Búsqueda y ordenamiento

Temario

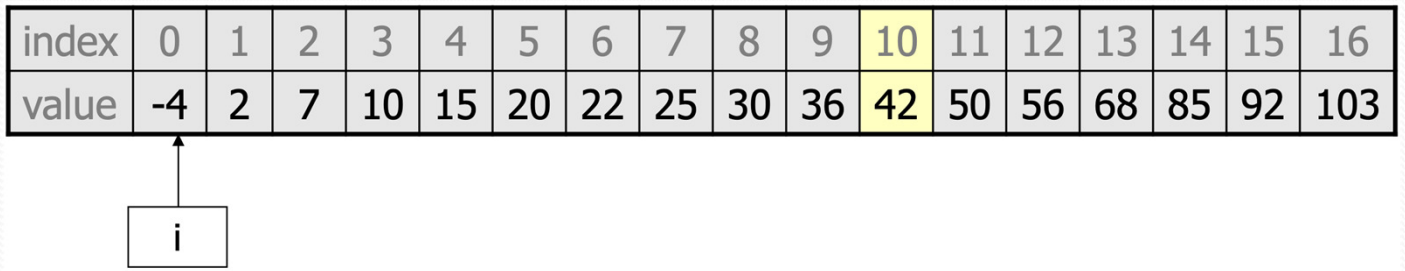
- Búsqueda (searching).
- Complejidad.
- Ordenamiento (sorting).

Búsqueda

- **Búsqueda**. Dada una colección de datos y un dato, buscar el dato en la colección.
- El resultado de la búsqueda puede ser:
  - true / false.
  - El índice dentro de la colección (o –1 si el dato no está).
- Se verán dos algoritmos de búsqueda: búsqueda secuencial y búsqueda binaria.

Búsqueda secuencial

- **Búsqueda secuencial**. Localiza un valor en una colección examinando cada elemento de principio a fin.
- ¿Cuántos elementos se necesitan examinar?
- Ejemplo: buscar el valor 42 en el siguiente arreglo:



Implementación búsqueda secuencial

```
// Devuelve el índice de la primera ocurrencia de value en a o -1 si no está
public static int indexOf(int[] a, int value)
{
    for (int i = 0; i < a.length; i++) {
        if (a[i] == value) {
            return i;
        }
    }
    return –1;
}
```

Búsqueda binaria

- **Búsqueda binaria**. Localiza un valor en una colección **ordenada** eliminando la mitad del arreglo en cada comparación.
- Algoritmo para buscar un valor V:

1. Obtener el elemento de en medio, M.
2. Comparar V y M.
3. Si V es igual a M regresa éxito.
4. Si $V > M$ buscar en la mitad derecha.
5. Si $V < M$ buscar en la mitad izquierda.
6. Ir al paso 1 si todavía hay elementos, en otro caso regresa falla.

Búsqueda binaria

- Ejemplo: buscar el valor 42 en el arreglo siguiente:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Clase Arrays

- La clase Arrays en java.util tiene métodos para realizar búsquedas binarias.

binarySearch(array, value)	Devuelve el índice del valor dado en un arreglo *ordenado* (o < 0 si no se encuentra)
binarySearch(array, minIndex, maxIndex, value)	Devuelve el índice del valor dado en un arreglo *ordenado* entre los índices *min* / *max* – 1 (o < 0 si no se encuentra)

Arrays.binarySearch

- El método Arrays.binarySearch busca en un arreglo de manera eficiente si el arreglo está ordenado.
- Si el arreglo no está ordenado, se debe ordenar primero.

Usando binarySearch

```
// índice   0  1  2  3   4   5   6   7   8   9  10  11  12  13 14  15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};
int index1 = Arrays.binarySearch(a, 42);        // 10
int index2 = Arrays.binarySearch(a, 21);        // –7
```

- binarySearch devuelve el índice donde se encuentra el valor.
- Si no se encuentra el valor, binarySearch regresa –(insertionPoint + 1).
- Donde insertionPoint es el índice donde *habría* estado el elemento, si estuviera en el arreglo ordenado.
- Para obtener insertionPoint:

```
int insertionPoint = –(index2 + 1);       // 6
```

Implementación búsqueda binaria

- Escribir un método binarySearch.
- Si no encuentra el valor, devuelve su punto de inserción negativo.
- Ejemplos:

```
// índice      0  1  2  3   4   5   6   7   8   9  10  11  12  13  14  15  16
int[] data = {-4, 2, 7, 10, 15, 20, 22, 25, 30, 36, 42, 50, 56, 68, 85, 92, 103};
int index1 = binarySearch(data, 42);    // 10
int index2 = binarySearch(data, 66);    // –14
```

Implementación de búsqueda binaria

```
// Devuelve el índice de una ocurrencia de target en a,
// o un número negativo si target no se encuentra.
// Precondición: los elementos de a están ordenados
public static int binarySearch(int[] a, int target)
{
    int min = 0;
    int max = a.length – 1;
```

Implementación de búsqueda binaria

```
    while (min <= max) {
        int mid = (min + max) / 2;
        if (target > a[mid]) {
            min = mid + 1;              // mitad derecha
        } else if (target < a[mid]) {
            max = mid – 1;              // mitad izquierda
        } else {
            return mid;                 // se encontró target
        }
    }
    return –(min + 1);                  // no se encontró target
}
```

Búsqueda binaria recursiva

- Escribir un método binarySearch recursivo.

Búsqueda binaria recursiva

```
// Devuelve el índice de una ocurrencia del valor dado en
// el arreglo dado, o un número negativo si no se encuentra.
// Precondición: los elementos del arreglo están ordenados
public static int binarySearch(int[] a, int target)
{
    return binarySearch(a, target, 0, a.length – 1);
}
```

Búsqueda binaria recursiva

```
// Ayudante recursivo para implementar el comportamiento de búsqueda.
private static int binarySearch(int[] a, int target, int min, int max)
{
    if (min > max) {
        return –(min + 1);          // no se encontró target
    }
    else {
        int mid = (min + max) / 2;
```

Búsqueda binaria recursiva

```
        if (target > a[mid]) {              // mitad derecha
            return binarySearch(a, target, mid + 1, max);
        }
        else if (target < a[mid]) {         // mitad izquierda
            return binarySearch(a, target, min, mid – 1);
        }
        else {
            return mid;                     // se encontró target
        }
    }
}
```

Búsqueda binaria en arreglos de objetos

- ¿Se puede usar Arrays.binarySearch con un arreglo de strings?
- Si. Java sabe comparar objetos String.
- El arreglo debe estar en orden alfabético.
- ¿Se puede usar Arrays.binarySearch con un arreglo de objetos arbitrarios?
- Si, siempre y cuando los objetos se puedan comparar.

Búsqueda binaria en arreglos de objetos

- Para usar Arrays.binarySearch con un arreglo de objetos arbitrarios, hay dos opciones:

1. Hacer que la clase implemente la interface Comparable.
2. Utilizar el método definido en la clase Arrays:

```
static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
```

- Pasando un objeto Comparator en el tercer argumento.

Ejemplo

```
Person[] people = new Person[4];
people[0] = new Person(555, “Ana”);
people[1] = new Person(888, “Blanca”);
people[2] = new Person(111, “Carlos”);
people[3] = new Person(222, “Daniel”);
// Persona a buscar
Person p = new Person(888, “Blanca”);
```

Ejemplo

```
// Búsqueda binaria por nombre
Arrays.sort(people, Comparator.comparing(Person::name));
int index = Arrays.binarySearch(people, p,
                                Comparator.comparing(Person::name));
System.out.println("index binary search: " + index); // 1
// Búsqueda binaria por id
Arrays.sort(people, Comparator.comparing(Person::id));
index = Arrays.binarySearch(people, p, Comparator.comparing(Person::id));
System.out.println("index: " + index);          // 3
```

Búsqueda secuencial en listas

- La clase ArrayList define el método:

  int indexOf(Object o)

- Regresa el índice de la primera ocurrencia del objeto en la lista o –1 si no está.
- La clase de la lista debe implementar el método equals.

Búsqueda binaria en listas

- La clase Collections define los métodos:

  static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

  static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)

- Para usar el primer método, la clase de la lista debe extender a Comparable.
- Para usar el segundo método, basta con definir un objeto Comparator.

Ejemplo

```
List<Person> list = new ArrayList<>();
list.add(new Person(555, “Ana”));
list.add(new Person(888, “Blanca”));
list.add(new Person(111, “Carlos”));
list.add(new Person(222, “Daniel”));
// Persona a buscar
Person p = new Person(888, “Blanca”);
// Búsqueda secuencial
index = list.indexOf(p);
System.out.println(“index linear search: ” + index);  // 1
```

Ejemplo

```
// Búsqueda binaria por id
list.sort(Comparator.comparing(Person::id));
index = Collections.binarySearch(list, p, Comparator.comparing(Person::id));
System.out.println("index binary search: " + index); // 3
```

Conclusión

- Para utilizar el método indexOf con listas de objetos arbitrarios, la clase debe sobreponer el método equals.
- Para utilizar los métodos binarySearch y sort con arreglos o listas de objetos arbitrarios hay dos opciones:

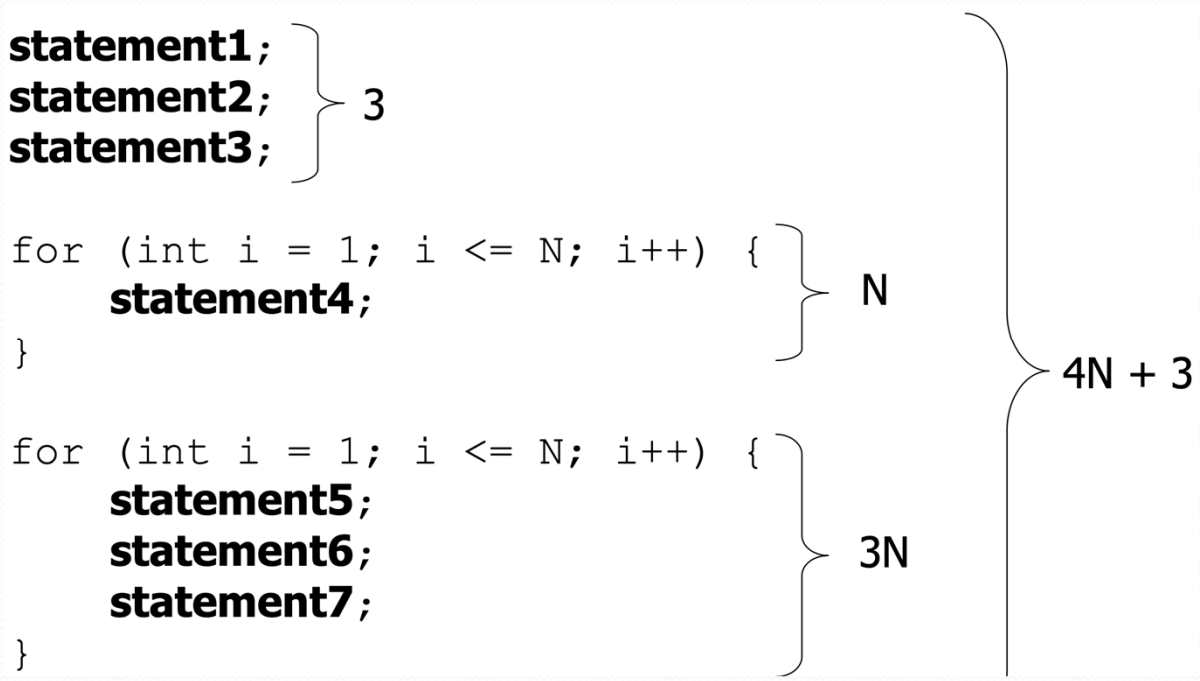
a) Que la clase implemente la interface Comparable.

b) Pasar una expresión lambda o una referencia a método a los métodos binarySearch y sort.

Complejidad

- **Complejidad**. Mide el uso de recursos computacionales utilizado por un algoritmo.
  - Puede ser relativo a la velocidad (tiempo) o la memoria (espacio).
  - Comúnmente se refiere al tiempo de ejecución.
- Suponer lo siguiente:
  - Todas las instrucciones de Java tardan la misma cantidad de tiempo en ejecutarse.
  - El tiempo de ejecución de una llamada a un método es el total de instrucciones que ejecuta el método.
  - Si un ciclo se repite N veces su tiempo de ejecución es N veces el tiempo de ejecución de las instrucciones en su cuerpo.

Ejemplo 1



Fuente: BJP, p. 853

Ejemplo 2

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}

for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

N^2

4N

$N^2 + 4N$

Fuente: BJP, p. 854

- ¿Cuántas instrucciones se ejecutan si N = 10? ¿Si N = 1000?

Tasas de crecimiento de un algoritmo

- **Tasa de crecimiento**. Es el cambio en el tiempo de ejecución a medida que N cambia.
- El tiempo de ejecución se mide en proporción al número de datos de entrada, N.

Tasas de crecimiento del algoritmo

- Ejemplo: suponer que un algoritmo ejecuta $0.5 \times N^3 + 8 \times N + 17$ instrucciones.
- Considerar el tiempo de ejecución cuando N es extremadamente grande (por ejemplo $N = 1{,}000{,}000 = 10^6$).
- Resultado: $0.5 \times 10^{18} + 8 \times 10^6 + 17 \approx 10^{18}$.
- El término de orden mayor ($0.5 \times N^3$) domina el tiempo de ejecución.
- Se dice que este algoritmo se ejecuta "en el orden de" N^3.
- Para abreviar se escribe $O(N^3)$ ("Gran O de N al cubo")

Clases de complejidad

- **Clase de complejidad.** Una categoría de eficiencia del algoritmo basada en la relación del algoritmo con el tamaño de la entrada N.

Clase	**Gran O**	**Si se duplica N**
Constante	$O(1)$	No cambia
Logarítmica	$O(\log_2 N)$	Incrementa un poco
Lineal	$O(N)$	Duplica
Log-lineal	$O(N \log_2 N)$	Poco más que el doble
Cuadrática	$O(N^2)$	Cuadruplica
Cúbica	$O(N^3)$	Octuplica
…	…	…
Exponencial	$O(2^N)$	Crece drásticamente

Ejemplo

- Suponer que una instrucción corre en 1 ms = 10^{-3} s.

N	O(1)	O($\log_2$ N)	O(N)	O(N $\log_2$ N)	O(N^2)	O(N^3)	O(2^N)
10	1 ms	3.3 ms	10 ms	33.2 ms	100 ms	1 s	1 s
20	1 ms	4.3 ms	20 ms	86.4 ms	400 ms	8 s	17.5 m
40	1 ms	5.3 ms	40 ms	212.9 ms	1.6 s	64 s	34.8 y
100	1 ms	6.6 ms	100 ms	664.4 ms	10 s	16.7 m	$4x10^9$ y
1,000	1 ms	10 ms	1 s	10 s	16.7 m	11.6 d	$1.2x10^{293}$ y
100,000	1 ms	16.6 ms	1.7 m	27.7 m	115.7 d	$3.2x10^4$ y	NaN

Fuente: BJP, p. 860

Complejidad de la búsqueda binaria

- Para un arreglo de tamaño N, en cada paso se elimina la mitad hasta que quede un elemento: $N, N / 2, N / 4, N / 8, ..., 4, 2, 1$
- ¿Cuántas divisiones se necesitan?
- Pensando en la otra dirección: comenzando en uno ¿cuántas veces hay que multiplicar por 2 para llegar a N?
- $1, 2, 4, 8, ..., N / 4, N / 2, N$
- A este número de multiplicaciones se le llama x.
- $2^x = N$
- $x = \log_2 N$
- La búsqueda binaria está en la clase de complejidad **logarítmica**.

Ejemplo: obtener el rango

- Dado un arreglo de números, el rango es la diferencia entre los elementos máximo y mínimo del arreglo.
- Ejemplo:

```
int[] a = {17, 29, 11, 4, 20, 8};
int r = rango(a);              // devuelve 25
```

- ¿Cuál es la complejidad del siguiente algoritmo?

Algoritmo rango v1

```
public static int range(int[] numbers) {
    int maxDiff = 0;
    for (int i = 0; i < numbers.length; i++) {
        for (int j = 0; j < numbers.length; j++) {   // revisa cada pareja de valores
            int diff = Math.abs(numbers[j] – numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return diff;
}
```

Algoritmo rango v1

- El algoritmo es de clase $O(N^2)$.
- ¿Se puede mejorar?
- Si.

Algoritmo rango v2

```
public static int range(int[] numbers)
{
    Arrays.sort(numbers);          // O(N log2 N)
    return numbers[numbers.length – 1] – numbers[0];          // O(1)
}
```

- Ahora es de clase O($N \log_2 N$).
- ¿Se puede mejorar?
- Si.

Algoritmo rango v3

```
public static int range(int[] numbers) {
    int max = numbers[0];        // find max / min values
    int min = max;
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] < min) {
            min = numbers[i];
        }
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max - min;
}
```

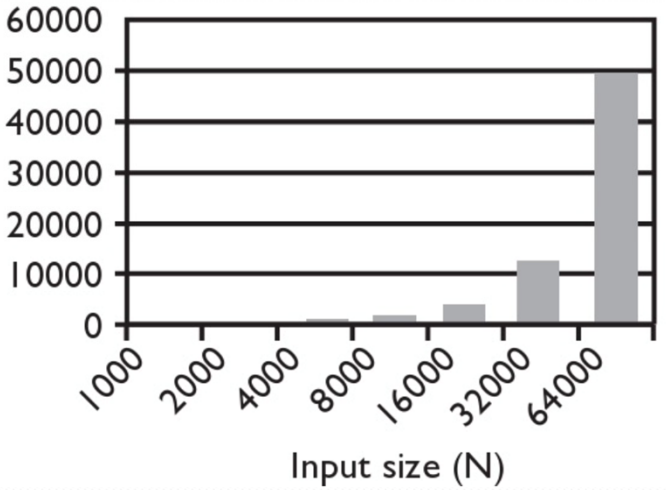
Algoritmo rango v3

- Ahora es O(N).

Versión 1 O(N^2)

- Version 1:

N	Runtime (ms)
1000	15
2000	47
4000	203
8000	781
16000	3110
32000	12563
64000	49937

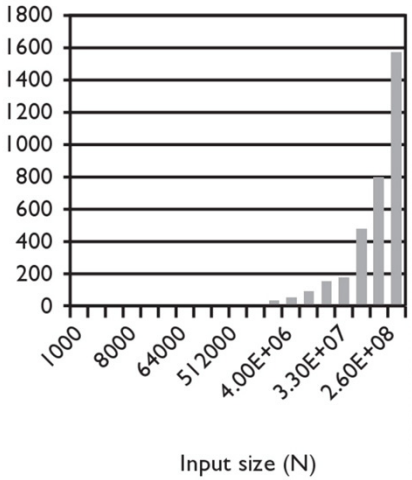


Fuente: BJP, p. 857

Versión 3 O(N)

- Version 3:

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	0
64000	0
128000	0
256000	0
512000	0
1e6	0
2e6	16
4e6	31
8e6	47
1.67e7	94
3.3e7	188
6.5e7	453
1.3e8	797
2.6e8	1578



Fuente: BJP, p. 858

Ordenamiento

- **Ordenamiento**. Organizar los valores en un arreglo o colección en un orden específico (generalmente en su orden natural).
- Hay muchos algoritmos de ordenamiento.
- Algunos son más rápidos o más lentos que otros.
- Algunos usan más o menos memoria que otros.
- Algunos funcionan mejor con tipos específicos de datos.
- Algunos pueden utilizar múltiples procesadores.

Ordenamiento

- El ordenamiento está basado en la comparación.
- La determinación del orden es mediante la comparación de pares de elementos: <, >, compareTo, ...

Algunos algoritmos de ordenamiento

- **Selection sort**. Selecciona los elementos menores de la parte no ordenada y los pone en su lugar en la parte ordenada.
- **Bubble sort**. Intercambia pares adyacentes que están fuera de orden.
- **Insertion sort**. Inserta en su lugar los elementos de la parte no ordenada en la parte ordenada.
- **Merge sort**. Divide recursivamente el arreglo por la mitad y lo ordena.
- **Heap sort**. Ordena los elementos utilizando un heap.
- **Quick sort**. Parte el arreglo recursivamente en base a un pivote, moviendo los menores a la izquierda y los mayores a la derecha.

Algunos algoritmos de ordenamiento

- Algoritmos de ordenamiento especializados:
- **Bucket sort**. Agrupa los elementos en cubetas, luego ordena cada cubeta.
- **Radix sort**. Ordena enteros por el último dígito, luego el penúltimo, y así hasta ordenarlos por el primer dígito.

Selection sort

- **Selection sort**. Ordena una lista de valores colocando repetidamente el valor más pequeño o más grande que no esté ya en su posición final.
- Algoritmo:
  - Examinar la lista para encontrar el valor más pequeño.
  - Intercambiarlo para que esté en el índice 0.
  - Examinar el resto de la lista para encontrar el segundo valor más pequeño.
  - Intercambiarlo para que esté en el índice 1.
  - ...
  - Repetir hasta que todos los valores estén en sus lugares correctos.

Ejemplo de Selection sort

- Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	**-4**	18	12	**22**	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	**2**	12	22	27	30	36	50	7	68	91	56	**18**	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	**7**	22	27	30	36	50	**12**	68	91	56	18	85	42	98	25

Código de Selection sort

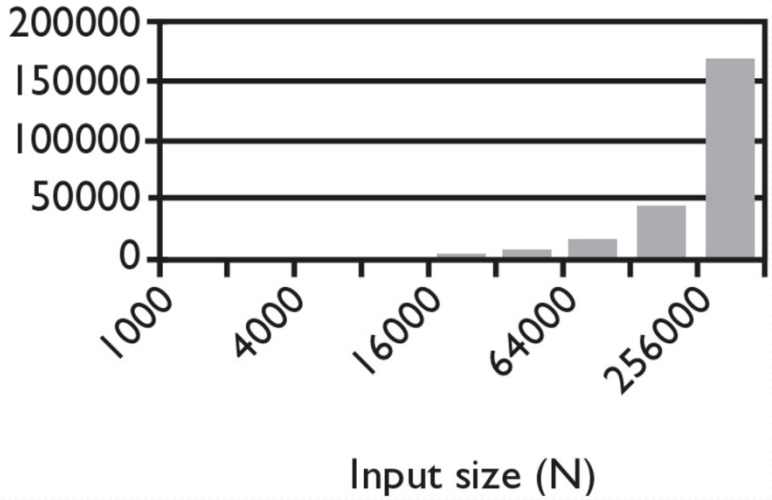
```
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length – 1; i++) {
        int min = i;      // encuentra el índice del valor no colocado más pequeño
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        swap(a, i, min);          // intercambia el valor más pequeño en su lugar, a[i]
    }
}
```

Complejidad de Selection sort

- Mejor caso (arreglo ordenado): $O(N^2)$.
- Peor caso (arreglo ordenado al revés): $O(N^2)$.
- Caso medio (arreglo aleatorio): $O(N^2)$.

Complejidad de Selection sort

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Fuente: BJP, p. 872

Bubble sort

- **Bubble sort**. Recorre el arreglo repetidas veces, intercambiando valores adyacentes que están fuera de orden.
- Más lento que el Selection sort (tiene que hacer más intercambios).

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	18	12	-4	22	27	30	36	7	50	68	56	2	85	42	91	25	98

Bubble sort

```
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    int temp = 0;
     for(int i = 0; i < n; i++){
            for(int j = 1; j < (n – i); j++){
                    if(arr[j – 1] > arr[j]){
                        // swap elements
                        temp = arr[j – 1];
                        arr[j – 1] = arr[j];
                        arr[j] = temp;
                    }
            }
     }
}
```

Bubble sort mejorado

```
public static void bubbleSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        boolean is_sorted = true;
        // skip the already sorted largest elements
        for (int j = 0; j < a.length – i; j++) {
            if (a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
                is_sorted = false;
            }
        }
        if (is_sorted) return;
    }
}
```

Complejidad de Bubble sort

- Mejor caso (arreglo ordenado): $O(N)$.
- Peor caso (arreglo ordenado al revés): $O(N^2)$.
- Caso medio (arreglo aleatorio): $O(N^2)$.

Insertion sort

- **Insertion sort**. Inserta cada elemento de la parte no ordenada de un arreglo en la parte ordenada.
- Más rápido que el Selection sort (examina menos valores).

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	12	18	22	27	30	36	50	7	68	91	56	2	85	42	98	25

sorted sub-array (indexes 0-7)

Insertion sort

```
public static void insertSort(int[] A) {
    for(int i = 1; i < A.length; i++) {
        int value = A[i];
        int j = i – 1;
        while(j >= 0 && A[j] > value) {
            A[j + 1] = A[j];
            j = j – 1;
        }
        A[j + 1] = value;
    }
}
```

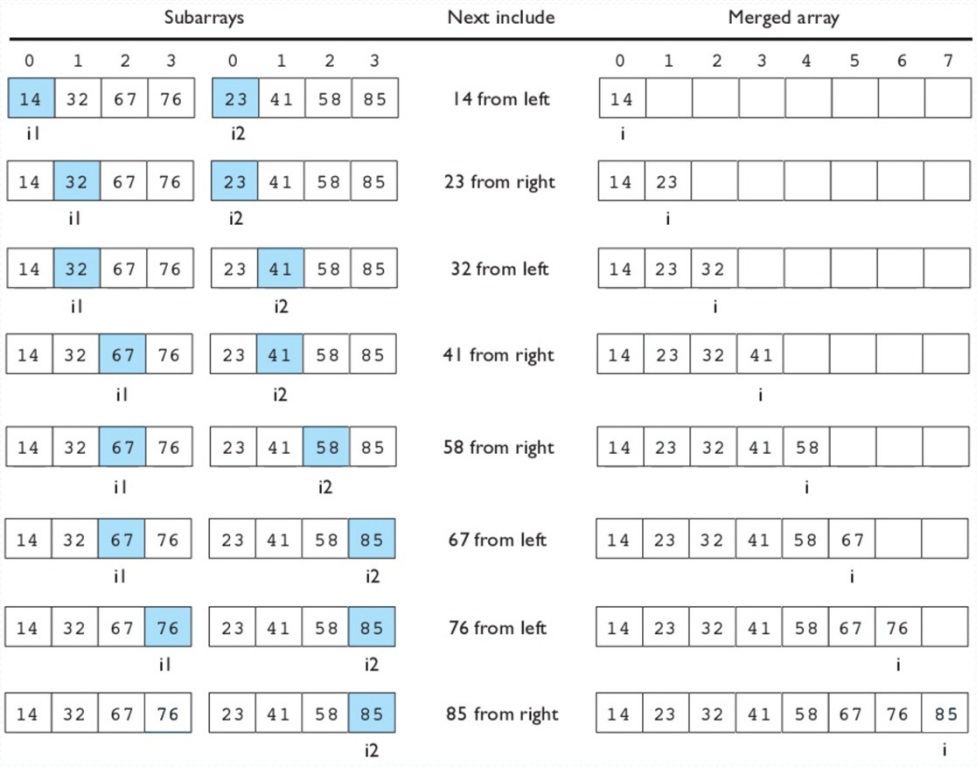
Complejidad de Insertion sort

- Mejor caso (arreglo ordenado): $O(N)$.
- Peor caso (arreglo ordenado al revés): $O(N^2)$.
- Caso medio (arreglo aleatorio): $O(N^2)$.

Merge sort

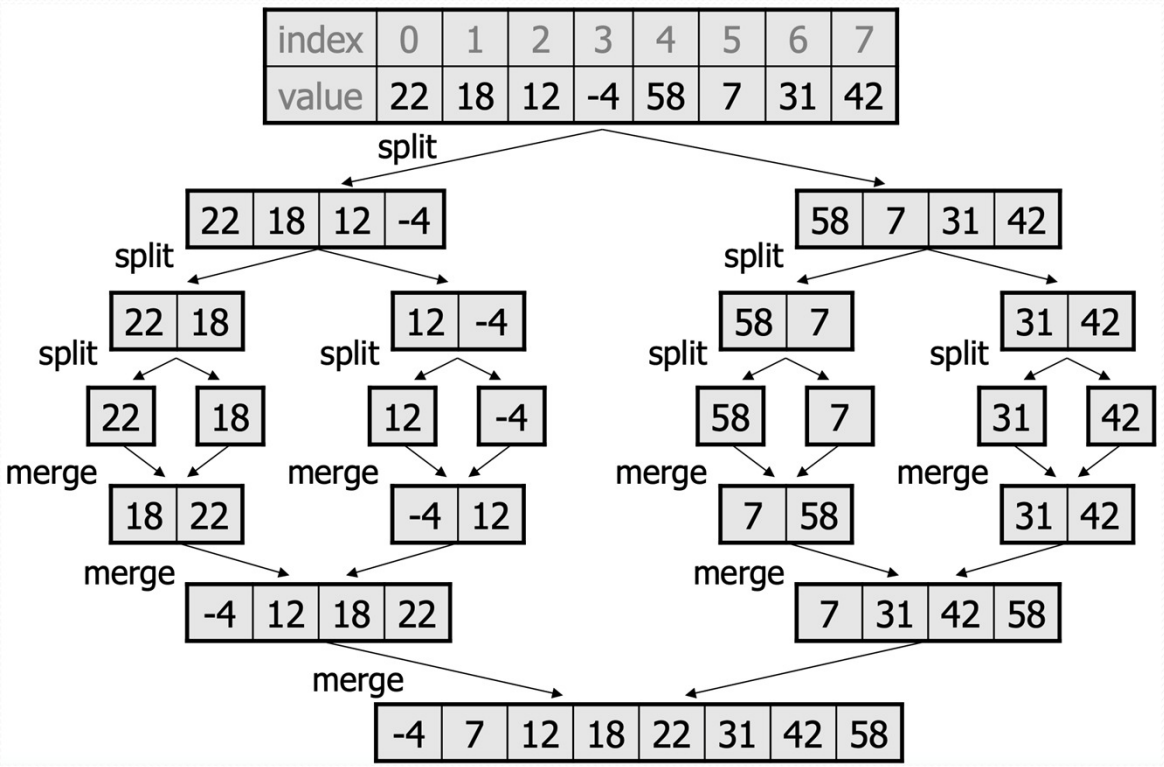
- **Merge sort**. Divide repetidamente los datos por la mitad, ordena cada mitad y combina (merge) las mitades ordenadas en un todo ordenado.
- Algoritmo:
  - Divide la lista en dos mitades aproximadamente iguales.
  - Ordena la mitad izquierda.
  - Ordena la mitad derecha.
  - Combina (merge) las dos mitades ordenadas en una lista ordenada.
- A menudo se implementa de forma recursiva.
- Ejemplo de un algoritmo “divide y vencerás”.
- Inventado por John von Neumann en 1945

Operación merge



Fuente: BJP, p. 874

Ejemplo de Merge sort



Código de la operación merge

```
// Merge los elementos izquierdo / derecho en un resultado ordenado.
// Precondición: izquierda / derecha están ordenadas
public static void merge(int[] result, int[] left, int[] right) {
    int i1 = 0;         // índice del arreglo left
    int i2 = 0;         // índice del arreglo right
    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length || (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];         // toma de left
            i1++;
        }
```

Código de la operación merge

```
        else {
            result[i] = right[i2];        // toma de right
            i2++;
        }
    }
}
```

Código de Merge sort

```
public static void mergeSort(int[] a) {
   if (a.length >= 2) {
      // parte el arreglo en dos mitades
      int[] left = Arrays.copyOfRange(a, 0, a.length / 2);
      int[] right = Arrays.copyOfRange(a, a.length / 2, a.length);
      // ordena las dos mitades
      mergeSort(left);
      mergeSort(right);
      // combina las dos mitades en un todo ordenado
      merge(a, left, right);
   }
}
```

Complejidad de Merge sort

- Mejor caso (arreglo ordenado): $O(N \log_2 N)$.
- Peor caso (arreglo ordenado al revés): $O(N \log_2 N)$.
- Caso medio (arreglo aleatorio): $O(N \log_2 N)$

Comparación

Método	Complejidad temporal			Complejidad espacial
	Mejor caso	**Caso medio**	**Peor caso**	**Peor caso**
Bubble sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Merge sort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N)$
Heap sort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(1)$
Quick sort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$	$O(\log_2 N)$
Radix sort	$O(N\ k)$	$O(N\ k)$	$O(N\ k)$	$O(N + k)$
Count sort	$O(N + k)$	$O(N + k)$	$O(N + k)$	$O(k)$
Bucket sort	$O(N + k)$	$O(N + k)$	$O(N^2)$	$O(N)$

Comparación

- N es el número de elementos.
- k es el número de dígitos (Radix sort).
- k es el rango de los números no negativos (Count sort).
- k es el número de cubetas (Bucket sort).

Resumen

- Búsqueda es la tarea de intentar encontrar un valor particular en una colección o en un arreglo.
- Ordenamiento es la tarea de organizar los elementos de una lista o arreglo en un orden natural.
- La biblioteca de clases de Java contiene varios métodos para buscar y ordenar arreglos y listas, como Arrays.binarySearch y Collections.sort.
- Un objeto Comparator describe una forma personalizada de comparar objetos, lo que permite buscar y ordenar arreglos o listas de estos objetos de varias formas.

Resumen

- Búsqueda secuencial es un algoritmo de búsqueda $O(N)$ que examina cada elemento de una colección hasta que encuentra el valor objetivo o se termina la colección.
- Búsqueda binaria es un algoritmo de búsqueda $O(\log_2 N)$ que opera en una colección ordenada y elimina sucesivamente la mitad de los datos hasta que encuentra el elemento o se termina la colección.
- Hay varios algoritmos de ordenamiento.
- Los algoritmos más sencillos de implementar son $O(N^2)$.
- Los algoritmos generales más eficientes son $O(N \log_2 N)$.

Resumen

- Análisis empírico es la técnica de ejecutar un programa o algoritmo para determinar su tiempo de ejecución.
- Análisis de algoritmos es la técnica de examinar el código o pseudocódigo de un algoritmo para hacer inferencias sobre su complejidad.
- Los algoritmos se agrupan en clases de complejidad, que se describen utilizando la notación gran O.
- La complejidad puede ser temporal (tiempo de ejecución) o espacial (memoria utilizada).