```
// Project 4 Solution

#include <string>

using namespace std;

void exchange(string& s1, string& s2)
{
    string t = s1;
    s1 = s2;
    s2 = t;
}

int appendToAll(string a[], int n, string value)
{
    if (n < 0)
        return -1;
    for (int k = 0; k < n; k++)
        a[k] += value;
    return n;
}

int lookup(const string a[], int n, string target)
{
    if (n < 0)
        return -1;
    for (int k = 0; k < n; k++)
        if (a[k] == target)
            return k;
    return -1;
}

int positionOfMax(const string a[], int n)
{
    if (n <= 0)
        return -1;
    int maxPos = 0;  //  assume to start that the max is at position 0
    for (int k = 1; k < n; k++)  // so start at position 1 to look at the rest
        if (a[k] > a[maxPos])
            maxPos = k;
    return maxPos;
}

int rotateLeft(string a[], int n, int pos)
{
    if (n < 0  ||  pos < 0  ||  pos >= n)
        return -1;

      // save the element that is to be moved
    string toBeMoved = a[pos];

      // shift left the elements that are after the one to be moved
    for (int k = pos; k < n-1; k++)
        a[k] = a[k+1];

      // place the moved element at the end
    a[n-1] = toBeMoved;
    return pos;
}

int countRuns(const string a[], int n)
{
    if (n < 0)
        return -1;
```

```
    if (n == 0)
        return 0;
    int nRuns = 1;  // there's at least one element, so at least one run
    for (int k = 1; k < n; k++)  // start at 1, not 0
        if (a[k] != a[k-1])  // start of new run?
            nRuns++;
    return nRuns;
}

int flip(string a[], int n)
{
    if (n < 0)
        return -1;
      // exchange elements at positions 0 and n-1, then 1 and n-2, then 2 and
      // n-3, etc., stopping in the middle of the array
    for (int k = 0; k < n/2; k++)
        exchange(a[k], a[n-1-k]);
    return n;
}

int differ(const string a1[], int n1, const string a2[], int n2)
{
    if (n1 < 0  ||  n2 < 0)
        return -1;
    int n = (n1 < n2 ? n1 : n2);  // minimum of n1 and n2
    for (int k = 0; k < n; k++)
        if (a1[k] != a2[k])
            return k;
    return n;
}

int subsequence(const string a1[], int n1, const string a2[], int n2)
{
    if (n1 < 0  ||  n2 < 0)
        return -1;

      // Try matching the a2 sequence starting at each position of a1 that
      // could be the start of sequence of length n2

    for (int k1 = 0; k1 < n1 - n2 + 1; k1++)
    {
          // See if the a2 sequence matches starting at k1 in a1

        bool match = true;  // Assume they match until proved otherwise
        for (int k2 = 0; k2 < n2; k2++)
        {
            if (a1[k1+k2] != a2[k2])
            {
                match = false;
                break;
            }
        }
        if (match)  // We never found a mismatch, so they match
            return k1;
    }
    return -1;
}

int lookupAny(const string a1[], int n1, const string a2[], int n2)
{
    if (n1 < 0  ||  n2 < 0)
        return -1;
    for (int k = 0; k < n1; k++)
        if (lookup(a2, n2, a1[k]) != -1)
            return k;
```

```
    return -1;
}

int separate(string a[], int n, string separator)
{
    if (n < 0)
        return -1;

      // It will always be the case that just before evaluating the loop
      // condition:
      //  firstNotLess <= firstUnknown and firstUnknown <= firstGreater
      //  Every element earlier than position firstNotLess is < separator
      //  Every element from position firstNotLess to firstUnknown-1 is
      //    == separator
      //  Every element from position firstUnknown to firstGreater-1 is
      //     not known yet
      //  Every element at position firstGreater or later is > separator

    int firstNotLess = 0;
    int firstUnknown = 0;
    int firstGreater = n;

    while (firstUnknown < firstGreater)
    {
        if (a[firstUnknown] > separator)
        {
            firstGreater--;
            exchange(a[firstUnknown], a[firstGreater]);
        }
        else
        {
            if (a[firstUnknown] < separator)
            {
                exchange(a[firstNotLess], a[firstUnknown]);
                firstNotLess++;
            }
            firstUnknown++;
        }
    }
    return firstNotLess;
}
```