Algoritmos y Estructuras de Datos – Primer Parcial

Licenciatura en Tecnología Digital, UTDT

Primer Semestre 2024

- El examen es individual.
- No está permitido comunicarse por ningún medio con otros estudiantes ni con otras personas durante el examen, excepto con los docentes de la materia.
- Puede consultarse a los docentes solo por aclaraciones específicas del enunciado.
- Cada ejercicio debe resolverse en hoja aparte.
- **Cada módulo se aprueba con 60 puntos totales y un mínimo de 20 puntos en cada ejercicio.**

Módulo 1 - Especificación de problemas y Corrección

Problema 1. (50 puntos)
Se desea especificar el juego "La frase del día"; un derivado del *Wordle* en el que el objetivo es descubrir una frase oculta. Dada una frase a adivinar, el jugador tiene que proveer de a una hasta 5 palabras de 5 letras cada una. Las letras de cada palabra provista por el jugador se van "destapando" en la frase hasta que no queda ninguna por descubrir. Lo que se desea conocer es, dada una frase y un conjunto de palabras, qué letras de la frase falta descubrir. Por ejemplo, para la frase `{"la", "vida", "es", "bella"}` y las palabras `{"canto", "poder"}`, restaría descubrir las letras `"lvisb"`.
Especificar el problema *Frase del día*, en el que dado un `vector<vector<char>>` con la frase a adivinar y un `vector<vector<char>>` con entre 1 y 5 palabras de 5 letras, se devuelve un `vector<char>` con las letras restantes a descubrir (sin repetir).

Problema 2. (50 puntos)
Dada la siguiente especificación:

La función `capitalizar` *recibe un* `vector<char>` *que sólo contiene una frase que consiste en letras y espacios, y la* ***capitaliza****, es decir, pone la primera letra de cada palabra en mayúscula y las consecutivas en minúscula, dejando espacios sin modifcar. Por ejemplo, para el input* `" dabale ARROZ a la ZORRA el AbAD"`*, el output correcto sería* `" Dabale Arroz A La Zorra El Abad"`*. Además, devuelve cuántos caracteres debió modificar del texto original.*

`int capitalizar(vector<char>& frase)`
Pre: $frase = frase_0$
$\quad \wedge (\forall j : \texttt{int})\ (0 \leq j < |frase| \rightarrow esLetraOEspacio(frase[j]))$
Post: $|frase| = |frase_0|$
$\quad \wedge (\forall j : \texttt{int})\ (0 \leq j < |frase| \implies frase[j] = capitalizarCaracter(frase_0, j))$
$\quad \wedge\ res = \sum_{j=0}^{|frase|-1} \beta(frase[j] \neq frase_0[j])$

$esLetraOEspacio(c : \texttt{char}) \equiv$
$\quad c = \texttt{' '} \vee \texttt{'a'} \leq c \leq \texttt{'z'} \vee \texttt{'A'} \leq c \leq \texttt{'Z'}$
$capitalizarCaracter(f : \texttt{vector<char>}, j : \texttt{int}) : \texttt{char} \equiv$
$\quad$ *if* $j = 0 \vee f[j-1] = \texttt{' '}$ *then* $mayuscula(f[j])$ *else* $minuscula(f[j])$ *fi*
$mayuscula(c : \texttt{char}) : \texttt{char} \equiv$
$\quad$ *if* $\texttt{'a'} \leq c \leq \texttt{'z'}$ *then* $\texttt{'A'} + c - \texttt{'a'}$ *else* c *fi*
$minuscula(c : \texttt{char}) : \texttt{char} \equiv$
$\quad$ *if* $\texttt{'A'} \leq c \leq \texttt{'Z'}$ *then* $\texttt{'a'} + c - \texttt{'A'}$ *else* c *fi*

Y dado el siguiente programa:

```
1  int res = 0;
2  int i = 0;
3  while (i < frase.size()) {
4    if (frase[i] != ' ') {
5      c = frase[i];
6      if (i == 0 or frase[i-1] == ' ') {
7        if ('a' <= c and c <= 'z')
8           c = c - 'a' + 'A';
9      }
10     else if ('A' <= c and c <= 'Z'){
11          c = c - 'A' + 'a';
12     }
13     if (c != frase[i]) {
14       res = res + 1;
15       frase[i] = c;
16     }
17   }
18   i = i + 1;
19 }
```

(a) Especificar el ciclo dando unas P_c, Q_c, I y fv adecuadas.

(b) Probar que $Pc \Rightarrow I$.

(c) Probar que $I \wedge \neg B \Rightarrow Q_c$.

(d) Probar que $I \wedge fv \leq 0 \Rightarrow \neg B$.

Módulo 2 - Recursión y complejidad

Problema 1. (50 puntos)
Se quiere implementar la función

```
vector<char> invertir(const vector<char> & v)
```

que debe generar un vector nuevo con los mismos elementos que `v`, pero en orden inverso. Por ejemplo, `invertir({'a', 'f', 'u', 'z'})` debe devolver `{'z', 'u', 'f', 'a'}`.

(a) Dar una implementación de `invertir` que utilice un único llamado recursivo.

(b) Dar una implementación de `invertir` que utilice la técnica *Divide and Conquer*, partiendo los casos recursivos en 2 subproblemas de aproximadamente el mismo tamaño.

(c) Graficar utilizando un árbol de recursión el seguimiento de la función dada en (b) al invocarse `invertir({'a', 'f', 'u', 'z'})` desde una función `main`. Tener en cuenta:

- el árbol debe tener `main` como nodo raíz
- se deben poder visualizar todos los llamados a `invertir`, y cualquier función auxiliar introducida
- por cada llamado a una función, se deben poder visualizar sus argumentos y su valor de retorno

En cada caso, si necesita cambiar los parámetros de la función debe hacerlo en una función auxiliar y **mostrar cómo se llama a esa función auxiliar desde la función `invertir` original.**

Problema 2. (50 puntos)
Dada la siguiente función *Divide and Conquer*:

```
int potencia(int a, int b) {
    if (b == 0) {
        return 1;
    }
    int res = potencia(a, b / 2);
    if (b % 2 == 1) {
        return res * res * a;
    }
    else {
        return res * res;
    }
}
```

(a) Identificar el tamaño de entrada n que determina el costo temporal de la función en el peor caso y escribir la recurrencia $T(n)$ que calcula dicho costo.

(b) Dibujar el árbol de recursión asociado a $T(n)$.

(c) En base al árbol de recursión, reescribir $T(n)$ de forma cerrada (no recurrente) y mostrar (usando álgebra de órdenes o aplicando la definición de O) que $T(n) \in O(f(n))$ para alguna $f(n)$ adecuada.

Aclaración: para los fines de este problema, asumiremos que los parámetros de entrada a y b pueden ser arbitrariamente grandes (es decir, que no están acotados por una constante).