University of Waterloo
CS240, Winter 2010
Assignment 2

Due Date: Wednesday, February 10, at 5:00pm

Please read `http://www.student.cs.uwaterloo.ca/~cs240/w10/guidelines.pdf` for guidelines on submission. In particular, don't forget to make a **cover page** (`https://www.student.cs.uwaterloo.ca/~isg/makeCover?course=cs240`) and attach it to the front of any written assignment.

This assignment has 4 problems, each of which has multiple parts. For problems 1–3, you may use the result of any part to solve any of the following parts, even if you do not complete both parts. Problems 1–3 and Problem 4(e) are written problems; submit your solutions on paper to the assignment boxes in MC. Problem 4(a–d) is a programming problem; submit all four indicated files electronically. There are 92 marks available, plus up to 14 bonus marks for the problems indicated.

There are three elementary sorting methods which have been covered in previous courses. We give pseudocode for them here, in case you need to refer to them in this assignment. You may also refer to any of the algorithms covered in the course notes.

```
insertion-sort($A$)
$A$: array of size $n$
1.  for $i \leftarrow 1$ to $n-1$ do
2.      $j \leftarrow i$
3.      while $j \geq 1$ and $A[j] < A[j-1]$ do
4.          swap($A[j], A[j-1]$)
5.          $j \leftarrow j-1$
```

```
selection-sort($A$)
$A$: array of size $n$
1.  for $i \leftarrow 0$ to $n-2$ do
2.      for $j \leftarrow i+1$ to $n-1$ do
3.          if $A[j] < A[i]$ then
4.              swap($A[i], A[j]$)
```

```
bubble-sort($A$)
$A$: array of size $n$
1.  for $i \leftarrow 1$ to $n-1$ do
2.      for $j \leftarrow i$ to $n-1$ do
3.          if $A[j] < A[j-1]$ then
4.              swap($A[j], A[j-1]$)
```

Problem 1 Counting swaps [5+5+3+7=20 marks]

The *swap* operation takes two elements in the same array and exchanges their contents. Most of the algorithms in modules 1–3 use only *swap*s to move items in an array (MergeSort being a notable exception).

These algorithms fall under the *comparison-swap* model, very similar to the comparison model discussed in lectures, but slightly more restricted: In the *comparison-swap* model, data can only be accessed in two ways:

- comparing two elements
- swapping two elements

Part 1(a) [5 marks]

Consider the *partition* problem: Given an array A of size n and a pivot index p, define $pivot = A[p]$ and rearrange A so that all items less than or equal to $pivot$ come first, followed by $pivot$, followed by all items greater than $pivot$.

Prove that any algorithm which solves the partition problem in the comparison-swap model must perform at least $\lceil n/2 \rceil$ swaps in the worst case. (Hint: what is the worst-case input for this problem?)

Part 1(b) [5 marks]

Show that the partition algorithm on slide 5 of module 3 performs at most $\lceil n/2 \rceil + 1$ swap operations for any input of size n.

Part 1(c) [3 marks]

Prove that any sorting algorithm in the comparison-swap model must perform $\Omega(n)$ swaps in the worst case to sort an array of size n.

Part 1(d) [7 marks]

Give pseudocode for a sorting algorithm in the comparison-swap model that is asymptotically *swap-optimal*, meaning that it only performs $O(n)$ swaps in the worst case. (Hint: this can be accomplished with a slight modification to one of the sorting algorithms we have seen.) Give a brief justification that your algorithm performs $O(n)$ swaps in the worst case.

Problem 2 Sortedness detection [2+2+(5)+10+6=20 marks]

We say an array A of size n is *sorted* if and only if $A[i] < A[j]$ for all integers i, j satisfying $0 \le i < j < n$.

In this problem, we will consider algorithms to solve the *sortedness* problem: Given an array A of size n, determine whether A is sorted. If it is, return "true"; otherwise, return "false".

Part 2(a) [2 marks]

Give pseudocode for a simple, comparison-based algorithm called *sort-detect1* that solves the sortedness problem. Show that your algorithm has $\Theta(n)$ worst-case running time and $\Theta(1)$ best-case running time.

Part 2(b) [2 marks]

Consider the following algorithm *sort-detect2*:

```
sort-detect2($A$)
$A$: array of size $n$
1.  for $i \leftarrow 0$ to $\min(\lfloor n/2 \rfloor, \lceil \lg n \rceil) - 1$ do
2.      if $A[2i] > A[2i+1]$ then
3.          return false
4.  return sort-detect1($A$)
```

Show that *sort-detect2* also solves the sortedness problem with $\Theta(n)$ worst-case and $\Theta(1)$ best-case running times.

Part 2(c) Bonus [maximum 5 marks]

There are $n!$ total possible orderings, or permutations, of an input array A with length n. Since *sort-detect2* is comparison-based, its running time depends only on the ordering of A.

For any permutation σ of n elements, define $f(\sigma)$ to be the largest value that i is ever set to for an input A with ordering σ. So for example, for any ordering σ where $A[0] > A[1]$, $f(\sigma) = 0$.

Prove that the number of permutations σ such that $f(\sigma) \ge k$, for any integer $k \ge 0$, is at most $n!/2^k$.

Part 2(d) [10 marks]

Prove that the average running time of *sort-detect2* is $O(1)$. You should use your result from Part (c). It may also be helpful to use the fact that

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2.$$

Part 2(e) [6 marks]

Briefly explain why randomizing the *sort-detect2* algorithm would **not** give $O(1)$ expected worst-case time.

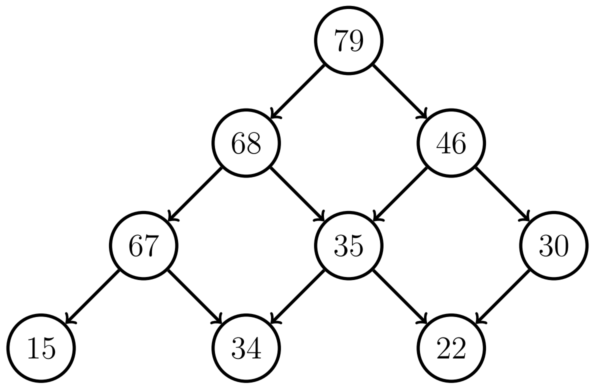
Problem 3 Pyramids [3+(4)+3+5+5+8+(5)=24 marks]

A *pyramid* is a data structure similar to a heap that can be used to implement the priority queue ADT.

As with a heap, a pyramid is defined by two properties:

- **Structural property**: A pyramid P consists for ℓ levels, $\ell \geq 0$. The ith level, for $0 \leq i < \ell$, contains at most $i + 1$ entries, indicated as $P_{i,j}$ for $0 \leq j \leq i$. All levels but the last are completely filled, and the last level is left-justified.
- **Ordering property**: Any node $P_{i,j}$ has at most two *children*: $P_{i+1,j}$ and $P_{i+1,j+1}$, if those nodes exist. The priority of a node is always greater than or equal to the priority of either child node.

For example, the following diagram shows a pyramid with 9 nodes and 4 levels. The arrows indicate "$\geq$" relationships.



Part 3(a) [3+(4) marks]

A pyramid P with n nodes can be stored in an array A of size n, similarly to an array-based heap. So, for example, if $n \geq 1$, the top of the pyramid $P_{0,0}$ will be stored in $A[0]$.

Give formulas for the array index that the following pyramid entries will have. Assume that n, i, and j are such that all indicated pyramid nodes actually exist. You do not need to justify your answers.

i. $P_{i,j}$

ii. Left and right children of $P_{i,j}$

iii. Left and right parents of $P_{i,j}$

iv. [**maximum 2 bonus marks**]
(Left and right children of the node corresponding to $A[i]$

v. [**maximum 2 bonus marks**]
Left and right parents of the node corresponding to $A[i]$

Part 3(b) [3 marks]

Give upper and lower bounds for the number of nodes n in a pyramid P with ℓ levels, where $\ell \geq 1$. For example, a pyramid with 2 levels has at least 2 and at most 3 nodes. Make your bounds as tight as possible.

Part 3(c) [5 marks]

Give pseudocode for the *delete-max* operation that takes a pyramid stored in an array A of size n, removes the largest element from the pyramid, and returns it.

Show that your algorithm has time complexity $O(\sqrt{n})$.

Part 3(d) [5 marks]

Give pseudocode for the *insert* operation that takes a pyramid stored in an array A of size n and an element x and inserts the new element into the pyramid.

Show that your algorithm has time complexity $O(\sqrt{n})$.

Part 3(e) [8 marks]

Consider the *contains* problem: Given an array A of size n and an number x, determine whether x is an element of A.

For example, if A is sorted, the contains problem can be solved in $O(\log n)$ time by using a binary search.

Give pseudocode for an algorithm to solve the contains problem when the input array A is a pyramid as described above.

Show that the running time of your algorithm is $\Theta(\sqrt{n} \log n)$. (Hint: A pyramid contains some sorted lists.)

Part 3(f) Bonus [maximum 5 marks]

Prove that any algorithm to solve the contains problem on an **ordinary heap** must take $\Omega(n)$ time.

Problem 4 Comparing comparisons [3+8+8+5+4=28 marks]

In class, we analyzed the number of comparisons performed by the *heap-sort* and *quick-sort1* algorithms and determined that they both perform $\Theta(n \log n)$ comparisons for an input array A of size n.

In this problem, you will implement these two algorithms and count the *exact* number of comparisons they perform on randomly-chosen inputs.

Facilitating this task will be the `CompareList` class, most of which has been written for you. This class represents a list A containing the numbers 0 through $n-1$ and allowing comparisons and swaps between elements. The following methods have already been defined:

- `CompareList(int n)`: Creates a new `CompareList` initialized to $[0, 1, \ldots, n-1]$.
- `void copy(CompareList L)`: Copies the list stored in L into this list.
- `int size()`: Returns the size of the list
- `boolean lessThan(int i, int j)`: Returns `true` if and only if $A[i] < A[j]$.
- `void swap(int i, int j)`: Swaps $A[i]$ and $A[j]$
- `void shuffle()`: Randomly changes the order of the array.

Part 4(a) [3 marks]

We want to use the `CompareList` class to count the number of comparisons (that is, `lessThan` operations) that are performed. Complete the implementation of the `getCount` and `resetCount` methods, adding a new private field and modifying the `lessThan` method accordingly.

File: `compare_list.h` or `CompareList.java`

Part 4(b) [8 marks]

Complete the implementation of the `HeapSort` class. The only method you need to write is `sort()`, which sorts the `CompareList` object A using the HeapSort algorithm. You may write and use whatever helper methods you like. (Hint: You should **not** have to write a *bubble-up* function.)

File: `heap_sort.h` or `HeapSort.java`

Part 4(c) [8 marks]

Complete the implementation of the `QuickSort` class. The only method you need to write is `sort()`, which sorts the `CompareList` object A using the *quick-sort1* algorithm presented in class. Again, you may write and use whatever helper functions you like.

File: `quick_sort.h` or `QuickSort.java`

Part 4(d) [5 marks]

Write an executable `main` method that does the following:

1. Initialize two integers n and *reps* from command line arguments
2. Create two `CompareList` objects A and B, both of size n
3. Shuffle A, copy A to B, and reset both counters
4. Call `HeapSort.sort()` on A and `QuickSort.sort()` on B
5. Print out the number of comparisons used by each algorithm. The format is the following, on a single line: the number of comparisons used by HeapSort, followed by a single tab character, followed by the number of comparisons used by QuickSort.
6. Repeat steps 3–5 *reps* times.

File: `sorting.cc` or `Sorting.java`

Part 4(e) [4 marks]

Run some experiments on various sizes from $n = 10$ to 1000 and briefly summarize the results. Based on your observations, make some generalizations about the number of comparisons made by HeapSort versus QuickSort.

This is a written part; no files to submit for this part.