Tarea examen 1

Diseño y análisis de algoritmos

Gibrán Zazueta Cruz

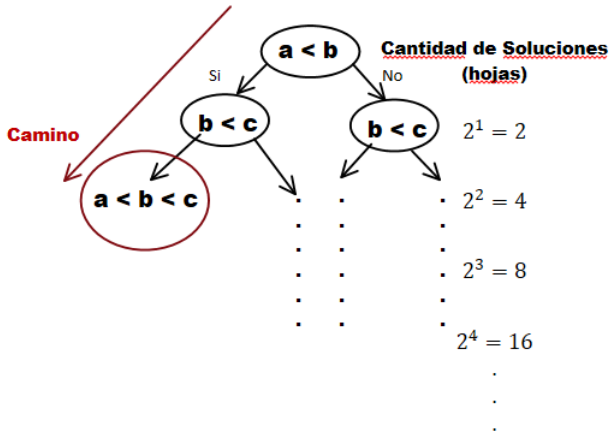
02 de septiembre de 2021

---

1. ¿Cuantas comparaciones son necesarias y suficientes para ordenar cualquier lista de cinco elementos? Justifique su respuesta.

Los árboles de desición son útiles para encontrar cotas inferiores en comparaciones.

Debido a que cada nodo es una comparación, la cantidad de comparaciones necesarias son el camino (cantidad de nodos) desde la hoja (solución) hasta la raiz.



Por otro lado, se tienen tantas hojas como permutaciones posibles, pues el número de permutaciones de n elementos será la cantidad de soluciones posibles. Tomando esto en cuenta, para ordenar cualquier lista de 5 elementos el árbol debe tener, al menos, 5! = 120 hojas. Se calcula la profundidad necesaria (mínima) para tener las 120 hojas

$$2^k \geq 5!$$

$$K \geq \log_2 120$$

$$K \geq 6{,}9$$

$$K \geq 7$$

Para tener 120 hojas se necesitan, al menos, 7 niveles de profundidad. Cada nivel de profundidad equivale a 1 comparación. Por lo tanto **7** comparaciones son necesarias para ordenar cualquier lista de 5 elementos.

Para encontrar el número de comparaciones suficientes se plantea un algoritmo que ordena cualquier lista en la menor cantidad de comparaciones.

El algoritmo propuesto es el siguiente:

Considerando un arreglo A y los elementos a, b, c, d y e

$$A = [a, b, c, d, e]$$

- Se comparan y ordenan los 4 primeros elementos en pareja A[1] con A[2] y A[3] con A[4]
- Se comparan y ordenan los elementos mayores de cada pareja A[2] con A[4]. Se ordenan por pareja

Hasta este punto consideremos por simplicidad que $A[1] < A[2]$, $A[3] < A[4]$ y $A[2] < A[4]$

El arreglo se mantiene:

$$A = [a, b, c, d, e]$$

Sin embargo ya sabemos que $a < b < d$ y que $c < d$

para aprovechar los 3 elementos ya ordenados: a, b y d, se colocan en un arreglo auxiliar.

$$B = [a, b, d]$$

Sobre este arreglo se intenta ordenar e, para garantizar hacerlo en solo 2 comparaciones, se empieza comparando contra el elemento del centro (b).

- Comparar A[5] con B[2]
    1. Si es mayor comparar con B[3] y ordenar
    2. Si es menor comparar con B[1] y ordenar

Digamos que $A[5] > B[2]$ y $A[5] > B[3]$. Entonces ya sabemos que $a < b < d < e$ y aún $c < d$

Para completar la información de c se inserta en el arreglo B, de la misma manera, con solo 2 comparaciones.

- Comparar A[3] con B[2]
    1. Si es mayor comparar con B[3] y ordenar
    2. Si es menor comparar con B[1] y ordenar

Este algoritmo garantiza conocer la posición de cada elemento siempre en la misma cantidad de comparaciones. Como se puede ver en el desglose del algortimo, en el primer paso se hicieron 2 + 1 comparaciones, en el tercero y cuarto dos en cada una, en total suman 7 comparaciones.

Como ya se demostró que se necesitan al menos 7 comparaciones para ordenar la lista, se concluye que 7 comparaciones son necesarias y suficientes para ordenar cualquier lista de 5 elementos.

2. Supongamos que tenemos que ordenar una lista L de n enteros cuyos valores estan entre 1 y m. Pruebe que si m es O(n) entonces los elementos de L pueden ser ordenados en tiempo lineal. ¿Que pasa si m es de O(n^k) con k O(1)? ¿Se puede realizar en tiempo lineal? ¿Por que? ?

Si los valores que deseamos ordenar son enteros y están acotados en un rango, se pueden ordenar en tiempo lineal y sin comparaciones, utilizando Counting Sort.

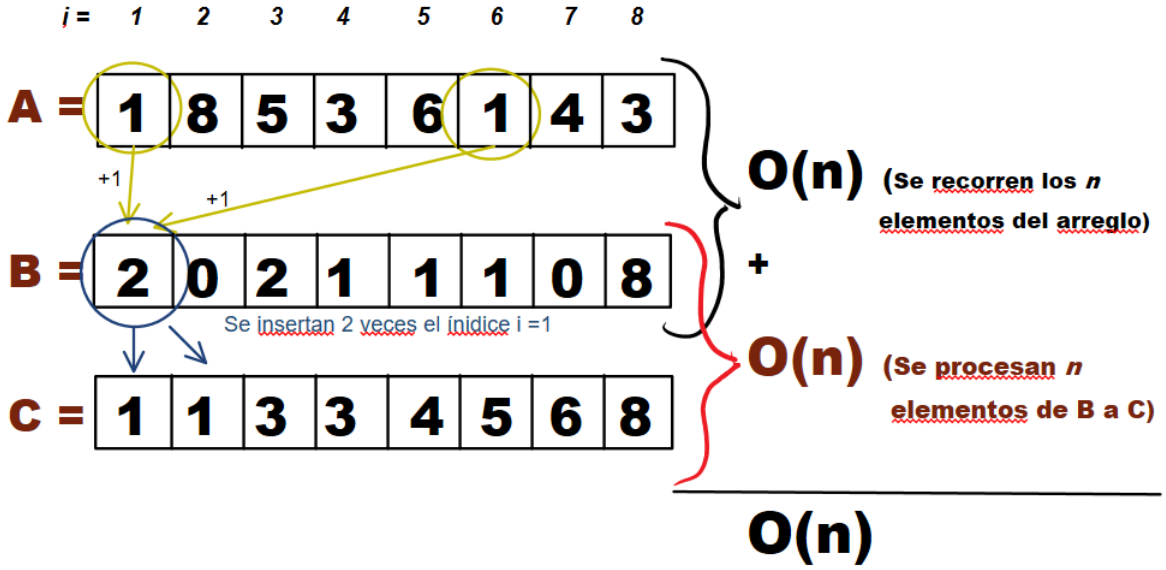
Counting sort asume que cada uno de los n elementos en una lista tiene un valor en un rango de 0 a m, donde m es un entero O(n).

Este algoritmo utiliza tres arreglos: un arreglo A para la entrada, B como un arreglo auxiliar de tamaño 'm' donde se almacenan la cantidad de apariciones de cada elemento de A. Finalmente C, que es la salida ordenada.

Sus pasos son:

- Tomar el primer elemento de A; A[0]
- Usar el elemento A[1] como indice del arreglo auxiliar B[A[1]] y sumar + 1
- Repetir para cada elemento A[i] y B[A[i]]
- Recorrer el arreglo auxiliar B de izquierda a derecha e insertar en C el indice i, la cantidad de veces del valor C[i]

Para visualizarlo se resuelve un ejemplo $n = 8$ y $rango = [1, 8]$



En el ejemplo se visualiza el funcionamiento del algoritmo y se incluye un análisis

de complejidad en cada paso.

Algorithm 1: Counting Sort Pseudocódigo

```
Data: A , m
Result: C = A ordenado
\\Se inicializa el arreglo auxiliar con ceros;
\\Complejidad O(k);
for i := 1 hasta m step 1 do
    B[i]=0;
end
\\Se cuentan las apariciones de cada elemento;
for i := 1 hasta n step 1 do
    B[A[i]]= B[A[i]] + 1;
end
\\Se cuentan las apariciones de cada elemento;
\\La complejidad estará limitada por el valor más grande entre n y k, como k
  es O(n): Complejidad O(n);
for i := 1 hasta k step 1 do
    for j := 1 hasta B[i] step 1 do
        C[j] = i;
    end
end
```

El tiempo total del algoritmo será O(n+k). Se compreba que los elementos pueden ser ordenados en tiempo lineal.

2.1. Si m es O(n^k) con k O(1)

Este problema se puede analizar con el algoritmo de Radix Sort que ordena n números dígito por dígito y en cada ordenación de dígitos utiliza Counting Sort. Sus pasos son:

- Se ordenan los dígitos más a la derecha utilizando counting sort
- Se continua ordenando hacia la izquierda d veces, donde d es la mayor cantidad de dígitos

Algorithm 2: Radix Sort Pseudocódigo

```
Data: A , m
Result: C = A ordenado
\\Ciclo principal de 1 hasta d (mayor cantidad de digitos);
\\Complejidad O(d);
for d_i := 1 hasta d step 1 do
    \\Se inicializa arreglo auxiliar de counting sort, tamaño 10;
    B[10]=0;
    \\Se cuentan las apariciones de cada elemento tomando en cuenta solo el
      dígito d_i ;
    \\Complejidad O(n);
    for i := 0 hasta |A| step 1 do
        B[A[i] en dígito d_i]= B[A[i] en dígito d_i] + 1;
    end
    \\Se cuentan las apariciones de cada elemento;
    for i := 0 hasta 10 step 1 do
        for j := 1 hasta B[i] step 1 do
            C[j] = i;
        end
    end
    \\Actualizar arreglo original A;
    for i := 0 hasta |A| step 1 do
        A[i] = C[i];
    end
end
```

Se repite counting sort tanto como d dígitos, por la tanto la complejidad toal es O(d*n).

Para demostrar porque el algoritmo se mantiene lineal, se muestra la notación desarrollada de n^2

$$n^2 = p(n) + r$$

Donde n^2 es representada en función de n como pareja de 2 valores p y r. Como ejemplo si n=5

$$25 = 5(n) + 0$$

$$18 = 3(n) + 3$$

Se observa que r solo tomará valores de [0, 4] y p de [0, 5], es decir, su rango va de

[0,n-1] y [0, n], respectivamente. Ambos tienen rangos lineales, se ordenan en O(n). Así el algortimo se mantiene O(n).

De igual manera si m es O(n^k), rango=[1,m^k]

$$n^k = \underbrace{p(m^k)}_{[0,m]} + \underbrace{q(m^{k-1})}_{[0,m-1]} + ... + \underbrace{r}_{[0,m-1]} \{$$

Se observa que de cada dígito se obtienen rangos lineales que pueden ser ordenados en O(n), por lo tanto el algoritmo se mantiene lineal cuando m es de O(n^k) con k O(1)

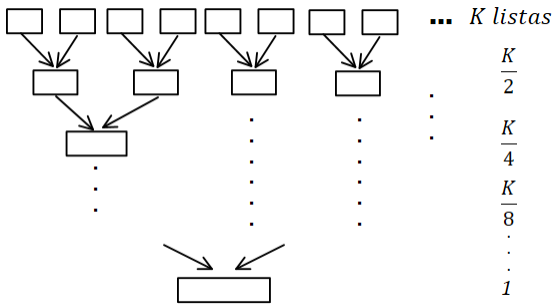
3. Dadas k listas ordenadas con n elementos en total. Diseña un algoritmo de complejidad O(n log k) para obtener la lista ordenada de los n elementos. ¿Que pasa si cada una de las k listas tienen n elementos? Explique.

Para resolver el problema se plantea unir recursivamente las k listas de 2 en 2. Para unir dos arreglos A y B se debe:

- Comparar los primeros elementos de cada arreglo (menores)
- Elegir el menor y avanzar el índice del arreglo de donde se tomó.
- Comparar nuevamente y repetir el proceso
- Cuando un arreglo se queda sin elementos significa que el resto de los elementos del segundo arreglo son mayores, por lo que se acomodan en su mismo orden al final.

Este algoritmo realiza, en el peor caso, tantas comparaciones como la suma de sus elementos. Es decir, en la primera comparación será O(n/k + n/k) y en la última O(n).

Al comparar recursivamente las k listas se observa el siguiente comportamiento



El conjunto de lista K se divide hasta llegar a 1 lista ordenada. Como muestra el comportamiento, debió salir de k entre 2 eleveado a un exponente.

$$\frac{k}{2^a} = 1$$

Donde a es el exponente. Se despeja:

$$log(k) = a$$

Por la tanto la altura del árbol es log(k).

En resumen, por un lado unir dos listas ordenadas cuyos elementos suman n requiere un tiempo O(n),y por otro, este tiempo aumentará de manera logarítmica respecto a la cantidad de listas que se deban unir, es por esto que nuestro algoritmo se comporta con complejidad nlog(k).

A continuación se muestra el algoritmo completo

Algorithm 3: Ordenar k-listas ordenadas

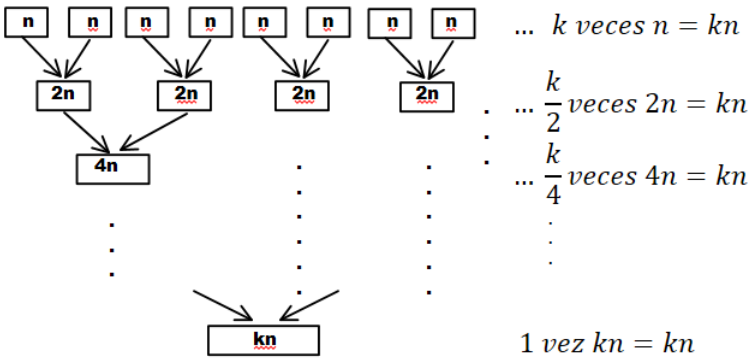
Data: K: arreglo con k-listas ordenadas; n
Result: C = lista ordenada de los n elementos

```
\\Indice del arreglo K;
m ← 0;
\\Mientras el tamaño de la salida sea menor a n;
while |C| < n do
    \\Tomar 2 sublistas de k, para comparar;
    A ← K[m];
    B ← K[m + 1];
    \\Inicializar indices de A, B, C;
    i ← 0, j ← 0, k ← 0 ;
    \\Comparar elementos y colocar en C el menor. Ciclo termina cuando
      acaba alguno de los 2 arreglos;
    while i < |A| and j < |B| do
        if A[i] < B[i] then
            C[k] = A[i] ;
            i=i+1;
        end
        else
            C[k] = B[j] ;
            j=j+1;
        end
        k=k+1;
    end
    \\Acompletar C si quedaron elementos en A o B;
    while i < |A| do
        C[k] = A[i];
        k=k+1;
        i=i+1;
    end
    while j < |B| do
        C[k] = B[j];
        k=k+1;
        i=j+1;
    end
    \\Se reinserta el resultado en la lista de arreglos K para seguir
      comparandolo hasta llegar a |C| = n;
    Insertar lista C en arreglo K;
    \\Aumentar indice de K;
    m=m+2;
end
```

3.1. Cada una de las k listas tiene n elementos

Este caso es muy parecido al anterior. La diferencia es que en lugar de sumar en el primer paso $\frac{n}{k} + \frac{n}{k}$ elementos, ahora se sumarán n+n, mientras que en la última suma la cantidad de elementos en total dependerá del número de k listas.

Se muestra un diagrama para el problema de sumar k listas de n elementos:



Como se puede observar en el diagrama el comportamiento logarítmico se mantiene, sin embargo, ahora se requerirán kn comparaciones en cada paso. Tomando en cuenta este cambio la complejidad de nuestro algoritmo quedará como $knlog(k)$

4. Dado un arreglo A de tamano n que solo contiene 0s, 1s y 2s. Sin usar algoritmos conocidos de ordenamiento con complejidad lineal (por ejemplo Counting sort, radixsort, etc.). Da un algoritmo que en tiempo lineal ordene A

La solución se basa en elegir el número 1 como pivote. Al ser de los 3 números el valor del medio se puede decidir que hacer con los números mayores o menores que él.

La idea del algoritmo es preguntar si el número actual es mayor, menor o igual que 1 si es mayor se extrae y acomoda a la derecha del arreglo, si es menor se extrae y acomoda a la izquierda del arreglo, si es igual se mantiene en su posición.

El algoritmo solo tiene un ciclo que se terminará despues de $n = |A|$, es decir, se recorre el arreglo elemento por elemento solo una vez. Esto nos da una complejidad O(n).

Lo anterior se puede ver en el pseudocódigo.

Algorithm 4: Ordenamiento 0's, 1's, 2's

Data: A: areglo no ordenado

Result: A ordenado

```
\\i es el indice para recorrer el arreglo ;
i ← 0;
\\n registra la cantidad de datos procesados ;
n ← 0;
\\El ciclo termina cuando se hayan procesado la misma cantidad de elmentos
  que el tamaño de A;
\\Complejidad O(n);
while n ≤ |A| do
    \\Si el valor actual es menor al pivote 1 se remueve el elemento de su
     posición original y se inserta la izquierda de A;
    if A[i] < 1 then
        c ← A[i];
        Remover A[i] ;
        Insertar c en a la izquierda de A ;
    else
        if A[i] > 1 then
            \\Si el valor actual es mayor al pivote 1 se remueve el elemento de
             su posición original y se inserta la a la derecha de A;
            c ← A[i];
            Remover A[i] ;
            Insertar c en a la derecha de A ;
        end
        else
            \\Si no es mayor ni menor entonces es igual, se mantiene el elemento
             en su posición y se avanza el ìndice en 1;
            i ← i + 1
        end
    end
    n ← n + 1;
    \\Independientemene del caso n aumenta en 1 ;
end
```

5. Imaginemos que tenemos un barril lleno de sake y n contenedores de madera y n contenedores de vidrio...Nuestro problema es encontrar para cada contenedor de madera su contenedor de vidrio que le cabe la misma cantidad de sake.

5.1. Describe un algoritmo que use $\Theta(n^2)$ comparaciones para resolver el problema.

Para resolver el poblema en $\Theta(n^2)$, una solución consiste en comparar cada n contenedor de madera con cada n contenedor de sake.

Se propone el algoritmo:

Algorithm 5: Algoritmo mayores contenedores $O(n^2)$

```
Data: M, V. Arreglos de contenedores de madera y vidrio
Result: Y Arreglo de parejas de contenedores
\\Ciclo para cada elemento de M;
\\Complejidad O(n);
for i := 0 hasta |M| step 1 do
    \\Se busca en todos los elementos de V la pareja de M actual. El ciclo
      termina cuando se encuentra y llega a un break;
    \\Complejidad O(n);
    for j := 0 hasta |V| step 1 do
        \\Si los elementos son iguales se almacenan los indices de la pareja y se
          hace un break del ciclo for actual ;
        if M[i] == M[j] then
            Guardar indices en Y;
            break;
        end
    end
end
```

Como se puede observar el algortimo tiene dos ciclos, cada uno realizará n comparaciones O(n), por lo que la complejidad del algoritmo es O(n)*O(n) = $O(n^2)$, olo que es lo mismo, se realizaran n^2 comparaciones para resolver el problema.

5.2. Suponga por obvias razones, que solo nos interesa encontrar los contenedores que les cabe más sake. Pruebe que esto puede hacerse con a lo más 2n-2 comparaciones.

El problema se analiza con el algoritmo siguiente

1. Comparar los primeros elementos de cada contenedor
2. Seleccionar el mayor como referencia, si son iguales se toma cualquiera
3. Comparar el máximo de referencia con el siguiente elemento de la lista contraria

4. Si el elemento contrario es menor continuar con el siguiente paso; si es igual tomar registro de su posición y continuar con el siguiente paso; si es mayor tomar registro de su posición y seleccionarlo como el nuevo máximo de referencia.

5. Repetir la comparación del máximo con su lista contraria

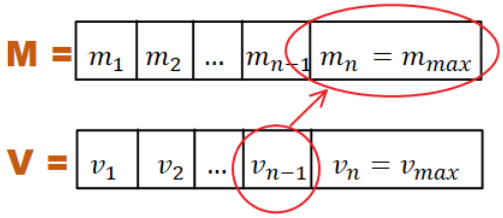
6. El ciclo continua hasta uno de los siguientes casos

   a) Se llega la final de uno de los arreglos y ambos elementos máximos coinciden

   b) Se llega la final de uno de los arreglos y el otro está en su penultimo dato

Para poder considerar todos los elementos es necesario que al menos uno de los arreglos se recorra por completo. Sin embargo, ya que se tenga el elemento máximo es suficiente encontrar a su pareja dentro del arreglo contrario para terminar el algoritmo.

En el peor caso, que es con los elementos de ambos arreglos ordenados de menor a mayor, la condición de terminación será que el arreglo contrario solo le quede un elemento por comparar. El algortimo se detiene porque se sabe que este último elemento será máximo



En rojo se señala ver la última comparación. m_{max} quedará como máximo de referencia y ya no será necesario compararlo otra vez, pues sabemos que el último elemento tiene que ser v_{max} Tomando en cuenta esto podemos deducir que el número de comparaciones máximas necesarias serán n-1 para el arreglo M (pues m_{max} no se compara con el mismo) y n-1 para el arreglo V (pues quedo un último elemento sin comparar). De esta manera, el número de comparaciones máximas del algoritmo es $2n - 2$.

Algorithm 6: Algoritmo mayores contenedores

Data: M, V

Result: max_m, max_v Indices con los valores máximos de m y v

```
1  i ← 0;
2  j ← 0;
3  \\Inicializar índices en 0;
4  max_m ← i;
5  max_v ← j;
6  \\Inicializar máximos en en primer índice;
7  buscar_cont ← 0 \\Variable booleana que dice al algoritmo si debe buscar en el
     arreglo M (0) o V (1);
8  while (i ≤ |M|  or  j ≤ |V|) and (max_m =/ max_v) do
9      \\Ciclo se detiene cuando una de las 2 lista se ha recorrido por completo y
         ya se encontraron los maximos de cada arreglo;
10     if buscar_cont is equal to 0 then
11         \\ Si buscar_cont = 0, buscar en los contenedores de madera;
12         if V[j] is bigger than M[i] then
13             \\ Si buscar_cont = 0, buscar en los contenedores de vidrio;
14             i=i+1;
15         else
16             if V[j] is equal to M[i] then
17                 max_m=i;
18                 i=i+1;
19             end
20             else
21                 max_m=i;
22                 buscar_cont = 1;
23                 j=j+1;
24             end
25         end
26     end
27     if buscar_cont is equal to 1 then
28         if M[i] is bigger than V[j] then
29             j=j+1;
30         else
31             if M[i] is equal to V[j] then
32                 max_j=j;
33                 j=j+1;
34             end
35             else
36                 max_j=j;
37                 buscar_cont = 0;
38                 i=i+1;
39             end
40         end
41     end
42 end
```

5.3. ¿Puedes mejorar el número de comparaciones esperadas del inciso a? Explica.

Se hace un análisis de la cota inferior o complejidad del problema, es decir, la respuesta más rápida en la que se espera que un algoritmo adecuado de una solución.

Para encontrar esta cota se propone la técnica del 'Argumento del adversario'.

Mientras el algoritmo busca realizar la menor cantidad de operaciones, el adversario busca una estrategia para maximimar el número de rondas necesarias para que el problema encuentre solución.

Para este problema cada ronda corresponderá a una comparación, por lo que nos ayudará a saber conocer cuántas comparaciones puede obligar el adversario.

Se propone que el adversario tome una lista $L = \{v_1, v_1, ..v_n\}$ que contenga todos los contenedores de vidrio. El objetivo del algoritmo (jugador) será tomar elementos de la lista de contenedores de Madera y encontrar su pareja.

Consideremos que se toma el contenedor m_1 y se hace la comparación. El adversario deberá responder que no hay igualdad, sin embargo, como debe ser consistente con sus respuestas, eliminará un elemento del arreglo L. Este elemento representa que ya existe una 'no-pareja' de m_1.

Nombramos a L_0 como la lista incial y L_1 la lista después de la comparación

$$L_0 = n$$

$$L_1 = n - 1$$

Esto continuará para las siguientes comparaciones hasta llegar a $L = 1$ elemento. Se puede deducir que se necesitarán al menos n-1 comparaciones para encontrar a la pareja de m_1.

Resimiendo, para cada 0(n) contenedores de madera el adversario siempre dará respuestas que obligen a comparar con una cantidad lineal O(n) de contenedores de vidrio.

Se confirma entonces que la complejidad del problema es O(n^2) y no se podrá dar un algoritmo que mejore el número de comparaciones.

6. Sea A[1 . . . n] un arreglo de n números distintos. Si $i < j$ y $A[i] > A[j]$ entonces la pareja (i, j) es una inversión de A.

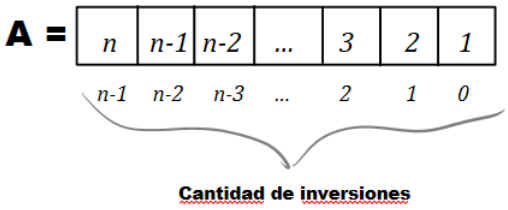
6.1. Lista las 5 inversiones del arreglo [2, 3, 8, 6, 1].

Las inversiones son:

- 2, 1
- 3,1
- 8,1
- 6,1
- 8,6

6.2. ¿Qué arreglo con elementos del conjunto 1, 2, . . . n tiene más inversiones? ¿Cuántos tiene?

El arreglo con mayor número de inversiones será el ordenado de mayor a menor desde n hasta 1. Asi se tendrá un arreglo de la forma:



Donde el elemento n tendrá n-1 inversiones, y el último cero. La suma de todas las inversiones queda como:

$$n-1+n-2+n-3+...+2+1+0$$

que tiene la forma de una progresión aritmética. Se calcula la suma como:

$$\frac{n(a_i+a_n)}{2}$$

donde $a_i = 0 y a_n = n-1$

$$\frac{n(0+n-1)}{2} = \frac{n(n-1)}{2}$$

Por lo tanto la cantidad máxima de inversiones será $\frac{n(n-1)}{2}$

6.3. Diseña un algoritmo que determine el número de inversiones en cualquier permutación de n elementos en $\Theta(nlogn)$ tiempo

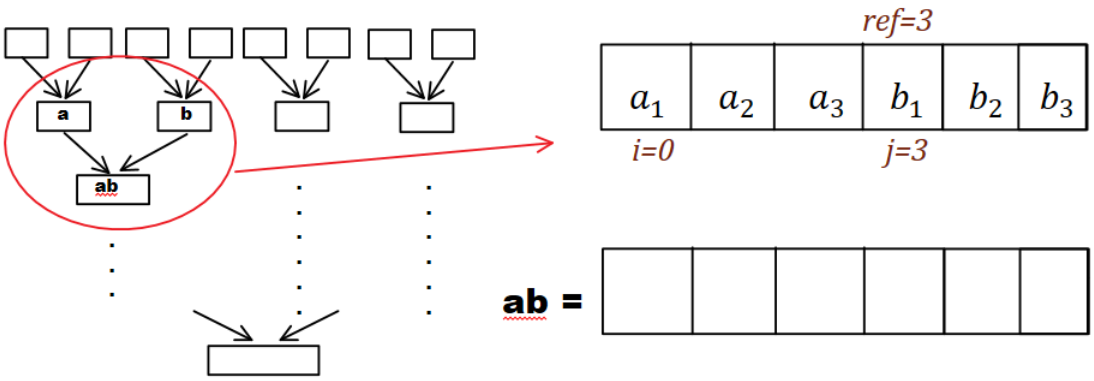
Para el diseño de este algoritmo se usará como base el de Mergesort. Este algoritmo ya es de complejidad O(nlogn) y es posible aprovechar las comparaciones de la etapa de unión para contar inversiones.

Para empezar se divide el arreglo hasta llegar a un elemento, igual que en mergesort.

1. Se marcan los indices de los extremos del arreglo de entrada. $izq = 0, der = n-1$
2. Se declara un indice para el punto medio se le asigna: $\frac{izq-der}{2}$
3. Se llama recursivamente al paso 1 con las dos sublistas (izq, mid) y (mid+1, der)
4. Continua la recursividad hasta que queden sublistas de 1 elemento $der \leq izq$.

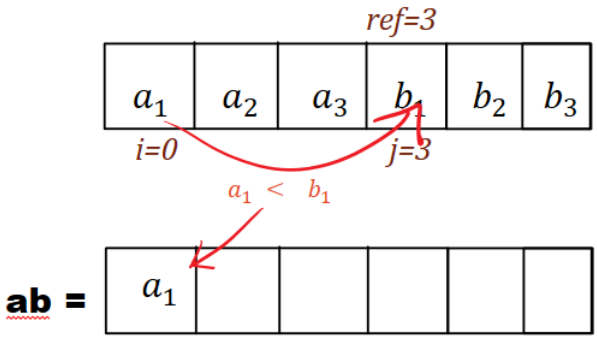
El proceso de unión y cuenta de inversiones se realiza de la siguiente manera:

Se recibe el arreglo con dos sublistas a unir y se definen 3 indices, uno al inicio del arreglo de la sublista izquierda, uno al inicio del de la derecha y otro en esta misma posición que servirá de referencia para contar inversiones. Se visualiza en el diagrama.



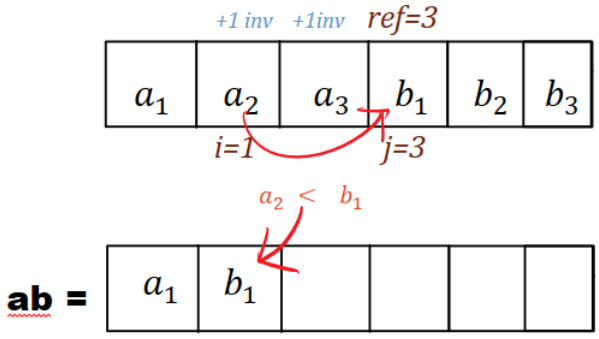
Donde a son los elementos de la izquierda, b los de la derecha. Los indices i, j y ref se pueden tomar de los indices izq y mid que utiliza la separación.

Siguiendo con el proceso de unión: se compara a_1 con b_1, si a_1 es menor se guarda en el nuevo arreglo ab.



Se avanza el indice i y se compara con b_1, si es menor significa que hay al menos una inversión. Debido a que ambas lista están ordenadas, se deduce que los elementos restantes en 'a' también son una inversión. En este caso tanto a_2 como a_3 causarán inversión.

Para calcularlo dentro del arreglo se aprovecha el valor de referencia con la resta $ref - i$.



$$ref - i = 3 - 1 = 2 \textit{ inversiones}$$

Se sigue el mismo procedimiento hasta completar ab y en el proceso contar todas las inversiones.

A continuación se describe el pseudocódigo para la unión y cuenta de inversiones.

Algorithm 7: Contar inversiones con mergesort

Data: a (arreglo de sublistas); izq; der; mid (indices)
Result: ab (arreglo con sublistas ordenadas); $cuentaInv$ (total de inversiones entre sublistas)

```
\\Iniciar indices;
i ← izq;
j ← mid + 1;
ref ← mid + 1;
k ← izq;
\\Ciclo termina cuando se completa una sublista;
while i < mid and j < |B| do
    \\Comparar elementos de cada sublista;
    if a[i] ≤ a[j] then
        \\En este caso no hay inversion;
        ab[k] = a[i] ;
        i=i+1;
    end
    else
        \\En este caso si hay inversion. Se utiliza la diferencia de indices ref-i.;
        inversiones = inversiones + (ref - i);
        ab[k] = a[j] ;
        j=j+1;
    end
    k=k+1;
end
\\Acompletar ab si quedaron elementos en a;
while i ≤ mid do
    ab[k] = a[i];
    k=k+1;
    i=i+1;
end
while j ≤ der do
    ab[k] = a[j];
    k=k+1;
    j=j+1;
end
\\Se copia el nuevo arreglo ordenado ab al arreglo original a;
for i=izq hasta i=der;step 1 do
    a[i] = ab[i]
end
```

Practicamente la operación de la linea 16 es la única diferencia con el algoritmo de ordenamiento Mergesort. Como esta operación teinee complejidad O(1) no se aumenta la complejidad completa origial del algortimo. Es decir se tiene una solución de $O(nlogn)$.

7. Describa un algoritmo que dado un conjunto R de n enteros entre 1 y k pre-procese los datos de entrada y responda en tiempo constante cualquier pregunta del tipo: ¿Cuántos elementos de R caen en el rango [a, b]? El algoritmo tiene que pre-procesar R en tiempo O(n + k).

Algoritmo:

1. Emulando al algoritmo de Counting sort. Crear un arreglo auxiliar de tamaño k donde se almacenen la cantidad de elementos de cada número.

2. Obtener la suma acumulada de este arreglo, a cada elemento (i) le sumas el de su izquierda (i-1) de manera recursiva

3. Para obtener el resultlado se resta del arreglo de sumas el elemento del índice b menos el elemento del índice a-1 (si a=1 se resta 0)

El algoritmo cuanta con dos ciclos, el primero es O(n) y el segundo O(k) por lo que su complejidad es O(n+k) y la de la repsuesta es lineal.

Algorithm 8: Algoritmo Pre-procesamiento $O(n^2)$

Data: R, k, a, b
Result: *elementos_en_rango*
1 *Ciclo para contar la cantidad de elementos de cada número en R*;
2 *Complejidad O(n)*;
3 **for** $i := 1$ *hasta* $|R|$ *step* 1 **do**
4 | aux[R[i]] se incrementa en 1;
5 **end**
6 *Ciclo para hacer la suma acumulada de aux*;
7 *Complejidad O(k)*;
8 **for** $i := 2$ *hasta k step* 1 **do**
9 | aux[i] = aux[i] + aux[i-1];
10 **end**
11 *Cálculo de elementos en R que caen en el rango. si a=1:elementos = R[b], si no: elementos = R[b] − R[a − 1], ;*
12 *Complejidad O(1)*;
13 **if** *a==1* **then**
14 | elementos_en_rango = aux[b];
15 **end**
16 **else**
17 | elementos_en_rango = aux[b] - aux[a-1];
18 **end**

8. Se dice que un arreglo A es k-único si no contiene una pareja de elementos duplicados dentro de k posiciones entre sí, es decir, no existen i y j tal que A[i] = A[j] y $|j-i| \leq$ k. Diseña un algoritmo de tiempo O(n log k) para probar si A es k-único

Para resolver el problema se plantéa utilizar un árbol de búsqueda AVL. Se busca aprovechar las propiedades de estos árboles para buscar los elementos repetidos en tiempo log(n).

Un árbol AVL es un árbol binario de búsqueda autobalanceable. La característica de los árboles de búsqueda es que para un nodo dado todos sus subárboles izquierdos tendrán valores menores que este y todos los subárboles derechos tendrán valores mayores.

Para mantener esta característica es necesario definir un algoritmo para las operaciones de inserción y eliminación. A continuación se menciona de forma general como se realizarán estas operaciones en el árbol de búsqueda.

Eliminación:

1. Si el elemento es hoja se elimina
2. Si el elemento tiene 1 solo descendiente se sustituye por el descendiente
3. Si tiene 2 descendientes, se sustituye por el nodo más a la derecha de su subárbol izquierdo o por el nodo más a la izquierda de su subárbol derecho.

Inserción:

1. Comparar elemento a insertar con la raiz del arbol.
2. Si es mayor se vuelve a aplicar el algoritmo con el nodo de la derecha, si es menor con el de la izquierda.
3. Repetir recursivamente hasta que suceda alguna de las siguientes condiciones:
   a) Alguno de los subárboles no existe (es vacio). El elemento se insertá en esta posición
   b) El elemento a insertar está en el nodo analizado, es decir, está duplicado en el árbol

La segunda condición para detener la recursión es de gran relevancia para nuestro algoritmo ya que indica que la existencia de elementos duplicados.

Si construimos un árbol de búsqueda con k elementos contiguos de A, podremos verificar en la inserción de cada elemento si este está duplicado para este sub-arreglo seleccionado.

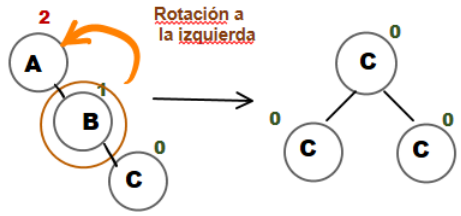
Otra propiedad importante que debe tener el árbol es ser autobalanciable. Esto se refiere a que las alturas de los subárboles izquierdo y derecho de cada nodo tengan una diferencia máxima de 1. Mantener el árbol balanceado garantizará que sus operaciones de inserción y eliminación se mantengan O(n logn). Para ello, durante estas operaciones se llevará registro del *Factor de Equilibrio (FE)* de cada nodo.

Factor de equilibrio = altura subarbol derecho - altura subarbol izquierdo

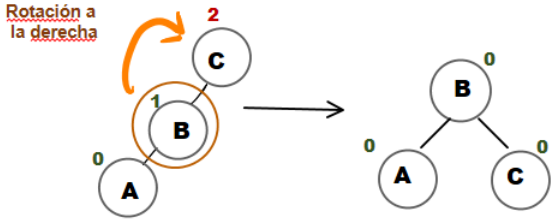
Cuando el FE del nodo esta entre [-1,1] significa que el nodo está equilibrado, cuando se salga de este rango será necesario llamar a un algoritmo de reequilibrio

El reequilibrado del árbol se produce a partir de *Rotaciones* que reacomodan los nodos moviendolos sólo verticalmente. Las rotaciones son del tipo:

Rotación a la derecha:



Rotación a la izquierda:



Habiendo descrito de manera general el funcionamiento del árbol AVL se procede a su implementación en nuestro problema.

Se muestra el pseudocódigo con comentarios para el algoritmo que recorre A y el de inserción en árbol AVL.

Como el árbol se mantiene de tamaño k, la complejidad de sus operaciones será (log(k)). Las operaciones de inserción y eliminación se realizan para n números. La complejidad total es $O(nlogk)$

Algorithm 9: Arreglo k-único

Data: A, K

Result: *duplicado* True o False

Variable booleana que sera True cuando se encuentre un elmento duplicado. Su valor será un retorno de la función insertar;

$duplicado \leftarrow False$;

El primer elemento será la raiz del árbol Inicializar árbol binario con elemento A[0];

; *índice para el siguiente elemento a insertar al árbol en cada iteración. Empieza en 2*;

$i \leftarrow 2$;

Índice para el siguiente elemento a eliminar del árbol en cada iteración. Empieza en 0;

$j \leftarrow 0$;

Variable para contar cuantos elementos hay en total en el árbol y evitar que pase de k. Empieza con 1 (la raiz);

$total\_arbol \leftarrow 1$;

Ciclo para comparar cada n elemento con los que están k posiciones a su alrededor;

Complejidad O(n);

while $i \leq n$ **do**

  Si hay mas de k elementos en el árbol eliminar un elemento;

  if $total\_arbol > k$ **then**

    Remover A[j] del árbol AVL;

    $j \leftarrow j + 1$;

    $total\_arbol \leftarrow total\_arbol - 1$;

  end

  Se inserta un nuevo elemento en el árbol;

  Si se encontró un elemento duplicado durante la inserción se detiene el algoritmo y la respuesta es True;

  Insertar A[i] en árbol AVL; Función retorna 'duplicado';

  if *duplicado* **then**

    break;

  end

  Incrementar indice y total de elementos;

  $i \leftarrow i + 1$ $total\_arbol \leftarrow total\_arbol + 1$;

end

Algorithm 10: Insertar en árbol AVL

Data: *a*: elemento del arreglo, *nodo*: nodo del árbol al que se intentará realizar la inserción
Result: *duplicado* = True or False

```
\\Si 'a' es menor al valor actual del nodo se busca a su izquierda;
if a < nodo then
    if la izquierda del nodo está vacia (null) then
        Colocar 'a' en esta posición;
        if FE es mayor a 1 o menor a -1 then
            Llamar función de reequilibrio
        end
        \\Valor de retorno de la función insertar es False. El elelmento
          insertado no está duplicado;
        duplicado = False ;
    end
    else
        \\LLamada recursiva a función insertar;
        Insertar a a la izquierda del nodo actual;
    end
else
    \\Si 'a' es mayor al valor actual del nodo se busca a su derecha;
    if a > nodo then
        if la derecha del nodo está vacia (null) then
            Colocar 'a' en esta posición;
            if FE es mayor a 1 o menor a -1 then
                Llamar función de reequilibrio
            end
            \\El elemento no está duplicado;
            duplicado = False;
        end
        else
            \\LLamada recursiva a función insertar;
            Insertar a a la derecha del nodo actual;
        end
    end
end
else
    \\Si los casos anteriores son falsos a es igual al valor del nodo. Concluimos
      que se encontró un duplicado. No se inserta el elemento y se regresa un
      valor True;
    duplicado = True;
end
```