Jonathan Tso
Homework 01
CS 251

Question 1:

```
bool has_dups(NODE *lst) {

    //if list does not exist or only has 1, it has no duplicates and stops fx here
    if (lst == nullptr || lst->next == nullptr) {
        return false;
    }

    //gets the first value in the linked list
    int comparedToVal = lst->val;

    //creates a comparing node so that we can shift it each time it is incorrect
    NODE *compareNODE = lst->next;
    int comparisonVal = compareNODE->val;

    // if the values number isn't equal to the compared number, it's not a dupe
    while (comparedToVal != comparisonVal) {
        //if the comparing node isn't the end of list, update compare node to be next
        //and then update comparisonVal
        if (compareNODE->next != nullptr) {
            compareNODE = compareNODE->next;
            comparisonVal = compareNODE->val;
        }
        //compareNODE is at end of list. then we try to update lst
        //and then comparedToVal
        else if (lst->next != nullptr) {
            lst = lst->next;
            comparedToVal = lst->val;
            if (lst->next != nullptr) {
                compareNODE  = lst->next;
            }
            else {
                return false;
            }
        }
    }

    return true;
}
```

Question 2:

```
int squaredRecursive(unsigned int n) {
    int squaredValue = 0;

    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return n;
    }
    else {
        squaredValue = squaredRecursive(n-1) + (n-1) + (n-1) + 1;
        return squaredValue;
    }
}
/*  the squaredRecursive function will intake a number n and return the squared value.
    the logic behind the function is first that if n == 0, the squared value is 0.
    if n == 1, the squared value is 1.
    for any remaining one, we can note that the squared value is actually a result of:
        previous square + the nth value * 2 + 1. for example:
            2^2 = 1 (1 squared) + 1 + 1 + 1
            3^2 = 4 (2 squared) + 2 + 2 + 1
            4^2 = 9 (3 squared) + 3 + 3 + 1
        and so forth. this is under the logic that (n+1)^2 = n^2 + n + n + 1
    the function has solution = previous (squaredRecursive(n-1) + 2 * (the last n (n-1)) +1
*/
```

Question 3:

```
/*  A.
        The initial attempt of clone_array is faulty in that it creates the array b[n]
        within the function and the memory is allocated on the stack. This means that
        the array is given memory upon function call, but once the function is done,
        the memory is allowed to be overwritten. Thus, the array b[n] may potentially
        give us incorrect information when calling on it.*/

    B.
        int main() {

            int data[] = {1,2};
            int *clone;

            clone = clone_array(data,2);

            for (int i=0; i < 2; i++) {
                cout << clone[i] << endl;
            }

            return 0;
        }
```

```
[MacBook-Pro:cs251 jonathantso$ g++ HW01.cpp
HW01.cpp:93:10: warning: address of stack memory associated with local variable
      'b' returned [-Wreturn-stack-address]
  return b;
         ^
1 warning generated.
[MacBook-Pro:cs251 jonathantso$ ./a.out
1
32767
```

```
    C.
        int * clone_array(int a[], int n) {

        int * b;
        b = new int[n];
        int i;

          for(i=0; i<n; i++) {
              b[i] = a[i];
          }
          return b;
        }
/*
        The revised version above creates a pointer b. The pointer b then is set to point
        to an array of int[2] that is created on the heap via new.
*/
```

Question 4:

HW Ø1

4. A.

n	# of ticks
Ø	0
1	1+1 = 2
2	2+4 = 6
3	3+9 = 12
4	4+16 = 20

B. For all $n \geq Ø$, calling fubar(n) will result in $n + n^2$ ticks being printed.

Justification: The first part will call a tick for every n times as long as $n > Ø$. This $= n$

The second part, which has the nested for loop, will tick n times for every n^{th} instance. This $= n^2$

The total of these ticks becomes $n + n^2$.

A.

n	# of ticks

Question 5:

5 A.

n	# of ticks
0	1
1	3
2	7
3	15
4	31

B. For all $n \geq 0$, calling foo(n) results in $2^{n+1} - 1$ ticks being printed.

C. Base Case: $n = 1$

Function: $2^{n+1} - 1 = 2^{2} - 1 = 3$ Proven base case

Suppose that for some value $K \geq 1$, that $F(K) = 2^{K+1} - 1$

Need to prove: $F(K+1) = 2^{K+1+1} - 1$ for number of ticks

$F(K+1) = 1 + \underbrace{F(K)}_{K+1} + \underbrace{F(K)}_{K+1}$ By the code

$= 1 + (2^{K+1} - 1) + (2^{K+1} - 1)$ By induction

$= 2^{K+1} + 2^{K+1} - 1$

$= 2^{K+1+1} - 1 = 2^{K+2} - 1$

Thus, we have proven by induction that $F(K) = 2^{K+1} - 1$

Question 6:

```
/*  A.

    The function to see if the sudoku row is ok is faulty because it does not check
    if there are duplicates inside the row array. For instance, the following array:
    {1,2,3,4,5,5,8,8,9} will also yield a sum = 45, but this is not a valid sudoku row.

B.
*/
bool sudoku_row_ok(int row[]) {
    int sum = 0;
    int i, j;
    int max_row_size = 9;

    for (i=0; i<9; i++) {
        if(row[i] < 1 || row[i] > 9) {
            return false;
        }

        for (int j=i+1; j<max_row_size; j++) {
            if (row[i] == row[j]) {
                return false;
            }
        }

        sum += row[i];
    }

    if (sum == 45) {
        return true;
    }
    else {
        return false;
    }
}
```