**ESTRUCTURAS DE DATOS Y ALGORITMOS**

**PRUEBA 1**

**26 DE ABRIL 2019**

**PARTE 1: REVISIÓN DE CONCEPTOS (80 PUNTOS)**

1. Para el siguiente arreglo y el algoritmo de búsqueda lineal:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 5 | 67 | 89 | 3 | 12 | 65 | 32 | 26 | 7 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |

1. ¿Cuántas casillas revisa el algoritmo para decidir que el número 65 sí se encuentra en el arreglo? **(1 pto)**

**7 casillas**

1. ¿Cuántas casillas revisa el algoritmo para decidir que el número 2 no se encuentra en el arreglo? **(1 pto)**

**10 casillas (la totalidad del arreglo)**

1. Para el siguiente arreglo ordenado de manera ascendente:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 5 | 7 | 12 | 13 | 26 | 32 | 65 | 67 | 89 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |

1. ¿Cuántas casillas revisa el algoritmo de búsqueda lineal para decidir que el número 3 sí se encuentra en el arreglo? **(1 pto)**

**1 casilla (el número 3 se encuentra en la primera casilla verificada)**

1. ¿Cuántas casillas revisa el algoritmo de búsqueda binaria para decidir que el número 3 sí se encuentra en el arreglo? **(1 pto)**

**3 casillas (revisa las casillas 4, 1 y 0)**

1. Si la búsqueda binaria es O(logN) y la búsqueda lineal es O(N) ¿cómo se explican los resultados anteriores? **(5 puntos)**

**Las expresiones O() se refieren al comportamiento de los tiempos de ejecución para dos condiciones: una alta cantidad de datos y un comportamiento de peor caso. Bajo esas condiciones, el tiempo de ejecución de la Búsqueda Binaria es menor que el de la Búsqueda Lineal (ya que logN es menor que N). El hecho de que en este particular ejemplo haya sido al revés (la búsqueda binaria tarda más que la búsqueda lineal se debe a que las condiciones de validez de las expresiones O() no se cumplen. En particular, no es el peor caso para ambos algoritmos.**

1. El siguiente es el seudo-código del algoritmo de búsqueda binaria para un arreglo ordenado de manera ascendente (de menor a mayor):

|  |
| --- |
| 1. **function** BB(A,k) 2. inf ← 0 3. sup ← size(A)-1 4. **while** (inf <= sup) **do** 5. mid ← floor( (inf+sup/2)) 6. **if** A[mid]=k **then** 7. **return** mid 8. **else** 9. **if** A[mid]<k **then** 10. inf=mid+1 11. **else** 12. sup=mid-1 13. **end if** 14. **end if** 15. **end while** 16. **return** -1 17. **end function** |

¿Qué líneas debes modificar para que este algoritmo funcione ahora con un arreglo ordenado de manera descendente? Indica los números de línea que debes cambiar y cuál es el nuevo código para esas líneas. **(10 puntos)**

**Hay al menos dos maneras de modificar el código.**

**Manera 1:**

**Se modifica la línea 9, el símbolo < se cambia por el símbolo >**

**Manera 2:**

**Se modifican las líneas 10 y 12.**

**La línea 10 cambia a sup=mid-1 y la línea 12 cambia a inf=mid+1**

**Otra manera :**

**Se modifican las líneas 2,3, 4 10 y 12:**

**Línea 2 cambia a inf <-size(A)-1**

**Línea 3 cambia a sup<-0**

**Línea 4 cambia a while(sup<=inf) do**

**Línea 10 cambia a inf=mid-1**

**Línea 12 cambia a sup=mid+1**

1. ¿Cuál es el resultado (valor de retorno) de buscar el número el número 13 usando el algoritmo de búsqueda binaria con el siguiente arreglo? Explica **(5 puntos)**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 5 | 67 | 89 | 3 | 12 | 65 | 32 | 26 | 7 |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |

**Se define el valor de retorno como la posición en la que se encuentra el número. Si el número no se encuentra en el arreglo, se retorna -1. Dada esa definición, el valor de retorno es -1 (el número no se encuentra en el arreglo). Al ejecutar el algoritmo de búsqueda binaria sobre un arreglo no ordenado, el algoritmo actúa “erráticamente”. En este caso, inicia la verificación en la casilla 4 (que almacena el número 3). Como el número buscado (13) es mayor que 3, la siguiente casilla a verificar es la casilla 7 (que almacena el 32). Como el 13 es menor que el 32, la siguiente casilla a verificar es la casilla (que contiene el número 12). Finalmente, se verifica la casilla 6 y como el número buscado no se encuentra ahí, se “concluye” que el número 13 no se encuentra en el arreglo y se retorna el valor 0**

1. Para las siguientes descripciones, indica:
   1. cuál es el algoritmo de ordenamiento que se describe
   2. si está ordenando los datos de manera ascendente o descendente

Descripción 1: “*Se selecciona el mayor elemento del arreglo A y se intercambia con el elemento en la dirección 0, luego se escoge el segundo mayor elemento de arreglo y se intercambia con el elemento en la dirección 1 y así sucesivamente*” **(5 puntos)**

1. **Selection Sort**
2. **Descendente (ya que cada vez elige el mayor elemento y lo coloca al inicio de la parte no ordenada del arreglo)**

Descripción 2: “*Se cuenta cuántas veces aparece cada elemento en el arreglo original A. El conteo de veces que aparece el número i se almacena en la posición i del arreglo C. Se recorre el arreglo C a partir de la posición N-1 (N: número de elementos del arreglo C). El número (N-1) se copia en el arreglo R (a partir de la posición 0) C[N-1] veces. Luego, el número (N-2) se copia en el arreglo R (a partir de la posición C[N-1]-1) C[N-2] veces y así sucesivamente”* ***(5 puntos)***

1. ***Counting Sort***
2. ***Descendente (ya que el arreglo C se recorre de mayor a menor número)***
3. Para el siguiente algoritmo, que recibe como argumento de entrada un arreglo A de tamaño N:

|  |
| --- |
| **function** F(A, N)  x=0  **for** 1 ≤ i ≤ N-1 **do**  **if** (A[i]<A[x]) **then**  x=i  **return** x  **end function** |

1. Describe cuál es la función que cumple este algoritmo. La descripción del código línea por línea no sirve. Ejemplos de descripciones que sirven: “*este algoritmo ordena los números de un arreglo de menor a mayor*”, “*este algoritmo calcula la sumatoria de los primeros N números naturales*”. **(5 puntos)**

**El algoritmo retorna la posición en la que se encuentra el menor valor del arreglo. Si hay más un valor menor (el menor valor está repetido), retorna la posición del primero encontrado.**

1. Calcula su complejidad computacional (análisis asintótico del tiempo de ejecución) **(5 puntos)**

**Debido a que el algoritmo recorre el arreglo completo siempre, su complejidad es O(N), Theta(N) y Omega(N).**

1. Para el algoritmo descrito por el siguiente seudo-código:

|  |
| --- |
| **1. function** Ordena(A)  **2.** n=A.size()  **3.** cambio=true  **4.** **while** (cambio) **do**  **5.**  cambio=false  **6. for** 0 <= i < n-1 **do**  **7.** **if** list[i] > list[i+1] **then**  **8.** swap(list[i],list[i+1])  **9.** cambio=true  **10.** **end if**  **11.** **end for**  **12.** n=n-1  **13. end while**  **14.**  **return** A  **15. end function** |

1. ¿Cuál es la función del ciclo while? **(5 puntos)**

**Determinar cuándo ya no se ejecuta n cambios en el contenido del arreglo y por lo tanto, se da por finalizado el proceso de ordenamiento.**

1. ¿Cuál es la función del ciclo for? **(5 puntos)**

**Recorrer la parte no ordenada del arreglo e intercambiar los pares de números que no estén ordenados de manera ascendente.**

1. ¿Cuál es la complejidad computacional del algoritmo (O(), Θ() y Ω())? **(6 puntos).** No es necesario que hagas la tabla completa con el conteo de operaciones. Con una explicación breve en castellano basta.

**Las instrucciones del ciclo for interno se ejecutan en el peor de los casos n-1 veces. Las instrucciones del while se ejecutan en el peor de los casos n-1 veces. Por lo tanto, este algoritmo es O(n2).**

**En el mejor de los casos (arreglo ya ordenado), las instrucciones del ciclo for se ejecutan n-1 veces y el ciclo while se ejecuta una sola vez. Por lo tanto, el algoritmo es Omega(n).**

1. Describe un caso en el que no es posible aplicar el algoritmo Counting Sort **(5 puntos)**

**Cuando los números a ordenar son decimales, ya que counting sort usa los índices del arreglo C (que son enteros) para representar la frecuencia de aparición de los números a ordenar.**

1. Para la función hash h(n)=(2\*n+3)%10 con sondeo lineal, dibuja la tabla hash y su contenido luego de haber insertado los siguientes datos (en el orden dado): 20, 13, 5, 0, 3 **(10 puntos)**

**Los valores de los índices en los que se debieran almacenar los números, en principio, son:**

**h(20)=(2\*20+3)%10=43%10=3**

**h(13)=(2\*13+3)%10= 29%10=9**

**h(5)=(2\*5+3)%10=13%10=3**

**h(0)=(2\*0+3)%10=3%10=3**

**h(3)=(2\*3+3)%10=9%10=9**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **3** |  |  | **20** | **5** | **0** |  |  |  | **13** |
| **[0]** | **[1]** | **[2]** | **[3]** | **[4]** | **[5]** | **[6]** | **[7]** | **[8]** | **[9]** |

**El número 20 se inserta en la posición 3.**

**El número 13 en la posición 9.**

**El número 5 se debiera insertar en la posición 3, pero ya está ocupada. Por lo tanto, se busca la siguiente posición vacía que es la posición 4.**

**El número 0 se debiera insertar en la posición 3, pero ya está ocupada. Por lo tanto, se busca la siguiente posición vacía que es la posición 5.**

**El número 3 se debiera insertar en la posición 9, pero ya está ocupada. Por lo tanto, se busca la siguiente posición vacía que es la posición 0.**

1. Para una tabla hash de n buckets y m valores ya almacenados en ella, explica por qué el algoritmo de resolución de colisiones de encadenamiento separado (separate-chaining) es O(m) cuando se considera buscar un dato en el peor escenario posible de colisiones. **(5 puntos)**

**En el peor escenario, los m valores almacenados en la tabla hash se encuentran en la misma cadena. Por lo tanto, cuando se quiere buscar un elemento el peor caso consiste en que el elemento se encuentre al final de la cadena (o que no se encuentre).**