Problema 1

```
#include <iostream>
#include <list>
using namespace std;


list<int>::iterator i_pos_insertar_rec (int x, list<int>& sel, list<int>::iterator it) {
/* Pre: sel está ordenada crecientemente. it apunta a un elemento de sel o a sel.end().
Los elementos de la sublista comprendida entre sel.begin() y el elemento anterior al que
apunta it son menores o iguales que x. */

/* Post: El resultado apunta al elemento de sel delante del cual se
debe insertar x para que sel siga estando ordenada crecientemente
después de la inserción, si existe dicho elemento; en otro caso, el
resultado apunta a sel.end(). */

  list<int>::iterator pos_it;
  if (it == sel.end()) pos_it = it;
  else if (x < *it) pos_it = it;
  else {
    ++it;
    pos_it = i_pos_insertar_rec (x, sel, it);
  }
  return pos_it;
}



list<int>::iterator pos_insertar_rec (int x, list<int>& sel) {
/* Pre: sel está ordenada crecientemente. */

/* Post: El resultado apunta al elemento de sel delante del cual se
debe insertar x para que sel siga estando ordenada crecientemente
después de la inserción, si existe dicho elemento; en otro caso, el
resultado apunta a sel.end(). */

  return i_pos_insertar_rec (x, sel, sel.begin());
}



void seleccionados (int n, list<int>& sel) {
/* Pre: En el canal estándar de entrada hay una secuencia de naturales no repetidos
terminada en 0; n > 0; sel = SEL; y SEL es vacía. */

/* Post: sel contiene los n números más altos de la secuencia ordenados crecientemente. */

  int x;
  cin >> x;
/* Inv: La parte tratada de la secuencia tiene longitud m.
    sel contiene los min(n,m) números más altos de la parte tratada de la secuencia
    ordenados crecientemente.
*/
  while (x ≠ 0) {
```

```
    bool insertar;
    list<int>::iterator it;
    if (sel.empty()) {
        insertar = true;
        it = sel.end();
    }
    else if (sel.size() < n) {
        insertar = true;
        it = pos_insertar_rec(x, sel);
    }
    else {
        it = sel.begin();
        if (*it < x) {
            it = sel.erase(it);
            insertar = true;
            it = pos_insertar_rec(x, sel);
        }
        else insertar = false;
    }
    if (insertar) sel.insert(it, x);
    cin >> x;
  }
}


int main() {
  int n;
  cin >> n;
  list<int> sel;
  seleccionados(n, sel);
  for (list<int>::const_iterator it = sel.begin(); it != sel.end(); ++it) {
    cout << " " << *it;
  }
  cout << endl;
}
```

Problema 1

Posible solución del problema 2 del primer parcial de Pro2, otoño 2017.

Problema 2.1: Explicación de la compatibilidad de A en función de la compatibilidad de sus hijos A1, A2, su raiz x, y otras cantidades sobre los árboles hijos. La cantidad sobre los árboles que podemos utilizar es cuantos elementos distintos aparecen en el árbol. Esta cantidad será 1 si solo aparecen 1's o solo 0's, y será 2 si en ese árbol aparecen nodos con 1 y nodos con 0. Llamaremos a esta cantidad num.

1) *El valor x es diferente de los valores de las raices de los hijos. En este caso el árbol no es compatible, porque los árboles A1 y A2 tienen raices iguales, pero no pueden pertenecer al mismo subárbol.*
2) *El valor de x es igual al valor de la raiz de uno de sus hijos. Supongamos que A1 tiene en su raiz el valor x. En este caso todos los nodos de A1 han de tener el mismo valor, y la misma propiedad se ha de dar con A2. Por tanto la cantidad num de A1 ha de ser 1, y la de A2 también.*
3) *El valor de x es igual a las raices de los dos hijos. En este caso, A1 y A2 han de ser compatibles, y sus correspondientes cantidades num no pueden ser las dos 2. Al menos una de ellas ha de ser 1.*

Una manera alternativa de solucionar el problema es contar cuantas raices de hijos son diferentes a la raiz del padre. En este contexto, el caso 1) suma 2 a ese contador, el caso 2) suma 1, y el caso 3) no suma nada. El código de esta solución estará al final de este documento.

Problema 2.2:

```
bool i_compatible (const BinTree<int> &a,int &num);
//Pre: los nodos de a son 0's o 1's. Los nodos tienen 0 o 2 hijos.
//Post: el valor que se devuelve indica si a es compatible. num indica cuantos
        valores distintos tiene a. num es 1 si a solo tiene 1's o solo tiene
        0's, y num es 2 si a tiene 0's y 1's. Si a es vacio num es igual a 0.
```

Problema 2.3:

```
bool compatible(const BinTree<int> &a){
  int num;
  return i_compatible (a,num);
}
```

Problema 2.4 y 2.5:

```
bool i_compatible (const BinTree<int> &a,int &num){
  bool comp, compi, compd;
  int numi=0;
  int numd=0;
  BinTree<int> ai;
  BinTree<int> ad;
  if(not a.empty()){
    int arrel =a.value ();
    ai=a. left ();
    ad=a.right ();
    if(ai .empty()){
      comp=true;
      num=1;
    }
```

```
    else{
      compi=i_compatible(ai, numi);
      //HI: lo que devuelve la llamada indica si el hijo izquierdo de a, ai, es
      //compatible. numi tiene el numero de valores diferentes que contiene ai.
      if(compi) compd=i_compatible(ad, numd);
      comp=compi and compd and (arrel==ai.value() or arrel==ad.value());
      if(ai.value()==ad.value() and comp){
        comp=(numi<2 or numd<2);
        num=max(numi,numd);
      }
      if(ai.value()≠ad.value() and comp){
        comp=(numi==1 and numd==1);
        num=2;
      }
    }
  }
  else{
    comp=true;
    num=0;
  }
  return comp;
}
```

Problema 2.4 *alternativo* :

```
bool i_compatible (const BinTree<int> &a,int &num){
  bool comp;
  if(a.empty()){
    comp=true;
    num=0;
  }
  else{
    BinTree<int> ai;
    BinTree<int> ad;
    ai=a.left();
    ad=a.right();
    if(ai.empty()){
      comp=true;
      num=0;
    }
    else{
      bool compi, compd;
      int numi, numd;
      compi=i_compatible(ai, numi);
      if(compi) compd=i_compatible(ad, numd);
      num=numi+numd;
      if(ai.value()==ad.value() and a.value()≠ai.value()) num=num+2;
      if(ai.value()≠ad.value()) ++num;
      comp=compi and compd and num<2;
    }
  }
  return comp;
}
```