

Algoritmos y Estructuras de Datos. TPL1. Trabajo Práctico de Laboratorio 1. [2014-08-30]

PASSWD PARA EL ZIP: **IBALB792CRZ1**

Ejercicios

[Ej. 1] **[large-even-list]** Dado un `vector<list<int>> &VL` buscar aquella lista `VL[j]` que contiene la máxima cantidad de pares y retornar la sublista de pares correspondientes (en el mismo orden que están en `VL[j]`). Por ejemplo, si

```
VL[0]: 0 1 2 3 4 5 7
VL[1]: 0 1 2 3
VL[2]: 2 2 2 1 0
```

entonces vemos que las listas contienen 3, 2, y 4 elementos pares. De tal forma que la función debe retornar los pares de la última `VL[2]`, es decir (2 2 2 0).

Consigna: Escribir la función

```
void large_even_list(vector< list<int> >&VL, list<int>&L);
```

Ayuda: Mantener una lista `Lmax` que tiene la lista con los elementos pares del máximo actual. Para `VL[j]` calcular la lista `tmp` de los elementos pares correspondientes y si su longitud es mayor que la de `Lmax` reemplazarla.

```
Lmax = lista vacia;
for (Lj in VL) {
    tmp = lista con elementos pares de Lj;
    if (tamaño de tmp > tamaño de Lmax) Lmax = tmp
}
```

[Ej. 2] **[interlaced-split]** Dada una lista de enteros `L` y un entero positivo `m` dividir a `L` en `m` sublistas (en un vector de listas `vector< list<int> > VL`) en forma **entrelazada** es decir $a_0, a_1, a_2 \dots$ van correspondientemente a las listas `VL[0], VL[1], VL[m-1], VL[0], VL[1] \dots`. Es decir, el elemento a_j va a la lista `VL[k]` donde $k=j \% m$.

Consigna: Escribir la función

```
void interlaced_split(list<int>&L, int m, vector< list<int> >&VL);
```

Por ejemplo, si `L = (0 1 2 3 4 5 6 7 8)` y `m=4` entonces debemos tener

```
VL[0]: 0 4 8
VL[1]: 1 5
VL[2]: 2 6
VL[3]: 3 7
```

[Ej. 3] **[interlaced-join]** Es la inversa de `interlaced_split()`. Dado un `vector< list<int> > VL` juntarlos en una lista `L` de uno a la vez. Es decir, primero los primeros elementos de `VL[0], VL[1], \dots, VL[m-1]`, después los segundos elementos, hasta que se acaben todas las listas.

Consigna: Escribir la función

```
void interlaced_join(vector< list<int> >&VL, list<int>&L);
```

Ejemplo 1: si `VL` es

```
VL[0]: 0 4 8
VL[1]: 1 5
VL[2]: 2 6
VL[3]: 3 7
```

entonces debe dar $L = (0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8)$ (la inversa del ejemplo presentado en la definición de `interlaced_split()`).

Nota: Si VL es el resultado de un `interlaced_split()` entonces las diferentes listas de VL difieren en longitud en a lo sumo 1, y las primeras son más largas que las últimas. En el ejemplo anterior la primera $VL[0]$ tiene 3 elementos y las demás tienen 2. Pero en general no podemos asumir que esto sea así, por ejemplo

```
VL[0]: 0 1
VL[1]: 2
VL[2]: 3 4
VL[3]: 5 6
VL[4]: 7
```

En cuyo caso el resultado esperado es $L = (0 \ 2 \ 3 \ 5 \ 7 \ 1 \ 4 \ 6)$.

Ayuda: Utilizar el siguiente pseudocódigo

```
L = lista vacía;
while (1) {
    if (todas las listas están vacías) break;
    for j in [0,N) {
        if (VL[j] no es vacía) {
            sacar el primer elemento de VL[j] e insertarlo al final de L;
        }
    }
}
```

Nota: Como mencionamos, la composición de `interlaced_split()` con `interlaced_join()` (en ese orden) es la identidad, es decir:

```
interlaced_split(L1,m,VL);
interlaced_join(VL,L2);
```

da siempre $L1 == L2$. Mientras que

```
interlaced_join(VL1,L);
m = VL1.size();
interlaced_split(L,m,VL2);
```

no necesariamente da $VL1 == VL2$. Esto sólo ocurre si las longitudes de los $VL[j]$ difieren entre sí en a lo sumo 1, y son decrecientes, es decir $VL[j+1] \leq VL[j]$.

Instrucciones

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado **program.cpp**. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Funciones utilitarias que pueden servir

```
void dump(list<int> &L, string s="");  
void dump(vector< list<int> > &VL, string s="");
```

Imprimen por pantalla los contenedores. Ambas son métodos en el namespace **aed**, de manera que hay que usarlos así:

```
aed::dump(VL);  
aed::dump(L);
```

- El TPL se autoevalúa, cada una de las funciones **Eval: :eval1()** a **eval3()** reporta una serie de diagnósticos. Deben dar todos **OK**, por ejemplo

```
Evaluando ejercicio 3  
EJ3|Caso0. Estado: OK  
EJ3|Caso1. Estado: OK  
EJ3|Caso2. Estado: OK  
EJ3|Caso3. Estado: OK  
ESTADO EJ3: total 4, ok 4, mal 0. Todos bien? OK
```

- Las funciones **eval()** tienen un segundo argumento **int vrbs** que es la **verbosidad** con la cual queremos que la clase evaluadora reporte errores o información adicional. Una primera forma de debug es poner el nivel máximo de verbosidad **vrbs=2**.

- Para cada ejercicio hay un segundo control que llamamos **seed/hash**, la función es del tipo

```
evallr(function f, int X, int vrbs);
```

donde **f** es la función del usuario a evaluar, **X** una semilla y **vrbs** la verbosidad.

La clase evaluadora genera una serie de contenedores a partir de la semilla, se los pasa a la función del usuario **f()**. Las respuestas de la **f()** van siendo procesadas por la función interna de hash que genera un **checksum H** de las respuestas. Por ejemplo para el primer ejercicio si **X=123** entonces el checksum es **H=523**. Una vez que el alumno termina su tarea se le pedirá que corra la clase evaluadora con un valor determinado de la semilla **X** y se comprobará que genere el valor correcto del checksum **H**.

Desde el punto de vista del alumno esto no trae ninguna complicación adicional, simplemente debe llenar el parámetro **X** con el valor indicado por la cátedra, recompilar el programa y correrlo. La cátedra verificará el valor de salida de **H**.