2011 BIOSTAT 615/815 Homework #2

Due is Thursday October 6th, 08:30AM (before the class starts)

Problem 1 - Pivoting in quickSort algorithms

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

void quickSort(std::vector<int>& A, int p, int r, int& numcomp) {
  if ( p < r ) {
    // this part corresponds to PARTITION algorithm
    int pivIdx = r; // ** LINE TO EDIT **  CURRENT PIVOT IS RIGHTMOST ELEMENT
    int pivVal = A[pivIdx];
    // move the pivot to right if needed
    if ( pivIdx != r ) { A[pivIdx] = A[r]; A[r] = pivVal; pivIdx = r; }
    int i = p-1;
    int tmp;
    for(int j=p; j < r; ++j) {
      ++numcomp;
      if ( A[j] <= pivVal ) {
        ++i;
        tmp = A[i]; A[i] = A[j]; A[j] = tmp; // swap A[i] and A[j]
      }
    }
    tmp = A[i+1]; A[i+1] = A[pivIdx]; A[pivIdx] = tmp;
    // this part is divide-and-conquer algorithm
    quickSort(A, p, i, numcomp);
    quickSort(A, i+2, r, numcomp);
  }
}

int main(int argc, char** argv) {
  int tok, cost;
  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); }
  }
  cost = 0;
  quickSort(v,0,(int)v.size()-1,cost);
  for(int i=0; i < v.size(); ++i) {
    std::cout << v[i] << std::endl;
  }
  std::cout << "Total # comparisons is " << cost << std::endl;
  return 0;
}
```

1. Create an input sequence of 1 through 20 in the increasing order, and apply to the program above. What is he total number of comparisons?
2. Create an input sequence of 1 through 20 in the decreasing order, and apply to the program above. What is he total number of comparisons?

3. Create an input sequence of 1 through 20 in a random order, and apply to the program above. What is he total number of comparisons?

4. What is the number of comparisons in the worst-case scenario? Is current quickSort algorithm performing efficiently across all above three cases? If not, explain why it performs poorly.

5. Modify the line marked as `"** LINE TO EDIT **"` so that it can select the median among (a) leftmost element (b) rightmost element (c) element located at the center. You may modify the line into multiple lines. Write down how you changed the line, and repeat (1)-(3) using the modified program.

Problem 2. Radixsort Optimization

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

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 3 - Elementary Data Structures

1. Implement `mySortedArray`, `myList`, and `myTree` data structure from lecture note. You must type them on your own. Note that, because you're using templates, all the class body needs to be included in header (.h) file and you won't need .cpp file for each class. Using the `main()` function below, submit the full set of source code to the instructor by E-mail. You may assume that all input values are unique (i.e. no need to specially handle duplicated values)

2. 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.

3. (BIOSTAT815 only) Modify `myList` class to be doubly-linked list. Use the modified version of class when submitting the code.

```
#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;
}
```