Relación 2

Ejercicio 1

Para un vector creciente:

```
5  int indiceK (int *v, int ini, int fin) {
6    if (fin < ini)
7      return -1;
8
9    int mitad = (fin + ini) / 2;
10
11   if (v[mitad] == mitad)
12     return mitad;
13
14   else if (v[mitad] > mitad)
15     return indiceK(v, ini, mitad);
16
17   else
18     return indiceK(v, mitad+1, fin);
19 }
```

- Si v[mitad] es igual que mitad, hemos encontrado el índice.
- Si v[mitad] es estrictamente mayor que mitad, puesto que el vector es creciente el índice no se puede encontrar en la parte superior del vector.
- En otro caso, comprobamos el vector en la parte superior.

Para un vector decreciente:

```
int indiceK (int *v, int ini, int fin) {
  if (fin < ini)
    return -1;

  int mitad = (fin + ini) / 2;

  if (v[mitad] == mitad)
    return mitad;

  else if (v[mitad] < mitad)
    return indiceK(v, ini, mitad);

  else
    return indiceK(v, mitad+1, fin);
}
```

Simplemente se sustituye en la condición de la línea 14 (del código anterior) el > por un <.

El orden de eficiencia es ambas funciones es de O(log(n)), esto se demuestra de la siguiente manera:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$
$$\downarrow \qquad n = 2^m$$

$$T(2^m) - T(2^{m-1}) = 1$$

$$\downarrow \qquad x = T(2^m) \qquad 1 = b^m * p(m) \qquad b = 1 \quad p(m) = 1 \quad d = 0$$

$$(x-1)(x-1) = 0$$

$$T(2^m) = c_1 * 1^m + \ c_2 * m * 1^m$$
$$T(n) = c_1 * 1^{\log_2 n} + \ c_2 * \log_2 n \ * 1^{\log_2 n}$$

$$T(n) = c_1 + \ c_2 * \log_2 n$$

Como c_1 y c_2 son positivos el orden de eficiencia es θ **($\log_2 n$).**

Ejercicio 3

Precondición: el vector de edificios está ordenado por menor xmin.

- Si solo hay 1 edificio, devolverlo
- Si hay 2 edificios:
  - Si se cruzan:
    - La altura máxima de uno hasta donde se cruzan, la altura restante
  - Si no se cruzan:
    - Concatenarlos
- En otro caso:
  - Dividir en dos el vector de edificios recursivamente
  - Comprobar los edificios de la mitad por si se cruzan.

```cpp
struct Edificio {
  int xmin, xmax, h;
};
```

```cpp
// @pre Se "tapan" los edificios (xmax > xmin)
// @return 2 edificios o 1 si uno de ellos "cubre" al otro en su totalidad
vector<Edificio> Mezcla2Edif(Edificio &i, Edificio &j) {
  int hi = i.h, hj = j.h;

  Edificio uno, dos;
  vector<Edificio> res;

  if (hi == hj) {
    uno.xmin = i.xmin;
    uno.xmax = j.xmax;
    uno.h = hi;

    res.push_back(uno);
  }
  else if (hi > hj) {
    uno.xmin = i.xmin;
    uno.xmax = i.xmax;
    uno.h = hi;

    dos.xmin = i.xmax;
    dos.xmax = j.xmax;
    dos.h = hj;

    res.push_back(uno);
    res.push_back(dos);
  }
  else {
    uno.xmin = i.xmin;
    uno.xmax = j.xmin;
    uno.h = hi;

```

```
    dos.xmin = j.xmin;
    dos.xmax = j.xmax;
    dos.h = hj;

    res.push_back(uno);
    res.push_back(dos);
  }

  return res;
}

vector<Edificio> Skyline (vector<Edificio> edif, int i, int j) {
  vector<Edificio> resultado;
  int n = j - i + 1;

  if (n == 1)
    resultado.push_back(edif[i]);
  else if (n == 2) {
    if (edif[i].xmax > edif[j].xmin) {
      vector<Edificio> mezcla = Mezcla2Edif(edif[i], edif[j]);

      int tam = mezcla.size();
      for (int a=0; a < tam; a++)
        resultado.push_back(mezcla[a]);

    }
    else {
      resultado.push_back(edif[i]);
      resultado.push_back(edif[j]);
    }
  }
  else {
    vector<Edificio> primeraMitad = Skyline(edif, i, i+(n/2)),
                     segundaMitad = Skyline(edif, i+(n/2)+1, j),
         interseccion = Mezcla2Edif(primeraMitad[primeraMitad.size() - 1],
                                    segundaMitad[segundaMitad.size() - 1]);

    // Concatenamos los edificios por orden sin la intersección
    int tamP = primeraMitad.size();
    for (int a=0; a < (tamP-1); a++)
      resultado.push_back(primeraMitad[a]);

    // La intersección
    int tamI = interseccion.size();
    for (int a=0; a < tamI; a++)
      resultado.push_back(interseccion[a]);


    int tamS = segundaMitad.size();
    for (int a=1; a < tamS; a++)
      resultado.push_back(segundaMitad[a]);
  }

  return resultado;
}
```

La eficiencia de Mezcla2Edif es O(1) puesto que es un mera comparación.

La de la función Skyline es:

$$T(n) = n + 2 * T\left(\frac{n}{2}\right)$$

$$\downarrow \qquad n = 2^m$$

$$T(2^m) - 2 * T(2^{m-1}) = 2^m$$

$$\downarrow \qquad x = T(2^m) \qquad 2^m = b^m * p(m) \qquad b = 2 \qquad p(m) = 1 \quad d = 0$$

$$(x-2)(x-2) = 0$$
$$T(2^m) = c_1 * 2^m + c_2 * m * 2^m$$
$$T(n) = c_1 * 2^{\log_2 n} + c_2 * \log_2 n * 2^{\log_2 n}$$
$$T(n) = c_1 * n + c_2 * \log_2 n * n$$

Como c_1 y c_2 son positivos el orden de eficiencia es θ **($\log_2$(n) *n).**

Ejercicio 4

```
int TorYTuer(int *tornillos, int *tuercas, int ini, int fin) {
  pair<int,int> pivote1 = Pivote(tornillos, ini, fin, tuercas[0]);
                pivote2 = Pivote(tuercas, ini, fin, tornillos[pivote1.first]);

  TorYTuer(tornillos, tuercas, ini, pivote2.first-1);
  TorYTuer(tornillos, tuercas, pivote2.second+1, fin);
}

pair<int, int> Pivote (int *v, int i, int j, int piv) {
  int k, l;

  bool encontrado;

  // Puesto que el pivote puede no estar en la primera posición, lo buscamos
  encontrado = (v[i] == piv) ? true : false;

  for (int c=i; c<j && !encontrado; c++)
    if (v[c] == piv) {
      swap(v+c, v+i); encontrado = true; }

  k = i;
  l = j+1;

  do {
    k += 1;
  } while (v[k] <= piv && k < j);

  do {
    l -= 1;
  } while (v[l] > piv);

  while (k < l) {
```

```
    swap(v+k, v+l);

    do {
      k += 1;
    } while (v[k] <= piv);


    do {
      l -= 1;
    } while (v[l] > piv);
  }

  swap(v+i, v+l);

  pair<int,int> pivotes = OrdenarPivotes(v, i, l);

  return pivotes;
}
pair<int,int> OrdenarPivotes(int *v, int ini, int fin) {
  int valor = v[fin],
      i=ini, j=fin-1;

  while (i < j) {
    if (v[i] == valor) {
      swap(v[i], v[j]);
      j--;
    }
    else
      i++;
  }

  pair<int,int> piv;

  piv.second = fin;

  for (i=ini; i<fin; i++)
    if (v[i] == valor) {
      piv.first = i;
      return piv;
    }

  piv.first = i;

  return piv;
}
```

El algoritmo Pivote busca la posición donde va el elemento pasado como parámetro (al existir la posibilidad de haber repetidos, se devuelve la posición primera y la última) y sitúa el elemento en esa posición.

OrdenarPivotes recorre el vector juntando los pivotes, puesto que recorre el vector linealmente, es de orden O(n).

Pivote (sin contar la llamada a OrdenarPivotes) es también **O(n)** puesto que se recorre el vector colocando el pivote en su posición. Por la regla de la suma la función entera es de orden **O(n)**.

Por tanto, la eficiencia de TorYTuer es:

- En el peor caso el pivote se coloca en la primera posición del vector:

$$T(n) = 2 * Pivote + T(n-1)$$
$$T(n) - T(n-1) = 2 * n$$

$$\downarrow \qquad x = T(n) \qquad 2 * n = b^n * p(n) \qquad b = 1 \qquad p(n) = n \qquad d = 1$$

$$(x-1)(x-1)^2 = 0$$

$$T(n) = c_1 * 1^n + \ c_2 * n * 1^n + \ c_3 * n^2 * 1^n$$

Como c_1 , c_2 y c_3 son positivos el orden de eficiencia es **O(n²)**.

- En el caso mejor y promedio se coloca en la mitad:

$$T(n) = 2 * Pivote + 2 * T\left(\frac{n}{2}\right)$$
$$\downarrow \qquad n = 2^m$$

$$T(2^m) - 2 * T(2^{m-1}) = 2 * 2^m$$

$$\downarrow \qquad x = T(2^m) \qquad 2 * 2^m = b^m * p(m) \qquad b = 2 \qquad p(m) = 2 \qquad d = 0$$

$$(x-2)(x-2) = 0$$
$$T(2^m) = c_1 * 2^m + \ c_2 * m * 2^m$$
$$T(n) = c_1 * 2^{\log_2 n} + \ c_2 * \log_2 n * 2^{\log_2 n}$$
$$T(n) = c_1 * \ n + \ c_2 * \log_2 n * \ n$$

Como c_1 y c_2 son positivos el orden de eficiencia es θ **($\log_2$(n) *n)**.

Ejercicio 5

```
pair<int,int> Subsecuencia (int *v, int ini, int fin) {
  pair<int,int> resultado;

  if ((fin-ini) == 0) {
    resultado.first = resultado.second = ini;
  }
  else if ((fin-ini) == 1) {
    resultado.first = ini;

    if (v[ini] <= v[fin])
      resultado.second = fin;
    else
      resultado.second = ini;
  }
  else {
    int mitad = (fin-ini)/2;
    pair<int,int> primeraMitad = Subsecuencia(v, ini, mitad),
          segundaMitad = Subsecuencia(v, mitad+1, fin);

    // Si están juntos los dos subvectores y continua la secuencia creciente
    if ((segundaMitad.first-1) == primeraMitad.second)
          && (v[segundaMitad.first] >= v[primetaMitad.second])) {
      resultado.first = primetaMitad.first;
      resultado.second = segundaMitad.second;
    }
    else {
      int long1 = primetaMitad.second - primeraMitad.first + 1;
          long2 = segundaMitad.second - segundaMitad.first + 1;

      if (long1 > long2)
        resultado = primetaMitad;
      else
        resultado = segundaMitad;
    }
  }

  return resultado;
}
```

La eficiencia del algoritmo es:

$$T(n) = 1 + 2 * T\left(\frac{n}{2}\right)$$

$$\downarrow \qquad n = 2^m$$

$$T(2^m) - 2 * T(2^{m-1}) = 1$$

$$\downarrow \qquad x = T(2^m) \qquad 1 = b^m * p(m) \qquad b = 1 \qquad p(n) = 1 \qquad d = 0$$

$$(x-2)(x-1) = 0$$
$$T(2^m) = c_1 * 1^m + c_2 * 2^m$$
$$T(n) = c_1 * 1^{\log_2 n} + c_2 * 2^{\log_2 n}$$
$$T(n) = c_1 + c_2 * n$$

Como c_1 y c_2 son positivos el orden de eficiencia es θ **(n).**

Ejercicio 6

No lo terminaba de conseguir por lo que lo he completado con el algoritmo que hay en Prado. No caía en como calcular la subsecuencia máxima de la fusión de las dos mitades.

```
int SumaMaxima (int *v, int &ini, int &fin) {
  int suma = 0;

  if ((fin-ini) == 0) {
    suma = v[ini];
  }
  else if ((fin-ini) == 1) {
    suma = v[ini] + v[fin];

    if (v[ini] > suma)
      suma = v[ini];

    if (v[fin] > suma)
      suma = v[fin];

  }
  else {
    int mitad = (fin-ini)/2,
        iniPrimera = ini, finPrimera = mitad,
        primeraMitad = SumaMaxima(v, iniPrimera, finPrimera),
        iniSegunda = mitad+1, finSegunda = fin,
        segundaMitad = SumaMaxima(v, iniSegunda, finSegunda);

    int maximoMitades, maximoIzq, maximoDch;

    // El máximo de las dos mitades y su intervalo
    maximoMitades = (primeraMitad > segundaMitad) ? primeraMitad : segundaMitad;

    if (maximoMitades == primeraMitad) {
      ini = iniPrimera;
      fin = finPrimera;
    }
```

```
      else {
        ini = iniSegunda;
        fin = finSegunda;
      }

      // Se calcula el máximo de las partes que mezclan de las dos mitades

       int puntero = mitad-1,
           punteroInicio;
       int maxProvisional=numeric_limits<int>::min();

       // Ver la maxima subsecuencia de la parte izquierda con el centro
       for (puntero; puntero >= finPrimera; puntero--){
        suma += vector[puntero];

        if (maxProvisional<suma){
          maxProvisional=suma;
          pi=posi;
        }
       }

       if (suma + primeraMitad > maxProvisional){
         punteroInicio = iniPrimera;
         maxProvisional = suma + primeraMitad;
       }


       int punteroFin = mitad;
       puntero = mitad;
       suma = maxProvisional;

       // A la subsecuencia máxima anterior añadirle (comprobando si aumenta)
       // con la segunda mitad
       for (puntero; puntero < iniSegunda; puntero++){
         suma += vector[puntero];

         if (maxProvisional < suma){
           maxProvisional = suma;
           punteroFin = puntero;
         }
       }

       if (suma + segundaMitad > maxProvisional) {
         punteroFin = finSegunda;
         maxProvisional = suma + segundaMitad;
       }

       // Ver si la parte que incluye al centro es mayor que una de las
       // subsecuencias de las mitades
       if (maxProvisional > maximoMitades){
         maximoMitades = maxProvisional;
         ini = punteroInicio;
         fin = punteroFin;
       }

       suma = maximoMitades;
    }

    return suma;
  }
```

La eficiencia de este algoritmo es la siguiente:

Puesto que contiene 2 llamadas recursivas a sus dos mitades y unos bucles que en total (y en el peor caso) recorren n-2, la ecuación recurrente es:

$$T(n) = n - 2 + 2 * T\left(\frac{n}{2}\right)$$

$$\downarrow \qquad n = 2^m$$

$$T(2^m) - 2 * T(2^{m-1}) = 2^m - 2$$

$$\downarrow \qquad x = T(2^m) \qquad 2 * 2^m = b^m * p(m) \qquad b = 2 \qquad p(m) = 2 \qquad d = 0$$

$$\downarrow \qquad 2 = b^m * p(m) \qquad b = 1 \qquad p(m) = 2 \qquad d = 0$$

$$(x-2)(x-2)(x-1) = 0$$

$$T(2^m) = c_1 * 2^m + c_2 * m * 2^m + c_3 * 1^m$$

$$T(n) = c_1 * 2^{\log_2 n} + c_2 * \log_2 n * 2^{\log_2 n} + c_3 * 1^{\log_2 n}$$

$$T(n) = c_1 * n + c_2 * \log_2 n * n + c_3$$

Lo cual lleva a una eficiencia de **O(n*log$_2$n)**