# CS 32 Solutions Week 6

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 go to any of the LA office hours.

## Concepts

**Recursion, Templates, STL**

## Recursion Problems (some problems may require a helper function)

1) Implement the function `getMax` recursively. The function returns the maximum value in `a`, an integer array of size `n`. You may assume that `n` will be at least 1.

```
int getMax(int a[], int n) {
      if (n == 1)
            return a[0];        // base case: array is 1 element
      int x = getMax(a, n-1); // reduce: getMax of n-1 elems
      if (x > a[n-1])         // compare max to last element
            return x;
      else
            return a[n-1];
}
```

2) 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 left to right 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){
     return sumOverThreshold2Helper(x, length, threshold, 0);
}

//sum argument holds the running sum so far before looking at
//current element
int sumOverThreshold2Helper(int x[], int length, int threshold,
                          int sum)
{
  if (sum > threshold) {
    return sum;
  }
  if (length == 0) { //base case: end of array reached
    return -1;
  }
  return sumOverThreshold2Helper(x+1, length-1, threshold,
                               sum+x[0]);
} // reduce by adding first element of array to sum

/******************
OR
******************/
int sumOverThreshold3(int x[], int length, int threshold) {
  if(threshold < 0){
    return 0;
  } else if (length == 0) { //base case: end of array reached
    return -1;
  }

  int returnOfRest = sumOverThreshold3(x+1, length-1,
     threshold-x[0]);
  if (returnOfRest == -1){
    return -1;
  } else {
    return x[0] + returnOfRest;
  }
}
// reduce: find sum over new threshold t - x[0] in x after x[0]
```

3) 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);
   ```

   Hint: [https://www.cplusplus.com/reference/string/string/substr/](https://www.cplusplus.com/reference/string/string/substr/)

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

   ```
   string endX(string str) {
     if (str.length() <= 1)   // base case: no x's to shift to end
       return str;
     if (str[0] == 'x')
       return endX(str.substr(1)) + 'x';   // reduce: handle
   str[0]
                                          // first
     else
       return str[0] + endX(str.substr(1));
   }
   ```

4) 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);
}

// reduce: slowly take out x to check (a-1)x + by = c - x
        or slowly take out y to check ax + (b-1)y = c - y
```

## Template/STL Problems

1) What is the output of this program?

```
template <class T>
void foo(T input) {
      cout << "Inside the main template foo(): " <<input<< endl;
}

template<>
void foo(int input) {
      cout << "Specialized template for int: " << input << endl;
}

int main() {
      foo<char>('A');
      foo<int>(19);
      foo<double>(19.97);
}
```

```
Inside the main template foo(): A
Specialized template for int: 19
Inside the main template foo(): 19.97
```

2) The following code has 3 errors that cause either runtime or compile time errors. Find and fix all of the errors.

```
class Potato {
public:
  Potato(int in_size) : size(in_size) { };
  int getSize() const {
    return size;
  };
private:
  int size;
};

int main() {
  vector<Potato> potatoes;
  Potato p1(3);
  potatoes.push_back(p1);
  potatoes.push_back(Potato(4));
  potatoes.push_back(Potato(5));
  vector<~~int~~ Potato>::iterator it = potatoes.begin(); // 1
  while (it != potatoes.end()) {
    it = potatoes.erase(it);  // 2
    ~~it++;~~
  }

  for (it = potatoes.begin(); it != potatoes.end(); it++) {
    cout << it->getSize() << endl;  // 3
  }
}
```

1: potatoes.begin() gives iterator of potatoes, which is a vector<Potato>, so the iterator given will be of the type vector<Potato>::iterator

2: After calling erase with the iterator it, it is invalidated. Instead of incrementing it, the return value of potatoes.erase(it) should be assigned to it. The erase method returns the iterator of the element that is after the erased element.

3: Iterators use pointer syntax, so the last for loop should use it->getSize() instead of it.getSize().

3) Implement a stack class *Stack* that can be used with any data type using templates. Use a linked list (not an STL `list`) to store the stack and implement the functions *push(), pop(), top(), isEmpty(),* a default constructor, and a destructor that deletes the linked list nodes.

```
template<typename Item>
class Stack {
 public:
  Stack() : m_head(nullptr) {}

  bool isEmpty() const {
    return m_head == nullptr;
  }

  Item top() const {
    // We'll return a default-valued Item if the Stack is
empty,
    // because you should always check if it's empty before
    // calling top().
    if (m_head != nullptr)
      return m_head->val;
    else
      return Item();

  }

  void push(Item item) {
    Node* new_node = new Node;
    new_node->val = item;
    new_node->next = m_head;
    m_head = new_node;
  }

  void pop() {
    // We'll simply do nothing if the Stack is already empty,
    // because you should always check if it's empty while
    // popping.
    if (m_head == nullptr) {
      return;
    }
    Node* temp = m_head;
    m_head = m_head->next;
    delete temp;
  }

  ~Stack() {
    while (m_head != nullptr) {
      Node* temp = m_head;
      m_head = m_head->next;
      delete temp;
    }
 }

 private:
  struct Node {
    Item val;
    Node* next;
  };
  Node* m_head;
};
```

4) Implement a vector class *Vector* that can be used with any data type using templates. Use a dynamically allocated array to store the data. Implement only the *push_back()* function, default constructor, and destructor.

```
template <typename T>
class Vector {
  public:
    Vector();
    ~Vector();
    void push_back(const T& item);
  private:
    // Total capacity of the vector -- doubles each time
    int m_capacity;
    // The number of elements in the array
    int m_size;
    // Underlying dynamic array
    T* m_buffer;
};

template <typename T>
Vector<T>::Vector()
: m_capacity(0), m_size(0), m_buffer(nullptr)
{}

template <typename T>
Vector<T>::~Vector() {
     delete[] m_buffer;
}

template <typename T>
void Vector<T>::push_back(const T& item) {
  // if space is full, allocate more capacity
  if (m_size == m_capacity)
  {
    // double capacity(doesn't have to be doubled, but
recommended);
    //special case for capacity 0
    if (m_capacity == 0)
      m_capacity = 1;
    else
      m_capacity *= 2;

    // allocate an array of the new capacity
    T* newBuffer = new T[m_capacity];
    // copy old items into new array
    for(int i = 0; i < m_size; i++)
      newBuffer[i] = m_buffer[i];

     // delete original array (harmless if m_buffer is null)
    delete [] m_buffer;

     // install new array
     m_buffer = newBuffer;
  }

  // add item to the array, update m_size
  m_buffer[m_size] = item;
  m_size++;
}
```

## Extra Practice

Recursion:

1) Implement the function `sumOfDigits` recursively. The function returns the sum of all of the digits in the given *positive* integer `num`.

```
int sumOfDigits(int num);

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

int sumOfDigits(int num) {
        if (num < 10)
                return num;
        return num % 10 + sumOfDigits(num/10);
}
```

2) Write the following linked list functions recursively.

```
// Node definition for singly linked list
struct Node {
        int data;
        Node* next;
};

// inserts a value in a sorted linked list of integers
//          returns list head
// before: 1 → 3 → 5 → 7 → 15
// insertInOrder(head, 8);
// after: 1 → 3 → 5 → 7 → 8 → 15
Node* insertInOrder(Node* head, int value);

// deletes all nodes whose keys/data == value, returns list
head
// use the delete keyword
Node* deleteAll(Node* head, int value);

// prints the values of a linked list backwards
// e.g. 0 → 2 → 1 → 4 → 1 → 7
// reversePrint(head) will output 714120
void reversePrint(Node* head);
```

```
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 nodes whose keys/data == value, returns list
head
Node* deleteAll(Node* head, int value) {
      if (head == nullptr)
            return nullptr;
      else {
            if (head->data == value) {
                  Node* temp = head->next;
                  delete head;
                  return deleteAll(temp, value);
            }
            else {
                  head->next = deleteAll(head->next, value);
                  return head;
            }
      }
}

// prints the values of a linked list backwards
// e.g. 0 → 2 → 1 → 4 → 1 → 7
// reversePrint(head) will output 714120
void reversePrint(Node* head) {
      if (head == nullptr)
            return;
      reversePrint(head->next);
      cout << head->data;
}
```

3) 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. (Note : the robot doesn't have to only end up at the first position past the end of the array)

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]);
}

// example 3 shows hint that you can reduce problem two ways:
     going up the stairs or down the stairs

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;
}
```

## Template/STL:

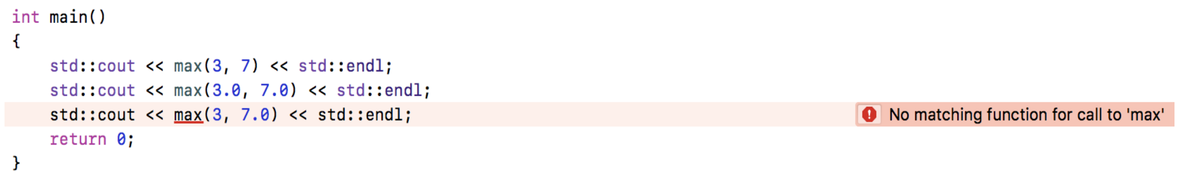
1) Will this code compile? If so, what is the output? If not, what is preventing it from compiling?

   Note: We did not use `namespace std` because `std` has its own implementation of `max` and `namespace std` will thus confuse the compiler.

```
template <typename T>
T max(T x, T y)
{
      return (x > y) ? x : y;
}

int main()
{
      std::cout << max(3, 7) << std::endl;      // line 1
      std::cout << max(3.0, 7.0) << std::endl; // line 2
      std::cout << max(3, 7.0) << std::endl;   // line 3
}
```

   On Xcode, it gives the following error messages:



```
int main()
{
    std::cout << max(3, 7) << std::endl;
    std::cout << max(3.0, 7.0) << std::endl;
    std::cout << max(3, 7.0) << std::endl;                No matching function for call to 'max'
    return 0;
}
```

   For max, the compiler expects two arguments that are of the same type, as indicated in the template declaration T. In the third call, 3 is an integer and 7.0 is a double, so there is no matching function call for this instance.
   If we were to remove line 3, lines 1 and 2 would both output 7.

2) The following code has 3 errors that cause either runtime or compile time errors. Find and fix all of the errors.

```
class Potato {
public:
  Potato(int in_size) : size(in_size) { };
  int getSize() const {
    return size;
  }
private:
  int size;
};

int main() {
  set<Potato> potatoes;  // 1
  Potato p1(3);
  Potato p2(4);
  Potato p3(5);
  potatoes.insert(p1);
  potatoes.insert(p2);
  potatoes.insert(p3);

  set<Potato>::iterator it = potatoes.begin();
  while (it != potatoes.end()) {
    it = potatoes.erase(it);  // 2
    ~~it++;~~
  }

  for (it = potatoes.begin(); it != potatoes.end(); it++) {
    cout << it->getSize() << endl;  // 3
  }
}
```

```
1: The type set<Potato> requires that the Potato object size
can be compared with operator<.  Here's an example of how to
define <:
bool operator<(const Potato& a, const Potato& b) {
  return a.getSize() < b.getSize();
}

2: After calling erase with the iterator it, it is invalidated.
Instead of incrementing it, the return value of
potatoes.erase(it), which is the iterator to the next element
after the erased one, should be assigned to it.

3: Iterators use pointer syntax, so the last for loop should
use it->getSize() instead of it.getSize().
```

3) You are given an STL `set<list<int>*>`. In other words, you have a set of pointers, and each pointer points to a list of ints. Consider the sum of a list to be the result of adding up all elements in the list. If a list is empty, treat its sum

as zero.
Write a function that removes the lists with odd sums from the set. The lists with odd sums should be deleted from memory and their pointers should be removed from the set. This function should return the number of lists that are removed. You may assume that none of the pointers is null.

```
int deleteOddSumLists(set<list<int>*>& s) {
   int numDeleted = 0;

   // iterate over the set
   set<list<int>*>::iterator set_it = s.begin();
   while (set_it != s.end())
   {
      // iterate over each list and get the sum
      int sum = 0;
      list<int>::iterator list_it = (*set_it)->begin();
      list<int>::iterator list_end = (*set_it)->end();
      while (list_it != list_end)
      {
         sum += *list_it;
         list_it++;
      }

      // delete list and remove from set if sum is odd
      // otherwise, proceed to check the next list
      if (sum % 2 == 1)
      {
         delete *set_it;
         set_it = s.erase(set_it);
         numDeleted++;
      }
      else
         set_it++;
   }

   return numDeleted;
}

// Sample driver code:
int main()
{
    set<list<int>*> s;
    list<int>* l1 = new list<int>;
    l1->push_back(1);
    l1->push_back(2);
    list<int>* l2 = new list<int>;;
    l2->push_back(1);
    l2->push_back(1);
    list<int>* l3 = new list<int>;;
    l3->push_back(1);
    l3->push_back(0);
    s.insert(l1);
    s.insert(l2);
    s.insert(l3);
    cout << deleteOddSumLists(s) << endl;
}
```