UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 05: Búsquedas

ALUMNOS:

Castañon Maldonado Carlos Emilio
Nepomuceno Escarcega Arizdelcy Lizbeth

PROFESOR

María de Luz Gasca Soto

AYUDANTES

Brenda Margarita Becerra Ruíz
Enrique Ehecatl Hernández Ferreiro (Link)

ASIGNATURA

Análisis de Algoritmos

1 Consideremos el siguiente juego, entre dos personas:
El jugador J_A piensa un número entero en un rango.
El jugador J_B intenta encontrar tal número haciendo preguntas de la forma:

¿Es el número menor que x? o ¿Es mayor que y?

El objetivo es realizar el menor número de preguntas.
Se supone que nadie hace trampa.

(a) Diseñar una buena estrategia para el juego... cuando el jugador J_A indica un rango específico, digamos de 1 a N.

Algoritmo Adivina: Como entrada tenemos un número de inicio i, un número final, digamos n y un número a buscar x con $i \leq x \leq n$. La salida es el número que queremos encontrar.

1. Calculamos $j = \lfloor \frac{|n-i|}{2} \rfloor + i$
2. Si la secuencia tiene un único elemento digamos z, preguntamos si $x = z$, entonces, regresamos z.
3. Si la secuencia tiene dos elementos z, y, preguntamos si $x = y$, en ese caso regresamos y; en otro caso regresamos z.
4. Si $x \leq j$, entonces aplicamos $Adivina(i, j, x)$, en otro caso, $Adivina(j, n, x)$.

En este caso, ¿Cuál resulta ser la complejidad del algoritmo? **Justificar**
Podemos ver que el algoritmo es una modificación del algoritmo **Búsqueda Binaria**, además la complejidad de dicho algoritmo es $O(log(n))$, por lo que la complejidad de nuestro algoritmo es $O(log(n))$.

(b) Diseñar una buena estrategia para el juego... cuando el jugador J_A **no** indica el rango del número que pensó.
Algoritmo principal: Como entrada tenemos un número x. La salida es el número que queremos encontrar.

1. Si $x \geq 0$, entonces vamos al paso 2, en otro caso vamos al paso 3.
2. **Búsqueda lineal** $(x, 0)$.
3. **Búsqueda LinealN** $(x, -1)$.

Búsqueda lineal: Tenemos como entrada dos números x, y. La salida es el número que queremos encontrar.

1. Si $x = y$ regresamos y; en otro caso ejecutamos **Búsqueda Lineal**$(x, y + 1)$.

Búsqueda LinealN: Tenemos como entrada dos números x, y. La salida es el número que queremos encontrar.

1. Si $x = y$ regresamos y; en otro caso ejecutamos **Búsqueda LinealN**$(x, y - 1)$.

¿Cuál es la complejidad del algoritmo propuesto? **Justificar**
Notemos que los algoritmos **Búsqueda Lineal** y **Búsqueda LinealN** tiene complejidad $O(n)$, y el **Algoritmo principa** ejecuta solo uno de los dos algoritmos anteriores, por lo tanto la complejidad del **Algoritmo principal** es $O(n)$.

2 Dada un arreglo de n enteros, $A[0...n-1]$, tal que $\forall i, 0 \leq i \leq n$, se tiene que:
$|A[i] - A[i+1]| \leq 1$; si $A[0] = x$ y $A[n-1] = y$, se tiene que $x < y$.

Diseñar un algoritmo de búsqueda eficiente, de orden logarítmico, que localice al índice j tal que:
$A[j] = z$, para un valor dado de z, $x \leq z \leq y$. **Justificar.**

Antes de empezar notemos que A es un arreglo ordenado de numeros desde 0 hasta $n-1$, la diferencia entre un elemento y el siguiente elemento a este, es de a lo mas de 1, además, por esto mismo y que $\forall i, 0 \leq i \leq n$, notemos que podemos tener elementos repetidos.

Con esto en mente podemos notar que podríamos tener arreglos como los siguientes:

$$\{1, 0, -1, 0, 1, 2, 3, 4, 5, 5, 6\}$$

$$\{1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 3\}$$

Ahora, recordando que nuestro objetivo es localizar el índice j tal que; $A[j] = z$ con $x \leq z \leq y$.
Tendremos que nuestra mejor opción será una versión modificada de Búsqueda Binaria para cumplir con el requerimiento de $A[j] = z$ y el que el algoritmo sea logarítmico.

Estrategia:
Pre Condiciones: El arreglo recibido cumple con que sus elementos sean n enteros, $A[0...n-1]$, tal que $\forall i, 0 \leq i \leq n$, tales que $|A[i] - A[i+1]| \leq 1$; si $A[0] = x$ y $A[n-1] = y$, se tiene que $x < y$.
El $A[j] = z$ a buscar cumple con $x \leq z \leq y$.

Post Condiciones: Lo único que se regresa es el $A[j] = z$ y un mensaje sobre que si se encontró, en su defecto, se despliega un mensaje diciendo que no se encontró.

1) Declaramos dos variables, inicio y fin las cuales serán:
   $inicio = 0 \qquad fin = n - 1$
2) Mientras inicio sea menor o igual a fin, repetir el siguiente proceso:
   a) Calcular el índice medio del subarreglo como $medio = (inicio + fin)/2$.
   b) Comparar $A[medio]$ con $A[j] = z$:
   Si $A[medio]$ es igual a $A[j] = z$, hemos encontrado el valor $A[j] = z$ y podemos devolver medio como la posición de $A[j] = z$ en el arreglo.
   Si $A[medio]$ es menor que $A[j] = z$, actualizamos inicio a $medio + 1$ para buscar en la mitad derecha del subarreglo.
   Si $A[medio]$ es mayor que $A[j] = z$, actualizamos fin a $medio - 1$ para buscar en la mitad izquierda del subarreglo.
3) Si llegamos al punto donde inicio es mayor que fin y no hemos encontrado $A[j] = z$, entonces $A[j] = z$ no está presente en el arreglo.

Justificación:
Hemos escogido usar una variante de búsqueda binaria ya que esta nos permite manejar de una manera rápida e eficiente nuestro $A[j] = z$ buscado, además de que al ser una variante de Búsqueda Binaria, su complejidad es de $O(\log n)$ y podemos cumplir con la complejidad logarítmica requerida.

3 Supongamos que se tiene un programa que manipula textos muy muy grandes, como un procesador de palabras.
El programa toma como entrada un texto, representado como una secuencia de carácteres y produce alguna salida.
Si en algún momento el programa encuentra un error del cual no puede recuperarse y además no puede indicar qué error es o dónde está, entonces la única acción que el programa toma es escribir *ERROR* y **abortar** el proceso.
Supongamos que el error es local, esto es, se tiene una cadena en particular del texto la cual, por alguna extraña razón, al programa no le gusta.
El error es independiente del contexto en el cual aparece la cadena "ofensiva".
Diseñar una estrategia logarítmica para localizar la fuente del error.

Estrategia:

Pre Condiciones: La entrada es un texto, representado como una secuencia de carácteres.

Post Condiciones: En caso de existir un error, el programa devuelve la cadena que ofensiva que causo el error, en caso contrario no devuelve nada.

Dado que tenemos una entrada de la cual desconocemos su longitud y un error el cual es independiente del contexto, podemos diseñar una estrategia logarítmica basándonos en búsqueda binaria y búsqueda secuencial.

1) Primero guardamos la cadena total de palabras en un arreglo de la siguiente manera; tomaremos como elementos a las cadenas de caracteres y los "espacios" entre estos como la separación de cada elemento, teniendo de esta forma algo parecido a esto:
   $cadena =$ "$cadena1$ $soylaCadena2$ ... $ultimaCadena$"
   $A\{cadena1, soylaCadena2, ..., ultimaCadena\}$
2) Ahora dividimos al texto en dos partes iguales.

3) Ahora que tenemos nuestros dos sub-arreglos, procedemos a ejecutar búsqueda secuencial sobre el primer sub-arreglo, buscando elemento por elemento aquella cadena de caracteres que no cumpla con ser una cadena de caracteres valida para el programa, si la encontramos hemos terminado, en caso contrario buscamos en el segundo sub-arreglo.

Justificación:
Debido a que estamos manejando una caja negra y que no sabemos donde o que es el elemento que esta fallando, hemos escogido la combinación de búsqueda binaria y de búsqueda secuencial para poder garantizar el que encontremos el error, ahora con la complejidad logarítmica esta la tenemos garantizada ya que estamos dividiendo el arreglo original en dos sub-arreglos, sin embargo, al aplicar búsqueda secuencial en los dos sub-arreglos restantes, esta aumenta en n, quedándonos de esta forma, una complejidad de $O(n \log n)$ la cual sigue siendo logarítmica.

4 Consideremos el Algoritmo de Búsqueda por Interpolación.

1. Presentar un ejemplo de al menos 400 datos para el cual el algoritmo termine la búsqueda (exitosa o no) en pocas iteraciones. `Ejemplificar`.
   Sea $S = \{1, 2, 3, \ldots, 400\}$ y $z = 50$ el número que vamos a buscar, aplicamos el algoritmo, el cuál en la primera iteración calculamos *pos* con la fórmula de la función de la interpolación lineal:

$$pos = izq + \left[\frac{(z - S[izq])(der - izq)}{S[der] - S[izq]}\right]$$

$$pos = 1 + \left[\frac{(50 - S[1])(400 - 1)}{S[400] - S[1]}\right]$$

$$= 1 + \left[\frac{(49)(\cancel{399})}{\cancel{399}}\right] = 1 + 49 = 50$$

   Notemos que $S[pos] = 50$ ya que $S[50] = 50$, por lo tanto, encontramos z en una sola iteración.

2. Dar un ejemplo de, al menos, 400 datos para el cual el algoritmo termine la búsqueda (exitosa o no) en muchas iteraciones. `Ejemplificar`.
   Ejemplo:

```
1  public class Interpol{
2
3  public static void main(String[] args){
4    //Secuencia en donde buscaremos
5    int[] arr = {
6            1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
7            11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
8            21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
9            31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
10           41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
11           51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
12           61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
13           71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
14           81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
15           91, 92, 93, 94, 95, 96, 97, 98, 99, 100,
16           200, 300, 400, 500, 600, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709,
17           710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724,
                 725,
18           726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740,
                 741,
19           742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756,
                 757,
20           758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772,
                 773, 774,
21           775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789,
                 790, 791,
22           792, 793, 794, 795, 796, 797, 798, 799, 800, 900,900, 902, 904, 906, 908,
                 910, 912, 914,
23           916, 918, 920, 922, 924, 926, 928, 930, 932, 934, 936, 938, 940, 942, 944,
                 946, 948, 950,
```

```
24            952, 954, 956, 958, 960, 962, 964, 966, 968, 970, 972, 974, 976, 978, 980,
                  982, 984, 986,
25            988, 990, 992, 994, 996, 998, 1000, 1002, 1004, 1006, 1008, 1010, 1012,
                  1014, 1016, 1018,
26            1020, 1022, 1024, 1026, 1028, 1030, 1032, 1034, 1036, 1038, 1040, 1042,
                  1044, 1046, 1048,
27            1101, 1203, 1304, 1409, 1505, 1602,1700, 1800, 1900, 2000, 2100, 2200, 2300,
                   2400, 2500, 2600,
28            2700, 2800, 2900, 3000,
29            3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000,
30            4100, 4200, 4300, 4400, 4500, 4600, 4700, 4800, 4900, 5000,
31            5100, 5200, 5300, 5400, 5500, 5600, 5700, 5800, 5900, 6000,
32            6100, 6200, 6300, 6400, 6500, 6600, 6700, 6800, 6900, 7000,
33            7100, 7200, 7300, 7400, 7500, 7600, 7700, 7800, 7900, 8000,
34            8100, 8200, 8300, 8400, 8500, 8600, 8700, 8800, 8900, 9000,
35            9100, 9200, 9300, 9400, 9500, 9600, 9700, 9800, 9900, 10000,
36            11000, 12000, 13000, 14000, 15000, 16000, 17000, 18000, 19000, 20000,
37            21000, 22000, 23000, 24000, 25000, 26000, 27000, 28000, 29000, 30000,
38            31000, 32000, 33000, 34000, 35000, 36000, 37000, 38000, 39000, 40000,
39            4100, 42000, 43000, 44000, 45000, 46000, 47000, 4800, 49000, 50000,
40            51000, 52000, 53000, 54000, 55000, 56000, 57000, 58000, 59000, 60000,
41            61000, 62000, 63000, 64000, 65000, 66000, 67000, 68000, 69000, 70000,
42            71000, 72000, 73000, 74000, 75000, 76000, 77000, 78000, 79000, 80000,
43            81000, 82000, 83000, 84000, 85000, 86000, 87000, 88000, 89000, 90000,
44            91000, 92000, 93000, 94000, 95000, 96000, 97000, 98000, 99000,
                  100000,100001, 100002,
45            100003, 100004, 100005, 100006, 100007, 100008, 100009, 100010, 100011,
                  100012, 100013,
46            100014, 100015, 100016, 100017, 100018, 100019, 100020, 100021, 100022,
                  100023, 100024,
47            100025, 100026, 100027, 100028, 100029, 100030, 100031, 100032, 100033,
                  100034, 100035,
48            100036, 100037, 100038, 100039, 100040, 100041, 100042, 100043, 100044,
                  100045, 100046,
49            100047, 100048, 100049, 100050, 100051, 100052, 100053, 100054, 100055,
                  100056, 100057,
50            100058, 100059, 100060, 100061, 100062, 100063, 100064, 100065, 100066,
                  100067, 100068,
51            100069, 100070, 100071, 100072, 100073, 100074, 100075, 100076, 100077,
                  100078, 100079,
52            100080, 100081, 100082, 100083, 100084, 100085, 100086, 100087, 100088,
                  100089, 100090,
53            100091, 100092, 100093, 100094, 100095, 100096, 100097, 100098,
                  100099,100100,100101,100102,
54            100103,100104,100105,100106,100107,100108,100109,100110,100111
55        };
56
57        // extremo izquierdo de la secuencia
58        int izq = 0;
59        // elemento a buscar
60        int z = 1050;
61        //elemento en la primera posicion de la secuencia
62        int sIzq = 1;
63        //extremo derecho de la secuencia
64        int der = 399;
65        //elemento final de la secuencias
66        int sDer = 100111;
67        // contador que sirve patra contar el numero de itereaciones
68        int cont = 0;
69        //declaramos pos para usarlo dentro del while
70        int pos = -1;
71        //variable auxiliar para que no se cicle el programa
72        int aux = 0;
73
74
75        //Algoritmo de busqueda de interpolacion
76        while((arr[der] >= z && arr[izq] < z)){
77          if(cont >= 400){
78            break;
79          }else{
80           // Formula de interpolacion para calcular la posicion
81            pos = (int)(izq + Math.ceil( ((z-sIzq)*(der-izq)) / (sDer-sIzq)) );
```

```
            // Comprobacion para evitar un ciclo infinito
            if(aux == pos){
            // Comprueba si el valor de 'aux' es igual al valor de 'pos'
              break;
            }else{
            // Si 'aux' no es igual a 'pos', actualiza 'aux' con el valor de 'pos'
              aux = pos;
              if(z > arr[pos]){
              // Si el valor buscado 'z' es mayor que el valor en 'arr[pos]',
              // actualiza 'izq' para reducir el rango de busqueda.
                izq = pos;
              }else if(z < arr[pos]){
              // Si el valor buscado 'z' es menor que el valor en 'arr[pos]',
             // actualiza 'der' para reducir el rango de busqueda.
                der = pos-1;
              }else{
             // Si 'z' es igual al valor en 'arr[pos]', actualiza 'izq' para continuar
                 buscando
                izq = pos;
              }
            }

          }
          // Incrementamos el contador de iteraciones
          cont++;
        }
       // Imprime el numero de iteraciones realizado
        System.out.println("El numero de itereaciones es: " + cont);
        if(arr[izq] == z){
        // Verifica si se encontro el elemento y muestra la posici n si es el caso
          System.out.println("Esta en la posicion: " + izq);
        }else{
          System.out.println("No se encontro...");
        }
      }
}
```

```
El numero de itereaciones es: 157
No se encontro...
```

5 **X. opcional.** Se tiene un conjunto de N rocas, todas ellas de diferente tamaño, forma y consistencia. Rocas que se ven del mismo tamaño pueden tener peso muy diferente.
Un experto desea convencer a un jurado que una roca especial, de entre las N rocas dadas, es la de menor peso. El experto **sí** sabe los pesos de cada una de las rocas y el Jurado **no**. El experto quiere mostrar que la roca especial es la de menor peso usando una balanza menos de $N-1$ veces. ¿Eso es posible? Justifica.

Primero meteremos todas las piedras en un arreglo de forma aleatoria, donde los índices del arreglos están en el rango de $0 \leq i \leq n-1$.

Busca Piedra Entrada: un arreglo A de piedras de tamaño n. Salida: la piedra de menor peso.

1. Si A tiene un solo elemento regresamos dicho elemento, en otro caso vamos al paso 2.
2. Si n es par entonces vamos al paso 3, en otro caso vamos al paso 4.
3. Para cada índice i del arreglo con $0 \leq i \leq n-1$ con i par. Comparamos el elemento del índice i con el elemento del índice $i+1$, y añadimos al elemento de peso más pequeño a un nuevo arreglo A', y ejecutamos **Busca Piedra**(A').
4. Añadimos el elemento del índice 0 a un nuevo arreglo A' y para casa índice i del arreglo con $1 \leq i \leq n-1$ con i impar. Comparamos el elemento del índice i con el elemento del índice $i+1$, y añadimos al elemento de peso más pequeño al arreglo A', y ejecutamos **Busca Piedra**(A').

Justificación:

1. Notemos que el caso en el que el algoritmo **Busca Piedra** usa más comparaciones es cuando el tamaño del arreglo es potencia de dos, ya que cuando el arreglo tiene tamaño impar, hay siempre un elemento que no se compara.
2. Notemos que para los arreglos con tamaño potencia de dos, la primera iteración usará $\frac{n}{2}$ comparaciones, ya que una comparación utiliza dos elementos y ningún elemento se compara más de una vez. Además el tamaño del nuevo arreglo A' será $\frac{n}{2}$. por lo que al aplicar **Busca Piedra**(A') usará $\frac{n}{4}$ comparaciones y el tamaño del siguiente arreglo será $\frac{n}{4}$, podemos repetir esto hasta llegar a un arreglo de solo un elemento, por lo que el número de comparaciones se puede expresar como $\sum_{i=1}^{\infty} \frac{n}{2^i}$, lo cuál es una suma convergente, esto quiere decir que la suma se acerca a n, pero la suma nunca llegará a n, por lo tanto en número total de comparaciones es menor o igual a $n - 1$.