Suggested Solutions to Midterm Problems

1. The following is a recursive, but simplified, variant of Euclid's algorithm for computing the greatest common divisor of two positive integers.

```
int gcd(int a, int b)
{
  if (a == 0)
    return b;
  if (b == 0)
    return a;
  if (a > b)
    return gcd(a - b, b);
  else
    return gcd(a, b - a);
}
```

   (a) (4 points) What are the base cases? Why are they appropriate?

   Solution. The base cases are when $a = 0$ or $b = 0$. They are appropriate, as under either of the conditions, the computation may stop with the correct greatest common divisor (represented by the other non-zero argument). This means that the algorithm terminates correctly if it ever will.

   Further details: when $a > b > 0$, $\gcd(a, b) = \gcd(a - b, b)$; and when $b \geq a > 0$, $\gcd(a, b) = \gcd(a, b - a)$. Reasoning inductively, the final returned result equals the gcd of the original two input numbers. □

   (b) (6 points) Will the base cases always be reached? Why? What precondition should be stated?

   Solution. Yes, one of the base cases will eventually be reached, assuming that both of the original input numbers are positive. This means that the algorithm will terminate.

   When $a > b > 0$, `gcd` is invoked with $a - b$ and b, and when $b \geq a > 0$, `gcd` is invoked with a and $b - a$. In either case, one of the two numbers becomes strictly smaller, yet still greater than or equals to 0. Eventually, one of them will become 0.

   Precondition: $a \geq 0$ and $b \geq 0$ and ($a \neq 0$ or $b \neq 0$). □

2. Consider the array-based implementation of the ADT bag, the C++ class `ArrayBag`. We have implemented the function `remove()` with a static array as follows:

```
template<typename ItemType>
bool ArrayBag::remove(const typename& anEntry)
{
  int locatedIndex = getIndexOf(anEntry);
  bool canRemoveItem = (locatedIndex > -1);
  if(canRemoveItem)
  {
```

```
    itemCount--;
    items[locatedIndex] = items[itemCount];
  }
  return canRemoveItem;
}
```

Suppose that we now want to implement `ArrayBag` with dynamic memory allocation. The class definition has been changed to

```
template<typename ItemType>
class ArrayBag: public BagInterface
{
private:
  static const int DEFAULT_CAPACITY = 6;
  ItemType* items;
  int itemCount;
  int maxItems;
  int getIndexOf(const ItemType& target) const;
public:
  // public member functions are omitted
};
```

where `items` is a pointer pointing to the dynamic array. We now want to modify `remove()` and add the following feature: *When the number of items is less than half of the current bag capacity, cut the bag capacity by a half.* Rewrite `remove()` to add this feature.

(Note: you may decide to round up or down when the capacity happens to be odd. The function `getIndexOf()` returns the array index of a given item if it exists or -1 otherwise.)

Solution. See the slides for TA Session #3 on 11/21. □

3. Convert the infix expression `a * (b / c - d) - e * f` to the postfix form. Please follow the conversion algorithm discussed in class (see the appendix), and show the status of the stack and the current postfix expression after each character of the infix expression is processed. (Note: you may ignore the blank spaces.)

   Solution. See the slides for TA Session #3 on 11/21. □

4. Consider again the infix to postfix conversion algorithm, which assumes all operators are associated to the left. Suppose we want to allow the unary `-` (negation), which has higher precedence than `*` and `/` and is associated to the right, i.e., `- - a` equals `- (- a)`. Please modify the conversion algorithm to allow the additional unary `-`. Is it necessary to use a new symbol to represent the unary `-` in the postfix expression?

   Solution. Below is the modified algorithm that also allows the unary `-`. We use `~` to represent the unary `-` internally and in the postfix expression, assuming that the semantics and the precedence level of `~` are appropriately defined. Though not absolutely necessary (a get-around is possible, e.g., $-a = 0 - a$), a new symbol for the unary `-` makes it possible to preserve the structure of the original expression.

```
lastCh = undefined;
```

```
for (each character ch in the infix expression) {
  switch (ch) {
    case operand:  // Append operand to the end of postfixExp
      postfixExp = postfixExp + ch
      lastCh = operand;
      break
    case '(':      // Save '(' on the stack aStack
      aStack.push(ch);
      lastCh = '(';
      break
    case operator: // Process operators of higher precedence
      if (ch=='-' and ((lastCh == undefined) or (lastCh == '(') or
                       (lastCh is an operator)) {
        a.Stack.push('~');
        lastCh = operator;
        break
      }
      while (!aStack.isEmpty() and
             aStack.peek() is not a '(' and
             precedence(ch) <= precedence(aStack.peek())) {
        postfixExp = postfixExp + aStack.peek()
        aStack.pop()
      }
      aStack.push(ch); // Save the operator
      lastCh = operator;
      break
    case ')': // Pop the stack until matching '('
      while (aStack.peek() is not a '(') {
        postfixExp = postfixEP + aStack.peek()
        aStack.pop()
      }
      aStack.pop();  // Remove the matching '('
      lastCh = `)';
      break
  }
}
// Append to postfixExp the operators remaining in the stack
while (!aStack.isEmpty()) {
  postfixExp = postfixExp + aStack.peek()
  aStack.pop()
}
```

□

5. How are stacks and recursion related? Please explain the relationship by considering solutions to the task of determining the existence of a path on a directed graph. You should use an example graph to make your explanation more concrete.

   Solution. The activities of invocation and return during the execution of a recursive function may be emulated by operations a stack, as the activities exhibit a last-in-first-out

property just like a stack. (In fact, the invocation and return of any function, recursive or non-recursive, is implemented with the help of a stack.) A recursive call can be emulated by a push and a return by a pop.

To be completed. □

6. Design an algorithm (in pseudocode) to sort a stack of integers in increasing order with the smallest element on the top. You may use additional stacks and integer variables, but no other data structures (including arrays). You should try to use as few additional stacks as possible.

   Solution. See the slides for TA Session #3 on 11/21. □

7. Compare array-based (probably with dynamic memory allocation) and link-based implementations of the ADT list.

   Solution. To be completed. □

8. Below is the merge sort algorithm that we discussed in class.

```
void merge(ItemType theArray[], int first, int mid, int last)
{
  ItemType tempArray[MAX_SIZE];
  int first1 = first;
  int last1 = mid;
  int first2 = mid + 1;
  int last2 = last;

  int index = first1;
  while ((first1 <= last1) && (first2 <= last2))
    if (theArray[first1] <= theArray[first2])
      tempArray[index++] = theArray[first1++];
    else
      tempArray[index++] = theArray[first2++];

  while (first1 <= last1)
    tempArray[index++] = theArray[first1++];

  while (first2 <= last2)
    tempArray[index++] = theArray[first2++];

  for (index = first; index <= last; index++)
    theArray[index] = tempArray[index];
}

void mergeSort(ItemType theArray[], int first, int last)
{
  if (first < last) {
    int mid = first + (last - first) / 2;
    mergeSort(theArray, first, mid);
    mergeSort(theArray, mid + 1, last);
    merge(theArray, first, mid, last);
```

```
  }
}
```

Apply the algorithm to the following array. Show the contents of the array after each merge operation.

1	2	3	4	5	6	7	8	9	10	11	12
9	4	10	6	12	7	5	1	2	11	3	8

Solution. See the slides for TA Session #3 on 11/21. □

9. A sorting algorithm is *stable* if it does not exchange the items that have the same sort key. Among the four sorting algorithms: selection sort, bubble sort, merge sort, and quick sort, which are stable and which are not? Please give a brief explanation for each case. For those that are not stable, propose one single uniform adaptation to turn them into stable ones.

   Solution. See the slides for TA Session #3 on 11/21. □

10. Design an algorithm that, given an array of positive and negative numbers, rearranges the array entries so that all negative numbers appear before the positive numbers. Please present your algorithm in suitable pseudocode. Give a performance analysis of your algorithm. The more efficient your algorithm is the more points you will be credited for this problem.

    Solution. See the slides for TA Session #3 on 11/21. □