Congratulations! You passed!

Grade received 100% **Latest Submission Grade** 100% **To pass** 80% or higher

Go to next item

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;
}
```

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

✓ **Correct**
Even though the number of times the variable count is incremented is n*(n-1)/2 = (1/2) n^2 - (1/2) n, the "Big Oh" notation is only concerned about the order of the growth, which is the growth of the highest degree term (e.g. n^2) ignoring any constant factors of that term (e.g. 1/2).

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 / 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?

- 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.
- [✓ Doubling the length of the array every time an item is added when the array is already full.]

✓ **Correct**
Recall that this strategy requires O(n) operations instead of the others which require O(n^2) to add n items to the array.

3. Which one statement below is FALSE? Assume we are using the most efficient algorithms discussed in lecture. **1 / 1 point**

- ○ 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 linked list of n items takes O(n) time.
- ● 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.

✓ **Correct**
This is indeed the false statement, because If the array is sorted, then one can perform a binary search of the array by going to the middle of the array to see if that is the correct item. If not, then the correct item would be one one side or the other, such that only half of the array needs to be search. Hence the search space is cut in half each step and O(lg n) steps are needed. (Recall "lg n" is the base 2 logarithm of n, such that if x = lg n, then n = 2^x. If x is not an integer, then O(lg n) would refer to the ceiling of lg n.)

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 / 1 point**

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

✓ **Correct**
Once you have the 25th item in the list, replacing it with a different item takes just a single step for both an array and a linked list. However, the array can just compute the index (or address) of the 25th item in a single step with a simple formula, whereas a linked list would require 24 steps to follow 24 links past the first item at the head of the list.

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 / 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.

✓ **Correct**
Inserting a new item in the middle of the list implemented as an array requires copying each of the remaining items (25th to 100th) to new locations (26th to 101st), and may require resizing the array. Inserting a new item in the middle of the list implemented as a linked list would require setting the "next" pointer at the 24th item to the new item, and the "next" pointer at the current item to the (previously) 25th 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 / 1 point**

- [✓] 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.

✓ **Correct**

Yes, a full doubly-linked list implementation can do it, because the "head" and "tail" pointers give you O(1) access to both ends of the list, and the "next" and "previous" pointers also allow you to add or remove nodes at either end of the list easily in O(1). But there is also a less memory-intensive implementation that can be used here.

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

✓ **Correct**
Yes, if you choose to "push" (or "enqueue") incoming items at the tail end of the list, then you can use the "tail" pointer and the tail node's "next" pointer to append a new item to the tail in O(1). You can use the "head" pointer and the head node's "next" pointer to "pop" (or "dequeue") the existing head item in O(1) while updating the head pointer to point to the next item remaining after it in the list. But there is also another implementation option here that could be used.

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 / 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 linked list implementation of a queue has a better run time complexity than does the array implementation.
- ○ The array implementation of a queue has a better run time complexity than does the linked list implementation.

✓ **Correct**
The push() and pop() methods can be implemented in constant time O(1) for both an array and a linked list. Push and pop run in amortized constant time O(1) for an array, but the large number of calls to push() and pop() will ensure the cost of resizing the array is properly amortized by the benefits of the array.

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 / 1 point**

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

✓ **Correct**
The push() and pop() methods can be implemented in constant time O(1) for both an array and a linked list. Push and pop run in amortized constant time O(1) for an array, but the large number of calls to push() and pop() will ensure the cost of resizing the array is properly amortized by the benefits of the array.

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

```
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?

- ○ A stack cannot be implemented using a linked list.

- ○ 1
- ○ 2
- ● 3

Correct
When implementing a stack as a linked list, a call to push() method creates a new node, sets the *head* pointer of the linked list to this new node, and the new node's *next* pointer points to the previous head node of the linked list. Since this value is the last value pushed onto the stack, it indeed will be placed at the head of the list.

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

(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.)

```
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?

```
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() << " ";
}

```

- ○ 0 1 2 3 4 4 3 2 1 0
- ○ 4 3 2 1 0
- ● 0 1 2 3 4
- ○ 4 3 2 1 0 0 1 2 3 4

Correct