Some CPSC 259 Sample Midterm and Final Exam Questions (Part 1)
Sample Solutions

DON'T LOOK AT THESE SOLUTIONS UNTIL YOU'VE MADE AN HONEST ATTEMPT AT ANSWERING THE QUESTIONS YOURSELF.

1. {4 marks} You will get 3 marks if you get one wrong, 2 marks if you get 2 wrong, and 0 marks if you get 3 or more wrong. Circle ALL correct answers because there may be more than 1 correct answer!

Which of the following statements about stacks, queues, and general lists are true?

a) Stacks and queues can be implemented using arrays.
b) A queue allows a user to retrieve nodes using the LIFO principle, in O(1) time.
c) A stack allows a user to retrieve nodes using the FIFO principle, in O(1) time.
d) Suppose we have a circular array holding only integers, and having a capacity (maximum size) of *n*. If the array is a circular array, then we can store more than *n* entries in it at any given time because we can make use of the modulus operator.
e) Using a doubly-linked list, it is not possible to keep the nodes sorted.

ANSWER: Circle (a) only

2. {2 marks} Put these Big-O sets of functions into ascending order of complexity:

$O(n^{1/2})$, $O(\lg n)$, $O(n^{1/3})$, $O(n)$, $O(1/n)$.

ANSWER: $O(1/n)$, $O(\lg n)$, $O(n^{1/3})$, $O(n^{1/2})$, $O(n)$

3. {6 marks} Compute the complexity of the following function, in terms of number of dollar signs ($) printed, as a function of *N*. Note that *N* is distinct from *n*. Justify your answer by briefly explaining your work (i.e., by showing your calculations). It is *not* necessary to compute witnesses.

```
void myCalc(int N)
{
      int  w, k;

      j = 0;
      n = N;

      while (n > 1)
      {
            n = n / 32;
            j++;
      }
```

```
        for (w = 0; w < (N/3 + 32); w++)
              for (k = 1; k <= j*N; k++)
                    printf("$");
  }
```

ANSWER:

- `Assignment statements have O(1) complexity`
- `WHILE loop has O(log_32 n) = O(lg N) complexity`
- `Outer FOR loop has O(N) complexity`
- `Inner FOR loop has O(lg N) * O(N) complexity ... because we looped j times N, and j came from the O(lg N) calculation`
  ⇨ `O(N lg N) complexity`
- `printf has O(1) complexity`
- `Overall:  O(1) + O(lg N) + O(N)*O(N lg N)*O(1) =` $O(N^2 \lg N)$ `complexity`

4. {10 marks} The following C function traverses a doubly-linked list consisting of 0 or more nodes from the current node, in the direction indicated by the input parameter (going left, or going right). For example, if you are at node "Susan" and you're supposed to go left, then, on the screen, print out all the names found in the nodes from the one before Susan to the start of the list, in that order (but do not include Susan (in the current node)). The same idea applies when going to the right.

Assume that the following declaration is correct:

```
      struct Node {
            char *          name;
            struct Node *   prev;
            struct Node *   next;
      };
```

Note that if `curr` is a `struct Node` pointer, then:

```
      printf("%s\n", curr->name);
```

prints out the name.

Finish the following function. There is no dummy node at the start or the end of the list (i.e., it's just an ordinary list of nodes, possibly empty); so, you'll have to test for where the list ends. Note also that the function returns the number of names that were printed.

```
int printNodesinOneDirection(struct Node * curr, char LeftOrRight)
{
// POST-CONDITION: This function starts at Node "curr" and then prints out
// all the names in the nodes to the left ('L') or right ('R') of curr (but
// excluding the name in curr.  The function returns a count of the number
// of names printed on the screen.
// PRE-CONDITION: curr is either NULL or points to a valid Node shown above
//                LeftOrRight is either 'L' or 'R'

/* Declare any needed variables here.  Follow this with your code. (Some code
is provided further below.) */


      int count = 0;

      if (curr == NULL)     /* check for empty list */
            return 0;



      if (LeftOrRight == 'L')       /* be sure to do the 'R' part, too */
      {
            /* print out all the names of the nodes to the left of curr's
               node */

            curr = curr->prev;
            while (curr != NULL)
            {
                  printf("%s\n", curr->name);
                  count++;
                  curr = curr->prev;
            }
            return count;
      }

      /* otherwise we must be going to the right */
      curr = curr->next;
      while (curr != NULL)
      {
            printf("%s\n", curr->name);
            count++;
            curr = curr->next;
      }
      return count;
}
```

5. {2 marks} If an algorithm runs in O($n \lg n$) time, it is not possible to find another algorithm that performs *worse* on the same task:

a) True
b) False

ANSWER: Circle (b)

6. {2 marks} What is the most appropriate Big-O complexity for the following code fragment:

```
for (k=1; k < 10*n; k++)
{
      print k;
      print n*n;
}
```

a) O(*n*).
b O(10*n*).
c) O(1).
d) O(*k*).
e) O(n^2)
f) None of the above.

ANSWER: Circle (a) because the code fragment loops O(*n*) times, and the two statements inside the FOR loop are both O(1) operations.

7. {2 marks} What is the most appropriate Big-O complexity of the task of finding *and* deleting an element in a singly linked list (with *n* nodes) in the worst case? You can assume that you have a head pointer.

a) O(1)
b) O(*n*)
c) O(*n*+1)
d) O(n^2)
e) O(*n* + *n*)
f) None of the above.

ANSWER: Circle (b)

8. {6 marks} Consider the following C code. After each line, write down the contents of *x* and *y*. If at any point you cannot determine the contents, write "unknown".

```
int main(void)
{
      int  x = 5, y = 15;
      int  * p1,  * p2;

      p1 = &x;        // x contains _____5____; y contains ____15_____
      p2 = &y;        // x contains _____5____; y contains ____15_____
      *p1 = 10;       // x contains ____10____; y contains ____15_____
      *p2 = *p1;      // x contains ____10____; y contains ____10_____
      p1 = p2;        // x contains ____10____; y contains ____10_____
      *p1 = *p2+10;   // x contains ____10____; y contains ____20_____

      system("pause");
      return 0;
}
```

9. {10 points} Write a function merge to merge two sorted arrays *a* and *b* into one, so that merged version of the array contains all elements of the two arrays combined into one list and they're in proper numeric order from smallest to largest). Use the following specification:

```
void merge( int a[], int n1, int b[], int n2, int c[] )
{
     Pre-conditions: a is a sorted array of integers, indexed from 0
     to n1-1; b is a sorted array of integers, indexed from 0 to n2-1;
     c is a previously allocated array (in the caller) with size n1+n2

     Post-conditions: c contains all the items from a, plus all the
     items from b, in sorted order, indexed from 0 to n1+n2-1; that
     is, the array c contains the merged result of the items in arrays
     a and b.
```

Use the space given to plan your algorithm before you start coding.

Hint: There are two steps: comparing the next unused element from each of a and b, and pick the smaller of the two to add to c. Be sure to correctly handling the remaining data once a or b is empty.

```
     /* Note:   Comments are optional, but might help with part
                marks. */

     int   ax = 0;          /* a's index */
     int   bx = 0;          /* b's index */
     int   cx = 0;          /* c's index */


     /* Compare the next entry of each list to see which is smaller.
        Write the smaller of the two into the new array. */

     while ( ax < n1  &&  bx < n2 )
     {
          if ( a[ax] < b[bx] )
          {
               /* a's next value was smaller */
               c[cx] = a[ax];
               cx++;
               ax++;
          }
          else
          {
               /* b's next value was smaller */
               c[cx] = b[bx];
               cx++;
               bx++;
          }
     }
```

... continued on next page

```
     /* we must have run out of either a’s entries or b’s entries */

     while ( ax < n1 )
     {
          /* write the rest of a’s entries */
          c[cx] = a[ax];
          cx++;
          ax++;
     }

     while ( bx < n2 )
     {
          /* write the rest of b’s entries */
          c[cx] = b[bx];
          cx++;
          bx++;
     }

}  /* end of function */
```