PROBLEM 1:  \(N\)-List (12 points)\)
A \(N\)-list stores integers, the integers are stored in non-decreasing order, each integer \(k\) occurs exactly \(k\) times and \(N\) is the largest integer in the list. For example, a 4-list is \((1,2,2,3,3,3,4,4,4,4)\) and a 5-list is a 4-list with \((5,5,5,5,5)\) appended to the end.

a. Using the standard \texttt{Node} definition listed at the back of the test and replacing \texttt{strings} with \texttt{ints}, write \texttt{MakeNList}.

\[
\text{Node* MakeNList(int n)}
\]
\[
// \text{precondition: } n > 0
\]
\[
// \text{postcondition: returns the } N\text{-List}
\]
\[
\{
\text{Node *list = 0;}
\text{for (int i = n; i >= 1; i--)}
\text{for (int j = 0; j < i; j++)}
\text{list = new Node(i, list);}
\text{return list;}
\}
\]

b. What is the Big-Oh of MakeNList?

Look at how many times \texttt{list = new Node(i, list);} is executed.
\[
n + n - 1 + \cdots + 1 = n(n + 1)/2 \in O(n^2)
\]

PROBLEM 2:  Analysis (9 points)\)

a. What is the big-Oh of the following algorithm? Briefly explain why.

\[
\text{bool IsPrime(int n)}
\]
\[
\{
\text{int i = 3;}
\text{if (n == 2 || n == 3)}
\text{return true;}
\text{if (n % 2 == 0)}
\text{return false;}
\text{while (i * i <= N)}
\{
\text{if (n % i == 0)}
\}
```cpp
return false;
i = i + 2;
}
return true;
}
O(\sqrt{n})
b. Suppose \( T_1(n) \in O(f(n)) \) and \( T_2(n) \in O(f(n)) \). Which of the following are true:
\( T_1(n) + T_2(n) \in O(f(n)) \)
c. Which of the following functions grows fastest as \( n \) grows large:
\( 2^n \)
d. Which of the following functions grows fastest as \( n \) grows large:
\( n \log n \)

PROBLEM 3: (IsBalanced (12 points))

In the syntax of most programming languages, there are some characters that occur only in nested pairs, which are called bracketing operators. C++ has the following bracketing operators:

\( ( \ldots ) \)
\[ \ldots ]
\{ \ldots \}

In a properly formed program, these characters will be properly nested and matched. To determine whether this condition holds for a particular program, you can ignore all the other characters and look simply at the pattern formed by the parentheses, brackets, and braces. In a legal configuration, all the operators match up correctly, as shown in the following example:

\( \{ x = (s = v[1] + 2); y = 4*(v[v.size() - 1] + x ); \} \)

The following configurations are illegal for the reasons stated:

- ( ( [ 4 ] ) The line is missing a close parenthesis.
- AB) ( The close parenthesis comes before the open parenthesis.
- ( ( x} ) The parentheses and curly braces are improperly nested.

For this problem, your task is to write a function

```cpp
bool isBalanced(string s)
```

that takes a string \( s \) with all characters except the bracketing operators removed. The method should return `true` if the bracketing operators in \( s \) are balanced, which means they are correctly nested and aligned, otherwise it should return `false`.

You must either use a `tstack` or recursion in your solution.

Assume you have the following helper functions.
bool IsOpener(char ch); // returns true if ch is ( { or [, else false
bool IsCloser(char ch); // returns true if ch is ) } or ], else false
char MatchingChar(char ch); // returns matching char.  
// In other words, returns { for },  
// } for {, ] for [, and so on  
// Returns ' ' if ch is not opener  
// or closer

// stack version
static boolean isBalanced(string s) {
    tstack<char> brackets;
    for (int i=0; i < s.length(); i++) {
        char curr = s[i];
        if (IsOpener(curr))
            brackets.push(curr);
        else if (IsCloser(curr))
        {
            if (brackets.isEmpty() ||
                MatchingChar(curr) != brackets.top())
                return false;
            brackets.pop();
        }
    }
    return brackets.IsEmpty();
}

// recursive version
// helper function that returns new string with only brackets
string JustBrackets(string s) {
    string result = "";
    for (int i = 0; i < s.length(); i++)
        if (IsOpener(s[i]) || IsCloser(s[i]))
            result = result + s[i];
    return result;
}

boolean isBalancedRecurr(string s)
// pre: s only consists of the bracketing characters ([]{}))
// post: returns whether brackets are balanced
{
    int pos;
    // base case
    if (s.length() == 0) return true;
    if ((pos = s.find("()")) == string::npos &&
        (pos = s.find("[]")) == string::npos &&
        (pos = s.find("{}")) == string::npos)
        return false;
    return isBalanced(s.substr(0,pos) + s.substr(pos+2,s.length()));
}
bool isBalanced(string s) {
    return isBalancedRecur(JustBrackets(s));
}

PROBLEM 4:  (No Curve (18 points))
Consider a sorted Queue of elements - front is the minimum element, back is the maximum - implemented
with a doubly linked list:

class SortedQueue
{
    public:

    // constructors/destructor
    tqueue( ) // construct empty queue
        : front(0), back(0), mysize(0)
    {
    }

    // accessors
    const string & front( ) const; // return front (no dequeue)
    bool isEmpty( ) const; // return true if empty else false
    int size( ) const; // return number of elements in queue

    // modifiers
    void enqueue( const string & item ); // insert item (at back)
    void dequeue( ); // remove first element
    void dequeue( string & item ); // combine front and dequeue
    void makeEmpty( ); // make queue empty
    void clear( ); // same as makeEmpty

    private:
        struct QNode
        {
            string info;
            QNode * next; // next node
            QNode * prev; // previous node

            QNode(const string & val, QNode* before, QNode* after)
                : info(val), next(before), prev(after)
            {
            }
        };
        QNode *front;
        QNode *back;
        int mySize;
}
a. Write **enqueue** so that it satisfies its postcondition. You can write helper functions for the insertion if you like

```cpp
void SortedQueue::enqueue(const string & item)
// precondition: queue is [e1, e2, ..., en] with n >= 0
// ei <= ej if i < j
// postcondition: queue is [e1, e2, ..., en+1] with item now
// inserted in sorted order
{

    // NULL case
    if (front == 0)
        front = back = new QNode(item, 0, 0);

    // at front
    else if (item < front->info)
        { 
            front = new QNode(item, front, 0);
            front->next->prev = front;
        }
    // at back
    else if (item > back->info)
        { 
            back = new QNode(item, 0, back);
            back->prev->next = back;
        }
    // in the middle
    else
        { 
            Node *curr = front->next;
            while (item > curr->info)
                { 
                    curr = curr->next;
                }
            // item should be placed BEFORE curr
            Node *add = new QNode(item, curr, curr->prev);
            add->prev->next = add;
            add->next->prev = add;
        }
    mySize += 1;
}
```
b. Write a function that returns the median element of the queue. The median element is the $n/2$th largest element. For a queue of 5 elements, the median should return the 3 largest element. For a queue of 6 element, the median should return either the 3rd or 4th largest element.

```cpp
string MedianHelper(QNode* front, QNode *back)
{
    if (front == 0)
        return "";
    else if (front == back || back == 0)
        return front->info;
    else
        return MedianHelper(front->next, back->prev);
}
string SortedQueue::median()
// precondition: queue has 0 or more elements
// postcondition: returns the n/2th largest element
// queue is not changed
{
    return MedianHelper(front, back);
}

// Iterative version
string SortedQueue::median()
// precondition: queue has 0 or more elements
// postcondition: returns the n/2th largest element
// queue is not changed
{
    Node * curr = front;
    if (mySize == 0 || front == 0)
        return "";
    for (int i = 0; i < mySize/2; i++)
    {
        curr = curr->next;
    }
    return curr->info;
}
PROBLEM 5:  (Hashing (12 points))

Suppose you have a hash map where the keys are ints with 10 buckets and the hash function, \( h(k) = k \mod 10 \). Given the following order of input keys:

\[
4371 \ 1323 \ 6173 \ 4199 \ 4344 \ 9679 \ 1989
\]

A. Show the resulting hash map where collisions are resolved using linear probing.

B. Show the resulting hash map where collisions are resolved using quadratic probing.

C. Show the resulting hash map where collisions are resolved using chaining.
PROBLEM 6: (Just like crunch (10 points))

Write a function that removes all zeroes from a vector. Your function must operate in $O(n)$ time where $n$ is the size of the vector and cannot create a new vector.

```cpp
global int lastZeroIndex = -1;
global void RemoveZeroes(tvector<int> v)
    // pre: vector is of size >= 0
    // post: vector elements in same order but all zero elements have been removed and vector has been resized
    {
        int lastZeroIndex = -1;
        for (int k = 0; k < v.size(); k++)
        {
            if (v[k] != 0)
            {
                lastZeroIndex += 1;
                v[lastZeroIndex] = v[k];
            }
        }
        list.resize(lastZeroIndex+1);
    }
```