```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>

// global variables
enum {LCS, ED, SW, NONE} alg_type; // which algorithm to run
char *alg_desc; // description of which algorithm to run
char *result_string; // text to print along with result from algorithm
char *x, *y; // the two strings that the algorithm will execute on
char *filename; // file containing the two strings
int xLen, yLen, alphabetSize; // lengths of two strings and size of alphabet
bool iterBool = false, recNoMemoBool = false, recMemoBool = false; // which type
 of dynamic programming to run
bool printBool = false; // whether to print table
bool readFileBool = false, genStringsBool = false; // whether to read in strings
 from file or generate strings randomly

// NEW VARIABLES
//Struct for tuple in a table containing value and pointer to secondary array
typedef struct {
  int entry;
  int pointer;
} tableTuple;

//Struct for tuple in secondary array, x-index and y-index
typedef struct {
  int x_index;
  int y_index;
} compTableTuple;

//table for printable table
tableTuple **table; //2d array of tableTuples
compTableTuple *comp_array; //1d array of compTableTuples
long long ins_count = 0; //Insertion count
int answer = 0; //Final answers from algorithms
long long rec_counter = 0; //A counter for recursive calls
//NEW VARIABLES END


// functions follow

// determine whether a given string consists only of numerical digits
bool isNum(char s[]) {
        int i;
        bool isDigit=true;
        for (i=0; i<strlen(s); i++)
                isDigit &= s[i]>='0' && s[i]<='9';
        return isDigit;
}

// get arguments from command line and check for validity (return true if and on
ly if arguments illegal)
bool getArgs(int argc, char *argv[]) {
        int i;
        alg_type = NONE;
        xLen = 0;
        yLen = 0;
        alphabetSize = 0;
        for (i = 1; i < argc; i++) // iterate over all arguments provided (argum
```



```
ent 0 is name of this module)
                if (strcmp(argv[i],"-g")==0) { // generate strings randomly
                        if (argc>=i+4 && isNum(argv[i+1]) && isNum(argv[i+2]) &&
 isNum(argv[i+3])) { // must be three numerical arguments after this
                                xLen=atoi(argv[i+1]); // get length of x
                                yLen=atoi(argv[i+2]); // get length of y
                                alphabetSize = atoi(argv[i+3]); // get alphabet
size
                                genStringsBool = true; // set flag to generate s
trings randomly
                                i+=3; // ready for next argument
                        }
                        else
                                return true; // must have been an error with -g
arguments
                }
                else if (strcmp(argv[i],"-f")==0) { // read in strings from file
                        if (argc>=i+2) { // must be one more argument (filename)
 after this)
                                i++;
                                filename = argv[i]; // get filename
                                readFileBool = true; // set flag to read in stri
ngs from file
                        }
                        else
                                return true; // must have been an error with -f
argument
                }
                else if (strcmp(argv[i],"-i")==0) // iterative dynamic programmi
ng
                        iterBool = true;
                else if (strcmp(argv[i],"-r")==0) // recursive dynamic programmi
ng without memoisation
                        recNoMemoBool = true;
                else if (strcmp(argv[i],"-m")==0) // recursive dynamic programm
ing with memoisation
                        recMemoBool = true;
                else if (strcmp(argv[i],"-p")==0) // print dynamic programming t
able
                        printBool = true;
                else if (strcmp(argv[i],"-t")==0) // which algorithm to run
                        if (argc>=i+2) { // must be one more argument ("LCS" or
"ED" or "SW")
                                i++;
                                if (strcmp(argv[i],"LCS")==0) { // Longest Comm
on Subsequence
                                        alg_type = LCS;
                                        alg_desc = "Longest Common Subsequence";
                                        result_string = "Length of a longest common subs
equence is";
                                }
                                else if (strcmp(argv[i],"ED")==0) { // Edit Dis
tance
                                        alg_type = ED;
                                        alg_desc = "Edit Distance";
                                        result_string = "Edit distance is";
                                }
                                else if (strcmp(argv[i],"SW")==0) { // Smith-Wa
terman Algorithm
                                        alg_type = SW;
                                        alg_desc = "Smith-Waterman algorithm";
                                        result_string = "Length of a highest scoring local s
```

```
imilarity is");
                                }
                                else
                                        return true; // none of these; illegal c
hoice
                        }
                        else
                                return true; // algorithm type not given
                else
                        return true; // argument not recognised
                // check for legal combination of choices; return true (illegal)
 if user chooses:
                // - neither or both of generate strings and read strings from f
ile
                // - generate strings with length 0 or alphabet size 0
                // - no algorithm to run
                // - no type of dynamic programming
                return !(readFileBool ^ genStringsBool) || (genStringsBool && (x
Len <=0 || yLen <= 0 || alphabetSize <=0)) || alg_type==NONE || (!iterBool && !r
ecMemoBool && !recNoMemoBool);
}

// read strings from file; return true if and only if file read successfully
bool readStrings() {
        // open file for read given by filename
        FILE * file;
        file = fopen(filename, "r");
        // firstly we will measure the lengths of x and y before we read them in
 to memory
        if (file) { // file opened successfully
                // first measure length of x
                bool done = false;
                int i;
                do { // read from file until newline encountered
                        i = fgetc(file); // get next character
                        if (i==EOF) { // EOF encountered too early (this is firs
t string)
                                // print error message, close file and return fa
lse
                                printf("Incorrect file syntax\n");
                                fclose(file);
                                return false;
                        }
                        if ((char) i=='\n' || (char) i=='\r') // newline encounte
red
                                done = true; // terminate loop
                        else // one more character
                                xLen++; // increment length of x
                } while (!done);
                // next measure length of y
                if ((char) i=='\r')
                        fgetc(file); // get rid of newline character
                done = false;
                do { // read from file until newline or EOF encountered
                        int i = fgetc(file); // get next character
                        if (i==EOF || (char) i=='\n' || (char) i=='\r') // EOF or
 newline encountered
                                done = true; // terminate loop
                        else // one more character
                                yLen++; // increment length of y
                } while (!done);
                fclose(file);
```



```
                // if either x or y is empty then print error message and return
 false
                if (xLen==0 || yLen==0) {
                        printf("Incorrect file syntax\n");
                        return false;
                }
                // now open file again for read
                file = fopen(filename, "r");
                // allocate memory for x and y
                x = malloc(xLen * sizeof(char));
                y = malloc(yLen * sizeof(char));
                // read in x character-by-character
                for (i=0; i<xLen; i++)
                        x[i]=fgetc(file);
                i = fgetc(file); // read in newline between strings and discard
                if ((char) i=='\r')
                        fgetc(file); // read \n character and discard if previou
s character was \r
                // read in y character-by-character
                for (i=0; i<yLen; i++)
                        y[i]=fgetc(file);
                // close file and return boolean indicating success
                fclose(file);
                return true;
        }
        else { // notify user of I/O error and return false
                printf("Problem opening file %s\n",filename);
                return false;
        }
}

// generate two strings x and y (of lengths xLen and yLen respectively) uniforml
y at random over an alphabet of size alphabetSize
void generateStrings() {
        // allocate memory for x and y
        x = malloc(xLen * sizeof(char));
        y = malloc(yLen * sizeof(char));
        // instantiate the pseudo-random number generator (seeded based on curre
nt time)
        srand(time(NULL));
        int i;
        // generate x, of length xLen
        for (i = 0; i < xLen; i++)
                x[i] = rand()%alphabetSize +'A';
        // generate y, of length yLen
        for (i = 0; i < yLen; i++)
                y[i] = rand()%alphabetSize +'A';
}

// free memory occupied by strings
void freeMemory() {
  free(x);
        free(y);
}


//NEW FUNCTIONS
//Max and Min functions
//max2 - returns greater of 2 numbers
int max2(int num, int num2){
        if (num > num2){
                return num;
```

```
        }else{
                return num2;
        }
}

//max3 − returns greater of 3 numbers
int max3(int num, int num2, int num3){
        return max2(max2(num, num2),num3);
}

//max4 − returns greater of 4 numbers
int max4(int num, int num2, int num3, int num4){
        return max2(max2(num, num2), max2(num3, num4));
}

//min2 − returns lesser of 2 numbers
int min2(int num, int num2){
        if (num < num2){
                return num;
        }else{
                return num2;
        }
}

//min3 − returns lesser of 3 numbers
int min3(int num, int num2, int num3){
        return min2(min2(num,num2),num3);
}


//Table functions
//Checks if index of 2−d virtually initialized array has been initialized
bool is_real_value(int i, int j){
        //Get pointer value
        int a = table[i][j].pointer;

        //If nothing been added yet or is too large
        if (a > ins_count − 1){
                return false;
        //Else check if a real value i.e is validated by comp_array
        }else{

                bool insx = (comp_array[a].x_index == i);
                bool insy = (comp_array[a].y_index == j);
                if ((insx) & (insy)){
                    return true;
                }
                return false;
        }
}

//Adds to the 2−d virtually initialized array
void add_to_table(int i, int j, int value){
        //Add one to insertion counter
        table[i][j].entry = value;
        table[i][j].pointer = ins_count;
        comp_array[ins_count].x_index = i;
        comp_array[ins_count].y_index = j;
        ins_count = ins_count + 1;
}

//Virutally Initialise a 2−d array and companion array of real insertions
```



```
void init_table(int x_size, int y_size){
  int i = 0;
  //Initialise table
  table = (tableTuple **) malloc ((x_size+1)*sizeof(tableTuple *));
  if(table == NULL){
    printf("Malloc error");
  }
  for (i=0; i <= x_size; i++){
    table[i]=(tableTuple *) malloc ((y_size+1)*sizeof(tableTuple));
    if(table[i] == NULL){
      printf("Malloc error");
    }
  }

  //Secondary array − at maximum will need xLen+1 * yLen+1spaces
  comp_array = (compTableTuple *) malloc(((x_size+1)*(y_size+1))*sizeof(compTabl
eTuple));
  if(comp_array == NULL){
    printf("Malloc error");
  }
  ins_count = 0;
}

//free_table − frees the table and second array from the memory
void free_table(){
  int i;
  for (i=0; i<= yLen;i++){
    free(table[i]);
  }
  free(table);
  free(comp_array);
}


//Printing functions
//print_space − just prints multiple tabs for format
void print_space(int width){
        printf("%−*s%−*s%−*s", width, " ", width, " ", width, " ");
}

//print_table − prints out a neat table for the algorithms
//args − the table − int[][]
void print_table(int col_width){

        int i;
        int j;
        //Print x axis − indices
        printf("\n");
        print_space(col_width);
        for (i=0; i<=yLen; i++){
                        printf("%−*d", col_width, i % 10);
        }

        //Print x axis − characters
        printf("\n");
        print_space(col_width);
        printf("%−*s", col_width, " ");
        for (i=1; i<=yLen; i++){
                        printf("%−*c", col_width, y[i−1]);
        }

        //Print x axis − border
```

```
        printf("\n");
        print_space(col_width);
        for (i=0; i<=yLen; i++){
                        for (j=0; j<col_width; j++){
                                printf("-");
                        }
        }

        //For every row
        printf("\n");
        for (i=0; i<=xLen; i++){

                //Print y axis
                printf("%-*d", col_width, i % 10);
                if (i==0){
                        printf("%-*s", col_width, " ");
                }else{
                        printf("%-*c", col_width, x[i-1]);
                }
                printf("%-*s", col_width, "|");

                //Print table values
                for (j=0; j<=yLen; j++){
      if (is_real_value(i, j) == 0){
        //If doing LCS/ED/SW table then should be -, if q table of computation c
ounts should be 0
        if (recMemoBool){
          printf("%-*s", col_width, "-");
        }else{
          printf("%-*d", col_width, 0);
        }
      }else{
        printf("%-*d", col_width, table[i][j].entry);
      }
                }
                printf("\n");
        }
}

//print_align - prints the optimal alignment for a LCS table
void print_align(){
  int i,j;
  i = xLen;
  j = yLen;
  int current_val;
  char *first_line = malloc((xLen+yLen) * sizeof(char));
  char *snd_line = malloc((xLen+yLen) * sizeof(char));
  char *third_line = malloc((xLen+yLen) * sizeof(char));
  int count = 0;

  //Get optimal alignment
  while (i > 0 | j > 0){
    current_val = table[i][j].entry;

    //If cell to left is equal, move to that cell
    if (j > 0 & (table[i][j-1].entry == current_val)){
      first_line[count] = '-';
      snd_line[count] = ' ';
      third_line[count] = y[j-1];
      j--;
    }
    //If cell above is equal, move to that cell
```



```
    else if (i > 0 & (table[i-1][j].entry == current_val)){
      first_line[count] = x[i-1];
      snd_line[count] = ' ';
      third_line[count] = '-';
      i--;
    }
    //Else move to top-left diagonal cell
    else{
      first_line[count] = x[i-1];
      snd_line[count] = '|';
      third_line[count] = y[j-1];
      i--;
      j--;
    }
    count++;
  }

  //Print optimal alignment
  printf("Optimal Alignment:\n");
  for(i=count;i>0;i--){
    printf("%c",first_line[i-1]);
  }
  printf("\n");
  for(i=count;i>0;i--){
    printf("%c",snd_line[i-1]);
  }
  printf("\n");
  for(i=count;i>0;i--){
    printf("%c",third_line[i-1]);
  }
  printf("\n");

  //Free memory
  free(first_line);
  free(snd_line);
  free(third_line);
}

//print_answer - prints out the answer to the algorithm
void print_answer(int alg, int type){
  switch (alg){
    case 1:
      printf("Length of longest common subsequence is: %d\n", answer);
      break;
    case 2:
      printf("Edit distance is: %d\n", answer);
      break;
    case 3:
      printf("Length of a highest scoring local similarity is: %d\n", answer);
      break;
  }
  switch (type){
    case 2:
      printf("Total number of times entry computed: %d\n",rec_counter);
      break;
    case 3:
      printf("Total number of times entry computed: %d\n",rec_counter);
      long cells = (xLen*yLen);
      double prop_comp = (((double)ins_count*100)/(double)cells);
      printf("Proportion of table computed: %.1f%%\n", prop_comp);
      break;
  }
```

```c
  //Print out table if required
  if (printBool){
    //Get a column-width and use to print
    int biggest = max3(1, answer, rec_counter);
    int col_width = floor (log10 (abs (biggest))) + 3;
    print_table(col_width);

    //If to print out optimal alignment
    if (alg == 1 & type == 1){
      printf("\n");
      print_align();
    }
  }
}


//Longest Common Subsequence functions
//lcs_iterative_alg - iterative algorithm for longest common subsequence
// We dont need to check for real values or check if a table is to printed as ev
ery value is calculated anyway
int lcs_iterative_alg(){
        int i, j, value;
        for (i=0; i <= xLen; i++){
                for (j=0; j <= yLen; j++){
                        if ((i == 0) | (j == 0)){
        value = 0;
                        }

                        else if (x[i-1] == y[j-1]){
                                value = table[i-1][j-1].entry + 1;
                        }

                        else {
        value = max2(table[i-1][j].entry, table[i][j-1].entry);
                        }
      add_to_table(i, j, value);
                }
        }
        return table[xLen][yLen].entry;
}

//lcs_recursive_alg - recursive algorithm with no memoisation
// If table to be printed then make add to it, remembering to check for real val
ues
int lcs_recursive_alg(int a, int b){
  //Update table if to print
  if (printBool){
    if (is_real_value(a, b) == 0){
      add_to_table(a, b, 0);
    }
    table[a][b].entry = table[a][b].entry + 1;
  }
  //Add one to recursion counter
        rec_counter++;
  //Recursive algorithm
        if ((a==0) | (b==0)){
                return 0;
        }
        else if (x[a-1] == y[b-1]){
                return 1 + lcs_recursive_alg(a-1, b-1);
        }
```



```c
        else{
                return max2(lcs_recursive_alg(a-1, b),lcs_recursive_alg(a, b-1))
;
        }
}

//lcs_recursive_memo_alg - recursive algorithm with memoisation
int lcs_recursive_memo_alg(int a, int b){
  //Add one to recursion counter
        rec_counter++;
  int value;
  //Recursive algorithm
  if ((a==0) | (b==0)){
    value = 0;
        }
        else if (x[a-1] == y[b-1]){
    //Check if value exists first
    if (is_real_value(a-1, b-1)){
      value = 1 + table[a-1][b-1].entry;
    }else{
      value = 1 + lcs_recursive_memo_alg(a-1, b-1);
    }
        }
        else{
    //Check if values exists first
    int hor_value;
    int ver_value;
    if (is_real_value(a-1, b)){
        hor_value = table[a-1][b].entry;
    }else{
      hor_value = lcs_recursive_memo_alg(a-1, b);
    }
    if (is_real_value(a, b-1)){
        ver_value = table[a][b-1].entry;
    }else{
      ver_value = lcs_recursive_memo_alg(a,b-1);
    }

                value = max2(hor_value, ver_value);
        }
  add_to_table(a, b, value);
  return value;
}

//lcs - calls necessary algorithm and prints
void lcs() {
        //Call required algorithms
        if (iterBool){
    printf("Iterative version\n");
    init_table(xLen, yLen);
    clock_t start = clock();
                answer = lcs_iterative_alg();
    double time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
    print_answer(1, 1);
    free_table();
    printf("Time taken: %f seconds\n\n", (time_spent));
        }
        if (recNoMemoBool){
    printf("Recursive version without memoisation\n");
    if (printBool){
      init_table(xLen, yLen);
    }
```

```
    clock_t start = clock();
                answer = lcs_recursive_alg(xLen, yLen);
    double time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
    print_answer(1, 2);
    if (printBool){
      free_table();
    }
    printf("Time taken: %f seconds\n\n", (time_spent));
    rec_counter = 0;
        }
  if (recMemoBool){
    printf("Recursive version with memoisation\n");
    init_table(xLen, yLen);
    clock_t start = clock();
                answer = lcs_recursive_memo_alg(xLen, yLen);
    double time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
    print_answer(1, 3);
    free_table();
    printf("Time taken: %f seconds\n\n", (time_spent));
    rec_counter = 0;
        }
}


//Edit Distance functions
//ed_iterative_alg - iterative algorithm for edit distance
int ed_iterative_alg(){
        int i, j, value;
        for (i=0; i <= xLen; i++){
                for (j=0; j <= yLen; j++){
                        if ((i == 0) | (j == 0)){
        value = 0;
                        }

                        else if (x[i-1] == y[j-1]){
                                value = table[i-1][j-1].entry;
                        }

                        else {
        value = min3(table[i-1][j].entry, table[i][j-1].entry, table[i-1][j-1].e
ntry) + 1;
                        }
    add_to_table(i, j, value);
                }
        }
        return table[xLen][yLen].entry;
}

//ed_recursive_alg - recursive algorithm for edit distance no memoisation
int ed_recursive_alg(int i, int j){
  //Update table if to print
  if (printBool){
    if (is_real_value(i, j) == 0){
      add_to_table(i, j, 0);
    }
    table[i][j].entry++;
  }
  //Add one to recursion counter
        rec_counter = rec_counter + 1;
  //Recursive algorithm
        if ((i==0) | (j==0)){
                return 0;
```



```
        }
        else if (x[i-1] == y[j-1]){
                return(ed_recursive_alg(i-1, j-1));
        }
        else{
                return min3(ed_recursive_alg(i-1, j), ed_recursive_alg(i, j-1),
ed_recursive_alg(i-1,j-1)) + 1;
        }
}

//ed_recursive_memo_alg - iterative algorithm for edit distance with memoisation
int ed_recursive_memo_alg(int i, int j){
  //Add one to recursion counter
        rec_counter++;
  int value;
  //Recursive algorithm
  if ((i==0) | (j==0)){
    value = 0;
        }
        else if (x[i-1] == y[j-1]){
    //Check if value exists first
    if (is_real_value(i-1, j-1)){
      value = table[i-1][j-1].entry;
    }else{
      value = ed_recursive_memo_alg(i-1, j-1);
    }
        }
        else{
    //Check if values exists first
    int hor_value, ver_value, diag_value;
    if (is_real_value(i-1, j)){
        hor_value = table[i-1][j].entry;
    }else{
      hor_value = ed_recursive_memo_alg(i-1, j);
    }
    if (is_real_value(i, j-1)){
        ver_value = table[i][j-1].entry;
    }else{
      ver_value = ed_recursive_memo_alg(i,j-1);
    }
    if (is_real_value(i-1, j-1)){
        diag_value = table[i-1][j-1].entry;
    }else{
      diag_value = ed_recursive_memo_alg(i-1,j-1);
    }

    value = min3(hor_value, ver_value, diag_value) + 1;

  }
  add_to_table(i, j, value);
  return value;
}

//ed -  calls necessary algorithm and prints
void ed(){
  //Call required algorithms
  if (iterBool){
    printf("Iterative version\n");
    init_table(xLen, yLen);
    clock_t start = clock();
    answer = ed_iterative_alg();
    double time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
```

```
    print_answer(2, 1);
    free_table();

    printf("Time taken: %f seconds\n\n", (time_spent));
  }
  if (recNoMemoBool){
    printf("Recursive version without memoisation\n");
    if (printBool){
      init_table(xLen, yLen);
    }
    clock_t start = clock();
    answer = ed_recursive_alg(xLen, yLen);
    double time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
    print_answer(2, 2);
    if (printBool){
      free_table();
    }
    printf("Time taken: %f seconds\n\n", (time_spent));
    rec_counter = 0;
  }
  if (recMemoBool){
    printf("Recursive version with memoisation\n");
    clock_t start = clock();
    init_table(xLen, yLen);
    answer = ed_recursive_memo_alg(xLen, yLen);
    print_answer(2, 3);
    free_table();
    double time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n\n", (time_spent));
    rec_counter = 0;
  }
}


//Smith-Waterman functions
//sw_iterative_alg - iterative algorithm for Smith-Waterman algorithm
int sw_iterative_alg(){
  int i, j, value, max;
  max = 0;
  for (i=0; i <= xLen; i++){
    for (j=0; j <= yLen; j++){
      if ((i == 0) | (j == 0)){
        value = 0;
      }

      else if (x[i-1] == y[j-1]){
        value = table[i-1][j-1].entry + 1;
      }

      else {
        value = max4((table[i-1][j].entry-1), (table[i][j-1].entry-1), (table[i-
1][j-1].entry-1), 0);
      }
    max = max2(value, max);
    add_to_table(i, j, value);
    }
  }
  return max;
}

//sw - calls necessary algorithms and prints
void sw(){
```



```
  if (iterBool){
    printf("Iterative version\n");
    init_table(xLen, yLen);
    clock_t start = clock();
    answer = sw_iterative_alg();
    double time_spent = (double)(clock() - start) / CLOCKS_PER_SEC;
    print_answer(3, 1);
    free_table();
    printf("Time taken: %f seconds\n\n", (time_spent));
  }
}

//NEW FUNCTIONS END

// main method, entry point
int main(int argc, char *argv[]) {
        printf("\n\n"); //Just some whitespace for aesthetics
        bool isIllegal = getArgs(argc, argv); // parse arguments from command li
ne
        if (isIllegal) // print error and quit if illegal arguments
                printf("Illegal arguments\n");
        else {
                printf("%s\n\n", alg_desc); // confirm algorithm to be executed
                bool success = true;
                if (genStringsBool)
                        generateStrings(); // generate two random strings
                else
                        success = readStrings(); // else read strings from file
                if (success) { // do not proceed if file input was problematic
      //CODE START

                        //Call the problem solution
                        if (alg_type==LCS){
                                //Case of Longest Common Substring algorithm
                                lcs();
      }
                        else if (alg_type==ED){
                                //Case of Edit Distance algorithm
                                ed();
      }
                        else if (alg_type==SW){
                                //Case of Highest Scoring Local Similarity algor
ithm
                                sw();
      }

      //CODE END
                        freeMemory(); // free memory occupied by strings
                }
        }
        return 0;
}
```