UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación

Algoritmos y Estructuras de Datos I

Taller de Searching y Complejidad

1. Introducción

En este taller vamos a implementar algoritmos de búsqueda y en algunos casos vamos a medir sus tiempos de cómputo con la intención de obtener intuición respecto de su complejidad temporal. Recuerden que cuando se habla en términos de complejidad lo que se está haciendo es señalar de qué depende principalmente el número de operaciones que realiza, e ignorar todos los elementos de "menor relevancia".

Por ejemplo, un algoritmo que hace $2n^3+n^2+5$ operaciones en peor caso, y otro que hace $\frac{1}{100}n^3$, en ambos casos decimos que tiene complejidad cúbica, $O(n^3)$. Claramente no hacen la misma cantidad de operaciones. Pero en ambos la eficiencia depende aproximadamente $c.n^3$ milisegundos en terminar en el peor caso, con c una constante que no depende de n.

$$2n^3 < 2n^3 + n^2 + 5 < 4n^3 \implies 2n^3 + n^2 + 5 \approx c\,n^3 \qquad \text{(en el peor caso)}$$

En este taller, a diferencia de los anteriores, nos interesa particularmente resolver los ejercicios mediante algoritmos eficientes. Por eso pedimos no usar búsqueda lineal o cualquier algoritmo que tenga que recorrer toda la secuencia en caso de que exista una alternativa posible. Y además vamos a medir tiempos para estimar la constante c que se ignora cuando hablamos en términos de complejidad, que es muy difícil conocer en términos analíticos por la cantidad de capas de abstracción que hay entre el código que escribimos y lo que la computadora realmente hace al final de cuentas.

Para los ejercicios de complejidad recomendamos utilizar la función `construir_vector` provista por la cátedra. Esta función recibe un entero n y un `string` *disposicion* y devuelve un vector de n elementos en la disposición especificada (opciones para disposición: "asc", "desc", "azar", "iguales").

1. Dado un vector v **ordenado** devolver el índice donde se encuentra el elemento x. De no encontrarse, devolver -1.

   a) Implementar el algoritmo *búsqueda binaria*

   ```
   int busquedaBinaria(vector<int> v, int x);
   ```

   b) Implementar el algoritmo *jump search*

   ```
   int busquedaJumpSearch(vector<int> v, int x);
   ```

2. Medir el tiempo en milisegundos de búsqueda binaria y jump search y completar la siguiente tabla.

	$O(log(n))$	$O(\sqrt{n})$
n=100		
n=1000		
n=10000		
n=100000		

   a) ¿Cambia el valor encontrado dependiendo el valor de los elementos del vector?

   b) ¿Cuál cree que es puede ser el valor de c?

3. Realizar simulaciones para estimar la complejidad temporal de las siguientes operaciones sobre vectores en C++. Pueden ayudarse con una tabla similar al ejercicio precedente, haciendo variar el tamaño del vector.

   a) `v.size()`

   b) `v.push_back(e)`

   c) `v.pop_back()`

   d) `v[i]` (¿cambia la complejidad estimada según el valor de `i`?)

   e) `v[i] = e;` (¿cambia la complejidad estimada según `i`?)

   f) `v.flip()` (sólo para `v` de tipo `vector<bool>`)

   g) `v.clear()`

   h) `v.empty()`

4. Dado un arreglo encontrar *el índice* de algún pico. Un pico es un elemento que no es menor a ninguno de sus dos vecinos. Para ser un pico en los bordes alcanza con compararlo con su único vecino o con ninguno si hay un solo elemento. En caso de haber más de uno, devolver cualquiera de ellos.

```
int indicePico(vector<int> v);
```

5. Dado un arreglo ordenado ascendentemente y sin repetidos, encontrar una posición i tal que $v[i] = i$. Si hay más de una posición, devolver la menor. Devolver -1 en caso de que tal índice no exista.

```
int puntoFijo(vector<int> v);
```

6. Encontrar un elemento y devolver su índice en un arreglo ordenado ascendentemente que está rotado k posiciones. En caso de no encontrarse, devolver -1. Asumir que todos los elementos del array son distintos.

```
int encontrarRotado(vector<int> v, int x);
```

7. Dado un arreglo ordenado en orden creciente y sin repetidos, encontrar el índice del menor elemento que sea mayor al pasado por parámetro. Devolver -1 si tal elemento no existe.

```
int menorMasGrande(vector<int> v, int x);
```

8. Dado un arreglo ordenado y dos enteros k y x, devolver los primeros k números más cercanos (por valor absoluto) a x, en orden creciente. En caso de empate, elegir el menor valor. Si hay $l < k$ elementos cercanos, devolver un vector de l elementos. Si x está en el arreglo, no considerarlo como output.

```
vecotr<int> masCercanoK(vector<int> v, int k,  int x);
```