x_1 \leq \ldots \leq x_{N-1}$. In the case where N is odd, the above inequalities determine a unique median value, the value

[1]The definition given in [3] is more general in that it allows for weights to be associated with the values. The definition given here is for the special (and most common) case in which the weights are all 1.

$x_{(N-1)/2}$ with central index $(N-1)/2 = \lfloor N/2 \rfloor$. In the even case, there is a central pair of values $x_{N/2-1}$ and $x_{N/2}$, and our definition allows the median to be chosen as any value $V \in [x_{N/2-1}, x_{N/2}]$. Many authors define the median in this case to be the midpoint of the range $[x_{N/2-1}, x_{N/2}]$, i.e., the arithmetic mean $(x_{N/2-1} + x_{N/2})/2$. But the above definition allows it to be chosen more simply. Our m-tiles function computes it as $x_{N/2}$, the maximum of the possible values. Note that when N is even, $N/2 = \lfloor N/2 \rfloor$, so that regardless of the parity of N, we can compute the median as $x_{\lfloor N/2 \rfloor}$ (provided, again, that S is in ascending order, but the basic definition of m-tiles doesn't require ascending order, and we shall see that in actually computing the quantiles we do not need to fully sort the sequence).

Let us now derive the possible choices for an index k such that x_k is an r-th m-tile. Suppose the values of the multiset S are written as a sequence $x_0, x_1, \ldots, x_{N-1}$, and r and m are integers such that $0 \leq r \leq m \leq N$. Assume also that for some index k,

1. $x_p \leq x_k$ for all $p \in [0, k)$,
2. $x_k \leq x_q$ for all $q \in [k+1, N)$.

Then from 1,

$$|\{x \in S : x < x_k\}| \leq k,$$

and from 2,

$$|\{x \in S : x_k < x\}| \leq N - k - 1.$$

Thus for x_k to be an r-th m-tile, it suffices that $k/N \leq r/m$ and $(N-k-1)/N \leq (m-r)/m$. The latter inequality is equivalent to

$$\begin{aligned} 1-(k+1)/N &\leq 1-r/m, \quad \text{or} \\ r/m &\leq (k+1)/N, \quad \text{or} \\ rN/m &\leq k+1 \end{aligned}$$

Thus

$$k \leq rN/m \leq k+1. \tag{1}$$

Now there are two cases, depending on whether or not rN/m is an integer.

1. If rN/m is not an integer, we must have

   $$k < rN/m < k+1,$$

   and $k = \lfloor rN/m \rfloor$.

2. If rN/m is an integer, it appears from (1) that we can take $k = rN/m$ or $k = rN/m - 1$. Either choice is possible except in the boundary cases: when $r = 0$, we can only choose $k = 0$; and when $r = m$, we can only choose $k = N - 1$.

Thus a simple and consistent way of choosing k is always as $\lfloor rN/m \rfloor$, unless $r = N$, in which case we must choose $k = N - 1$. Finally, note that when computing interior quantiles only, i.e., $0 < r < m$, we can always choose $k = \lfloor rN/m \rfloor$.

For handy reference, we restate the main conclusion we will use in the showing correctness of the function in Section 6:

Theorem 1 *Let the values of the multiset S be written as a sequence $x_0, x_1, \ldots, x_{N-1}$, let r and m be integers such that $0 \leq r < m \leq N$, and let $k = \lfloor rN/m \rfloor$. If we furthermore have*

$$x_p \leq x_k \leq x_q \quad \textit{for all } p \in [0, k) \textit{ and } q \in [k+1, N)$$

then x_k is an r-th m-tile of S.

As a simple application of this theorem, consider the case $m = 4$, where the m-tiles are called quartiles. Thus the first quartile, Q_1, is defined as any value V that satisfies

$$|\{x \in S : x < V\}|/N \leq 1/4,$$

and

$$|\{x \in S : V < x\}|/N \leq 3/4.$$

If the values of S are $x_0, x_1, \ldots, x_{N-1}$ and are ordered such that $x_p \leq x_k \leq x_q$ for all $p \in [0, k)$ and $q \in [k+1, N)$ where $k = \lfloor N/4 \rfloor$, then by the theorem we can choose $Q_1 = x_k = x_{\lfloor N/4 \rfloor}$." [2]

3 Changes

I have changed the previous version [1] of the quantiles algorithm and associated program in several ways.

3.1 Changes to the Main Program

First of all, the program that I created to call the quantiles function and run tests on it, also allows user input. In creating the user interface, I assume that the user will redirect a file to standard input. This file will contain a list of numbers whitespace delimited.

This file will also contain 2 other pieces of information. The first number in the file will indicate the m-tile that the user wants computed. For example, if the first number in the file is a 4, the program will compute the quartiles on the user's input. The second number in the file will indicate if the user wants the program to output the number sequence (0 = no, 1 = yes). If the sequence is large, then this is a time consuming process. Plus, it is not helpful in determining the correctness of the algorithm unless the sequence is small. The remaining numbers are the user's input sequence.

My version also computes many more test sequences. My program performs a series of "hard wired" quartiles tests as well as tests that vary depending on the number of quantiles the user chooses to compute. These tests are discussed below.

3.2 Changes to the Quantiles Function

My version also has a more efficient version of the quantiles function. This efficiency is particularly notable when the number of quantiles is variable (and large). The correctness, design and efficiency of this algorithm is discussed below.

My quantiles algorithm also outputs to a back insertion iterator, and outputs iterators that point to the given quantile rather than outputting the quantile itself. This has the added benefit that the iterators can be used to manipulate the quantile, iterate forward or backward in the container, compute the distance between the quantiles, or sort/search between two quantiles.

Note: I changed the quantiles calculation of so that it reflects the upper bound rather than the lower bound as shown in [2].

4 Compile and Run Instructions

4.1 Compile Instructions

I have included a Makefile that generates a pdf file, cpp file and an executable file from this w file. It also compiles the cpp file (using GNU C++ version 3.2 on CS Solaris Machines) and runs the executable on the file "input.txt". The input.txt file is not included because it is large. However, I have created an input file that is set up to compute 7-tiles on 9,998 distict integers (3 - 10,000). This combination is used because it produces asymmetric rounding issues which would cause problems with a simpler divide and counquer algorithm. The file tells the program not to output the sequence.

The file is located at: http://www.cs.rpi.edu/~bushl2/files/input.txt

I also created an input file that is set up to compute 177-tiles on 10,000 distict integers (1 - 10,000). This file also tells the program not to output the sequence.

That file is located at: http://www.cs.rpi.edu/~bushl2/files/input2.txt

The program passes all tests on these seqences.

4.2 Run Instructions

The program can be run by typing "./larry-quantiles". The program expects as input, the m-tile number, an indicator of whether or not you want the seqence to be output (1 for yes, 0 for no), and the sequence of numbers.

For example the input : 4 1 1 2 3 4 5 8 7 6 would generate the quartiles of the sequence 1 - 8. It would also output the sequence. This first number “4” indicates quartiles and the second number “1” tells the program to output the sequence.

Most likely, you will prefer to redirect a file to the program.

The command: ./larry-quantile < input.txt

will do this.

5 A program that computes quantiles

Here is a program that demonstrates the features of the `quantiles` functon. The first order of business is to typedef and instantiate some containers. “D” is the main container. It will store the sequence. “D0” is a backup container. It will allow us to restore the sequence “D” to its original form, so that we can run some tests. “Dnew” is used when we need 2 containers for our test. “D_user” is a container reserved to perform the final acceptance test.

"larry-quantiles.cpp" 7a ≡

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <cassert>
using namespace std;

⟨Define quantiles function 9⟩
⟨Define quantiles helper function 10a⟩
⟨Define acceptance test function 17a⟩

int main()
{
  typedef vector<double> container;
  container D, D0, Dnew, D_user;
  ⟨Read sequence of numbers from standard input into D 7b⟩
  ⟨Save D in D0 for later checking 13b⟩
  ⟨Output the sequence of numbers 8⟩
  ⟨Compute quartiles (4-th quantiles) 12a⟩
  ⟨Output all 5 quartiles, including the min and max 12c⟩
  ⟨Hard wired tests 15⟩
  ⟨Compute m tiles 12b⟩
  ⟨Output all user specified quantiles, including the min and max 13a⟩
  ⟨User input specific tests 16⟩
  ⟨Perform quantiles acceptance test 18b⟩
  return 0;
}
```

Read in the numbers:

⟨Read sequence of numbers from standard input into D 7b⟩ ≡

```
typedef istream_iterator<int> int_input;

int_input ii(cin);
int user_quantile = *ii;
++ii;
int output_the_sequence = *ii;

typedef istream_iterator<double> double_input;

for (double_input j(cin); j != double_input(); ++j)
    D.push_back(*j);
```

Used in part 7a.

The following code outputs the sequence of numbers. Note that this feature can be turned off if the 2nd number in the input fule is 0.

⟨Output the sequence of numbers 8⟩ ≡

```
if(output_the_sequence != 0) {
cout << "D[0], ..., D[" << D.size() - 1 << "]:" << endl;
copy(D.begin(), D.end(), ostream_iterator<double>(cout, " "));
cout << "\n" << endl;
}
```

Used in part 7a.

6 The quantiles function

My version of the quantiles function is presented here. This version works as follows.

For a given m-tile, m quantiles must be computed. The quantiles function uses the C++ standard library **nth_element** function to compute the r-th m-tile.

The **nth_element** function performs in linear time. Therefore, computing **nth_element** m times would be $O(n * m)$ time. However, this function performs in $O(n \log m)$ time. It does so by only limiting the application of the **nth_element** function to the sub-sequences between the previously computed quantiles in the subsequent iterations of the algorithm. Therefore, at each recursive level of the algorithm, **nth_element** operates on n elements.

Since there are n operations performed at each level of the algorithm, and the algorithm has $\log_2 m$ levels this function performs in $O(n \log_2 m)$ time.

The divide and counquer strategy exploits the fact that the nth-element function partially orders the range of elements that it operates on.

The primary operation of Nth_element is to arrange the range $[first, last)$ such that the element pointed to by the iterator nth is the same as the element that would be in that position if the entire range [first, last) had been sorted.

However, as a side effect, none of the elements in the range [nth, last) is less than any of the elements in the range [first, nth). [5]

Using this property, we construct a divide and counquer algorithm that culminates with each quantile being in its respective position in the sequence.

The function outputs a sequence of back insertion iterators via a back-insertion iterator that is passed in as a parameter.

The construction of this divide and counquer algorithm starts with the priming quantiles function. The quantiles function passes some new variables to the quantile helper function, to maintian the integrity of the calculation. This is necessary because in certain situations, successive rounding can cause the sequence to not meet the definition because there will be too many spaces between some of the computed quantiles.

These variables are denoted with the term “absolute” in their name. This means that their value is the same as it was at the beginning of the recursion.

⟨Define quantiles function 9⟩ ≡

```
template <typename RandomAccessIterator, typename OutputIterator>
void
 quantiles(     RandomAccessIterator first,
                        RandomAccessIterator last,
                        OutputIterator result, int m )
{
        typedef typename iterator_traits<RandomAccessIterator>::value_type
          value_type;

        if ( m < 2 ) { return; }
        RandomAccessIterator absolute_first = first;
        RandomAccessIterator absolute_last = last;
        int absolute_m = m;
        int base_m = 0;
        quantiles_helper(       first, last, result, m, absolute_m,
                                                base_m, absolute_first, absolute_last   );
        return;
}
```

Used in part 7a.

The helper function performes most of the work. The helper function computes the median. It then recursively calls itself to compute the median of the lower and upper half of the sequence. An iterator pointing to the median is output to the back insertion iterator between the upper and lower stages. This is so that it is in the proper order.

Note that the median must be computed before the lower steps so that the nth element function re-orders the list, leaving the lower numbers below the median and the higher numbers above the median.

⟨Define quantiles helper function 10a⟩ ≡

```
template <typename RandomAccessIterator, typename OutputIterator>
void
 quantiles_helper(      RandomAccessIterator first,
                RandomAccessIterator last,
                OutputIterator result,
                int m, int absolute_m, int base_m,
                RandomAccessIterator absolute_first,
                RandomAccessIterator absolute_last )
{
  typedef typename iterator_traits<RandomAccessIterator>::value_type
          value_type;
        RandomAccessIterator next;

    int m_new = m/2;
        ⟨Compute m_new-tile 10b⟩

        if ( 1 < m_new ) {
                ⟨Compute lower half 11a⟩
        }
        *result++ = next;
        if ( 1 < (m - m_new) ) {
                ⟨Compute upper half 11b⟩
        }
        return;
}
```

Used in part 7a.

Note: I changed the calculation of so that it reflects the upper bound rather than the lower bound as shown in [2].

⟨Compute m_new-tile 10b⟩ ≡

```
    next = absolute_first + ((absolute_last - absolute_first)*
                        (base_m + m_new))/absolute_m;
    nth_element(first, next, last);
```

Used in part 10a.

This code computes the lower half of the recursion. Note that the next iterator is passed to both. However, it is not “used” by both. For the lower half, it marks one position past the end of the sequence it will consider.

The absolute variables are used to calculate the m tile number. That calculation is done in this way to avoid the possiblility of a deleterious rounding effect due to the recursive division.

⟨Compute lower half 11a⟩ ≡

```
            base_m = base_m;
            quantiles_helper(first, next, result, m_new, absolute_m,
                             base_m, absolute_first, absolute_last);
```

Used in part 10a.

This code computes the upper half of the recursion. Care must be taken to ensure that all m-tiles are computed and computed correctly.

⟨Compute upper half 11b⟩ ≡

```
            base_m = base_m + m_new;
            quantiles_helper(       next, last, result, m - m_new, absolute_m,
                             base_m, absolute_first, absolute_last   );
```

Used in part 10a.

7 Efficiency

This function performs in $O(N \log m)$ time. This is an improvement over the efficency of [2] if m is not constant. The efficiency is gained by employing a divide and counquer stratgy where the `nth_element` function is performed only on the sub-sequences of the lower levels of the algorithm.

The `nth_element` function performs in linear time. In other words, n operations are performed at each level of the algorithm, and the algorithm has $\log m$ levels. The quantiles function exploits the fact that the nth-element function partially orders the range of elements that it operates on.

The primary operation of `nth_element` is to arrange the range $[first, last)$ such that the element pointed to by the iterator nth is the same as the element that would be in that position if the entire range [first, last) had been sorted.

However, as a side effect, none of the elements in the range [nth, last) is less than any of the elements in the range [first, nth). [5]

Since m is halved each iteration, the number of recursions is $\log m$. The number of operations per recursion is $O(N)$.

For example, if we compute the octiles of a sequence of 16 items, this diagram shows how the problem is broken down, so that `nth_element` operates on sub-sequences of the initial sequence. Each operation is $O(n)$ where n is the size of a sub-sequence of the initial sequence of size N. The sum of all the `nth_element` operations at each recursive step is $O(N)$.

Step	Operations	Sub-sequence representation
m = 8	`nth_element`	xxxxxxxxxxxxxxxx
m = 4	`nth_element`	xxxxxxxx xxxxxxxx
m = 2	`nth_element`	xxxx xxxx xxxx xxxx

Therefore, the efficiency is $O(N \log m)$.

8 Computing quantiles

The interior quantiles are computed by calling the quantiles function.

Note that you must instantiate Q with 1 element as a placeholder for the min element. This is because the quantiles function is defined such that it only computes the interior quantiles.

⟨Compute quartiles (4-th quantiles) 12a⟩ ≡

```
int m = 4;
vector<container::iterator> Q(1);
quantiles(D.begin(), D.end(), back_inserter(Q), m);
```

Used in part 7a.

The interior m tiles are computed by calling the quantiles function.

⟨Compute m tiles 12b⟩ ≡

```
D_user = D0;
vector<container::iterator> Q_user(1);
quantiles(D_user.begin(), D_user.end(), back_inserter(Q_user), user_quantile);
```

Used in part 7a.

Finally, we compute the minimum and maximum elements using generic functions from the C++ standard library then output them along with the interior quartiles. We save time computing this function by only taking the minimum of the first section, up to the first quartile marker.

⟨Output all 5 quartiles, including the min and max 12c⟩ ≡

```
Q[0] = min_element(D.begin(), Q[1]);
Q[m] = max_element(Q[m-2],D.end());
cout << "The hard wired quartiles are: " << "\n";
for (int r = 0; r < m+1; ++r)
        cout << "Q[" << r << "] = " << *(Q[r]) << "\n";
cout << endl;
```

Used in part 7a.

You can save time computing this function by only taking the minimum of the first section, up to the first m tile marker.

⟨Output all user specified quantiles, including the min and max 13a⟩ ≡

```
    Q_user[0] = min_element(D_user.begin(), Q_user[1]);
    Q_user[user_quantile] = max_element(Q_user[user_quantile-2],D_user.end());
    cout << "The user specified quantiles are: " << "\n";
    for (int r = 0; r < user_quantile+1; ++r)
      cout << "Q_user[" << r << "] = " << *(Q_user[r]) << "\n";
    cout << endl;
```

Used in part 7a.

An extra variable “D0” is used to save the vector “D” so that it can be put back to its original form in order to run tests on the quantiles function.

⟨Save D in D0 for later checking 13b⟩ ≡

```
  D0 = D;
```

Used in part 7a.

9 The Test Functions

9.1 Hard Wired Tests

The following portion of the program performs consistency checks. This is to ensure that the quantiles function conforms to its definition. For example we expect that $median = \lfloor N/2 \rfloor$. It is also to ensure that the function is consistent with itself when computing different yet analogous quantiles.

For example, suppose we computed 2-tiles (the median of a sequence), quartiles, and octiles. The median should be the same as the second quartile, the second octile should be the same as the first quartile, and the sixth octile should be the same as the third quartile. This should be true regardless of whether the size of the sequence is even or odd.

For our function, we also need to test the position of the iterator as well as the value it points to. To test for the value, we perform the above tests on separate containers and check that the values $x == y$. However, when we test the consistency of the iterator, we perfrom the above tests on the same container and expect that for a given value $\& * x == \& * y$. In other words the iterators are equivalent.

- Test 1 is an iterator test to see if quartile and 2-tile are consistent.
- Test 2 is an iterator test to see if quartile and 8-tile are consistent.
- Test 3 is a value test to see if quartile and 2-tile are consistent.
- Test 4 is a value test to see if quartile and 8-tile are consistent.

- Test 6 determines if the distances between each quantile vary by more than 1. If they do not, then, in that respect, the result complies with the definition of a quantile stated above.

⟨Hard wired tests 15⟩ ≡

```
//Test 1
D = D0;
vector<container::iterator> R(1);
quantiles(D.begin(), D.end(), back_inserter(R), 2);
assert (R[1] == Q[2]);
cout<<"Passed Test 1";

//Test 2
D = D0;
vector<container::iterator> S(1);
quantiles(D.begin(), D.end(), back_inserter(S), 8);
assert (S[2] == Q[1]);
assert (S[4] == Q[2]);
assert (S[6] == Q[3]);
cout<<"Passed Test 2\n";

//Test 3
Dnew = D0;
vector<container::iterator> T(1);
quantiles(Dnew.begin(), Dnew.end(), back_inserter(T), 2);
assert ( *(T[1]) == *(Q[2]) );
cout<<"Passed Test 3\n";

//Test 4
Dnew = D0;
vector<container::iterator> U(1);
quantiles(Dnew.begin(), Dnew.end(), back_inserter(U), 8);
assert (*(U[2]) == *(Q[1]));
assert (*(U[4]) == *(Q[2]));
assert (*(U[6]) == *(Q[3]));
cout<<"Passed Test 4\n";

//Test 5
Dnew = D0;
vector<container::iterator> V(1);
quantiles(Dnew.begin(), Dnew.end(), back_inserter(V), 12);
assert (*(V[3]) == *(Q[1]));
assert (*(V[6]) == *(Q[2]));
assert (*(V[9]) == *(Q[3]));
cout<<"Passed Test 5\n";

//Test 6
int i;
int high = Q[1]-Q[0];
int low =  Q[1]-Q[0];
int size = (Q.end()) - (Q.begin());
for ( i = 2; i < size ; i++ ) {
    high = max( Q[i]-Q[i-1], high );
    low  = min( Q[i]-Q[i-1], low );
}
assert ( (high - low) < 2 );
cout<<"Passed Test 6\n";
cout << "Hard wired consistency checks passed." << endl;
```

Used in part 7a.

9.2 User input dependent Tests

The following portion of the program performs consistency checks on the quantiles function that are dependent on the user's input. This allows additional testing using different quantiles.

These test both the position of the iterator (run on the same containers), as well as the value that the iterator points to (run on separate containers) .

- Test 7 is a value test. It checks up to 10 m-tile points in 10 other quartile computations on the given sequence.
- Test 8 determines if the distances between each quantile in the user specified m-tile computation vary by more than 1. If they do not, then the result complies with the definition of a quantile stated above.

⟨User input specific tests 16⟩ ≡

```
//Test 7
for (int n = 2; n < 12; n++) {

        Dnew = D0;
        vector<container::iterator> Uu(1);
        quantiles(Dnew.begin(), Dnew.end(), back_inserter(Uu), user_quantile*n);

        for (int p = 1; p < 10; p++) {

                if ( p < user_quantile ) {
                        assert (*(Uu[p*n]) == *(Q_user[p]));
                }
        }
        cout<<"Passed user specific Test 7 - "<<n<<"\n";

}
//Test 8
high = Q_user[1]-Q_user[0];
low =  Q_user[1]-Q_user[0];
size = (Q_user.end()) - (Q_user.begin());
for ( i = 2; i < size ; i++ ) {
    high = max( Q_user[i]-Q_user[i-1], high );
    low  = min( Q_user[i]-Q_user[i-1], low );
}
assert ( (high - low) < 2 );
cout<<"Passed Test 8\n";

cout << "User input specific tests passed." << endl;
```

Used in part 7a.

9.3 Acceptance Test

The following acceptance test is a minor adaptation of the acceptance test presented in [2]. Although the above tests check for everything that was expected of the function as presented in [1], a minor error in that function was discovered, which was corrected and the following new accepatance test was created in [2] to test for this. The main thing that this acceptance test check for (that the above tests do not check for) is the relative size of the quantiles (upper versus). This test ensures that the quantiles produced comply with the above definition by directly check the requirements for being an r-th m-tile, for all of the m-tiles in Q.

Note that the acceptance test is to be run on the saved sequence D0 and Q via iterators.

The test function was addapted to my implementation by dereferencing Q twice. This produces the value which is then used by count_if to do the necessary test.

⟨Define acceptance test function 17a⟩ ≡

```
template <typename RandomAccessIterator, typename InputIterator>
bool check_quantiles(RandomAccessIterator first,
                     RandomAccessIterator last,
                     InputIterator qfirst, InputIterator qlast,
                     int m0)
{
  typedef typename iterator_traits<RandomAccessIterator>::value_type
          value_type;
  ⟨Define needed types and variables 17b⟩
  ⟨Check each of the m-tiles 18a⟩
  return true;
}
```

Used in part 7a.

“We cast the sequence length to a double so that later divisions by, or into, integers will be done using floating point arithmetic rather than integer arithmetic. We do the same for the value m.”[2]

⟨Define needed types and variables 17b⟩ ≡

```
double N = static_cast<double>(last - first);
double m = static_cast<double>(m0);
int r = 0;
```

Used in part 17a.

“The variable r keeps track of which quantile we are checking. In checking each quantile, we use the generic algorithm count_if from the standard library to count the elements in the range that satisfy the predicate given as `count_if`'s last argument. We construct the appropriate predicates using function objects and function object adaptors from the standard library."[2]

⟨Check each of the m-tiles 18a⟩ ≡

```
for (InputIterator i = qfirst; i != qlast; ++i, ++r) {
  int c_less =
     count_if(first, last, bind2nd(less<value_type>(),**i));
  int c_greater =
     count_if(first, last, bind2nd(greater<value_type>(), **i));
  if (!(c_less/N <= r/m)) {
    cout << "Test fails: r = " << r << "; c_less = " << c_less
         << "; r/m = " << r/m
         << "; c_less/N = " << c_less/N << endl;
    return false;
  }
  if (!(c_greater/N <= (m-r)/m)) {
    cout << "Test fails: r = " << r << "; c_greater = " << c_greater
         << "; (m-r)/m = " << (m-r)/m
         << "; c_greater/N = " << c_greater/N << endl;
    return false;
  }
}
```

Used in part 17a.

This code is inserted into the main program to perform the acceptance test.

Note that Q_user is the output sequence tested. Q_user is a vector of iterators which points to D_user. D_user is a container set aside specifically for this test.

⟨Perform quantiles acceptance test 18b⟩ ≡

```
if (check_quantiles(D0.begin(), D0.end(), (Q_user.begin()), (Q_user.end()), user_quantile)) {
  std::cout << "Result passes the acceptance test\n" << std::endl;
  return 0;
} else {
  std::cout << "Result doesn't pass the acceptance test!\n"
            << std::endl;
  return 1;}
```

Used in part 7a.

References

[1] D. R. Musser, "Computing Quantiles, Version 1," *Generic Software Design Course Website*, http://www.cs.rpi.edu/~musser/gsd/quartiles/quartiles.pdf (September 2002). (document), 3, 9.3

[2] D. R. Musser, “Computing Quantiles, Version 1, Revision 1,” *Generic Software Design Course Website*, http://www.cs.rpi.edu/~musser/gsd/quantiles/quantiles.w (October 2002). (document), 1, 2, 1, 3.2, 6, 7, 9.3

[3] *C. R. C. Standard Mathematical Tables, Eleventh Edition*, C. D. Hodgman, editor-in-chief, Chemical Rubber Publishing Co., Cleveland, Ohio, 1957. 2, 1

[4] D. R. Musser, “Introspective Sorting and Selection Algorithms,” *Software—Practice and Experience*, Vol. 27(8), 983–993 (August 1997).

[5] Silicon Graphics, Inc., “Standard Template Library Programmer’s Guide,” *Silicon Graphics, Inc. Website*, http://www.sgi.com/tech/stl/nth_element.html (1993-2002). 6, 7