Ejercitación Nº 1: Complejidad

1) Suponiendo el siguiente fragmento de código:

```
bool buscar_elemento(int arreglo[], int tamanio, int elemento) {
    bool existe = false;
    for (int i = 0; i < tamanio; i++)
        if (arreglo[i] == elemento)
            existe = true;
    return existe;
}
```

¿Cuál es la función del fragmento de código?

Verificar si existe un elemento dado dentro de un arreglo de enteros.

¿Cuáles son las entradas al programa que afectan la complejidad temporal? Calcular la complejidad temporal

Analizando el código para detectar las entradas al programa que impactan sobre el costo del mismo, se puede ver que la variable `tamanio` acota la complejidad temporal del ciclo implementado para la búsqueda. Por lo tanto, la complejidad se puede expresar de la siguiente manera:

$$T(tamanio) <= C_1 + \sum_{i=0}^{tamanio-1} C_2 = C_1 + tamanio.C_2$$

$T(tamanio) \in O(tamanio)$

¿Cómo se puede "medir" el costo de este programa?

A la hora de probar nuestros programas, un concepto muy importante es el de métrica. Una métrica es cualquier medida destinada a conocer o estimar alguna característica de un programa.

En este caso, podemos analizar la complejidad de nuestro programa midiendo el número de operaciones básicas realizadas. Para el ejemplo, la operación básica es la comparación de un elemento del arreglo con el elemento buscado. Esta métrica está directamente relacionada con la complejidad temporal calculada anteriormente. Para esto, se puede modificar el código de la siguiente forma:

```
bool buscar_elemento(int arreglo[], int tamanio, int elemento) {
    int cont = 0;
    bool existe = false;
    for (int i = 0; i < tamanio; i++) {
        cont++;
        if (arreglo[i] == elemento)
            existe = true;
    }
    cout << "costo de búsqueda: " << cont << " comparaciones." <<
endl;
    return existe;
}
```

Ejemplos

Podemos probar con 2 ejemplos concretos, para ver los resultados:

Ejemplo 1:

```
int arreglo[10] = {2, 3, 10, 4, 1, 5, 7, 10, 17, 8};
int elemento = 10;
```

Resultado:

```
El elemento existe. Costo de búsqueda: 10 comparaciones.
```

Ejemplo 2:

```
int arreglo[10] = {2, 3, 10, 4, 1, 5, 7, 10, 17, 8};
int elemento = 9;
```

Resultado:

```
El elemento no existe. Costo de búsqueda: 10 comparaciones.
```

¿Se puede mejorar este código?

Una mejora que se puede hacer a este código es la siguiente:

```
    bool buscar_elemento(int arreglo[], int tamanio, int elemento) {
        int cont = 0;
        bool existe = false;
        for (int i = 0; !existe && (i < tamanio); i++) {
            cont++;
            if (arreglo[i] == elemento)
                existe = true;
        }
        cout << "costo de búsqueda: " << cont << " comparaciones." <<
endl;
        return existe;
    }
```

Como se puede ver, cómo sólo nos interesa verificar si existe un elemento (y no cuántas veces existe, por ejemplo) podemos dejar de recorrer el arreglo una vez encontrado.

Si volvemos a probar los ejemplos, los resultados ahora serían.

Ejemplo 1:

```
int arreglo[10] = {2, 3, 10, 4, 1, 5, 7, 10, 17, 8};
int elemento = 10;
```

Resultado:

```
El elemento existe. Costo de búsqueda: 3 comparaciones.
```

Ejemplo 2:

```
int arreglo[10] = {2, 3, 10, 4, 1, 5, 7, 10, 17, 8};
int elemento = 9;
```

Resultado:

```
El elemento no existe. Costo de búsqueda: 10 comparaciones.
```

Con lo cual, ha mejorado la situación para ciertos casos, es decir, cuándo se encuentra el elemento antes del final del arreglo. Sin embargo, la complejidad temporal sigue acotada por el tamaño del arreglo, ya que la función `big-oh` analiza el peor caso posible, que igual sigue siendo recorrer el arreglo hasta el final.

Si se cuenta siempre con un arreglo ordenado ¿Se implementaría la misma solución?

Facilmente se puede ver que, si el arreglo está ordenado, se pueden aprovechar las características de los arreglos para realizar una búsqueda por mitades. Es decir, partiendo el arreglo por la mitad solamente nos quedamos con la porción del mismo dónde podría existir el elemento.

```
    bool buscar_elemento(int arreglo[], int izq, int der,
                                                int elemento, int & cont) {
    if (der == izq) {
        if (arreglo[der] == elemento)
            return true;
        else
            return false;
    } else {
        int medio = (der + izq ) / 2;
        if (arreglo[medio] == elemento)
            return true;
        else if (elemento < arreglo[medio]) {
            cont++;
            return buscar_elemento(arreglo, izq, medio - 1,
                                                    elemento, cont);
        } else {
            cont++;
            return buscar_elemento(arreglo, medio + 1, der,
                                                    elemento, cont);
        }
    }
}
```

¿Cuáles son las entradas al programa que afectan la complejidad temporal? Calcular la complejidad temporal

La función `buscar_elemento` recibe como argumentos un arreglo y los índices `izq` y `der`, que determinan la parte del arreglo a buscar delimitada por esos índices. En el primer llamado se buscará el elemento en `arreglo[izq,...,der]`. Considerando a `n` como la cantidad de elementos del arreglo entre `izq` y `der`; luego, al particionar el arreglo en `arreglo[izq,...,medio-1]` y `arreglo[medio+1,...,der]` cada porción contendrá `n / 2` elementos. Este llamado recursivo, reduciendo a la mitad los elementos del arreglo, continuará hasta que `izq` y `der` delimiten un solo elemento.

Nota: Para simplificar los cálculos se considera a n potencia de 2 y se supone que la elección en el if va a ser para la rama que contenga n / 2 elementos.

$$T(n) <= \begin{cases} C_0 & \text{si } n = 1 \\ C_1 + T(n / 2) & \text{si } n > 1 \end{cases}$$

1º iteración $T(n) <= C_1 + T(n / 2)$

2º iteración $T(n) <= C_1 + C_1 + T(n / 2 / 2)$

iº iteración $T(n) <= i * C_1 + T(n / 2^i)$

Si suponemos que $n / 2^i = 1$, para el caso de corte cuando hay sólo un elemento (es decir, `izq == der`), entonces $\log_2 n = i$, quedando:

$T(n) <= \log_2 n + C_0$

$T(n)$ $O(\log_2 n)$

Ejemplos

Podemos probar con los 2 ejemplos que utilizamos para la implementación de la búsqueda anterior, pero ordenando primero los elementos del arreglo de forma creciente:

Ejemplo 1:

```
int arreglo[10] = {1, 2, 3, 4, 5, 7, 8, 10, 10, 17};
int elemento = 10;
```

Resultado:

```
El elemento existe. Costo de búsqueda: 1 comparación.
```

Ejemplo 2:

```
int arreglo[10] = {1, 2, 3, 4, 5, 7, 8, 10, 10, 17};
int elemento = 9;
```

Resultado:

```
El elemento no existe. Costo de búsqueda: 3 comparaciones.
```