

### Tarea 3

1) Usaremos programación dinámica

para facilitar las cosas, diremos que la puntuación que obtenga el rival sera puntuación negativa para nosotros, entonces lo que buscamos es maximizar nuestra puntuación (al final veremos como obtener la puntuación real)

① caracterizar estructura de la solución óptima

Tenemos el numero  $d_1 d_2 d_3 d_4 \dots d_n$ , entonces la puntuación máxima de los valores  $d_1 a d_n$  es tomar el máximo de dos casos:  
tomar el valor  $d_1$  y restarle la puntuación máxima del rival de  $d_2 a d_n$   
o tomar el valor  $d_n$  y restarle la puntuación máxima del rival de  $d_1 a d_{n-1}$

② definir recursivamente el valor de la solución óptima

queremos la puntuación máxima de  $d_i a d_j$ , entonces es el mayor de entre  $d_i - \text{maxima puntuación de } d_{i+1} a d_j$  y  $d_j - \text{maxima puntuación de } d_i a d_{j-1}$ , entonces

$$a[i, j] = \begin{cases} d_i & \text{si } i=j \\ \max(d_i - a[i+1, j], d_j - a[i, j-1]) & \text{(ya que solo hay un elemento)} \end{cases}$$

(3) calcular solución óptima de abajo hacia arriba

para esto tenemos 2 casos

i		di
j		

si  $i = j$

i		x
j		

donde  $x$  es el máximo de  
 $di - c$  y  $dj - e$

y para esto usamos una matriz de  $n \times n$ , vamos iterando sobre  $i$  y  $j$

donde  $i$  va de 1 a  $n$  y  $j$  va de  $i$  a  $n$ , es decir

vamos construyendo la matriz de abajo hacia arriba y de izq a derecha

(solo armamos el triángulo superior de la matriz)

(4) construir solución óptima

luego de llenar la matriz (el triángulo superior), nuestra respuesta

esta en la posición  $a[1][n]$  (arreglo indexado), ya que es lo

lo mejor de elegir  $d_1 \dots d_n$  mas la puntuación máxima del rival

(de los elementos que quedan), pero esto no es nuestra puntuación real,

calculemoslo : digamos que  $a[1][n] = P$ , la suma de  $d_i$  hasta

$d_n$  es  $T$  y la puntuación del oponente es  $O$ , entonces sabemos

que  $T = P + 2 * (\text{valor absoluto de } O)$  debido a que

como en  $P$  se resta la puntuación del oponente, es como si obtuviera dobles puntos, entonces tenemos  $T = P + 2 \cdot |O|$ , por lo tanto nuestra puntuación es  $P + |O|$  (ya que al inicio digimos que restaríamos la puntuación del rival) entonces haciendo cuentas

$$T = P + 2 \cdot |O| \rightarrow 2 \cdot |O| = T - P \rightarrow |O| = \frac{T - P}{2}$$

$$\rightarrow |O| + P = \frac{T - P}{2} + P \rightarrow |O| + P = \frac{T + P}{2}, \text{ entonces}$$

nuestra puntuación real es  $\frac{a[1][n] + (d_1 + d_2 + \dots + d_n)}{2}$

Todo este algoritmo toma  $O(n^2)$ , ya que construir la matriz nos toma  $O(n^2)$  (rellenar cada casilla) y calcular la puntuación nos toma  $O(n)$  (ya que sumamos  $d_1 + d_n$ ), por lo tanto todo es  $O(n^2)$

Ejemplo con  $N = "19677"$

$i$   
 $j$   
 $1 \ 2 \ 3 \ 4 \ 5$

1	1	8	-2	9	-2
2		9	3	8	3
3			6	1	6
4				7	0
5					7

1 2 3 4 5

$$1 + 9 + 6 + 7 + 7 = 30$$

$$a[1][5] = -2$$

$$\frac{30 + -2}{2} = \frac{28}{2} = 14$$

puntuación

14

2) primero veamos cuantos posibles caminos existen, entonces sabemos que el robot necesita tomar  $X$  movimientos a la derecha y  $Y$  hacia abajo para un total de  $X+Y$  movimientos, necesitamos saber de cuantas formas puede elegir esos  $X$  movimientos ( $o Y$ ). , por lo tanto

usamos el coeficiente binomial  $\binom{X+Y}{X}$  ( $\circ \frac{X+Y}{Y}$ , de lo mismo)

$$\text{que nos da } \binom{X+Y}{X} = \frac{(X+Y)!}{X! (X+Y-X)!} = \frac{(X+Y)!}{X! Y!}$$

entonces hay  $\frac{(X+Y)!}{X! Y!}$  posibles caminos

ahora veamos el algoritmo  $O(XY)$ , usaremos programación dinámica

① caracterizar estructura de la solución óptima

en la posición  $(X-1, Y-1)$  del arreglo tendremos el numero de caminos hacia ahí,

en caso que si exista (dfo), entonces apuntaremos a la posición anterior

del camino ( $\leftarrow$ , movimiento derecha o  $\uparrow$  movimiento abajo)

② definir recursivamente el valor de la solución óptima

queremos el camino desde  $(0,0)$  hasta  $(i,j)$ , entonces checamos

si existen caminos hacia  $(i-1, j)$  y  $(i, j-1)$  (diferentes a 0), entonces

Si existen ambos pasos en  $(i, j)$  hacia abajo y apanteados

hacia  $\leftarrow$ , si solo existe uno ponemos en el nombre de camino y apanteados hacia

ese camino, si no existe ningún camino entorno ponemos 0.

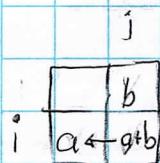
{ Caso especial si  $i-1 = -1$  o  $j-1 = -1$ , entonces los consideramos

como que no existen), el  $a[0][0]$  es 1, a menos que este bloqueado,

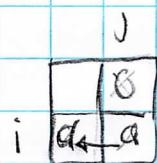
entorno  $a[i, j] = \begin{cases} a+b & \text{con } a[i-1, j] = b \text{ y } a[i, j-1] = a \\ 0 & \text{si está bloqueado} \\ 1 & \text{si } a[0][0] \text{ (en caso de no estar bloqueado)} \end{cases}$  (aparte de que existe)

③ calcular solución óptima de abajo hacia arriba

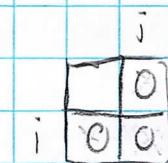
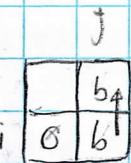
para esto tenemos 3 casos



si existen ambos  
caminos



existe 1 solo camino



no existe ningún camino o bloqueado

Considerar los  
casos especiales.  
renombrados  
arriba)

y para esto usamos una matriz de  $X \times X$ , al inicio creamos la

matriz y ponemos todos en blanco, luego marcamos con  $X$  las

celdas bloqueadas, luego iteramos sobre  $i$  y  $j$  donde  $i$  va

de 0 a  $n-1$  y  $j$  va de 0 a  $n-1$ , construir la matriz

de izquierda a derecha y arriba a abajo , ignorando las posiciones que marcamos con X (las bloqueadas)

#### ④ construir solución óptima

Llego de llenar la matriz , nuestra respuesta esta en la posición

a $[X-1][Y-1]$  (aresta 0 indexado) , entonces revisamos si tiene un 0

! o  $n \in \text{Nat}$  , si tiene 0 ese no existe un camino,

pero si es 1 , entonces vamos recorriendo las posiciones a las

que apuntamos ( $\leftarrow$  o  $\uparrow$ ) , obtenemos el camino partiendo posiciones  
y  $n \rightarrow$  el num de caminos

Todo el algoritmo toma  $O(XY)$  , ya que primero marcamos

las celdas bloqueadas esto toma a lo mas  $O(XY)$  (puede pasar

que todas estén bloqueadas) , luego construir la matriz toma

$O(XY)$  (rellena todo) , y por ultimo obtener el camino

nos toma  $O(X+Y)$  (hacemos X movimientos a la derecha y Y

abajo) , por lo tanto todo nos toma  $O(XY)$

ejemplo  $X=5$   $Y=4$ , con  $(4,1), (0,2), (3,1)$

$$\underbrace{y=j}$$

0 1 2 3

0	1	1	0	0
1	1	2	2	2
2	1	3	5	0
3	1	0	5	5
4	1	0	5	10

$\rightarrow (2,3)$  bloqueados

$$i=x$$

si existe el camino y es

$$(0,0) \downarrow (1,0) \downarrow (2,0) \rightarrow (2,1) \rightarrow (2,2)$$

$$\downarrow (3,2) \downarrow (4,2) \rightarrow (4,3)$$

y hay 10 posibles caminos

\* supongo en la redacción del ejercicio debía decir el camino de

$(0,0)$  a  $(X-1, Y-1)$  o de  $(1,1)$  a  $(X,Y)$ ,

yó use  $(0,0)$  a  $(X-1, Y-1)$  pero se arregla solo sumando

1 a los índices creo \*

3) Usaremos programación dinámica

① caracterizar estructura de la solución óptima

la solución será esta , revisamos las cadenas y si alguna

coincide (es igual) al final de la cadena D entonces necesitamos

1 más la longitud de la codificación del resto de la cadena D (lo que no coincide en la cadena , el inicio)

② definir recursivamente el valor de la solución óptima

queremos saber la longitud de la mejor codificación de  $D[1..i]$

(con  $i$  de 0 a  $n$ ) , entonces revisamos todas las cadenas y si una cadena

coincide con el final de  $D[1..i]$  entonces nos quedamos con el

mínimo entre la mejor longitud actual y la longitud de la mejor codificación

de  $D[1..i - \text{longitud(cadena)}]$  más 1 (de la cadena que coincide)

entonces el caso base es cuando  $i=0$  , que la longitud es 0 , ya

que es para la cadena vacía y al inicio todos los otros i tendrán

infinito o un número muy grande , usamos un arresto de tamaño

$n+1$  (0 indexado)

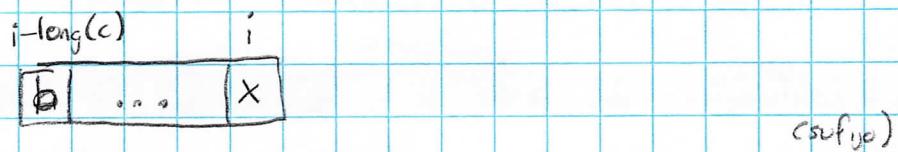
$c$  debe cumplir  $i \geq \text{leng}(c)$  (no pase el punto de la operación)

$$y D[i-\text{leng}(c), \dots i] == c$$

(es igual al sufijo de  $D[1 \dots i]$ )

$$a[i] = \begin{cases} \min(a[i], i + b[i] - \text{leng}(c)) & \text{dónde } c \text{ es una de las } m \text{ cadenas tal que coincide con el final de } D[1 \dots i] \\ \infty & \text{initialmente o si no encontramos cadena adecuada} \end{cases}$$

③ calcular solución óptima de abajo hacia arriba.



si encontramos una cadena  $c$ , que es igual al final de  $D[1 \dots i]$  entonces

$x$  es el mínimo de  $a[i]$  y  $a[b] + 1$ , donde  $b$  es  $i - \text{leng}(c)$

usamos un arreglo de tamaño  $n+1$  (0 indexado), ponemos

todos los valores como infinito, excepto  $a[0]=0$ , luego

iteramos sobre  $i$  ( $i$  va de 1 a  $n$ ) y vamos actualizando el arreglo

④ construir solución óptima

luego de actualizar el arreglo, la longitud estará en  $a[n]$

ya que es la longitud de la mejor codificación de  $D[1 \dots n]$ ,

entonces el valor de este ahí es la respuesta, si fuera infinito

es que no podemos codificar con esos  $m$  cadenas

Todo el algoritmo toma  $O(nmk)$  ya que iteramos con  $i$  de  $1 \dots n$  (esto es  $O(n)$ , sin contar el proceso dentro del ciclo) en cada iteración de  $i$ , revisamos todas las  $m$  cadenas y medimos su tamaño (que a lo más es  $k$ ) y comparar para ver si es sufijo de  $D[1..i]$ , por lo tanto esto toma  $O(k)$ , revisar todas las cadenas entonces es  $O(mk)$ , y como hacemos esto  $n$  veces entonces es  $O(nmk)$

ejemplo

$D = \text{bababbaababa}$   
 1 2 3 4 5 6 7 8 9 10 11 12

cadenas ( $a$ ,  $ba$ ,  $abab$ ,  $b$ )

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	1	2	2	2	3	3	4	5	5	4	5

resultado es (5)

i=1	i=4	i=7	i=10
cadena "b"	cadena "ba"	cadena "ba"	cadena "ba"
newa long es 1	newa long es 2	newa long es 3	newa long es 5
(a[0]+1)	(a[1]+1)	(a[5]+1)	(a[8]+1)
i=2	i=5	i=8	i=11
cadena "ba"	cadena "abab"	cadena "a"	cadena "abab"
newa long es 1	newa long es 2	newa long 4	newa long es 4
(a[0]+1)	(a[1]+1)	(a[7]+1)	(a[7]+1)
i=3	i=6	i=9	i=12
cadena "b"	cadena "b"	cadena "b"	cadena "a"
newa long es 2	newa long es 3	newa long 5	newa long 5
(a[1]+1)	(a[5]+1)	(a[8]+1)	(a[7]+1)

ahora si  $D = \text{pato}$

0 1 2 3 4  
| 0 | inf | inf | inf | inf |

resultado es no hay (ya que fin inf)

$i=1$

cadena no hay  
se queda inf.

$i=3$

cadena no hay  
se queda int

$i=2$

cadena "a"  
nueva long es inf

$i=4$

cadena no hay  
se queda int

(a[i]+1)

4) "Uno no se muere cuando debe, sino cuando puede"

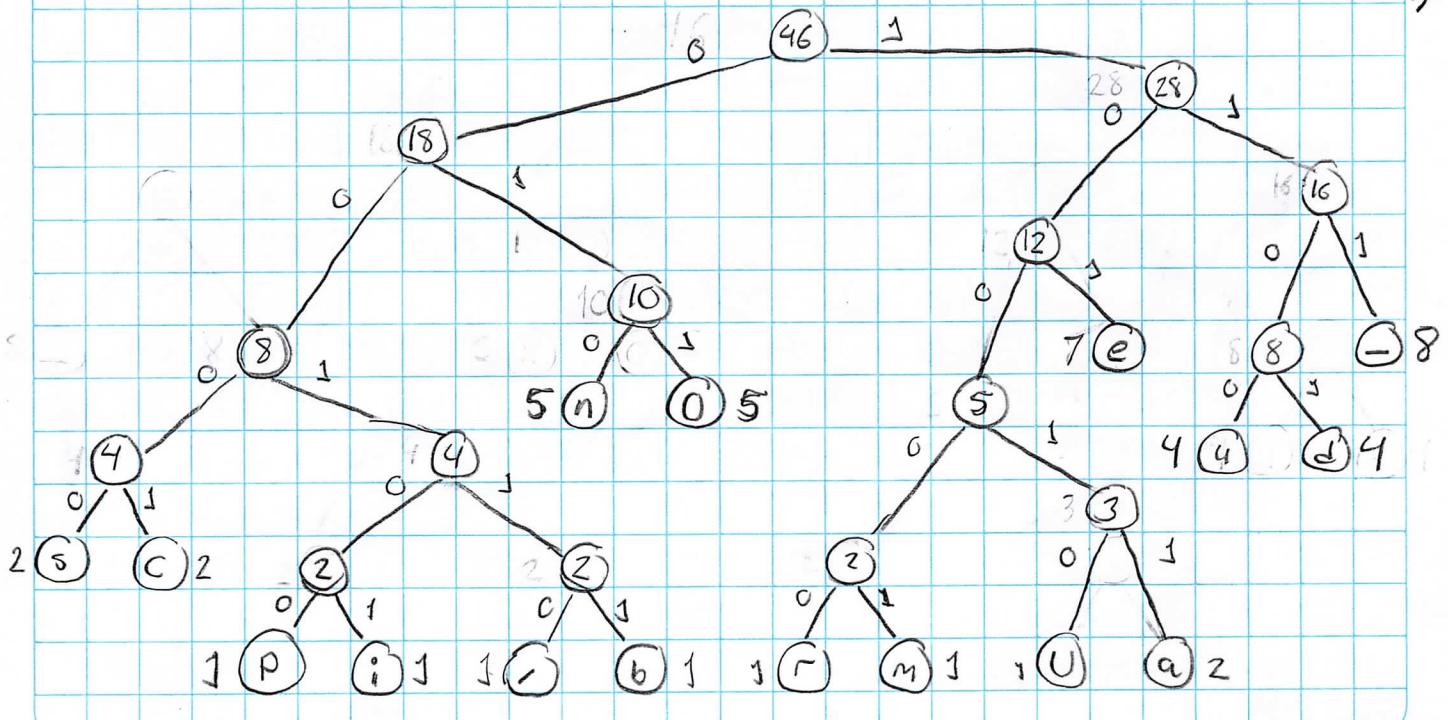
paso 1) hacer frecuencia de apariciones

$$\begin{array}{llll} U \rightarrow 3 & S \rightarrow 2 & r \rightarrow 1 & b \rightarrow 1 \\ n \rightarrow 5 & c \rightarrow 7 & C \rightarrow 2 & i \rightarrow 1 \\ o \rightarrow 5 & m \rightarrow 1 & a \rightarrow 2 & i \rightarrow 1 \\ - \rightarrow 8 & u \rightarrow 4 & d \rightarrow 4 & p \rightarrow 1 \end{array}$$

paso 2) ordenar frecuencias

$$\begin{array}{llll} - \rightarrow 8 & u \rightarrow 4 & a \rightarrow 2 & b \rightarrow 1 \\ e \rightarrow 7 & d \rightarrow 4 & U \rightarrow 1 & i \rightarrow 1 \\ n \rightarrow 5 & s \rightarrow 2 & m \rightarrow 1 & i \rightarrow 1 \\ o \rightarrow 5 & c \rightarrow 2 & r \rightarrow 1 & p \rightarrow 1 \end{array}$$

paso 3) construir arbol (las frecuencias de arriba son pesos, unir los nodos de menor frecuencia)



pass 4) codificación grupal

$$- = 111$$

$$u = 1100$$

$$a = 10011$$

$$b = 00111$$

$$e = 101$$

$$d = 1101$$

$$U = 10010$$

$$r = 00110$$

$$n = 010$$

$$s = 0000$$

$$m = 10001$$

$$i = 00101$$

$$o = 011$$

$$c = 0001$$

$$r = 10000$$

$$p = 00100$$

5) b) Primero veremos el algoritmo y luego veremos si las cadenas son o no shuffles

Usaremos programación dinámica, usaremos un arreglo a de tamaño  $(n+1) \times (m+1)$  (será 0 indexado), las cadenas X, Y, Z serán 1 indexadas

① caracterizar estructura de la solución óptima

si es shuffle o no lo sabremos si el último carácter de X es el mismo que el último de Z y  $Z[1\dots n+m-1]$  es shuffle de Y y  $X[1\dots n-1]$  o si el último carácter de Y es el mismo que el último de Z y  $Z[1\dots n+m-1]$  es shuffle de X y  $Y[1\dots m-1]$

② definir recursivamente el valor de la solución óptima

en la posición  $a[i][j]$  del arreglo está si los primeros  $i, j$  caracteres de Z ( $Z[1\dots i, j]$ ) son shuffle de los primeros  $i$  caracteres de X ( $X[1\dots i]$ ) y de los primeros  $j$  caracteres de Y ( $Y[1\dots j]$ ) tenemos un caso base, con  $a[0][0] = \text{True}$ , ya que es la cadena vacía

\* (y), ⓒ logicos \*

entonces

$$\text{True si } i=j=0$$

$$a[i][j] = \begin{cases} \text{True si } i=j=0 \\ (a[i-1][j] \text{ y } z[i+j] == x[i]) \circ (a[i][j-1] \text{ y } z[i+j] == y[j]) \end{cases}$$

pero hay unos casos especiales , la primera columna y fila (si no salimos de la matriz)

entonces

$$a[i][0] = \begin{cases} \text{True si } i=0 \\ (a[i-1][0] \text{ y } z[i] == x[i]) \end{cases}$$

$$a[0][j] = \begin{cases} \text{True si } j=0 \\ a[0][j-1] \text{ y } z[j] == y[j] \end{cases}$$

(3) calcular solucion optima de abajo hacia arriba

entonces

i

	b
i	a X

donde  $x$  es  $z[i+j] == x[i]$  y  $b$  o  $z[i+j] == y[j]$  y a

(recordar los casos especiales de la primera columna > fila)

entonces primero llenaremos la primera fila y columna de la matriz

y luego ya llenamos i de 1 a n y j de 1 a m , para llenar el resto de la matriz

como solo necesitamos la columna de arriba entonces en memoria podemos

usar solo dos columnas , por que sera mas eficiente con memoria

④ construir solución óptima

luego de llenar la matriz, la respuesta estará en  $a[9][5]$

si es True es que Z si es shuffle, en otro caso (False) no es shuffle

Todo el algoritmo toma  $O(nm)$ , ya que vamos recorriendo el arreglo para llenarlo, las comparaciones lógicas nos toman  $O(1)$  y obtener la respuesta toma  $O(1)$ , entonces al final es  $O(nm)$

a) ahora veremos el algoritmo para ver si son shuffle o no

Y r c h i p s

0 1 2 3 4 5

1 2 3 4 5 6 7 8 9 10 11 12 13 4

0 T T F F F F

Z = c c h o c o h i / a p t e s

1 T T T F F F

c 1 T T T F F F

2 F T F F F F

h 2 F T F F F F

3 F T F F F F

o 3 F T F F F F

4 F T F F F F

c 4 F T F F F F

5 F T T T F F

o 5 F T T T F F

6 F F F T F F

l 6 F F F T F F

7 F F F T T F

g 7 F F F T T F

8 F F F F T F

t 8 F F F F T F

9 F F F F T T

e 9 F F F F T T

entonces en  $a[9][5] = T$

por lo tanto si es shuffle

cchocohilaptes

*y*

	c	h	i	p	s									
	0	1	2	3	4	5								
x	0	T	T	T	F	F	F							
	1	T	F	F	F	F	F							
	2	T	F	F	F	F	F							
	3	T	T	F	F	F	F							
	4	T	F	F	F	F	F							
	5	T	T	T	T	F	F							
	6	F	F	F	T	F	F							
	7	F	F	F	T	F	F							
	8	F	F	F	T	F	F							
	9	F	F	F	F	F	(F)							

↑

1 2 3 4 5 6 7 8 9 10 11 12 13 14

$z = \text{chocochochi} / \text{at} \text{ s p e}$

entonces  $a[9][5] = F$

por lo tanto no es shuffle

chocochochi / at s p e

6)

a) no siempre utilizar el mínimo numero de monedas

si tenemos que el cambio es 12 , el algoritmo gloton primero escoge a 50 , que es la mayor , luego a 1 y por ultimo { entonces usa 3 monedas , pero podemos solo usar 2 monedas , dos monedas de 6 , ya que  $6+6=12$

b) algoritmo eficiente , usaremos programacion dinamica

① caracterizar estructura de la solucion optima

el minimo numero de monedas sera el minimo de los siguientes valores  
si  $d_i < n$  entonces el valor es  $1 + \text{el minimo de monedas para } n - d_i$ ,  
entonces nos quedamos con el minimo de todos los posibles valores de las denominaciones

② definir recursivamente el valor de la solucion optima

usamos un arreglo de tamaño  $n+1$  , entonces en la posicion  $a[i]$   
esta el minimo numero de monedas para regresar el cambio de  $i$  unidades,  
y esto se obtiene con el minimo de  $a[i]$  y  $a[i - d_j]$

dónde  $d_j$  son cualquier denominación  $d_j \leq i$

entonces

$$a[i] = \begin{cases} \infty & \text{initialmente} \\ 0 & \text{si } i=0 \\ 1 + \min(a[i], a[i-d_j]+1) & \text{para todos los } d_j \leq i \end{cases}$$

al inicio todos los valores son infinitos y  $a[0]=0$

③ calcular solución óptima de abajo hacia arriba

entonces

$i-d_j$	$i$
$b$	$x$

si encontramos una denominación  $d_j \leq i$  entonces  $x$  es el mínimo

de  $a[i]$  y  $a[b]+1$ , con  $b = i-d_j$ , entonces para cada  $i$  tenemos que revisar todas las denominaciones menores o iguales a  $i$ ,

entonces iteramos  $i$  de 0 a  $n$  y actualizamos el arreglo

④ construir la solución óptima

Luego de actualizar el arreglo, el mínimo número de monedas estará

en  $a[n]$ , ya que por como construimos el arreglo en la posición  $i$

esta el mínimo num de monedas para  $i$  unidades

extra podemos guardar guardar en  $a[i]$  la denominación que uso el mínimo y de esa forma obtener las monedas que se usaron

Oscemos todos los posibles  $d_j$ 's

Todo el algoritmo toma  $O(nk)$  ya que vamos calculando el

mínimo número de monedas para los valores de 1 a  $n$ , y para

cada valor  $i$  tenemos que checar, en el peor caso, con todas las

$k$  denominaciones, por lo tanto checar con cada denominación toma  $C(k)$

y como hacemos esto para  $n$  valores, todo nos toma  $O(nk)$

No creo que sea muy óptimo, ya que con  $n$  muy grandes se vuelve

lento, pero obtiene lo que queremos

Ejemplo

$$d = \{1, 6, 10\}$$

$$n = 12$$

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	1	2	3	4	1	2	2

resultado es (2) monedas

$i = 1$ moneda 1 nuevo min es 1 $(a[0]+1)$	$i = 4$ moneda 1 nuevo min es 4 $(a[3]+1)$	$i = 7$ moneda 1 nuevo min es 2 $(a[6]+1)$	$i = 10$ moneda 10 nuevo min es 1 $(a[0]+1)$	v son monedas 6 y 6 (la de $i=12$ ) (la de $i=6$ ) (12-6)
$i = 1$ moneda 1 min es 2 $(a[1]+1)$	$i = 5$ moneda 1 min es 5 $(a[4]+1)$	$i = 8$ moneda 1 min es 3 $(a[7]+1)$	$i = 11$ moneda 1 min es 2 $(a[10]+1)$	
$i = 3$ moneda 1 min es 3 $(a[2]+1)$	$i = 6$ moneda 6 min es 1 $(a[5]+1)$	$i = 9$ moneda 1 min es 4 $(a[8]+1)$	$i = 12$ moneda 6 min es 2 $(a[6]+1)$	

7) a) algoritmo voraz de tiempo  $\Theta(n \log n)$

el algoritmo sera este , primero ordenamos los tesoros

(solo merge sort - pero los ordenamos de mayor a menor )

y para comparar dos tesoros i y j - diremos que el mayor sera  
el mayor de comparar  $\frac{v_i}{w_i}$  y  $\frac{v_j}{w_j}$

Luego de tener ordenados los tesoros , llenamos la mochila vorazmente

tendremos un booleano  $b = \text{false}$  (que nos indica si la mochila esta llena)

variable V y variable i que inicia en 0 (cuantos elementos tiene la mochila )

entonces iteramos i hasta n (cuando es n , nos detenemos) , entonces

en cada iteracion tomamos el  $i+1$  tesoro y revisamos

si su peso w es menor o igual a M , (capacidad de carga) , si

pasa entonces lo metemos a la mochila , incrementamos a i ,

le restamos el peso a M , sumamos su valor a V y seguimos

iterando sobre i o si  $b = \text{True}$  entonces ya terminamos ,

pero si w es igual o mayor a M , entonces calculamos X (la fraccion

que cabe) ,esta se calcula  $X = \frac{w}{M}$  , entonces metemos la

x parte del tesoro , incrementando i , ponemos b=true ,

le restamos  $\frac{w_i}{x}$  a M , de sumamos  $\frac{v_i}{x}$  a V y seguimos.

Iterando así o hasta que b=false (que es lo que pasa)

al final del algoritmo tendremos en nuestra mochila los tesoros para maximizar el valor

Veamos la complejidad del algoritmo , lo primero es

ordenar usando mergesort , sabemos que mergesort siempre toma  $O(n \log n)$  por lo tanto en el caso promedio es  $\Theta(n \log n)$ ,

luego meter los tesoros a la mochila nos toma  $O(h)$  ya que

recorremos todos los tesoros y como podemos ver solo es

recorrer los tesoros que son n, notaremos ninguna constante extra,

entonces es  $\Theta(n)$  , por lo tanto la parte más compleja es

ordenar y todo toma  $\Theta(n \log n)$

o) probar el algoritmo

el algoritmo se divide en dos partes ordenar y meter los tesoros a la mochila, para ordenar usamos merge sort entonces sabemos que es correcta (no quiero ni se demostren merge sort uno) entonces falta ver que el ciclo de meter los tesoros sea correcto, para eso usaremos esta invariant: "en la mochila tenemos los  $i$  tesoros que maximizan nuestro valor respecto al peso  $\frac{\text{valor}}{\text{peso}}$ " entonces veamos que se cumple la invariant antes, durante y luego del ciclo, primero antes del ciclo, tenemos que  $i=0$  entonces en la mochila tenemos los 0 tesoros que maximizan nuestro valor respecto al peso" esto se cumple trivialmente ya que la mochila esté vacía, ahora durante el ciclo supongamos que al final de la iteración  $j$  se cumple ( $i=j$ ) entonces tenemos que en la mochila están los  $j$  tesoros que maximizan el valor/peso, entonces tomamos el  $j+1$  tesoro y revisamos si su peso es menor al espacio que nos queda caso 1) es menor al peso entonces retomamos al tesoro a la mochila

y actualizamos las variables necesarias, ahora veamos si "en la mochila tenemos los  $j+1$  tesoros que maximizan valor/peso" entonces veamos que el tesoro  $j+1$  es el  $\leftarrow$  máximo en relación valor/peso de los tesoros que quedan, por lo tanto al meterlo en la mochila tenemos los  $j+1$  primeros elementos que ordenamos que sería los que maximizan, ya que si un tesoro  $k$ ,  $k < j+1$ , maximiza el valor/peso, entonces tendría que estar antes que el tesoro  $j+1$  en la ordenación pero esto no puede pasar ya que merge sort es correcto

caso 2) el peso es mayor o igual al espacio resta, entonces metemos a la mochila la  $x$  parte del tesoro  $j+1$  y de esta manera tenemos en la mochila tenemos los  $j+1$  tesoros que maximizan el valor/peso" (actualizamos las variables necesarias), y es el elemento correcto ya que si existiera un tesoro  $k$  tal que  $\frac{k}{x}$  cabe en la mochila y maximiza, entonces debería ir antes en la ordenación, pero esto no pasa como vemos en el caso de arriba y además  $\frac{j+1}{x} > \frac{k}{x}$  ya que no usamos fracciones negativas por lo tanto la invariante se mantiene al final de la iteración  $j+1$

por ultimo veamos al final del ciclo , el ciclo se detiene si

$i = n$  o  $b = \text{true}$

caso 1)  $i = n$  , entonces significa que ya retiramos a todos los tesoros

a la mochila por lo tanto trivialmente se cumple "en la mochila tenemos los  $n$  tesoros que maximizan  $\text{valor/peso}$ "

caso 2)  $b = \text{true}$  , significa que la mochila ya no tiene mas espacio

entonces tenemos  $i$  objetos en la mochila y sabemos que son los

$i$  primeros (que es una fraccion) de nuestra ordenacion por lo tanto maximizan

ya que son los mayores, si no fuera asi la ordenacion esta mal como

vimos antes , pero no puede pasar , entonces "en la mochila tenemos

los  $i$  tesoros que maximizan  $\text{valor/peso}$ "

por lo tanto la invariant se mantiene  $\rightarrow$  la invariant es valida

entonces el ciclo es correcto , ahora al final del ciclo

tenemos que todos los tesoros estan (en este caso si cumple b que

buscamos trivialmente) a la mochila estan llena , en este caso

retiramos en la mochila los elementos mayores en relacion  $\text{valor/peso}$

entonces en V siempre tendremos el mejor valor posible

por lo tanto son los tesoros más eficientes para tener el mayor valor en la mochila, entonces nuestro algoritmo es correcto y cumple con lo que queremos

- ) algoritmo en  $\Theta(n)$

el algoritmo va a ser similar al primero pero esta vez no ordenaremos a los tesoros, para identificar a cada tesoro y compararlos usaremos su ~~valor~~ peso, luego usaremos algo como medir de medianas (el que vimos la clase 8/9/23) pero en vez de encontrar el k-ésimo elemento al ver los elementos en el conjunto menor, usaremos otra condición para detenernos, entonces veamos como será el algoritmo, primero encontraremos un buen pivote, para esto dividiremos a los tesoros en grupos de tamaño 5 y luego encontraremos la media de cada grupo, y por último usaremos de pivote a la mediana de las medianas (como en el algoritmo visto en la clase), entonces ahora con el pivotes, dividiremos a, dividiremos los tesoros en 3 conjuntos, el conjunto  $M_1$  serán los tesoros i tales que

$\frac{v_i}{w_i}$  es menor que  $\frac{v_a}{w_a}$ , ademas tenia una variable  $W_M$  que es el peso total de los tesoros del conjunto (al ir formando el conjunto se va actualizando), el conjunto  $M_A$ , seran los tesoros i tales que  $\frac{v_i}{w_i}$  es mayor que  $\frac{v_a}{w_a}$ , con  $W_M$  el peso total del conjunto y el ultimo conjunto  $I$  seran los tesoros i tal que  $\frac{v_i}{w_i} = \frac{v_a}{w_a}$  con  $W_I$  el peso total del conjunto, entonces luego de usar nuestro pivote para crear nuestros 3 conjuntos, restemos nuestra condicion de detenernos, si  $W_M > M$  (si el peso de los tesoros mas grandes que el pivote son mas pesados que nuestra capacidad), si esto pasa volvemos a buscar un pivote y hacer conjuntos pero ahora solo en el conjunto  $M_A$ , pero si no pasa entonces vamos a meter tesoros, como en el primer algoritmo, primero metemos los tesoros del conjunto  $M_A$  y luego del conjunto  $I$ , ahora si la mochila se llena (recordando que el primer elemento que no cabe completo lo partimos para que si tenga espacio) entonces ya terminamos, pero si los conjuntos  $M_A$  y  $I$  estan vacios > aun tenemos espacio entonces volvemos

a repetir el proceso de encontrar pivot y conjuntos pero ahora usando el conjunto  $M_i$  (recordando actualizar cuánto espacio queda en la mochila) y de esta manera terminamos de llenar nuestra mochila.

El algoritmo funciona ya que es similar a como fue el primer algoritmo, solo que no ordenamos esta vez, pero seguimos respetando que primero metemos los elementos más grandes que el pivote, pero asegurandonos que el peso de los mayores sea menor a la capacidad de la mochila, pero así no meter al menor de los tesoros mayores, y luego el espacio que queda en la mochila se resuelve recursivamente, además cuando  $W_{M_i} > M$ , descartamos los menores ya que podemos llenar la mochila con los mayores y por eso la recursión en  $M_i$ .

Ahora veamos la complejidad / como usamos la mediana de medianas (o algo muy muy similar) entonces en clase vimos que es  $O(n)$ , y esto es que cuando hacemos recursión, encontramos la mediana de medianas, y hacer los conjuntos nos toma  $O(n)$ , luego restaremos

si volvemos a hacer recursion , en promedio eliminamos a la mitad de tesoros y por lo tanto cada vez que repetimos buscar el pivote y todo eso solo lo hacemos en la mitad de tesoros y asi hasta acabar , entonces en promedio esto nos toma  $O(n)$  (con una constante no muy grande) , como vimos en clase (8/9/23) y ahora cuando metemos los tesoros a la mochila en el peor caso toma  $O(n)$  (toma todos) , por lo tanto si juntamos todo lo del pivote , recursion y meter elementos nos toma  $O(n)$  , entonces en el caso promedio nos toma  $O(n)$  , con una constante no muy grande y por lo tanto condicione que es  $\Theta(n)$  , espero un v

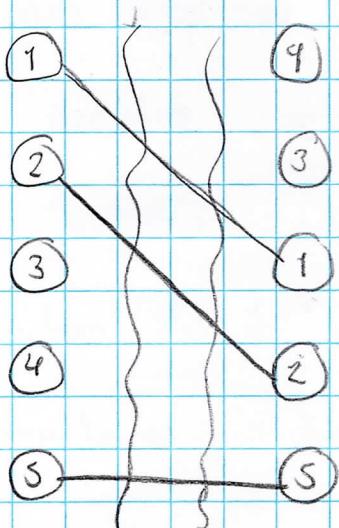
8) para esto usaremos el algoritmo para obtener la subsecuencia común más larga que vimos en clase, entonces usamos las  $n$  ciudades del lado izquierdo como la cadena  $X$ , y las del lado derecho como  $Y$ , entonces usamos programación dinámica para encontrar la subsecuencia común más larga (visto en clase 20 y 22 agosto), entonces el numero que este en  $a[n, m]$  sera el mayor numero de puentes entre las ciudades con mismo numero, si obtenemos la subsecuencia podemos ver que puentes van a ser y esto funciona que las ciudades las podemos ver como cadenas, y al obtener la subsecuencia común más larga, podemos encontrar si ciudades están en el mismo orden (están abajo de otras) en ambos lados del río, de este modo sabemos que si ponemos los puentes de estas ciudades con el mismo orden, sabemos que nunca se cruzan (tienen mismo orden en ambos lados del río) y si pusieramos algún puente de una ciudad que no estuviera la subsecuencia, nos pasaria que cruzaría algun

punto, ya que las demás ciudades no tienen el mismo orden o posición en ambos lados del río

Todo lo anterior toma  $O(nm)$  en espacio (con  $n$  la longitud de X (el número de ciudades de un lado) y  $m$  la longitud de Y (el número de ciudades del otro lado)), y  $O(1m)$  para llenar la matriz de la programación dinámica y  $O(\max\{m, n\})$  para construir la subsecuencia, como se vio en la clase 22/9/23

ejemplo

subsecuencia



	4	3	1	2	5
1	0	0	0	0	0
2	0	0	1	1	1
3	0	1	1	1	2
4	0	1	1	1	2
5	0	1	1	1	3

entonces son ③ puentes  
y los puentes son  
de las ciudades 1, 2 y 5

LCS = 1, 2, 5

tamaño 3