PROBLEM 1: (The Sorted Link (12 points))
Using the following definition of a linked list node, write a function that inserts a string into a sorted linked list.

```cpp
struct Node
{
    string info;
    Node * next;

    Node(const string& s, Node * ptr)
        : info(s), next(ptr)
        { }
};

Node* InsertSorted(Node *list, const string & word)
// pre: list (may be empty) is in sorted order
// post: list still sorted, new node with word added
{
    if (list == NULL)
        return new Node(word, NULL);
    if (word < list->info)
        return new Node(word, list);
    list->next = InsertSorted(list->next, word);
    return list;
}

// iteratively
Node* InsertSorted(Node *list, const string & word)
// pre: list (may be empty) is in sorted order
// post: list still sorted, new node with word added
{
    Node* first = list;
    // if new node is first, handle this case and return
    if (first == 0 || word < first->info)
        return new Node(word, first);

    // assert: word >= list->info;
    while (list->next != 0 && list->next->info < s)
        list = list->next;
}
```
// assert: s>= list->info and s < list->next->info
list->next = new Node(s, list->next);
return first;
}

PROBLEM 2:  (Piling it on (12 points))
Given the following interface to tstack:

```cpp
template <class Type>
class tstack
{
    public:
        tstack( ); // construct empty stack
        const Type & top( ) const; // return top element (NO pop)
        bool isEmpty( ) const; // return true if empty, else false
        void push( const Type & item ); // push item onto top of stack
        void pop( Type & item ); // pop top element into item
    }
};
```

Draw a box and pointer diagram of what list points to after executing the following code fragment:

```cpp
string *name = 0;
tstack<string*> s;
Node *list = 0;

s.push(new string("Nelson"));
s.push(new string("Malcolm"));
s.push(new string("Martin"));
s.push(new string("Marcus"));

*(s.top()) = "Patrice";
name = new string("");
while (!s.empty())
{
    s.pop(name);
    list = new Node(name, list);
}
```

PROBLEM 3:  (Pledge Oh Theta Omega! (18 points))
For the following, pick the correct relation \( O, \Omega \text{, or } \Theta \) and give justification in the form of correct constants for the definition of \( O(\cdot) \), \( \Omega(\cdot) \) or \( \Theta(\cdot) \).
For example:

\[ n \in n^n \]

Answer: \( O(n^n) \) for \( c = 1 \) and \( n_0 = 1 \). That is, \( n \leq 1 \cdot n^n \) for all \( n > 1 \).

a. (6 points) \( n^3 + 100n^2 \in O(n^4) \) with \( c = 1000 \)

b. (6 points) \( n + n \log n \in \Theta(n \log n) \)
   
   \( n + n \log n < 100n \log n \) and \( n \log n < 1 \cdot (n + n \log n) \), so it is both \( O \) and \( \Omega \) so \( \Theta \)

c. (6 points) \( 2^n \in O(3^n) \) with \( c = 1 \) and \( N_0 = 1 \)

**PROBLEM 4 : (The All Powerful Registrar (26 points))**

You are in charge of scheduling classes for the Love Auditorium. Your job is to schedule as many classes as possible. You are given a `vector` of `Interval`s. An `Interval` is a start and end time (in integer time units) described below:

```cpp
struct Interval
{
    int start; // begin time (in minutes after midnight)
    int end; // end time (in minutes after midnight)
    Interval(int a, int b)
        : start(a), end(b) // constructor
    { }
};
```

Luckily, your friend has taken CPS 100 and can describe an algorithm that yields the maximum number of intervals without conflicts in pseudo-code:

1. Sort all of the intervals in increasing order by `finish` time, breaking ties by length of the interval.
2. Loop starting from the first element until you reach the end of the vector
   - (3) Let \( x \) be the current item in the vector.
   - (4) Add \( x \) to your result vector
   - (5) Skip over all intervals that overlap with \( x \)

As an example, if the proposed class intervals were:

\[ (2, 5), (1, 3), (1, 2), (1, 4), (3, 5), (4, 5), (6, 7), (3, 4). \]

The sorting would yield:

\[ (1, 2), (1, 3), (3, 4), (1, 4), (4, 5), (3, 5), (2, 5), (6, 7) \]

and the best set of intervals would then be:

\[ (1, 2)(3, 4)(4, 5), (6, 7). \]

a. (10 points) In thinking about step (1), you reflect back on your vast experience with *Anagram*. You then realize that you can just use `QuickSort` to sort and define the less than operator \( (<) \). Define the \( < \) for the `Interval` struct for the requirements of the sort described in step (1).
bool operator < (const Interval& lhs, const Interval& rhs)
// post: return true if lhs < rhs
{
    if (lhs.end < rhs.end)
        return true;
    if (lhs.end == rhs.end)
        if (lhs.end - lhs.start < rhs.end - rhs.start)
            return true;
    return false;
}

b. (12 points) Now, it is time to sit down and code up the algorithm. Your friend was also kind enough to
give you a function that determines if two intervals overlap.

bool IntervalsOverlap(const Interval & a, const Interval & b)
// pre: a and b are both proper nonzero intervals where a.end > a.start
// post: returns whether a and b overlap
{
    if (a.begin >= b.end || b.begin >= a.end)
        return false;
    else
        return true;
}

She also started the scheduling function for you. Finish it so that it implements her pseudo-code.

void ScheduleClasses(tvector<Interval> classList,
                     tvector<Interval>& result)
// pre: classList contains legal intervals
    result is empty
// pos: result contains elements
{
    int i=0;
    Interval x;

    // Sort the list
    QuickSort(classList, classList.size());

    while (i < classList.size())
    {
        x = classList[i];
        // add x to the result list
        result.push_back(x);

        // move through the list
        i= i +1;
        while (i < classList.size() && IntervalsOverlap(x, classList[i]))
            i = i + 1;
    }
c. (4 points) What is the Big-Oh of ScheduleClasses in terms of \( n \) the number of intervals in classList?
Time for quicksort is \( O(n \lg n) \) and time for the while loop is \( O(n) \), so \( n \lg n + n \in O(n \lg n) \).

**PROBLEM 5**: *(Thoughts on vectors and lists (8 points))*

Describe an application where using a vector of linked lists would be more efficient than either using a vector or a linked list alone.

Lots of answers here. A good one from class is for reading in words. You could have a 26 element vector of linked lists. One for each letter in the alphabet. The individual lists would presumably be shorter, cutting down on search time.

**PROBLEM 6**: *(Cool Math (2 points EXTRA CREDIT))*

What is
\[
4 = \frac{2 \cdot 4 \cdot 6 \cdot 8 \cdots 2n}{3 \cdot 3 \cdot 5 \cdot 7 \cdots 2n - 1}
\]
as \( n \to \infty \)?

\( \pi \)

John Wallis (1616-1703) in around 1650 (he may have been predated or certainly at least contemporary with Japanese mathematicians who came up with the same idea) came up with the Wallis Formula which says that:
\[
\frac{\pi}{2} = \frac{2 \cdot 2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdots 2n}{3 \cdot 3 \cdot 5 \cdot 7 \cdots 2n - 1}
\]
It was remarkable in that it was the first infinite series for pi that did not involve irrational numbers and was computed before calculus. It involves computing the area under a quadrant of a circle.

**PROBLEM 7**: *(Summing Up (24 points))*

In this problem, you will analyze the big-Oh for 3 solutions to the Maximum Contiguous Subsequence Problem. Given possibly negative integers \( A_1, A_2, \ldots, A_n \), find the maximum value of \( \sum_{k=i}^{j} A_k \). The maximum contiguous subsequence would be zero (the empty sequence) if all integers all negative. For example, if the input is \( \{-2, 11, -4, -13, -5, 2\} \), then the answer is 20 for the subsequence in bold (elements 2-4). Your job is to analyze the three algorithms below, mark the line(s) of code that is executed the greatest number of times, and give the tight big-Oh bounds with justification. \( n \) is the number of elements in the vector \( a \).

a. (7 points) Analyze maxSubSum1

```cpp
int maxSubSum1(const tvector<int> & a )
{
    int maxSum = 0;
    for( int i = 0; i < a.size(); i++ )
        for( int j = i; j < a.size(); j++ )
        {
            int thisSum = 0;
            for( int k = i; k <= j; k++ )
                thisSum += a[ k ];          <-- most executed line
            if( thisSum > maxSum )
            {
```
```cpp
int maxSubSum2(const tvector<int> & a)
{
    int maxSum = 0;
    int thisSum = 0;
    for( int i = 0, j = 0; j < a.size(); j++ )
    {
        thisSum += a[j];  // most executed line
        if( thisSum > maxSum )
        {
            maxSum = thisSum;
            seqStart = i;
            seqEnd = j;
        }
        else if( thisSum < 0 )
        {
            i = j + 1;
            thisSum = 0;
        }
    }
    return maxSum;
}
```

\(O(n^3)\)

b. (7 points) Analyze \texttt{maxSubSum2}

```cpp
int maxSubSum2(const tvector<int> & a)
{
    int maxSum = 0;
    int thisSum = 0;
    for( int i = 0, j = 0; j < a.size(); j++ )
    {
        thisSum += a[j];  // most executed line
        if( thisSum > maxSum )
        {
            maxSum = thisSum;
            seqStart = i;
            seqEnd = j;
        }
        else if( thisSum < 0 )
        {
            i = j + 1;
            thisSum = 0;
        }
    }
    return maxSum;
}
```

\(O(n)\)

c. (10 points) Give the recurrence relation and big-Oh for \texttt{maxSubSum3}.

```cpp
/* Return maximum of three integers. */
int max3( int a, int b, int c )
{
    if (a > b)
        if (a > c)
            return a;
        else
            return c;
    else if (b > c)
        return b;
    else
        return c;
```
/* Finds maximum sum in subvector spanning a[left..right]. */
int maxSumRec(const tvector<int> & a, int left, int right)
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = (left + right) / 2;

    if (left == right) // Base case
        if (a[left] > 0)
            return a[left];
        else
            return 0;

    int maxLeftSum = maxSumRec(a, left, center);
    int maxRightSum = maxSumRec(a, center + 1, right);
    for (int i = center; i >= left; i--)
    {
        leftBorderSum += a[i]; // most executed
        if (leftBorderSum > maxLeftBorderSum)
            maxLeftBorderSum = leftBorderSum;
    }

    for (int i = center + 1; i <= right; i++)
    {
        rightBorderSum += a[i]; // most executed
        if (rightBorderSum > maxRightBorderSum)
            maxRightBorderSum = rightBorderSum;
    }

    return max3(maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum);
}

int maxSubSum3(const tvector<int> & a)
{
    return maxSumRec(a, 0, a.size() - 1);
}

\[ T(n) = 2T(n/2) + nT(1) = 12T(n/2) + n \rightarrow O(n \log n) \]