CPSC 221: Algorithms and Data Structures
Assignment #2, due Tuesday, 2015 March 17, at 5 pm

Submission Instructions

Type or write your assignment on clean sheets of paper with question numbers prominently labeled. Answers that are difficult to read or locate may lose marks. We recommend working problems on a draft copy then writing a separate final copy to submit.

Each submission should include the cover page (see course website) and names and student IDs of the authors at the top of **each** page. (You may work in pairs, but not in groups of three or more.) On the cover page, make sure that you sign the academic conduct statement. (See: http://www.ugrad.cs.ubc.ca/~cs221/current/syllabus.shtml#conduct.) In keeping with the policy, you should also acknowledge on your first page any collaborators or resources that helped you with the assignment. Also, clearly mark "Hu" or "Khosravi" on the front page, depending which lecture section you attend, so we can return your assignment to you more easily. Finally, **staple** your submission's pages together! We are not responsible for lost pages from unstapled submissions.

Submit your assignment to Box #35 in room ICCS X235. **Late submissions are not accepted.** Note: the number of marks allocated to a question appears in square brackets before the question number.

Questions

[10] 1. In this question, we'll explore some trade-offs between what's asymptotically best versus what might be best in practice.

(a) Suppose you measure that it takes t seconds to sort 100000 values using an implementation of insertion sort. About how long would it take to sort 200000 values using the same implementation?

(b) Suppose you measure that it takes t seconds to sort 100000 values using an implementation of merge sort. About how long would it take to sort 200000 values using the same implementation?

(c) Suppose that we are comparing implementations of insertion sort and merge sort on the same machine. After measuring the run times for various inputs sizes, you determine that the average running time (for input of size n) for your insertion sort is about $T_i(n) = 3n^2$, and the average running time of your merge sort is about $T_m(n) = 22n \lg n$. For what values of n is the insertion sort faster than the merge sort?

(d) Although merge sort runs in $O(n \lg n)$ worst-case time and insertion sort runs in $O(n^2)$ worst-case time, it makes sense to use insertion sort within merge sort as a base case, when subproblems become sufficiently small. Based on the value of n that you computed in part (c), modify the following implementation of msort, which you have previously seen in lecture, and call `insertion_sort(int data[], int lo, int hi)` for "sufficiently small" cases.

```cpp
void msort(int x[], int lo, int hi, int tmp[]) {
  if (lo >= hi) return;
  int mid = (lo+hi)/2;
  msort(x, lo, mid, tmp);
  msort(x, mid+1, hi, tmp);
  merge(x, lo, mid, hi, tmp);
}
```

[5] 2. **Bubble Sort** is a popular sorting algorithm in courses on algorithms, because it's easy to explain and analyze and has a cute name. (It's rarely used in practice because in almost any situation where Bubble Sort is a reasonable choice, Insertion Sort is better. This is why we didn't spend lecture time on it.)

Consider the following implementation of Bubble Sort.

```cpp
void print(int a[], int n){
  for (int i = 0; i < n; i++)
    cout << a[i] << "␣";
  cout << "\n";
}

/*
 Purpose: sorts elements of an array of integers using
     bubble sort

 Param: x - integer array to be sorted
        n - size of the array
 */
void bubbleSort(int x[], int n){
  for(int i = 1; i < n; i++){
    for (int j=n-1; j >= i; j--)
      if (x[j] < x[j-1])
        swap(x[j], x[j-1]);

    print(x,n);
  }
}
```

(a) What is the asymptotic best-case, average-case, and worst-case run times for this implementation of Bubble Sort? (Note that the most standard implementation of Bubble Sort has a better best-case run time than this one does, but ignore that for this problem.)

(b) What would be printed on the screen if you ran `bubbleSort` with the following parameters.

x = {32, 99, 77, 2, 87, 24, 16, 94, 28, 33} and n=10.

Note: **Do this by hand!** (Or at least do the first few iterations by hand.) The point of this question is to give you practice understanding how some code works, not to give you practice typing in code and compiling it. You may want to check your work by typing in the code, but don't just mindlessly type it in and copy down the output!

[25] 3. For this problem, you will formally prove that the above implementation of Bubble Sort correctly sorts the array `x`. More specifically, you will prove that when `bubbleSort` terminates, the data in `x` is in increasing order, and that `x` contains the same data that `x` contained originally.

You may assume without proof that `swap` behaves correctly (swapping the two specified elements in the array), and you may ignore the call to `print`. You may also assume that no two integers in the array have the same value.

(a) Let's start with an easy loop-invariant proof. Prove that the following loop invariant holds for the `for`-loop in lines 15–17:

> Loop invariant: The subarray $x[j \dots n-1]$ is a permutation of the values that were in $x[j \dots n-1]$ at the time that the loop started.

A *permutation* just means that the same data is in those array entries, but possibly in a different order. Be sure to write your answer as a base case (for when the loop starts initially), and an inductive step (in which you assume the loop invariant holds at the top of the loop and prove that it still holds at the bottom of the loop). You may want to rewrite the loop as a `while`-loop if you wish, but that's optional.

(b) Next, prove that the following loop invariant also holds for the `for`-loop in lines 15–17:

> Loop invariant: The value of $x[j]$ is the smallest value in all of $x[j \dots n-1]$.

(c) Prove that the `for`-loop in lines 15–17 terminates, and then use the termination condition and the loop invariants you proved in parts (a) and (b) to prove that whenever the code reaches line 18, $x[i-1]$ contains the smallest value of $x[i-1 \dots n-1]$, and that $x[i-1 \dots n-1]$ contains a permutation of the data that was originally in $x[i-1 \dots n-1]$. (Hint: This should be **really** short and easy. If you find yourself trying to write more than a few sentences, you're probably doing something wrong.)

(d) Now that you've proven the inner `for`-loop correct, you can prove invariants for the outer `for`-loop from lines 14–20. In particular, prove that the following loop invariant holds for the for loop in lines 14-20

> Loop invariant: the subarray $x[0 \dots i-2]$ contains the $i-1$ smallest values originally in $x[0 \dots n-1]$, in sorted order, and the array always contains a permutation of the original array.

(e) Prove that the outer `for`-loop in lines 14–20 terminates, and then use the termination condition and your loop invariant to prove that `bubbleSort` sorts the entire array correctly. (Hint: This is almost as easy as part (c), but there's one extra step to prove that you sorted the *entire* array.)