**Some CPSC 259 Sample Final Exam Questions (Part 3)**
**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 are true about sorting algorithms?

a) Quicksort runs in O(lg $n$) average time, and O($n^2$) worst case time.
b) In the worst case, Mergesort runs in O($n$ lg $n$) time.
c) There are no pivot elements in Mergesort.
d) Quicksort and Insertion Sort perform equally well (when sorting a list of numbers).
e) It is not possible to perform a comparison-based sort in less than O($n$ lg $n$) time for $n$ arbitrary elements.

**ANSWER**: Circle (b), (c), and (e).

2. {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 are general design goals of an *efficient algorithm*?

a) It must be correct (i.e., it must work).
b) It can easily be turned into program code by a programmer of average ability.
c) As a rule of thumb, it should run in less than a minute on today's PCs.
d) It shouldn't use more than 64K of memory.
e) It probably shouldn't take O($k^n$) steps, where $k > 1$, and $n$ is the size of the input.

**ANSWER**: Circle (a) and (e).

3. {3 marks} Perform Quicksort on the following array of integers. Show the result of the array just before the first recursive call (for the left partition) is made. Use the same method shown in class (Bentley's algorithm). Don't write any code. Assume that the pivot value is taken from the first subscript value (which in the case below for the full array is index 0).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 24 | 55 | 33 | 19 | 7 | 44 | 12 | 50 | 22 |

**ANSWER**:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 22 | 19 | 7 | 12 | **24** | 44 | 55 | 50 | 33 |

4. {7 marks} A security guard has $m$ keys and $n$ locks, but forgot which keys open which locks. (The keys are just for these locks.) Suppose that there are exactly 2 keys that open the same lock, while the rest of the keys open exactly one of those $n$ locks each.

a) Give the best Big-O estimate for the worst-case number of steps required to open all the locks, given $m$ keys and $n$ locks. No witnesses are required, but briefly justify your answer (by showing your calculations).

**ANSWER**: First, note that $m = n + 1$. In a worst-case situation, we try lock #1, and go through all $m$ keys before finding the right key (assume the two duplicate keys are at the start):

| **Lock #** | **How long it takes to find the right key:** |
|---|---|
| *1* | $m$ |
| 2 | $m - 1$ |
| 3 | $m - 2$ |
| … | … |
| $m - 3$ | 4 |
| $m - 2$ | 3 (2 identical keys + 1 unique key) |
| $m - 1 = n$ | 1 (2 identical keys) |

Sum of RHS is given by $T(m) = 1 + 2 + \ldots + m$ steps $- 2$ steps
$= m(m + 1) / 2 - 2$
$= \frac{1}{2} m^2 + \frac{1}{2} m - 2$

Therefore, $T(m) \in O(m^2)$

Equivalently, since $m = n + 1$, we can write that $T(n) \in O(n^2)$ using similar reasoning.

(Don't worry about getting the exact number. Save that for another course.)

b) How would your analysis change, if we changed the rules a bit: this time, some of the $m$ keys open no lock at all (because they are for some other lock not found in this set of $n$ locks)?

**ANSWER**:

Suppose there are $k$ extra keys. Then, $m = n + k$.

In the simplest case, $k = 0$ and we still have quadratic time, as above: $T(m, n) \in O(m^2)$. In other words, we still have to go through the keys one-by-one with repeated unsuccessful matches.

Conversely, $k$ (and hence, $m$) could be an arbitrarily large number that dwarfs $n$. And, in other cases, we're somewhere in between. The bottom line is that we should include both $m$ and $n$ in our final expression. If we have $m$ = one billion keys and only $n = 10$ locks, then common sense says we make at most 10 * 1,000,000,000 tests, i.e., $nm$ tests $\in O(mn)$. This general form takes into account both the keys and the locks. Note that if $m$ and $n$ are equal, we still get the $m^2$ term.

Conclusion: $T(m, n) \in O(mn)$ … but for this course, if you argued that $T(m, n) \in O(m^2)$, that should be OK.

More formally, just in case you're interested …

| **Lock #** | **How long it takes to find the right key:** |
|---|---|
| 1 | $m$ |
| 2 | $m - 1$ |
| 3 | $m - 2$ |
| … | … |
| $m - k = n$ | $m - (m - k - 1) = k + 1$ |

Sum of RHS is given by $T(m, n)$

$$
\begin{aligned}
&= (1 + 2 + \ldots + m) - (1 + 2 + \ldots + k) \text{ steps} \\
&= m\,(m + 1) / 2 - [\,(m - n)\,(m - n + 1) / 2\,] \\
&= m\,(m + 1) / 2 - [\,m^2 - mn + m - mn + n^2 - n] / 2 \\
&= m\,(m + 1) / 2 - (\,\tfrac{1}{2} m^2 + \tfrac{1}{2} m - mn + \tfrac{1}{2} n^2 - \tfrac{1}{2} n) \\
&= \tfrac{1}{2} m^2 + \tfrac{1}{2} m - \tfrac{1}{2} m^2 - \tfrac{1}{2} m + mn - \tfrac{1}{2} n^2 + \tfrac{1}{2} n \\
&= mn - \tfrac{1}{2} n^2 + \tfrac{1}{2} n
\end{aligned}
$$

If $m = n$, then $T(m, n) = m^2 - \tfrac{1}{2} m^2 + \tfrac{1}{2} m$; *therefore,* $T(m) \in O(m^2)$.

Since, $m \geq n$ in the question, the $mn$ term tends to dominate the other terms, and we get:

$$T(m, n) = mn - \tfrac{1}{2} n^2 + \tfrac{1}{2} n$$

Therefore $T(m, n) \in O(mn)$

5. {5 marks} Write a recursive algorithm (using pseudo-code) to compute the product of 2 positive integers *m* and *n* using only addition.

**ANSWER**:

```
recursive_product(m,n)
{
     if n = 1
          return  m
     else
          return  recursive_product(m, n-1) + m
}
```

or, in C …

```
int recursive_product( int m, int n )
{
     if (n == 1)
          return  m;
     return  recursive_product(m, n-1) + m;
}
```

6. {10 marks} Suppose the nodes of a doubly-linked list structure are defined as follows:

```
struct node
{
  int           item;
  struct node *  next;
  struct node *  prev;
};
```

Define a function `add_ordered` that takes a new item and a list whose items are kept in increasing order and inserts the new item in the right place in the list (so that the items in the list continue to be in *increasing order* after the insertion).

For example, if `ls` is the list `(3,5,6,9),` then after calling "`add_ordered(&head_of_ls, 8);`", list `ls` becomes `(3,5,6,8,9).`

Note 1: The list may be empty before the first call.
Note 2: Use the function `make_node` to create a node. Its prototype is:

```
struct node * make_node(int new_item, struct node * next,
                                      struct node * prev);
```

**ANSWER**:

```
/* Let's assume that we pass in the address of the head of the
   list, so that *head is the actual start of the list.  We'll
   need to do this if we attempt to add to an empty list. */

void add_ordered( struct node * head, int key )
{
      struct node * curr  = *head;
      struct node * prev  = NULL;
      struct node * new_node;

      while ( curr  &&  curr->item <= key ) /* find right pos. */
      {
            prev = curr;
            curr = curr->next;
      }

      /* create and initialize a new node */
      new_node = make_node(key, NULL, NULL);
      if ( prev != NULL )              /* insert: general case */
      {
            prev->next = new_node;
            new_node->next = curr;
            new_node->prev = prev;
            curr->prev = new_node;
      }
      else                             /* insert at front */
      {
            new_node->next = *head;     /* old head is 2nd node */
            (*head)->prev = new_node;   /* new_node->prev is OK */
            *head = new_node;           /* we have a new head */
      }
}
```

7. {6 marks} Suppose the nodes of a binary tree structure are defined as follows:

```
struct node
{
  int           value;
  struct node *  left;
  struct node * right;
};
```

Define a recursive function `count` which takes as an argument a binary tree and returns the *number of nodes* that are in the tree. The function must leave its argument unchanged. Hint: The number of nodes in a tree with non-empty root *r* is 1 + the number of nodes in its left and right subtrees.

**ANSWER**:

```
int count( struct node * root )
{
     if ( root == NULL )
          return 0;

     return   1 + count(root->left) + count(root->right);
}
```

8. {8 marks} Suppose the nodes of a binary tree structure are defined as follows:

```
struct Bnode
{
  int            value;
  struct Bnode * left;
  struct Bnode *  right;
}
```

Define a recursive function `height` that takes the address of a node in a binary tree and returns the height of the node in that tree.

Recall: The height of a node in a tree is the length of the longest path from the node to a leaf. The height of a node without children is 0, and for this function (only), if the user passes in a null node, you can assume that its height is 0, too.

Hint: The height of a binary tree is largely determined by the height of its respective children.

Here's the function header to help you get started:

```
int height(struct Bnode * node)
{
```

**ANSWER**:

```
int height( struct Bnode * node)
{
  int  left_height, right_height;

  if ( node == NULL ||
         ( node->left == NULL && node->right == NULL ) )
                return  0;

  /* find the height of its children */
  left_height = height( node->left );
  right_height = height( node->right );
```

```
  /* The distance from the current node to its children
     is 1;  so, we know the height has to be (1 + the
     maximum height of its two children). */
  return  1 + max(left_height, right_height);
}
```