EJERCICIOS DE BACKTRACKING DE FINALES ANTERIORES

EXAMEN FINAL 24 DE JULIO DE 2019

4. En una localidad cordobesa, vive el José Agustín Goytisolo quien, apenado por los padecimientos de la mayoría de los vecinos, decide comprometerse en solucionarlos postulándose a la intendencia. Gracias a su entusiasmo y creatividad, en pocos minutos enumera una larga lista de N propuestas para realizar. Pronto descubre que a pesar de que cada una de ellas generaría una satisfacción popular $p_1, p_2, \ldots, p_N$ también provocaría desagrado $q_1, q_2, \ldots, q_N$ en el sector más acomodado de la sociedad local. En principio, el desagrado de cada propuesta es insignificante en número de votos ya que la alta sociedad no es muy numerosa. Pero a José le interesa cuidar su relación con este sector, ya que el mismo tiene suficientes recursos como para dificultar su triunfo en caso de proponérselo.

   Pronto descubre que las propuestas elaboradas son demasiadas para ser publicitadas: tantas propuestas (N) generarían confusión en el electorado. Esto lo lleva a convencerse de seleccionar solamente K de esas N propuestas ($K \leq N$). Se dispone, entonces, a seleccionar K de esas N propuestas de forma tal que la suma de satisfacción popular de las K propuestas elegidas sea máxima y que el descontento total de esas K propuestas en la alta sociedad no supere un cierto valor M.

   José Agustín te contrata para que desarrolles un algoritmo capaz de calcular el máximo de satisfacción popular alcanzable con K de esas N propuestas sin que el descontento supere M.

Datos:

- N propuestas
- Cada propuesta i generaría una satisfacción popular p_i y un desagrado q_i, para $1 \leq i \leq N$.

Se desea calcular el MÁXIMO DE SATISFACCIÓN POPULAR alcanzable con K de las N propuestas sin que el descontento supere M.
Resulta obvio que debo calcular un máximo.

FUNCIÓN RECURSIVA

propuestas(i, k, j) = "máxima satisfacción popular alcanzable eligiendo k propuestas entre 1 e i, sin que el descontento acumulado por esas propuestas supere j"

Rol de los argumentos:

El argumento i indica hasta qué propuesta se está considerando.
El argumento k indica cuántas propuestas quedan por elegirse.
El argumento j indica el descontento que no puede superarse.

LLAMADA PRINCIPAL

propuestas(N, K, M)

FUNCIÓN MATEMÁTICA

```
propuestas(i, k, j) =
{- CASOS BASE -}
{- Tengo que elegir 0 propuestas. Es claro que eligiendo 0 propuestas, la máxima satisfacción que puedo obtener es 0. -}
|i ≥ 0 && k = 0 ----------------> 0

{- Tengo que elegir más de 0 propuestas de entre 0 propuestas. No hay solución. -}
```

```
|i = 0 && k > 0 ----------> -infinito

{- CASOS RECURSIVOS -}
{- Caso recursivo 1: el descontento que generaría la propuesta i es mayor al descontento que puedo tolerar. Entonces, no elijo esa propuesta. -}
|i > 0 && k > 0 && qᵢ > j -----------> propuestas(i-1, k, j)

{- Caso recursivo 2: el descontento que generaría la propuesta i es menor o igual al descontento que puedo tolerar. -}
|i > 0 && k > 0 && qᵢ ≤ j --> max(pᵢ + propuestas(i-1, k-1, j-qᵢ), propuestas(i-1,k,j))
```

EXAMEN FINAL 7 DE JULIO DE 2021 (TEMA 2)

1. La municipalidad de tu pueblo te contrata como experto en problemas de optimización para que diseñes una solución al problema de la combinación de la segunda dosis de la vacuna del covid. Hay n personas, donde cada persona $i \in \{1..n\}$ se ha dado la primera dosis con la vacuna $v_i \in \{AZ, SV, PF\}$, y se han recibido d_k nuevas dosis de cada vacuna para administrar como segunda dosis a las n personas, con $k \in \{AZ, SV, PF\}$.

   Se conoce el porcentaje de inmunidad p_{ij} con $i, j \in \{AZ, SV, PF\}$ que se adquiere al combinar la primera dosis de la vacuna i con la segunda de la vacuna j.

   Se pide maximizar la suma de los porcentajes de inmunidad de las n personas, eligiendo para cada persona i qué vacuna se le aplicará en la segunda dosis, de manera tal que las n personas sean vacunadas.

   (a) (Backtracking) Resolvé el problema utilizando la técnica de backtracking dando una función recursiva. Para ello:
   - Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.
   - Da la llamada o la expresión principal que resuelve el problema.
   - Definí la función en notación matemática.

   (b) (Programación dinámica) Implementá un algoritmo que utilice Programación Dinámica para resolver el problema.
   - ¿Qué dimensiones tiene la tabla que el algoritmo debe llenar?
   - ¿En qué orden se llena la misma?
   - ¿Se podría llenar de otra forma? En caso afirmativo indique cuál.

Datos:

- n personas
- Cada persona i se dio la primera dosis con la vacuna v_i
- Se recibieron d_k nuevas dosis de cada vacuna para administrar como segunda dosis a las n personas.
- Se conoce el porcentaje de inmunidad p_{ij} con i,j ∈ {AZ, SV, PF} que se adquiere al combinar la primera dosis de la vacuna i con la segunda de la vacuna j.

Se desea MAXIMIZAR LA SUMA DE LOS PORCENTAJES DE INMUNIDAD de las n personas, eligiendo para cada persona i qué vacuna se le aplicará en la segunda dosis, de manera tal que las n personas sean vacunadas.

a) BACKTRACKING

FUNCIÓN RECURSIVA

vacunas(i, a, s, p) = “máximo porcentaje de inmunidad que se puede alcanzar para las personas entre la 1 y la i, cuando la cantidad de segundas dosis de la vacuna AZ es a, de la vacuna SV es s, y de la vacuna PF es p”

Rol de los argumentos:

El argumento i indica hasta qué persona se está considerando.
El argumento a indica la cantidad de segundas dosis restantes de la vacuna AZ.
El argumento s indica la cantidad de segundas dosis restantes de la vacuna SV.
El argumento p indica la cantidad de segundas dosis restantes de la vacuna PF.

LLAMADA PRINCIPAL

vacunas(n, d_{AZ}, d_{SV}, d_{PF})

FUNCIÓN MATEMÁTICA

vacunas(i, a, s, pf) =

{- CASOS BASE -}

{- No hay más personas por vacunar. Es claro que el máximo porcentaje de inmunidad que puedo obtener vacunando a 0 personas es 0. -}

| i = 0 ----------> 0

{- Quedan personas por vacunar pero no hay vacunas. No hay solución -}

| i > 0 && a = 0 && s = 0 && pf = 0 ------------> -infinito

{- CASOS RECURSIVOS -}

{- No me quedan más vacunas AZ disponibles (a=0) -}

| i > 0 && a = 0 && s > 0 && pf > 0 ------------->

max($p_{i,SV}$ + vacunas(i-1, 0, s-1, p), $p_{i,PF}$ + vacunas(i-1, 0, s, pf-1))

{- No me quedan más vacunas SV disponibles (s=0) -}

| i > 0 && a > 0 && s = 0 && pf > 0 ------------->

max($p_{i,AZ}$ + vacunas(i-1, a-1, 0, pf), $p_{i,PF}$ + vacunas(i-1, 0, s, pf-1))

{- No me quedan más vacunas PF disponibles (p=0) -}

| i > 0 && a > 0 && s > 0 && pf = 0 ------------->

max($p_{i,AZ}$ + vacunas(i-1, a-1, s, 0), $p_{i,SV}$ + vacunas(i-1, a, s-1, 0))

{- No me quedan más vacunas AZ ni SV disponibles (a=0 && s=0) -}

| i > 0 && a = 0 && s = 0 && pf > 0 ------------->

$p_{i,PF}$ + vacunas(i-1, 0, 0, p-1))

{- No me quedan más vacunas AZ ni PF disponibles (a=0 && pf=0) -}

| i > 0 && a = 0 && s > 0 && pf = 0 ------------->

$p_{i,SV}$ + vacunas(i-1, 0, s-1, 0)

{- No me quedan más vacunas SV ni PF disponibles (s=0 && pf=0) -}
| i > 0 && a > 0 && s = 0 && pf = 0 ---------->
$p_{i,AZ}$ + vacunas(i-1, a-1, 0, 0)

{- Tengo vacunas disponibles de todas las marcas -}
| i > 0 && a > 0 && s > 0 && pf > 0 ---------->
max($p_{i,AZ}$ + vacunas(i-1, a-1, s, pf), $p_{i,SV}$ + vacunas(i-1, a, s-1, pf), $p_{i,PF}$ + vacunas(i-1, a, s, pf-1))

b) PROGRAMACIÓN DINÁMICA

Representaré a las vacunas con un tipo enumerado:

```
type vacunas = enumerate
                AZ
                SV
                PF
               end enumerate

fun vacunas(primera:array[1..n] of vacunas, A:nat, S:nat, P:nat,
            inmunidad[AZ..PF,AZ..PF]) ret max_inmunidad: nat
     var tabla: array[0..n,0..A,0..S,0..P]

     {- CASOS BASE -}
     for a:=1 to A do
       for s:=1 to S do
         for p:=1 to P do
           tabla[0,a,s,p] := 0
         od
       od
     od

     for i:=1 to n do
       tabla[i,0,0,0] := -infinito
     od

     {- CASOS “RECURSIVOS” -}
     {- i > 0 && a = 0 && s > 0 && p > 0 -}
     for i:=1 to n do
       for s:=1 to S do
         for p:=1 to P do
            tabla[i,0,s,p] := max(inmunidad[primera[i], SV] + tabla[i-1,0,s-
     1,p],
```

```
                          inmunidad[primera[i], PF] + tabla[i-1,0,s,p-1])
    od
  od
od

{- i > 0 && a > 0 && s = 0 && p > 0 -}
for i:=1 to n do
  for a:=1 to A do
    for p:=1 to P do
     tabla[i,a,0,p]  :=  max(inmunidad[primera[i],  AZ]  +  tabla[i-1,a-
1,0,p],
                         inmunidad[primera[i], PF] + tabla[i-1,a,0,p-1])
    od
  od
od

{- i > 0 && a > 0 && s > 0 && p = 0 -}
for i:=1 to n do
  for a:=1 to A do
    for s:=1 to S do
     tabla[i,a,s,0]  :=  max(inmunidad[primera[i],  AZ]  +  tabla[i-1,a-
1,s,0],
                         inmunidad[primera[i], SV] + tabla[i-1,a,s-1,0])
    od
  od
od

{- i > 0 && a = 0 && s = 0 && p > 0 -}
for i:=1 to n do
  for p:=1 to P do
    tabla[i,0,0,p] := inmunidad[primera[i], PF] + tabla[i-1,0,0,p-1]
  od
od

{- i > 0 && a = 0 && s > 0 && p = 0 -}
for i:=1 to n do
  for s:=1 to S
    tabla[i,0,s,0] := inmunidad[primera[i], SV] + tabla[i-1,0,s-1,0]
  od
od


{- i > 0 && a > 0 && s = 0 && p = 0 -}
```

```
    for i:=1 to n do
      for a:=1 to A do
        tabla[i,a,0,0] := inmunidad[primera[i], AZ] + tabla[i-1,a-1,0,0]
      od
    od

    {- CASO RECURSIVO (propiamente dicho) -}
    {- i > 0 && a > 0 && s > 0 && p > 0 -}
    for i:=1 to n do
      for a:=1 to A do
        for s:=1 to S do
          for p:=1 to P do
              tabla[n,a,s,p]:= max(inmunidad[primera[i], AZ] + tabla[i-1,a-
1,s,p],
                                   inmunidad[primera[i], SV] + tabla[i-1,a,s-
                                1,p],
                                   inmunidad[primera[i], PF] + tabla[i-1,a,s,p-
                                1])
          od
        od
      od
    od

    {- LLAMADA PRINCIPAL -}
    max_inmunidad := tabla[n, A, S, P]
end fun
```

PRÁCTICO 3- PARTE 3, EJERCICIO 9

9. El juego ↖U↑P↗ consiste en mover una ficha en un tablero de n filas por n columnas desde la fila inferior a la superior. La ficha se ubica al azar en una de las casillas de la fila inferior y en cada movimiento se desplaza a casillas adyacentes que estén en la fila superior a la actual, es decir, la ficha puede moverse a:

   - la casilla que está inmediatamente arriba,
   - la casilla que está arriba y a la izquierda (si la ficha no está en la columna extrema izquierda),
   - la casilla que está arriba y a la derecha (si la ficha no está en la columna extrema derecha).

   Cada casilla tiene asociado un número entero c_{ij} $(i, j = 1, \ldots, n)$ que indica el puntaje a asignar cuando la ficha esté en la casilla. El puntaje final se obtiene sumando el puntaje de todas las casillas recorridas por la ficha, incluyendo las de las filas superior e inferior.

   Determinar el máximo y el mínimo puntaje que se puede obtener en el juego.

Datos:

- Tablero de n filas por n columnas
- Cada casilla tiene asociado un número entero c_{ij} (i,j = 1,...,n) que indica el puntaje a asignar cuando la ficha está en la casilla [i,j]
- El puntaje final se obtiene sumando el puntaje de todas las casillas recorridas por la ficha, incluidas las de las filas superior e inferior.

Se desea determinar el máximo y el mínimo puntaje que se puede obtener en el juego.

BACKTRACKING

FUNCIÓN RECURSIVA (para obtener el puntaje máximo)

up(i,j) = "puntaje máximo que se puede obtener jugando al up partiendo del casillero [i,j] (es decir partiendo del casillero en la fila i, columna j)"

LLAMADA PRINCIPAL

max(up(1,1), up(1,2),…, up(1,n))

FUNCIÓN MATEMÁTICA

up(i,j) =

{- CASO BASE -}

{- Ya llegué a la última fila. Solo puedo sumar el puntaje del casillero en el que estoy. -}

| i = n --------------> c_{nj}

{- CASOS RECURSIVOS -}

{- Estoy en algún casillero de la primera columna (j=1), entonces no puedo subir al casillero de arriba a la izquierda. -}

| i < n && j=1 -------> c_{i1} + max(up(i+1,1), up(i+1, 2))

{- Estoy en algún casillero de la última columna (j=n), entonces no puedo subir al casillero de arriba a la derecha. -}

| i < n && j=n -------> c_{in} + max(up(i+1,n), up(i+1, n-1))

{- Estoy en algún casillero de las columnas del "no extremas". -}

| i < n && 1<j<n ----> c_{ij} + max(up(i+1,j-1), up(i+1,j), up(i+1, j+1))

PROGRAMACIÓN DINÁMICA

```
fun up(puntos: array[1..n,1..n]) ret max_puntaje: nat
     var tabla: [1..n, 1..n] {- tabla[i,j] = up(i,j) -}

     {- CASO BASE -}
     for j:=1 to n do
       tabla[n,j] := puntos[n,j]
```

```
     od

     {- CASOS RECURSIVOS -}
     for i:=n-1 downto 1 do
       tabla[i,1] := puntos[i,1] + max(tabla[i+1,1], tabla[i+1,2])
     od

     for i:=n-1 downto 1 do
       tabla[i,n] := puntos[i,n] + max(tabla[i+1,n], tabla[i+1,n-1])
     od

     for i:=n-1 downto 1 do
       for j:=2 to n-1 do
         tabla[i,j] := puntos[i,j] + max(tabla[i+1,j-1], tabla[i+1,j],
tabla[i+1,j+1])
       od
     od


     {- ¿se puede así? -}
     for i:=n-1 downto 1 do
       for j:=1 to n do
         if j = 1 then
           tabla[i,1] := puntos[i,1] + max(tabla[i+1,1], tabla[i+1,2])
         else if j = n then
           tabla[i,n] := puntos[i,n] + max(tabla[i+1,n], tabla[i+1,n-1])
         else
           tabla[i,j] := puntos[i,j]+max(tabla[i+1,j-1], tabla[i+1,j],
tabla[i+1,j+1])
         fi
       od
     od


{- LLAMADA PRINCIPAL: max_puntaje := max(tabla[1,1], tabla[1,2],...,
tabla[1,n]) -}
     max_puntaje := -infinito
     for j:=1 to n do
       if tabla[1,j] > max_puntaje then
         max_puntaje := tabla[1,j]
       fi
     od
end fun
```

SEGUNDO PARCIAL 15 DE JUNIO DE 2016

3. Se dispone de una balanza de dos platillos (llamémosles A y B) con un objeto sobre el platillo A cuyo peso es W. Se disponde de n pesas, cuyos pesos son $w_1, w_2, \ldots, w_n$. Se pide dar un algoritmo que utilice backtracking para determinar cuál es el menor número de pesas que alcancen para equilibrar la balanza. Es muy importante tener presente que pueden colocarse pesas en ambos platillos. Por ejemplo, si se cuenta con 4 pesas de 1 ($w_1 = w_2 = w_3 = w_4 = 1$) y con una pesa de 5 ($w_5 = 5$), y $W = 2$ el menor número de pesas es 2, lo que se logra colocando dos pesas de 1 en el platillo B. Si $W = 4$, el menor número sigue siendo 2, ya que podemos equilibrar la balanza colocando una pesa de 1 en el platillo A y la de 5 en el platillo B. Si $W = 3$ hay dos formas de obtener el menor número de pesas, que es 3.

   Para obtener el algoritmo se sugiere definir $m(i, j)$ = "el menor número de pesas, eligiendo entre las primeras i pesas, que alcancen para equilibrar la balanza cuando el peso en el platillo A excede en j al peso del platillo B". Con esta definición, el resultado deseado se obtiene calculando $m(n, W)$. Observar que para el caso ilustrado en los ejemplos de arriba ($w_1 = w_2 = w_3 = w_4 = 1$ y $w_5 = 5$) se obtiene $m(5, 2) = m(5, 4) = 2$ y $m(5, 3) = 3$.

   (a) Inciso de entrenamiento. Para el ejemplo dado ($w_1 = w_2 = w_3 = w_4 = 1$ y $w_5 = 5$) ¿cuánto valen $m(5, 1)$, $m(5, 5)$, $m(5, 0)$, $m(2, 1)$, $m(1, 2)$, $m(2, -1)$, $m(5, -5)$? En los dos últimos, j asume valores negativos, lo que significa que el platillo A tiene menos peso que el B.

   (b) Dar una definición recursiva de $m(i, j)$, para el caso general, es decir, cualquiera sean $w_1, w_2, \ldots, w_n$ y W. Puede convenir analizar los siguientes casos: ¿qué ocurre con la balanza cuando j es 0? ¿qué ocurre si i es 0 y $j \neq 0$? En el caso general, $i > 0$ y $j \neq 0$, ¿qué opciones tengo con la i-ésima pesa?

FUNCIÓN RECURSIVA

balanza(i ,j) = "menor número de pesas, eligiendo entre las primeras i pesas, que alcancen para equilibrar la balanza cuando el peso en el platillo A excede en j al peso del platillo B"

Rol de los argumentos:

El argumento i indica hasta qué pesa estamos considerando.

El argumento j indica en cuánto excede el peso del platillo A al peso del platillo B, es decir peso(A) - j = peso(B).

OBSERVACION: j puede ser negativo, caso en el cual el peso del platillo B es mayor que el peso del platillo A.

LLAMADA PRINCIPAL
balanza(n, W)

a) ENTRENAMIENTO
Tengo w1 = w2 = w3 = w4 = 1 y w5 = 5

- balanza(5,1) = 1 (pongo una pesa de 1 en el platillo B y se equilibra la balanza)
- balanza(5,5) = 1 (pongo la pesa de 5 en el platillo B y se equilibra la balanza)
- balanza(5,0) = 0 (ambos platos tienen el mismo peso, la balanza ya está equilibrada)
- balanza(2,1) = 1 (pongo una pesa de 1 en el platillo B y se equilibra la balanza)
- balanza(1,2) = NO HAY SOLUCIÓN, pues solo puedo poner una pesa de 1 (i = 1) pero el peso del platillo A excede en 2 al peso del platillo B
- balanza(2,-1) = 1 (pongo una pesa de 1 en el platillo A y se equilibra la balanza)
- balanza(5,-5) = 1 (pongo la pesa de 5 en el platillo A y se equilibra la balanza)

b) FUNCIÓN MATEMÁTICA
balanza(i, j) =

{- CASOS BASE -}
{- Caso base 1: la balanza ya está equilibrada, no me hace falta poner ninguna pesa -}
|j = 0 ----------> 0

{- Caso base 2: la balanza aún está desequilibrada (j ≠ 0) y no tengo más pesas disponibles para tratar de equilibrarla (i=0). El problema no tiene solución. -}
| i = 0 && j ≠ 0 ---------> infinito

{- CASOS RECURSIVOS -}
{- Pruebo poniendo la pesa i en la balanza y no ponerla. -}
{- Caso recursivo 1: el peso del platillo A es menor al peso del platillo B (j < 0). Entonces, para tratar de equilibrar la balanza, tengo que poner alguna pesa en el platillo A. -}
| i > 0 && j < 0 --------> min(1 + balanza(i-1, j+w_i), balanza(i-1, j))

{- Caso recursivo 2: el peso del platillo B es menor al peso del platillo A (j > 0). Entonces, para tratar de equilibrar la balanza, tengo que poner alguna pesa en el platillo B. -}
| i > 0 && j > 0 --------> min(1 + balanza(i-1, j-w_i), balanza(i-1,j))