CS302 Midterm Exam – Answers & Grading

James S. Plank
September 30, 2010

Question 1

Part 1, Program A:

This program reads integers on standard input and stops when it encounters EOF or a non-integer. It prints out each integer, one per line in the order in which it was read.

Part 1, Program B:

This program reads integers on standard input and stops when it encounters EOF or a non-integer. It prints out each integer, one per line in reverse of the order in which it was read.

Part 1, Program C:

This program reads integers on standard input and stops when it encounters EOF or a non-integer. It prints out each integer, one per line sorted from smallest to largest. If duplicate integers are entered, it only prints that integer once.

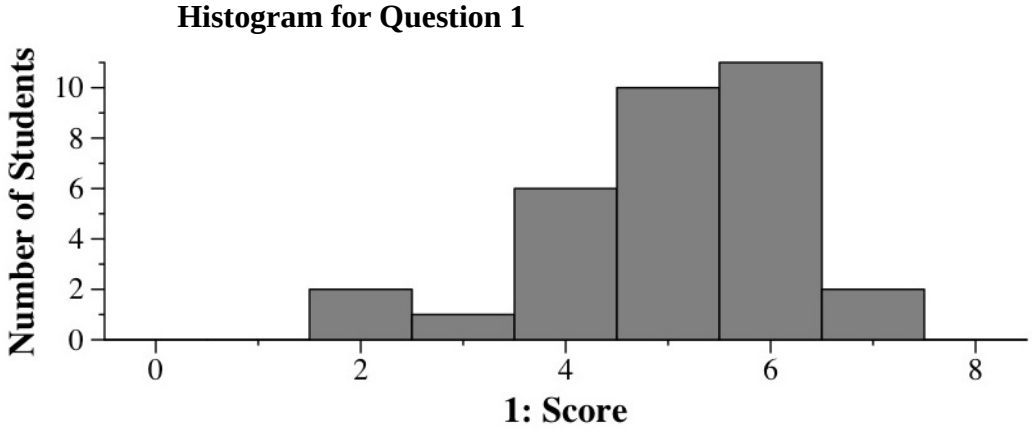
Part 2: Program A: This is $O(n)$, where n is the number of integers entered.

Part 2: Program B: This is $O(n^2)$, where n is the number of integers entered. This is because inserting into a beginning of a vector moves all elements one place to the right first. You should never do this.

Part 2: Program C: This is $O(n \log(m))$, where n is the number of integers entered and m is the number of distinct integers. Think about this. If you enter the value 1 one thousand times, the program will insert the value into to set once, and after that it will simply discard the value.

Grading:

- Reads integers from standard input: 0.5 points
- Stops on EOF or non-integer: 0.5 points
- Prints each integer on its own line: 0.5 points
- Program A: Prints them in the order entered: 1 point
- Program B: Prints them in reverse order: 1 point
- Program C: Prints them in sorted order: 0.5 points
- Program C: Doesn't print duplicates: 0.5 points
- Program A: $O(n)$: 1 point
- Program B: $O(n^2)$: 1 point
- Program C: $O(n \log something)$: 1 point
- Program C: $O(n \log m)$: 0.5 points



Question 2

Recall this program from the lecture notes on strings and vectors:

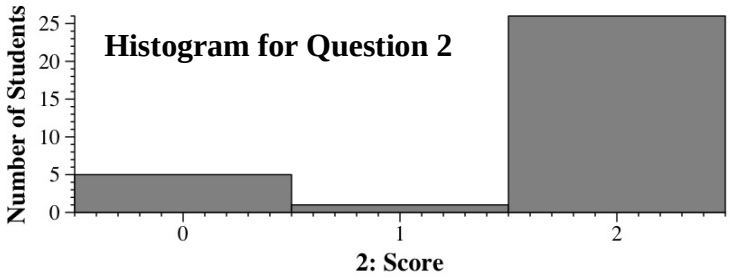
The four **cout** lines all feature overloading – the plus operator, the equality operator, and the less than operator are all examples of operator overloading, since they are in reality members of the **string** class.

Other examples:

- Using ++ with an iterator.
- Using << with cout and >> with cin
- Using [] with maps.

Grading: 2 points

```
#include <stdio.h>
#include <iostream>
#include <string>
using namespace std;

main()
{
  string s1, s2, s3;

  s1 = "AAA";
  s2 = "BBB";
  s3 = "BBB";

  cout << s1 + s2 << endl;
  cout << (s1 == s2) << endl;
  cout << (s2 == s3) << endl;
  cout << (s1 < s3) << endl;
}
```



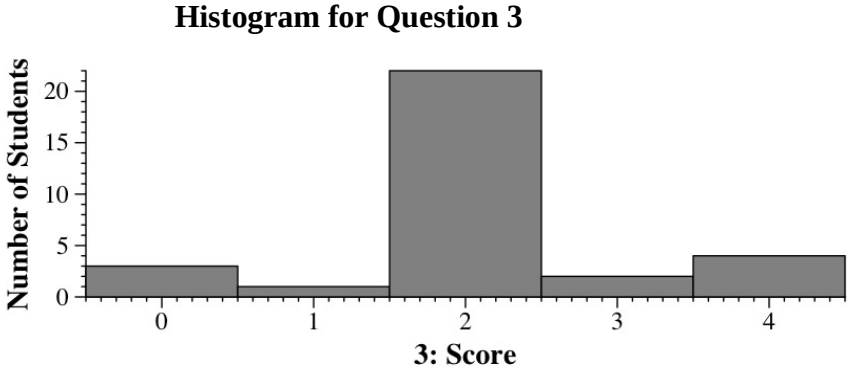
Question 3

The procedure as written does not use a reference parameter. Therefore, it makes a copy of **v** whenever it is called, and throws away the copy when it is done. This is expensive, especially when **v** is large. We can easily rewrite it and convert v to a reference parameter. However, we must be careful because the procedure as written modifies **v**. Thus, if we simply add an ampersand in front of **v**, we've introduced a bug into the procedure, because the caller's vector will be modified. Fixing this bug is relatively easy – we can calculate the average without modifying **v**. The answer is below:

```
#include <iostream>
#include <vector>
using namespace std;

double weighted_average(vector <double> &v)
{
  int i;
  double total;

  total = 0;
  for (i = 0; i < v.size(); i++) total += (v[i]*(double)(i+1));
  if (v.size() == 0) return 0;
  return total / (double) v.size();
}
```

Grading

2 points for stating the problem (making a copy). 2 points for the program. If you simply shoved an ampersand in front of v in the parameter specification, you got 0.5 points for the second part. Those of you said that there was a bug in the program were incorrect – the problem stated that you need to make it "better," not "correct." The program was specified as calculating a weighted average, which it does. Calculating this average should not modify the values.



Question 4

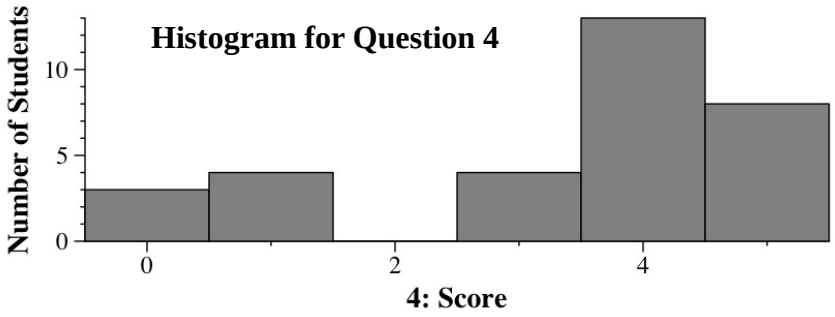
A heap is valid if each node is less than or equal to its children. Recall from the Priority Queue lecture notes that if a node on the heap is at vector element *i*, then its left child is at (2*i*+1), and its right child is at (2*i*+2). Therefore, a simple heap checker simply runs through every element of the vector, and as long as it has children, it tests to see if the children are greater than or equal to it. That is on the left. Perhaps an easier way is to have each element except the root check itself with its parent, which is at node (*i*-1)/2. That is on the right.

```
int is_heap(vector <double> &h)
{
  int i, lc, rc;

  for (i = 0; i < h.size(); i++) {
    lc = 2*i+1;
    if (lc >= h.size()) return 1;
    if (h[lc] < h[i]) return 0;
    rc = 2*i+2;
    if (rc >= h.size()) return 1;
    if (h[rc] < h[i]) return 0;
  }
  return 1;
}
```

```
int is_heap(vector <double> &h)
{
  int i, parent;

  for (i = 1; i < h.size(); i++) {
    parent = (i-1)/2;
    if (h[i] < h[parent]) return 0;
  }
  return 1;
}
```

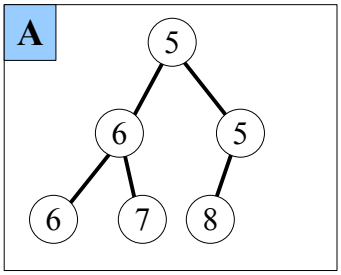
Grading

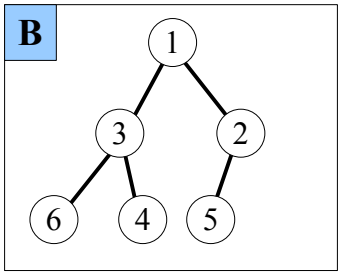
Five points. You got 3 if you ran through the heap checking parents with children or children with parents, even with some bugs (like calculating the children/parents incorrectly, or not checking to see whether children actually exist). The extra points come from being bug free.

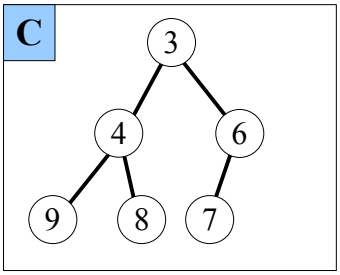


Question 5

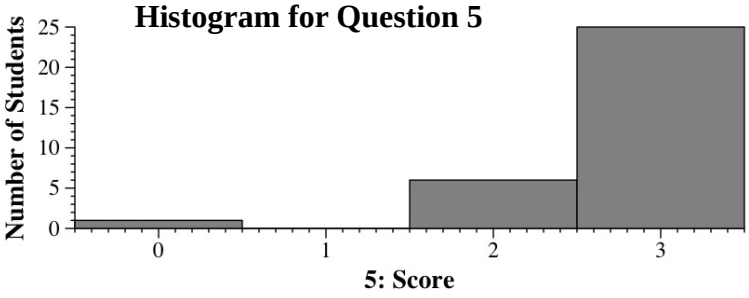
Pop() will return the root, replace it with the rightmost element on the bottom row, and percolate down:





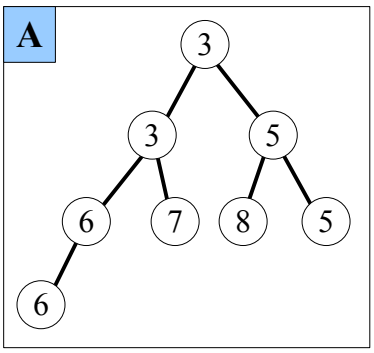


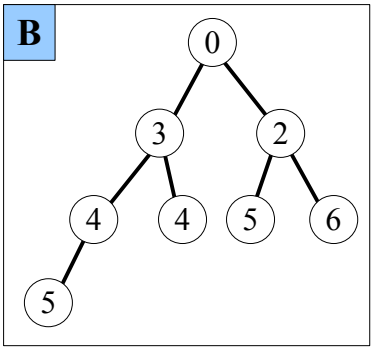
Grading: 1 point each

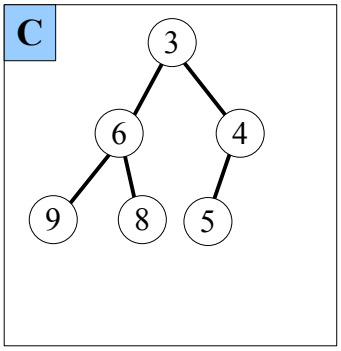


Question 6

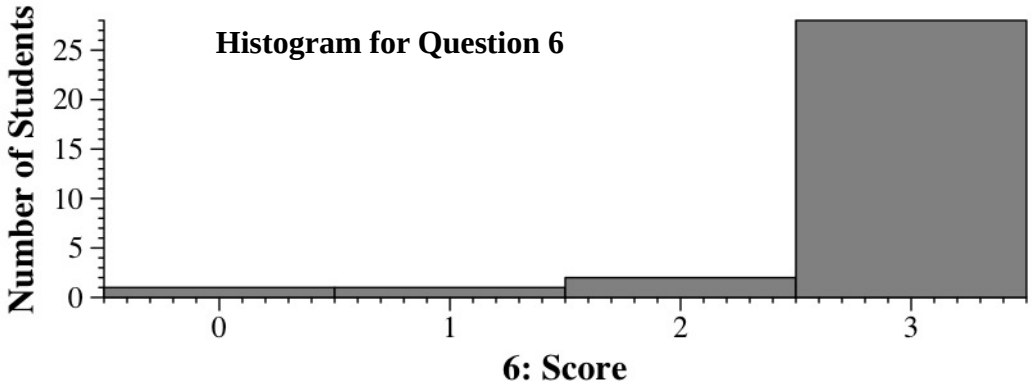
To push onto a heap, you create a new node at right side of the lowest level. If that level is incomplete, you create a new level. After that, you percolate up so that the heap is valid.







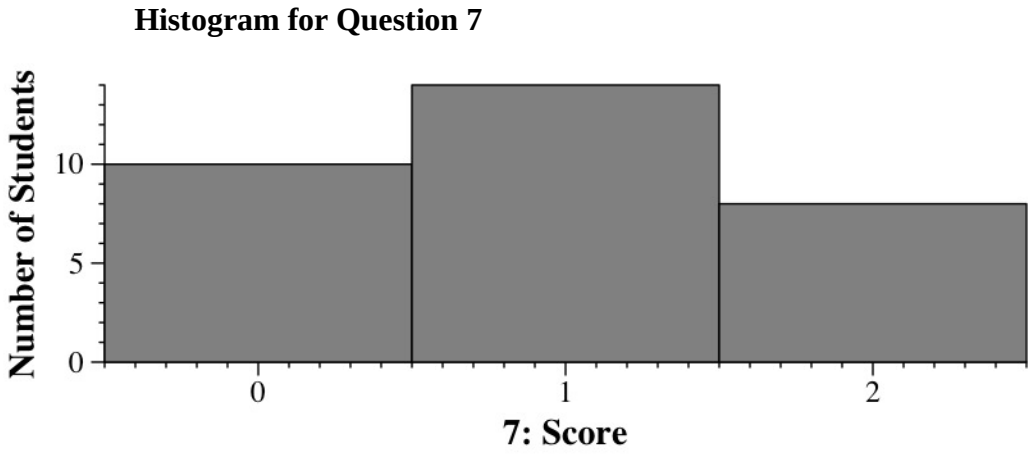
Grading: 1 point each



Question 7

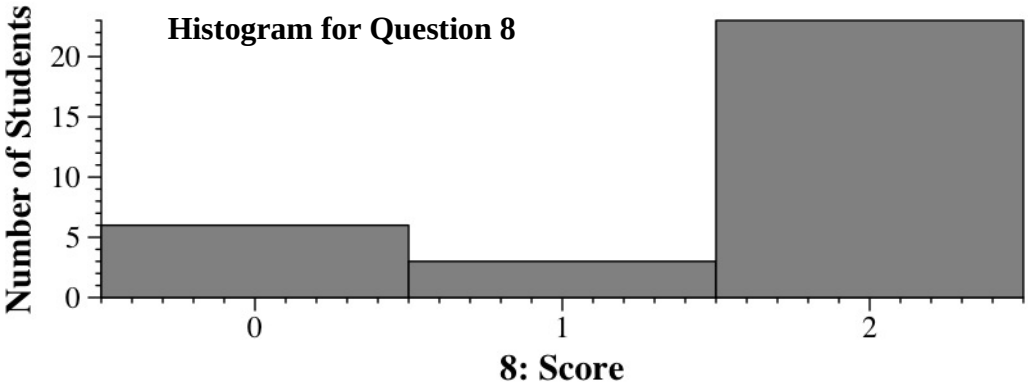
Both implementations perform Push() and Pop() in *O(log n)* time, where *n* is the number of elements on the priority queue. However, since the heap implementation uses a vector rather than a binary tree (which is how multisets are implemented), it will be faster with each operation. Why? Because the vector uses less memory, and does not have to mess with pointers or call **malloc()/new//free()/delete**. That's reason number one.

Reason number two is because you can create a valid heap from a vector in *O(n)* time. See the Priority Queue lecture notes for an explanation. Using a multiset results in *O(n* log *n)* time. **Grading**: 1 point for each reason.



Question 8

If the input is nearly sorted, then insertion sort will run faster. In the Sorting lecture notes, I give an example where each element of the vector is within ten places of its final resting place. In that case, insertion sort will run in linear time. Bubble and selection sort do not have that property. It is not correct to say that insertion sort's best case is *O(n)* and the others is not – it's not the best case that we care about, but the fact that a large class of input (nearly sorted vectors) runs in *O(n)*. **Grading**: 2 points.

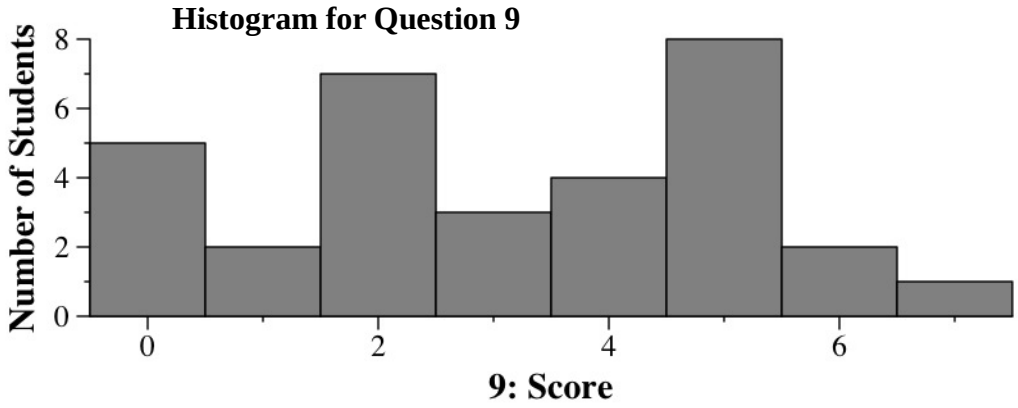


Question 9

Straight from the lecture notes:

```
void insertion_sort(vector <double> &v)
{
  int i, j, best;
  double tmp;

  if (v.size() == 0) return;
  best = 0;
  for (i = 1; i < v.size(); i++) {
    if (v[i] < v[best]) best = i;
  }
  tmp = v[0];
  v[0] = v[best];
  v[best] = tmp;

  for (i = 2; i < v.size(); i++) {
    tmp = v[i];
    for (j = i-1; v[j] > tmp; j--) v[j+1] = v[j];
    v[j+1] = tmp;
  }
}
```

Grading: 4 points for having it correct modulo some bugs. 1.5 more for having it completely correct. 1.5 points for properly doing the sentinel.



Question A

In each output, I start with the original vector:

Bubble Sort

```
E H B F A U G M S P
E B F A H G M S P U
B E A F G H M P S U
```

1 point.

Selection Sort

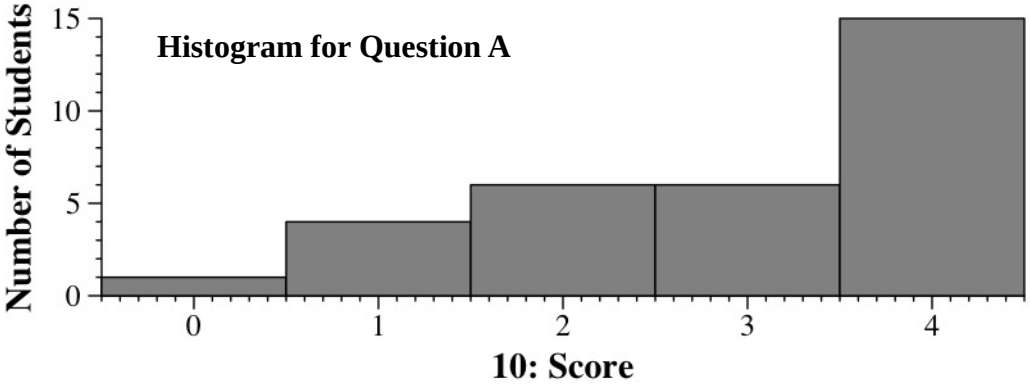
```
E H B F A U G M S P
A H B F E U G M S P
A B H F E U G M S P
A B E F H U G M S P
```

1 point.

Insertion Sort

```
E H B F A U G M S P
E H B F A U G M S P
B E H F A U G M S P
B E F H A U G M S P
A B E F H U G M S P
A B E F H U G M S P
A B E F G H U M S P
A B E F G H M U S P
A B E F G H M S U P
A B E F G H M P S U
```

2 points



Question B

This is pretty much straight from the topcoder problems:

```
#include <iostream>
using namespace std;

main()
{
  int i, j;

  for (i = 0; i < (1 << 12); i++) {
    for (j = 0; j < 12; j++) {
      if (i & (1 << j)) cout << "A"; else cout << "B";
    }
    cout << endl;
  }
}
```

Grading:5 points total: 3 points for getting the idea pretty much right, 2 points for the details.