Relación 3

Ejercicio 1

- Candidatos: Programas(long)
- Solución: S = (p_1, p_2, …, p_k) siendo pi un programa
- Función factible: Devuelve true hasta que no caben más elementos en la cinta
- Función de selección: Coger el programa de menor longitud en la cinta

```
// Habría que ordenar previamente el vector P por menor longitud
// Devuelve en S los índices de los programas que deben ir en la cinta
void Cinta(const vector<int> &P, int Long, vector<int> &S) {
  int n = P.size(),
      x;            // Elemento seleccionado de P
  bool noLlena = true;

  for (int i=0; i<n && noLlena; i++) {
    x = P[i];

    if (Long-x >= 0) {
      S.push_back(x);
      Long -= x;
    }
    else {
      noLlena = false;
    }
  }
}
```

Eficiencia

Teniendo el vector de programas ordenado por menor longitud, la eficiencia de la función es **O(n)** en el peor caso. Este se daría cuanto caben todos los programas en la cinta menos el último (por el propio enunciado del ejercicio no se pueden añadir todos).

Ejercicio 2

- Candidatos: Programa (long)
- Solución: S = (P_1, P_2, ..., P_k) siendo Pi una tarea
- Función factible: Siempre devuelve true.
- Función de selección: Elegir el programa restante de menor longitud.

Ordenando los programas de menor a mayor, obtenemos la menor espera para poder leer un programa (en media)

```
// Habría que ordenar previamente el vector P por menor longitud
// Devuelve en S los índices de los programas que deben ir en la cinta
void Cinta(const vector<int> &P, vector<int> &S) {
  int n = P.size(),
      x;            // Elemento seleccionado de P

  for (int i=0; i<n; i++) {
    S.push_back(P[i]);
  }
}
```

Eficiencia

Sin contar la ordenación, este algoritmo es de eficiencia **O(n)**.

Ejercicio 3

- Candidatos: Tareas (t, p)
- Solución: S = (T_1, T_2, ..., T_k) siendo Ti una tarea
- Función factible: Devuelve true mientras las tareas vayan cumpliendo con su tiempo de terminación.
- Función de selección: Coger la tarea con menor t_i

Como las tareas no son interrumpibles la mejor opción es intentar terminar la tarea con el tiempo de terminación más pequeño.

```
struct Tarea {
  int t, p;
}

// Devuelve en S el orden de Tareas seleccionadas, si tiene solución coincidirá
// con el de T (tras ser ordenado)
// @return Si existe solución o no

// Habría que ordenar previamente el vector T por menor ti
bool Planificacion(const vector<Tarea> &T, vector<Tarea> &S) {
  int n = T.size(), terminacionAnterior,
      x;    // Elemento seleccionado de T

  for (int i=0; i<n; i++) {
    x = T[i];

    if (terminacionAnterior <= x.t) {
      S.push_back(x);
      terminacionAnterior = x.t;
    }
    else {
      return false;
    }
  }

  return true;
}
```

Realmente la función solo indica si el problema tiene solución, puesto que, en el caso de que la tenga solo insertará los elementos ordenados de **T** en **S**.

Eficiencia

Al tener que recorrer el vector **T**, en todos los casos con solución la eficiencia es de **O(n)**.

Ejercicio 4

- Candidatos: Programas (reg)
- Solución: S = (P_1, P_2, ..., P_k) siendo P_i una cinta.
- Función factible: Siempre devuelve true
- Función de selección: La menor de las adyacentes

Los programas están ordenados de menor a mayor, pero con la condición de que para todo (x_1, ..., x_n) el programa x_i tiene que ser adyacente a x_{i-1} y x_{i+1}

El problema sería similar a la multiplicación de matrices, pues queremos minimizar el número de movimientos sin que al final se desordene el vector. Por lo que consistiría en "poner paréntesis" a la mezcla de los registros.

```
// Las cintas están ordenadas según el criterio de la función Ordenacion
int Mezcla(const vector<int> &C, vector<int> &S) {
  int n = C.size(),
      movimientos = 0;  // Número de movimientos que realiza

  for (int i=0; i<n; i++) {
    S.push_back(C[i])
    movimientos += movimientos + C[i];  // La mezcla recorre otra vez lo que tenía
                                        // más el nuevo registro.
  }
}
```

```
// Aquí el vector entra ordenado de la forma que queremos que entre en la cinta
// Se devuelve otro vector según el criterio de ordenación
vector<int> Ordenación (vector<int> &C) {
  vector<int> resultado;

  int x = menorElemento(C); // Devuelve el elemento con menos registros
  resultado.push_back(x)

  int n = C.size(), sig;

  while (n > 0) {
    if (x+1 < n && x-1 >= 0) // Si está en el rango del vector
      sig = min(C[x+1]), C[x-1]);
    else if (x+1 < n)
      sig = C[x+1];
    else if (x-1 >= 0)
      sig = C[x-1];

    resultado.push_back(sig);
    C.erase(x);
    x = sig;
    n = C.size();
  }
}
```

La función Ordenación añade el menor de los elementos y continúa eligiendo entro el más pequeños de los adyacentes.

Eficiencia

Por la regla de la suma, la eficiencia del algoritmo es **O(n)**.

Ejercicio 5

- Candidatos: Gasolineras (kilómetro en la carretera)
- Solución: S = (G_1, G_2, ..., G_k) siendo G_i una gasolinera.
- Función factible: Puesto que se puede llegar de una gasolinera a otra con el depósito lleno, siempre devuelve true
- Función de selección: La gasolinera más lejana a la que podemos llegar con lo que queda de depósito

```
// Devuelve el número de paradas, en S los índices de las gasolineras donde
// debe repostar

// @param B kilómetro del punto de destino
// @param X kilómetros que se pueden hacer con el depósito lleno
int Trayecto(const vector<int> &G, int X, int B, vector<int> &S) {
  int kmActual = 0,
      paradas  = 0,
      i        = 0,
      x;    // Elemento seleccionado de G

  bool fin = false;   // True cuando llegamos a B


  while (!fin) {
    x = Seleccion(G, i, X, kmActual);

    S.push_back(x);
    paradas++;
    kmActual = G[x];

    if (G[x] >= B)
      fin = true;
    else
      i = x;
  }

  return paradas;
}
```

```cpp
// Calculamos la distancia hasta la siguiente gasolinera, si es menor que X,
// miramos la siguiente. En caso contrario seleccionamos esa gasolinera o la
// anterior, según se pueda llegar hasta a ella o no
int Seleccion (const vector<int> &G, int indice, int X, int kmActual){

  while (X > 0) {
    X -= G[indice] - kmActual;
    kmActual = G[indice];
    indice++;
  }

  // Puesto que la distancia a la primera gasolinera es menor que X
  // siempre devuelve >= 0
  if (X < 0)
    return indice-1;
  else
    return indice;
}
```

Esta **X** indica el tamaño del depósito

Eficiencia

Como mucho se valorarán todas las gasolineras, por tanto, la eficiencia es de **O(n)** en el peor caso.

Ejercicio 6

- Candidatos: [Clases(d, hi, hf), aula]
- Solución: S = (A_1, A_2, ..., A_k) siendo A_i el aula asignada a la clase Ci.
- Función factible: Siempre devuelve true.
- Función de selección: La clase siguiente (no importa el orden)

Se intenta asignar un aula a todas las clases posibles, luego se repite hasta que todas las clases tengan un aula.

```
void Aulas (vector<Clase> &C, vector<int> &S){
  int n = C.size,
      aula = 0;     // Aula actual a asignar

  vector<bool> asignadas(n);  // Inicialmente a false

  while (hayClasesSinAsignar(asignadas) {
    aula++;

    while (hayClasesSinAsignar(asignadas)) {
      x = seleccionaClase(C, asignadas, x);

      if (sePuedeAsignar(x, asignadas, aula) {
        asignadas[x] = true;
        S[x] = aula;
      }
    }
  }
}
```

Eficiencia

En el peor caso todas las clases tienen el mismo horario, y por tanto se le deben asignar **n** aulas. La eficiencia del algoritmo (sin contar la función hayClasesSinAsignar y sePuedeAsignar) es de **O(n^2)**.

Ejercicio 7

- Candidatos: [Nodos del grafo G, Color]
- Solución: S = (C_1, C_2, ..., C_k) siendo C_i el color asignado al nodo N_i.
- Función factible: Siempre devuelve true.
- Función de selección: El nodo siguiente (no importa el orden).

Similarmente al ejercicio anterior, se asigna un color a todos los nodos posibles. Se repite hasta que todos los nodos estén coloreados

```cpp
void Colorear (Grafo &G, vector<int> &S) {
  int n = G.nodos(),    // Nodos del grafo
      color = 0,    // Color actual
      x;      // Nodo a comprobar

  vector<bool> colores;   // Indica si el nodo i está coloreado
                          // Inicialmente a false para todos

  while (hayNodosSinColorear(colores)) {
    color++;

    while (hayNodosSinColorear(colores)) {
      x = siguienteNodoSinColorear(x, colores); // Recorre el vector buscando
                                                // un índice superior a x no
                                                // coloreado

      if (sePuedeColorear(G, x, color) { // Recorre el grafo viendo si alguno
                                         // de sus nodos adyacentes tiene el
                                         // mismo color
        S[x] = color;
        colores[x] = true;
      }
    }
  }
}
```

Eficiencia

Al igual que en el ejercicio 6, la eficiencia es de **O(n^2)** cuando el grafo es completo, es decir, se necesitan **n** colores diferentes para colorearlo. (No teniendo en cuenta la eficiencia de hayNodosSinColorear ni sePuedeColorear)