CS 32 Worksheet 3

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.
Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

If you have any questions or concerns please contact apoorvapanse@ucla.edu or go to any of the LA office hours.

Concepts

Recursion

Problems

1. What does the following code outputs and what does the function LA_power do?

```
#include <iostream>
using namespace std;

int LA_power(int a, int b)
{
   if (b == 0)
       return 0;
   if (b % 2 == 0)
       return LA_power(a+a, b/2);

   return LA_power(a+a, b/2) + a;
}

int main()
{
  cout << LA_power(3, 4);
  return 0;
}
```

It outputs 12. LA_power returns the result of multiplying its arguments.

2. Given a singly-linked list LL with a member variable *head* that points to the first *Node* struct in the list, write a function to recursively delete the whole list.

```
void LL::deleteList();
```

```
void deleteListHelper(Node* head) {
  if (head == nullptr)
    return;
  deleteListHelper(head->next);
  delete head;
}

void LL::deleteList() {
  deleteListHelper(head);
}
```

3. Implement a recursive function that returns the maximum value in an array of size n. The header of the function is given: `int fun(int a[],int n)`

```
int fun(int a[], int n) {
     if (n == 1)
          return a[0];
     int x = fun(a, n-1);
     if (x > a[n-1])
          return x;
     else
          return a[n-1];
}
```

4. Given a string *str*, recursively compute a new string such that all the ‘x’ chars have been moved to the end.

```
string endX(string str);
```

```
Example:
endX("xrxe") → "rexx"
```

```
string endX(string str) {
  if (str.length() <= 1)
    return str;
  else if (str[0] == 'x')
    return endX(str.substr(1)) + 'x';
  else
```

```
    return str[0] + endX(str.substr(1));
}
```

5. Implement the following function in a recursive fashion:

   ```
   bool isSolvable(int x, int y, int c);
   ```

   This function should return true if there exists nonnegative integers *a* and *b* such that the equation *ax* + *by* = *c* holds true. It should return false otherwise.

   Ex: `isSolvable(7, 5, 45) == true //a == 5 and b == 2`
   Ex: `isSolvable(1, 3, 40) == true //a == 40 and b == 0`
   Ex: `isSolvable(9, 23, 112) == false`

   ```
   bool isSolvable(int x, int y, int c) {
     if (c == 0)
       return true;
     if (c < 0)
       return false;

     return isSolvable(x, y, c - x) || isSolvable(x, y, c - y);
   }
   ```

6. A robot you have programmed is attempting to climb a flight of stairs, for which each step has an associated number. This number represents the size of a leap that the robot is allowed to take backwards or forwards from that step (the robot, due to your engineering prowess, has the capability of leaping arbitrarily far). The robot must leap this exact number of steps.

   Unfortunately, some of the steps are traps, and are associated with the number 0; if the robot lands on these steps, it can no longer progress. Instead of directly attempting to reach the end of the stairs, the robot has decided to first determine if the stairs are climbable. It wishes to achieve this with the following function:

   ```
   bool isClimbable(int stairs[], int length);
   ```

   This function takes as input an array of int that represents the stairs (the robot starts at position 0), as well as the length of the array. It should return true if a path exists for the robot to reach the end of the stairs, and false otherwise.

   Ex: `isClimbable({2, 0, 3, 0, 0}, 5) == true`
   ```
        //stairs[0]->stairs[2]->End
   ```

Ex: `isClimbable({1, 2, 4, 1, 0, 0}, 6) == true`
`//stairs[0]->stairs[1]->stairs[3]->stairs[2]->End`
Ex: `isClimbable({4, 0, 0, 1, 2, 1, 1, 0}, 8) == false`

```
bool isClimbableHelper(int stairs[], bool visited[], int
length, int pos) {
  if (pos < 0)
    return false;
  if (pos >= length)
    return true;

  if (stairs[pos] == 0 || visited[pos])
    return false;
  visited[pos] = true;

  return isClimbableHelper(stairs, visited, length, pos -
stairs[pos]) || isClimbableHelper(stairs, visited, length, pos
+ stairs[pos]);
}

bool isClimbable(int stairs[], int length) {
  if (length < 0)
    return false;

  bool* visited = new bool[length];
  for (int x = 0; x < length; x++)
    visited[x] = false;

  bool res = isClimbableHelper(stairs, visited, length, 0);
  delete[] visited;
  return res;
}
```

7. Implement the function `sumOfDigits` recursively. The function should return the sum of all of the digits in a *positive* integer.

```
int sumOfDigits(int num);

sumOfDigits(176); // should return 14
sumOfDigits(111111); // should return 6
```

```
int sumOfDigits(int num) {
     if (num < 10)
```

```
            return num;
        return num % 10 + sumOfDigits(num/10);
    }
```

8. Implement the function `isPalindrome` recursively. The function should return whether the given string is a palindrome. A palindrome is described as a word, phrase or sequence of characters that reads the same forward and backwards.

```
bool isPalindrome(string foo);

isPalindrome("kayak"); // true
isPalindrome("stanley yelnats"); // true
isPalindrome("LAs rock"); // false (but the sentiment is true
:))

bool isPalindrome(string foo) {
        int len = foo.length();
        if (len <= 1)
                return true;
        if (foo[0] != foo[len-1])
                return false;
        return isPalindrome(foo.substr(1, len-2));
}
```

9. Write the following linked list functions recursively.

```
// inserts a value in a sorted list, returns a head to new list
// before: head → 1 → 3 → 5 → 7 → 15
// insertInOrder(head, 8);
// after: head → 1 → 3 → 5 → 7 → 8 → 15
Node* insertInOrder(Node* head, int value) {
        if (head == nullptr || value < head->data) {
                Node* p = new Node;
                p->data = value;
                p->next = head;
                head = p;
        } else
                head->next = insertInOrder(head->next, value);
        return head;
}
// deletes all instances of value in a linked list
Node* delete(Node* head, int value) {
```

```
    if (head == nullptr)
        return nullptr;
    else {
        if (head->data == value)
            return deleteAll(head->next, value);
        else {
            head->next = deleteAll(head->next, value);
            return head;
        }
    }
}

// prints the values of a linked list backwards
// head → 0 → 2 → 1 → 4 → 1 → 7
// reversePrint(head) will output 7 1 4 1 2 0
void reversePrint(Node* head) {
    if (head == nullptr)
        return;
    reversePrint(head->next);
    cout << head->data << endl;
}
```

10. Write a recursive function `isPrime` to determine whether a given integer input is a prime number or not.

    Example:

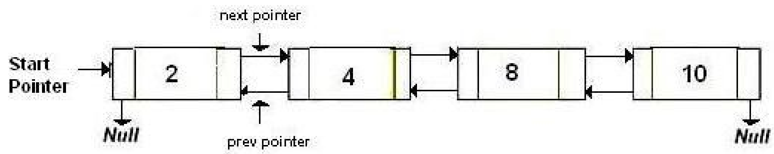
    isPrime(11) → true
    isPrime(4) → false

```
// There is a trick here. We notice that without a secondary
// parameter to keep count of where we are in our check, this
// problem is impossible recursively. Thus the solution can be
// done with either a default parameter or with an auxiliary
// helper function. This solution utilizes the former.
bool isPrime(int num, int i = 2) {
    if (num <= 2)
        return num == 2;
    // Divisible therefore not prime
    if (num % i == 0)
        return false;
    // Exhausted all possible divisors
    if (i*i > num)
        return true;
```

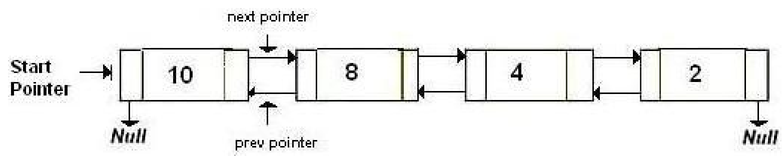
```
        return isPrime(num, i + 1);
}
```

11. Write a recursive function to reverse a doubly linked list. Assume the linked list has all its functions implemented. The integer value in each node must not be changed (but of course the pointers can be).

    Example:
    Original:



    After:




```
// Struct for Node that linked list is made from
struct Node {
        int val;
        Node* next, prev;
};

// Recursively reverse the doubly linked list
void reverse(Node* head) {
        if (node == nullptr)
                return;
        // Swap next and prev
        Node* temp = head->next;
        head->next = head->prev;
        head->prev = temp;
        // If previous is null then we are done
        if (head->prev == nullptr)
                return;
        reverse(head->prev);
}
```

12. Implement the following recursive function:

```
string longestCommonSubsequence(string s1, string s2);
```

The function should return the longest common subsequence of characters between the two strings s1 and s2. Basically, it should return a string of characters that are common to both strings and are in the same order in both strings. Example:

```
string res = longestCommonSubsequence("smallberg",
"nachenberg");

//res should contain "aberg" as seen in the green chars

res = longestCommonSubsequence("los angeles", "computers");

//res should contain the string "oes"
```

```
string lcs(string s1, string s2) {
    if (s1 == "") // base case: s1 is empty
        return "";
    else if (s2 == "") // base case: s2 is empty
        return "";
    else {
        // split the strings into head and tail for simplicity
        char s1_head = s1[0];
        string s1_tail = s1.substr(1);
        char s2_head = s2[0];
        string s2_tail = s2.substr(1);

        // if heads are equal, use the head and
        // recursively find rest of common subsequence
        if (s1_head == s2_head) {
            return s1_head + lcs(s1_tail, s2_tail);
        }
        // otherwise, check the common subsequences from
        // not including one of the heads
        else {
            string move_s1 = lcs(s1_tail, s2);
            string move_s2 = lcs(s1, s2_tail);

            // return the longer of the subsequences we found
            if (move_s1.length() >= move_s2.length())
                return move_s1;
            else
                return move_s2;
```

```
        }
    }
}
```

13. Implement a recursive function that merges two sorted linked lists into a single sorted linked list. The lists are singly linked; the last node in a list has a null next pointer. The function should return the head of the merged linked list. No new Nodes should be created while merging. Example:

    List 1 = 1 -> 4 -> 6 -> 8
    List 2 = 3 ->9 -> 10
    After merge: 1 -> 3 -> 4 -> 6 -> 8 -> 9 -> 10

    Use the following definition of a Node of a linked list:

    ```
    struct Node {
          int val;
          Node* next;
    };
    ```

    Use the following function header to get started:

    ```
    Node* merge(Node* l1, Node* l2);
    ```

    ```
    Node* merge(Node* l1, Node* l2)
    {
    // base cases: if a list is empty, return the other list
          if (l1 == nullptr)
                return l2;
          if (l2 == nullptr)
                return l1;

    // determine which head should be the head of the merged list
    // then set the next pointer to the head of the recursive calls
          Node* head;
          if (l1->val < l2->val) {
                head = l1;
                head->next = merge(l1->next, l2);
          }
          else {
                head = l2;
                head->next = merge(l1, l2->next);
          }
    ```

```
// return the head of the merged list
     return head;
}
```

14. Rewrite the following function recursively. You can add new parameters and completely change the function implementation, but you can't use loops.

    This function sums the numbers of an array from right to left until the sum exceeds some threshold. At that point, the function returns the running sum. Returns -1 if the threshold is not exceeded before the end of the array is reached.

```
int sumOverThreshold(int x[], int length, int threshold) {
  int sum = 0;
  for(int i = 0; i < length; i++) {
    sum += x[i];
    if (sum > threshold) {
      return sum;
    }
  }

  return -1;
}

int sumOverThreshold2(int x[], int length, int threshold, int
sum) {
  if (sum > threshold) {
    return sum;
  }

  if (length == 0) {
    return -1;
  }

  return sumOverThreshold2(x+1, length-1, threshold, sum+x[0]);
}
/******************
OR
******************/
int sumOverThreshold3(int x[], int length, int threshold) {
  if(threshold < 0){
    return 0;
```

```
  } else if (length == 0) {
    return -1;
  }

  int returnOfRest = sumOverThreshold3(x+1, length-1,
threshold-x[0]);
  if (returnOfRest == -1){
    return -1;
  } else {
    return x[0] + returnOfRest;
  }
}
```

15. Given the following program, give the output of each function call for parts a, b, and, c.

```
void fizzbuzz(int x){
  if (x == 0) {
    cout << "fizzbuzz" << endl;
    return;
  }

  cout << "fizz" << endl;
  fizzbuzz(x-1);
  fizzbuzz(x-1);
}
```

a. fizzbuzz(1)

fizz
fizzbuzz
fizzbuzz

b. fizzbuzz(2)

fizz
fizz
fizzbuzz
fizzbuzz
fizz
fizzbuzz
fizzbuzz

c. fizzbuzz(3)

fizz

fizz
fizz
fizzbuzz
fizzbuzz
fizz
fizzbuzz
fizzbuzz
fizz
fizz
fizzbuzz
fizzbuzz
fizz
fizzbuzz
fizzbuzz

i.e.,

fizz
what fizzbuzz(2) writes
what fizzbuzz(2) writes