2011 BIOSTAT 615/815 Homework #2

Due is Tuesday February 7th, 08:30AM (before the class starts)

Problem 1. Evaluation of sorting algorithms

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <ctime>

void radixSortDivide(std::vector<int>& A, std::vector< std::vector<int> >& B, int shift, int mask) {
  for(int i=0; i < (int)A.size(); ++i) {
    B[ (A[i] >> shift) & mask ].push_back(A[i]);
  }
}

void radixSortMerge(std::vector<int>& A, std::vector< std::vector<int> >&B ) {
  for(int i=0, k=0; i < (int)B.size(); ++i) {
    for(int j=0; j < (int)B[i].size(); ++j) {
      A[k] = B[i][j];
      ++k;
    }
  }
}

void radixSort(std::vector<int>& A, int radixBits, int max) {
  int nIter = (int)(ceil(log((double)max)/log(2.)/radixBits));
  int nCounts = (1 << radixBits);
  int mask = nCounts-1;
  std::vector< std::vector<int> > B;
  B.resize(nCounts);
  for(int i=0; i < nIter; ++i) {
    for(int j=0; j < nCounts; ++j) {
      B[j].clear();
    }
    radixSortDivide(A, B, radixBits*i, mask);
    radixSortMerge(A, B);
  }
}

int main(int argc, char** argv) { // sorting software using std::sort
  int tok;
  int max = 0;
  std::vector<int> v;
  if ( argc == 1 ) {
    while( std::cin >> tok ) {
      if ( tok < 0 ) std::cerr << "countingSort works only for nonnegative integer inputs" << std::endl;
      v.push_back(tok);
      if ( max < tok ) max = tok;
    }
  }
  else { // if argument is given, read from file
    std::ifstream fin(argv[1]);
    if ( !fin.is_open() ) {
      std::cerr << "Cannot open file " << argv[1] << " for reading" << std::endl;
      return -1;
    }
```

```
    while( fin >> tok ) {
      v.push_back(tok);
      if ( tok < 0 ) std::cerr << "countingSort works only for nonnegative integer inputs" << std::endl;
      if ( max < tok ) max = tok;
    }
    fin.close();
  }

  std::vector<int> copy = v; // copy the original array

  clock_t start = clock();
  std::sort(copy.begin(),copy.end());
  clock_t finish = clock();
  double duration = (double)(finish-start)/CLOCKS_PER_SEC;

  std::cout << "std::sort - Elapsed time is " << duration << " seconds" << std::endl;

  for(int radixBits = 1; radixBits <= 20; ++radixBits) {
    copy = v;
    start = clock();
    radixSort(copy,radixBits,max);
    finish = clock();
    duration = (double)(finish-start)/CLOCKS_PER_SEC;
    std::cout << "radixSort with " << radixBits << " bits - Elapsed time is " << duration << " seconds" << std::endl;
  }
  return 0;
}
```

1. Write the code above to compare the computational efficiency of `std::sort` algorithm and `radixSort` across various radix bits. Run comparison experiments the following two example datasets

   (a) 1,000,000 random samples from 1 to 1,000,000 (with or without replacement). In the case you need example input files, you can use `shuf-1M.txt.gz` (after decompressing) posted in the class web pages, but it is okay to generate your own input file.

   (b) 1,000,000 random samples from 1 to 1,000. It is okay to use the example input file `rand-1M-3digits.txt.gz` posted on the web page if you want.

   Copy and paste your screen outputs. Which number of radix bits was optimal empirically?

2. Extend the code to incorporate `mergeSort` and `quickSort` (as described in the class). Your code will now print out performance of three sorting algorithms. Copy and paste your outputs, and rank the algorithms (`std::sort`, mergeSort, quickSort, radixSort with best-performing radix) in order of efficiency, for each dataset (a) and (b)

3. For `std::sort`, to which input data was the program more efficient? (a) or (b)? How about `mergeSort` and `quickSort`? Why do you think it is?

Problem 2 - Quicksort algorithms

```
#include <iostream>
#include <fstream>
#include <vector>
#include <climits>

void quickSort(std::vector<int>& A, int p, int r) {
  if ( p < r ) {                // cost 1
    int piv = A[r];             // cost 5
    int i = p-1;                // cost 1
    int tmp;
    for(int j=p; j < r; ++j) {  // cost 1
      if ( A[j] <= piv ) {      // cost 5
        ++i;                    // cost 1
        tmp = A[i];             // cost 5
        A[i] = A[j];            // cost 10
        A[j] = tmp;             // cost 5
      }
    }
    tmp = A[i+1];               // cost 5
    A[i+1] = A[r];              // cost 10
    A[r] = tmp;                 // cost 5
    quickSort(A, p, i);         // cost 20
    quickSort(A, i+2, r);       // cost 20
  }
}

int main(int argc, char** argv) {
  int tok;
  std::vector<int> v;
  if ( argc > 1 ) {
    std::ifstream fin(argv[1]);
    while( fin >> tok ) { v.push_back(tok); }
    fin.close();
  }
  else {
    while( std::cin >> tok ) { v.push_back(tok); }
  }
  quickSort(v,0,(int)v.size()-1);   // cost 20
  for(int i=0; i < v.size(); ++i) {
    std::cout << v[i] << std::endl;
  }
  return 0;
}
```

1. Assume that the computational cost of running a `quickSort` is proportional to the cost as commented in the code and there is no extra cost. Assuming that the input sequences can be any possible permutations of 1,2,3,4,5,6,7,8,

   (a) Give an example of input sequence that will provide the smallest cost among all possible permutations

   (b) Give an example of input sequence that will provide the largest cost among all possible permuations

   (Hint: It will probably be the best to modify the program to calculate the cost for every possible permutations)

Problem 3 - Elementary Data Structures

1. Implement the `remove()` algorithm of binary search tree. E-mail the complete set of `myTree.h`, `myTree.cpp`, `myTreeNode.h`, and `myTreeNode.cpp` (with full features of `insert`, `search`, and `delete`) to the instructor. Also, print out the hard copy of the code.

2. Implement `mySortedArray`, `myList`, and `myTree` as described in the class, and write down the following `main()` function (or copy from the class webpage). Use 50,000 random inputs (such as input files posted in the web page) to evaluate the running time of insertion ad search of each data structure, and report your outcome.

```
#include <iostream>
#include <vector>
#include <ctime>
#include <fstream>
#include <set>
#include "mySortedArray.h"
#include "myTree.h"
#include "myList.h"

int main(int argc, char** argv) {
  int tok;
  std::vector<int> v;
  if ( argc > 1 ) {
    std::ifstream fin(argv[1]);
    while( fin >> tok ) { v.push_back(tok); }
    fin.close();
  }
  else {
    while( std::cin >> tok ) { v.push_back(tok); }
  }

  mySortedArray<int> c1;
  myList<int> c2;
  myTree<int> c3;
  std::set<int> s;

  clock_t start = clock();
  for(int i=0; i < (int)v.size(); ++i) {
    c1.insert(v[i]);
  }
  clock_t finish = clock();
  double duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "Sorted Array (Insert) " << duration << std::endl;

  start = clock();
  for(int i=0; i < (int)v.size(); ++i) {
    c2.insert(v[i]);
  }
  finish = clock();
  duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "List (Insert) " << duration << std::endl;

  start = clock();
  for(int i=0; i < (int)v.size(); ++i) {
    c3.insert(v[i]);
  }
  finish = clock();
  duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "Tree (Insert) " << duration << std::endl;

  start = clock();
```

```
  for(int i=0; i < (int)v.size(); ++i) {
    s.insert(v[i]);
  }
  finish = clock();
  duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "std::set (Insert) " << duration << std::endl;

  start = clock();
  for(int i=0; i < (int)v.size(); ++i) {
    c1.search(v[i]);
  }
  finish = clock();
  duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "Sorted Array (Search) " << duration << std::endl;

  start = clock();
  for(int i=0; i < (int)v.size(); ++i) {
    c2.search(v[i]);
  }
  finish = clock();
  duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "List (Search) " << duration << std::endl;

  start = clock();
  for(int i=0; i < (int)v.size(); ++i) {
    c3.search(v[i]);
  }
  finish = clock();
  duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "Tree (Search) " << duration << std::endl;

  start = clock();
  for(int i=0; i < (int)v.size(); ++i) {
    s.find(v[i]);
  }
  finish = clock();
  duration = (double)(finish-start)/CLOCKS_PER_SEC;
  std::cout << "std::set (Search) " << duration << std::endl;
}
```