Algoritmos - Asier Llano - Final

July 28, 2023

1 Algoritmos de optimización - Seminario

Nombre y Apellidos: Asier Llano

Url: [https://github.com/asierllano/03MIAR—Algoritmos-de-Optimizacion—2023]

Problema: Organizar los horarios de partidos de La Liga

Descripción del problema:

- Desde la La Liga de fútbol profesional se pretende organizar los horarios de los partidos de liga de cada jornada. Se conocen algunos datos que nos deben llevar a diseñar un algoritmo que realice la asignación de los partidos a los horarios de forma que maximice la audiencia.
- Los horarios disponibles se conocen a priori y son los siguientes: Viernes 20h; Sábado 12h,16h,18h,20h; Domingo 12h,16h,18h,20h; Lunes 20h
- En primer lugar se clasifican los equipos en tres categorías según el numero de seguidores(que tiene relación directa con la audiencia). Hay 3 equipos en la categoría A, 11 equipos de categoría B y 6 equipos de categoría C.
- Se conoce estadísticamente la audiencia que genera cada partido según los equipos que se enfrentan y en horario de sábado a las 20h (el mejor en todos los casos). A-A: 2 Millones, A-B: 1.3 Millones, B-B: 0.9 Millones, A-C: 1 Millones, B-C: 0.75 Millones, C-C: 0.47 Millones
- Si el horario del partido no se realiza a las 20 horas del sábado se sabe que se reduce según los coeficientes de la siguiente tabla. Viernes 20h (0.4); Sábado 12h (0.55), 16h (0.7), 18h (0.8), 20h (1); Domingo 12h (0.45), 16h (0.75), 18h (0.85), 20h (1); Lunes 20h (0.4)
- Debemos asignar obligatoriamente siempre un partido el viernes y un partido el lunes
- Es posible la coincidencia de horarios pero en este caso la audiencia de cada partido se verá afectada y se estima que se reduce en porcentaje según la siguiente tabla dependiendo del número de coincidencias: 0 (0%), 1 (25%), 2 (45%), 3 (60%), 4 (70%), 5 (75%), 6 (78%), 7 (80%), 8 (80%)

....

(*) La respuesta es obligatoria

(*) **¿Cuantas posibilidades hay sin tener en cuenta las restricciones?**

¿Cuantas posibilidades hay teniendo en cuenta todas las restricciones.

Respuesta

Analizamos tanto el espacio de enunciados, como el espacio de soluciones para entender la magnitud del problema completa:

- **Espacio de enunciados**: Consideramos el espacio de enunciados como las posibilidades de combinaciones de partidos que tiene una jornada. Según mis cálculos el número de jornadas totales son en total $9.38 \cdot 10^{12}$ pero tienen muchísimas simetrías, teniendo en cuenta que todos los equipos se clasifican en 'A', 'B' y 'C' y que todos los partidos en general se pueden clasificar en tan solo 6 categorías diferentes en base a la categoría de los equipos 'AA', 'AB', 'AC', 'BC', 'BB' y 'CC', y que su orden es irrelevante para la audiencia... el espacio de enunciados se reduce enormemente. El número total de enunciados normalizados a las 6 categorías de partidos, ordenados alfabéticamente, son solo 19. Todas las jornadas posibles se pueden reducir a una de las 19 posibles jornadas normalizadas. Se pueden generar todas ellos con el siguiente programa:

[ ]:
```
def genera_enunciados(a=3, b=11 ,c=6):
    if a >= 1:
        if a >= 2:
            prefix = ['AA']
            for i in genera_enunciados(a-2,b,c):
                yield prefix + i
        if b >= 1:
            prefix = ['AB']
            for i in genera_enunciados(a-1,b-1,c):
                yield prefix + i
        if c >= 1:
            prefix = ['AC']
            for i in genera_enunciados(a-1,b,c-1):
                yield prefix + i
    elif b >= 1:
        if b >= 2:
            prefix = ['BB']
            for i in genera_enunciados(a,b-2,c):
                yield prefix + i
        if c >= 1:
            prefix = ['BC']
            for i in genera_enunciados(a,b-1,c-1):
                yield prefix + i
    elif c >= 1:
        assert(c >= 2)
        prefix = ['CC']
        for i in genera_enunciados(a,b,c-2):
            yield prefix + i
    else:
        yield []

enunciados=set()
for e in genera_enunciados():
    e.sort()
    enunciados.add(tuple(e))
enunciados=list(enunciados)
```

```
enunciados.sort()

print(f"Total de enunciados: {len(enunciados)}")
for e in enunciados:
    print(f" * {e}")
```

```
Total de enunciados: 19
 * ('AA', 'AB', 'BB', 'BB', 'BB', 'BB', 'BB', 'CC', 'CC', 'CC')
 * ('AA', 'AB', 'BB', 'BB', 'BB', 'BB', 'BC', 'BC', 'CC', 'CC')
 * ('AA', 'AB', 'BB', 'BB', 'BB', 'BC', 'BC', 'BC', 'BC', 'CC')
 * ('AA', 'AB', 'BB', 'BB', 'BC', 'BC', 'BC', 'BC', 'BC', 'BC')
 * ('AA', 'AC', 'BB', 'BB', 'BB', 'BB', 'BB', 'BC', 'CC', 'CC')
 * ('AA', 'AC', 'BB', 'BB', 'BB', 'BB', 'BC', 'BC', 'BC', 'CC')
 * ('AA', 'AC', 'BB', 'BB', 'BB', 'BC', 'BC', 'BC', 'BC', 'BC')
 * ('AB', 'AB', 'AB', 'BB', 'BB', 'BB', 'BB', 'CC', 'CC', 'CC')
 * ('AB', 'AB', 'AB', 'BB', 'BB', 'BB', 'BC', 'BC', 'CC', 'CC')
 * ('AB', 'AB', 'AB', 'BB', 'BB', 'BC', 'BC', 'BC', 'BC', 'CC')
 * ('AB', 'AB', 'AB', 'BB', 'BC', 'BC', 'BC', 'BC', 'BC', 'BC')
 * ('AB', 'AB', 'AC', 'BB', 'BB', 'BB', 'BB', 'BC', 'CC', 'CC')
 * ('AB', 'AB', 'AC', 'BB', 'BB', 'BB', 'BC', 'BC', 'BC', 'CC')
 * ('AB', 'AB', 'AC', 'BB', 'BB', 'BC', 'BC', 'BC', 'BC', 'BC')
 * ('AB', 'AC', 'AC', 'BB', 'BB', 'BB', 'BB', 'BB', 'CC', 'CC')
 * ('AB', 'AC', 'AC', 'BB', 'BB', 'BB', 'BB', 'BC', 'BC', 'CC')
 * ('AB', 'AC', 'AC', 'BB', 'BB', 'BB', 'BC', 'BC', 'BC', 'BC')
 * ('AC', 'AC', 'AC', 'BB', 'BB', 'BB', 'BB', 'BB', 'BC', 'CC')
 * ('AC', 'AC', 'AC', 'BB', 'BB', 'BB', 'BB', 'BC', 'BC', 'BC')
```

- **Espacio de soluciones**: El espacio de soluciones parece a priori también enorme, teniendo en cuenta que son 10 partidos y 10 slots diferentes de tiempo, podría ser de 10^{10}. Esto es falso, en realidad es muchísimo más reducido, porque es un problema con elementos repetitivos. Del mismo modos que con los enunciados tienen muchísimas simetrías, no tiene sentido probar permutaciones diferentes del mismo tipo de partidos. Además hay que tener en cuenta que los lunes y viernes hay un único partido. Todo esto hace que el número de combinaciones también se vea muy reducido. El siguiente programa calcula el número de combinaciones totales por tipo de jornada.

[ ]:
```
import collections
import math

def binomio(a,b):
    return math.factorial(a)//(math.factorial(b)*math.factorial(a-b))
def combinaciones_con_repeticion(slots, rep):
    return binomio(slots+rep-1, rep)

def contar_soluciones(enunciado):
    def contar_soluciones_eliminar(ec, key):
        ec = dict(ec)
        if ec[key] == 1:
```

```
            del ec[key]
        else:
            ec[key] -= 1
        return ec
    # Contamos los tipos de jornadas
    ec = dict(collections.Counter(e))
    # Ponemos un tipo de partido el viernes
    soluciones = 0
    for key_v in ec.keys():
        # Partidos a distribuir de sábado a lunes
        ec_sl = contar_soluciones_eliminar(ec, key_v)
        # Ponemos un tipo de partido el lunes
        for key_l in ec_sl.keys():
            # Partidos a distribuir de sábado a domingo
            ec_sd = contar_soluciones_eliminar(ec_sl, key_l)
            # Ahora es una solución de combutaciones con repetición y reemplazo
            soluciones2 = 1
            for key_sd,rep in ec_sd.items():
                # Hay 8 posibles slots para colocar rep repeticiones de este
                # tipo de partidos
                # Es un problema de combinaciones con repeticion (que se
                # reduce por simetría mucho)
                soluciones2 *= combinaciones_con_repeticion(8, rep)
            soluciones += soluciones2
    return soluciones

def enunciado_a_cadena(enunciado):
    resultado = ''
    for e in enunciado:
        if len(resultado):
            resultado += ' '
        resultado += e
    return resultado

print("Soluciones posibles por tipo de jornada:")
num_soluciones={}
peor_enunciado = None
peor_soluciones = 0
for e in enunciados:
    soluciones = contar_soluciones(e)
    num_soluciones[e] = soluciones
    print(f" * {enunciado_a_cadena(e)} con {soluciones:8d} soluciones")
    if soluciones > peor_soluciones:
        peor_enunciado = e
        peor_soluciones = soluciones

print("Tipo de jornada con el máximo número de soluciones posibles:")
```

```
print(f"  {enunciado_a_cadena(peor_enunciado)}")
print(f"  {peor_soluciones} soluciones posibles")
```

```
Soluciones posibles por tipo de jornada:
 * AA AB BB BB BB BB BB CC CC CC con  5217408 soluciones
 * AA AB BB BB BB BB BC BC CC CC con 27971904 soluciones
 * AA AB BB BB BB BC BC BC BC CC con 19979520 soluciones
 * AA AB BB BB BC BC BC BC BC BC con  3156384 soluciones
 * AA AC BB BB BB BB BB BC CC CC con 13684224 soluciones
 * AA AC BB BB BB BB BC BC BC CC con 19979520 soluciones
 * AA AC BB BB BB BC BC BC BC BC con  5217408 soluciones
 * AB AB AB BB BB BB BB CC CC CC con  4080960 soluciones
 * AB AB AB BB BB BB BC BC CC CC con 18680832 soluciones
 * AB AB AB BB BB BC BC BC BC CC con 10699200 soluciones
 * AB AB AB BB BC BC BC BC BC BC con  1196448 soluciones
 * AB AB AC BB BB BB BB BC CC CC con 27971904 soluciones
 * AB AB AC BB BB BB BC BC BC CC con 34841088 soluciones
 * AB AB AC BB BB BC BC BC BC BC con  7320384 soluciones
 * AB AC AC BB BB BB BB BB CC CC con  7320384 soluciones
 * AB AC AC BB BB BB BB BC BC CC con 27971904 soluciones
 * AB AC AC BB BB BB BC BC BC BC con 10699200 soluciones
 * AC AC AC BB BB BB BB BB BC CC con  5217408 soluciones
 * AC AC AC BB BB BB BB BC BC BC con  4080960 soluciones
Tipo de jornada con el máximo número de soluciones posibles:
  AB AB AC BB BB BB BC BC BC CC
  34841088 soluciones posibles
```

Se descubre que el enunciado más difícil de todos ellos tiene solo 34 millones de posibilidades. En este salen las máximas combinaciones porque tiene 5 tipos de partidos diferentes y el número de repeticiones de tipo de partido es más bajo que en el resto, de ahí que el número de soluciones con la misma audiencia garantizada por simetrías del problema se reduce menos que en el resto.

El número es suficientemente bajo como para que se pueda calcular por fuerza bruta en un tiempo muy razonable.

Tiene pinta que podemos llegar con garantías a la solución óptima de las 19 jornadas.

Realmente el número de soluciones posibles son aún muchas menos, porque el slot de S20h y D20h tienen el mismo factor de corrección, así como el slot de V20h y L20h, por lo que estos slots son intercambiables siempre. Esta simetría no se ha explotado en los números arriba indicados.

Modelo para el espacio de soluciones

(*) **¿Cual es la estructura de datos que mejor se adapta al problema? Argumentalo.(Es posible que hayas elegido una al principio y veas la necesidad de cambiar, arguméntalo)**

Respuesta

Inicialmente la estructura que se ha pensado para los siguientes conceptos es la siguiente:

- **Partido:** Una cadena del tipo “AA”, “AB”, “BB”, “AC”, “BC”, “CC”
- **Problema:** Una tupla de partidos

- **Slot:** Una cadena de la forma “V20”, “S12”, “S16”, “S18”, “S20”, “D12”, “D16”, “D18”, “D20”, “L20”
- **Solución:** Un diccionario con clave slot, y con el valor siendo la lista de partidos en dicho slot.
- **Tablas de audiencias:** La tabla de audiencias por tipo de partido es un diccionario cuya clave es el partido y cuyo contenido es un número real con la audiencia en millones. La tabla de factores de audiencia por slot, es un diccionario cuya clave es el slot y cuyo contenido es el número real factor de audiencia. La tabla de factores de audiencia por simultaneidad se plantea como una lista en el que en el índice que coincide con el número de partidos simultáneos se coloca el factor de simultaneidad.

Después de esto, me he dado cuenta de que se puede optimizar en velocidad para la parte interna del algoritmo. Se han conservado las interfaces del algoritmo arriba propuestas, pero internamente se operan con tipos más sencillos y más rápido, dando prioridad a los números enteros en vez de a cadenas para representar valores discretos de elementos y dando prioridad a las listas frente a los diccionarios para buscar elementos:

- **Partido interno:** Un entero que valdrá 0 para partidos AA, 1 para AB, 2 para BB, 3 para AC, 4 para BC y 5 para CC.
- **Problema interno:** Una tupla de partidos internos.
- **Slot interno:** Un entero que representa el slot: 0 para V20, 1 para S12, 2 para S16, 3 para S18, 4 para S20, 5 para D12, 6 para D16, 7 para D18, 8 para D20, 9 para L20
- **Solucion intermedia:** Lista que para el índice de un slot de partido devuelve una lista de partidos internos presentes en ese slot.
- **Tablas de audiencias internas:** En vez de diccionarios son listas.

De esta forma se ofrece una interfaz amigable como la anterior propueta, pero internamente se trabajan con estructuras más eficientes de programación que funcionan un poco más rápido. No obstante, el incremento de velocidad por el tipo de dato va a ser más modesto, que por la reformulación del algoritmo en sí mismo, tal y como se verá en próximas secciones.

Según el modelo para el espacio de soluciones

(*) **¿Cual es la función objetivo?**

(*) **¿Es un problema de maximización o minimización?**

Respuesta

La función objetivo es la audiencia total en base a las reglas dadas. Es un problema de maximización.

Diseña un algoritmo para resolver el problema por fuerza bruta

Respuesta

El truco de poder llegar a hacer la solución por fuerza bruta es utilizar todas las simetrías para no calcular todas las combinaciones de partidos posibles de la liga, sino todas las combinaciones posibles que den audiencias diferentes. Esto se debe a que muchos de los partidos son intercambiables entre sí y dado que son del mismo tipo se obtiene la misma audiencia. Trabajando correctamente con combinatoria el espacio de soluciones no es demasiado amplio y realizable perfectamente por fuerza bruta. Esto permite calcular la solución a un problema en un tiempo medio de unos 37s y en todo el espacio de soluciones en 11.7 minutos en un único core de un ordenador portátil convencional.

[ ]:
```
%%time

# Crea un objeto de combinaciones exactas de n elementos en 8 slots que
# vamos a utilizar todo el rato
comb8=[]
for rep in range(7):
    pos = [0 for _ in range(rep)]
    posend = [7 for _ in range(rep)]
    comb8rep = []
    while True:
        comb8rep.append(tuple(pos))
        if pos == posend:
            break
        for j in range(rep):
            pos[j] += 1
            if pos[j] < 8:
                break
        for j in reversed(range(rep)):
            if pos[j] == 8:
                pos[j] = pos[j+1]
    assert(len(comb8rep) == combinaciones_con_repeticion(8,rep))
    comb8.append(comb8rep)

# Función que obtiene la audiencia de una solucion
audiencia_partido = { 'AA': 2, 'AB': 1.3, 'BB': 0.9, \
                      'AC': 1, 'BC': 0.75, 'CC': 0.47 }
audiencia_slot = { 'V20': 0.4, 'S12': 0.55, 'S16': 0.7, 'S18': 0.8, \
    'S20': 1, 'D12': 0.45, 'D16': 0.75, 'D18': 0.85, 'D20': 1, 'L20': 0.4 }
audiencia_simultaneidad = [ 1, 1, 0.75, 0.55, 0.4, 0.3, 0.25, 0.22, 0.2, 0.2 ]
def audiencia(solucion):
    resultado = 0
    for slot,partidos in solucion.items():
        resultado_slot = 0
        for partido in partidos:
            resultado_slot += audiencia_partido[partido]
        resultado_slot *= audiencia_slot[slot]
        resultado_slot *= audiencia_simultaneidad[len(partidos)]
        resultado += resultado_slot
    return resultado

# Realiza la solución por fuerza bruta
def soluciones_fuerza_bruta(enunciado):
    # Elimina una entrada del enunciado
    def enunciado_eliminar(ec, key):
        ec = dict(ec)
        if ec[key] == 1:
            del ec[key]
```

```
        else:
            ec[key] -= 1
        return ec    # Contamos los tipos de jornadas
    # Contamos los tipos de jornadas
    ec = dict(collections.Counter(enunciado))
    # Ponemos un tipo de partido el viernes
    for key_v in ec.keys():
        # Partidos a distribuir de sábado a lunes
        ec_sl = enunciado_eliminar(ec, key_v)
        # Ponemos un tipo de partido el lunes
        for key_l in ec_sl.keys():
            # Partidos a distribuir entre sábado y domingo
            ec_sd = enunciado_eliminar(ec_sl, key_l)
            # Prepara las combinaciones
            keys_sd = []
            combs_sd = []
            for key_sd, rep in ec_sd.items():
                keys_sd.append(key_sd)
                combs_sd.append(comb8[rep])
            combinacion = [0 for _ in keys_sd]
            # Reporta cada una de las combinaciones
            while True:
                # Crea la combinacion
                slots=[[] for _ in range(8)]
                for i in range(len(keys_sd)):
                    key = keys_sd[i]
                    for s in combs_sd[i][combinacion[i]]:
                        slots[s].append(key)
                # Distribuimos en todas las combinaciones en los 8 slots
                # diferentes
                yield {'V20': [key_v], 'S12': slots[0], 'S16': slots[1], \
                       'S18': slots[2], 'S20': slots[3], 'D12': slots[4], \
                       'D16': slots[5], 'D18': slots[6], 'D20': slots[7], \
                       'L20': [key_l]}
                # Pasa a la siguiente combinacion
                for i in range(len(keys_sd)):
                    combinacion[i] += 1
                    if combinacion[i] < len(combs_sd[i]):
                        break
                    combinacion[i] = 0
                else:
                    break

# Valora cada solución de las soluciones de fuerza bruta y obtiene la mejor
# Además comprobamos que el número de soluciones generado coincide con el
# predicho
def solucion_fuerza_bruta(enunciado):
```

```
    mejor_audiencia = 0
    mejores_soluciones = None
    n = 0
    for s in soluciones_fuerza_bruta(enunciado):
        a = audiencia(s)
        if math.isclose(a, mejor_audiencia):
            mejores_soluciones.append(s)
        elif a > mejor_audiencia:
            mejor_audiencia = a
            mejores_soluciones = [s]
        n += 1
    assert(n == num_soluciones[enunciado])
    return mejores_soluciones

# Convierte la solucion en una cadena
def solucion_a_cadena(solucion):
    resultado = ''
    for key, partidos in solucion.items():
        if len(resultado):
            resultado += ' '
        resultado += key
        resultado += ':'
        if len(partidos):
            primero = True
            for partido in partidos:
                if not primero:
                    resultado += ','
                resultado += partido
                primero = False
        else:
            resultado += '-'
    return resultado

print("Soluciones mejores obtenidas por fuerza bruta:")
for e in enunciados:
    soluciones = solucion_fuerza_bruta(e)
    print(f"* {enunciado_a_cadena(e)}: {audiencia(soluciones[0]):.4f} Millones")
    for s in soluciones:
        print(f"  - {solucion_a_cadena(s)}")
        assert(math.isclose(audiencia(s), audiencia(soluciones[0])))
```

```
Soluciones mejores obtenidas por fuerza bruta:
* AA AB BB BB BB BB BB CC CC CC: 7.1725 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:CC D16:BB D18:BB D20:AA L20:CC
  - V20:CC S12:BB S16:BB S18:BB S20:AA D12:CC D16:BB D18:BB D20:AB L20:CC
* AA AB BB BB BB BB BC BC CC CC: 7.2160 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:CC
```

```
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:CC
* AA AB BB BB BB BC BC BC BC CC: 7.2230 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:CC
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:CC
  - V20:CC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:BC
  - V20:CC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:BC
* AA AB BB BB BC BC BC BC BC BC: 7.2225 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BC D18:BB D20:AA L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BC D18:BB D20:AB L20:BC
* AA AC BB BB BB BB BB BC CC CC: 7.0135 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:BB,BB D12:- D16:BB D18:AC D20:AA L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:- D16:BB D18:AC D20:BB,BB L20:CC
* AA AC BB BB BB BB BC BC BC CC: 7.0280 Millones
  - V20:BC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:CC
  - V20:BC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:BC
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:BC
* AA AC BB BB BB BC BC BC BC BC: 7.0350 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:BC
* AB AB AB BB BB BB BB CC CC CC: 6.8125 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:CC D16:BB D18:AB D20:AB L20:CC
* AB AB AB BB BB BB BC BC CC CC: 6.8560 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:CC
* AB AB AB BB BB BC BC BC BC CC: 6.8630 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:CC
  - V20:CC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:BC
* AB AB AB BB BC BC BC BC BC BC: 6.8625 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BC D18:AB D20:AB L20:BC
* AB AB AC BB BB BB BB BC CC CC: 6.6835 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:CC
* AB AB AC BB BB BB BC BC BC CC: 6.7130 Millones
  - V20:BC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:BC
* AB AB AC BB BB BC BC BC BC BC: 6.7200 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:BC
* AB AC AC BB BB BB BB BB CC CC: 6.4760 Millones
  - V20:CC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:CC
  - V20:CC S12:BB S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:CC
* AB AC AC BB BB BB BB BC BC CC: 6.5055 Millones
  - V20:BC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:CC
  - V20:BC S12:BC S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:CC
  - V20:CC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:BC
  - V20:CC S12:BC S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:BC
* AB AC AC BB BB BB BC BC BC BC: 6.5250 Millones
  - V20:BC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:AC D20:AB L20:BC
  - V20:BC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AC L20:BC
* AC AC AC BB BB BB BB BB BC CC: 6.2880 Millones
```

```
  - V20:BC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:CC
  - V20:BC S12:BB S16:BB S18:AC S20:AC D12:- D16:BB D18:AC D20:BB,BB L20:CC
  - V20:CC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:BC
  - V20:CC S12:BB S16:BB S18:AC S20:AC D12:- D16:BB D18:AC D20:BB,BB L20:BC
* AC AC AC BB BB BB BB BC BC BC: 6.3175 Millones
  - V20:BC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:BC
  - V20:BC S12:BC S16:BB S18:AC S20:AC D12:- D16:BB D18:AC D20:BB,BB L20:BC
CPU times: user 11min 43s, sys: 3 µs, total: 11min 43s
Wall time: 11min 43s
```

Aún así, el algoritmo es optimizable aplicando mejoras en las técnicas de programación, todavía en fuerza bruta. Se han aplicado las siguientes mejoras:

- Utilizar índices para los tipos de partidos en vez de diccionarios para todos los calculos intermedios.
- Generación recursiva de las soluciones aprovecha cálculos intermedios.
- Cálculo de la audiencia a la vez que se genera la jornada para reutilizar el cálculo de audiencia.
- Reducción de las simetrías de L20 y V20, así como de S20 y D20.

La nueva solución por fuerza bruta acelerada tarda 2 minutos y 18 segundos en obtener la solución óptima de todo el espacio de problemas (unos 7.3 segundos por cada enunciado). Esto ha sido ejecutado en un único core de un portátil convencional. Sigue siendo de fuerza bruta, desde el punto de vista que itera sobre todas las posibles soluciones de audiencia de valor diferente para encontrar las óptimas.

[ ]:
```
%%time

# Convierte un partido a indice
partido_a_indice = {'AA': 0, 'AB': 1, 'BB': 2, 'AC': 3, 'BC': 4, 'CC': 5}
indice_a_partido = ['AA', 'AB', 'BB', 'AC', 'BC', 'CC']
slot_a_indice = { 'V20': 0, 'S12': 1, 'S16': 2, 'S18': 3, 'S20': 4, \
                  'D12': 5, 'D16': 6, 'D18': 7, 'D20': 8, 'L20': 9 }
indice_a_slot = [ 'V20', 'S12', 'S16', 'S18', 'S20', \
                  'D12', 'D16', 'D18', 'D20', 'L20' ]
audiencia_indice_partido = [ 2, 1.3, 0.9, 1, 0.75, 0.47 ]
audiencia_indice_slot = [ 0.4, 0.55, 0.7, 0.8, 1, 0.45, 0.75, 0.85, 1, 0.4 ]

# Función que convierte un enunciado a indices
def enunciado_a_indices(enunciado):
    return tuple(partido_a_indice[e] for e in enunciado)

# Función para convertir los indices a soluciones
def indices_a_solucion(solucion_indice):
    resultado = {}
    for i in range(len(solucion_indice)):
        resultado[indice_a_slot[i]] = \
            [indice_a_partido[p] for p in solucion_indice[i]]
    return resultado
def indices_a_soluciones(solucion_indice):
```

```
    resultado = indices_a_solucion(solucion_indice)
    reslist = [resultado]
    if resultado['V20'] != resultado['L20']:
        resultado2 = resultado.copy()
        resultado2['L20'] = resultado['V20']
        resultado2['V20'] = resultado['L20']
        reslist.append(resultado2)
    if resultado['S20'] != resultado['D20']:
        resultado2 = resultado.copy()
        resultado2['S20'] = resultado['D20']
        resultado2['D20'] = resultado['S20']
        reslist.append(resultado2)
    if resultado['S20'] != resultado['D20'] and \
       resultado['V20'] != resultado['L20']:
        resultado2 = resultado.copy()
        resultado2['L20'] = resultado['V20']
        resultado2['V20'] = resultado['L20']
        resultado2['S20'] = resultado['D20']
        resultado2['D20'] = resultado['S20']
        reslist.append(resultado2)
    return reslist

# Obtiene el número de soluciones representados por una solución en índices
# canónica
def indices_a_soluciones_num(solucion_indice):
    soluciones = 1
    if solucion_indice[0] != solucion_indice[-1]:
        soluciones *= 2
    if solucion_indice[4] != solucion_indice[8]:
        soluciones *= 2
    return soluciones

# Realiza la solución por fuerza bruta
def soluciones_fuerza_bruta_indice(enunciado):
    def recursiva(slot,audiencia,keys,numbers,index=0,pos=1):
        # Go for the next type of data
        if numbers[index] == 0:
            index+=1
            pos = 1
            if index == len(numbers):
                # Aceleración porque el factor de S20 y D20 es el mismo
                if slot[4] >= slot[8]:
                    a = 0
                    for i in range(len(audiencia)):
                        a += audiencia[i]*audiencia_simultaneidad[len(slot[i])]
                    yield [slot, a]
                return
```

```
        # Place the next element
        partido = keys[index]
        partido_audiencia = audiencia_indice_partido[partido]
        for p in range(pos, 9):
            slot[p].append(partido)
            prev_audiencia = audiencia[p]
            audiencia[p] += partido_audiencia * audiencia_indice_slot[p]
            numbers[index] -= 1
            yield from recursiva(slot, audiencia, keys, numbers, index, p)
            del slot[p][-1]
            numbers[index] += 1
            audiencia[p] = prev_audiencia
    # Convierte el enunciado a su versión indixada
    enunciado = enunciado_a_indices(enunciado)
    # Elimina una entrada del enunciado
    def enunciado_eliminar(ec, key):
        ec = dict(ec)
        if ec[key] == 1:
            del ec[key]
        else:
            ec[key] -= 1
        return ec    # Contamos los tipos de jornadas
    # Contamos los tipos de jornadas
    ec = dict(collections.Counter(enunciado))
    # Ponemos un tipo de partido el viernes
    for key_v in ec.keys():
        # Partidos a distribuir de sábado a lunes
        ec_sl = enunciado_eliminar(ec, key_v)
        # Ponemos un tipo de partido el lunes
        for key_l in ec_sl.keys():
            # Aceleración porque el L20 y V20 tienen el mismo factor
            if key_l < key_v:
                continue
            # Partidos a distribuir entre sábado y domingo
            slot = [[key_v],[],[],[],[],[],[],[],[],[key_l]]
            audiencia = \
                [audiencia_indice_partido[key_v]*audiencia_indice_slot[0],\
                 0,0,0,0,0,0,0,0,\
                 audiencia_indice_partido[key_l]*audiencia_indice_slot[9]]
            ec_sd = enunciado_eliminar(ec_sl, key_l)
            keys=list(ec_sd.keys())
            numbers=[ec_sd[k] for k in keys]
            yield from recursiva(slot, audiencia, keys, numbers)

# Valora cada solución de las soluciones de fuerza bruta y obtiene la mejor
# Además comprobamos que el número de soluciones generado coincide con el
# predicho (Utilizamos la versión indexada)
```

```python
def solucion_fuerza_bruta(enunciado):
    mejor_audiencia = 0
    mejores_soluciones = None
    n = 0
    for s,a in soluciones_fuerza_bruta_indice(enunciado):
        if math.isclose(a, mejor_audiencia):
            mejores_soluciones += indices_a_soluciones(s)
            #assert(math.isclose(mejor_audiencia,\
            #       audiencia(mejores_soluciones[-1])))
        elif a > mejor_audiencia:
            mejor_audiencia = a
            mejores_soluciones = indices_a_soluciones(s)
            #assert(math.isclose(mejor_audiencia,\
            #       audiencia(mejores_soluciones[-1])))
        n += indices_a_soluciones_num(s)
    assert(n == num_soluciones[enunciado])
    return mejores_soluciones

print("Soluciones mejores obtenidas por fuerza bruta optimizada:")
for e in enunciados:
    soluciones = solucion_fuerza_bruta(e)
    print(f"* {enunciado_a_cadena(e)}: {audiencia(soluciones[0]):.4f} Millones")
    for s in soluciones:
        print(f"  - {solucion_a_cadena(s)}")
        assert(math.isclose(audiencia(s), audiencia(soluciones[0])))
```

```
Soluciones mejores obtenidas por fuerza bruta optimizada:
* AA AB BB BB BB BB BB CC CC CC: 7.1725 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:CC D16:BB D18:BB D20:AA L20:CC
  - V20:CC S12:BB S16:BB S18:BB S20:AA D12:CC D16:BB D18:BB D20:AB L20:CC
* AA AB BB BB BB BB BC BC CC CC: 7.2160 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:CC
* AA AB BB BB BB BC BC BC BC CC: 7.2230 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:CC
  - V20:CC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:CC
  - V20:CC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:BC
* AA AB BB BB BC BC BC BC BC BC: 7.2225 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BC D18:BB D20:AA L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BC D18:BB D20:AB L20:BC
* AA AC BB BB BB BB BB BC CC CC: 7.0135 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:BB,BB D12:- D16:BB D18:AC D20:AA L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:- D16:BB D18:AC D20:BB,BB L20:CC
* AA AC BB BB BB BB BC BC BC CC: 7.0280 Millones
  - V20:BC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:BC
```

```
  - V20:BC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:BC
* AA AC BB BB BB BC BC BC BC BC: 7.0350 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:BC
* AB AB AB BB BB BB BB CC CC CC: 6.8125 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:CC D16:BB D18:AB D20:AB L20:CC
* AB AB AB BB BB BB BC BC CC CC: 6.8560 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:CC
* AB AB AB BB BB BC BC BC BC CC: 6.8630 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:CC
  - V20:CC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:BC
* AB AB AB BB BC BC BC BC BC BC: 6.8625 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BC D18:AB D20:AB L20:BC
* AB AB AC BB BB BB BB BC CC CC: 6.6835 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:CC
* AB AB AC BB BB BB BC BC BC CC: 6.7130 Millones
  - V20:BC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:BC
* AB AB AC BB BB BC BC BC BC BC: 6.7200 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:BC
* AB AC AC BB BB BB BB BB CC CC: 6.4760 Millones
  - V20:CC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:CC
  - V20:CC S12:BB S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:CC
* AB AC AC BB BB BB BB BC BC CC: 6.5055 Millones
  - V20:BC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:CC
  - V20:CC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:BC
  - V20:BC S12:BC S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:CC
  - V20:CC S12:BC S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:BC
* AB AC AC BB BB BB BC BC BC BC: 6.5250 Millones
  - V20:BC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:AC D20:AB L20:BC
  - V20:BC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AC L20:BC
* AC AC AC BB BB BB BB BB BC CC: 6.2880 Millones
  - V20:BC S12:BB S16:BB S18:AC S20:AC D12:- D16:BB D18:AC D20:BB,BB L20:CC
  - V20:CC S12:BB S16:BB S18:AC S20:AC D12:- D16:BB D18:AC D20:BB,BB L20:BC
  - V20:BC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:CC
  - V20:CC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:BC
* AC AC AC BB BB BB BB BC BC BC: 6.3175 Millones
  - V20:BC S12:BC S16:BB S18:AC S20:AC D12:- D16:BB D18:AC D20:BB,BB L20:BC
  - V20:BC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:BC
CPU times: user 2min 18s, sys: 0 ns, total: 2min 18s
Wall time: 2min 18s
```

Calcula la complejidad del algoritmo por fuerza bruta

Respuesta

No queda muy bien claro la magnitud que tiene a infinito en el cálculo de complejidad de big O. Haremos diferentes suposiciones. Para este análisis supondremos n es el número de partidos

tendiendo a infinito y mantenemos fijos el número de slots y la clasificiación de equipos en A, B y C.

El algoritmo de fuerza bruta aparentemente parece que debe tener una complejidad de n partidos a repartir en 10 slots diferentes, por lo que será de 10^n. Por lo que nos encontraríamos con un problema exponencial $O(e^n)$.

La realidad es que el hecho de que haya solo 6 tipos de partidos simplifica el problema muchísimo, porque los partidos se repiten entre sí, reduciendo muchisimo las posibildiades. Supondremos que cada uno de los 6 tipos de partidos en una jornada tienen n_1, n_2, n_3, n_4, n_5 y n_6 partidos respectivamente. Por simplificidad supondremos que todos ellos crecen a infinito de forma proporcional a n. Todos los partidos se reparten en los 8 slots de sábado y domingo, excepto 2 de ellos.

Como el número de combinaciones de 6 tipos de elementos en lunes y viernes (siendo dos únicos slots de un único partido) es un número finito. Esto implica una constante multiplicativa adicional, que para el cálculo de complejidad descartaremos.

Teniendo en cuento que es son combinaciones de elemento repetidos, esto hace que las combinaciones finales sean proprocionales a: $\prod_{i=1}^{6} \binom{7+n_i}{n_i}$

Se puede demostrar que $\binom{7+n_i}{n_i}$ es proporcional a n_i^7 cuando n_i tiende a infinito. Teniendo en cuenta que n_i es proporcional a n Y teniendo en cuenta el productorio, la complejidad será $O(n^{42})$. En general siendo k un número finito de slots de tiempo de libre distribución y m un número de tipo de partidos, acaba resultado ser un problema $O(n^{(k-1)m})$.

Por lo que el problema deja de ser exponencial y pasa a ser polinomial. En este caso se ha podido resolver por fuerza bruta porque los números n_i son extremadamente pequeños (al fin y al cabo suman 10 entre los 6 tipos de partidos), por lo que la complejidad es extremadamente baja, por resultar un problema polinomial de un número bajo de elementos.

Nótese que lo que ha convertido el problema en un problema polinomial, en vez de exponencial es que a pesar del número de partidos tender a infinito, los hemos seguido clasificando en 6 tipos de partidos. Si a su vez, la clasificación de partidos fuera en un número infinito de tipos de partidos, el problema volvería a ser exponencial.

(*) **Diseña un algoritmo que mejore la complejidad del algortimo por fuerza bruta. Argumenta porque crees que mejora el algoritmo por fuerza bruta**

Respuesta

Ya hemos encontrado todas las soluciones óptimas de todo el espacio de problemas posibles con un algoritmo de fuerza bruta en menos de 3 minutos, tardando menos de 10 segundos por problema planteado de media. Se nos pide mejorar la solución, para ello solo cabe reemplazar el algoritmo por uno igualmente eficaz, que garantice entontrar todas las soluciones óptimas, pero en un tiempo mucho menor.

Para ello, descartamos algoritmos heurísticos porque perderíamos la garantía de obtener todas las soluciones óptimas, cosa que dado que el problema es bastante pequeño es interesante mantener. Por ello nos decantamos por realizar una ramificación y poda, de forma que no evaluemos todas las soluciones del problema, sino solo aquellas que son susceptibles de ser las mejores con el conocimiento adquirido. El objetivo es poder encontrar todas las soluciones óptimas del problema con garantías pero en unos pocos segundos.

Se ha implementado el algoritmo de ramificación y poda, partiendo de ir por las ramas más probables. De esta forma, el algoritmo empezará asignando los partidos de menos audiencia a lunes y viernes, además empezará distribuyendo los partidos más probables primero y poda cuando con el resto de los partidos es imposible alcanzar el máximo. De esta forma, el algoritmo encuentra de forma muy rápida una solución muy buena, y continúa probando todas las soluciones que no se puedan podar (que no se puedan preveer como peores). De esta forma es extramadamente rápido y ofrece garantías de encontrar todas las soluciones óptimas.

El algoritmo tarda 2.96s en encontrar todas las soluciones del óptimas del espacio completo de problemas. Tarda 156ms de media en encontrar todas las soluciones para una jornada dada.

Nótese que dado lo rápido que es el algoritmo, se ha eliminado la optimización de suponer que los coeficientes de L20 y V20 son iguales, así como los de S20 y D20, para que el algoritmo genérico sirva igual si los coeficientes cambiasen, en función de las encuestas de audiencia televisivas.

[ ]:
```
%%time

import copy

# Realiza la solución por fuerza bruta
def solucion_poda(enunciado):
    def recursiva(slot,audiencia,mejor_audiencia,keys,numbers,index=0,pos=1):
        # Pasa al siguiente dato disponible
        if numbers[index] == 0:
            index+=1
            pos = 1
            if index == len(numbers):
                a = 0
                for i in range(len(audiencia)):
                    a += audiencia[i]*audiencia_simultaneidad[len(slot[i])]
                if math.isclose(a, mejor_audiencia) or a > mejor_audiencia:
                    yield [slot, a]
                return
        # Realiza la poda si es posible calculando una cota de la máxima
        # audiencia alcanzable desde la posición actual y ver si no podría
        # llegar a mejorar la audiencia mejor conocida, pues abortamos misión
        a = 0
        for i in range(len(audiencia)):
            a += audiencia[i]*audiencia_simultaneidad[len(slot[i])]
        maxfactor = 0
        for i in range(pos, 9):
            factor = \
                audiencia_indice_slot[i]*audiencia_simultaneidad[len(slot[i])+1]
            if factor > maxfactor:
                maxfactor = factor
        for i in range(index, len(keys)):
            a += audiencia_indice_partido[keys[i]]*numbers[i]*maxfactor
        if a < mejor_audiencia:
```

```
            return
        # Coloca el siguiente elemento en uno de los slots, los ordena por slots
        # más probables a menos probables (los libres y que tengan un factor
        # mayor)
        partido = keys[index]
        partido_audiencia = audiencia_indice_partido[partido]
        for p in range(pos,9):
            # Añade el elemento en su posición
            slot[p].append(partido)
            prev_audiencia = audiencia[p]
            audiencia[p] += partido_audiencia * audiencia_indice_slot[p]
            numbers[index] -= 1
            # Opera recursivamente
            for solucion in recursiva(slot, audiencia, mejor_audiencia,\
                                      keys, numbers, index, p):
                if math.isclose(solucion[1], mejor_audiencia) or \
                   solucion[1] > mejor_audiencia:
                    yield solucion
                    if solucion[1] > mejor_audiencia:
                        mejor_audiencia = solucion[1]
            # Quita el elemento de la posición
            del slot[p][-1]
            numbers[index] += 1
            audiencia[p] = prev_audiencia
    # Convierte el enunciado a su versión indixada
    enunciado = enunciado_a_indices(enunciado)
    # Elimina una entrada del enunciado
    def enunciado_eliminar(ec, key):
        ec = dict(ec)
        if ec[key] == 1:
            del ec[key]
        else:
            ec[key] -= 1
        return ec    # Contamos los tipos de jornadas
    # Contamos los tipos de jornadas
    ec = dict(collections.Counter(enunciado))
    # Ponemos un tipo de partido el viernes
    solucion = [[], 0]
    for key_v in sorted(ec.keys(),reverse=True):
        # Partidos a distribuir de sábado a lunes
        ec_sl = enunciado_eliminar(ec, key_v)
        # Ponemos un tipo de partido el lunes
        for key_l in reversed(sorted(ec_sl.keys())):
            # Partidos a distribuir entre sábado y domingo
            slot = [[key_v],[],[],[],[],[],[],[],[],[key_l]]
            a = [audiencia_indice_partido[key_v]*audiencia_indice_slot[0],\
                 0,0,0,0,0,0,0,0,\
```

```
                    audiencia_indice_partido[key_l]*audiencia_indice_slot[9]]
            ec_sd = enunciado_eliminar(ec_sl, key_l)
            keys=list(sorted(ec_sd.keys()))
            numbers=[ec_sd[k] for k in keys]
            for s in recursiva(slot, a, solucion[1], keys, numbers):
                # Comprueba si la solución es mejor la reemplaza y
                # si es igual la agrega
                if math.isclose(s[1], solucion[1]):
                    solucion[0].append(copy.deepcopy(s[0]))
                elif s[1] > solucion[1]:
                    solucion[0] = [copy.deepcopy(s[0])]
                    solucion[1] = s[1]
    # Calcula la lista de soluciones totales
    resultado = []
    for s in solucion[0]:
        resultado.append(indices_a_solucion(s))
    # Comprueba que todas ellas tienen la audiencia calculada
    for s in resultado:
        assert(math.isclose(audiencia(s), solucion[1]))
    # Devuelve el resultado
    return resultado

print("Soluciones mejores obtenidas por ramificacion y poda:")
for e in enunciados:
    soluciones = solucion_poda(e)
    print(f"* {enunciado_a_cadena(e)}: {audiencia(soluciones[0]):.4f} Millones")
    for s in soluciones:
        print(f"  - {solucion_a_cadena(s)}")
        assert(math.isclose(audiencia(s), audiencia(soluciones[0])))
```

```
Soluciones mejores obtenidas por ramificacion y poda:
* AA AB BB BB BB BB BB CC CC CC: 7.1725 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AA D12:CC D16:BB D18:BB D20:AB L20:CC
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:CC D16:BB D18:BB D20:AA L20:CC
* AA AB BB BB BB BB BC BC CC CC: 7.2160 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:CC
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:CC
* AA AB BB BB BB BC BC BC BC CC: 7.2230 Millones
  - V20:CC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:BC
  - V20:CC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AB L20:CC
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:BB D20:AA L20:CC
* AA AB BB BB BC BC BC BC BC BC: 7.2225 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BC D18:BB D20:AB L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BC D18:BB D20:AA L20:BC
* AA AC BB BB BB BB BB BC CC CC: 7.0135 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:- D16:BB D18:AC D20:BB,BB L20:CC
```

```
  - V20:CC S12:BC S16:BB S18:BB S20:BB,BB D12:- D16:BB D18:AC D20:AA L20:CC
* AA AC BB BB BB BB BC BC BC CC: 7.0280 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:BC
  - V20:CC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:BC
  - V20:BC S12:BC S16:BB S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:CC
  - V20:BC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:CC
* AA AC BB BB BB BC BC BC BC BC: 7.0350 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AA D12:BC D16:BB D18:BB D20:AC L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AC D12:BC D16:BB D18:BB D20:AA L20:BC
* AB AB AB BB BB BB BB CC CC CC: 6.8125 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:CC D16:BB D18:AB D20:AB L20:CC
* AB AB AB BB BB BB BC BC CC CC: 6.8560 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:CC
* AB AB AB BB BB BC BC BC BC CC: 6.8630 Millones
  - V20:CC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:BC
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AB D20:AB L20:CC
* AB AB AB BB BC BC BC BC BC BC: 6.8625 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BC D18:AB D20:AB L20:BC
* AB AB AC BB BB BB BB BC CC CC: 6.6835 Millones
  - V20:CC S12:BB S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:CC
* AB AB AC BB BB BB BC BC BC CC: 6.7130 Millones
  - V20:CC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:BC
  - V20:BC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:CC
* AB AB AC BB BB BC BC BC BC BC: 6.7200 Millones
  - V20:BC S12:BC S16:BC S18:BB S20:AB D12:BC D16:BB D18:AC D20:AB L20:BC
* AB AC AC BB BB BB BB BB CC CC: 6.4760 Millones
  - V20:CC S12:BB S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:CC
  - V20:CC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:CC
* AB AC AC BB BB BB BB BC BC CC: 6.5055 Millones
  - V20:CC S12:BC S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:BC
  - V20:CC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:BC
  - V20:BC S12:BC S16:BB S18:AC S20:AB D12:- D16:BB D18:AC D20:BB,BB L20:CC
  - V20:BC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AB L20:CC
* AB AC AC BB BB BB BC BC BC BC: 6.5250 Millones
  - V20:BC S12:BC S16:BB S18:BB S20:AB D12:BC D16:BB D18:AC D20:AC L20:BC
  - V20:BC S12:BC S16:BB S18:BB S20:AC D12:BC D16:BB D18:AC D20:AB L20:BC
* AC AC AC BB BB BB BB BB BC CC: 6.2880 Millones
  - V20:CC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:BC
  - V20:BC S12:BB S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:CC
* AC AC AC BB BB BB BB BC BC BC: 6.3175 Millones
  - V20:BC S12:BC S16:BB S18:AC S20:BB,BB D12:- D16:BB D18:AC D20:AC L20:BC
CPU times: user 2.94 s, sys: 20 ms, total: 2.96 s
Wall time: 2.96 s
```

(*)Calcula la complejidad del algoritmo

Respuesta

No he sido capaz de calcular de forma efectiva la complejidad del algoritmo de ramificación y poda.

Evidentemente, por su naturaleza, está acotada para ser inferior que la de la fuerza bruta. Pero, lo que sucede es que dado el diseño del algoritmo, y teniendo en cuenta que empieza por las ramas más probables, la gran mayoría de las ramas son podadas.

No obstante, las reglas del algoritmo son complejas y la capacidad de poda, siendo muy eficaz, es complicada de evaluar de forma aritmética, como para obtener la complejidad con notación de $O(f(n))$.

Según el problema (y tenga sentido), diseña un juego de datos de entrada aleatorios

Respuesta

No tiene sentido generar más datos de entrada. Ya he solucionado el problema para todos los posibles datos de entrada.

Aplica el algoritmo al juego de datos generado

Respuesta

Ya he solucionado el problema para todos los posibles datos de entrada.

Enumera las referencias que has utilizado(si ha sido necesario) para llevar a cabo el trabajo

Respuesta

No se ha utilizado ninguna referencia.

Describe brevemente las lineas de como crees que es posible avanzar en el estudio del problema. Ten en cuenta incluso posibles variaciones del problema y/o variaciones al alza del tamaño

Respuesta

Para mi es difícil como mejorar el problema sin crecer en el tamaño. Ya encuentra todas las soluciones óptimas del problema con garantías en 156ms de media ejecutando en un único core de un ordenador portátil. Para acelerarlo, solo se me ocurre diseñar un algoritmo voraz o GRASP más eficaz antes de la ramificiación y poda, para encontrar una solución mejor antes. Después me sigue pareciendo interesante realizar la ramificación y poda, teniendo en cuenta, el poco tiempo que tarda, para ofrecer garantías de encontrar todas las soluciones óptimas.

No obstante, si el número partidos (y sobre todo el número de tipo de partidos) creciera mucho, puede que el algoritmo de ramificación y poda ya no sea asumible. Llegado a este caso, puede que esa ramificación en todas las posibles combinaciones deje de tener sentido y limitemos a probar solo las ramas más probables. Para ello se puede priorizar las ramas donde los partidos coincidan menos o con las ramas con los partidos de mayor audiencia en los slots con mayor índice de audiencia.

Por los resultados obtenidos, se puede ver, que con los coeficientes dados, en general, es una mala idea tener partidos simultáneos o poner los mejor partidos en slots de tiempo malos (cosa que a priori parecía evidente).

Para esto probablemente, una vez que creciera el tamaño, podemos sacrificar las garantías de encontrar todas las soluciones óptimas que ofrece la ramificación y poda, por un coste computacional acotado que puede ofrecer un algoritmo GRASP, por ejemplo.