Curso de Programación en C++.
TPL0-2021. Trabajo Práctico de Laboratorio. [2021-04-28]

PASSWD PARA EL ZIP: VMRQ 6GWF 45PW

Ejercicios

[Ej. 1] **[isinverse]** Podemos representar una **permutación** de **n** objetos con un vector de enteros de longitud **n** cuyos elementos sean los enteros **[0,n)** reordenados de alguna forma. Por ejemplo si **n=4** entonces el vector sin permutar **[0,1,2,3]**, entonces la permutación **p=[1,2,3,0]** representa que el objeto 0 fue movido a la posición 1, el 1 al 2 y así siguiendo (una rotación hacia la derecha). Es decir el objeto **j** es movido a la posición **p[j]**. La rotación **inversa** es aquella que vuelve los objetos a su lugar, es decir en el caso anterior sería **q=[3,0,1,2]** porque manda el 1 al 0, el 2 al 1... (una rotación a la izquierda).
Otro ejemplo: La rotación inversa de **p=[1,0,2,3]** (intercambia los dos primeros objetos entre sí, y los dos últimos quedan inalterados) es ella misma **q=p=[1,0,2,3]** ya que sólo hay que volver a invertir los dos primeros.

Consigna: Escribir un predicado `bool isinverse(vector<int> &p,vector<int> &q);` que determina si las permutaciones **p,q** son una la inversa de la otra.

Ejemplos:

```
p=[1,2,3,0],   q=[3,0,1,2]   => true
p=[1,0,2,3],   q=[1,0,2,3]   => true
p=[1,2,0],     q=[2,0,1]     => true
p=[1,2,0],     q=[1,2,0]}    => false
p=[1,4,0,2,3], q=[2,0,3,4,1] => true
```

Ayuda: Recorrer los valores de **p** y verificar que si en la posición **j** de **p** está el valor **k**, entonces en la posición **k** de **q** debe estar el valor **j**.

[Ej. 2] **[mkmonotone]** Dado un vector de enteros **v** escribir una función `void mkmonotone(vector<int> &v,vector<vector<int> > &vv);` que particiona al vector en una serie de subvectores **vv** de tal forma que cada uno de ellos es monótono ascendente.

```
v:[1,2,0,1,2,3,2,3,4,5,3,4,5,6]},
=> vv:[[1,2],[0,1,2,3],[2,3,4,5],[3,4,5,6]]},

v:[5,9,3,9,6,4,1,1,2,4,1,9]},
=> vv:[[5,9],[3,9],[6],[4],[1,1,2,4],[1,9]]},

v:[1,6,4,6,8,4,7,2,2,1,0,0,3]},
=> vv:[[1,6],[4,6,8],[4,7],[2,2],[1],[0,0,3]]},
```

Ayuda:

- Mantener un índice `j` sobre `v` inicialmente en 0.
- Ir incrementando `j` mientras `v[j]<=v[j+1]` e ir insertando los elementos en un vector temporario `tmp`.
- Cuando ya `j` no se puede avanzar porque se viola la condición de monotonicidad insertar el temporario en `vv` y arrancar nuevamente desde la última posición.

[Ej. 3] **[incluido]** Dados dos vectores de enteros `v1,v2` escribir una función `bool incluido(vector<int> &v1,vector<int> &v2);` que determinar si el vector `v2` está incluido en el vector `v1`.

Ejemplos:

```
v1=[0,1,2,3,4,6]              v2=[1,2,3,4]    => true
v1=[0,1,2,3,4,6]              v2=[1,2,3,4,5]  => false
v1=[5,9,3,9,6,4,1,1,2,4,1,9]  v2=[3,9,6,4,5]  => false
```

Ayuda:

- Si la longitud de `v2` es mayor que la de `v1` entonces no es necesario hacer nada ya que claramente no puede estar contenido.
- Caso contrario, si las dimensiones son `n1,n2` entonces el vector `v2` debe estar contenido en `v1` en un cierto rango `[k,k+n2)`. La estrategia es ir buscando para cada `k` y chequear si a partir de `k` encontramos el vector `v2` o no.
- Las posiciones de `k` a buscar van desde 0 hasta `n1-n2` ya que más allá de esa posición no hay espacio para que esté contenido `v2`.
- Entonces, para cada una de esas posiciones `k` verificar si los `n2` elementos siguientes de `v1` (o sea el rango `[k,k+n2)`) es igual (elemento a elemento) a `v1`.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más arriba; 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.
- Salvo indicación contraria 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.
- Para cada ejercicio hay dos funciones de evaluación, por ejemplo si `f` es la función a evaluar tenemos

```
ev.eval<j>(f,vrbs);
hj = ev.evalr<j>(f,seed); // para SEED=123 debe dar Hj=170
```

`j` es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (`eval<1>` y `evalr<1>`). La primera `ev.eval<j>(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

```
m: 10, k: 3
T(ref): (10 (7 (4 1) 1) (4 1) 1)
T(user): (10 (7 (4 1) 1) (4 1) 1)
EJ1|Caso0. Estado: OK
```

- **ucase**: Además las funciones **eval()** tienen dos parámetros adicionales:
  Eval::eval(func_t func,int vrbs,int ucase);
  El tercer argumento **ucase** (caso pedido por el usuario), permite que el usuario seleccione uno solo de todos los ejercicios para chequear. Por defecto está en **ucase=-1** que quiere "hacer todos". Por ejemplo **ev.eval4(prune_to_level,1,51);** corre sólo el caso 51.
- **Archivo con casos tests JSON:** Los casos test que corre la función **eval<j>** están almacenados en un archivo **test1.json** o similar. Es un archivo con un formato bastante legible. Abajo hay un ejemplo. **datain** son los datos pasados a la función y **output** la salida producida por la función de usuario. **ucase** es el número de caso.

```
{ "datain": {
   "T1": "( 0 (1 2) (3 4 5 6) )",
   "T2": "( 0 (2 4) (6 8 10 12) )",
   "func": "doble" },
  "output": { "retval": true },
  "ucase": 0 },
```

- La segunda función **evalr<j>** es el chequeo que llamamos **SEED/HASH**. La clase evaluadora genera una serie de contenedores a partir de la semilla **seed**, 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 **seed=123** entonces el checksum es **H=523**. Una vez que el alumno termina su tarea se le pedirá que corra la función **evalr<j>()** de la clase evaluadora con un valor determinado de la semilla **seed** y se comprobará que genere el valor correcto del checksum **H**.