



Algoritmos de Búsqueda y Ordenación.

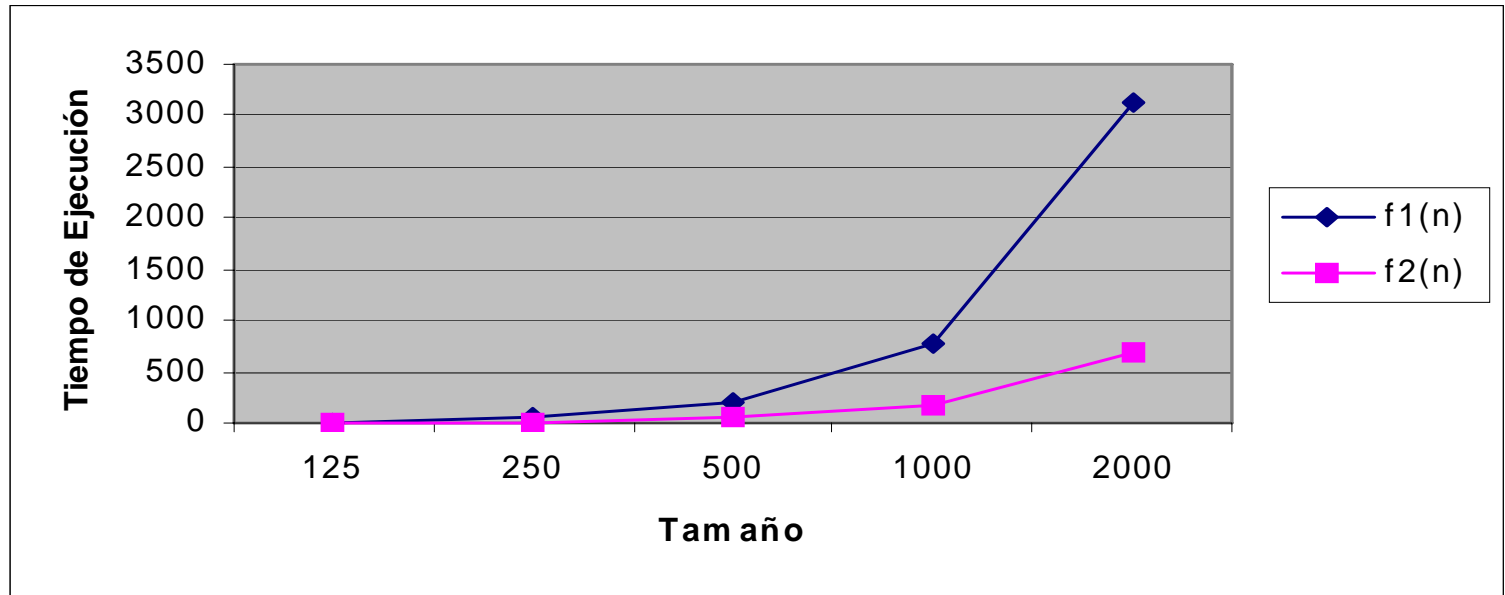
Rosalía Laza Fidalgo.
Departamento de Informática.
Universidad de Vigo

Complejidad

- ¿ Cómo podemos medir y comparar algoritmos, si estos se ejecutan a distintas velocidades y requieren diferentes cantidades de espacio según en qué ordenador se ejecuten, y con qué lenguaje de programación (o compilador) hayan sido implementados? Con la simple recogida de los tiempos de ejecución de los algoritmos indudablemente **no**.

Tamaño del array n	Ordenador A (ms)	Ordenador B (ms)
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	172.9
2000	3114.9	690.5

Complejidad



$$f_1(n) = 0,0007772n^2 + 0,00305n + 0,001$$

$$f_2(n) = 0,0001724n^2 + 0,00400n + 0,100$$

- Independientemente de la máquina, el lenguaje de programación o el compilador que se haya utilizado para la ejecución del algoritmo A , el tiempo de ejecución que consume dicho algoritmo SIEMPRE dará lugar a una *función cuadrática* del tamaño del array que trate. Es decir, la *forma de la curva* será la misma.

Notación O.

- Los tiempos de ejecución para diferentes algoritmos dan lugar a diferentes *clases de complejidad*. Cada clase de complejidad se caracteriza por una familia de curvas distinta. Todas las curvas de una clase de complejidad concreta tienen la misma *forma básica*, pues solo se diferencian entre ellas por el valor de sus constantes.
- La notación O (también llamada *O mayúscula*), se utiliza para comparar la eficiencia de los algoritmos.
- La notación O se centra en el **término dominante** an^2 (es decir, en aquel que más crece cuando n aumenta; tal y como se observa en la Tabla2), e ignora el resto de términos. Por lo tanto, desprecia también la constante de proporcionalidad a . Así:
 - $O(an^2 + bn + c) = O(an^2) = O(n^2)$

$f(n)=an^2+bn+c$ donde $a = 0,0001724$, $b = 0,00400$, $c = 0,1$			
n	f(n)	an^2	Término n^2 como % del total
125	2.8	2.7	94.7
250	11.0	10.8	98.2
500	43.4	43.1	99.3
1000	172.9	172.4	99.7
2000	690.5	689.6	99.9

Propiedades Notación O

- $O(f(x)) + k = O(f(x))$
- $O(k f(x)) = O(f(x))$
- $O(f(x)) * O(g(x)) = O(f(x) * g(x))$
- $O(f(x)) + O(g(x)) = \max(O(f(x)), O(g(x)))$

Complejidad de Sentencias en Java

- **REGLA 1 - CICLOS FOR:**

*El tiempo de ejecución de un ciclo **for** es, como máximo, el tiempo de ejecución de las instrucciones que están en el interior del ciclo (incluyendo las condiciones) por el número de iteraciones. Se supone que las sentencias simples tienen un tiempo de ejecución constante.*

- **REGLA 2 - CICLOS FOR ANIDADOS:**

*Analizarlos de adentro hacia afuera. El tiempo de ejecución total de una proposición dentro del grupo de ciclos **for anidados** es el tiempo de ejecución de la proposición multiplicado por el producto de los tamaños de todos los ciclos **for**.*

- **REGLA 3 - PROPOSICIONES CONSECUTIVAS:**

Simplemente se suman

- **REGLA 4 - CONDICION IF THEN /ELSE:**

if (expresión)

instrucción1

else

instrucción2

su tiempo de ejecución nunca es más grande que el tiempo de ejecución de la expresión más el mayor de los tiempos de ejecución de la instrucción1 y la instrucción2.

Complejidad de un Algoritmo

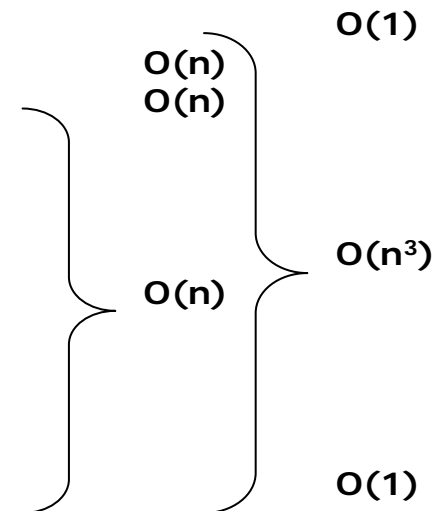
- A continuación se muestra un ejemplo de cálculo de subsecuencia de suma máxima resuelto de tres formas diferentes y cómo influye su resolución en la complejidad del algoritmo.
- *Dada la secuencia de enteros (posiblemente negativos) A_1, A_2, \dots, A_n , encontrar (e identificar la subsecuencia correspondiente) el valor máximo de $\sum_{k=i}^j A_k$. Cuando todos los enteros son negativos entenderemos que la subsecuencia de suma máxima es la vacía, siendo su suma cero.*

Por ejemplo para la entrada $\{-2, \mathbf{11}, \mathbf{-4}, \mathbf{13}, -5, 2\}$, la respuesta es 20.
Y para $\{-2, \mathbf{11}, \mathbf{1}\}$, la respuesta es 12.

Se marca en negrita la subsecuencia de suma máxima

Subsecuencia de suma máxima I

```
public static int subsecuenciaSumaMaxima (int [] a)
{
    int sumaMax = 0;
    for (int i = 0; i < a.length; i++)
        for (int j = i; j < a.length; j++)
            {
                int sumaActual = 0;
                for (int k = i; k <= j; k++)
                    sumaActual += a [k];
                if (sumaActual > sumaMax)
                    sumaMax = sumaActual;
            }
    return sumaMax;
}
```



Complejidad: $O(1) + O(n^3) + O(1) = \underline{O(n^3)}$

Subsecuencia de suma máxima II

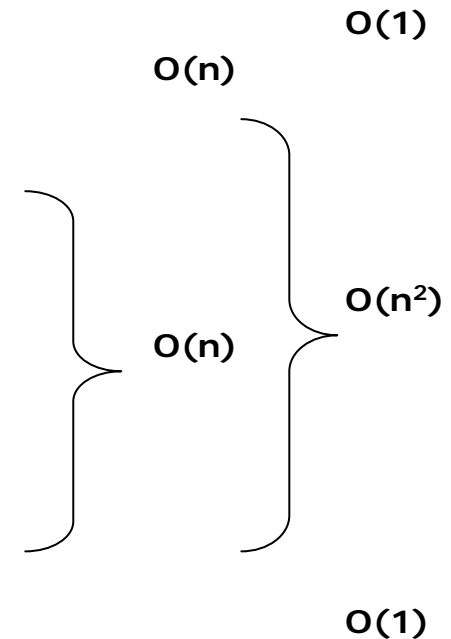
- En el algoritmo anterior el bucle más interno es exageradamente costoso. Una vez que hemos calculado la suma de la subsecuencia que se extiende desde i hasta $j-1$, calcular la suma que va desde i hasta j no debería llevar mucho tiempo, es simplemente una suma más, sin embargo el algoritmo cúbico no utiliza esta información.

Subsecuencia de suma máxima II

```
public static int subsecuenciaSumaMaxima (int [] a)
{
    int sumaMax = 0;
    for (int i = 0; i < a.length; i++)
    {
        int sumaActual = 0;
        for (int j = i; j < a.length; j++)
        {
            sumaActual += a [j];
            if (sumaActual > sumaMax)

                sumaMax = sumaActual;
        }
    }
    return sumaMax;
}
```

Complejidad: $O(1) + O(n^2) + O(1) = \underline{O(n^2)}$





Subsecuencia de suma máxima III

- Cuando se detecta una subsecuencia negativa, no sólo podemos salir del bucle interno del algoritmo anterior, sino que se puede avanzar i hasta $j+1$.

Subsecuencia de suma máxima III

```
public static int subsecuenciaSumaMaxima (int [] a)
{
    int sumaMax = 0;
    int sumaActual = 0;
    for (j = 0; j < a.length; j++)
    {
        sumaActual += a [j];
        if (sumaActual > sumaMax)

            sumaMax = sumaActual;

        else if (sumaActual < 0)
        {
            sumaActual = 0;
        }
    }

    return sumaMax;
}
```

$O(n)$

$O(1)$

$O(1)$

$O(1)$


$O(n)$

$O(1)$

Complejidad: $O(1) + O(1) + O(n) + O(1) = \underline{\underline{O(n)}}$

Notación O (de menor a mayor)

FUNCIÓN	NOTACIÓN O
Constante	$O(1)$
Logarítmica	$O(\log n)$
Lineal	$O(n)$
Logarítmica	$O(n \log n)$
Cuadrática	$O(n^2)$
Cúbica	$O(n^3)$
Exponencial	$O(2^n)$
Exponencial	$O(10^n)$



Complejidad logarítmica

- En la complejidad logarítmica no se tiene en cuenta la base, porque un cambio entre bases sólo implica multiplicar por una constante.
- $\text{Log}_{10}n = \log_2n / \log_210$
- $1/\log_210 = 0.301$
- $\text{Log}_{10}n = 0.301 * \log_2n$

Complejidad Logarítmica

- Es adecuado implementar algoritmos con una complejidad logarítmica, porque los logaritmos crecen muy lentamente aunque el número de datos aumente considerablemente. Se define formalmente un logaritmo para cualquier $b, n > 0$, como:
 - $\text{Log}_b n = k$ si $b^k = n$
- Los logaritmos crecen lentamente. Ej. Un logaritmo de un millón es aproximadamente 20, y el algoritmo de un billón es aproximadamente 40.


$$\text{Log}_b n = k \text{ si } b^k = n$$

Ejemplos de complejidad logarítmica

○ BITS EN UN NÚMERO BINARIO.

¿Cuántos bits son necesarios para representar N enteros consecutivos? Sabiendo que, un entero short de 16 bits representa 65.536 enteros (-32.768 a 32.767) $= 2^{16}$.

Se puede afirmar que con K bits se representan aproximadamente 2^K enteros.

$2^K = N$, por lo que se puede concluir que $K = \log_2 N$, por lo tanto el número de bits mínimo es $\log N$.

Para representar N enteros consecutivos son necesarios $\log N$ bits.

$$\text{Log}_b n = k \text{ si } b^k = n$$

Ejemplos de complejidad logarítmica

DUPLICACIONES REPETIDAS.

Empezando con $X=1$, ¿cuántas veces debe ser X duplicado antes de que sea mayor que N ?

Empezando en 1 se puede duplicar un valor repetidamente solamente *un número de veces logarítmico* antes de alcanzar N .

N ° de Veces que se duplica	1	2	3	...	p
Valor de X	2	4	8	...	N
Valor de X	2^1	2^2	2^3	...	2^p

El número de veces que se duplica es p .

$2^p = N$, es decir $p = \log_2 N$.

Por lo tanto se duplica un número logarítmico de veces.

$$\text{Log}_b n = k \text{ si } b^k = n$$

Ejemplos de complejidad logarítmica

DIVISIONES A LA MITAD.

Empezando con $X=N$, si N se divide de forma repetida por la mitad ¿cuántas veces hay que dividir para hacer N menor o igual a 1?

N ° de Veces que se divide	1	2	3	...	p
N° de datos	N/2	N/4	N/8	...	N/N
N° de datos	$N/2^1$	$N/2^2$	$N/2^3$...	$N/2^p$

El número de veces que se divide es p .

$2^p = N$, es decir $p = \log_2 N$.

Por lo tanto se divide N hasta llegar a 1 un número logarítmico de veces.

Consideraciones sobre la eficiencia de un programa.

- Para resolver un problema debemos encontrar un algoritmo que tenga una complejidad lo más pequeña posible. Si dudamos entre dos algoritmos de la misma complejidad podemos calcular la complejidad en media.
- Si un programa va a ser usado con una cantidad de datos pequeña, la notación O no es muy significativa. En este caso, si es posible, se puede acudir al tiempo medio.
- Los accesos a memoria secundaria suelen ser muy lentos. Deben, por tanto, reducirse a los imprescindibles.

Consideraciones sobre la eficiencia de un programa.

```
public boolean Pertenece1 (int [] A, int X)
{
    int i;
    boolean encontrado = false;
    for (i = 0; i < A.length; i++)
    {
        if (A[i] == X)
            encontrado = true;
    }
    return encontrado;
}
```

Tiempo de Ejecución método Pertenece1 $\rightarrow O(n)$

Consideraciones sobre la eficiencia de un programa.

```
public boolean Pertenece2 (int [] A, int X)
{
    int i = 0;

    while (i < A.length - 1 && A[i] != X)
        i++;

    return A[i] == X;
}
```

Tiempo de Ejecución método Pertenece2 $\rightarrow O(n)$

Consideraciones sobre la eficiencia de un programa.

```
public boolean Pertenece1 (int [] A, int X)
{
    int i;
    boolean encontrado = false;
    for (i = 0; i < A.length; i++)
    {
        if (A[i] == X)
            encontrado = true;
    }
    return encontrado;
}
```

```
public boolean Pertenece2 (int [] A, int X)
{
    int i = 0;
    while (i < A.length - 1 && A[i] != X)
        i++;

    return A[i] == X;
}
```

¿Número de iteraciones?

Pertenece1 ----- La media es **n**.

Pertenece2 ----- La media es **n/2**.



Búsqueda Interna

- El problema de la búsqueda es un problema de *recuperación de la información* lo más rápidamente posible y consiste en localizar un elemento en una lista o secuencia de elementos.
- La operación de búsqueda puede llevarse a cabo sobre elementos ordenados o sobre elementos desordenados.
- **Búsqueda interna** si todos los elementos se encuentran en memoria principal (por ejemplo, almacenados en arrays, vectores o listas enlazadas)
- **Búsqueda externa** si los elementos se encuentran en memoria secundaria.

Búsqueda Interna

- Definición en Java de un array.

```
//declaración de un array de tipo_elemento  
tipo_elemento [] array;
```

```
//reserva de espacio para almacenar los datos en el array  
array = new tipo_elemento[n];
```

- Cada algoritmo de búsqueda trata de localizar en un array un elemento **X**. Una vez finalizada la búsqueda puede suceder:
 - que la búsqueda haya tenido éxito, habiendo localizado la posición donde estaba almacenado el elemento X, o
 - que la búsqueda no haya tenido éxito, concluyéndose que no existía ningún elemento X.

Búsqueda Lineal

- Es la forma más simple de buscar un elemento y consiste en examinar secuencialmente uno a uno hasta encontrar el buscado o haber revisado todos los elementos sin éxito.

```
public static int busquedaLineal(int X, int [] A)
{
    int i = 0;
    int n = A.length - 1;

    while (i < n && A[i] != X)
        i++;

    /* Si se encuentra el elemento se devuelve su posición sino se devuelve -1
    (indica que el elemento no está)*/

    if (A[i] == X) return i;
    else return -1;
}
```

Complejidad Búsqueda Lineal

- **MEJOR CASO:** Si tenemos mucha suerte, puede ser que la primera posición examinada contenga el elemento que busquemos, en cuyo caso el algoritmo informará que tuvo éxito después de una sola comparación. Por tanto, su complejidad será **$O(1)$** .
- **PEOR CASO:** Sucede cuando encontramos X en la última posición del array. Como se requieren n ejecuciones del bucle *mientras*, la cantidad de tiempo es proporcional a la longitud del array n , más un cierto tiempo para realizar las condiciones del bucle mientras y para la llamada al método. Por lo tanto, la cantidad de tiempo es de la forma $an + b$ para ciertas constantes a y b . En notación O , $O(an+b) = O(an) = \mathbf{O(n)}$.
- **CASO MEDIO:** Supongamos que cada elemento almacenado tiene la misma probabilidad de ser buscado. La media se puede calcular tomando el tiempo total de encontrar todos los elementos y dividiéndolo por n :
Total = $a(1 + 2 + \dots + n) + bn = a(n(n+1) / 2) + bn$
Media = (Total / n) = $a((n+1) / 2) + b$ que es **$O(n)$** .

Búsqueda Binaria

- Si los elementos sobre los que se realiza la búsqueda están ordenados, entonces podemos utilizar un algoritmo de búsqueda mucho más rápido que el secuencial, la búsqueda binaria.
- Consiste en reducir paulatinamente el ámbito de búsqueda a la mitad de los elementos, basándose en comparar el elemento a buscar con el elemento que se encuentra en la mitad del intervalo y en base a esta comparación:
 - Si el elemento buscado es **menor** que el elemento medio, entonces sabemos que el elemento está en la mitad inferior de la tabla.
 - Si es **mayor** es porque el elemento está en la mitad superior.
 - Si es **igual** se finaliza con éxito la búsqueda ya que se ha encontrado el elemento.

Código Búsqueda Binaria

```
public static int busquedaBinaria(int X, int [] A){
    int inicio = 0;
    int fin = A.length - 1;
    int medio;

    while (inicio <= fin)
    {
        medio = (inicio+fin)/2;

        if (A[medio] < X) inicio = medio+1;

        else if (A[medio] > X) fin = medio - 1;
        else return medio;
    }

    return -1;
}
```

Código recursivo Búsqueda Binaria

```
public static int binariaRec(int [] A, int X, int inicio, int fin)
{
    int medio;

    if (inicio > fin) return -1;
    else
    {
        medio = (inicio+fin)/2;

        if (A[medio] < X) return binariaRec(A,X,medio+1,fin);

        else if (A[medio] > X) return binariaRec(A,X,inicio,medio -1);
        else return medio;
    }
}
```

Complejidad Búsqueda Binaria

- **CASO ÓPTIMO:** La búsqueda binaria requiere sólo una comparación; esto significa que su tiempo de ejecución óptimo no depende de la cantidad de datos: es constante y por tanto proporcional a 1, es decir, **$O(1)$** .
- **PEOR CASO:** En el peor caso sí dependen de N . La búsqueda binaria divide el array, requiriendo sólo un tiempo $O(\log n)$.
El algoritmo de búsqueda binaria progresivamente va disminuyendo el número de elementos sobre el que realizar la búsqueda a la mitad: $n, n/2, n/4, \dots$. Así, tras **$\log_2 n$** divisiones se habrá localizado el elemento o se tendrá la seguridad de que no estaba.

Búsqueda Indexada

- Necesita que los elementos sobre los que se realiza la búsqueda estén ordenados de acuerdo a una determinada clave. El método utiliza un array auxiliar denominado **array índice**. Cada objeto en el array índice consta de un valor y la posición que ocupa dicho valor en el correspondiente array ordenado:

En Java se define como se muestra a continuación:

```
class ElemIndice{
    private int  clave;
    private int posicion;

    public ElemIndice(int c, int p){
        clave=c;
        posicion=p; }

    public int getClave(){
        return clave;
    }

    public int getPosicion(){
        return posicion;
    }
}
```

Búsqueda Indexada

- Mediante cada elemento del array índice se asocian grupos de elementos del array inicial. Los elementos en el índice y en el array deben estar ordenados. El método consta de dos pasos:
 1. Buscar en el array_índice el intervalo correspondiente al elemento buscado.
 2. Restringir la búsqueda a los elementos del intervalo que se localizó previamente.
- La ventaja del método estriba en que la búsqueda se realiza inicialmente en el array de índices, cuantitativamente más pequeña que el array de elementos. Cuando se ha encontrado el intervalo correcto, se hace una segunda búsqueda en una parte reducida del array. Estas dos búsquedas pueden ser secuenciales o binarias y el tiempo de ejecución dependerá del tipo de búsqueda utilizado en cada uno de los arrays

Ejemplo de Búsqueda Indexada

3	4	5	6	7	8	9	10	14	16	19
---	---	---	---	---	---	---	----	----	----	----

- Se decide dividir el array inicial en bloques de tres elementos.
- El array índice estará formado por $n/k + 1$ elementos, por no ser el último bloque de igual tamaño, es decir, $11/3 + 1 = 4$

Clave	→	3	6	9	16
Posición	→	0	3	6	9

Ejemplo de Búsqueda Indexada

3	4	5	6	7	8	9	10	14	16	19
---	---	---	---	---	---	---	----	----	----	----

- Se puede implementar la búsqueda binaria en el array de índices.
- Cuando finaliza la búsqueda, o ha encontrado el elemento buscado o está en la posición que contiene al elemento mayor de los más pequeños al elemento buscado
- Si $X = 10$, en el array de índice indicará la posición 2. Por lo tanto, la búsqueda en el array inicial comenzará en la posición 6 hasta el final del bloque.

Clave	→	3	6	9	16
Posición	→	0	3	6	9

Código para crear el array de índices

```
public ElemIndice [] crearArrayIndice(int k){
    int n_indices;
    int n = A.length;

    if (n%k == 0) n_indices = n/k;
    else n_indices = n/k + 1;

    ElemIndice [] arrayIndice = new ElemIndice[n_indices];

    int cont = 0;
    for (int i = 0; i<n_indices; i++)
    {
        array_indice[i]= new ElemIndice(A[cont],cont);

        //cont apunta al primer elemento de cada bloque del array grande
        cont += k;  }

    return arrayIndice;
}
```

Ejercicio

Implementa los algoritmos de búsqueda indexada usando la búsqueda binaria tanto en el array de índices como en el array inicial.

```
public int Bus_Binaria_array_indice(Elem_indice [] array_indice, int X){
    int inicio = 0;
    int fin = array_indice.length -1;
    int medio;

    while (inicio <= fin)
    {
        medio = (inicio+fin)/2;

        if (array_indice[medio].getClave() < X) inicio = medio+1;
        else if (array_indice[medio].getClave() > X) fin = medio -1;
        else return array_indice[medio].getPosicion();
    }

    //Caso especial el elemento buscado es más pequeño que el primer elemento
    if (fin == -1) return -1;
    else return array_indice[fin].getPosicion();
}
```



//Busqueda Binaria array

```
public int Bus_Binaria_array(int [] A, Elem_indice [] array_indice, int X, int k){  
  
    int i = Bus_Binaria_array_indice(array_indice, X);  
    if (i != -1){  
        int f = i+k-1;  
  
        if (f >= A.length) f=A.length-1;  
  
        while (i <= f)  
        {  
            int medio = (i+f)/2;  
  
            if (A[medio] < X) i = medio+1;  
            else if (A[medio] > X) f = medio -1;  
            else return medio;  
        } // fin while  
    } // fin if  
    return -1;  
}
```

Otra propuesta para crear el array de Índices de la Búsqueda Indexada.

```
/* El array de índices sólo almacena el primer elemento de cada
   bloque, su posición en el array grande se obtiene multiplicando la
   posición del array de índices por K */
public int [] crearArrayIndice(int k){
    int n_indices;
    int n = A.length;

    if (n%k == 0) n_indices = n/k;
    else n_indices = n/k + 1;

    int [] arrayIndice = new int[n_indices];

    int cont = 0;
    for (int i = 0; i<n_indices; i++)
    {
        array_indice[i] = A[cont];
        cont += k;
    }
    return arrayIndice;
}
```

Búsqueda mediante Hashing

- Existe un método que puede aumentar la velocidad de búsqueda donde los datos no necesitan estar ordenados y esencialmente es independiente del número n . Este método se conoce como **transformación de claves** o **hashing**. El hashing consiste en convertir el elemento almacenado (numérico o alfanumérico) en una dirección (índice) dentro del array.
- La idea general de usar la clave para determinar la dirección del registro es una excelente idea, pero se debe modificar de forma que no se desperdicie tanto espacio. Esta modificación se lleva a cabo mediante una función que transforma una clave en un índice de una tabla y que se denomina **función de Randomización o Hash**. Si H es una función hash y X es un elemento a almacenar, entonces $H(X)$ es la función hash del elemento y se corresponde con el índice donde se debe colocar X .

Búsqueda mediante Hashing

- El método anterior tiene una deficiencia: suponer que dos elementos X e Y son tales que $H(X) = H(Y)$. Entonces, cuando un el elemento X entra en la tabla, éste se inserta en la posición dada por su función Hash, **$H(X)$** . Pero cuando al elemento Y le es asignado su posición donde va a ser insertado mediante la función hash, resulta que la posición que se obtiene es la misma que la del elemento X . Esta situación se denomina ***Randomización o Hashing con colisión o choque***.
- Una buena función Hash será aquella que minimice los choques o coincidencias, y que distribuya los elementos uniformemente a través del array. Esta es la razón por la que el tamaño del array debe ser un poco mayor que el número real de elementos a insertar, pues cuanto más grande sea el rango de la función de randomización, es menos probable que dos claves generen el mismo valor de asignación o hash, es decir, que se asigne una misma posición a más de un elemento.

Métodos de Transformación de claves

TRUNCAMIENTO: Ignora parte de la clave y se utiliza la parte restante directamente como índice. Si las claves, por ejemplo, son enteros de 8 dígitos y la tabla de transformación tiene 100 posiciones, entonces el 1º, 2º y 5º dígitos desde la derecha pueden formar la función hash. Por ejemplo, 72588495 se convierte en 895.

El truncamiento es un método muy rápido, pero falla para distribuir las claves de modo uniforme.

MÉTODO DE DIVISIÓN: Se escoge un número m mayor que el número n de elementos a insertar, es decir, el array posee más posiciones que elementos a insertar. La función hash H se define por:

$$H(X) = X \% m \quad \text{o} \quad H(X) = (X \% m) + 1$$

donde $X \% m$ indica el resto de la división de X por m . La segunda fórmula se usa cuando queremos que las direcciones hash vayan de 1 a m en vez de desde 0 hasta $m-1$.

El mejor resultado del método de división se obtiene cuando m es primo (es decir, m no es divisible por ningún entero positivo distinto de 1 y m).

Métodos de Transformación de claves

MÉTODO DEL MEDIO DEL CUADRADO: La clave es multiplicada por sí misma y los dígitos del medio (el número exacto depende del rango del índice) del cuadrado son utilizados como índice.

MÉTODO DE SUPERPOSICIÓN: Consiste en la división de la clave en diferentes partes y su combinación en un modo conveniente (a menudo utilizando suma o multiplicación) para obtener el índice. La clave X se divide en varias partes X_1, X_2, \dots, X_n , donde cada una, con la única posible excepción de la última, tiene el mismo número de dígitos que la dirección especificada. A continuación, se suman todas las partes. En esta operación se desprecian los dígitos más significativos que se obtengan de arrastre o acarreo.

Hay dos formas de conseguir la función hash mediante este método:

1. **superposición por desplazamiento** donde todas las partes se suman entre sí.
2. **superposición por plegado** se hace la inversa a las partes pares, X_2, X_4, \dots , antes de sumarlas, con el fin de afinar más.

Ejemplos de Transformación de claves

Supongamos un array con 100 posiciones para guardar las claves de los empleados. Aplica las funciones hash anteriores para cada uno de los empleados: 3205 7148 2345.

- **Método de la división:** Escojo un n° primo próximo a 100 ($m = 97$). Aplico la función Hash $H(X) = X \% m$.
 - $H(3205) = 4$, $H(7148) = 67$, $H(2345) = 17$.
- **Método del medio del cuadrado:** Se escogen el 4º y el 5º dígitos por la derecha par obtener la dir. Hash
 - K: 3205 7148 2345
 - K^2 : 10272025 51093904 5499025

Ejemplos de Transformación de claves

○ Método de Superposición:

- Por desplazamiento:

- $H(3205) = 32 + 05 = 37$

- $H(7148) = 71 + 48 = 19$

- $H(2345) = 23 + 45 = 68$

- Por plegado:

- $H(3205) = 32 + 50 = 82$

- $H(7148) = 71 + 84 = 55$

- $H(2345) = 23 + 54 = 77$

Soluciones al problema de las colisiones

- Supongamos que queremos añadir un nuevo elemento k , pero la posición de memoria $H(k)$ ya está ocupada. Esta situación se llama **colisión**.
- Para resolver las colisiones se utilizan los siguientes métodos:
 - Rehashing o Reasignación.
 - Encadenamiento o Tablas Hash Abiertas.
 - Zona de Desbordamiento.

Soluciones al problema de las colisiones

- Un factor importante en la elección del procedimiento a utilizar es la relación entre el número n de elementos y el número m de tamaño del array. Conocido como **factor de carga**, $\lambda = n/m$.
- La **eficiencia** de una función hash con un procedimiento de resolución de colisiones se mide por el número medio de *pruebas* (comparaciones entre elementos) necesarias para encontrar la posición de un elemento X dado. La eficiencia depende principalmente del factor de carga λ . En concreto, interesa conocer:
 - $S(\lambda)$ = nº medio de celdas examinadas para una búsqueda CON éxito
 - $U(\lambda)$ = nº medio de celdas examinadas para una búsqueda SIN éxito.

Rehasing o Reasignación

- Se trata de insertar o buscar un elemento cuya posición ya está ocupada, en otra posición disponible en la tabla.
- Esto se hace mediante la **función de reasignación $R(H(X))$** , la cual acepta un índice de la tabla y produce otro.
- Métodos de reasignación:
 - Prueba lineal
 - Prueba cuadrática
 - Doble dirección hash

Reasignación. Prueba Lineal

- Consiste en que una vez detectada la colisión se debe recorrer el array secuencialmente a partir del punto de colisión, buscando al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o bien cuando se encuentra una posición vacía.
- Se trata al array como a una estructura circular: el siguiente elemento después del último es el primero.
- La función de rehashing es, por tanto, de la forma: **$R(H(X)) = (H(X) + 1) \% m$**

Código Prueba Lineal

```
public static int pruebaLineal(int X, int[] A){
    int m = A.length;
    int dirHash = X%m;

    if (A[dirHash] == X) return dirHash;
    else {
        int dirReh = (dirHash + 1)%m;
        while ((A[dirReh] != X) && (A[dirReh] != 0) && (dirReh !=
dirHash))
            dirReh = (dirReh + 1)%m;

        /* Se ha encontrado el elemento buscado*/
        if (A[dirReh] == X) return dirReh;
        else return -1;
    }
}
```

Complejidad Prueba Lineal

- Utilizando este método de resolución de colisiones, el número medio de pruebas para una búsqueda CON éxito es:
$$S(\lambda) = \frac{1}{2} (1 + (1 / (1 - \lambda)))$$
- y el número medio de pruebas para una búsqueda SIN éxito es:
$$U(\lambda) = \frac{1}{2} (1 + (1 / (1 - \lambda)^2))$$

La principal desventaja de este método es que puede haber un fuerte agrupamiento alrededor de ciertas claves, mientras que otras zonas del array permanezcan vacías. Si las concentraciones de claves son muy frecuentes, la búsqueda será principalmente secuencial perdiendo así las ventajas del método hash. Una solución es que la función hash acceda a todas las posiciones de la tabla, pero no de una en una posición sino avanzando varias posiciones.

Rehasing, mejora

- Para evitar que la Prueba Lineal se convierta en una búsqueda Lineal: Se cumple que, para cualquier función $R(H(i)) = (i+c) \% m$, donde m es el número de elementos de la tabla y c es una constante tal que c y m son *primos relativos* (es decir, no tienen factores en común), **genera valores sucesivos que cubren toda la tabla**.
- Sin embargo, si m y c tienen factores en común, el número de posiciones diferentes de la tabla que se obtienen será el cociente de dividir m entre c .
- Basándonos en esta última afirmación, el algoritmo de búsqueda (y el de carga) tienen un problema: utilizando una función de rehashing de este tipo podrían salir sin inserción aun cuando exista alguna posición vacía en la tabla

Ejemplo:

Dada la función de reasignación:

$RH(i) = (i + 200) \% 1000$ Con esta función, cada clave solo puede ser colocada en cinco posiciones posibles: $(1000/200 = 5)$

si $i = 215$, y esta posición está ocupada, entonces se reasigna de la siguiente forma:

$RH(215) = (215 + 200) \% 1000$	=	415
$RH(415) = (415 + 200) \% 1000$	=	615
$RH(615) = (615 + 200) \% 1000$	=	815
$RH(815) = (815 + 200) \% 1000$	=	15
$RH(15) = (15 + 200) \% 1000$	=	215
$RH(215) = (215 + 200) \% 1000$	=	415

.....

Si estas cinco posiciones posibles están ocupadas, saldremos sin inserción aun cuando haya posiciones libres en la tabla.

Rehashing: Agrupamiento o clustering

- A pesar de la utilización de una función de rehashing donde c y m sean primos, con el método de prueba lineal los elementos siguen tendiendo a *agruparse*, o sea, a aparecer unos junto a otros, cuando el factor de carga es mayor del 50%.
- Cuando la tabla está vacía es igualmente probable que cualquier elemento al azar sea colocado en cualquier posición libre dentro de la tabla. Pero una vez que se han tenido algunas entradas y se han presentado varias colisiones en la asignación, esto no es cierto.

Ejemplo de Agrupamiento

Clave	Índice
4618396	396
4957397	397
	398
1286399	399
	400
	401
.....	
0000990	990
0000991	991
1200992	992
0047993	993
	994
9846995	995

- En este caso, es cinco veces más probable que un registro sea insertado en la posición 994 que en la posición 401. Esto se debe a que cualquier registro cuya clave genera la asignación 990, 991, 992, 993 o 994 será colocado en la 994, mientras que cualquier registro cuya clave genera la asignación 401 será colocado en su posición.
- Este fenómeno en el que **dos claves que inicialmente generan una asignación en dos sitios diferentes, luego compiten entre sí en reasignaciones sucesivas**, se denomina **agrupamiento o clustering**.

Agrupamiento o clustering

- **Agrupamiento o clustering:** $H(X) < > H(Y)$ y $RH(X) = RH(Y)$ en reasignaciones sucesivas, se cumple que:
 - cualquier función de reasignación que dependa únicamente del índice dará lugar a agrupamiento.
 - daría agrupamiento la función de reasignación **$Rh(i) = (i + c) \% m$** aunque c y m fuesen primos.



Rehashing. Prueba cuadrática

- El clustering se debe a que asignaciones sucesivas siguen la misma secuencia. Si se consigue que esa secuencia varíe en cada reasignación se evitaría la formación de cluster.
- Las técnicas de **prueba cuadrática** y **doble dirección hash** minimizan el agrupamiento.

Rehashing. Prueba cuadrática.

- Este método es similar al de la prueba lineal. La diferencia consiste en que, en lugar de buscar en las posiciones con direcciones: dir_Hash , $\text{dir_Hash} + 1$, $\text{dir_Hash} + 2$, $\text{dir_Hash} + 3$,
- buscamos linealmente en las posiciones con direcciones: dir_Hash , $\text{dir_Hash} + 1$, $\text{dir_Hash} + 4$, $\text{dir_Hash} + 9$, ..., **$\text{dir_Hash} + i^2$**
- Si el número m de posiciones en la tabla T es un número primo y el factor de carga no excede del 50%, sabemos que siempre insertaremos el nuevo elemento X y que ninguna celda será consultada dos veces durante un acceso.

Código. Prueba Cuadrática.

```
public static int pruebaCuadratica(int X, int[] A){
    int m = A.length;
    int dirHash = X%m;
    if (A[dirHash] == X) return dirHash;
    else {
        int i = 1;
        int cont = 0;
        int dirReh = (dirHash + 1)%m;

        while ((A[dirReh] != X) && (A[dirReh] != 0) && (cont < m*10))
        {
            i++;
            dirReh = (dirHash + (i*i))%m;
            cont++;
        }
        if (A[dirReh] == X) return dirReh;
        else return -1;
    }
}
```

Rehashing. Doble dirección hash.

- Consiste en que una vez detectada la colisión se debe generar otra dirección aplicando una función hash H_2 a la dirección previamente obtenida. Entonces buscamos linealmente en las posiciones que se encuentran a una distancia $H_2(X)$, $2 H_2(X)$, $3 H_2(X)$, ...
- La función hash H_2 que se aplique a las sucesivas direcciones puede ser o no ser la misma que originalmente se aplicó a la clave. No existe una regla que permita decidir cuál será la mejor función a emplear en el cálculo de las sucesivas direcciones. Pero una buena elección sería $H_2(X) = R - (X \% R)$, siendo R un número primo más pequeño que el tamaño del array. Y se obtienen unos resultados mejores cuando el tamaño del array es un número primo.

Código. Doble dirección Hash.

```
public static int dobleDireccionamiento(int X, int R, int[] A){
    int m = A.length;
    int dirHash = X%m;
    if (A[dirHash] == X) return dirHash;
    else {
        int dirHash2 = R - (X%R);
        int i = 1;
        int dirReh = (dirHash + dirHash2)%m;
        while ((A[dirReh] != X) && (A[dirReh] != 0) )
        {
            i++;
            dirReh = (dirHash + i*dirHash2)%m;
        }
        if (A[dirReh] == X) return dirReh;
        else return -1;
    }
}
```

Problemas de Rehasing

- **INSERCIÓN:** Como se asume una tabla de tamaño fijo, si el número de elementos aumenta más allá de ese tamaño es imposible insertarlo sin que sea necesario asignar una tabla más grande y recalcular los valores de asignación de las claves de todos los elementos que ya se encuentran en la tabla utilizando una nueva función de asignación.
- **BORRADO:** Es difícil eliminar un elemento. Por ejemplo, si el elemento $r1$ está en la posición p , para añadir un elemento $r2$ cuya clave $k2$ queda asignada en p , éste debe ser insertado en la primera posición libre de las siguientes: $Rh(p)$, $Rh(Rh(p))$.. Si luego $r1$ es eliminado y la posición p queda vacía, una búsqueda posterior del elemento $r2$ comenzará en la posición $H(k2) = p$. Como esta posición está ahora vacía, el proceso de búsqueda puede erróneamente llevarnos a la conclusión de que el elemento $r2$ no se encuentra en la tabla.
Solución: marcar el elemento eliminado como '*eliminado*' en vez de '*vacío*', y continuar la búsqueda cuando se encuentra una posición como '*eliminada*'.

Tablas Hash Abiertas

- Otro método para resolver las colisiones consiste en mantener una lista encadenada de todos los elementos cuyas claves generan la misma posición. Si la función hash genera valores entre 0 y $(m - 1)$, declaramos un array de nodos de encabezamiento de tamaño m , de manera que $\text{Tabla}[i]$ apunta a la lista con todos los elementos cuyas claves generan posiciones en i .
- ¿INSERCIÓN?: Al buscar un elemento cuya función hash le hace corresponder la posición i , se accede a la cabeza de la lista correspondiente a esa posición, $\text{Tabla}[i]$, y se recorre la lista que dicha posición inicia. Si éste no se encuentra entonces se inserta al final de la lista.
- ¿ELIMINACIÓN?: La eliminación de un nodo de una tabla que ha sido construida mediante randomización y encadenamiento se reduce simplemente a eliminar un nodo de la lista encadenada. Un nodo eliminado no afecta a la eficiencia del algoritmo de búsqueda. El algoritmo continúa como si el nodo nunca se hubiera insertado.

Tablas Hash Abiertas.

- Estas listas pueden reorganizarse para maximizar la eficiencia de búsqueda, utilizando diferentes métodos:
 - **PROBABILIDAD.** Consiste en insertar los elementos dentro de la lista en su punto apropiado. Esto significa que si **pb** es la probabilidad de que un elemento sea el argumento buscado, entonces el elemento debe insertarse entre los elementos $r(i)$ y $r(i+1)$, donde i es tal que $P(i) \geq pb \geq P(i+1)$
 - **MOVIMIENTO AL FRENTE:** Cuando una búsqueda ha tenido éxito, el elemento encontrado es retirado de su posición actual en la lista y colocado en la cabeza de dicha lista.
 - **TRASPOSICIÓN:** Cuando una búsqueda de un elemento ha tenido éxito es intercambiado con el elemento que le precede inmediatamente, de manera que si es accedido muchas veces llegará a ocupar la primera posición de la lista.

Complejidad Tablas Hash Abiertas

- El número medio de pruebas, con encadenamiento, para una búsqueda con éxito y para una búsqueda sin éxito tiene los siguientes valores aproximados:
$$S(\lambda) = 1 + \frac{1}{2} \lambda$$
$$U(\lambda) = e^{-\lambda} + \lambda$$
- ¿*VENTAJAS?*: Es el método más eficiente debido al dinamismo propio de las listas. Cualquiera que sea el número de colisiones registradas en una posición, siempre será posible tratar una más.
- ¿*DESVENTAJAS?*: La desventaja principal es el espacio extra adicional que se requiere para la referencia a otros elementos. Sin embargo, la tabla inicial es generalmente pequeña en esquemas que utilizan encadenamiento comparado con aquellos que utilizan reasignación. Esto se debe a que el encadenamiento es menos catastrófico si la tabla llega a llenarse completamente, pues siempre es posible asignar más nodos y añadirlos a varias listas.



Zona de desbordamiento

- Se trata de mantener una zona reservada para aquellos elementos que llegan a colisionar, de manera que cuando se produzca una colisión el elemento se va a localizar en esta zona de desbordamiento.
- Al realizar la búsqueda y comprobar que la posición está ocupada por otro elemento con el mismo valor de hashing, se seguirá buscando a partir del inicio de la zona de desbordamiento de manera secuencial, hasta encontrar el elemento o llegar al final de dicha zona de desbordamiento.



Algoritmos de Ordenación.

Ordenación

- La ordenación de elementos según un orden ascendente o descendente influye notablemente en la velocidad y simplicidad de los algoritmos que los manipulan posteriormente.
- En general, *un conjunto de elementos se almacenan en forma ordenada con el fin de simplificar la recuperación de información manualmente*, o facilitar el acceso mecanizado a los datos de una manera más eficiente.
- Los métodos de ordenación se suelen dividir en:
 - **ordenamiento interno**, si los elementos que han de ser ordenados están en la Memoria Principal.
 - **ordenamiento externo**, si los elementos que han de ser ordenados están en un dispositivo de almacenamiento auxiliar.

Complejidad algoritmos de ordenación

- La complejidad de cualquier algoritmo estima el **tiempo de ejecución** como una **función del número de elementos a ser ordenados**.
- Cada algoritmo estará compuesto de las siguientes operaciones:
 - COMPARACIONES que prueban si $A_i < A_j$ ó $A_i < B$ (donde B es una variable auxiliar)
 - INTERCAMBIOS: permutar los contenidos de A_i y A_j ó A_i y B
 - ASIGNACIONES de la forma $B \leftarrow A_i$, $A_j \leftarrow B$ ó $A_j \leftarrow A_i$
- Generalmente, la función de complejidad solo computa COMPARACIONES porque el número de las otras operaciones es como mucho constante del número de comparaciones.



Ordenación por inserción

- También conocido como **método de la baraja**.
- Consiste en **tomar elemento a elemento e ir insertando cada elemento en su posición correcta** de manera que se mantiene el orden de los elementos ya ordenados.
- Es el método habitual usado por los jugadores de cartas para ordenar: tomar carta por carta manteniendo la ordenación.

Ordenación por Inserción


- Inicialmente se toma el primer elemento, a continuación se toma el segundo y se inserta en la posición adecuada para que ambos estén ordenados, se toma el tercero y se vuelve a insertar en la posición adecuada para que los tres estén ordenados, y así sucesivamente.
 1. Suponemos el primer elemento ordenado.
 2. Desde el segundo hasta el último elemento, hacer:
 1. suponer ordenados los **(i - 1)** primeros elementos
 2. tomar el elemento **i**
 3. buscar su posición correcta
 4. insertar dicho elemento, obteniendo **i** elementos ordenados

Código Ordenación por Inserción

```
public void OrdInsercion()
{
    for (int i=1; i < A.length; i++) // Supone el primer elemento ordenado
    {
        int elem = A[i]; // Elemento a ordenar
        int j = (i-1); // Posición a comparar

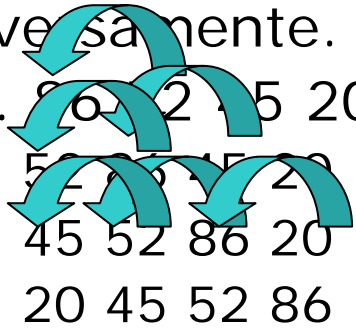
        /*Si el elemento a comparar es mayor que el elemento a ordenar
        entonces desplazo el elemento a comparar una posición a la derecha
        para insertar el elemento a ordenar en la posición correcta*/
        while ((j >= 0) && (elem < A[j]))
            A[j+1] = A[j--]; /*Desplazo el elemento una posición a la
derecha
                                y disminuyo en una unidad la Posición a
comparar*/
        // Se inserta el elemento a ordenar en su posición correcta
        A[j+1] = elem;
    }
}
```

Complejidad Ordenación por Inserción

- **CASO MEJOR:**
Cuando el array está ordenado. Entonces sólo se hace una comparación en cada paso.

- Ej. 15 20 45 60 $n=4$
- En general, para n elementos se hacen $(n-1)$ comparaciones. Por tanto, complejidad $O(n)$.

Pasada	Nº de Intercambios	Nº Comparaciones
1	0	1
2	0	1
3	0	1

Complejidad Ordenación por Inserción

- **CASO PEOR:** Cuando el array está ordenado inversamente.
- Ej.  $n=4$
 - 86 2 45 20
 - 45 52 86 20
 - 20 45 52 86
- En general, para n elementos se realizan $(n-1)$ intercambios y $(n-1)$ comparaciones. Por tanto, $O(n^2)$.

Pasada	Nº de Intercambios	Nº Comparaciones
1	1	1
2	2	2
3	3	3

Complejidad Ordenación por Inserción

- **CASO MEDIO:** Los elementos aparecen de forma aleatoria.
- Se puede calcular como la suma de las comparaciones mínimas y máximas dividida entre dos:
$$((n-1) + n(n-1)/2)/2 = (n^2 + (n-2))/4$$
 , por tanto complejida $O(n^2)$.

Inserción Binaria.

```
public void OrdInsercionBin()
{
    for (int i=1; i < A.length; i++)
    {
        int elem = A[i];
        int bajo = 0;
        int alto = (i-1);

        while (bajo <= alto)//Se busca la posición donde se debe almacenar el
            elemento a ordenar
            {
                int medio = (alto + bajo)/2;
                if (elem < A[medio]) alto = medio -1;
                else bajo = medio + 1;
            }
        for (int j = (i-1); j >= bajo; j--)//Se desplazan todos los elementos
            mayores que el elemento a ordenar una posición a la derecha
            A[j+1] = A[j];
        A[bajo] = elem;
    }
}
```



Complejidad Inserción Binaria

- Con la búsqueda binaria se reduce el número de comparaciones desde $O(n^2)$ hasta un $O(n \log n)$. Sin embargo, el número de sustituciones requiere un tiempo de ejecución de $O(n^2)$. Por lo tanto, el orden de complejidad no cambia, además la ordenación por inserción se usa normalmente sólo cuando n es pequeño, y en este caso la búsqueda lineal es igual de eficiente que la búsqueda binaria.

Ordenación por Selección.

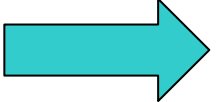
- Este método se basa en que **cada vez que se mueve un elemento, se lleva a su posición correcta**. Se comienza examinando todos los elementos, se localiza el más pequeño y se sitúa en la primera posición. A continuación, se localiza el menor de los restantes y se sitúa en la segunda posición. Se procede de manera similar sucesivamente hasta que quedan dos elementos. Entonces se localiza el menor y se sitúa en la penúltima posición y el último elemento, que será el mayor de todos, ya queda automáticamente colocado en su posición correcta.
- Para **i** desde la primera posición hasta la penúltima
 localizar menor desde **i** hasta el final
 intercambiar ambos elementos

Código Ordenación por Selección

```
public void OrdSeleccion()
{
    //Para todos los elementos desde el primero hasta el penultimo
    for (int i=0; i < (A.length-1); i++)
    {
        int menor = i;
        /*Se localiza el elemento menor desde la posición
        desde la cual se está ordenando hasta el último elemento*/
        for (int j=(i+1); j < A.length; j++)
            if (A[j] < A[menor]) menor = j;

        /*Si el elemento a ordenar es distinto del de la
        posición que se está ordenando se intercambian los elementos*/
        if (menor != i)
            intercambiar(i,menor); //intercambia los elementos
        de esas posiciones
    }
}
```

Complejidad Ordenación por Selección

- El tiempo de ejecución de dicho algoritmo viene determinado por el **número de comparaciones**, las cuales son independientes del orden original de los elementos, el tiempo de ejecución es $O(n^2)$.
- $f(n) = (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2$
 $O(n^2)$. 
- El número de intercambios es $O(n)$.
- Por tanto complejidad **$O(n^2)$** .



Desventajas Ordenación por Selección.

- Muy lenta con arrays grandes.
- No detecta si el array está todo ni parcialmente ordenado.



Ordenación por Intercambio o Burbuja

- Se basa en el principio de **comparar e intercambiar pares de elementos adyacentes hasta que todos estén ordenados.**
- Desde el primer elemento hasta el penúltimo no ordenado
 - comparar cada elemento con su sucesor
 - intercambiar si no están en orden

Código Ordenación por Burbuja

```
public void OrdBurbuja(){  
    for (int pasada=0; pasada < A.length-1; pasada++)  
        for (int j=0; j < (A.length-pasada-1); j++)  
            if (A[j] > A[j+1]) intercambiar(j,j+1);  
}
```

Si tenemos en cuenta que tras una pasada puede suceder que ya estén todos los elementos ordenados, en este caso no sería necesario seguir realizando comparaciones.

Mejora Ordenación Burbuja

```
public void OrdBurbuja2()
{
    boolean noOrdenados = true;
    int pasada = 0;
    while ((noOrdenados) && (pasada < A.length - 1))
    {
        noOrdenados = false;
        for (int j=0; j < (A.length-pasada-1); j++)
            if (A[j] > A[j+1])
            {
                intercambiar(j,j+1);
                noOrdenados = true;
            }
        pasada++;
    }
}
```

Complejidad Ordenación Burbuja

- El tiempo de ejecución de dicho algoritmo viene determinado por el número de comparaciones, en el peor de los casos $O(n^2)$.
- COMPARACIONES: $(n-1) + (n-2) + \dots + 3 + 2 + 1 = n(n-1)/2 \Rightarrow O(n^2)$
- INTERCAMBIOS: $(n-1) + (n-2) + \dots + 3 + 2 + 1 = n(n-1)/2 \Rightarrow O(n^2)$



Ventajas y Desventajas Ordenación Burbuja

- Su principal ventaja es la simplicidad del algoritmo.
- El problema de este algoritmo es que solo compara los elementos adyacentes del array. Si el algoritmo comparase primero elementos separados por un amplio intervalo y después se centrarse progresivamente en intervalos más pequeños, el proceso sería más eficaz. Esto llevo al desarrollo de ordenación Shell y QuickSort.

Ordenación Shell

- Es una mejora de la ordenación por inserción (colocar cada elemento en su posición correcta, moviendo todos los elementos mayores que él, una posición a la derecha), que se utiliza cuando el número de datos a ordenar es grande.
- Para ordenar una secuencia de elementos se procede así: **se selecciona una distancia inicial y se ordenan todos los elementos de acuerdo a esa distancia**, es decir, cada elemento separado de otro a *distancia* estará ordenado con respecto a él. Se disminuye esa distancia progresivamente, hasta que se tenga distancia 1 y todos los elementos estén ordenados.

Ejemplo Ordenación Shell

Posición	0	1	2	3	4	5	6	7	8
Original	4	14	21	32	18	17	26	40	6
Distancia 4	4	14	21	32	6	17	26	40	18
Distancia 2	4	14	6	17	18	32	21	40	26
Distancia 1	4	14	6	17	18	32	21	40	26
Final	4	6	14	17	18	21	26	32	40

Ordenación
por
Insercción

Código Ordenación Shell

```
public void OrdShell ()
{
    int intervalo = A.length /2; /* distancia, los elementos estará
                                   ordenados a una distancia igual a intervalo.*/
    while (intervalo > 0) //Se ordenan hasta que la distancia sea igual a 1
    {
        for (int i = intervalo; i < A.length; i++)
        {
            int elem=A[i];
            int j = i - intervalo; // Posicion del elemento a comparar
            while ((j > -1) && (elem<A[j]))
            {
                A[j+intervalo]=A[j];
                j -= intervalo;
            }
            A[j+intervalo]=elem;
        }
        intervalo = intervalo /2;
    }
}
```

Complejidad de Shell

- No es fácil de calcular. $O(n^{3/2})$. Si se divide la distancia por 2.2 obtiene una complejidad de $O(n^{5/4})$.
- El tiempo de ejecución depende de la secuencia de incrementos que se elige.
- El rendimiento de la ordenación Shell es bastante aceptable en la práctica, aún para n grande.
- Su simplicidad lo hace adecuado para clasificar entradas moderadamente grandes.

Ordenación por mezcla (mergesort)

- Para ordenar una secuencia de elementos S , un procedimiento podría consistir en dividir S en dos subsecuencias disjuntas, ordenarlas de forma independiente y unir los resultados de manera que se genere la secuencia final ordenada. Dentro de esta filosofía es posible hacer una distinción de los algoritmos en dos categorías:
 - Algoritmos de fácil división y difícil unión, el *MERGESORT*.
 - Algoritmos de difícil división y fácil unión, el *QUICKSORT*.



Ordenación por mezcla (mergesort)

- El **MERGESORT** consiste en :
 - Dividir los elementos en dos secuencias de la misma longitud aproximadamente.
 - ***Ordenar*** de forma independiente cada subsecuencia.
 - ***Mezclar*** las dos secuencias ordenadas para producir la secuencia final ordenada.

Código MergeSort

```
public void mergeSort (int [] A, int bajo, int alto){  
    if (bajo <  alto) //Si hay más de un elemento  
    {  
        int medio = (alto + bajo)/2;  
        mergeSort (A, bajo, medio);  
        mergeSort (A, medio+1, alto);  
        //Procedimiento que mezcla el resultado de  
        las dos llamadas anteriores  
        merge (A, bajo, medio+1, alto);  
    }  
}
```



Código Mezcla de MergeSort.

- El proceso de ***mezcla*** es el siguiente:
 - Repetir mientras haya elementos en una de las dos secuencias:
 - Seleccionar el menor de los elementos de las subsecuencias y añadirlo a la secuencia final ordenada.
 - Eliminar el elemento seleccionado de la secuencia a la que pertenece.
 - Copiar en la secuencia final los elementos de la subsecuencia en la que aún quedan elementos.

Código Mezcla de MergeSort.

```
public void merge (int [] A, int bajo, int bajo_2, int alto)
{
    int i = bajo; //Variable de primer elemento de la primera subsecuencia
    int finbajo = bajo_2 -1; //Variable del último elemento de la primera subsecuencia
    int j = bajo_2; //Variable del primer elemento de la segunda subsecuencia
    int k = bajo;
    /* Temp es un array ya definido*/
    while (( i <=  finbajo) && (j<= alto))
    {
        if (A[i] <= A[j])
            Temp[k++] = A[i++];
        else
            Temp[k++] = A[j++];
    }

    while (i <= finbajo) //Si se agotaron todos los elementos de la segunda subsecuencia
        Temp[k++] = A[i++];

    while (j <= alto) //Si se agotaron los de la primera subsecuencia
        Temp[k++] = A[j++];
    //Paso todos los elementos del Temporal al array
    for (i = bajo; i <=  alto; i++)
        A[i] = Temp[i];
}
```



Complejidad Mezcla de MergeSort.

- Teniendo en cuenta que la entrada consiste en el total de elementos n y que cada comparación asigna un elemento a Temp. El número de comparaciones es $(n-1)$ y el número de asignaciones es n . Por lo tanto, el algoritmo de mezcla se ejecuta en un tiempo lineal **$O(n)$** .

Complejidad MergeSort

- El análisis de eficiencia de la ordenación por mezcla da lugar a una ecuación recurrente para el tiempo de ejecución.
- Suponemos n potencia de 2, $n = 2^k$
 $\text{Log}_2 n = k$
- Para $N = 1$, Tiempo constante.
- Para $N > 1$, El tiempo de ordenación para n números es igual al tiempo para 2 ordenaciones recursivas de tamaño $n/2$ + el tiempo para mezclar (que es lineal).
- Por tanto, $T(n) = 2 T(n/2) + n$
Tiempo de Mezclar

Tiempo de Ordenación

Complejidad MergeSort

- Para resolver la ecuación se divide por n .
- $T(n) / n = (2 (T (n/2)) / n) + (n/n)$
- $T(n) / n = T(n/2) / (n/2) + 1$
- Esta ecuación es válida para cualquier potencia de 2, así que se puede escribir:
- $T(n/2) / (n/2) = (T(n/4) / (n/4)) + 1$
- $T(n/4) / (n/4) = (T(n/8) / (n/8)) + 1$
- Así sucesivamente,
hasta $T(2) = (T(1) / 1) + 1$

Complejidad MergeSort

- Se suman todas las ecuaciones anteriores, como todos los términos se anulan, se obtiene el siguiente resultado:
 - $T(n) / n = (T(1) / 1) + k$, siendo K el número de ecuaciones que tenemos, es decir, el número de divisiones a la mitad que realizamos, por tanto $k = \log_2 n$
 - Para resolver la ecuación se dividió entre n , por tanto ahora multiplicamos por n .
 - $T(n) = n + n \log n$. Por tanto la complejidad del algoritmo MergeSort es de **$O(n \log n)$**



Ordenación Rápida (QuickSort)

- En la ordenación rápida, **la secuencia inicial de elementos se divide en dos subsecuencias de diferente tamaño**. La obtención de las dos subsecuencias es el proceso que acarrea más tiempo mientras que la combinación de las subsecuencias ordenadas para obtener la secuencia final consume muy poco tiempo.
- Para dividir en dos la secuencia de elementos, se selecciona un elemento sobre el cual efectuar la división, el ***PIVOTE***. Se dividen los elementos en dos grupos, los elementos menores que el pivote y aquellos mayores o igual al pivote.



Ordenación QuickSort.

- La elección del elemento Pivote se puede seleccionar de diferentes formas:
 - El mayor de los dos primeros elementos distintos encontrados.
 - El primer elemento.
 - El último elemento.
 - El elemento medio.
 - Un elemento aleatorio.
 - Mediana de tres elementos (El primer elemento, el elemento del medio y el último elemento).

Pasos a seguir QuickSort.

- El método de ordenación rápida se basa en **ordenar los elementos comprendidos entre A_i y A_j conforme a las siguientes cinco etapas:**
 1. Si desde A_i a A_j hay al menos dos elementos distintos entonces comenzar la aplicación del algoritmo.
 2. **Seleccionar el PIVOTE** como el elemento mayor de los dos primeros elementos distintos encontrados.
 3. **Insertar PIVOTE en la última posición.**
 4. **Permutar los elementos** desde A_i hasta A_j de modo que, para algún $i \leq k \leq j$:
 $A_i, \dots, A_{k-1} < \text{PIVOTE}$
 $A_k, \dots, A_j \geq \text{PIVOTE}$
Es decir, en las $(k-1)$ primeras posiciones queden los elementos menores que pivote, mientras que en la posición k hacia delante queden los elementos mayores o iguales que el pivote.
 5. Invocar a:
QUICKSORT desde i hasta $(k - 1)$
QUICKSORT desde k hasta j

Paso 2: Elección del Pivote

- Para la elección del pivote se puede utilizar la siguiente función que localiza el elemento mayor de los dos primeros elementos distintos existentes entre el i y el j.

```
int buscaPivote (int i, int j)
{
    int primer = A[i];
    int k = i + 1;

    while (k <= j)
    {
        if (A[k] > primer)
            return k;
        else if (A[k] < primer)
            return i;
        else k++;
    }
    //Si llega al final del array y todos los elementos son iguales, o si sólo hay un elemento
    return -1;
}
```

Paso 4: Permutación de elementos

- Para el paso 4 de permutación de los elementos se utilizan dos cursores:
 - D : para mover a la derecha mientras el elemento sea menor que el pivote.
 - I : para mover a la izquierda mientras el elemento sea mayor o igual que el pivote
- de acuerdo a tres fases :
 - CAMBIO : Si $D < I$ se intercambian A_D y A_I , con lo cual probablemente:
 $A_D < \text{PIVOTE}$ y
 $A_I \geq \text{PIVOTE}$
 - EXPLORACIÓN: Mover
D hacia la derecha sobre cualquier elemento MENOR que el pivote y
I hacia la izquierda sobre cualquier elemento MAYOR o IGUAL que el pivote.
 - COMPROBACIÓN: Si $D > I$ hemos acabado con éxito la reordenación.

Código, paso 4.

```
int particion (int i, int j, int pivote)
{
    int derecha = i;
    int izquierda = j-1;
    while (derecha <= izquierda)
    {
        intercambiar(derecha,izquierda);
        while (A[derecha] < pivote)
            derecha++;
        while (A[izquierda] >= pivote)
            izquierda--;
    }
    return derecha;
}
```

Código QuickSort

```
void quickSort (int [] A, int i, int j)
{
    int indicePivote = buscaPivote(i, j);
    if (indicePivote != -1)
    {
        int pivote = A[indicePivote];
        intercambiar(indicePivote,j);
        int k = particion(i,j,pivote);
        quickSort(A, i, k-1);
        quickSort(A, k,j);
    }
}
```




Complejidad QuickSort.

- **Caso Mejor:** Cuando el pivote, divide al conjunto en dos subconjuntos de igual tamaño. En este caso hay dos llamadas con un tamaño de la mitad de los elementos, y una sobrecarga adicional lineal, igual que en MergeSort. En consecuencia el tiempo de ejecución es **$O(n \log n)$** .

Complejidad QuickSort.

- **Caso Peor:** Se podría esperar que los subconjuntos de tamaño muy distinto proporcionen resultados malos.
- Supongamos que en cada paso de recursión sólo hay un elemento menor a pivote. En tal caso el subconjunto I (elementos menores que pivote) será uno y el subconjunto D (elementos mayores o igual a pivote) serán todos los elementos menos uno. El tiempo de ordenar 1 elemento es sólo 1 unidad, pero cuando $n > 1$.
- $T(N) = T(N-1) + N$
- $T(N-1) = T(N-2) + (N-1)$
- $T(N-2) = T(N-3) + (N-2) \dots$
- $T(2) = T(1) + 2$
- $T(N) = T(1) + 2 + 3 + 4 + \dots + N = N(N+1)/2$, $O(n^2)$.



Complejidad QuickSort.

- **Caso Medio:** Complejidad $O(n \log n)$.