

# 75.40 Algoritmos y Programación I Curso 4

## Operaciones con Arreglos y Listas

Dr. Mariano Méndez<sup>1</sup>

<sup>1</sup>Facultad De Ingeniería. Universidad de Buenos Aires

23 de agosto de 2022

**Correcciones:** *Manuel Camejo - Martin Dardis - Ignacio Jordan.*

**Aportes:** *Gonzalo Hernandorena, Valentin Bonani, Andres Fernandez Bina, Nacho, Lucho, Charly Talavera, Javi.*

### 1. Introducción

Existe una serie de operaciones comunes que se realizan entre vectores ya estén compuestos estos por elementos de tipos de datos simples o estructurados. Entre otras operaciones se hallan:

- Insertar elementos en un vector manteniendo un orden.
- Eliminar elementos en un vector.
- Buscar un elemento dentro de un vector.
- Ordenar los elementos de un vector según algún criterio.
- Combinar los elementos de un vector obteniendo un tercer vector:
  - Obtener la Intersección.
  - Obtener la Mezcla.
  - Obtener la Diferencia.
  - Obtener la Unión.
  - Actualizar un vector con los datos de otro vector.

Se deberá tener en cuenta que todos los ejercicios están basados en la siguiente declaraciones:

```
1 #define MAX 1000
2 typedef int elemento_t;
3 typedef elemento_t vector_t[MAX];
```

## 2. Inserción Ordenada de un Elemento dentro de un Vector

La idea de este algoritmo es la de poder **insertar un elemento dentro de un vector manteniendo el orden del mismo**, para ello se insertará la siguiente secuencia de número 10, 5, 12, 2, 15, 7 y 6. La idea es básicamente la siguiente:

— arreglo vacío

insertar el 10

0 — tope  
10

insertar el 5

0 1 — tope  
5 10

insertar el 12

0 1 2 — tope  
5 10 12

insertar el 2

0 1 2 3 — tope  
2 5 10 12

insertar el 15

0 1 2 3 4 — tope  
2 5 10 12 15

insertar el 7

0 1 2 3 4 5 — tope  
2 5 7 10 12 15

La idea básica es que **si debe insertar un elemento, primero hay que buscar la posición correspondiente**, para ello **se avanza mientras  $v[i]$  menor que elemento a insertar**. Si por ejemplo se desea insertar el número 6, el procedimiento es el siguiente :

insertar el 6

0 1 2 3 4 5 — tope  
2 5 7 10 12 15  
6

insertar el 10

0 1 2 3 4 5 — tope  
2 5 6 7 12 15  
10

insertar el 7

0 1 2 3 4 5 — tope  
2 5 6 10 12 15  
7

insertar el 12

0 1 2 3 4 5 — tope  
2 5 6 7 10 15  
12

insertar el 15

0	1	2	3	4	5	6	— tope
2	5	6	7	10	12		
						15	

0	1	2	3	4	5	6
2	5	6	7	10	12	15

vector ordenado

## 2.1. El Algoritmo

```

1 #define FALSO 0
2 #define VERDADERO 1
3 #define MAX 1000
4
5 typedef int elemento_t;
6 typedef elemento_t vector_t[MAX];
7
8 void insertarOrdenado( vector_t vector, elemento_t elemento, int* tope){
9     int inserto;
10    int i;
11    elemento_t elem_aux;
12    inserto=FALSO;
13    for(i=0;i< (*tope); i++ ){
14        if (elemento < vector[i]){
15            elem_aux=vector[i];
16            vector[i]=elemento;
17            elemento = elem_aux;
18            inserto=VERDADERO;
19        }
20    }
21    if(inserto){
22        vector[*tope] = elem_aux;
23        *(tope)++;
24    }
25    if ((!inserto) || (tope==0) ){
26        vector[*tope] = elemento;
27        *(tope)++;
28    }
29 }

```

Code 1: Insertar Ordenado

## 2.2. Eliminar elementos en un vector

En este ejemplo se muestra como **eliminar un elemento de un vector sin dejar huecos** en el medio del vector:

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

Como se ve en la figura, se tiene un vector de 10 elementos y se desea eliminar el elemento 7 que se 15 que se encuentra en el vector, sin que queden huecos o basura en el lugar que ocupaba ese elemento:

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

eliminar

se debe eliminar el elemento de subíndice 3 del vector, sin que queden huecos:

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3		12	7	5	22	13	10	

Para ello se deben mover todos los elementos una posición hacia atrás, y restar 1 al tope:

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3		12	7	5	22	13	10	

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	12	7	5	22	13	10		

Se decrementa el valor del tope:

0	1	2	3	4	5	6	7	8		— tope
1	4	3	12	7	5	22	13	10		

A continuación se presenta el código fuente de la función que realiza dicha operación:

```

1
2 #define FALSO 0
3 #define VERDADERO 1
4 #define MAX 1000
5
6 typedef int          elemento_t;
7 typedef elemento_t   vector_t[MAX];
8
9 void eliminar_elemento( vector_t vector, elemento_t elemento, int* tope){
10
11     int esta = FALSO;
12     int i = 0;
13     int pos = 0;
14
15     while( ( i < *tope) && ( !esta ) ){
16         if ( vector[i]==elemento ){
17             esta=VERDADERO;
18             pos=i;
19         }
20         i++;
21     }
22     if ( esta ){
23         for( i=pos ; i<*tope-1 ; i++ )
24             vector[i]=vector[i+1];
25         *tope--;
26     }
27 }

```

Code 2: Insertar Ordenado

### 3. Buscar un Elemento dentro de un Vector

Una de las operaciones más comunes utilizadas cuando se trabaja con vectores es la **búsqueda de un determinado valor dentro del vector**. En otras palabras, si ese valor se encuentra almacenado en algún elemento del vector. Una primera aproximación a la solución de este problema es imaginarse al vector como una bolsa llena de caramelos y el valor a determinar es un caramelo de color rojo. Instintivamente se podría imaginar a la solución como aquella de ir sacando de a un caramelo de la bolsa hasta:

- Encontrar el caramelo rojo
- Haber sacado todos los caramelos de la bolsa, lo que implica que el caramelo buscado no está

Este tipo de proceso de búsqueda se denomina **búsqueda lineal** [?],[?]. La idea utilizando vectores es la siguiente, si se desea saber si el número entero 7 está en el vectos:

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

Se comienza a inspeccionar cada elemento desde el inicio del arreglo, buscando el el valor en cada uno de los elementos del arreglo, hasta el tope:

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

0	1	2	3	4	5	6	7	8	9	— tope
1	4	3	15	12	7	5	22	13	10	

encontrado

Una vez que el elemento es encontrado el algoritmo:

- debe finalizar
- puede devolver únicamente verdadero o falso
- puede devolver la posición donde se encuentra el elemento en cuestión

de no encontrarse debe informar tal situación.

```
1 #include <stdio.h>
2
3 #define FALSO 0
4 #define VERDADERO 1
5 #define MAX 1000
6
7 typedef int elemento_t;
8 typedef elemento_t vector_t[MAX];
9
10
11 int buscar_elemento(vector_t vector, int tope, elemento_t elemento_buscado){
12
13     int i=0;
14     int esta=FALSO;
15
16     while( (!esta) && (i<tope) ){
17         if (vector[i]==elemento_buscado)
18             esta=VERDADERO;
19         i++;
20     }
21
22     return esta;
23 }
24
25 int main() {
26
27     vector_t un_vector={1,2,3,5,6,7,8,9,};
28     int tope=8;
29     int un_valor;
30
31     un_valor=4;
32     if ( buscar_elemento(un_vector,tope,un_valor) )
33         printf("El %d no está\n",un_valor);
34
35     un_valor=3;
36     if ( buscar_elemento(un_vector,tope,un_valor) )
37         printf("El %d está\n",un_valor);
38
39     return 0;
40 }
```

Code 3: Búsqueda lineal

## 4. Búsqueda Binaria

Este algoritmo de búsqueda puede ser utilizado ÚNICAMENTE cuando el vector en el cual se realiza la operación posee sus elementos ordenados. Este metodo es tan bueno que puede reducir la cantidad de comparaciones en una búsqueda significativamente y tanto así que si tenemos un arreglo de  $n$  cantidad de elementos su orden es  $\log_2(n)$ , es decir, que en un vector con 1000000 elementos con solo realizar  $\log_2(1000000)=20$  comparaciones sabremos si el elemento se encuentra en el mismo. La implementación del mismo es la siguiente:

```

1 int busqueda_binaria(vector_t vector, int tope, elemento_t elemento_buscado) {
2     int centro;
3     int inicio=0;
4     int fin=tope-1;
5
6     while(inicio<=fin){
7         centro=(fin+inicio)/2;
8         if(vector[centro]==elemento_buscado)
9             return centro;
10        else if(elemento_buscado < vector[centro])
11            fin=centro-1;
12        else
13            inicio=centro+1;
14    }
15    return -1;
16 }
17

```

Code 4: Búsqueda Binaria

Veamos su funcionamiento en la figura ??

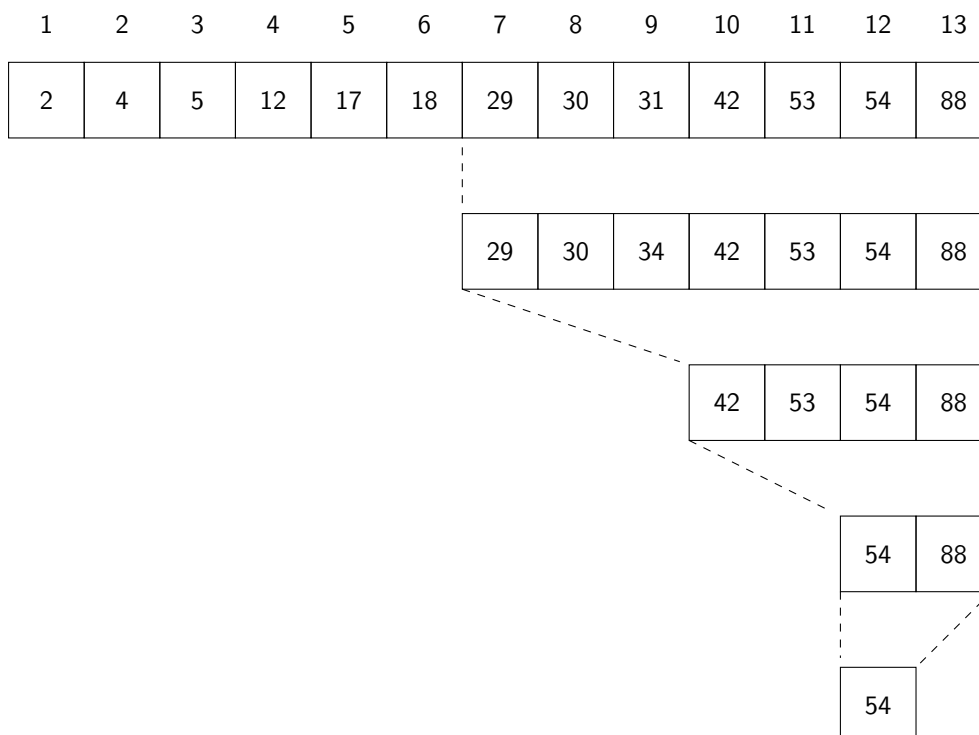


Figura 1: Ejemplo de funcionamiento de la Búsqueda Binaria

La versión recursiva de la busque da binaria se implementa de la siguiente forma:

```

1 int busqueda_binaria(vector_t vector, int inicio, int fin, elemento_t elemento_buscado) {
2     int centro;
3
4     if(inicio<=fin){
5         centro=(fin+inicio)/2;
6         if(vector[centro]==elemento_buscado)
7             return centro;
8         else if(elementoBuscado < vector[centro])
9             return busqueda_binaria(vector, inicio, centro-1, elemento_buscado);
10        else
11            return busqueda_binaria(vector, centro+1, fin, elemento_buscado);
12    } else
13        return -1;
14 }
15

```

Code 5: Búsqueda Binaria Recursiva

## 5. Ordenar Vectores

Los métodos de ordenamiento que a continuación se verán se denominan **métodos de ordenamiento interno**. Ya que **se lleva a cabo completamente en memoria principal**. Todos los objetos que se ordenan caben en la memoria principal de la computadora.

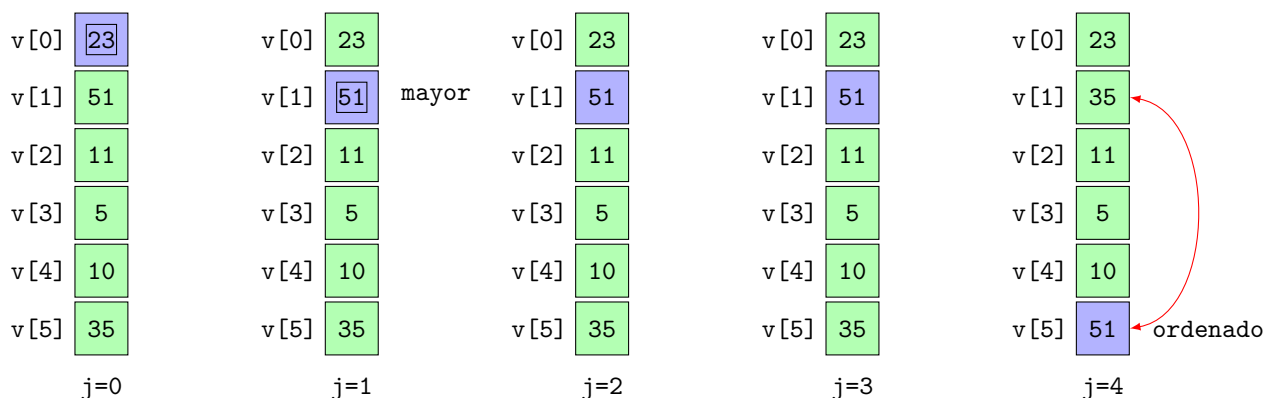
### 5.1. Método de Selección

Este método de ordenamiento consta de una serie sencilla de pasos:

1. Se busca el elemento de menor valor en el arreglo y se lo intercambia con el elemento que está en la primera posición del arreglo.
2. Posteriormente se busca el segundo elemento mas pequeño del arreglo y se lo intercambia con el elemento de la segunda posición del arreglo.
3. Se continua así hasta que todos los elementos del arreglo estén ordenados.

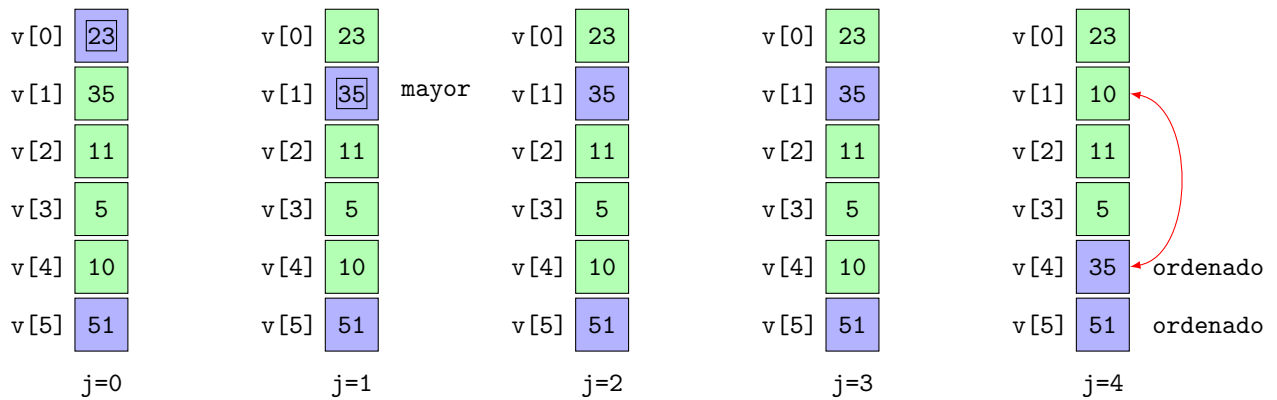
Dado el siguiente vector se ordenará el mismo utilizando el método de selección. Para ello lo único que hay que hacer es seguir el algoritmo. Para este método **se denomina pasada a la búsqueda del elemento máximo/mínimo**, es decir, recorrer todo o parte del arreglo buscando el máximo o mínimo:

Pasada 1: i =0

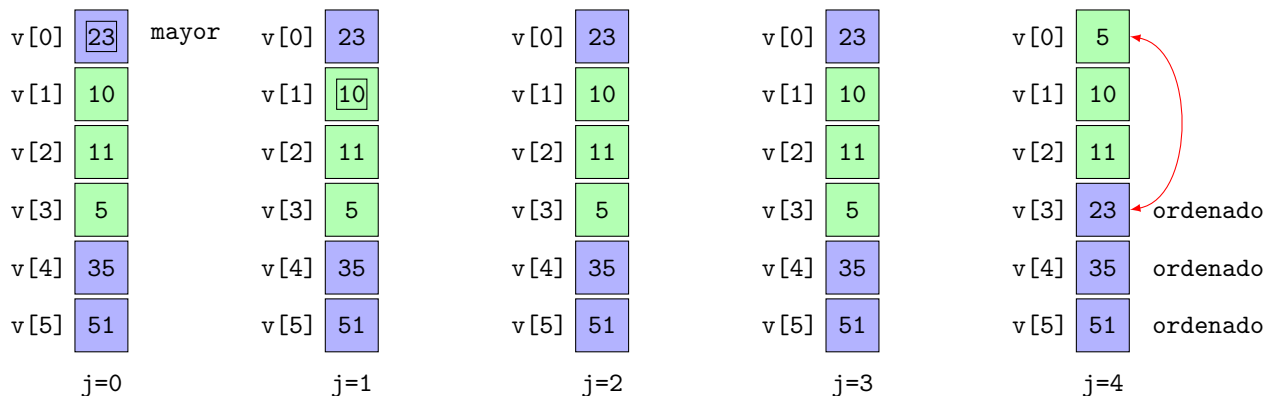




Pasada 2: i =1



Pasada 3: i =2



El proceso se continua así en las 6 pasadas hasta que el vector queda completamente ordenado. Como puede verse se denomina ordenamiento por **selección** porque en cada pasada se elige o selecciona el elemento que debe ocupar el último, penúltimo, antepenúltimo lugar, sucesivamente hasta la primer posición (o al revés).

Código Fuente:

```

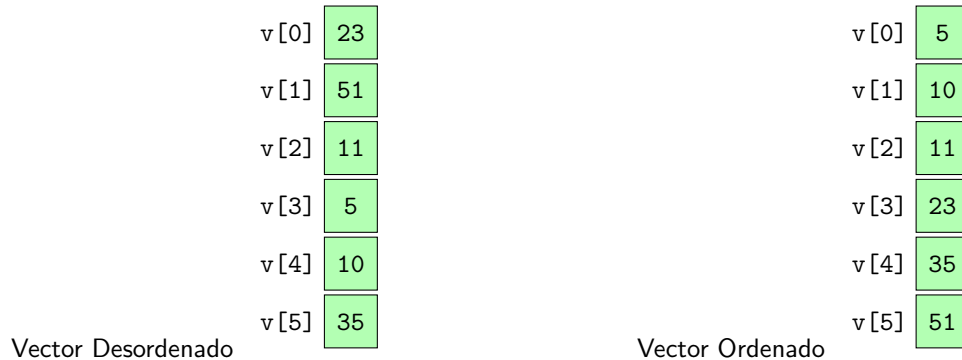
1 void selection_sort(vector_t vector, int tope){
2     int i, j;
3     int tmp;
4     int minimo;
5
6     for(i=0 ; i < (tope-1) ; i++){
7         minimo=i; /* minimo actual */
8         for (j=i+1 ; j < tope ; j++){ /* buscar el menor elemento del vector */
9             if (vector[minimo] > vector[j])
10                minimo=j; /* nuevo minimo */
11        }
12        /* swap vector[i] and vector[minimo] */
13        tmp = vector[i];
14        vector[i] = vector[minimo];
15        vector[minimo] = tmp;
16    }
17 }
18

```

Code 6: Ordenamiento por Selección

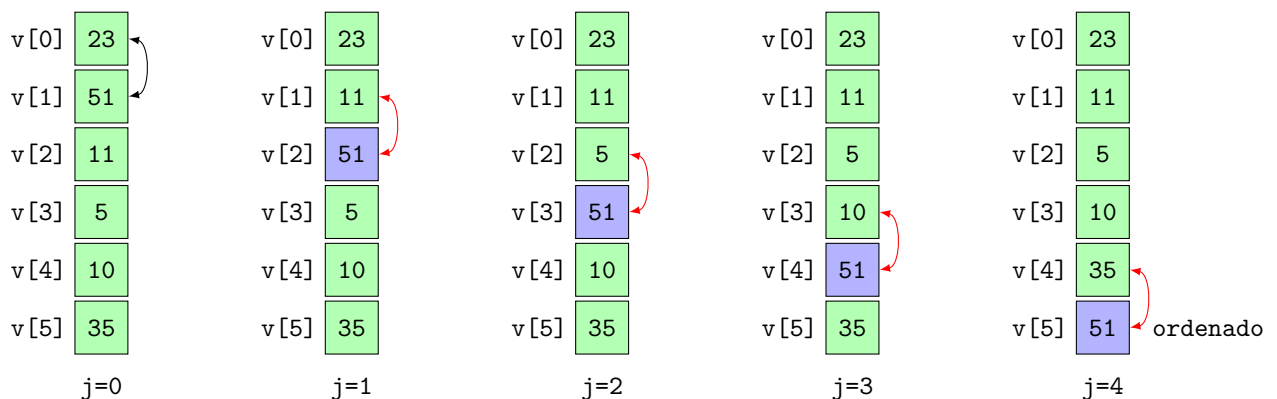
## 5.2. Método de Burbujeo

Este método es tal vez el que primero se enseña en cualquier curso inicial de algoritmos. La idea principal consiste en ir comparando a cada elemento de un vector  $v[]$ , con su adyacente y si estos están en el orden equivocado se intercambian; en primer lugar se compara  $v[0]$  con  $v[1]$ , si están desordenados, estos se intercambian; y se sigue así sucesivamente hasta el último elemento del vector. Esta serie de comparaciones se denomina *pasada*. Se llama burbujeo porque al ejecutarse parece que los elementos del vector burbujan hacia los extremos según corresponda.



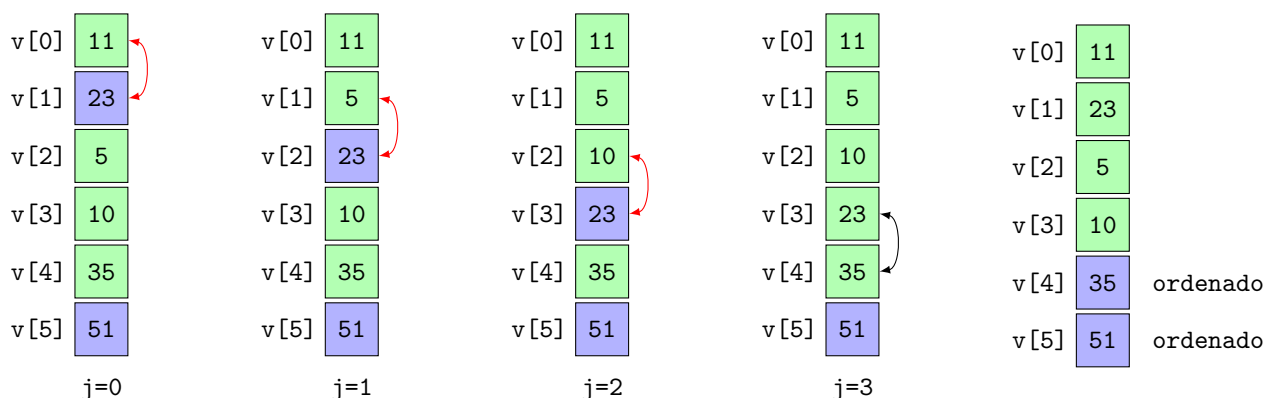
Se compara cada elemento  $j$ , con su correspondiente elemento  $j+1$ , y dependiendo si el orden es ascendente o descendente se realiza el intercambio (flecha roja) entre  $v[j]$  y  $v[j+1]$ . En este ejemplo será ascendentemente:

Pasada 1:  $i = 0$



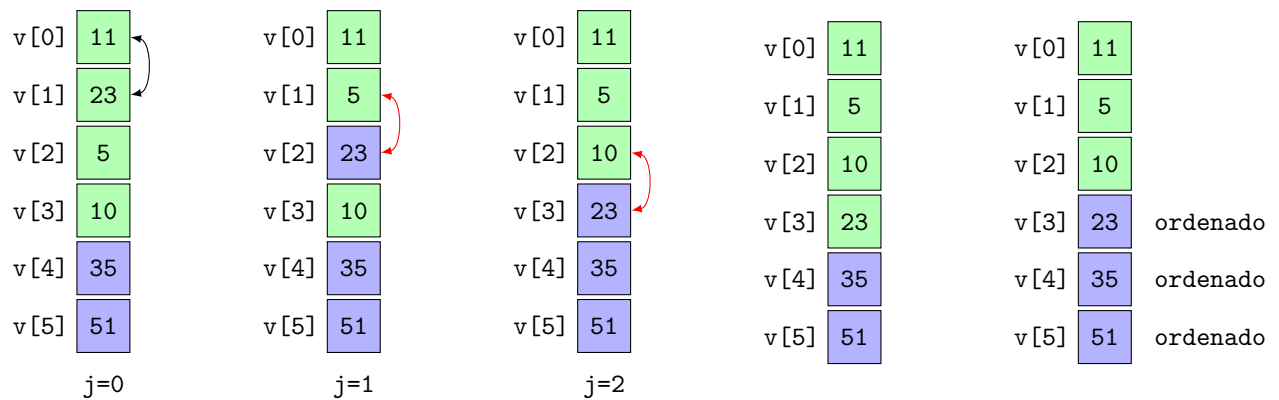
En la primer pasada, con  $i=0$ , el mayor elemento del arreglo ya quedó en la última posición, ya que es el elemento más grande.

Pasada 2:  $i = 1$



En la segunda pasada, con  $i=1$ , el segundo mayor elemento del arreglo ya quedó en la ante última posición, ya que es el segundo elemento más grande.

Pasada 3:  $i = 2$



En la tercer pasada, con  $i=2$ , el tercer mayor elemento del arreglo ya quedó en su posición de orden, ya que es el tercer elemento más grande. Como puede desprenderse del análisis del gráfico, en cada pasada, queda ordenado un elemento del arreglo, por lo tanto al saber eso no es necesario seguir haciendo comparaciones hasta el final del vector:

Cuadro 1: Evolución de  $i$  y  $j$

$i$ pasada	$j$ elemento	tope- $i$
0	0 1 2 3 4 5	$5-i=5$
1	0 0 0 0 0	$5-i=4$
2	0 1 2 3	$5-i=3$
3	0 1 2	$5-i=2$
4	0 1	$5-i=1$
5	0	$5-i=0$

Implementación:

```

1 void burbujeo(vector_t vector, int tope){
2     int i,j,auxiliar;
3     for(i=0 ; i < tope ; i++){
4         for(j=0 ; j < (tope-i) ; j++){
5             if(vector[j] > vector[j+1]){
6                 auxiliar=vector[j];
7                 vector[j]=vector[j+1];
8                 vector[j+1]=auxiliar;
9             }
10        }
11    }
12 }
```

Code 7: Ordenamiento por Burbujeo

La implementación anterior no está optimizada, pues si el vector se encuentra ordenado el método continua iterando. Veamos entonces el método mejorado u optimizado:

```

1 void burbujeo_mejorado(vector_t vector, int tope){
2     int i, j=0, auxiliar;
3     bool esta_ordenado=false;
4     while( (j < n) && (!esta_ordenado) ){
5         esta_ordenado=true;
6         for(i=0; i<n; i++){
7             if(a[i]>a[i+1]){
8                 aux=a[i];
9                 a[i]=a[i+1];
10                a[i+1]=aux;
11                esta_ordenado=false;
12            }
13        }
14        j++;
15    }
16 }

```

Code 8: Ordenamiento por Burbujeo Mejorado

### 5.3. Método de Inserción

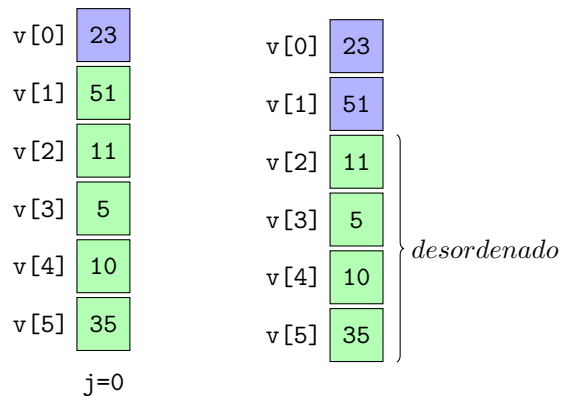
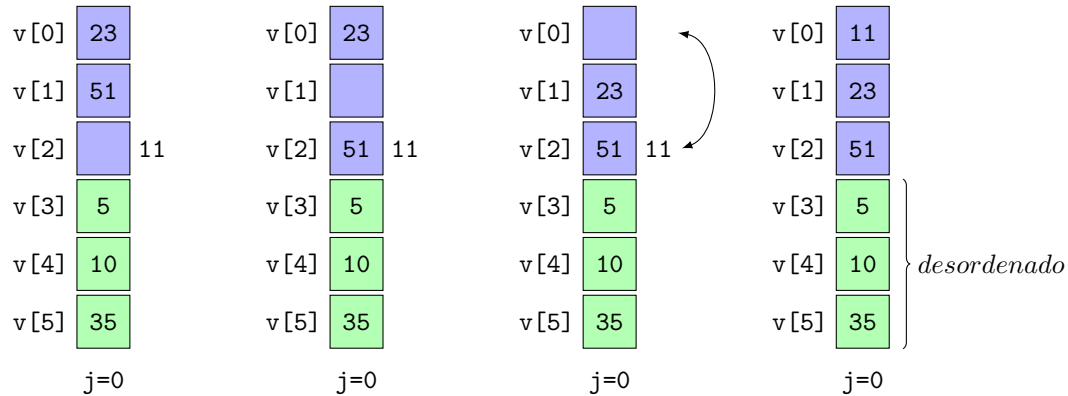
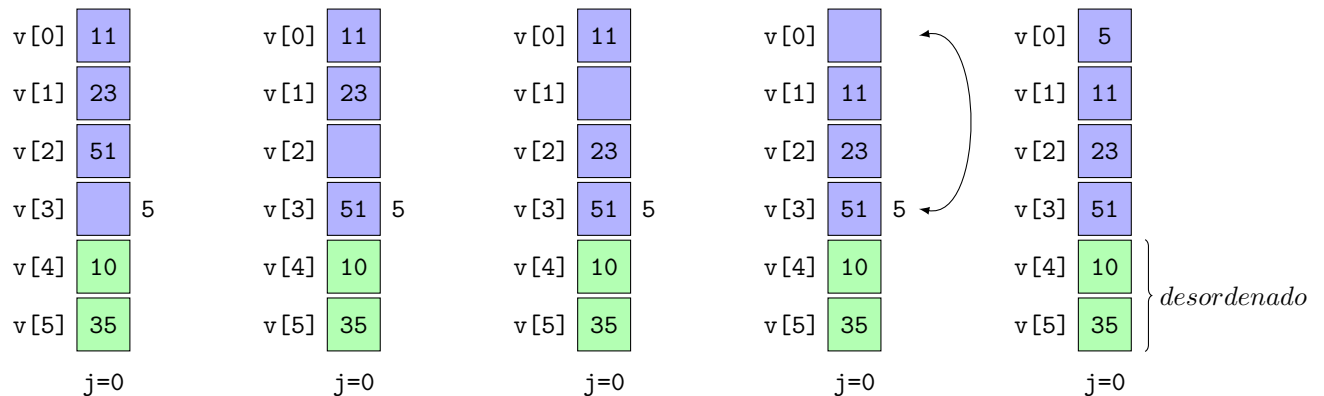
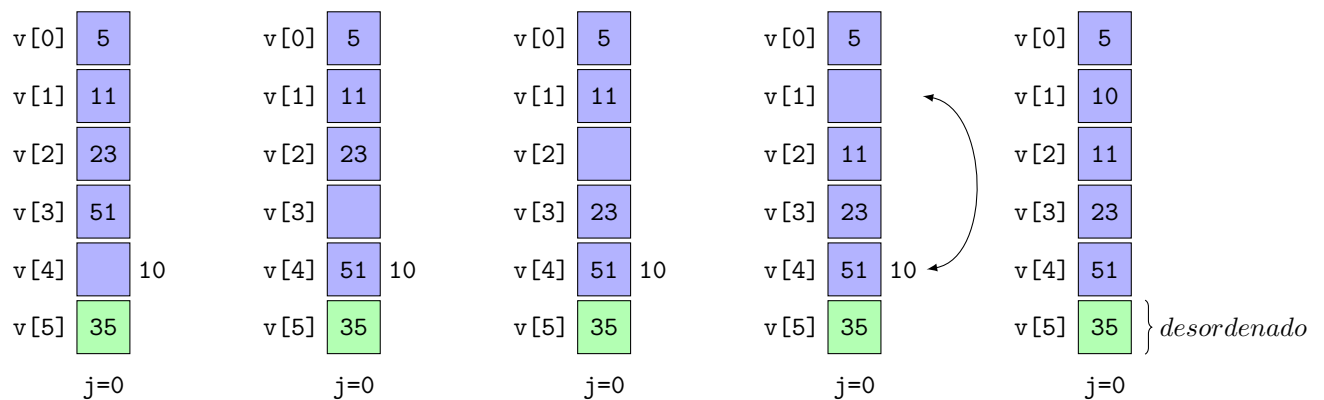
El ordenamiento por inserción técnicamente es la forma más lógica de ordenar cualquier cosa por ejemplo un conjunto de cartas. Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay  $k$  elementos ordenados de menor a mayor, se toma el elemento  $k+1$  y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha). En este punto se inserta el elemento  $k+1$  debiendo desplazar los demás elementos. A continuación se muestra un ejemplo:

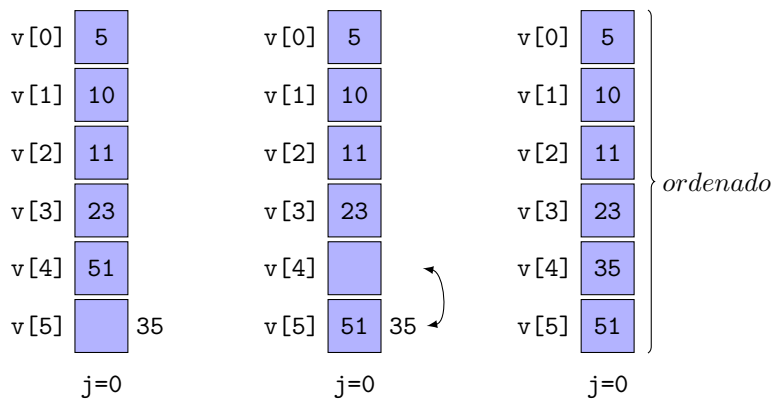
v[0]	23
v[1]	51
v[2]	11
v[3]	5
v[4]	10
v[5]	35

Pasada 1:  $i = 0$

v[0]	23	v[0]	23
v[1]	51	v[1]	51
v[2]	11	v[2]	11
v[3]	5	v[3]	5
v[4]	10	v[4]	10
v[5]	35	v[5]	35
j=0		} desordenado	

Pasada 1:  $i = 1$

Pasada 1:  $i = 2$ Pasada 1:  $i = 3$ Pasada 1:  $i = 4$ Pasada 1:  $i = 5$



A continuación se muestra una posible implementación del ordenamiento por inserción:

```

1 void insercion(vector_t vector, int tope){
2     int i,j,auxiliar;
3     for(i=1 ; i < tope ; i++){
4         j=i;
5         auxiliar=vector[i];
6         while( (j > 0) && (aux < vector[j-1]) ){
7             vector[j]=vector[j-1];
8             j--;
9         }
10        vector[j]=auxiliar;
11    }
12 }
13

```

Code 9: Ordenamiento por Inserción

## 5.4. Método de QuickSort

El ordenamiento QuickSort, basa su método de trabajo en dividir el vector en 2 de menor tamaño, en base al valor de un dato llamado **pivote**. De un lado de este quedaran todos los datos menores y del otro los mayores. Una vez realizado este proceso se llama a si mismo recursivamente con la parte del vector donde se encuentran los valores menores y de mayores, repitiendo el proceso hasta que solo haya un elemento.

```

1 void quicksort(vector_t vector ,int inicio,int tope){
2     int i, j, pivot, temp;
3     int ultimo = tope - 1;
4
5     if(inicio < ultimo){
6         pivot=inicio;//Elige el pivote
7         i=inicio;
8         j=ultimo;
9
10        while(i<j){
11            while( (vector[i] <= vector[pivot] ) && (i<ultimo) )
12                i++;
13            while(vector[j]>vector[pivot])
14                j--;
15            if(i<j){
16                temp=vector[i];
17                vector[i]=vector[j];
18                vector[j]=temp;
19            }
20        }
21        temp=vector[pivot];
22        vector[pivot]=vector[j];
23        vector[j]=temp;
24        quicksort(vector ,inicio ,j-1);
25        quicksort(vector ,j+1,ultimo);
26    }
27 }

```

Code 10: QuickSort

## 6. Operaciones avanzadas entre vectores

Un factor a tener en cuenta es que las operaciones que a continuación se detallan deben ser realizadas recorriendo cada vector UNA SOLA vez. Es posible y muy trivial realizar estas operaciones recorriendo cada vector más de una vez, pero ineficiente desde el punto de vista de la cantidad de operaciones de comparación que se realizan.

### 6.1. Mezcla

La idea de la mezcla entre dos vectores es generar un tercer vector que contenga los elementos de los anteriores. El núcleo del problema está en realizar un algoritmo que dados dos vectores ORDENADOS, cuyos topes son tope\_uno y tope\_dos, genere un tercer vector ORDENADO con los elementos de los vectores anteriores cuyo tope será igual a tope\_uno + tope\_dos.

v[0]	v[1]	v[2]	v[3]
2	4	78	97

w[0]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]	w[7]
-15	0	13	15	78	90	94	96

```

1 void mezcla(vector_t vector_uno, int tope_uno, vector_t vector_dos, int tope_dos, vector_t
  vector_resultado, int* tope_resultado ){
2
3     int i,j;
4     i=0;
5     j=0;
6     (*tope_resultado)=0;
7
8     while( (i < tope_uno) && (j < tope_dos) ){
9         if (vector_uno[i] <= vector_dos[j]) {
10             vector_resultado[*tope_resultado]=vector_uno[i];
11             i++;
12         }
13         else{
14             vector_resultado[*tope_resultado]=vector_dos[j];
15             j++;
16         }
17         (*tope_resultado)++;
18     }
19
20     /*Si alguno de los vectores llego a su tope se completa el vector resultado con los
    elementos del otro vector */
21
22     while(i < tope_uno){
23         vector_resultado[*tope_resultado]=vector_uno[i];
24         (*tope_resultado)++;
25         i++;
26     }
27     while(j < tope_dos){
28         vector_resultado[*tope_resultado]=vector_dos[j];
29         (*tope_resultado)++;
30         j++;
31     }
32 }

```

Code 11: Mezcla de Vectores

### 6.2. Unión

La Unión entre dos vectores consiste en confeccionar un algoritmo que tenga como parámetros de entrada a dos vectores y sus topes, y como parámetro de salida el vector Unión y su tope. A diferencia de la Mezcla vista

anteriormente, la unión no contiene elementos repetidos, es decir que si un elemento se encuentra en ambos vectores de partida, solo aparecerá una vez en el vector de salida.

Code 12: Unión de Vectores

### 6.3. Diferencia

La diferencia entre dos vectores consiste en confeccionar un algoritmo que tenga como parámetros de entrada dos vectores y sus topes, y como parámetro de salida el vector diferencia y su tope. En otras palabras el vector diferencia contiene los elementos del primer vector que no están en el segundo. Como precondition pedimos que ambos vectores estén ordenados.

```

1 void diferencia_vectores( vector_t vector_uno, int tope_uno, vector_t vector_dos,
2     int tope_dos, vector_t vector_diferencia, int* tope_diferencia){
3
4     int i = 0, j = 0;
5     (*tope_diferencia)=0;
6
7     while( i < tope_uno && j < tope_dos){
8         if(vector_uno[i] == vector_2[j])
9             i++;
10            j++;
11        } else if (vector_uno[i] < vector_2[j]){
12            vector_diferencia[*tope_diferencia] = vector_uno[i];
13            (*tope_diferencia)++;
14            i++;
15        } else {
16            j++;
17        }
18    }
19    while (i < tope_uno){
20        vector_diferencia[*tope_diferencia] = vector_uno[i];
21        (*tope_diferencia)++;
22        i++;
23    }
24 }
```

Code 13: Diferencia de Vectores.

### 6.4. Intersección

La Intersección entre dos vectores consiste en confeccionar un algoritmo que tenga como parámetros de entrada dos vectores y sus topes, y como parámetro de salida el vector intersección y su tope.

```

1 void interseccion(vector_t vector_uno[], int tope_uno, vector_t vector_dos[], vector_t
2     vector_interseccion[], int* tope_interseccion) {
3
4     int i=0;
5     int j=0;
6     (*tope_interseccion)=0;
7
8     while( ( i < tope_uno ) && ( j < tope_dos ) ){
9         if(vector_uno[i] == vector_dos[j]) {
10            vector_interseccion[*tope_interseccion]=vector_uno[i];
11            (*tope_interseccion)++;
12            i++;
13            j++;
14        } else if(vector_uno[i] < vector_dos[j]){
15            i++;
16        } else {
17            j++;
18        }
19    }
20 }
```

Code 14: Intersección de Vectores