```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

typedef pair<int, int> ii;

#define MAX_N 100010                         // second approach: O(n log n)
char T[MAX_N];                   // the input string, up to 100K characters
int n;                                        // the length of input string
int RA[MAX_N], tempRA[MAX_N];        // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];    // suffix array and temporary suffix array
int c[MAX_N];                                    // for counting/radix sort

char P[MAX_N];                  // the pattern string (for string matching)
int m;                                      // the length of pattern string

int Phi[MAX_N];                      // for computing longest common prefix
int PLCP[MAX_N];
int LCP[MAX_N];  // LCP[i] stores the LCP between previous suffix T+SA[i-1]
                                              // and current suffix T+SA[i]

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }      // compare

void constructSA_slow() {               // cannot go beyond 1000 characters
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
  sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) = O(n^2 log n)
}

void countingSort(int k) {                                          // O(n)
  int i, sum, maxi = max(300, n);   // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);                          // clear frequency table
  for (i = 0; i < n; i++)       // count the frequency of each integer rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t;
  }
  for (i = 0; i < n; i++)          // shuffle the suffix array if necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++)                     // update the suffix array SA
    SA[i] = tempSA[i];
}

void constructSA() {         // this version can go up to 100000 characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                 // initial rankings
  for (i = 0; i < n; i++) SA[i] = i;     // initial SA: {0, 1, 2, ..., n-1}
  for (k = 1; k < n; k <<= 1) {       // repeat sorting process log n times
    countingSort(k);  // actually radix sort: sort based on the second item
    countingSort(0);          // then (stable) sort based on the first item
    tempRA[SA[0]] = r = 0;             // re-ranking; start from rank r = 0
    for (i = 1; i < n; i++)                    // compare adjacent suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)                     // update the rank array RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;               // nice optimization trick
} }

void computeLCP_slow() {
  LCP[0] = 0;                                              // default value
  for (int i = 1; i < n; i++) {                // compute LCP by definition
    int L = 0;                                       // always reset L to 0
    while (T[SA[i] + L] == T[SA[i-1] + L]) L++;      // same L-th char, L++
    LCP[i] = L;
} }

void computeLCP() {
  int i, L;
  Phi[SA[0]] = -1;                                         // default value
  for (i = 1; i < n; i++)                            // compute Phi in O(n)
    Phi[SA[i]] = SA[i-1];    // remember which suffix is behind this suffix
  for (i = L = 0; i < n; i++) {             // compute Permuted LCP in O(n)
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }            // special case
    while (T[i + L] == T[Phi[i] + L]) L++;       // L increased max n times
    PLCP[i] = L;
    L = max(L-1, 0);                             // L decreased max n times
  }
  for (i = 0; i < n; i++)                            // compute LCP in O(n)
    LCP[i] = PLCP[SA[i]];   // put the permuted LCP to the correct position
}

ii stringMatching() {                      // string matching in O(m log n)
  int lo = 0, hi = n-1, mid = lo;              // valid matching = [0..n-1]
  while (lo < hi) {                                     // find lower bound
    mid = (lo + hi) / 2;                              // this is round down
    int res = strncmp(T + SA[mid], P, m);  // try to find P in suffix 'mid'
    if (res >= 0) hi = mid;        // prune upper half (notice the >= sign)
    else          lo = mid + 1;           // prune lower half including mid
  }                                     // observe `=' in "res >= 0" above
  if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);    // if not found
  ii ans; ans.first = lo;
  lo = 0; hi = n - 1; mid = lo;
  while (lo < hi) {            // if lower bound is found, find upper bound
    mid = (lo + hi) / 2;
    int res = strncmp(T + SA[mid], P, m);
    if (res > 0) hi = mid;                              // prune upper half
    else         lo = mid + 1;            // prune lower half including mid
  }                           // (notice the selected branch when res == 0)
  if (strncmp(T + SA[hi], P, m) != 0) hi--;                 // special case
  ans.second = hi;
  return ans;
} // return lower/upperbound as first/second item of the pair, respectively

ii LRS() {                 // returns a pair (the LRS length and its index)
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)                         // O(n), start from i = 1
    if (LCP[i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}

int owner(int idx) { return (idx < n-m-1) ? 1 : 2; }

ii LCS() {                 // returns a pair (the LCS length and its index)
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)                         // O(n), start from i = 1
    if (owner(SA[i]) != owner(SA[i-1]) && LCP[i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}

int main() {
  //printf("Enter a string T below, we will compute its Suffix Array:\n");
  strcpy(T, "GATAGACA");
  n = (int)strlen(T);
  T[n++] = '$';
  // if '\n' is read, uncomment the next line
  //T[n-1] = '$'; T[n] = 0;

  constructSA_slow();                                       // O(n^2 log n)
  printf("The Suffix Array of string T = '%s' is shown below (O(n^2 log n)
version):\n", T);
  printf("i\tSA[i]\tSuffix\n");
  for (int i = 0; i < n; i++) printf("%2d\t%2d\t%s\n", i, SA[i], T + SA[i]);

  constructSA();                                              // O(n log n)
  printf("\nThe Suffix Array of string T = '%s' is shown below (O(n log n)
version):\n", T);
  printf("i\tSA[i]\tSuffix\n");
  for (int i = 0; i < n; i++) printf("%2d\t%2d\t%s\n", i, SA[i], T + SA[i]);

  computeLCP();                                                     // O(n)

  // LRS demo
  ii ans = LRS();                 // find the LRS of the first input string
  char lrsans[MAX_N];
  strncpy(lrsans, T + SA[ans.second], ans.first);
  printf("\nThe LRS is '%s' with length = %d\n\n", lrsans, ans.first);

  // stringMatching demo
  //printf("\nNow, enter a string P below, we will try to find P in T:\n");
  strcpy(P, "A");
  m = (int)strlen(P);
  // if '\n' is read, uncomment the next line
  //P[m-1] = 0; m--;
  ii pos = stringMatching();
  if (pos.first != -1 && pos.second != -1) {
    printf("%s is found SA[%d..%d] of %s\n", P, pos.first, pos.second, T);
    printf("They are:\n");
    for (int i = pos.first; i <= pos.second; i++)
      printf("  %s\n", T + SA[i]);
  } else printf("%s is not found in %s\n", P, T);

  // LCS demo
  //printf("\nRemember, T = '%s'\nNow, enter another string P:\n", T);
  // T already has '$' at the back
  strcpy(P, "CATA");
  m = (int)strlen(P);
  // if '\n' is read, uncomment the next line
  //P[m-1] = 0; m--;
  strcat(T, P);                                                 // append P
  strcat(T, "#");                                    // add '$' at the back
  n = (int)strlen(T);                                           // update n

  // reconstruct SA of the combined strings
  constructSA();                                              // O(n log n)
  computeLCP();                                                     // O(n)
  printf("\nThe LCP information of 'T+P' = '%s':\n", T);
  printf("i\tSA[i]\tLCP[i]\tOwner\tSuffix\n");
  for (int i = 0; i < n; i++)
    printf("%2d\t%2d\t%2d\t%2d\t%s\n", i, SA[i], LCP[i], owner(SA[i]), T + SA[i]);

  ans = LCS();         // find the longest common substring between T and P
  char lcsans[MAX_N];
  strncpy(lcsans, T + SA[ans.second], ans.first);
  printf("\nThe LCS is '%s' with length = %d\n", lcsans, ans.first);

  return 0;
}
```