1. (1 punto) (a) Supongamos que hacemos las 3 siguientes afirmaciones:

(a.1) En un esquema de hashing puede usarse como función hash h(x)=(M-1) - (x% M) con M primo.
(a.2) Si insertamos un conjunto de (n>2) enteros ordenados en un ABB y en un APO, la altura de ambos es diferente.
(a.3) El orden relativo en que las hojas se listan en los recorridos preorden, inorden y postorden de un APO es el mismo en los 3 casos, pero no es así en un AVL en que es diferente en los 3 casos.
(1-a): Las tres son ciertas (1-b): Dos son ciertas y una falsa (1-c): Dos son falsas y una cierta; (1-d): Las tres son falsas. **Razona la respuesta.**

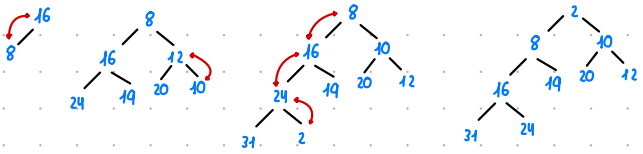
1) Verdadero, ya que solo devuelve valores entre 0 y M-1

2) Verdadero, en un ABB, si la secuencia es ordenada solo tendrá una única rama y la altura será el número de enteros insertado.
Un APO tiene como característica que todos los niveles deben estar completos, menos el último que está empujado a la izquierdo. Por lo que la altura será menor al número de enteros insertado

3) Falso, tanto en los APO, ABB, AVL, en todos los árboles binarios las hojas aparecen en el mismo orden, de izquierda a derecha, tanto en el preorden, inorden y postorden.

Respuesta: 1b

(b) Si inserto las claves {16, 8, 12, 24,19, 20, 10, 31, 2} en un APO de enteros:
(b1) Hay que hacer dos intercambios padre-hijo **(b2)** Hay que hacer tres intercambios padre-hijo
(b3) Hay que hacer cuatro intercambios padre-hijo **(b4)** Todo lo anterior es falso. **Mostrar el arbol final**



Respuesta: b4 (5 intercambios)

(c) Dados los siguientes recorridos:
preorden = {H,Z,W,X,Y,T,R,Q,V,L}, y postorden = {Z,W,X,R,Y,Q,V,L,T,H}
(c1) No hay ningún árbol binario con esos recorridos asociados; **(c2)** Hay 1 solo árbol binario con esos recorridos asociados: **(c3)** Hay dos árboles binarios con esos recorridos asociados ; **(c4)** Hay múltiples árboles binarios con esos recorridos asociados

descendientes de Z → CONTRADICCIÓN

pre = { H Z W X Y T R Q V L }
H_in

post = { Z W X R Y Q V L T H }
H_der raíz

descendientes de T

Respuesta: c1

(d) Dada la lista de enteros **list<int> L({1,5,7,9,4});** hacemos sobre ella las siguientes operaciones:
{ list<int>::iterator p,q;
q = L.begin(); p=q++; p++; q++; p = L.erase(p);}
(4-a) *q=5 y p es inválido. **(4-b)** *p=7 y *q=7 **(4-c)** *p=9 y *q=7 **(4-d)** Todas las anteriores son incorrectas. **Razona la respuesta.**

Respuesta: 4b

2. (1 punto) Se define una **matriz hipersingular** como una **matriz nxn** en la que se van alternando elementos pares e impares en círculos (siempre el mismo valor en cada caso). Un ejemplo de este tipo de matrices es la matriz que se muestra.

2	**2**	**2**	**2**	**2**	**2**
2	5	5	5	5	**2**
2	5	**8**	**8**	5	**2**
2	5	**8**	**8**	5	**2**
2	5	5	5	5	**2**
2	**2**	**2**	**2**	**2**	**2**

Proponer una representación eficiente (basada en el tipo vector<list>)para la matriz e implementar:
(a) el **operator()** que devuelva el elemento en la fila fil, columna col.
(b) un iterador que itere sobre los elementos impares de la matriz. Han de implementarse (aparte de las de la clase iteradora) las funciones **begin()** y **end()**.

```
class matriz_hipersingular {
    private:
        vector<list<int>> datos;
    public:
        int &operator()(int i, int j) {
            assert(i>=0 && i<datos.size() && j>=0 && j<datos.size());
            list<int>::iterator it = datos[i].begin();
            advance(it, j);
            return *it;
        }
        class iterator {
            private:
                vector<list<int>>::iterator it_row, final;
                list<int>::iterator it_col;
            public:
                iterator(){}
                bool operator==(const iterator &i) const {
                    return (it_row==final && i.it_row==i.final) ||
                           (it_row==i.it_row && it_col==i.it_col);
                }
                bool operator!=(const iterator &i) const {
                    return !(*this==i);
                }
                int &operator*(){
                    return *it_col;
                }
                iterator &operator++(){
                    do{
                        ++it_col;
                        if(it_col==it_row->end()){
                            ++it_row;
                            if(it_row!=final)
                                it_col=it_row->begin();
                        }
                    }while(it_row!=final && (*it_col)%2==0);
                    return *this;
                }
                friend class matriz_hipersingular;
        };
        iterator begin(){
            iterator i;
            i.it_row=datos.begin();
            i.final=datos.end();
            if(i.it_row!=datos.end()){
                i.it_col=i.it_row->begin();
                if((*(i.it_col))%2==0)
                    ++i;
            }
            return i;
        }
        iterator end(){
            iterator i;
            i.it_row=datos.end();
            i.final=datos.end();
            return i;
        }
};
```

3. (1 punto) Implementar una función

int max_sublist_m(list<int> &L, int m);

que dada una lista de enteros, L y un entero m tal que 0<m<=L.size() , encuentre y devuelva el valor de la mayor suma de entre todas las posibles sumas de sublistas de longitud m.

Por ejemplo: Si L={**1,2,-5,4,-3,2**}

m=1 ⇒ **4** (por {4} **m=2** ⇒ **3** (por {1,2}) **m=3** ⇒ **3** (por {4,-3,2})

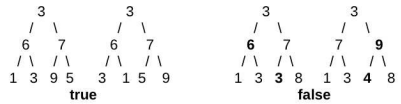
m=4 ⇒ **2** (por {1,2,-5,4}) **m=5** ⇒ **0** (por {2,-5,4,-3,2}

```
int max_sublist_m (list<int> &L, int m){
    int max_suma = 0;

    auto final = L.end();
    for(int i=0; i<m-1; i++)
        --final;

    for(auto it = L.begin(); it != final; ++it){
        auto it2 = it;
        int suma = 0;
        for(int i=0; i<m; i++, ++it2)
            suma += (*it2)
        if(suma > max_suma) max_suma = suma;
    }
    return max_suma;
}
```

4. (1 punto) Implementar una función
bool desordenequal (bintree<int> & A, bintree<int> & B);

que reciba dos árboles binarios y devuelva true si son "desordenadamente" iguales. Se dice que dos árboles son "desordenadamente" iguales si:
(a) las raíces de ambos son iguales,
(b) Los hijos de la raíz de uno son los mismos hijos de la raíz del otro (no importa el orden, por lo que esta condición implica que las raíces tienen los mismos hijos, aunque podrían estar en diferente orden), y
(c) la condición se cumple para cada par de subárboles de cada par de nodos equivalentes (uno de cada bintree).
Como **estructura auxiliar** solo se permite usar un **<map>**

Ejemplo:




```
bool desordenequal ( bintree<int> &A , bintree<int> &B ) {
    bintree<int>::node nA = A.root();
    bintree<int>::node nB = B.root();
    if ((*nA) != (*nB)) return false;
    return desordenequal ( nA , nB );
}
```

```
bool desordenequal ( bintree<int>::node nA , bintree<int>::node nB ) {
    int n_hijos_A = 0, n_hijos_B = 0;
    if( !nA.left().null() ) n_hijos_A ++;
    if( !nA.right().null() ) n_hijos_A ++;
    if( !nB.left().null() ) n_hijos_B ++;
    if( !nB.right().null() ) n_hijos_B ++;
    if ( n_hijos_A != n_hijos_B ) return false;
    if ( n_hijos_A == 2 ) {
        if( (*(nA.left())) == (*(nB.left())) || (*(nA.left())) == (*(nB.right())) &&
            (*(nA.right())) == (*(nB.left())) || (*(nA.right())) == (*(nB.right())) )
            return desordenequal (nA.left() , nB.left()) && desordenequal (nA.righ() , nB.right()) ;
        else
            return false;
    } else if ( n_hijos_A == 1 ) {
        bintree<int>::node nA_hijo = nA.left();
        bintree<int>::node nB_hijo = nB.left();
        if( !nA.right().null() ) nA_hijo = nA.right();
        if( !nB.right().null() ) nB_hijo = nB.right();
        if ( (*nA_hijo) == (*nB_hijo) )
            return true;
        else
            return false;
    } else {
        return true
    }
}
```

5. (1 punto) Implementar una función
bool en_todos (list<set<int> > & LS, set<int> & S);
que devuelva true si algún conjunto está incluido en todos los demás y tal conjunto lo devuelva en S
P.ej: Si LS = [{1,2,3}, {2,3,4}] devuelve FALSE
Si LS= [{**1,2,3**}, {**1,2,3**,4}, {**1,2,3**}] devuelve TRUE y S={**1,2,3**}
Si LS= [{**1**,2,3}, {**1**}, {**1**,2}] devuelve TRUE y S={**1**}

```
bool en_todos (list<set<int>> & LS, set<int> & S) {
    bool acabar = false;
    for (auto it = LS.begin(); it != LS.end() && !acabar; ++it) {
        bool condicion = true;
        for (auto it2 = LS.begin(); it2 != LS.end() && condicion; ++it2)
            if (it != it2) {
                if ((*it2).size() < (*it).size())
                    condicion = false
                auto it_set = it->begin();
                auto pos = (*it2).find(*it_set);
                if (pos == (*it2).end())
                    condicion = false;
                auto pos_anterior = pos;
                --pos_anterior;
                ++it_set;
                for (; it_set != it->end() && condicion; ++it_set) {
                    pos = (*it2).find(*it_set);
                    if (pos == (*it2).end())
                        condicion = false;
                    else {
                        ++pos_anterior;
                        if (pos_anterior != pos)
                            condicion = false;
                    }
                }
            }
        if (condicion) {
            S = (*it);
            acabar = true;
        }
    }
    return acabar;
}
```

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

Que te den **10 € para gastar** es una fantasía. ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Quiero el cash

Consulta condiciones **aquí**

do your thing

6. (1 punto) Determinar paso a paso las estructuras resultantes tras:

1-a **Insertar** las claves {18, 26, 30, 51, 20, 72, 37, 75, 23, 24} en una **Tabla Hash cerrada** de tamaño 13. A continuación **borrar** el 23 y el 51 y finalmente insertar el valor 61. Resolver las colisiones usando **rehashing doble**.

$M = 13$

$h_1(k) = h(k) = k \% 13$

- $h_i(k) = (h_{i-1}(k) + h_0(k)) \% 13 \quad i = 2, 3, 4 \ldots$
- $h_0(k) = 1 + (k \% 11)$

	18	26	30	51	20	72	37	75	23	24	61
$h_1(k)$	5	0	4	12	7	7	11	10	10	11	9
$h_0(k)$	8	5	9	8	10	7	5	10	2	3	7
$h_2(k)$						1			12	0	
$h_3(k)$									1	3	
$h_4(k)$									3	6	

	K	dir	estado
0	26		X
1	72		X
2			
3	23		B
4	30		X
5	18		X
6	24		X
7	20		X
8			
9	61		X
10	75		X
11	37		X
12	51		B

1-b **Insertar** las claves {11, 49, 62, 50, 72, 20, 37, 75, 23, 24, 33} en una **tabla hash abierta de tamaño 13,** resolviendo las colisiones usando árboles **AVL** en lugar de listas.

$h(k) = k \% 13$

$h(11) = 11$
$h(49) = 10$
$h(62) = 10$
$h(50) = 11$
$h(72) = 7$
$h(20) = 7$
$h(37) = 11$
$h(75) = 10$
$h(23) = 10$
$h(24) = 11$
$h(33) = 7$