UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

Programación II
Práctica Calificada 1
Pregrado
2022-II
Rubén Rivas
Lab 106

Indicaciones específicas:

- Esta evaluación contiene 8 páginas (incluyendo esta página) con 3 preguntas. El total de puntos son 20.
- El tiempo límite para la evaluación es 100 minutos.
- Cada pregunta deberá ser respondida incluyendo archivos (.h y .cpp) con el número de la pregunta.
  - `p1.h` / `p1.cpp`
  - `p2.h` / `p2.cpp`
  - `p3.h` / `p3.cpp`
- Deberás subir estos archivos directamente a www.gradescope.com, uno en cada ejercicio. También puedes crear un .zip

Competencias:

- Para los alumnos de la carrera de Ciencia de la Computación

  Aplicar conocimientos de computación y de matemáticas apropiadas para la disciplina. (Evaluar)

  Analizar problemas e identificar y definir los requerimientos computacionales apropiados para su solución.(Usar)

  Utilizar técnicas y herramientas actuales necesarias para la práctica de la computación. (Usar)

- Para los alumnos de las carreras de Ingeniería

  Capacidad de aplicar conocimientos de matemáticas (nivel 3)

  Capacidad de aplicar conocimientos de ingeniería(nivel 2)

  Capacidad para diseñar un sistema, un componente o un proceso para satisfacer las necesidades deseadas dentro de restricciones realistas (nivel 2)

Calificación:

Tabla de puntos (sólo para uso del professor)

Question	Points	Score
1	7	
2	6	
3	7	
Total:	20	

1. (7 points) **Máxima longitud del dígito repetido**
Dado un número entero de cualquier dimension, escribir un programa que permita ubicar la secuencia más larga de un determinado digito, de modo que el programa devuelva la longitud de esa secuencia.

Ejemplo #1: Dado el número: `011100001011000011110`, y el dígito a buscar es `1`, el número contiene las siguientes secuencias de 1's : `111, 1, 11, 11111`
La secuencia más larga es `11111` por tanto, el programa deberá retornar 5.

Ejemplo #2: Dado el número: `012133333413330113333339223`, y el dígito a buscar es `3`, el número contiene las siguientes secuencias de 3's : `33333, 333, 333333, 3`
La secuencia más larga es `333333` por tanto, el programa deberá retornar 6.

Ejemplo #3: Dado el número: `01144411413110444`, y el dígito a buscar es `7`, el número no contiene secuencias de 7's.
En consecuencia, el programa deberá retornar 0.

Casos de uso #1:
Input:

```
011100001011000011110
1
```

Output:

```
5
```

Casos de uso #2:
Input:

```
012133333413330113333339223
3
```

Output:

```
6
```

Casos de uso #3:
Input:

```
01111111111110233
7
```

Output:

```
0
```

La rúbrica para esta pregunta es:

Criterio	Excelente	Adecuado	Mínimo	Insuficiente
Ejecución	El diseño del algoritmo es ordenado y claro, siguiendo buenas prácticas en programación. La ejecución es correcta (`3pts`)	El diseño del algoritmo es ordenado y claro. La ejecución es correcta (`2pts`)	La ejecución es correcta (`1pts`).	La ejecución no es correcta (`0.5pts`)
Sintaxis	No existen errores sintácticos o de compilación (`2pts`)	Existen algunos errores sintácticos de menor relevancia, que no afectan el resultado (`1.5pts`).	Existen errores sintácticos en la forma de ejecución, que no afectan el resultado (`1pts`).	El código tiene errores de sintáxis que afectan el resultado (`0.5pts`).
Optimizacion	El código es óptimo y eficiente. De buen performance e interacción con el usuario (`2pts`)	El codigo es de buen performance durante la ejecución (`1.5pts`).	El codigo no está optimizado pero la ejecución no es deficiente(`1pts`).	El codigo no está optimizado y la ejecución es deficiente (`0pts`).

2. (6 points) **Cálculo de exponenciales**
Desarrollar la función **exponencial** que permita calcular el exponencial de un número utilizando la siguiente ecuación:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ... \tag{1}$$

Continuar la ejecución de la sumatoria mientras: $termino_n > 0$.

$$termino_n = \frac{x^n}{n!} \tag{2}$$

Declaración de función:

```
long double exponential(double x);
```

NOTA: NO SERA CONSIDERADO respuesta válida si se usa de la función `exp` de la libreria `<cmath>`.

Casos de uso #1:

```
cout << fixed << setprecision(5);
cout << exponential(0) << endl;
```

Output:

```
1
```

Casos de uso #2:

```
cout << fixed << setprecision(5);
cout << exponential(1) << endl;
```

Output:

```
2.71828
```

Casos de uso #3:

```
cout << fixed << setprecision(5);
cout << exponential(14) << endl;
```

Output:

```
1202604.28416
```

Casos de uso #2:

```
cout << fixed << setprecision(5);
cout << exponential(23) << endl;
```

Output:

```
9744803446.24890
```

La rúbrica para esta pregunta es:

Criterio	Excelente	Adecuado	Mínimo	Insuficiente
Ejecución	El diseño del algoritmo es ordenado y claro, siguiendo buenas prácticas en programación. La ejecución es correcta (`2pts`)	El diseño del algoritmo es ordenado y claro. La ejecución es correcta (`1.5pts`)	La ejecución es correcta (`1pts`).	La ejecución no es correcta (`0.5pts`)
Sintaxis	No existen errores sintácticos o de compilación (`2pts`)	Existen algunos errores sintácticos de menor relevancia, que no afectan el resultado (`1.5pts`).	Existen errores sintácticos en la forma de ejecución, que no afectan el resultado (`1pts`).	El código tiene errores de sintáxis que afectan el resultado (`0.5pts`).
Optimizacion	El código es óptimo y eficiente. De buen performance e interacción con el usuario (`2pts`)	El codigo es de buen performance durante la ejecución (`1.5pts`).	El codigo no está optimizado pero la ejecución no es deficiente(`1pts`).	El codigo no está optimizado y la ejecución es deficiente (`0pts`).

3. (7 points) **Distancia promedio**
Escribir la función **calculate_average_distance** que reciba como parametros un arreglo de **doubles** y la dimensión del arreglo y que calcule el promedio de las **distancias absolutas** entre 2 valores consecutivos del arreglo.

La **distancia absoluta** entre dos valores es igual al valor absoluto de la diferencia de los 2 valores:

$$distancia = fabs(x_i - x_{i-1}) \tag{3}$$

Declaración de función:

```
double calculate_average_distance(double* arr, int sz);
```

Ejemplo #1: Dado el arreglo: `double arr[4] = {-2, 2, 4, -5}`, las distancias entre los dígitos es la siguiente:

```
entre -2 y  2 la distancia es 4=fabs(-2-2)
entre  2 y  4 la distancia es 2=fabs( 4-2)
entre  4 y -5 la distancia es 9=fabs(-5-4)
```

Por tanto, la suma de distancias es 15 y el promedio es 5.

Casos de uso #1:

```
const int sz = 4;
double arr[sz] = {-2, 2, 4, -5};
cout << fixed << setprecision(1);
cout << calculate_average_distance(arr, sz) << endl;
```

Output:

```
5
```

Casos de uso #2:

```
const int sz = 7;
double arr[sz] = {2, 4, 6, 8, 10, 12, 14};
cout << fixed << setprecision(1);
cout << calculate_average_distance(arr, sz) << endl;
```

Output:

```
2
```

La rúbrica para esta pregunta es:

Criterio	Excelente	Adecuado	Mínimo	Insuficiente
Ejecución	El diseño del algoritmo es ordenado y claro, siguiendo buenas prácticas en programación. La ejecución es correcta (`3pts`)	El diseño del algoritmo es ordenado y claro. La ejecución es correcta (`2pts`)	La ejecución es correcta (`1pts`).	La ejecución no es correcta (`0.5pts`)
Sintaxis	No existen errores sintácticos o de compilación (`2pts`)	Existen algunos errores sintácticos de menor relevancia, que no afectan el resultado (`1.5pts`).	Existen errores sintácticos en la forma de ejecución, que no afectan el resultado (`1pts`).	El código tiene errores de sintáxis que afectan el resultado (`0.5pts`).
Optimizacion	El código es óptimo y eficiente. De buen performance e interacción con el usuario (`2pts`)	El codigo es de buen performance durante la ejecución (`1.5pts`).	El codigo no está optimizado pero la ejecución no es deficiente(`1pts`).	El codigo no está optimizado y la ejecución es deficiente (`0pts`).