CSS342 Data Structures, Algorithms, and Discrete Mathematics I

Autumn 2017

Assignment 4: Sorting - Mergesort

Due date: Monday 27 Nov

Purpose

Although merge sort shows the same execution complexity as quick sort, (i.e., *O(n log n)*), its practical performance is much slower than quick sort due to array copying operations at each recursive call. This programming assignments improves the performance of merge sort by implementing a non-recursive, semi-in-place version of the merge-sort algorithm.

In-Place Sorting

In-place sorting is to sort data items without using additional arrays. For instance, quicksort performs partitioning operations by simply repeating a swapping operation on two data items in a given array, which thus needs no extra arrays.

On the other hand, the mergesort algorithm we have studied allocates a temporary array, sorts partial data items in that array, and copies them back to the original array at each recursive call. Due to these repetitive array-copying operations, this implementation of mergesort is much slower than quicksort despite them both having an upper bound of *O(n log n)*.

There has been much interest in research to develop in-place mergesort algorithms. However, almost all of them are impractically complicated. Yet, we can improve the performance of mergesort with adding the following two restrictions:

1. Using a non-recursive method (use iterative method)
2. Using only one additional array, (i.e., a semi-in-place method). You should merge data from the original array into the additional array at the very bottom stage, and thereafter perform the next merge from the additional into the original array. It will then alternate between merging the data from the original into the additional and merging from the additional into the original. In other words, at the first merge, data is moved from the original to the additional array. At the next merge, data is from additional to original array, etc.

Note that we still need to allow data items to be copied between the original and this additional array as many times as *O(log n)*.

Statement of Work

1. Design and implement a non-recursive, semi-in-place version of the merge-sort algorithm.

```
The framework of your mergesort function  should be as follows:
#include <vector>
#include <math.h> // may need to use pow( )
using namespace std;
```

```
template <class Comparable>
void mergesortImproved( vector<Comparable> &a ) {

  int size = a.size( );
  vector<Comparable> b( size );  // this is only one temporary array.

  // implement a nonrecursive mergesort only using vectors a and b.
}
```

Since you are required to write a non-recursive algorithm, the above *mergesortImproved( )* function must not call itself or any other recursive function. Furthermore, the algorithm should *still* be based on the same divide-and-conquer approach.

2. Use the following driver function to verify and evaluate the performance of your nonrecursive, semi-in-place mergesort program.

The code below assumes that your program is implemented in the `mergesortImproved.cpp` file.

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <sys/time.h>
#include "mergesortImproved.cpp"         // implement your mergesort
using namespace std;


// array initialization with random numbers
void initArray( vector<int> &array, int randMax ) {
  int size = array.size( );

  for ( int i = 0; i < size; ) {
    int tmp = ( randMax == -1 ) ? rand( ) : rand( ) % randMax;
    bool hit = false;
    for ( int j = 0; j < i; j++ ) {
      if ( array[j] == tmp ) {
        hit = true;
        break;
      }
    }
    if ( hit )
      continue;
    array[i] = tmp;
    i++;
  }
}

// array printing
void printArray( vector<int> &array, char arrayName[] ) {
  int size = array.size( );

  for ( int i = 0; i < size; i++ )
    cout << arrayName << "[" << i << "] = " << array[i] << endl;
}
```

```
// performance evaluation
int elapsed( timeval &startTime, timeval &endTime ) {
  return ( endTime.tv_sec - startTime.tv_sec ) * 1000000
    + ( endTime.tv_usec - startTime.tv_usec );
}

int main( int argc, char* argv[] ) {
  // verify arguments

  if ( argc != 2 ) {
    cerr << "usage: a.out size" << endl;
    return -1;
  }

  // verify an array size

  int size = atoi( argv[1] );
  if ( size <= 0 ) {
    cerr << "array size must be positive" << endl;
    return -1;
  }

  // array generation

  srand( 1 );
  vector<int> items( size );
  initArray( items, size );
  cout << "initial:" << endl;   // comment out when evaluating performance only
  printArray( items, "items" ); // comment out when evaluating performance only

  // mergesort
  struct timeval startTime, endTime;
  gettimeofday( &startTime, 0 );
  mergesortImproved( items );
  gettimeofday( &endTime, 0 );
  cout << "elapsed time: " << elapsed( startTime, endTime ) << endl;

  cout << "sorted:" << endl;    // comment out when evaluating performance only
  printArray( items, "items" ); // comment out when evaluating performance only

  return 0;
}
```

3. Obtain the usual mergesort and quicksort from

Code is attached, we will use the source code from the textbook.

4. Compare the performance:

of the usual quicksort, the usual mergesort, and your improved mergesort while increasing the array size. These are the sizes you should use for testing: 10, 100, 1000, and 10000.

What to Turn in

Clearly state in your code comments any assumptions you have made. Turn in:
1) your nonrecursive, semi-in-place mergesort program, (i.e., template <class Comparable> void mergesortImproved(vector<Comparable> &a) in "mergesortImproved.cpp".) **Do not use different function name or file name!**
2) A report: in a separate .doc or .docx. The report must include:

a. A one-page output of your improved mergesort program (when #items = 30), and
b. A graph that compares the performance among the usual quicksort, the usual mergesort, and your improvedmergesort.

Grading Guide and Answers

Check the following grading guide to see how your homework will be graded.

```
Program 4 Grading Guidelines

1. Documentation (4pts)
   One page output (a.out 30 will fit one page.)
   Correctness (2pts)          1 ~ 2 errors(1pt)     3+ errors or no results(0pt)

   Performance comparison between your algorithm and ordinary merge/quick sorts
   Your algorithm worked faster when increasing the array size (2pts)
   Little difference between your algorithm and others         (1pt)
   Unsatisfactory comparison                                   (0pt)

2. Correctness (12 pts)

Compilation errors(0pt)

Successful Comparison (4 pts)
  + Non-recursive method(2 pts)
  + Only one additional array besides the original array passed from main(1 pt )
  + One way assignment from one array to another in each iteration ( 2 pts)
  + Your algorithm is still based on the same divide-and-conquer approach (2 pts)

3. Program Organization (4pts)
You must write plenty of comments to help the professor or the grader understand your
code.    Good (2pts)       Poor(1pts)        No explanations(0pt)

Coding style (proper indentations, blank lines, variable names, and non-redundant
code)    Good (2pts)       Poor(1pts)        No explanations(0pt)
```