Programación

Unidad 9: Apareo y Corte de Control

“La función de un buen software es hacer que lo complejo aparente ser simple.”

~ Grady Booch

Apareo de Vectores

El apareo de vectores (o mergeo) nos permite combinar dos vectores en un tercer vector que contendrá elementos de los vectores originales. En su versión más sencilla el vector resultante contendrá todos los elementos de ambos vectores, pero es muy sencillo adaptar el algoritmo original para agregar nuevas reglas o controles al formar el vector resultante.

Vamos de a poco, ¿Qué queremos lograr?

El objetivo de mínima es dados dos vectores obtener todos los elementos de ambos en un único vector. Ejemplo:

A = {-5, -3, 0, 1, 4, 6, 7, 9, 14} B = {-7, -5, -4, -1, 2, 5, 8, 10, 16}

Si alguien nos pidiera un vector C "con todos los elementos de A y B" sin mayores aclaraciones, podríamos decir que C = {-5, -3, 0, 1, 4, 6, 7, 9, 14, -7, -5, -4, -1, 2, 5, 8, 10, 16} cumple lo pedido. El vector es el resultado de "anexar" todos los elementos de B luego de los de A en un vector C.

Apareo de vectores ordenados

¿Pero qué sucedería si nos aclaran que A y B se encuentran ordenados y el objetivo es obtener un vector C ordenado de igual forma?

Es decir dados los mismos A y B el resultado debería ser:

C = {-7, -5, -5, -4, -3, -1, 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 14, 16}

Nuevamente hay una forma sencilla de resolverlo, primero "anexar" y luego ordenar el vector resultante (utilizando burbujeo por ejemplo). El problema es que si realizamos estos pasos estaremos recorriendo ambos vectores una vez, y posteriormente pagaremos el costo de ordenar el vector resultante, un costo que dependiendo del tamaño del vector podría ser imposible de justificar.

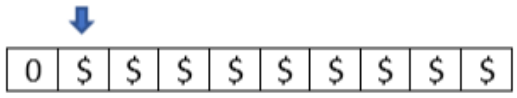
El dato de que ambos vectores están ordenados es el que nos ayudará a resolver este problema con una sola recorrida de ambos vectores.

Dado que el objetivo es obtener el vector resultante ordenado en forma ascendente (o descendente) igual que como están los vectores originales, podemos decir que lo que deseamos es llenar el vector tomando los valores más chicos. Y al estar ambos vectores ordenados,

sabemos que los valores más chicos se encuentran al inicio. Por lo cual lo que haremos será comparar valor por valor de ambos vectores iniciando desde la posición 0, y colocaremos en el vector resultante el valor que sea menor (o mayor si el orden fuera descendente).

Al colocar el valor menor en el vector resultante, en el vector original ya no nos interesa más, con lo cual pasaremos al siguiente elemento, del vector que hayamos tomado el valor. Repitiendo estas simples acciones completamos el vector resultante. Veámoslo con un gráfico:

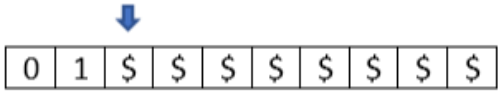
Nos paramos al inicio de ambos vectores y comparamos sus valores, como resultado obtenemos que 0 es el menor valor y es el primer elemento de nuestro vector resultante:



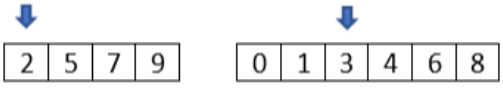
Colocamos el 0, y nos movemos para que el próximo valor se guarde en la posición siguiente.

Como ya tomamos el primer elemento del vector B, nos movemos para evaluar el siguiente, con lo cual ahora comparamos el primer elemento del vector A y el segundo de B

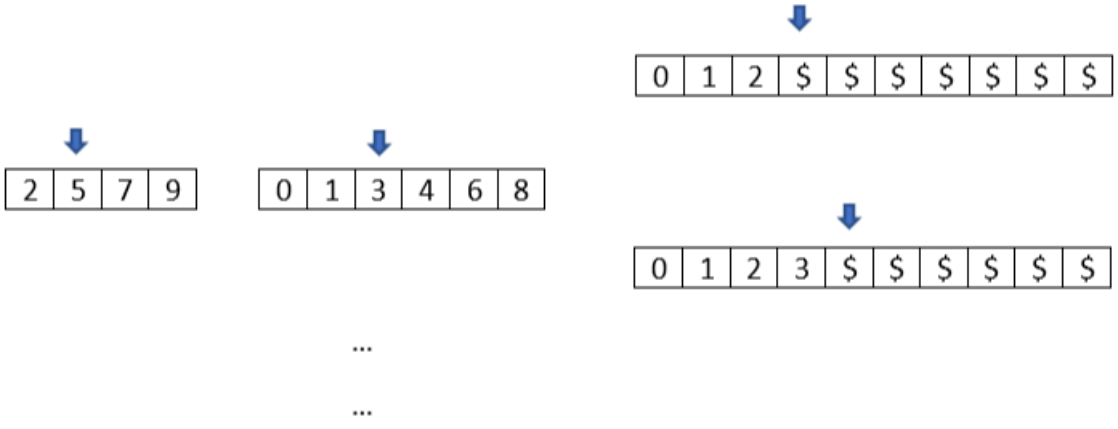
Nuevamente el elemento lo tomamos de B ya que 1 es menor que 2



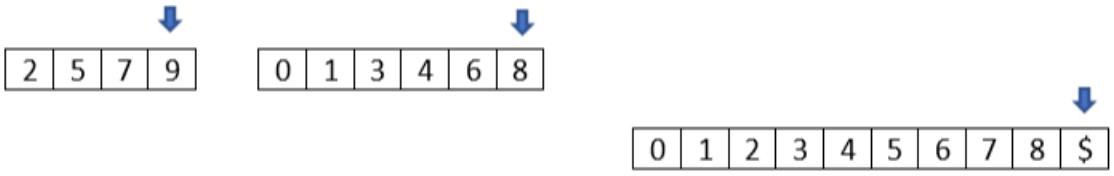
Nuevamente comparamos los elementos en donde estamos parados en cada uno de los vectores:



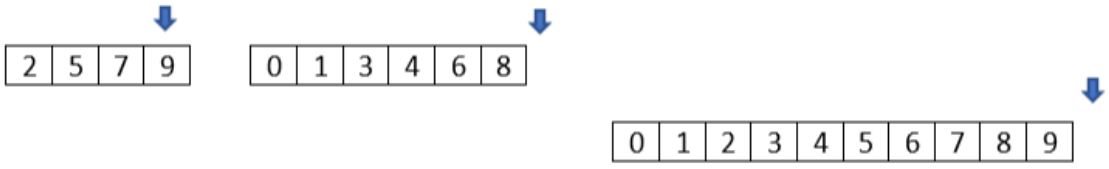
En este caso tomaremos el valor de A ya que 2 es menor que 3 y por lo tanto moveremos su posición para la próxima evaluación.



La misma operatoria se repetirá hasta que ya no pueda comparar elementos, es decir cuando alcancemos el final de algunos de los dos vectores. En nuestro ejemplo sucederá al llegar al final del vector B.



Como ya no hay elementos de B, lo único que queda es pasar todos los elementos de A.



Si en lugar de acabarse B se hubiera acabado A se haría lo mismo, se pasarían todos los elementos restantes de B.

Como desafío podés intentar hacer la implementación en código del algoritmo que acabamos de hacer

El código para hacer el apareo de dos vectores ordenado es el siguiente:

```
1   void apareo(int vecA[], int n, int vecB[], int m, int vecC[], int &k) {
2      // Contadores para la posicion de los vectores A y B.
3      int i = 0, j = 0;
4      // Contador para posicionarse en el vector resultante.
5      k = 0;
6
7       // Mientras pueda comparar valores (al menos uno de los vectores tiene
8   valor)
9      while (i < n && j < m) {
10        // Comparo los valores de los vectores
11        if (vecA[i] < vecB[j]) {
12           // Coloco el elemento de A porque es menor
13           vecC[k] = vecA[i];
14           // Me muevo en el vector A
15           i++;
16        } else {
17           vecC[k] = vecB[j];
18           j++;
19        }
20        // Incremento el contador de la posicion del vector resultante
21        k++;
22     }
23
24     // Paso todos los elementos restantes de A
25     while(i < n) {
26        vecC[k] = vecA[i];
27        i++;
28        k++;
29     }
30
31     // Paso todos los elementos restantes de B
32     while(j < m) {
33        vecC[k] = vecB[j];
34        j++;
35        k++;
36     }
   }
```

Como dijimos anteriormente, esta es la versión más sencilla pero sobre el mismo algoritmo podrían agregarse distintas validaciones como no poner valores repetidos, sólo porner valores repetidos, sólo poner valores distintos, etc.

Tener en cuenta, este algoritmo de apareo sirve solamente para aparear vectores ordenados en forma ascendente, es decir ambos vectores deben estar ordenados en forma ascendente y el resultante también quedará ordenado de esa forma.

Apareo de vectores de registros

Al igual que en los algoritmos vistos anteriormente, lo único que deberemos hacer para poder usar este mismo algoritmo con estructuras propias, será modificar los tipos de datos y la condición de comparación. De más está decir que los vectores deberán estar ordenados por el mismo criterio que usemos para la comparación en nuestro algoritmo.

```
1   struct Alumno {
2      int legajo;
3      int nota1;
4      int nota2;
5   };
6
7
8   void apareo(Alumno  vecA[], int n, Alumno vecB[], int m, Alumno vecC[], int &k)
9   {
10     // Contadores para la posicion de los vectores A y B.
11     int i = 0, j = 0;
12     // Contador para posicionarse en el vector resultante.
13     k = 0;
14
15      // Mientras pueda comparar valores (al menos uno de los vectores tiene
16  valor)
17     while (i < n && j < m) {
18        // Comparo los valores de los vectores
19        if (vecA[i].legajo < vecB[j].legajo) {
20           // Coloco el elemento de A porque es menor
21           vecC[k] = vecA[i];
22           // Me muevo en el vector A
23           i++;
24        } else {
25           vecC[k] = vecB[j];
26           j++;
27        }
28        // Incremento el contador de la posicion del vector resultante
29        k++;
30     }
31
32     // Paso todos los elementos restantes de A
33     while(i < n) {
34        vecC[k] = vecA[i];
35        i++;
36        k++;
37     }
38
39     // Paso todos los elementos restantes de B
40     while(j < m) {
41        vecC[k] = vecB[j];
42        j++;
43        k++;
44     }
   }
```

Corte de Control

El corte de control es una técnica que nos permitirá obtener información a partir de un conjunto de datos. Su uso principalmente es obtener impresiones, cantidades, promedios, etc. agrupados con algún criterio, y parte de la base de que los datos originales cumplen o se puede hacer que cumplan con el agrupamiento necesario para su procesamiento.

¿Qué quiere decir todo esto? Mejor veámoslo con un ejemplo

Supongamos que tenemos un conjunto de datos ya cargado con la información del presentismo de cada alumno de la materia:

Legajo	Fecha	Presente
11709099	20200504	Verdadero
13423223	20200504	Falso
12936274	20200518	Falso
11709099	20200525	Verdadero

Si alguien nos preguntara cuántas faltas tiene el alumno con legajo 11709099 una forma sería recorrer todo el conjunto y preguntar si es el legajo buscado, en caso de serlo contar en caso que Presente sea Falso. El problema, una vez más, es la cantidad de recorridas que deberíamos hacer si necesitamos imprimir el siguiente listado

Legajo	Faltas
11709099	2
...	...

Deberíamos recorrer el conjunto completo por cada Legajo distinto que haya, y para complicar más la tarea, al no saber cuántos legajos son realmente no podremos utilizar una estructura auxiliar como un vector de forma óptima ya que estaremos obligados a declararlo de una cantidad excesiva por las dudas.

Utilizar corte de control nos permitirá obtener el listado deseado pero en una única recorrida

Pero cuidado, es **requisito indispensable** que los datos que contiene el archivo estén agrupados y ordenados por la condición de Corte de Control que se solicita, en este caso por legajo.

Una característica de los datos, para aplicar la técnica de corte de control, es que se repitan, en posiciones consecutivas, valores de aquellos campos sobre los que se pretende realizar el corte, en este caso el alumno o legajo más propiamente dicho.

Siguiendo con el ejemplo, dado que para un mismo legajo existen muchos registros de presentismo, deberemos evaluar mientras que el legajo no cambie el valor del presente y contar en caso que corresponda.

Veamos como quedaría nuestro algoritmo que imprime el listado deseado.

```
1   struct Presentismo {
2      int legajo;
3      int fecha;
4      bool presente;
5   };
6
7   void listarPresentismo(Presentismo vec[], int n) {
8      int i = 0;
9      int ausentes = 0;
10     int key;
11
12     // Inicializamos contadores, acumuladores, etc. generales
13
14     // El primer ciclo es el que recorre el lote completo
15     while(i < n) {
16
17        // Guardo el valor de la clave o agrupador
18        key = vec[i].legajo;
19
20        // Inicializo contadores, acumuladores, etc cada sublote
21        ausentes = 0;
22
23        // El segundo ciclo se mantiene por el sublote, mientras sea el mismo
24        // legajo y aun no se haya acabado el vector
25        while(i < n && key == vec[i].legajo) {
26           // Cuento si es un registro de ausente
27           if (!vec[i].presente) {
28              ausentes++;
29           }
30
31           i++;   // Avanza a la siguiente posicion
32        }
33
34        // Mostramos resultados por cada sublote (legajo)
35        cout << “Legajo: ” << key << “ faltas: ” << ausentes << endl;
36     }
37
38     // Mostramos resultados generales
39  }
```

Corte de Control por múltiples criterios

En el ejemplo anterior nuestro criterio de corte es el legajo dado que el listado era por cada materia, pero qué pasaría si en realidad la tecnicatura tuviera un único listado como el siguiente:

Materia	Legajo	Fecha	Presente
INGLES1	11709099	20200529	Verdadero
PROG1	11709099	20200521	Verdadero
PROG1	11709099	20200528	Falso
PROG2	1232342	20200517	Verdadero

Si el listado resultante ahora es las faltas por materia por alumno:

Materia	Legajo	Faltas
INGLES1	11709099	0
PROG1	11709099	1

Deberemos agregar un nuevo ciclo al algoritmo anterior para generar un nuevo sublote anidado, es decir nuestro primer nivel será el sublote de la materia, y el segundo el legajo.

Cabe remarcar que siempre para poder utilizar corte de control el conjunto de datos debe estar ordenado (o poder ordenarse) por los criterios que se necesiten utilizar en el corte de control.

¿Cuándo conviene usarlo?

Para poder responder la pregunta de si es necesario o útil utilizar corte de control para resolver un problema deberemos analizar los datos brindados y el listado o conjunto de datos al que queremos llegar. En el ejemplo anterior no hay otra forma de resolverlo más que el corte de control, dado que no sabemos la cantidad de alumnos total, tampoco las materias, y menos la cantidad de clases dictadas por cada una. En general el corte de control nos será de gran utilidad cuando no hay otra forma de realizar los cálculos por la imposibilidad de almacenar los contadores, acumuladores, etc. que precisaríamos para poder resolverlo.

Bibliografía utilizada

Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Programación en C. Segunda Edición. España: McGRAW-HILL/INTERAMERICANA DE ESPAÑA. S.A.U., 2005. ISBN-84: 481-9844-1.

● Brian W. Kernighan, Rob Pike. La práctica de la programación. Pearson Educación. Méjico (2000)

● Debugging with CodeBlocks: http://wiki.codeblocks.org/index.php?title=Debugging_with_Code::Blocks