CMPS 12B
Introduction to Data Structures
Midterm 2 Review Problems
Solutions to Selected Problems

1. Write a *recursive* Java function called `product()`, which given a head reference to a linked list based on the Node class defined below, returns the product of the items in the list. The product of an empty list is defined to be one.

```
class Node{
   int item;
   Node next;
   Node(int x){
      item = x;
      next = null;
   }
}

// In some class in the same package as Node:

static int product(Node H){
   // Your code goes here

   if( H==null ){
      return 1;
   }else{
      return H.item*product(H.next);
   }

}
```

2. Write functions push() and pop() for the Java implementation of an integer stack outlined below. The stack is implemented as a singly linked list with a top Node reference. Function push() inserts a new item onto the stack by inserting a new Node at the head of the list.

```
class Stack{
   private class Node{
      int item;
      Node next;
      Node(int item){
         this.item = item;
         this.next = null;
      }
   }
   private Node top;
   private int numItems;
   public Stack(){top = null; numItems = 0;}

   void push(int x){
      // your code goes here

      if( numItems==0 ){
         top = new Node(x);
      }else{
         Node N = new Node(x);
         N.next = top;
         top = N;
      }
      numItems++;

   }

   int pop(){
      // your code goes here

      if( numItems==0 ){
         throw new RuntimeException("cannot pop() empty stack");
      }
      int x = top.item;
      top = top.next;
      numItems--;
      return x;

   }
   // other Stack methods would follow
}
```

6. Write a C function called `search()` with the protype below that takes as input a null terminated `char` array S (i.e. a string) and a single `char c`, and returns the leftmost index in S at which the target `c` appears, or returns -1 if no such index exists.

```
int search(char* S, char c){
   // your code goes here

   int i=0;

   while( S[i]!=’\0’ ){
      if( S[i]==c ) break;
      i++;
   }
   if( S[i]==’\0’ ) return -1;
   else return i;

}
```

8. Consider the C function below called `wasteTime()`. Your goal is to determine how much time `wastTime()` wastes. The stared (*) lines below are to be considered basic operations, which do nothing but waste a multiple of some unspecified time unit. Determine the total amount T(*n*) of time wasted on the input *n*. Find the asymptotic runtime of this algorithm, i.e. T(*n*) = Θ(some simple function of *n*).

```
     void wasteTime(int n){
        int i, j, k;
*       waste 2 units of time;
        for(i=0; i<n; i++){
*          waste 5 units of time;
           for(j=0; j<n j++){
*             waste 12 units of time;
              for(k=0; k<n; k++){
*                waste 3 units of time;
              }
           }
        }
     }
```

Solution: $T(n) = 2+n(5+n(12+n(3))) = 3n^3+12n^2+5n+2 = \Theta(n^3)$

12. Write a C function called `CountComparisons()` that takes as input an `int` array `A`, and `int n` giving the length of `A`, and an `int i` specifying an index to `A`. The function will return an `int` giving the number of elements in `A` that are less than `A[i]`. Determine the number of comparisons performed by your function (in terms of the array length *n*). How can you use your function as the basis for a sorting algorithm?

```
int CountComparisons(int* A, int n, int i){
   // your code goes here

   int j, count=0;
   for(j=0; j<n; j++){
      if( A[j]<A[i] ){
         count++;
      }
   }
   return count;

}
```

This function will perform exactly *n* comparisons on an array of length *n*.

If array `A[]` contains no repeated elements, then `CountComparisons(A, n, i)` is the index where the element `A[i]` belongs in a sorted array containing the same elements. If `B[]` is an output array of length *n*, we could set `B[CountComparisons(A, n, i)] = A[i]` in a loop controlled by *i* going from 0 to *n*-1. Array `B[]` is then the sorted version of `A[]`. The case where A[] contains repeated elements is delt with in the next problem.

13. Use the function `CountComparisons()` in the previous problem to create a sorting function with heading `void ComparisonSort(int* A, int* B, int n)` that takes an int array `A[]` as input, and copies the elements in `A[]` into the int array `B[]` in sorted order. (Hint: First assume the elements of `A[]` are distinct. In this case the number of numbers in `A[]` that are less than `A[i]` is the index where `A[i]` belongs in the output array `B[]`. Figure out what to do in the case that `A[]` contains repeated elements.)

```
void ComparisonSort(int* A, int* B, int n){
   int* Offset = calloc(n, sizeof(int));
   int i, j;
   // figure out where to put A[i] in the output array B[]
   for(i=0; i<n; i++){
      Offset[i] = CountComparisons(A, n, i);
   }
   // put A[i] there
   for(i=0; i<n; i++){
      B[Offset[i]] = A[i];
      //this loop is only necessary if there are repeated elements
      for(j=i+1; j<n; j++){
         if(Offset[j]==Offset[i]) Offset[j]++;
      }
   }
   free(Offset);
}
```