Ejercicio 3 (panaderia)

Datos:

- Una panadería recibe n pedidos por importes m1,...,mn
- Solo queda en depósito H de harina
- Los pedidos requieren una cantidad h1,...,hn de harina

Determinar el máximo importe que es posible obtener con la harina disponible

panaderia(i,j) = "mayor monto que puedo obtener realizando algunos pedidos entre 1 e i, de manera tal que la harina necesaria no supere el monto j de harina"

llamada panadería(i,H)

```
panaderia =
            i=0         -> 0                 {-no tengo pedidos-}
            i>0 && hi>j -> panaderia(i-1, j) {-tengo pedidos, pero la harina
no me alcanza para ese pedido-}
            i>0 && hi<j -> max(mi + panaderia(i-1, j-hi),
                                                    panaderia(i-1,j))
```

había pensado en un caso donde hay pedidos pero no tengo harina (que daría -infinito) pero el enunciado dice que ya hay algo de harina

Dinámica:

las filas se llenan de arriba para abajo y las columnas en cualquier orden.

```
fun panaderia(p: array[1..n], m: array[1..n] of nat, H: nat) ret
                                                              res: nat
      var tabla[0..n,0..H] of nat
      for j:=0 to H do
            tabla[0,j] = 0
      od
      for i:=0 to n do
            for j:= to H do
                  if h[i] > j then
                        tabla[i,j] := tabla[i-1,j]
                  else
                        tabla[i,j] := m[i] + tabla[i-1,j-hi] `max`
                                                      tabla[i-1,j]
                  fi
            od
      od
      res := tabla[i,H]
end fun
```

Ejercicio 4 (globo aerostático)

Datos:

- Tengo n objetos cuyos pesos p1,...,pn y valores v1,...,vn conozco.
- Si me desprendo de al menos P kilogramos logró recuperar altura y llegar a tierra firme.

Cuál es el menor valor total de los objetos que necesita arrojar para llegar sano y salvo a la costa?

globo(i,h) = "el menor valor posible del que debo desprenderme tirando algunos objetos entre 1 e i, de manera tal que su peso no sea mayor o igual a h"

llamada globo(n, P)

```
globo(i,h) =
      si h=0      -> 0 {-no me paso del peso-}
      si i=0,h>0  -> +infinito {-me paso del pedo, pero no puedo tirar
nada-}
      si i>0,h>0  -> min(vi + globo(i-1,h-pi), globo(i-1,h))
```

dinámica -> igual que el de panadería.

Ejercicio 5: teléfono

datos:

- Los amigos ofrecen el día de partida y de regreso y un monto a pagar por el alquiler.

Determinar el máximo valor alcanzable alquilando el teléfono.

telefono(i,d) = "Máximo monto al alquilar el teléfono a algunos amigos desde el 1 hasta el i, a partir del día d"

La llamada es telefono(n,1) calculamos que pasa si le alquiló al amigo n, y también si no se lo alquilo. En el primer caso calculare que pasa si se lo alquiló al n-1 o no pero solo si es posible.

```
telefono(i,d) | i=0  -> 0
              | i<d  -> telefono(i-1, d)
              | i>=d -> max(mi*(ri-pi) + telefono(i+1,ri+1),
                                                       telefono(i-1,d))
```

De otra forma, la anterior el supuesto era que hay solución.

telefono(d) = "máxima plata que puedo ganar prestando el teléfono a partir del día d hasta el último día"

llamada principal -> telefono(0)

```
telefono(d) =
            pi<d  -> 0
```

```
            pi>=d -> max(telefono(d+1), {-si el dia d no lo presto-}
                         mi*(ri-pi-1)+telefono(ri+1) {-se lo presto a i-})
```

Dinámica:

```
fun telefono(p: array[1..n] of nat, r: array[1..n] of nat, m:
                  array[1..n] of nat, ultima_partida: nat,
                                ultimo_dia: nat) ret gano: nat
      var tabla: array[0..ultimo_dia] of nat
      for d:=ultima_partida+1 to ultimo_dia do
            tabla[d] := 0
      od
      for d:=ultima_partida downto 0 do
            aux := 0
            for i:=1 to n do
                  if p[i] == d then
                        aux := aux `max` (m[i]*(r[i]-p[i]+1)+
                                                            tabla[r[i]+1]
                  fi
            od
            tabla[d] := tabla[d+1] `max` aux
      od
      gano := tabla[0]
end fun
```

Ejercicio 7 (dos mochilas)

En el problema de la mochila se buscaba el máximo valor alcanzable al seleccionar entre n objetos de valores v1,...,vn y pesos w1,...,wn respectivamente, una combinación de ellos que quepa en una mochila de capacidad W. Si se tienen dos mochilas con capacidad W1 y W2. Cuál es el valor máximo alcanzable al seleccionar objetos para cargar en ambas mochilas?

2mochilas(i,j,k) = “máximo valor posible al guardar algunos objetos entre 1 y el i, dado que a la mochila 1 le queda capacidad j y a la mochila 2 le queda capacidad k”

llamada -> 2mochilas(n,K1,K2)

```
2mochilas(i,j,k) |i=0               -> 0
                 |i>0,w_i>j,w_i>k   -> 2mochila(i-1,j,k)
                 |i>0,w_i<=j,w_i>k  ->
                     max(vi+2mochilas(i-1,j-w_i,k),2mochilas(i-1,j,k))
                 |i>0,w_i>j,w_i<=k  ->
                   max(vi+2mochilas(i-1,j,k-w_i),2mochilas(i-1,j,k))
                 |i>0,w_i<=j,w_i<=k ->
                   max(2mochilas(i-1,j,k), vi+2mochilas(i-1,j-w_i,k),
                      vi+2mochilas(i-1,j,k-w_i))
```

Análisis de los casos:

- Si no tengo elementos para guardar entonces i=0 y el resultado es 0.
- Si tengo al menos un elemento para guardar y la capacidad de esos elementos hace que las dos mochilas se sobrecarguen entonces paso al próximo elemento y hago un llamada recursiva con i-1.
- Después tengo todas las combinaciones del caso anterior

<u>Ejercicio 8 (Fábrica de autos)</u>

Datos:

- Dos líneas de ensamblaje y cada línea tiene n estaciones de trabajo, S11,...,S1n para la primera y S21,...,S2n para la segunda.
- Dos estaciones S1i y S2i para i=1…n
- Hacer en mismo trabajo pero lo hacen con costos a1i y a2i
- Para fabricar un auto debemos pasar por n estaciones de trabajo, no necesariamente todas de la otra línea de montaje.

Encontrar el costo mínimo de fabricar un automóvil usando ambas líneas.

autos(i,j) = "Menor costo de fabricar el auto desde la estación 1, hasta la estación j, haciendo la misma en la línea i"

```
autos(i,j) =
      si j=1     -> a_i,1
      si j>1,i=1 -> a_1,j + min(autos(1,j-1),autos(2,j-1)+t_2,j-1)
      si j>1,i=2 -> a_2,j + min(autos(2,j-1),autos(1,j-1)+t_1,j-1)
```

Casos:

- Si tengo 1 sola estación solo tengo un costo.
- Si tengo al menos una estación y lo hago en la misma línea de montaje.
- Si tengo al menos una estación y lo hago en dos líneas de montaje.

Programación dinámica

<u>Ejercicio 1</u>

1. Dar una definición de la función cambio utilizando la técnica de programación dinámica a partir de la siguiente definición recursiva (backtracking):

$$cambio(i,j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \land i = 0 \\ \min_{q \in \{0,1,\ldots,j \div d_i\}} (q + cambio(i-1, j - q * d_i)) & j > 0 \land i > 0 \end{cases}$$

Denominaciones: d_1,....,d_n
Dar cambio por un monto total k

Llamada principal: cambio(n,k)
La tabla tiene la dimensión: (n+1)x(k+1)

Cómo la tengo que llenar? Las filas de arriba abajo, las columnas no importa el orden. Xq? Para llenar cambio(i,j) debo mirar elementos en posiciones cambio(i-1,j-q*d_i) osea siempre en la fila de arriba a la actual.

```
fun cambio(d: array[1..n] of nat, k: of nat) ret r: nat
      var tabla: array[0..n,0..k] of nat
      var mimin: nat

      {- casos base -}
      for i:= 0 to n do
            tabla[i,0] := 0
      od
      for j:= 1 to k do
            tabla[0,j] := infinito
      od
      {- caso recursivo -}
      for i:=1 to n do
            for j:=1 to k do
                        mimin := +infinito
                        for q:=0 to (j/d[i]) do
                              mimin := mimin´min´(q+tabla[i-1,j-q*d[i]])
                        od
                        tabla[i,j] := mimin
            od
      od
      r := tabla[n,k]
end fun
```

Ejercicio 3

3. Se tiene la siguiente definición recursiva, para $0 \leq i, j \leq n$:

$$gunthonacci(i,j) = \begin{cases} 1 & \text{si } i = 0 \wedge j = 0 \\ 1 & \text{si } i = 0 \wedge j = 1 \\ 1 & \text{si } i = 1 \wedge j = 0 \\ gunthonacci(i, j-2) + gunthonacci(i, j-1) & \text{si } i = 0 \wedge j > 1 \\ gunthonacci(i-2, j) + gunthonacci(i-1, j) & \text{si } i > 1 \wedge j = 1 \\ gunthonacci(i, j-1) + gunthonacci(i-1, j) & \text{si } i > 0 \wedge j > 1 \end{cases}$$

donde la llamada principal es $gunthonacci(n, n)$

La tabla es de nxn, las filas se llenan de arriba para abajo y las columnas no importa el orden.

Como la llamada principal es gunthonacci(n,n) toma el mismo parámetro entonces la función en programación dinámica solo tiene a n: nat como parámetro.

```
fun gunthonacci(n: nat) ret r: nat
      var tabla: array[0..n][0..n] of nat

      {- Casos base -}
      tabla[0,0] := 1
```

```
      tabla[0,1] := 1
      tabla[1,0] := 1

      {- Casos recursivos -}
      for j:=2 to n do
            tabla[0,j] := tabla[0,j-2] + tabla[0,j-1]
      od

      for i:=2 to n do
            tabla[i,1] := tabla[i-2,1] + tabla[i-1,1]
      od

      for i:=1 to n do
            for j:=2 to n do
                  tabla[i,j] := tabla[i,j-1] + tabla[i-1,j]
            od
      od

      r := tabla[n,n] {-Llamada principal-}
end fun
```

______________________________ Final, Ejercicio 2

El presidente de tu país te acaba de elegir como asesor para tomar una serie de medidas de producción que mejoren la situación económica. En el análisis preliminar se proponen **n** medidas, donde cada medida **i** $\in$ **{1, .., n}** producirá una mejora económica de **mi** puntos, con **mi > 0**. También se analizó para cada una el nivel de daño ecológico di que producirá, donde **di > 0**. El puntaje que tendrá cada medida i está dado por la relación mi/di.

Se debe determinar cuál es el máximo puntaje obtenible eligiendo **K** medidas, con **K < n**, de manera tal que la suma total del daño ecológico no sea mayor a **C**. (tres parámetros)

Se pide lo siguiente:

(a) Especifica precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.

medidas(i,c,k) = "Obtiene la máxima mejora económica posible eligiendo k propuestas entre la 1 y la i, de manera tal que la suma total del daño de las medidas elegidas no supere c"

i -> representa hasta qué medida estoy considerando para la propuesta, desde 0 hasta i.
c -> representa cuánto daño ecológico puedo aún provocar eligiendo las restantes propuestas.
k -> representa cuántas propuestas me faltan elegir

(b) Da la llamada o la expresión principal que resuelve el problema.

La llamada principal -> medidas(n,C,K)

(c) Definir la función en notación matemática.

```
medidas(i,c,k) =
      si k=0             --> 0
      si i=0, k>0        --> -infinito
      si i>0, k>0, d_i>c --> medidas(i-1,c,k)
      si i>0, k>0, d_i<c --> max(medidas(i-1,c,k), m_i/d_i +
                                 medidas(i-1,c-d_i,k-1)
                                )
```

```
fun propuestas(p: array[1..n] of nat, q: array[1..n] of nat, K,C: nat) ret
                                                                      r:nat
      var tabla[0..n,0..C,0..K] of nat
```

---

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.

Una persona se postula para la intendencia.

Tiene N propuestas para realizar.
Generan una satisfacción popular p1,p2,...,pN.
Desagrado q1,q2,...,qN.
En principio, el desagrado de cada propuesta es insignificante en número de votos ya que la alta sociedad no es muy numerosa, pero al postulante le preocupa la relación con este grupo social.

Entonces, va a seleccionar K de las N propuestas (K <= N) 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 supero un cierto valor.

la función:

propuestas(i,k,m) = "La mayor satisfacción popular que se obtiene de elegir k propuestas entre todas las propuestas que van de 1 a i, sin que la suma de desagrados supere un monto m.

llamada -> propuestas(n,K,M)

```
propuestas(i,k,m) =
            i=0,k>0        -> -infinito
            i>=0,k=0       -> 0
            i>0,k>0,q_i>m -> propuestas(i-1,k,m)
            i>0,k>0,q_i<m -> max(propuestas(i-1,k,m),
                                          p_i+propuestas(i-1,k-1,m-q_i))
```

En programación dinámica:

```
fun propuestas(p: array[1..N], q: array[1..N], K, M: nat) ret r: nat
      var tabla[0..N,0..K,0..M]
      {- casos base -}
      for i:=0 to N do
            for m:=0 to M do
                  tabla[i,0,m] := 0
            od
      od
      for k:=0 to K do
            for m:=0 to M do
                  tabla[0,k,m] := -inf
            od
      od
      for i:=1 to N do
            for k:=0 to K do
                  for m:=0 to M do
                        if q[i] > m then
                              tabla[i,k,m] := tabla[i-1,k,m]
                        else
                              tabla[i,k,m] :=
                                          max(tabla[i-1,k,m],p[i]+tabla[i-1
                                                            ,k-1,m-q[i]])
                        fi
                  od
            od
      od
      r := tabla[N,K,M]
end fun
```

1. (Backtracking) En el piso 17 de un edificio que cuenta con n oficinas iguales dispuestas de manera alineada una al lado de la otra, se quieren pintar las mismas de modo tal que no haya dos oficinas contiguas que resulten pintadas con el mismo color. Se dispone de 3 colores diferentes cuyo costo por oficina es C_1, C_2 y C_3 respectivamente. Para cada oficina i, el oficinista ha expresado su preferencia por cada uno de los tres colores dando tres números p_1^i, p_2^i y p_3^i, un número más alto indica mayor preferencia por ese color. Escribir un algoritmo que utilice la técnica de backtracking para obtener el máximo valor posible de (sumatoria para i desde 1 a n, de $p_{j_i}^i/C_{j_i}$, es decir, que maximice $\sum_{i=1}^{n} p_{j_i}^i/C_{j_i}$), sin utilizar nunca el mismo color para dos oficinas contiguas.

Antes de dar la solución, especificá con tus palabras qué calcula la función recursiva que resolverá el problema, detallando el rol de los argumentos y la llamada principal.

Datos:

n oficinas.

3 colores diferentes cuyo costo por oficina es C1, C2 y C3.

Se quieren pintar de modo que no haya dos oficinas contiguas que resulten pintadas del mismo color.

Para cada oficina i, se expresó la preferencia por cada uno de los tres colores dando tres números p_1^i, p_2^i y p_3^i.

Escribir un algoritmo para obtener el máximo valor posible de la sumatoria que para i desde 1 a n de p_j^i/C_j, es decir, si utilizar nunca el mismo color para dos oficinas contiguas

función -> pintar(i,j) = "Valor máximo posible de pintar las oficinas 1,2,...,i sin usar el color j para la oficina i"

La llamada principal -> pintar(n,-1) con -1 no está pintada.

```
pintar(i,j) =
            i=0      -> 0
            i>0,j=1  -> max(p_2^i/C2+pintar(i-1,2),p_3^i/C3+pintar(i-1,3))
            i>0,j=2  -> max(p_1^i/C1+pintar(i-1,1),p_3^i/C3+pintar(i-1,3))
            i>0,j=3  -> max(p_1^i/C1+pintar(i-1,1),p_2^i/C2+pintar(i-1,2))
            i>0,j=-1 -> max(p_3^i/C3+pintar(i-1,3),
                            p_2^i/C2+pintar(i-1,2),p_1^i/C1+pintar(i-1,1))
```

Otra solución posible (me gustan más)

Función: pintar(i,j) = "Valor máximo posible de pintar las oficinas 1,2,...,i usando el color j para la oficina i"

Llamada principal -> max(pintar(n,1),pintar(n,2),pintar(n,3))

Definición recursiva:

```
pintar(i,j) =
            i=0 -> 0
            i>0,j=1 -> p_j^i/C_j + max(pintar(i-1,2),pintar(i-1,3))
            i>0,j=2 -> p_j^i/C_j + max(pintar(i-1,1),pintar(i-1,3))
            i>0,j=3 -> p_j^i/C_j + max(pintar(i-1,1),pintar(i-1,2))
```

1. (Backtracking) Debemos llenar con dominós una fila de n casilleros con n par. Cada ficha ocupa 2 casilleros. Contamos con infinitas fichas de todos los tipos. Cada ficha tiene dos números i,j (de 0 a 6) y tiene un puntaje $P_{i,j}$ $(= P_{j,i})$. Como siempre en el dominó, al poner dos fichas juntas los números de los casilleros adyacentes deben coincidir. El último número de la última ficha debe ser un 6. Además, cada casillero tiene un número prohibido $c_1, \ldots, c_n$. Al colocar una ficha en dos casilleros, ésta debe respetar los números prohibidos.

   Escribir un algoritmo que utilice la técnica de backtracking para obtener el máximo puntaje posible colocando las fichas de manera que se repeten todas las restricciones. Antes de dar la solución, especificá con tus palabras qué calcula la función recursiva que resolverá el problema, detallando el rol de los argumentos y la llamada principal.

   Ejemplo: Con n = 6 debemos usar tres fichas. Si los números prohibidos para los casilleros son

3	2	6	5	0	5

   una posible solución es:

4	3	3	1	1	6

Datos:

Llenar con dominós una fila de n casilleros con n par.
Cada ficha ocupa 2 casilleros.
Tenemos infinitas fichas de todos los tipos.
Cada ficha tiene 2 números i,j (0 a 6) y un puntaje Pi,j = Pj,i.
Los números de los casilleros adyacentes deben coincidir.
Cada casillero tiene un número prohibido c1,....,cn.

Escribir un algoritmo para obtener el máximo puntaje posible colocando las fichas de manera que se respeten todas las restricciones.

domino(i,j) = "máximo puntaje posible colocando fichas de dominó en las casillas 1 a i respetando los números prohibidos y poniendo como último valor del último domingo j"

llamada principal -> domino(n,6)

```
domino(i,j) =
        i=0 -> 0
        i>0 -> max(Pk,j + domino(i-2,k)) donde k=0..6,k!=C_i-1,k!=C_i-2
```

```
domino(i,j) =
        i=0   -> 0
        j=c_i -> -infinito
        i>0   -> max_{k=0..6,k!=c_i-1} dominos(i-2,k) + P_k,j
```

siempre tengo una ficha por eso solo hay una llamada de domino(.)

<u>programación dinámica:</u>

llamada, domino(n,6)

llenado? las filas de izquierda a derecha y las columnas no importa el orden

tabla: array[0..n,0..6]

```
fun domino(p: array[0..6,0..6] of int, c: array[1..n] of nat) ret t: int
    var tabla: array[0..n,0..6] of int
    var mimax: int

    for j:=0 to 6 do
        tabla[0,j] := 0
    od
    for i:=2 to n do
        if i % 2 == 0 then {-xq voy de 2 casilleros-}
            for j:=0 to 6 do
                mimax := -infinito
                for k:=0 to 6 do
                    if k != c[i-1] && k != c[i-2] then
                        mimax := max(mimax,
                                     domino[i-2,k]+P[k,j])
                    fi
                od
```

```
                        tabla[i,j] := mimax
                    od
                fi
        od
        r := tabla[n,6]
end fun
```

(Backtracking) Te vas de viaje a la montaña viajando en auto k horas hasta la base de un cerro, donde luego caminarás hasta el destino de tus vacaciones. Tu auto no es muy nuevo, y tiene un stereo que solo reproduce cds (compact-disks). Buscás en tu vasta colección que compraste en los años 90 y tenés p cds, con $p > k$, que duran exactamente una hora cada uno. Encontrás también un cuaderno donde le diste una puntuación entre 1 y 10 a cada cd de tu colección. Cuanto mayor la puntuación, más es el placer que te da escucharlo. Dado que no sos tan exigente, querés que el puntaje promedio entre dos discos consecutivos que escuches, no sea menor a 6. Así por ejemplo si en la hora 2 escuchás un cd que tiene puntaje 8, en la hora 3 podrías escuchar uno que tenga puntaje al menos 4.

Encontrar una combinación de cds para escuchar en las k horas de viaje, cumpliendo la restricción de que en dos horas consecutivas el puntaje promedio de los dos discos sea mayor o igual a 6, maximizando el puntaje total de los k discos que escucharás.

    Viaje en auto de j horas
    p cds, con p>k que duran exactamente 1hs
    cuaderno con puntuación de los cds (mientras más puntos más placer)
    el puntaje promedio, no sea menor a 6.

Encontrar una combinación de cds para escuchar en las k horas de viaje, cumpliendo la restricción de que en dos horas consecutivas el puntaje promedio de los dos discos sea mayor o igual a 6. Maximizando el puntaje total de los k discos que escucharás.

Función: cds(i,J,a) = “Máxima puntuación al escuchar en las horas 1..i, i cds seleccionados del conjunto J, con puntaje promedio >= 6 entre dos cds consecutivos de 1..i-1 salvo el último que en promedio con “a” debe ser >= 6.

llamada principal= max_{1 <= a <= 10} cds(k,{1,2,..,p},a) ó cds(k,{1,2,...,p},1000)

función recursiva:

```
cds(i,J,a) =
            i=0 -> 0
            i>0 -> max_{(s_j + a)/2 >= 6} cds(i-1, J-1, s_j) + s_j)
```

---

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.

Los dos últimos ejercicios, también pueden resolverse planteando un grafo dirigido y recurriendo al algoritmo de Dijkstra. ¿De qué manera? ¿Serán soluciones más eficientes?

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:

1) la casilla que está inmediatamente arriba
2) la casilla que está arriba y a la izquierda o derecha

Cada casilla tiene asociado un número entero cij que indica el puntaje a asignar cuando la ficha esté en la casilla.

Máximo puntaje y el mínimo puntaje que se puede obtener en el juego?

Función: mejor_juego(i,j) = "Mayor puntaje obtenible jugando al up, desde el casillero [i,j] hasta algún casillero de la última fila, es decir, hasta [n,k], donde k está entre 1 y n"

llamada = **max (mejor_juego(1,1), mejor_juego(1,2),.......,mejor_juego(1,n))**

```
mejor_juego(i,j) =
            i=n     -> c_n,j
            i<n,j=1 -> c_i,1 + max(mejor_juego(i+1,1),mejor_juego(i+1,2))
            i<n,j=m -> c_i,n + max(mejor_juego(i+1,n-1),mejor_juego(i+1,n))
            i<n,1<j<n -> c_i,j + max(mejor_juego(i+1,j-1),
                        mejor_juego(i+1,j),mejor_juego(i+1,j+1))
```

Ejercicio 4, final 4 de julio 2017

Tenemos n monedas con denominaciones d1,d2,...,dn.
Tienes que pagar un valor C por el café y la suma total de los valores de las monedas alcanza.
Se pide encontrar el menor monto a pagar que sea mayor o igual a C.

menor_monto(i,j) = "Menor monto posible pagar utilizando las monedas con las denominaciones d1,...,dn tal que la suma de las denominaciones de las monedas elegidas sea mayor o igual a j"

```
llamada principal -> menor_monto(n,C)

menor_monto(i,j) =
            j<0     -> 0
            i=0,j>0 -> +infinito
            i>0,j>0 -> min(menor_monto(i-1,j),
                                        d_i+menor_monto(i-1,max(0,j-d_i)))
```

______________________________ Parcial viejo

El profe de algoritmos 2 tiene n medias diferentes, con n n´umero par (digamos n = 2m). Hay una tabla P[1..n, 1..n] tal que P[i, j] es un n´umero que indica cu´an parecida es la media i con la media j. Tenemos P[i, j] = P[j, i] y P[i, i] = 0. Dar un algoritmo que determine la mejor manera de aparear las n medias en m pares. La mejor manera significa que la suma total de los P[i, j] lograda sea lo mayor posible. Es decir, si decidimos aparear i1 con j1, i2 con j2, ... im con jm, la sumatoria P[i1, j1]+P[i2, j2]+...P[im, jm] debe ser lo mayor posible. Un apareamiento debe aparear exactamente una vez cada media.

```
n medias diferentes(con n par).
Valores P[i,j] indicando parecido entre media i y media j.
Queremos armar m(=n/2) pares de medias maximizando la suma de parecidos.

Función -> aparear(S) = "Suma de parecidos del mejor apareamiento
considerando medias sacadas del conjunto S"
llamada principal-> aparear({1,2,...,n})

aparear(S) =
            S es vacio -> 0
            S no es vacio -> max_{i,j en S tal que
                                        i!=j}(P[i,j]+aparear(S-{i,j})
```

____

Ejercicio 2 - Final 18/6/2018

- Tenemos dados c_1, ..., c_n, d_1, ..., d_k.

  m(i, j) = (c_i → si j = k
  | d_j → si i = n y j < k
  | m(i, j + 1) + m(i + 1, j)
- Llamada principal: m(1, 1)

programación dinámica: las filas las llenó de abajo hacia arriba y las columnas de derecha a izquierda (downto)

tamaño de la tabla: array[0..n,0..k]

```
fun m(c: array[1..n] of int, d: array[1..k] of int) ret r: int
      var tabla: array[1..n,1..k] of int
      {- Casos base -}

      for i:=1 to n do
            tabla[i,k] := c[i]
      od
      for j:=1 to k do
            tabla[n,j] := d[j]
      od
      for i:=n-1 downto 1 do
            for j:=k-1 downto 1 do
                  tabla[i,j] := tabla[i,j+1] + tabla[i+1,j]
            od
      od
      r := tabla[1,1]
end fun
```

Ejercicio 4 6/8/2014

n objetos con pesos p1,...,pn
n cajas de capacidad p
minimizar el número de cajas a emplear

Función -> m(i,j1,...,jn) = "menor número de cajas que se necesitan emplear para embalar los objetos 1,...,i cuando las capacidades restantes de las cajas son j1,...,jn.

Llamada principal -> m(n,P,...,P)

Definición recursiva:

```
m(i,j1,...,jn) =
            i=0 -> k tal que jk < P, k=1..n {-xq ya hay cajas utilizadas-}
            i>0 -> min_{k=1,..n con jk>=pi} m(i-1,j_1,...,j_k -pi,...,jn)
```

Final 7/7/21 tema 1, ej 1

tiempo T fijo para cada cambio de ruedas
n sets de ruedas (quiero saber cuantos de estos uso)
t1,...,tn tiempos por vuelta para cada set de ruedas
v1,...,vn vida útil en cant de vueltas para cada set de ruedas
m vueltas

mejor_tiempo(i,j) = “tiempo de carrera mínimo considerando los juegos de ruedas 1,...,i para hacer j vueltas”

llamada -> mejor_tiempo(n,m)

Definición recursiva:

```
mejor_tiempo(i,j) =
            i=0,j>0 -> infinito
            j=0     -> 0
            j>0,i>0 -> min(T+(vi min j)*ti+mejor_tiempo(i-1,max(j-vi,0)),
                                    mejor_tiempo(i-1,j))
```

el (vi min j) es porque puede ser que me quedan x vueltas y use ruedas para más vueltas.

el max(j-vi,0) es para que no quede negativo.

Programación dinámica:
Tabla de (n+1)x(m+1)
Orden de llenado: filas de arriba hacia abajo y columnas no importa

```
fun mejor_tiempo(t: array[1..n] of real, v: array[1..n] of real, T: real,
                                                m:int) ret r: real
      var tabla: array[0..n,0..m] of real
      {- Casos base -}
      for i:=0 to n do
            tabla[i,0] := 0.0
      end
      for j:=0 to m do
            tabla[0,j] := infinito
      od

      for i:=1 to do n
            for j:=1 to do m
                  tabla[i,j] := min(T+t[i]*(v[i] min j) +
                              tabla[i-1,(j-v[i]) max 0], tabla[i-1,j])
```

```
            od
      od
      r := tabla[n,m]
end fun
```

____________________________________ 7/7/21 vacunas

n personas
v1,...,vn en {AZ,SV,PF} indicando la primera dosis de cada persona
dk con k en {AZ,SV,PF} indicando la cantidad de 2da dosis de cada vacuna
pi,j con i,j en {AZ,SV,PF} indica porcentaje de inmunidad que da la 1ra dosis i + 2da dosis j

Función -> mejor_inmunidad(i,a,s,f) = “Máxima suma de porcentajes de inmunidad vacunando a las personas 1,....,i con dosis disponibles a,s,f de las vacunas AZ, SV y PF respectivamente”

llamada principal -> mejor_inmunidad(n,d_AZ, d_SV, d_PF)

hay una versión con todos los casos

```
mi(i,a,s,f) =
      i=0 -> 0
      i>0, a=0 ó s=0 ó f=0 -> -infinito
      i>0, a=0 ó s=0 ó f>0 -> p_{vi,PF} + mi(i-1,a,s,f-1)
      i>0, a=0 ó s>0 ó f=0 ->
      i>0, a=0 ó s>0 ó f>0 -> max(....)
      i>0, a>0 ó s=0 ó f=0 ->
      i>0, a>0 ó s=0 ó f>0 -> max(....)
      i>0, a>0 ó s>0 ó f=0 -> max(....)
      i>0, a>0 ó s>0 ó f>0 -> max(p_{vi,AZ} + mi(i-1,a-1,s,f),
                                  p_{vi,SV} + mi(i-1,a,s-1,f),
                                  p_{vi,PF} + mi(i-1,a,s,f-1))
```

(bastante largo para pasarlo a dinámica)

las dimensiones -> [0..n,0..d_AZ,0..d_SV,0..d_PF]

la primera dimensión se llena de arriba hacia abajo, las demás no importa el orden

```
mi(i,a,s,f) =
      i=0 -> 0
      i>0,a=-1 ó s=-1 ó f=-1 -> -infinito
      i>0,a>=0 | s>=0 | f>=0 -> max(p_{vi,AZ}+mi(i-1,a-1,s,f),
                                    p_{vi,SV}+mi(i-1,a,s-1,f),
                                    p_{vi,PF}+mi(i-1,a,s,f-1))
```

Programación dinámica -> 0..n,-1..d_AZ,-1..d_SV-1..d_PF

llenado -> llenamos las filas de arriba hacia abajo, para el resto no importa el orden

```
fun mi(v: array[1..n] of nat, d: array[1..3] of nar, p: array[1..3,1..3] of
                                                       real) ret r: real
      var tabla: array[0..n,0..d[1],0..d[2],0..d[3]]

      for a:=0 to d[1] do
            for s:=0 to d[2] do
                  for f:=0 to d[3] do
                        tabla[0,a,s,f] := 0
                  od
            od
      od

      for i:=1 to n do
            for a:=0 to d[1] do
                  for s:=0 to d[2] do
                        for f:=0 to d[3] do
                              mimax := -infinito
                              if a>0 then
                                    mimax := mimax max
                                          (p[v[i],1]+tabla[i-1,a-1,s,f]
                              fi
                              if s>0 then
                                    mimax := mimax max
                                          (p[v[i],2]+tabla[i-1,a,s-1,f]
                              fi
                              fi f>0 then
                                    mimax := mimax max
                                          (p[v[i],3]+tabla[i-1,a,s,f-1]
                              fi
                              tabla[i,a,s,f] := mimax
                        od
                  od
            od
      od
      r := tabla[n,d[1],d[2],d[3]]
end fun
```

2. (Backtracking) Se tienen n objetos de peso $p_1, \ldots, p_n$ respectivamente. Se tiene una mochila de capacidad K. Dar un algoritmo que utilice backtracking para calcular el menor desperdicio posible de la capacidad de la mochila, es decir, aquél que se obtiene ocupando la mayor porción posible de la capacidad, sin excederla. Definir primero en palabras la función aclarando el rol de los parámetros o argumentos.

n objetos de peso p1,...,pn
mochila de capacidad K
calcular menor desperdicio posible de la capacidad de la mochila

mochila(i,j) = "El menor desperdicio posible de la mochila con capacidad j considerando 1..i objetos"

llamada -> mochila(n,K)

definición recursiva ->

```
mochila(i,j) =
            j=0          -> 0
            j>0,i=0      -> +infinito
            j>0,i>0,pi>j -> mochila(i-1,j) #la mochila tiene capacidad y hay
elementos pero el peso es mayor que la capacidad
            j>0,i>0,pi<j -> min(mochila(i-1,j),mochila(i-1,j-pi)) #la
mochila tiene capacidad y hay elementos y hay capacidad entonces se puede
```

programación dinámica:

Dimensiones de la tabla: (n+1)x(K+1)
Orden -> llenamos las filas de arriba hacia abajo, las columnas de izquierda a derecha

```
fun mochila(p: array[1..n] of nat, K: nat) ret r: nat
      var tabla: array[0..n,0..K] of nat

      {- casos base -}
      for i:=0 to n do
            tabla[i,0] := 0    od
      for j:=1 to K do
            tabla[0,j] := +infinito
      od
      for i:=1 to n do
            for j:=1 to K do
                  if p[i]>j then
                        tabla[i,j] := tabla[i-1,j]
                  else
                        tabla[i,j] := min(tabla[i-1,j],tabla[i-1,j-p[i]])
                  fi
            od
      od
      r := tabla[n,K]
end fun
```

2. Te encontrás frente a una máquina expendedora de café que tiene un letrero que indica claramente que la máquina "no da vuelto". Buscás en tu bolsillo y encontrás exactamente n monedas, con las siguientes denominaciones enteras positivas: $d_1, d_2, \ldots, d_n$. Una rápida cuenta te transmite tranquilidad: te alcanza para el ansiado café, que cuesta C. Teniendo en cuenta que la máquina no da vuelto, dar un algoritmo que determine el menor monto posible que sea mayor o igual al precio del café.

```
no da vuelta la máquina de café
tengo n monedas con las denominaciones d1,....,dn
el café cuesta C

menor monto posible que sea mayor o igual al precio del café?

cafe(i,j) = "minimo pago >= j usando las monedas 1,...,i"

llamada principal ->  cafe(n,C)

cafe(i,j) =
      j=0     -> 0
      i=0,j>0 -> infinito
      i>0,j>0 -> min(cafe(i-1,j), di+cafe(i-1, max(0,j-di)))
```

el max(0,j-di) es porque sino tengo que hacer otra guarda para comparar i con d.

Programación dinámica:

las filas las llenamos de arriba hacia abajo, las columnas no importa el orden.

```
proc cafe(d: array[1..n] of nat, c: nat) ret r: nat
      var tabla[0..n,0..c]
      for i:=0 to n do
            tabla[i,0] := 0
      od
      for j:=0 to c do
            tabla[0,j] := +infinito
      od
      for i:=1 to n do
            for j:=1 to c do
                  tabla[i,j] := min(cafe[i-1,j], d[i] + cafe[i-1, j-d[i]
                                                                'max' 0])
            od
      od
      r := cafe[n,c]
end fun
```

---

2. (Backtracking) Luego de que te dan el alta por intoxicación, Malena te pide de nuevo que le cuides el departamento. Esta vez te deja N productos que no vencen pero los tenés que pagar. Cada producto i tiene un precio p_i y un valor nutricional s_i. Tu presupuesto es M. Se pide comer productos para obtener el máximo valor nutricional sin superar el presupuesto M. No hace falta comer todos los días ni vaciar la heladera.

   (a) Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.

   (b) Da la llamada o la expresión principal que resuelve el problema.

   (c) Definí la función en notación matemática.

Datos:

N productos que no vencen pero tengo que pagarlos.
Cada producto i tiene un precio pi y un valor nutricional si.
Mi presupuesto es M

Se pide comer productos para obtener el máximo valor nutricional sin superar el presupuesto M

alimentos(i,m) = "Máximo valor nutricional que obtengo comiendo los productos 1..i,, sin superar mi presupuesto m"

llamada principal -> alimentos(N,M)

Función matemática:

```
alimentos(i,m) =
            i=0     -> 0  {- no hay productos -}
            i>0,m<0 -> -infinito {- no tengo presupuesto -}
            i>0,m>0 -> max(alimentos(i-1,m), {-no como el producto i-}
                                 (si+alimentos(i-1,max(0,m-pi)) {-como el
producto i-}
```

dinámica:

```
fun alimentos(p: array[1..n] of int, v: array[1..n] of int, m: nat) ret
                                                                    res: nat
      var tabla[0..n,0..m]
.
.
.
```

---

2. (Backtracking) Te vas de viaje a la montaña viajando en auto k horas hasta la base de un cerro, donde luego caminarás hasta el destino de tus vacaciones. Tu auto no es muy nuevo, y tiene un stereo que solo reproduce cds (compact-disks). Buscás en tu vasta colección que compraste en los años 90 y tenés p cds, con $p > k$, que duran exactamente una hora cada uno. Encontrás también un cuaderno donde le diste una puntuación entre 1 y 10 a cada cd de tu colección. Cuanto mayor la puntuación, más es el placer que te da escucharlo. Dado que no sos tan exigente, querés que el puntaje promedio entre dos discos consecutivos que escuches, no sea menor a 6. Así por ejemplo si en la hora 2 escuchás un cd que tiene puntaje 8, en la hora 3 podrías escuchar uno que tenga puntaje al menos 4.

   Encontrar una combinación de cds para escuchar en las k horas de viaje, cumpliendo la restricción de que en dos horas consecutivas el puntaje promedio de los dos discos sea mayor o igual a 6, maximizando el puntaje total de los k discos que escucharás.

   Se pide lo siguiente:

   (a) Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.
   (b) Da la llamada o la expresión principal que resuelve el problema.
   (c) Definí la función en notación matemática.

```
cds: {nat}xnatxnat -> [cd]

cds(S,h,i) = "El máximo puntaje total con las puntuaciones de S, para
escuchar en h horas, si el promedio entre el primero e i tiene que ser mayor
o igual a 6"
S es el conjunto de puntuaciones
h es la cantidad de horas
i es la puntuación del cd anterior

llamada principal -> max c in C: c+cds(C-{c},k,c)

Función matemática ->

cds(S,h,i) =
            h=0 ó S={} -> 0
            h>0 y S no vacio -> Max c in S: (c+i)/2>=6 : c+cds(S-{c},
                                                              h-1,c)
```

Medias

```
medias: {nat} -> Num

madias(A) = "La puntualización de la mejor manera de aparear las medias de A"

madias(A) =
            A es vacío    -> 0
            A no es vacío -> Max i,j in A: i != j: P_ij + medias(A-{i,j})

llamada principal -> medias({1..n})
```

---

1. (Backtracking) No es posible correr una carrera de 800 vueltas sin reemplazar cada tanto las cubiertas (las ruedas) del auto. Como los mecánicos trabajan en equipo, cuando se cambian las cubiertas se reemplazan simultáneamente las cuatro. Reemplazar el set de cuatro cubiertas insume un tiempo T fijo, totalmente independiente de cuál sea la calidad de las cubiertas involucradas. Hay diferentes sets de cubiertas: algunas permiten mayor velocidad que otras, y algunas tienen mayor vida útil que otras, es decir, permiten realizar un mayor número de vueltas. Sabiendo que se cuenta con n sets de cubiertas, que $t_1, t_2, \ldots, t_n$ son los **tiempos por vuelta** que pueden obtenerse con cada uno de ellos, y que $v_1, v_2, \ldots, v_n$ es la vida útil medida en **cantidad de vueltas** de cada uno de ellos, se pide obtener un algoritmo que devuelva el tiempo de carrera mínimo cuando la carrera consta de m vueltas.

   Antes de dar la solución, especificá con tus palabras qué calcula la función recursiva que resolverá el problema, detallando el rol de los argumentos y la llamada principal.

Carrera de 800 vueltas.
Reemplazar el set insume un tiempo T fijo (totalmente independiente)
Se cuenta con n sets de cubiertas, que t1,...,tn son los tiempos por vuelta y v1,...,vn es la vida útil medida en cantidad de vueltas.

obtener un algoritmo que devuelva el tiempo de carrera mínimo cuando la carrera consta de m vueltas.

carrera(i,w) = “tiempo de carrera mínimo para hacer w vueltas, usando los neumáticos 1,...,i”

llamada principal: carrera(n,m) - T

función recursiva:

```
carrera(i,w) =
            w<=0 -> 0
            i=0, w>0 -> infinito
            i>0, w>0 -> min(carrera(i-1,w), {-No hay neumáticos-}
                        (T+(w min vi)*ti+carrera(i-1,max(0,w-vi))) {-uso el
            i-}
```

Si uso las cubiertas i, tengo el tiempo T fijo más el costo por vueltas

---

3. (Programación Dinámica) Dados $c_1, c_2, \ldots, c_n$ y la siguiente definición recursiva de la función m, para $1 \leq j \leq i \leq n$, escribí un programa que utilice la técnica de programación dinámica para calcular el valor de $m(n,1)$.

$$m(i,j) = \begin{cases} c_i & \text{si } i = j \\ m(i-1,j) + m(i,j+1) & \text{si } i > j \end{cases}$$

Ayuda: Antes de comenzar, hacé un ejemplo para entender la definición recursiva. Podés tomar por caso $n = 4, c_1 = 3, c_2 = 1, c_3 = 2, c_4 = 5$, y calcular $m(4,1)$.

llamada principal es m(n,1)

filas se llenan de arriba para abajo, y las columnas de derecha a izquierda

```
fun m(a: array[1..n] of nat) ret res: nat
      var tabla[1..n,1..n] of nat
      for i:=1 to n do
            tabla[i,i] := c[i]
      end
      for i:=1 to n do
```

```
            for n downto 1 do
                  if i > j then
                        tabla[i,j] := tabla[i-1,j] + tabla[i,j+1]
                  fi
            od
      od
      res := tabla[n,1]
end fun
```

m(4,1) = ?

tenemos que 4>1 entonces:

```
m(4,1) = m(3,1) + m(4,2)
       = m(2,1) + m(3,2) + m(3,2) + m(4,3)
       = m(1,1) + m(2,2) + m(2,2) + m(3,3) + m(2,2) + m(3,3) +
                                                 m(3,3) + m(4,4)

      como i=j para todos los casos
      = c1 + c2 + c2 + c3 + c2 + c3 + c3 + c4
```

---

2. (Backtracking)

Se tiene un tablero de 9x9 con números enteros en las casillas. Un jugador se coloca en una casilla a elección de la primera fila y se mueve avanzando en las filas y moviéndose a una columna adyacente o quedándose en la misma columna. En cada movimiento, el jugador suma los puntos correspondientes al número de la casilla, pero nunca puede pisar una casilla de manera tal que el puntaje acumulado, contando esa casilla, dé un valor negativo. El juego termina cuando se llega a la novena fila, y el puntaje total es la suma de los valores de cada casilla por la que el jugador pasó.

Se debe determinar el máximo puntaje obtenible, si es que es posible llegar a la última fila.

Se pide lo siguiente:

(a) Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.

(b) Da la llamada o la expresión principal que resuelve el problema.

(c) Definí la función en notación matemática.

m(f,c,p) = "máximo puntaje bajando por la casilla ij teniendo p puntos, contando los ij.

llamada principal => max(m(1,c,P(1,c))

```
m(f,c,p) =
            p < 0 -> -infinito
            p >=0 -> max(m(f+1,c-1,p+P(f+1,c-1)),
                         m(f+1,c,p+P(f+1,c))
                         m(f+1,c+1,p+P(f+1,c+1)))
```

truco en dinámica -> tabla[1..9,1..9,-1..M] ya que los puntos pueden ser negativos y en ese caso cuando la dimensión de M sea -1 lo mandó directamente a -infinito

Otra forma:

max m(9,c) con c in {1..9}

```
m(f,c) =
      i=1,P(i,j)>=0 -> p(f,c)
      i=1,P(i,j)<0  -> -infinito
      i>0           -> max(m(f-1,c-1),m(f-1,c),m(f-1,c+1)) + p(f,c) >0
```

---

2. (Backtracking) Como sabés que esta medianoche aumentarán todos los precios, decidís gastar hoy mismo la mayor parte posible del dinero D de que disponés. Hay n objetos para comprar, cuyos precios hoy son $v_1, v_2, \ldots, v_n$. Suponemos que $D < \sum_{i=1}^{n} v_i$, por lo que deberás elegir cuáles objetos comprar, es imposible comprar todos. Se debe determinar el máximo monto que podés gastar sin exceder el dinero D disponible.

   Se pide lo siguiente:

   (a) Especificá precisamente qué calcula la función recursiva que resolverá el problema, indicando qué argumentos toma y la utilidad de cada uno.

   (b) Da la llamada o la expresión principal que resuelve el problema.

   (c) Definí la función en notación matemática.

```
Se dispone de D
hay n objetos para comprar, cuyos precios hoy son v1,v2,...,vn
Suponemos que D < sum vi,

Determinar el máximo monto que podés gastar sin exceder el dinero D
disponible

compras(i,j) = “máxima cantidad de productos del conjunto 1..i que puedo
comprar sin pasarme del monto j”

llamada principal => compras(n,D)

compras(i,j) =
            i=0, j>0  -> 0
            j=0       -> -infinito
            i>0, vi>j -> compras(i-1,j)
            i>0, vi<j -> max(vi+compras(i-1,j-vi), compras(i-1,j))


fun comprar(v: array[1..n] of nat, D: nat) ret res: nat
      var tabla[0..n,0..D] of nat
      .
      .
```

Final 2021/12/01

2. Finalmente tenés la posibilidad de irte N días (con sus respectivas noches) de vacaciones y en el recorrido que armaste, cada día/noche i estarás en una ciudad C_i. Contás con M pesos en total de presupuesto para gastar en alojamiento y para cada ciudad conocés el costo k_i por noche del único hotel que tiene. Cada noche i podés elegir entre dormir en el hotel de la ciudad, lo que te costará k_i, o dormir en una carpa que llevaste, que te cuesta 0. Además, tenés una tabla que indica para cada ciudad i, la puntuación p_i del hotel.

   Se debe encontrar la máxima puntuación obtenible eligiendo en qué ciudades dormirás en hotel, de manera tal que el presupuesto total gastado no supere el monto M. Notar que si decidís dormir en carpa en alguna ciudad, la puntuación correspondiente para la misma será 0.

   (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.

N dias de vacaciones
dia/noche i estarás en una ciudad Ci
Tengo M pesos en total de presupuesto para gastar en alojamiento y para cada ciudad conoces el costo ki por noche del único hotel que tiene.
Cada noche i podes elegir entre dormir en el hotel de la ciudad eso cuesta ki o dormir en una carpa eso cuesta 0

tabla que indica para cada ciudad i, la puntuación pi del hotel.

Maxima puntuación obtenible eligiendo en que ciudades dormiras en horel,, de manera tal que el presupuesto no supera el monto M.

vac(i,j) = “Máxima puntuación obtenible durmiendo en hotel en las ciudades que van desde 1..i, sin pasarme del presupuesto j”

llamada principal -> vac(n,M)

```
vac(i,j) =
      i=0           -> 0
      i>0,j=0       -> -infinito
      i>0, ki >= j  -> vac(i-1,j)
      i>0, ki < j   -> max(vac(i-1,j),
                           pi + vac(i-1,j-ki))
```

Programación dinámica:

tabla es [0..N,0..M]

Las filas se llenan de arriba hacia abajo y las columnas de izquierda a derecha.

```
fun vac(p: array[1..n] of nat, k: array[1..n] of nat, M, N: nat) ret
                                                                    res:nat
      var tabla[0..n,0..M]

      for j:=0 to M do
            tabla[0,j] := 0
      od
      for i:=0 to n do
            tabla[i,0] := -infinito
      od
      for i:=1 to n do
            for j:=1 to M do
                  if k[i] > j then
                        tabla[i,j] := tabla[i-1,j]
                  else
                        tabla[i,j] := max(tabla[i-1,j],
                                     p[i]+tabla[i-1,j-k[i]])
                  fi
            od
      od
      res := tabla[n,M]
end fun
```