Week 1 Quiz

TOTAL POINTS 10

1.

```
int tri(int n) {
    int i,j;
    int count = 0;

    for (j=0; j < n; j++)
        for (i=0; i < j; i++)
            count++;

    return count;
}
```

1 point

Perform a run-time analysis of the code above. Express the number of times the variable *count* is incremented in terms of "Big Oh" notation.

Recall that "Big Oh" notation is denoted as O() where the parameter of O() is a simple function of n that indicates how the run-time increases as n increases. For example, if the run-time grows as a polynomial of n, such as "5n^3 + 3n^2" then the "Big-Oh" notation would ignore constants and lower growing terms and simply state O(n^3) growth.

- O(n^2)
- O(n^2 + n)
- O((1/2) n^2)
- O(1)

2. You have an array that is currently length one and already contains one item. You need to implement a function Append(i) that adds the item i to the position after the current last item of the array. If the array is full, then your Append() function will need to expand the size of the array so that it can store the additional item i. Recall that expanding the size of an array requires allocating new memory for the expanded size and copying all of the current array items to the new (expanded) array before de-allocating the previous (full) array.

1 point

(It is okay to assume there is always enough memory to allocate for an array.)

Your Append() function will be called an unknown number of times. Which method for resizing the array would result in the fastest total run-time for calling Append() n times to add n items to the array?

- Doubling the length of the array every time an item is added when the array is already full.
- Expanding the array to length n + 1 the first time Append() is called.
- Increasing the length of the array by one billion every time an item is added and the array is already full.
- Increasing the length of the array by one every time an item is added.

3. Which one statement below is FALSE? Assume we are using the most efficient algorithms discussed in lecture.

1 point

- Finding an item in a sorted linked list of n items takes O(n) time.
- Adding n items, one at a time, to the front of a linked list takes O(n) time overall.
- Finding an item in a sorted array of n items cannot be done in better than O(n) time.
- Adding n items, one at a time, to the end of an array takes O(n) time overall.

4. You have a list of 100 items that are not sorted by the item value. Which one task below would run much faster on a list implemented as an array rather than implemented as a linked list?

1 point

- Replacing the 25th item in the list with a different item.

- ○ Finding the first item.
- ○ Searching the list for all items that match a given item.
- ○ Inserting a new item between the 24th item and the 25th item.

5. You have a list of 100 items that are not sorted by the item value. Which one task below would run much faster on a list implemented as linked list rather than implemented as an array? 1 point

- ○ Finding the first item
- ○ Inserting a new item between the 24th item and the 25th item.
- ○ Searching the list for all items that match a given item.
- ○ Replacing the 25th item in the list with a different item.

6. Suppose you want to implement a queue ADT using a linked list. Your queue needs to be able to "push" (or "enqueue") a single item in constant time, as well as "pop" (or "dequeue") a single item in constant time. The operations need to happen at opposite ends of the queue, as would be expected of the queue ADT. However, the people who use your queue implementation don't need to know about how exactly it is implemented, so you can be somewhat creative in how you implement it, as long as the "push" and "pop" operations behave as expected. Which of the following implementations can accomplish this? Select all that apply. (For the sake of this question, let's not consider any design strategies that would close the linked list into a circle.) 1 point

- ☐ You can't implement a queue as a linked list. You need a more advanced data structure.
- ☐ You can do it with a modified singly-linked list where the list has both a "head" pointer and a "tail" pointer, but each node has only a "next" pointer.
- ☐ You can do it with a singly-linked list where the list has only a "head" pointer and each node has only a "next" pointer.
- ☐ You can do it with a doubly-linked list where the list has a "head" pointer and a "tail" pointer and each node has a "next" pointer and a "previous" pointer.

7. When implementing a queue which will need to support a large number of calls to its push() and pop() methods, which choice of data structure results in a faster run time according to "Big Oh" O() analysis? 1 point

- ○ A linked list because an array cannot be used to implement a queue that supports both push() and pop() methods.
- ○ Both array and linked list implementations of a queue have the same run time complexity.
- ○ The array implementation of a queue has a better run time complexity than does the linked list implementation.
- ○ The linked list implementation of a queue has a better run time complexity than does the array implementation.

8. When implementing a stack which will need to support a large number of calls to its push() and pop() methods, which choice of data structure results in a faster run time according to "Big Oh" O() analysis? 1 point

- ○ The linked list implementation of a stack has a better run time complexity than does the array implementation.
- ○ An array implementation because a linked list cannot be used to implement a stack that supports both push() and pop() methods.
- ○ The array implementation of a stack has a better run time complexity than does the linked list implementation.
- ○ Both array and linked list implementations of a stack have the same run time complexity.

9. Suppose this stack is implemented as a linked list. 1 point

```
std::stack<int> s;
s.push(1);
s.push(2);
s.push(3);
```

What is the value at the head of the linked list used to implement the stack s?

- ◯ 2
- ◯ 3
- ◯ 1
- ◯ A stack cannot be implemented using a linked list.

10. Suppose we had the following interface for a stack and queue, along with a correct implementation.

1 point

(Note that in this simple version, the "pop" and "dequeue" methods will remove an item and also return a copy of that same item by value. This is a little different from how the C++ Standard Template Library implementations of a stack and queue work. In STL, you have separate functions for peeking at the next value that would be removed, and for actually removing the item.)

```cpp
class Stack{
    public:
        Stack();
        bool push(int x);
        int pop();
        bool isEmpty();
    // other lines omitted
};

class Queue{
    public:
        Queue();
        bool enqueue(int x);
        int dequeue();
        bool isEmpty();
    // other lines omitted
};

```

What output does the following code produce?

```cpp
main() {
    Stack s = Stack();
    Queue q = Queue();

    for(int i = 0; i < 5; i++){
        s.push(i);
        q.enqueue(i);
    }

    for(int i=0; i < 5; i++){
        s.push(q.dequeue());
        q.enqueue(s.pop());
    }

    while (!q.isEmpty())
        std::cout << q.dequeue() << " ";
}

```

- ◯ 4 3 2 1 0 0 1 2 3 4
- ◯ 4 3 2 1 0
- [✓ 0 1 2 3 4]
- ◯ 0 1 2 3 4 4 3 2 1 0

☐ I, **Jiarong Yang**, understand that submitting work that isn't my own may result in permanent failure of this course or deactivation of my Coursera account.

Learn more about Coursera's Honor Code

Learn more about Coursera's Honor Code

Save

Submit