75.40 Algortimos y Programación I Curso 4
Operaciones con Arreglos y Listas

Dr. Mariano Méndez[1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

19 de agosto de 2019

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.

```
#define FALSO 0
#define VERDADERO 1
#define MAX 1000

typedef int             elemento_t;
typedef elemento_t              vector_t[MAX];

void insertarOrdenado( vector_t vector, elemento_t elemento, int* tope){
         int    inserto;
         int    i;
         elemento_t  elem_aux;
         inserto=FALSO;
         for(i=0;i< (*tope); i++ ){
                if (elemento < vector[i]){
                        elem_aux=vector[i];
                        vector[i]=elemento;
                        elemento = elem_aux;
                        inserto=VERDADERO;
                }
         }
         if(inserto){
                vector[*tope] = elem_aux;
                *(tope)++;
         }
         if ((!inserto) || (tope==0) ){
                vector[*tope] = elemento;
                *(tope)++;
         }
}
```

Code 1: Insertar Ordenado

2.1. 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:

```

#define FALSO 0
#define VERDADERO 1
#define MAX 1000

typedef int            elemento_t;
typedef elemento_t    vector_t[MAX];

void eliminar_elemento( vector_t vector, elemento_t elemento, int* tope){

        int esta = FALSO;
        int  i = 0;
        int  pos = 0;

        while( ( i < *tope) && ( !esta ) ){
                if ( vector[i]==elem ){
                        esta=VERDADERO;
                        pos=i;
                }
                i++;
        }
        if ( esta ){
                for( i=pos ; i<tope-1 ; i++ )
                        vector[i]=vector[i+1];
                *tope--;
        }
}
```

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 [**?**],[**?**]:

```c
#include<stdio.h>

#define FALSO 0
#define VERDADERO 1
#define MAX 1000

typedef int elemento_t;
typedef elemento_t vector_t[MAX];


int buscar_elemento(vector_t vector, int tope, elemento_t elemento_buscado){

        int i=0;
        int esta=FALSO;

        while( (!esta)  && (i<tope) ){
                if (vector[i]==elemento_buscado)
                        esta=VERDADERO;
                i++;
        }

        return esta;
}

int main() {

        vector_t un_vector={1,2,3,5,6,7,8,9,};
        int      tope=8;
        int      un_valor;

        un_valor=4;
        if ( buscar_elemento(un_vector,tope,un_valor) )
                printf("El %d no está\n",un_valor);

        un_valor=3;
        if ( buscar_elemento(un_vector,tope,un_valor) )
                printf("El %d está\n",un_valor);

        return 0;
}
```

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 $log2(n)$, es decir, que en un vector con 1000000 elementos con solo realizar log2(1000000)=20 comparaciones sabremos si el elemento se encuentra en el mismo. La implementación del mismo es la siguiente:

```
int busqueda_binaria(vector_t vector, int tope, elemento_t elemento_buscado) {
        int centro;
        int inicio=0;
        int fin=tope-1;

        while(inicio<=fin){
                centro=(fin+inicio)/2;
                if(vector[centro]==elemento_buscado)
                        return centro;
                else if(elemento_buscado < vector[centro])
                        fin=centro-1;
                else
                        inicio=centro+1;
        }
        return -1;
}

```

Code 4: Búsqueda Binaria

Veamos su funcionamiento en la figura 1

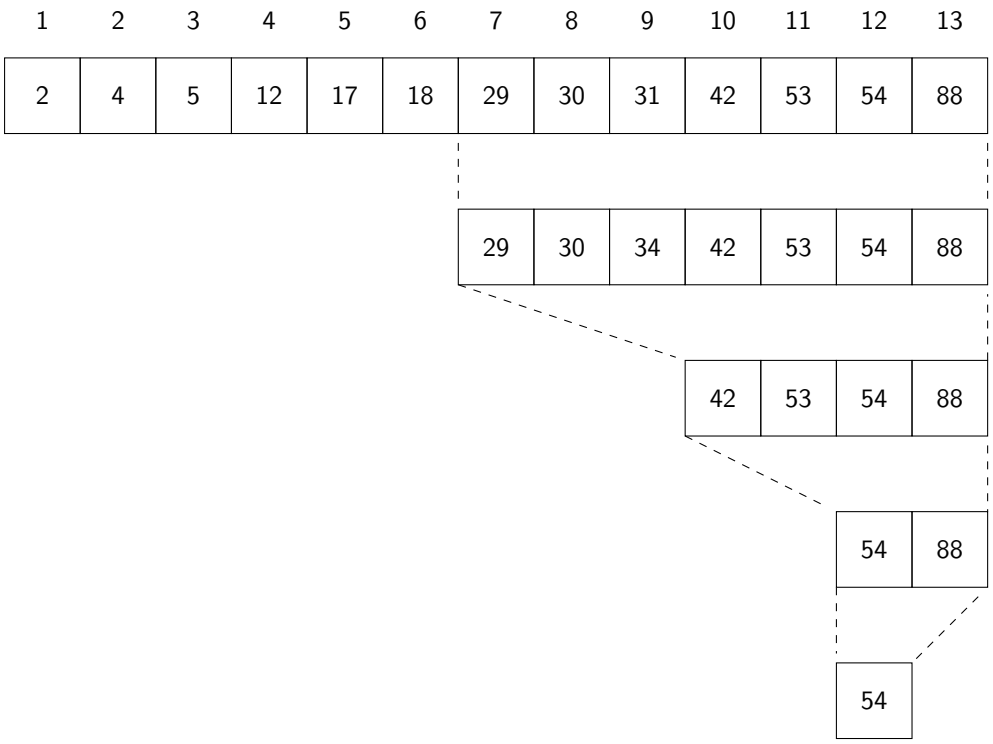

Figura 1: Ejemplo de funcionamiento de la Búsqueda Binaria

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

```
int busqueda_binaria(vector_t vector, int inicio, int fin, elemento_t elemento_buscado) {
        int centro;

        if(inicio<=fin){
                centro=(fin+inicio)/2;
                if(vector[centro]==elemento_buscado)
                        return centro;
                else if(elementoBuscado < vector[centro])
                        return busqueda_binaria(vector,inicio,centro-1,elemento_buscado);
                else
                        return busqueda_binaria(vector,centro+1,fin,elemento_buscado);
        } else
                return -1;
}

```

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.

Código Fuente:

```
void  selection_sort(vector_t vector, int tope){
        int i, j;
        int tmp;
        int minimo;

        for(i=0 ; i < (tope-1) ; i++){
                minimo=i; /* minimo actual */
                for (j=i+1 ; j < tope ; j++){   /* buscar el menor elemento del vector */
                        if (vector[minimo] > vector[j])
                                minimo=j; /* nuevo minimo */
                        }
                }
                /* swap vector[i] and vector[minimo] */
                tmp = vector[i];
                vector[i] = vector[minimo];
                vector[minimo] = tmp;
        }
}
```

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 con su adyacente y si estos están en el orden equivocado se intercambian. Se llama burbujeo porque al ejecutarse parece que los elementos del vector burbujean hacia los extremos según corresponda. Implementación:

```
void burbujeo(vector_t vector, int tope){
        int i,j,auxiliar;
        for(i=0 ; i < tope ; i++){
                for(j=0 ; j < (tope-i) ; j++){
                        if(vector[j] > vector[j+1]){
                                auxiliar=vector[j];
                                vector[j]=vector[j+1];
                                vector[j+1]=auxiliar;
                        }
                }
        }
}
```

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:

```
void burbujeo_mejorado(vector_t vector, int tope){
        int i, j=0 ,auxiliar;
        bool esta_ordenado=false;
        while( (j < n) && (!esta_ordenado) ){
                esta_ordenado=true;
                for(i=0;i<n;i++){
                        if(a[i]>a[i+1]){
                                aux=a[i];
                                a[i]=a[i+1];
                                a[i+1]=aux;
                                esta_ordenado=false;
                        }
                }
                j++;
        }
}
```

Code 8: Ordenamiento por Burbujeo Mejorado

5.3. Método de Inserción

El ordenamiento por inserción técnicamente es la forma mas 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 desplazarse los demás elementos.

```
void insercion(vector_t vector, int tope){
        int  i,j,auxiliar;
        for(i=1 ; i < tope ; i++){
                j=i;
                auxiliar=vector[i];
                while( (j > 0) && (aux < vector[j-1]) ){
                        vector[j]=vector[j-1];
                        j--;
                }
                vector[j]=auxiliar;
        }
}

```

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 todo 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.

```
void quicksort(vector_t vector ,int inicio,int tope){
        int i, j, pivot, temp;
        int ultimo = tope - 1;

        if(inicio < ultimo){
                pivot=inicio;//Elige el pivote
                i=inicio;
                j=ultimo;

                while(i<j){
                        while( (vector[i] <= vector[pivot] ) && (i<ultimo) )
                                i++;
                        while(vector[j]>vector[pivot])
                                j--;
                        if(i<j){
                                temp=vector[i];
                                vector[i]=vector[j];
                                vector[j]=temp;
                        }
                }
                temp=vector[pivot];
                vector[pivot]=vector[j];
                vector[j]=temp;
                quicksort(vector,inicio,j-1);
                quicksort(vector,j+1,ultimo);
        }
}
```

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.

```
void mezcla(vector_t vector_uno, int tope_uno, vector_t vector_dos, int tope_dos, vector_t
    vector_resultado, int* tope_resultado ){

        int i,j;
        i=0;
        j=0;
        (*tope_resultado)=0;

        while( (i < tope_uno) && (j < tope_dos)   ){
                if (vector_uno[i] <= vector_dos[j]) {
                   vector_resultado[*tope_resultado]=vector_uno[i];
                   i++;
                }
                else{
                   vector_resultado[*tope_resultado]=vector_dos[j];
                   j++;
                }
                (*tope_resultado)++;
        }

        /*Si alguno de los vectores llego a su tope se completa el vector resultado con los
    elementos del otro vector */

        while(i < tope_uno){
           vector_resultado[*tope_resultado]=vector_uno[i];
           (*tope_resultado)++;
           i++;
        }
        while(j < tope_dos){
           vector_resultado[*tope_resultado]=vector_dos[j];
           (*tope_resultado)++;
           j++;
        }
}
```

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 Mezla 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.

```c
void unir_vectores(vector_t vector_uno, int tope_uno, vector_t vector_dos, int tope_dos,
        vector_t vector_unido, int* tope_union){

        int i=0, j=0;
        (*tope_union) = 0;

        while ( (i < tope_uno) && (j < tope_dos)){
                if( vector_uno[i] == vector_dos[j] ){
                        vector_unido[*tope_union] = vector_uno[i];
                        (*tope_union)++;
                        i++;
                        j++;
                } else if( vector_uno[i] < vector_dos[j] ){
                        vector_unido[*tope_union] = vector_uno[i];
                        (*tope_union)++;
                        i++;
                } else {
                        vector_unido[*tope_union] = vector_dos[j];
                        (*tope_union)++;
                        j++;
                }
        }

        while ( i < tope_uno ){
                vector_unido[*tope_union] = vector_uno[i];
                (*tope_union)++;
                i++;
        }

        while ( j < tope_dos ){
                vector_unido[*tope_union] = vector_dos[j];
                (*top_union)++;
                j++;
        }
}
```

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 elemento del primer vector que no están en el segundo. Como precondición pedimos que ambos vectores estén ordenados.

```c
void diferencia_vectores( vector_t vector_uno, int tope_uno, vector_t vector_dos,
                int tope_dos, vector_t vector_diferencia, int* tope_diferencia){

        int i = 0, j = 0;
        (*tope_diferencia)=0;

        while( i < tope_uno && j < tope_dos){
                if(vector_uno[i] == vector_2[j])
                        i++;
                        j++;
                } else if (vector_uno[i] < vector_2[j]){
                        vector_diferencia[*tope_diferencia] = vector_uno[i];
                        (*tope_diferencia)++;
                        i++;
                } else {
                        j++;
                }
        }
        while (i < tope_uno){
                vector_diferencia[*tope_diferencia] = vector_uno[i];
                (*tope_diferencia)++;
                i++;
        }
}
```

Code 13: Diferencia de Vectores.

6.4. Intersección

La Intersección 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 intersección y su tope.

```
void interseccion(vector_t vector_uno[], int tope_uno, vector_t vector_dos[], vector_t
        vector_interseccion[], int* tope_interseccion) {

        int i=0;
        int j=0;
        (*tope_interseccion)=0;

        while(  ( i < tope_uno )  &&  ( j < tope_dos ) ){
                if(vector_uno[i] == vector_dos[j]) {
                        vector_interseccion[*tope_interseccion]=vector_uno[i];
                        (*tope_interseccion)++;
                        i++;
                        j++;
                } else if(vector_uno[i] < vector_dos[j]){
                        i++;
                } else {
                        j++;
                }
        }
}
```

Code 14: Intersección de Vectores