Strings

Problem 1: (20%) Escriba un programa que realice las siguientes operaciones sobre cadenas de caracteres (strings); no usar memoria dinámica:

a) **int longitud_str(str)**: Función que regrese la longitud de un string `str` (sin contar el caracter nulo);

b) **char *copia_str(dst, src)**: copia el string `src` a `dst`, incluyendo el caracter nulo `\0`; regresa un apuntador a `dst`. **Nota:** La función debe verificar que los strings no estén superpuestos y si hay suficiente espacio en `dst` para copiar `src`; en caso que no, incluir solo hasta donde sea posible almacenar en `dst`;

c) **int compara_str(str1, str2)**: Función que compare lexicográficamente dos strings y regrese un número mayor que, igual a o menor que cero, si `str1` es mayor que, igual a o menor que `str2`. El valor que regrese debe coincidir con la diferencia en magnitud del caracter actual comparado;

d) **int concatena_str(str1, str2, str3)**: Función que concatene tres strings (en el orden dado) separados por un espacio y alamacene en `str1`; debe regresar la nueva longitud de `str1`; **Nota**: La función debe verificar que los strings no estén superpuestos y si hay suficiente espacio en `str1` para almacenar todos los strings; en caso que no, incluir solo hasta donde sea posible almacenar en `str1`;

e) **int encuentra_str(str1, str2)**: Función que busque un sub-string `str1` en `str2` y regrese el número de veces que lo encuentra.

f) **int **frecuencia(str)**: Encuentre la frecuencia de cada elemento de `str`, y regrese un arreglo bidimensional donde se almacene la letra (1er columna) y frecuencia (2da columna). **Salida**: w — 1, r — 5, s — 4, etc... **Nota**: En este ejercicio puede usar memora estática o dinámica.

g) **char *sin_repetir(str)**: Función que encuentre las palabras en `str` que no tengan letras repetidas y las imprima (dentro de la misma función por simplicidad).

Solution (a). El siguiente program, al igual que los otros incisos, se encuentran en el archivo de cabecera `string.h`. Las implementaciones se encuentran en `string.c`. Para ver ejemplos de su uso, compilar mediante `gcc p1.c string.c -o p1`. Ejecutar usando `./p1`.

Listing 1: Programa que calcula la longitu de un string.

```
int longitud_str(const char *str) {
  int len = 0;
  while (*(str++)) {
    len++;
  }
  return len;
}
```

◇

Solution (b).

Listing 2: Programa que copia el string src al string dst.

```
/*

We assume source is a null ending string.
*destiny is any char pointer.
dst_len is the length of destiny,
and we assume that the number passed is consistent
with the memory assigned to it.

 */

char *copia_str(char *destiny, const char *source, int dst_len) {
  char *init = destiny; // store dst's starting memory address

  if (dst_len < longitud_str(source)) {
    fprintf(stderr, "destiny does not have\
    enough space to copy source.\n");
    exit(1);
  }

  if ((destiny < source + longitud_str(source) && destiny > source) ||
      (destiny < source && source < destiny + dst_len)) {
    fprintf(stderr, "Strings overlap in memory. Exiting program.\n");
    exit(2);
  }
  do {
    *destiny = *source;

    // do this while any of them are not the null char. By this
    // line, we are sure that source fits into destiny.
  } while (*(source++) | *(destiny++));
  return init;
}
```

◇

Solution (c).

Listing 3: Programa que compare el primer caracter de diferencia entre dos strings.

```
int compara_str(const char *str1, const char *str2) {
  do {
    if (*str1 > *str2) {
      return *str1 - *str2;
    }
    if (*str1 < *str2) {
      return *str1 - *str2;
    }
    // loop while both are not '\0'. Note that we do not use &&
    // to prevent short-circuiting the statement, allowing us
    // to always move both pointers
  } while (*++str1 & *++str2);
  // if both have been equal until one (or both) reaches the char
  // '\0', we return  the distance of these two last chars
  return *str1 - *str2;
}
```

◇

Solution (d).

Listing 4: Programa que concatena tres strings.

```
int concatena_str(char *str1, const char *str2, const char *str3) {
  // const pointer to start of str1, we will
  // return this after all concatenations are done
  char const *str1_copy = str1;
  // set pointer to null character of str1
  while (*(str1++))
    ;
  str1--; // go back so we can replace null character with space char
  *str1 = ' ';

  str1++;
  do { // set characters on str1 until null char of str2 is reached
    *(str1++) = *(str2++);
  } while (*str2);

  *str1 = ' ';
  str1++;

  do {
    *(str1++) = *(str3++);
  } while (*str3);

  *str1 = '\0'; // end str1 with null character so that
                // we can print it without
                // having memory problems

  // return length using pointer to start of str1
  return longitud_str(str1_copy);
}
```

◇

Solution (e).

Listing 5: Programa que encuentra la cantidad de ocurrencias del string str1 en str2.

```
int encuentra_str(const char *str1, const char *str2) {
  if (*str1 == '\0' ||
      *str2 == '\0') // If either of the strings are empty, return 0
    return 0;
  int occurrences = 0; // occurrence counter
  char const *cpy_str1 = str1;
  // constant pointer to where str1 starts, so we may reset str1
  // pointer after making a character match in the
  // following nested while loop
  while (*str2) {
    // while str2 does not point to the null terminating
    // character, we continue searching for occurrences of str1
    printf("b4 %p\n", str2);

    // this initializes a loop once we find that the first
    // character of str1 matches some character of str2
    while (*str1 == *str2) {
      str1++;
      str2++;
      if (*str1 != '\0' && *str2 == '\0') return occurrences;
      // if we reach the end of str1, we have found one occurrence
      if (*str1 == '\0') {
        occurrences++;
        str2--; // go back a position in str2, as we will add it back when
                // exiting this loop
        str1 = cpy_str1; // reset str1 pointer position for next possible
                         // character match
        if (*str2 == '\0')
          return occurrences;
        break;
      }
    }
    str2++;
  }
  return occurrences;
}
```

◇

Solution (f).

Listing 6: Programa que enlista la frecuencia de todos los caracteres del string str.

```
int **frecuencia(const char *str) {
  if (*str == '\0')
    return 0;

  // assuming 26 lower-case letters a-z (no ~n)
  int **freq = malloc(26 * sizeof(*freq));
  for (int i = 0; i < 26; i++) {
    // allocate space for each letter and its repetition value
    int *char_freq = malloc(2 * sizeof(int));
    *(char_freq) = 'a' + i;  // initializing array, a = 0th index
    *(char_freq + 1) = 0;    // initialize all counts to 0
    *(freq + i) = char_freq; // point freq[i] to each array
  }

  while (*str) {
    // str points to lowercase char
    if (*str - 'a' >= 0 && *str - 'a' < 26)
      *(*(freq + (*str - 'a')) + 1) += 1; // storing it in freq[str - a][1]
    str++;
  }

  return freq;
}
```

◇

Solution (g).

Listing 7: Programa que imprime las palabras de un string que no tienen caracteres repetidos.

```
char *sin_repetir(const char *str) {
  char word[90];    // storage for current word being analyzed.
  char *ptr = word; // pointer to start of array.
                    // This will be the object we
                    //  will be iterating to set the word array contents

  while (*str) {
    *ptr = *str;
    ptr++;
    str++;
    // We enter when a word is formed (that is, a space
    // character is found, or string str has ended).

    if (*str == ' ' || *str == '\0') {
      *ptr = '\0'; // End string so it can be processed by frecuencia function

      // we then reset pointer to initial position, so that the pointer passed
      // in to frecuencia is properly analyzed from start to finish
      ptr = word;
      int **freq = frecuencia(ptr); // char frequency array
      for (int i = 0; i < 26; i++) {
        if (freq[i][1] > 1) // if one char is repeated, end loop
          break;
        if (i == 25) { // if none have been repeated, print word
          printf("%s\n", word);
        }
      }
      for (int i = 0; i < 26; i++) {
        free(freq[i]);
      }
      free(freq);
    }
  }
  return ptr;
}
```

◇

Arreglos Bidimensionales

Problem 2: (10%) Escribir un programa que genere una caminata aleatoria en una matriz de 10x10. El arreglo debe contener inicialmente puntos '.', y debe recorrerse basado en el residuo de un número aleatoria (usar **srand()** y **rand()**) cuyos resultado puede ser 0 (arriba), 1 (abajo), 2 (izq), 3 (der), que indican la dirección a moverse. A) Verificar que el movimiento no se salga del arreglo de la matriz, y B) No se puede visitar el mismo lugar más de una vez. Si alguna de estas condiciones se presenta, intentar moverse hacia otra dirección definida; si todas las posiciones están ocupadas, finalizar el programa e imprimir el resultado.

Solution. El siguiente programa se compila mediante `gcc p2.c -o p2`, y se corre mediante `./p2`. El resultado es un tablero que se actualiza mientras se vaya generando el camino aleatorio. Correrlo de nuevo (con suficiente tiempo de separación) genera una caminata nueva.

Listing 8: Programa que realiza una caminata aleatoria en una matriz.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define N 10
#define M 10

#define x_0 0
#define y_0 0

char board[N][M];

void initialize_board(char board[N][M]) {
  char *p, *END = &board[0][0] + M * N;
  p = &board[0][0];
  int i;

  for (p = board[0], i = 0; p < END; p++, i++) {
    *p = '.';
  }
}

void print_board() {
  printf("\033[H");
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      printf("%c ", board[i][j]);
    }
    printf("\n");
    fflush(stdout);
  }
}

int main() {
  initialize_board(board);
  printf("\033[2J");

  srand(time(NULL));
  char *current = &board[x_0][y_0];
  unsigned char letter = '.' + 1; // avoid '.' char
  *current = letter++; // set first tile to first character, then increment char
                       // for next usage

  char *POS_0M = &board[0][M], *POS_N0 = &board[N][0]; // helper pointers
  short int iter = 0;                                 // iteration count
  // pos_x is used to keep track of column position in matrix of current block.
  // We use this to check if we can go left or right
  short int pos_x = x_0;
  unsigned char direction, blocked = 0;
  /*
  we use 4 bits (half a byte) to describe directions:
  0000 0001 = up
  0000 0010 = down
  0000 0100 = left
  0000 1000 = right
```

```
 blocked represents the directions that have been proven to be blocked
*/

 while (iter < 1000) {
   if (blocked == 15) {
     break; // if blocked == 00001111 (all directions are blocked), exit loop
   }
   iter++;
   direction = 1 << (rand() % 4);
   if (direction & blocked) { // if direction randomly chosen has at least one
                              // bit in common with blocked choices, try again
     continue;
   }
   switch (direction) {
   case 1:
     if (current < POS_0M ||
         *(current - N) != '.') // if current position is in first row or north
                                // character is not a '.', skip
     {
       blocked |= direction; // add blocked direction to blocked variable
       break;
     } else {
       current -= N; // go back N * sizeof(char) = N bytes. This places us +1
                     // position north
       *current = letter++; // set the new current position to letter, then
                            // increment the letter.

       // after moving, we block the square we come from. For
       // example, if we move north (this case), then we block
       // south, which is 0000 0010 (direction << 1)
       blocked = direction << 1;
       print_board();
       usleep(200000);
       break;
     }
   case 2:
     if (current > POS_N0 || *(current + N) != '.') {
       blocked |= direction;
       break;
     } else {
       current += N; // go 1 block south in matrix
       *current = letter++;
       blocked = direction >> 1; // since we moved south, blocked is 0000 0001
       print_board();
       usleep(200000);
       break;
     }
   case 4:
     if (pos_x == 0 || *(current - 1) != '.') {
       blocked |= direction;
       break;
     } else {
       current--;                // go left
       *current = letter++;      // set current position to letter
       blocked = direction << 1; // since we moved left, blocked is 0000 1000
       pos_x--;                  // update column position
       print_board();
```

```
        usleep(200000);
        break;
      }
    case 8:
      if (pos_x == M - 1 || *(current + 1) != '.') {
        blocked |= direction;
        break;
      } else {
        current++;                // go right
        *current = letter++;      // set current char to letter, then update
        blocked = direction >> 1; // since we moved right, blocked is 0000 01000
        pos_x++;                  // update column position
        print_board();
        usleep(200000);
        break;
      }
    default:
      fprintf(stderr, "Invalid direction.\n");
      exit(1);
    }
  }
}
```

◇

Problem 3: (10%)

a) Dado un arreglo bidimensional de enteros $M \times N$, encontrar el máximo valor para cada columna y cada renglón:

	9	6	8	1
8	3	6	8	1
4	2	4	3	1
9	9	5	2	1

Nota: Recorre el arreglo (columnas y renglones) en forma eficiente.

b) Cuente el número de bytes del arreglo bidimensional con valor 0 (recuerde que cada entero está representado por 4 bytes).

Solution. El siguiente programa se compila mediante `gcc p3.c -o p3`, y se corre con `./p3`. Luego, se piden 2 enteros en formato `n,m`. Después, los valores que tomará la matriz, separados por un espacio.

Listing 9: Programa que toma un arreglo e imprime los valores maximos de renglón y columna.

```
#include <limits.h> // used limits.h to have access to INT_MIN for compatibility
    with lower bit machines.
#include <stdio.h>

/*

pretty prints the values of the array a

 */
void print_array(int *a, int n, int m) {
  for (int i = 0; i < n; i++) {
```

```
    for (int j = 0; j < m; j++) {
      printf("%d ", *((a + i * n) + j));
    }
    printf("\n");
  }
}

/*

traverses the array a and stores the max of row i into row_max[i], as well as
storing the max of col j into col_max[j]. This is O(N*M) time complexity with
O(N+M) extra space (arrays row_max and col_max)
Note: if we do not consider the return values as extra space, then we only need
O(1) (curr pointer) extra space.

 */

void max_col_row(int *a, int n, int m, int *row_max, int *col_max) {
  int *curr = a;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (*curr > row_max[i])
        row_max[i] = *curr;
      if (*curr > col_max[j])
        col_max[j] = *curr;
      curr++;
    }
  }
}

/*

traverses the array a and returns the number of zeros found times sizeof(int).
That is, the number of bytes occupied by the number 0 in the array a.

 */

int zeros(int *a, int n, int m) {
  int byte_zero = 0;
  // the following pointers can also be signed
  unsigned char *ptr = (unsigned char *)a; // ptr points to a

  // END pointer is our limiter
  unsigned char *END = (unsigned char *)(a + n * m);
  for (; ptr < END; ptr++) {
    printf("%d\n", *ptr);
    if (*ptr == 0) // since pointer is 1 byte long, we only need to check if
                   // this char is 0
      byte_zero++;
  }
  return byte_zero;
}

int main() {
  unsigned int N, M;

  printf("Enter row and columns formatted as 'N,M':\n");
  scanf("%d,%d", &N, &M);
```

```
  printf("N = %d, M = %d\n", N, M);
  int row_max[N]; // array to store the max of each row
  int col_max[M]; // array to store the max of each column
  for (int k = 0; k < N; k++) {
    row_max[k] =
        INT_MIN; // initialize row_max to negative inf (minimum integer)
  }
  for (int k = 0; k < M; k++) {
    col_max[k] =
        INT_MIN; // initialize col_max to negative inf (minimum integer)
  }

  // array which will be used to store the values entered by the user

  int a[N][M];
  int *p = &a[0][0], // pointer p will be used to initialize array a.
      *END = &a[0][0] + N * M;
  // END is the last + sizeof(int) memory slot to
  // stop the initialization process

  printf("Enter cell values.\n");
  while (p < END) {
    scanf("%d", p++); // assign scanned value to whatever p is pointing to, then
                      // increment it by 1 (that is, sizeof(int))
  }
  // max_col_row((int *)a, N, M, row_max,
  //             col_max); // calculate row and col max, which will be stored in
  //                       // row_max and col_max.
  //
  // print_array(row_max, 1, N); // print row_max values
  // print_array(col_max, 1, M); // print col_max values

  printf("number of zero bytes in array a is %d\n",
         zeros((int *)a, N, M)); // print number of bytes used by the zeros
                                 // entered into the array a
}
```

◇

Memoria Dinámica

Problem 4: (20%) Dado una lista de nombres (string) de N personas (apellido_paterno, apellido_materno y nombre(s)), escribir una función que ordene los nombres alfabéticamente usando un arreglo de apuntadores:

char **crea_arreglo(char **arr, ...)
char **ordena(char **arr,...)

Los nombres pueden tener distinta longitud, pero la memoria que ocupan debe ser la justa (sin desperdicio); cuando un nombre sea prefijo de otro, considerar al nombre más corto como menor. El ordenamiento debe ser a través de una función que reciba el arreglo de apuntadores.

Solution. Para compilar el programa, use `gcc p4.c -o p4` y ejecute con `./p4`. El programa busca el archivo `names.csv`, que contiene una lista de nombres. Luego, se imprime esta misma lista ordenada.

Listing 10: Programa que ordena una lista de nombres.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void print(char **arr, int n) {
  for (int i = 0; i < n; i++) {
    printf("%s\n", *(arr + i));
  }
}

int count_new_line(FILE *file) {
  int c, count = 0;
  while ((c = fgetc(file)) != EOF) {
    if (c == '\n') {
      count++;
    }
  }
  return count;
}

char **create_array(FILE *file, int size) {
  int c, name_length = 0, name_count = 0;
  char **arr;
  arr = malloc((size + 1) * sizeof(*arr));
  rewind(file);

  char buffer[100];

  while ((c = fgetc(file)) != EOF) {
    if (c == '\n') {
      buffer[name_length] = '\0'; // end name with '\0'

      // name_length + 1 to include '\0'
      char *name = malloc((name_length + 1) * sizeof(char));
      strcpy(name, &buffer[0]); // copy stored name in buffer into name pointer
      *(arr + name_count) = name; // arr + name_count now points to name
      name_count++;               // one more name added to list
      name_length = 0;            // reset name_length for next name in csv
    } else {
      buffer[name_length] = c; // store char to buffer
      name_length++;
    }
  }
  fclose(file);
  return arr;
}

char **order(char **arr, int size) {
  char *temp, *compare;
  int str_diff;
  for (int i = 0; i < size; i++) {
    compare = *(arr + i);
    for (int j = i - 1; j >= 0; j--) {
      str_diff = strcmp(*(arr + j), compare);
      // str_diff is positive if rhs precedes lhs, that is,
      // current precedes *(arr+j)
```

```
      if (str_diff <= 0) { // if in order, skip to next i
        break;
      }

      if (str_diff > 0) { // if out of order, swap j and j + 1
        // note that we swap j and j+1 and not j and i, because if we swapped i
        // and j in the previous iteration, we would now be swapping the
        // incorrect pointers.
        temp = *(arr + j + 1);
        *(arr + j + 1) = *(arr + j);
        *(arr + j) = temp;
      }
    }
  }

  return arr;
}

int main() {
  FILE *file;
  file = fopen("names.csv", "r");

  int size = count_new_line(
      file); // count new lines to know how many names are in the csv

  char **names = create_array(file, size); // create array using file
  order(names, size);                      // sort array in place
  print(names, size);                      // print ordered names
  for (int i = 0; i < size; i++) {
    free(names[i]);
  }
  free(names);
  return 0;
}
```

◇

Problem 5: (20%) Separe un string en tokens de acuerdo a un caracter especial dado como entrada (puede ser espacio, /, %, etc.) y que regrese un arreglo que apunte a cada uno de los tokens esperados:

char **tokens(char *str, char ch)

o NULL en caso de no encontrar algún token.

Solution. Este programa se compila mediante `gcc p5.c -o p5` y se ejecuta con `./p5`. Luego, se le pide al usuario que inserte un string de a lo más 256 caracteres. Luego, un caracter que será el token para la tokenización.

Listing 11: Programa que tokeniza un string.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int repetitions(char *str, char ch) {
  int ch_count = 0;
```

```c
  while (*str) {
    if (*str == ch)
      ch_count++;
    str++;
  }
  return ch_count;
}

char **tokenize(char *str, char ch) {
  int ch_rep = repetitions(str, ch); // number of tokens in str

  // make space for ch_rep+1 tokenized words
  char **token_list = malloc((ch_rep + 1) * sizeof(*token_list));
  char temp_token[100]; // buffer to store words up until a token is found
  int token_len = 0,
      token_count = 0; // index functions to keep track of where to store words,
                       // and what char to replace inside of temp_token
  do {
    if (*str == ch || *str == '\0') {
      // once a token is found (or end of string), we store it in a new string
      char *token = malloc(token_len * sizeof(char) + 1);
      temp_token[token_len] = '\0'; // end buffer so it can be copied
      strcpy(token, temp_token); // make copy
      *(token_list + token_count) = token; // point towards newly found word
      token_count++;
      token_len = 0;
    } else {
      temp_token[token_len] = *str; // if no token has been found, keep storing
          chars
      token_len++;
    }
    str++;
  } while (token_count <= ch_rep); // keep going until we find all tokens (+ end
      of string)
  return token_list;
}

int main() {
  char tkn;
  char str[256];

  printf("Enter string to tokenize. (max 256)\n");
  fgets(str, sizeof(str), stdin);
  printf("Enter separating token. (Must be a char)\n");
  scanf("%c", &tkn);
  char **tokenized = tokenize(str, tkn);
  int rep = repetitions(str, tkn);
  printf("Tokenized list:\n");
  for (int i = 0; i < rep + 1; i++) {
    printf("%s\n", *(tokenized + i));
  }
  for (int i = 0; i < rep + 1; i++) {
    free(*(tokenized + i));
  }
  free(tokenized);
}
```

◇

Problem 6: Escriba una función que reciba N arreglos de enteros ordenados de menor a mayor, y mezcle los arreglos en un solo arreglo ordenado de igual forma.
Prototipo de la función: **int *merge(int **arr, int N, int *dim)** donde **arr** tiene la siguiente estructura:
N es el número total de arreglos y **dim** es un arreglo de enteros con la dimensión de cada uno de los arreglos de entrada. La función regresa un apuntador hacia el arreglo mezclado y generado dinámicamente dentro de la función **merge()**.
Nota: Generar dinámicamente todos los arreglos necesarios, e inicialice cada arreglo con valores aleatorios mediante la función **rand()**.

Solution. El siguiente programa se compila mediante `gcc p6.c -o p6` y se ejecuta mediante `./p6`. El programa le pide al usuario insertar el número máximo de filas y columnas que puede tener el arreglo semi-aleatorio ordenado que se generará. Luego, se imprime el arreglo generado, y después de algún tiempo, se imprime el arreglo fusionado, ya ordenado.

Listing 12: Programa que ordena N arreglos dimensiones distintas.

```c
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// #define MAX_ROW_SIZE 100
// #define MAX_COL_SIZE 20

void print(int **arr, int N, int *dim) {
  for (int i = 0; i < N; i++) {
    int curr_dim = *(dim + i);
    for (int j = 0; j < curr_dim; j++) {
      printf("%d ", *(*(arr + i) + j));
    }
    printf("\n");
  }
}

void print_1d(int *arr, int N) {
  for (int i = 0; i < N; i++) {
    printf("%d ", *(arr + i));
  }
  printf("\n");
}

int arr_sum(int *arr, int dim) {
  int result = 0;
  for (int i = 0; i < dim; i++) {
    result += *(arr + i);
  }
  return result;
}

int *merge(int **arr, int N, int *dim) {
  int new_dim = arr_sum(dim, N); // new dimension of merged arrays.
  int minimum = INT_MAX;         // where we keep track of smallest val.

  // array where we store sorted values
  int *merged = malloc(new_dim * sizeof(int));
```

```
  // to keep track of where we are in *(arr + k),
  // init at 0, then keep adding as we traverse *(arr + k).
  int *dim_idx = calloc(N, sizeof(int));
  int row_selected; // keep track of what row had the min value.
  for (int i = 0; i < new_dim; i++) {
    for (int j = 0; j < N; j++) {
      // we check if dim_idx is still within the dimension of *(arr + k),
      // and we also check if that value can be our new minimum.
      if (*(dim_idx + j) < *(dim + j) &&
          *(*(arr + j) + *(dim_idx + j)) < minimum) {
        minimum = *(*(arr + j) + *(dim_idx + j));
        row_selected = j; // keep track that min was found in row j.
      }
    }
    // afer first minimum was found, place it in result array.
    *(merged + i) = minimum;
    minimum = INT_MAX;              // reset minimum for next search.
    *(dim_idx + row_selected) += 1; // update *(row + k) current index, so we do
                                    // not repeat an already chosen value.
  }
  free(dim_idx);
  return merged;
}

int main() {
  int MAX_ROW_SIZE, MAX_COL_SIZE;
  printf("Enter max row size and max col size. Format: 'N,M'\n");
  scanf("%d,%d", &MAX_ROW_SIZE, &MAX_COL_SIZE);
  srand(time(NULL));
  float sTime = (float)clock() / CLOCKS_PER_SEC;
  unsigned int rand_dim = 1 + rand() % MAX_ROW_SIZE; // random number of rows.
  int **arr = malloc(rand_dim * sizeof(*arr));
  int *dim = malloc(rand_dim * sizeof(int)); // dimensions of each row.
  for (int i = 0; i < rand_dim; i++) {
    int rand_arr_size = 1 + rand() % MAX_COL_SIZE;  // random dimension for row.
    *(dim + i) = rand_arr_size;                     // point to array size.
    int *row = malloc(rand_arr_size * sizeof(int)); // make space for row.
    int max_val = 10;
    for (int j = 0; j < rand_arr_size; j++) {
      if (j > 0) {
        *(row + j) =
            *(row + j - 1) +
            rand() % max_val; // next value is in range (prev, prev + 9).
      } else {
        *row = rand() % max_val; // first value is in range (0, 9).
      }
      *(arr + i) = row; // points to row.
    }
  }
  float eTime = (float)clock() / CLOCKS_PER_SEC;
  printf("-----------------------------------------------------------------\n");
  printf("Randomly generated matrix: (execution time: %f ms)\n",
         1000 * (eTime - sTime));
  printf("-----------------------------------------------------------------\n");
  print(arr, rand_dim, dim);

  sTime = (float)clock() / CLOCKS_PER_SEC;
```

```
  int *merged = merge(arr, rand_dim, dim);
  eTime = (float)clock() / CLOCKS_PER_SEC;
  printf("-----------------------------------------------------------------\n");
  printf("Sorted merged array: (execution time: %f ms)\n",
         1000 * (eTime - sTime));
  printf("-----------------------------------------------------------------\n");
  print_1d(merged, arr_sum(dim, rand_dim));

  for (int i = 0; i < rand_dim; i++) {
    free(*(arr + i));
  }
  free(dim);
  free(arr);
  free(merged);

  return 0;
}
```

◇