Teórica 2: Técnicas de diseño de algoritmos

1. Técnicas de diseño de algoritmos - 2da parte

1.1. Algoritmos golosos

El método goloso es la técnica de diseño de algoritmos más simple. Generalmente los algoritmos golosos son utilizados para resolver problemas de optimización. Estos algoritmos son fáciles de desarrollar e implementar y, cuando ***funcionan***, son eficientes. Sin embargo, muchos problemas no pueden ser resueltos mediante esta técnica. En esos casos, proporcionan ***heurísticas*** sencillas que en general permiten construir soluciones razonables, pero sub-óptimas.

La idea es construir una solución paso por paso, de manera de hacer ***la mejor elección posible localmente*** según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección. Es decir, en cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras. Una decisión tomada nunca es revisada y no se evaluan alternativas.

Ejemplo 1. ***Problema de la mochila continuo:*** *Contamos con una mochila con una capacidad máxima C (peso máximo), donde podemos cargar n los objetos. Puede ser todo el objeto o una fracción de él, por eso lo de «continuo» en el nombre del problema. Cada elemento i, pesa un determinado peso p_i y llevarlo nos brinda un beneficio b_i.*

Queremos determinar qué objetos debemos incluir en la mochila, sin excedernos del peso máximo C, de modo tal de maximizar el beneficio total entre los objetos seleccionados. En la versión más simple de este problema vamos a suponer que podemos poner parte de un objeto en la mochila (continuo).

Un algoritmo goloso para este problema, agregaría objetos a la mochila mientras esta tenga capacidad libre. En líneas generales:

```
llenerMachila(C,n,P,B)
    entrada: C capacidad, n cantidad de elementos,
             P arreglo con los pesos, B arreglo con los beneficios
    salida: arreglo X, donde X[i] proporción del elemento i colocado en la mochila,
             benef beneficio obtenido

    ordenar los elementos según algún criterio
    mientras C > 0 hacer
        i ← siguiente elemento
        X[i] ← min(1, C/P[i])
        C ← C − P[i] × X[i]
        benef ← benef + B[i] × X[i]
    fin mientras
    retornar X,benef
```

Algunos criterios posibles para ordenar los objetos son:

- *ordenar los objetos en orden decreciente de su beneficio*
- *ordenar los objetos en orden creciente de su peso*
- *ordenar los objetos en orden decreciente según ganancia por unidad de peso* (b_i/p_i)

¿Sirven estas ideas? ¿Dan el resultado correcto?

Supongamos que $n = 5$, $C = 100$ *y los beneficios y pesos están dados en la tabla*

	1	2	3	4	5
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

Los resultados que obtendríamos ordenando los ítems según cada criterio son:

- *mayor beneficio* b_i*:* $66 + 60 + 40/2 = 146$.
- *menor peso* p_i*:* $20 + 30 + 66 + 40 = 156$.
- *maximice* b_i/p_i*:* $66 + 20 + 30 + 0{,}8 \cdot 60 = 164$.

Se puede demostrar que la selección según beneficio por unidad de peso, b_i/p_i*, da una solución óptima.*

Si los elementos deben ponerse completos en la mochila la situación es ***muy*** *diferente. Eso lo veremos más adelante.*

Ejemplo 2. ***Problema del cambio:*** *Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando monedas de 1, 5, 10 y 25 centavos. Por ejemplo, si el monto es* \$0,69*, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10 centavos, una de 5 centavos y 4 de un centavo.*

Un algoritmo goloso para resolver este problema es seleccionar la moneda de mayor valor que no exceda la cantidad restante por devolver, agregar esta moneda a la lista de la solución, y sustraer la cantidad correspondiente a la cantidad que resta por devolver (hasta que sea 0).

```
darCambio(cambio)
    entrada: cambio ∈ ℕ
    salida: M multiconjunto de enteros

    suma ← 0
    M ← {}
    mientras suma < cambio hacer
        proxima ← masgrande(cambio,suma)
        M ← M ∪ {proxima}
        suma ← suma + proxima
    fin mientras
    retornar M
```

Este algoritmo siempre produce la mejor solución, es decir, retorna la menor cantidad de monedas necesarias para obtener el valor cambio. Sin embargo, si también hay monedas de 12 centavos, puede ocurrir que el algoritmo no encuentre una solución óptima: si queremos devolver 21 centavos, el algoritmo retornará una solución con 6 monedas, una de 12 centavos, 1 de 5 centavos y 4 de 1 centavos, mientras que la solución óptima es retornar 2

monedas de 10 centavos y una de 1 centavo.

El algoritmo es goloso porque en cada paso selecciona la moneda de mayor valor posible, sin preocuparse que ésto puede llevar a una mala solución, y nunca modifica una decisión tomada.

Ejemplo 3. ***Minimización del tiempo de espera en un sistema:*** *Un servidor tiene n clientes para atender. Se sabe que el tiempo requerido para atender al cliente i es $t_i \in \mathbb{R}_+$. El objetivo es determinar en qué orden se deben atender los clientes para minimizar la suma de los tiempos de espera de los clientes.*
Si $I = (i_1, i_2, \ldots, i_n)$ es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera (finalización de ser atendido) es

$$\begin{aligned} T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \ldots \\ &= \sum_{k=1}^{n} (n - k + 1) t_{i_k}. \end{aligned}$$

Un algoritmo goloso para resolver este problema, atendería al cliente que requiera el menor tiempo de atención entre todos los que todavía están en la cola. Retorna una permutación $I_{GOL} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$.

Este enfoque da un algoritmo correcto.

Ejemplo 4. *Sean $P_1, P_2, \ldots, P_n$ programas que se quieren almacenar en una cinta. El programa P_i requiere s_i Kb de memoria. La cinta tiene capacidad para almacenar todos los programas. Se conoce la frecuencia π_i con que se usa el programa P_i. La densidad de la cinta y la velocidad del drive son constantes. Después que un programa se carga desde la cinta, la misma se rebobina hasta el principio. Si los programas se almacenan en orden $i_1, i_2, \ldots, i_n$ la esperanza del tiempo de carga de un programa es*

$$T = c \sum_j \left(\pi_{i_j} \sum_{k \leq j} s_{i_k} \right)$$

donde la constante c depende de la densidad de grabado y la velocidad del dispositivo.

Un algoritmo goloso para determinar el orden en que se almacenan los programas que minimice T es el siguiente: para cada programa P_i calcular π_i / s_i y colocar los programas en orden no creciente de este valor.

Otras posibilidades son colocarlos en orden no decreciente de los s_i o en orden no creciente de los π_i, pero estos dos criterios NO resultan en algoritmos correctos.

1.2. Programación Dinámica

Esta técnica fue introducida en 1953 por Richard Bellman. Es aplicada típicamente a problemas de optimización combinatoria, donde puede haber muchas soluciones factibles, cada una con un valor (o costo) asociado y prentendemos obtener la solución con mejor valor (o menor costo). Pero además también resulta adecuada para algunos problemas de naturaleza recursiva, como el cálculo de los números combinatorios.

Programación dinámica, al igual que dividir y conquistar (D&C), divide al problema en subproblemas de tamaños menores, que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original. D&C resulta eficiente cuando los subproblemas son de igual tamaño o parecido y, además, no es necesario resolver más de una vez el mismo subproblema (o muy pocas veces

lo es).

Por el contrario, PD es adecuada para problemas que tienen estas características que le molestan a D&C, permitiendo reducir la complejidad computacional propia de una resolución puramente recursiva. Esta técnica evita resolver más de una vez un mismo subproblema en diferentes llamadas recursivas. Para esto mantiene una estructura de datos donde almacena los resultados que ya han sido calculados hasta el momento para su posterior reutilización si se repite una llamada a un subproblema. Entonces, cuando se hace una llamada recursiva a un subproblema primero se chequea si este subproblema ya ha sido resuelto. Si lo fue, se utiliza el resultado almacenado. En caso contrario, se realiza la llamada recursiva. Este esquema se llama ***memoización*** y es la principal ventaja de los algoritmos de programación dinámica. Por supuesto que esto tiene sentido cuando se necesita más de una vez la solución del mismo subproblema. Si estamos seguros que un subproblema no será llamado nuevamente, no es necesario que almacenemos la solución a este subproblema, ahorrando así espacio.

Si bien las soluciones de programación dinámica son pensadas como funciones recursivas top-down, algunas veces una implementación no recursiva bottom-up puede llegar a resultar en algoritmos con mejor tiempo de ejecución.

Ejemplo 5. ***Coeficientes binomiales:*** *Si $n \geq 0$ y $0 \leq k \leq n$, se define*

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Para calcular $\binom{n}{k}$, usaremos la siguiente propiedad que da una fórmula recursiva:

$$\binom{n}{k} = \begin{cases} 1 & si\ k = 0\ o\ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & si\ 0 < k < n \\ 0 & caso\ contrario \end{cases}$$

El algoritmo recursivo directo en base a esta propiedad es el siguiente:

Coeficientes binomiales recursivo

```
combiRecu(n,k)
   entrada: n, k ∈ N
   salida: (n k)

   si k = 0 o k = n hacer
         retornar 1
   else
         retornar combiRecu(n − 1,k − 1) + combiRecu(n − 1,k)
   fin si
```

Si ejecutamos este algoritmo con por ejemplo $n = 5$ y $k = 3$ realizaremos 18 llamadas recursivas: una con parámetros $n = 4$ y $k = 2$, una con $n = 4$ y $k = 3$, una con $n = 3$ y $k = 1$, dos con $n = 3$ y $k = 2$, una con $n = 3$ y $k = 3$, una con $n = 2$ y $k = 0$, tres con $n = 2$ y $k = 1$, dos con $n = 2$ y $k = 2$, tres con $n = 1$ y $k = 0$, y tres con $n = 1$ y $k = 1$. Como vemos, de las 18 en realidad fueron 9 llamadas diferentes, en las otras 9 volvimos a calcular algo que ya habíamos calculado.

Con esto en mente, la primera mejora que podemos realizar es guardar los valores de los subproblemas que vamos calculando para evitar repetir su cálculo. Para esto podemos mantener en una estructura T estos valores, la cual la

inicializamos con todas las posiciones en NULL.

Coeficientes binomiales recursivo con memoización (top-down)

```
combiRecu(n,k,T)
  entrada: n, k ∈ N
  salida: (n k)

  si T[n][k] = NULL hacer
      si k = 0 o k = n hacer
          T[n][k] ← 1
      else
          T[n][k] ← combirecu(n − 1,k − 1,T) + combirecu(n − 1,k,T)
      fin si
  fin si
  retornar T[n][k]
```

*Podemos **invertir** el proceso y armar una tabla con los números combinarios para valores más pequeños de n y k. Por la propiedad anterior, podemos inicializar la tabla con 1 para $k = 0$ y $k = n$:*

$n\backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1		1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Ahora, teniendo ya calculados $\binom{1}{0}$ y $\binom{1}{1}$, aplicando la propiedad calculamos $\binom{2}{1}$.

$n\backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	**1**	**1**						
2	1	2	1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Luego, teniendo $\binom{2}{0}$ *y* $\binom{2}{1}$*, calculamos* $\binom{3}{1}$*, y con* $\binom{2}{1}$ *y* $\binom{2}{2}$*, calculamos* $\binom{3}{2}$*.*

$n \backslash k$	*0*	*1*	*2*	*3*	*4*	$\cdots$	$k-1$	k
0	*1*							
1	*1*	*1*						
2	***1***	***2***	***1***					
3	*1*	*3*	*3*	*1*				
4	*1*				*1*			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	*1*						*1*	
k	*1*							*1*
$\vdots$	$\vdots$							
$n-1$	*1*							
n	*1*							

Y después, con $\binom{3}{0}$ *y* $\binom{3}{1}$*, calculamos* $\binom{4}{1}$*, con* $\binom{3}{1}$ *y* $\binom{3}{2}$*, calculamos* $\binom{4}{2}$ *y con con* $\binom{3}{2}$ *y* $\binom{3}{3}$*, calculamos* $\binom{4}{3}$*.*

$n \backslash k$	*0*	*1*	*2*	*3*	*4*	$\cdots$	$k-1$	k
0	*1*							
1	*1*	*1*						
2	*1*	*2*	*1*					
3	*1*	*3*	*3*	*1*				
4	*1*	*4*	*6*	*4*	*1*			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	*1*						*1*	
k	*1*							*1*
$\vdots$	$\vdots$							
$n-1$	*1*							
n	*1*							

Y así siguiendo...

Esta idea deriva en el siguiente pseudocódigo de un algoritmo bottom-up no recursivo:

Coeficientes binomiales

combinatorio(n,k)
entrada: $n, k \in N$
salida: $\binom{n}{k}$

para $i = 1$ **hasta** n **hacer**
    $A[i][0] \leftarrow 1$
fin para
para $j = 0$ **hasta** k **hacer**
    $A[j][j] \leftarrow 1$
fin para
para $i = 2$ **hasta** n **hacer**
    para $j = 2$ **hasta** $\texttt{min}(i-1,\texttt{k})$ **hacer**
        $A[i][j] \leftarrow A[i-1][j-1] + A[i-1][j]$
    fin para
fin para
retornar $A[n][k]$

Un algoritmo recursivo (sin memoria) para calcular el $\binom{n}{k}$ *sería* $\Omega(\binom{n}{k})$*. En cambio, el algoritmo de programación dinámica recién presentado tiene complejidad* $O(nk)$ *y necesita espacio* $O(k)$*, ya que sólo necesitamos almacenar la fila anterior de la que estamos calculando.*

Ejemplo 6. ***Multiplicación de*** n ***matrices:*** *Dadas* n *matrices,* $M_1, M_2, \ldots, M_n$*, queremos calcular*

$$M = M_1 \times M_2 \times \ldots M_n$$

Por la propiedad asociativa del producto de matrices, ésto puede hacerse de muchas formas. De todas estas posibles formas, queremos determinar la que minimiza el número de multiplicaciones necesarias. Éste es un problema de optimización combinatoria, de todas las posibilidades queremos ***la mejor****. Por ejemplo, si la dimensión de* A *es* 13×5*, la de* B *es* 5×89*, la de* C *es* 89×3 *y la de* D *es* 3×34*. Tenemos que:*

- $((AB)C)D$ *requiere* 10582 *multiplicaciones.*
- $(AB)(CD)$ *requiere* 54201 *multiplicaciones.*
- $(A(BC))D$ *requiere* 2856 *multiplicaciones.*
- $A((BC)D)$ *requiere* 4055 *multiplicaciones.*
- $A(B(CD))$ *requiere* 26418 *multiplicaciones.*

Por simplicidad vamos a calcular la cantidad de multiplicaciones mínima y no la forma de hacerlo en sí. Es sencillo modificar el algoritmo resultante para calcular la forma de hacer la multiplicación.

Para multiplicar todas las matrices de forma óptima, deberemos multiplicar las matrices 1 a i *por un lado y las matrices* $i+1$ *a* n *por otro lado y luego multiplicar estos dos resultados, para algún* $1 \leq i \leq n-1$*, que es justamente lo que queremos determinar. En la solución óptima de* $M = M_1 \times M_2 \times \ldots M_n$*, estos dos subproblemas,* $M_1 \times M_2 \times \ldots M_i$ *y* $M_{i+1} \times M_{i+2} \times \ldots M_n$ *deben estar resueltos, a su vez, de forma óptima, es decir realizando la mínima cantidad de operaciones.*

Llamaremos $m[i][j]$ *a la cantidad mínima de multiplicaciones necesarias para calcular* $M_i \times M_{i+1} \times \dots M_j$. *Por comodidad, supongamos que las dimensiones de las matrices están dadas por un vector* $d \in N^{n+1}$, *tal que la matriz* M_i *tiene* $d[i-1]$ *filas y* $d[i]$ *columnas para* $1 \leq i \leq n$, *entonces:*

- *Para* $i = 1, \dots, n$, $m[i][i] = 0$
- *Para* $i = 1, \dots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- *Para* $s = 2, \dots, n-1$, $i = 1, \dots, n-s$,

$$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

Y la solución del problema será $m[1][n]$.

Analicemos si la forma de ir calculando los valores de m es correcta. Esto es, que cuando quiera calcular un valor ya tenga calculado el valor para los subproblemas que necesito.

Los dos primeros ítems de la fórmula, no son recursivos, así que los podemos llenar desde el comienzo.

Para el último ítem, cuando querramos calcular $m[i][i+s]$, *vamos a necesitar ya tener calculados* $m[i][k]$ *y* $m[k+1][i+s]$ *para todo* $k = i, \dots, i+s-1$. *Entonces, con lo que ya calculamos desde el comienzo, podemos calcular* $m[i][i+2]$ *(es decir* $s = 2$*) para todos* $i = 1, 2, \dots, n-2$ *(porque los únicos valores que puede tomar* k *son* i *e* $i+1$ *y ya tenemos a* $m[i][i]$, $m[i+1][i+2]$, $m[i][i+1]$ *y* $m[i+2][i+2]$*).*

Ahora, ya teniendo estos valores, podemos calcular $m[i][i+3]$ *(es decir* $s = 3$*) para todos* $i = 1, 2, \dots, n-3$ *(porque los únicos valores que puede tomar* k *son* $i+1$ *y* $i+2$ *y ya tenemos a* $m[i][i+1]$, $m[i+2][i+3]$, $m[i][i+2]$ *y* $m[i+3][i+3]$*).*

Y así siguiendo. Entonces, para cada $s = 2, \dots, n-1$ *recorremos todos los* $i = 1, 2, \dots, n-s$.

Esto ya es el algoritmo:

Multiplicación de n matrices

```
mult(d)
  entrada: d ∈ N^{n+1} dimensión de las matrices
  salida: cantidad mínima de multiplicaciones

  para i = 1 hasta n hacer
      m[i][i] ← 0
  fin para
  para i = 1 hasta n − 1 hacer
      m[i][i + 1] ← d[i − 1] ∗ d[i] ∗ d[i + 1]
  fin para
  para s = 2 hasta n − 1 hacer
      para i = 1 hasta n − s hacer
          min ← ∞
          para k = i hasta i + s − 1 hacer
              si m[i][k] + m[k + 1][i + s] + d[i − 1] ∗ d[k]d[i + s] < min hacer
                  min ← m[i][k] + m[k + 1][i + s] + d[i − 1] ∗ d[k] ∗ d[i + s]
              fin si
          fin para
          m[i][i + s] ← min
      fin para
  fin para
  retornar m[1][n]
```

Este algoritmo es $O(n^3)$ y requiere $O(n^2)$ espacio para almacenar m.

Veamos otro ejemplo de aplicación de PD.

Ejemplo 7. *Subsecuencia común más larga: Dada una secuencia, una subsecuencia de ella se obtiene eliminando 0 o más símbolos (sin modificar el orden de los símbolos que quedan). Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.*

En el problema de la subsecuencia común más larga (scml) el objetivo es encontrar la subsecuencia común más larga de dos secuencias dadas. Es decir, dadas dos secuencias A y B, queremos encontrar entre todas las secuencias que son tanto subsecuencia de A como de B la de mayor longitud.

Por ejemplo, si $A = [9, 5, 2, 8, 7, 3, 1, 6, 4]$ y $B = [2, 9, 3, 5, 8, 7, 4, 1, 6]$ las scml es $[9, 5, 8, 7, 1, 6]$.

Si resolvemos este problema por fuerza bruta, listaríamos todas las subsecuencias de S_1, todas las de S_2, nos fijaríamos cuales tienen en común, y entre esas elegiríamos la más larga.

Por simplicidad en la escritura, vamos a plantear el problema donde queremos buscar la longitud de la scml y no la subsecuencia. Fácilmente se podría adaptar el algoritmo que desarrollaremos al problema original.

Dadas las dos secuencias $A = [a_1, \ldots, a_r]$ y $B = [b_1, \ldots, b_s]$, existen dos posibilidades, $a_r = b_s$ o $a_r \neq b_s$. Analicemos cada caso:

1. *$a_r = b_s$: La scml entre A y B se obitene colocando al final de la scml entre $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_{s-1}]$ al elemento a_r (o b_s porque son iguales).*

2. $a_r \neq b_s$**:** *La scml entre A y B será la más larga entre las scml entre $[a_1., \ldots, a_{r-1}]$ y la scml entre $[b_1., \ldots, b_s]$ y $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$. Esto es, calculamos el problema aplicado a $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_s]$ y, por otro lado, el problema aplicado a $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$, y nos quedamos con la más larga de ambas.*

Nuevamente, esta forma recursiva de resolver el problema ya nos conduce al algoritmo. Si llamamos $l[i][j]$ a la longitud de la scml entre $[a_1., \ldots, a_i]$ y $[b_1., \ldots, b_j]$, entonces:

- $l[0][0] = 0$
- *Para* $j = 1, \ldots, s$, $l[0][j] = 0$
- *Para* $i = 1, 2, \ldots, r$, $l[i][0] = 0$
- *Para* $i = 1, \ldots, r$, $j = 1, \ldots, s$

  si $a_i = b_j$**:** $l[i][j] = l[i-1][j-1] + 1$
  si $a_i \neq b_j$**:** $l[i][j] = \max\{l[i-1][j], l[i][j-1]\}$

Y la solución del problema será $l[r][s]$.

Veamos ahora cómo ir llenando los valores de l. Los casos no recursivos son nuestra base y los podemos calcular desde el comienzo.

Luego, con esto ya podemos calcular $l[1][1]$, porque solo necesita los casos base. Ahora ya estamos en condiciones de calcular $l[1][2]$ y $l[2][1]$. Siguiendo, podemos calcular $l[1][3]$, $l[2][2]$ y $l[3][1]$.

De esta forma, obtenemos el siguiente algoritmo:

Subsecuencia común más larga

```
scml(A,B)
  entrada: A, B secuencias
  salida: longitud de a scml entre A y B

  l[0][0] ← 0
  para i = 1 hasta r hacer
      l[i][0] ← 0
  fin para
  para j = 1 hasta s hacer
      l[0][j] ← 0
  fin para
  para i = 1 hasta r hacer
      para j = 1 hasta s hacer
          si A[i] = B[j]
              l[i][j] ← l[i − 1][j − 1] + 1
          sino
              l[i][j] ← máx{l[i − 1][j], l[i][j − 1]}
          fin si
      fin para
  fin para
  retornar l[r][s]
```

Este algoritmo es $O(n^2)$ y requiere $O(n^2)$ espacio para almacenar l.

En la biliografía pueden encontrar una gran cantidad de algoritmo de PD, por ejemplo el problema del cambio y el problema de la mochila versión discreta (que vimos al iniciar la clase).

2. Algoritmos heurísticos y aproximados

Por ahora sólo vamos a mencionar qué es un algoritmo heurístico. Hacia la mitad del cuatrimestre vamos a entrar en detalle en esto.

Dado un problema Π , un algoritmo heurístico es un algoritmo que intenta obtener soluciones de buena calidad para el problema que se quiere resolver pero no necesariamente lo hace en todos los casos.

Sea Π un problema de optimización, I una instancia del problema, $x^*(I)$ el valor óptimo de la función a optimizar en dicha instancia. Un algoritmo heurístico obtiene una solución con un valor que se espera sea cercano a ese óptimo pero no necesariamente va a ser el óptimo.

Si H es un algoritmo heurístico para un problema de optimización llamamos $x^H(I)$ al valor que devuelve la heurística.

Decimos que H es un algoritmo ϵ – aproximado para el problema Π si para algún $\epsilon > 0$

$$|x^H(I) - x^*(I)| \leq \epsilon |x^*(I)|$$

3. Bibliografía recomendada

- Capítulos 6, 7, 8 y 9.6 de G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
- Secciones I.4, IV.15 IV.16 de T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 2001.