Tecnológico de Monterrey.

Campus Querétaro.

TC2038. Análisis y diseño de Algoritmos A

M.C. Ramona Fuentes Valdéz

rfuentes@tec.mx

26

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Solución de problemas

- La intuición no es suficiente.
- Eficiencia y facilidad de implementación
- Depende de la situación:
  - Manejo de sensores en tiempo real
  - Extracción de conocimiento de bases de datos
  - Concurso de programación

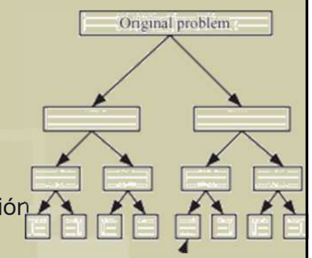
Diseño de algoritmos

- Con el tiempo se han observado patrones comunes a la hora de solucionar problemas
- Estos patrones son las técnicas de diseño de algoritmos.

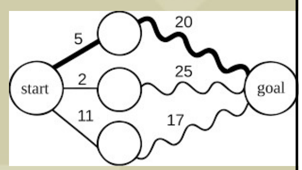
27

Técnicas de diseño de algoritmos

Divide y vencerás *no es perfecto*



- Dado un problema cuyas soluciones pueden ser expresadas recursivamente.
- Lo más natural es intentar con un **algoritmo recursivo** (tipo divide y vencerás)
- Sin embargo, muchas veces el tiempo de ejecución es **exponencial.**
- En muchas ocasiones debido a que **se resuelve más de una vez el mismo subproblema** (se traslapan los cálculos).
- **En la programación dinámica (PD), se busca:** evitar cálculos dos veces de una misma cosa, manteniendo una tabla de resultados conocidos que se vaya llenando a medida de que se resuelven los subcasos.



Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming

Programación Dinámica
(Dynamic programming**)**

Técnicas de diseño de algoritmos

Programación dinámica *(Dynamic programming)*

Un **algoritmo de programación dinámica tiene las siguientes características:**

- Almacena en una estructura de datos **soluciones parciales.**
- Parte de una **solución elemental conocida.**
- Debe ser posible obtener la solución a través de **una secuencia de decisiones óptimas.**

Clasificación

1. Problemas de optimización
2. Problemas combinatorios

Algoritmo

1. Plantear la solución como una sucesión de decisiones.
2. Definición recursiva de la solución.
3. Cálculo de la solución óptima mediante una tabla donde se almacenan soluciones parciales.
4. Construcción de la solución óptima.

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

1. Plantear solución como una sucesión de decisiones.
2. Definición recursiva de la solución.
3. Cálculo de la solución óptima mediante una tabla donde se almacenan soluciones parciales.
4. Construcción de la solución óptima.

Ejercicio:

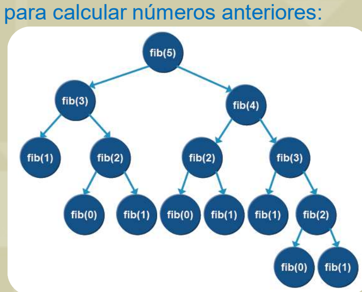
Sucesión de Fibonacci: $f_n = \begin{cases} n & si\ n=0 \text{ o } n=1 \\ f_{n-1}+f_{n-2} & \text{en otro caso} \end{cases}$

Algoritmo

Para divide y vencerás…

```
Función Fiborec(n)
    si  n <= 1 entonces
        devolver n
    sino
        devolver Fiborec(n-1) + Fiborec(n-2)
```

El problema se centraba en la repetición de trabajo para calcular números anteriores:



Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

Sucesión de Fibonacci: $f_n = \begin{cases} n & si\ n=0 \text{ o } n=1 \\ f_{n-1}+f_{n-2} & \text{en otro caso} \end{cases}$

Aplicando la PD:

1. **Plantear solución como una sucesión de decisiones:**
   - Se aprovechan los términos ya calculados:
   T[n] = T[n-1] + T[n-2]
2. **Definición recursiva de la solución:**
   - Partir de los casos base: T[0]=1 y T[1]=1
3. **Cálculo de la solución óptima mediante una tabla donde se almacenan soluciones parciales.**
   - Para este caso, un arreglo unidimensional donde se van almacenando los resultados parciales.
4. **Construcción de la solución óptima.**
   - La solución es la sucesión de elementos del arreglo.

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

Sucesión de Fibonacci: $f_n = \begin{cases} n & si\ n{=}0 \text{ o } n{=}1 \\ f_{n-1}+f_{n-2} & \text{en otro caso} \end{cases}$

Aplicando la PD:

1. Plantear solución como una sucesión de decisiones — T[n] = T[n-1] + T[n-2]
2. Definición recursiva de la solución — Casos base: T[0]=1 y T[1]=1
3. Cálculo de la solución óptima mediante una tabla donde se almacenan soluciones parciales — Arreglo unidimensional donde se van almacenando los resultados parciales.
4. Construcción de la solución óptima — La solución es la sucesión de elementos del arreglo.

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}

Algoritmo de PD para la serie de Fibonacci

```
int fibonacci_prog_dinamica(int n) {
    int i;
    int[] f=new int[n+1];
    f[0] = 1;
    if (n > 0) {
        f[1] = 1
        for (i=2 ; i<=n ; i++)
            f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

Ejemplo: **Casos base:**
f[0]=1
f[1]=1

f[i] = f[i-1] + f[i-2]

	Serie						
Fibo							
	0	1	2	3	4	5	6

32

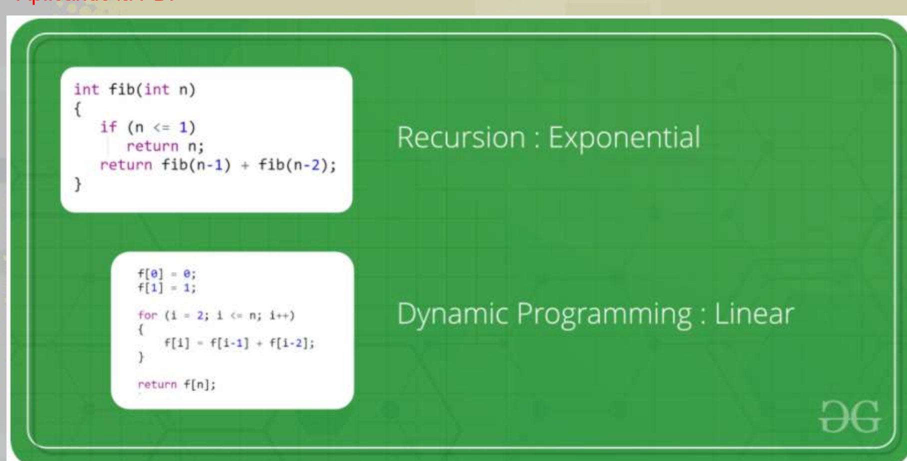
Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

Sucesión de Fibonacci: $f_n = \begin{cases} n & si\ n{=}0 \text{ o } n{=}1 \\ f_{n-1}+f_{n-2} & \text{en otro caso} \end{cases}$

Aplicando la PD:



}

33

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

Coeficiente binomial — El número de formas posibles de elegir *k* elementos de *n* posibles está dado por:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{para } 0 \le k \le n$$

Ejemplo:

Conjunto = {A, B, C, D, E, F}, k=2

A,B	A,C	A,D	A,E	A,F
	B,C	B,D	B,E	B,F
		C,D	C,E	C,F
			D,E	D,F
				E,F

$$\binom{n}{k} = \binom{6}{2} = 15$$

Problema:
para valores grandes de *k* y *n* los factoriales son números excesivamente grandes.

En forma recursiva:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{para } 0 < k < n \\ 1 & \text{para } k = 0 \text{ ó } k = n \end{cases}$$

Algoritmo:

```
int CoefBin(int n, int k){
    if (k==0 || k==n)
        return 1;
    else
        return CoefBin (n-1, k-1) + CoefBin (n-1, k);
}
```

El problema con la implementación recursiva es que muchos cálculos se repiten una y otra vez.

Ejemplo:

El algoritmo calcula CoefBin (5,3) como la suma de CoefBin(4,2) y CoefBin(4,3).

- Ambos resultados intermedios exigen calcular CoefBin(3,2).
- De igual forma, el valor CoefBin(2,2) se utiliza muchas veces.

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

Coeficiente binomial — El número de formas posibles de elegir *k* elementos de *n* posibles está dado por:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{para } 0 \le k \le n$$

Con PD:

- Se utilizará una matriz para almacenar las soluciones parciales.
- Para almacenar los resultados en la matriz, se utilizará la siguiente forma "recursiva":

```
C[i][j] = { C[i-1][j-1] + C[i-1][j]    0<j<i
          { 1                          j=0 ó j=i
```

¿Cómo se debe ir llenando la matriz C ?

j

i

	0	1	2	3	4	5	…	k-1	k
0									
1									
2									
3									
4									
5									
6									
n-1									
n									

$C(n\text{-}1,k\text{-}1) + C(n\text{-}1,k) \rightarrow C(n,k)$

- Observen que el patrón es igual a un triángulo de pascal.
- Cada línea es calculada solo en base a la anterior.

Algoritmo:

```
int CoefBin2(int n, int k){
   int i, j;
   int[][] C = new int[n+1][k+1];
   for (i=0 ; i<=n ; i++) {
      for (j=0 ; j<=min(i,k) ; j++) {
         if (j==0 || j==i)
            C[i][j] = 1;
         else
            C[i][j] = C[i-1][j-1]+C[i-1][j];
      }
   }
   return C[n][k];
}
```

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

Coeficiente binomial El número de formas posibles de elegir *k* elementos de *n* posibles está dado por:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{para } 0 \le k \le n$$

Con PD:

- Se utilizará una matriz para almacenar las soluciones parciales.
- Para almacenar los resultados en la matriz, se utilizará la siguiente forma "recursiva":

$$C[i][j] = \begin{cases} C[i-1][j-1] + C[i-1][j] & 0<j<i \\ 1 & j=0 \text{ ó } j=i \end{cases}$$

Análisis de complejidad

Veamos el número de veces que se ejecutan las instrucciones dentro de los ciclos anidados:

- Para *i=0*, el ciclo de *j* se ejecuta 1 veces
- Para *i=1*, el ciclo de *j* se ejecuta 2 veces
- …
- Para *i=k-1*, el ciclo de *j* se ejecuta *k* veces
- Para *i=k*, el ciclo de *j* se ejecuta *k+1* veces
- Para *i=k+1*, el ciclo de *j* se ejecuta *k+1* veces
- …
- Para *i=n*, el ciclo de *j* se ejecuta *k+1* veces

Algoritmo:

```
int CoefBin2(int n, int k){
   int i, j;
   int[][] C = new int[n+1][k+1];
   for (i=0 ; i<=n ; i++) {
      for (j=0 ; j<=min(i,k) ; j++) {
         if (j==0 || j==i)
            C[i][j] = 1;
         else
            C[i][j] = C[i-1][j-1]+C[i-1][j];
      }
   }
   return C[n][k];
}
```


Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

Coeficiente binomial El número de formas posibles de elegir *k* elementos de *n* posibles está dado por:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{para } 0 \le k \le n$$

Con PD:

- Se utilizará una matriz para almacenar las soluciones parciales.
- Para almacenar los resultados en la matriz, se utilizará la siguiente forma "recursiva":

$$C[i][j] = \begin{cases} C[i-1][j-1] + C[i-1][j] & 0<j<i \\ 1 & j=0 \text{ ó } j=i \end{cases}$$

Análisis de complejidad

Veamos el número de veces que se ejecutan las instrucciones dentro de los ciclos anidados:

$$W(n) = 1 + 2 + 3 + \ldots + k + \underbrace{(k+1) + (k+1) + \ldots + (k+1)}_{(n-k+1) \text{ veces}}$$

$$W(n) = \frac{k(k+1)}{2} + (n-k+1)(k+1)$$

$$= \frac{(2n-k+2)(k+1)}{2}$$

Por lo tanto:

O(W(n)) = O(*nk*)

Algoritmo:

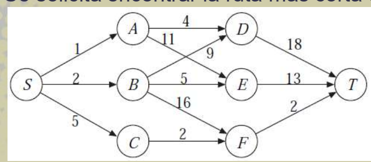
```
int CoefBin2(int n, int k){
   int i, j;
   int[][] C = new int[n+1][k+1];
   for (i=0 ; i<=n ; i++) {
      for (j=0 ; j<=min(i,k) ; j++) {
         if (j==0 || j==i)
            C[i][j] = 1;
         else
            C[i][j] = C[i-1][j-1]+C[i-1][j];
      }
   }
   return C[n][k];
}
```

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

- Se solicita encontrar la ruta más corta de *S* a *T*.



Solución voraz:

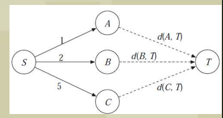
- Ruta $S \underset{1}{\rightarrow} A \underset{4}{\rightarrow} D \underset{18}{\rightarrow} T$ = 23

no es óptima

- Ruta $S \underset{5}{\rightarrow} C \underset{2}{\rightarrow} F \underset{2}{\rightarrow} T$ = 9

Proceso a través de PD

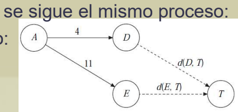
1. Al iniciar en *S* hay que pasar por *A*, *B* o *C*.
   a) La longitud de la ruta más corta de *S* a *T* se determina con la fórmula siguiente:



$d(S, T) = \text{mín}\{1 + d(A, T),\ 2 + d(B, T),\ 5 + d(C, T)\}$

donde $d(X, T)$ denota la longitud de la ruta más corta de *X* a *T*.

2. Para encontrar las rutas más cortas de *A*, *B* y *C* a *T*, se sigue el mismo proceso:
   a) Para el vértice *A*, se tiene el siguiente subgrafo:



Es decir, **$d(A, T) = \text{mín}\{4 + d(D, T),\ 11 + d(E, T)\}$.**

Como $d(D, T) = 18$ y $d(E, T) = 13$, se tiene:

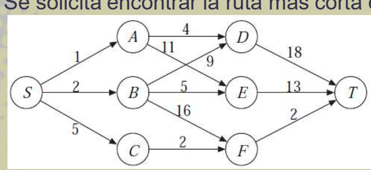
$$d(A, T) = \text{mín}\{4 + 18,\ 11 + 13\} = \text{mín}\{22, 24\} = 22.$$

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

- Se solicita encontrar la ruta más corta de *S* a *T*.



Solución voraz:

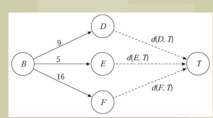
- Ruta $S \underset{1}{\rightarrow} A \underset{4}{\rightarrow} D \underset{18}{\rightarrow} T$ = 23

no es óptima

- Ruta $S \underset{5}{\rightarrow} C \underset{2}{\rightarrow} F \underset{2}{\rightarrow} T$ = 9

Proceso a través de PD

b) De manera similar se trabaja con B:



$$d(B, T) = \text{mín}\{9 + d(D, T),\ 5 + d(E, T),\ 16 + d(F, T)\} = \text{mín}\{9 + 18,\ 5 + 13,\ 16 + 2\} = \text{mín}\{27, 18, 18\} = 18.$$

c) Por otro lado, el camino $d(C, T)$ es 4.

3. Una vez que se han encontrado $d(A, T)$, $d(B, T)$ y $d(C, T)$ es posible encontrar $d(S, T)$:

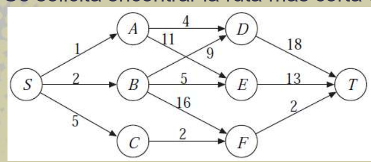
$$d(S, T) = \text{mín}\{1 + 22,\ 2 + 18,\ 5 + 4\} = \text{mín}\{23, 20, 9\} = 9.$$

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio:

- Se solicita encontrar la ruta más corta de *S* a *T*.



Solución voraz:

- Ruta $S \xrightarrow{1} A \xrightarrow{4} D \xrightarrow{18} T = 23$ — *no es óptima*
- Ruta $S \xrightarrow{5} C \xrightarrow{2} F \xrightarrow{2} T = 9$

Proceso a través de PD

También pudiera resolverse de la siguiente manera:

1. Primero se encuentran $d(S, A)$, $d(S, B)$ y $d(S, C)$.

   $d(S, A)$ 1
   $d(S, B)$ 2
   $d(S, C)$ 5.

2. Luego se determinan $d(S, D)$, $d(S, E)$ y $d(S, F)$ como sigue:

$d(S, D)$ = mín{$d(A, D) + d(S, A)$, $d(B, D) + d(S, B)$}
= mín{4 1, 9 2}
= mín{5, 11}
= 5

$d(S, E)$ = mín{$d(A, E) + d(S, A)$, $d(B, E) + d(S, B)$}
= mín{11 1, 5 2}
= mín{12, 7}
= 7

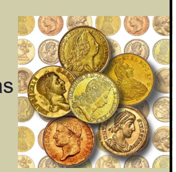
$d(S, F)$ = mín{$d(B, F) + d(S, B)$, $d(C, F) + d(S, C)$}
= mín{16 2, 2 5}
= mín{18, 7}
= 7.

3. Ahora es posible determinar la distancia más corta de *S* a *T* como sigue:

$d(S, T)$ = mín{$d(D, T) + d(S, D)$, $d(E, T) + d(S, E)$, $d(F, T) + d(S, F)$}
= mín{18 + 5, 13 + 7, 2 + 7}
= mín{23, 20, 9}
= 9.

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)



Ejercicio: Problema de dar cambio

- Supón que vives en un país donde sólo están disponibles las monedas de 100, 25, 10, 5 y 1.
- Debes diseñar un algoritmo para pagar una cantidad *utilizando el menor número posible de monedas.*
- *Ejemplo*, para pagar 289 → mejor solución son 2 monedas de 100, 3 de 25, 1 de 10 y 4 de 1).

¿Qué pasa si…?

- En el lugar donde vives, sólo hay monedas de 1, 4 y 6 unidades.

  Tenemos que dar cambio de 8 unidades.

  Se pudiera utilizar:
  - 1 moneda de 6 y 2 de 1 unidad.
  - 2 monedas de 4 unidades.

Con PD:

- Sean:
  - **Cant:** La cantidad a devolver
  - **M:** El número de denominaciones distintas
  - **v_m:** el valor de una moneda m
- Cálculo de la solución mediante una tabla donde se almacenan soluciones parciales:
  - Arreglo bidimensional **cambio[1..M] [0..Cant]** donde **cambio[m][c]** representa el número de monedas de tipo **m o menos** necesarias para devolver una cantidad **c.**

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio: Problema de dar cambio

- Supón que vives en un país donde sólo están disponibles las monedas de 100, 25, 10, 5 y 1.
- Debes diseñar un algoritmo para pagar una cantidad *utilizando el menor número posible de monedas.*
- *Ejemplo*, para 289 → 2 monedas de 100, 3 de 25, 1 de 10 y 4 de 1).

Cant: Cantidad a devolver
M: Número denominaciones distintas
$\mathbf{v_m}$: Valor de una moneda m

Con PD:

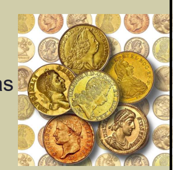
- Cálculo de la solución mediante una tabla donde se almacenan soluciones parciales:
  - Arreglo bidimensional **cambio[1..M] [0..Cant]** donde **cambio[m][c]** representa el número de monedas de tipo **m o menos** necesarias para devolver una cantidad **c.**
- Definición recursiva de la solución y caso base:
  - Tomar en cada paso las monedas que se tienen + una moneda ***m*** (*o quedarme sólo con las monedas que se tienen*).
  - 0 cuando la cantidad a devolver es cero.
- Construcción de la solución:
  - Elegimos en cada paso no utilizar monedas de valor $\mathbf{v_m}$ y en este caso **cambio[m][c] = cambio[m-1][c]**
  - O incluir al menos una moneda de valor $\mathbf{v_m}$, en este caso **cambio[m][c] = 1+cambio[m][c-v_m]**
  - Por lo tanto, **cambio[m][c] = min (cambio[m-1][c], 1+cambio[m][c-v_m])**

IMTA Dr. Alberto González

42

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)



Ejercicio: Problema de dar cambio

- Supón que vives en un país donde sólo están disponibles las monedas de 100, 25, 10, 5 y 1.
- Debes diseñar un algoritmo para pagar una cantidad *utilizando el menor número posible de monedas.*
- *Ejemplo*, para 289 → 2 monedas de 100, 3 de 25, 1 de 10 y 4 de 1).

Cant: Cantidad a devolver
M: Número denominaciones distintas
$\mathbf{v_m}$: Valor de una moneda m

Ejemplo:

- Monedas de 1, 4 y 6 unidades.

  Tenemos que dar cambio de 8 unidades.

  Conjunto de tres monedas (M=3) → v_1=1, v_2=4 y v_3=6

  Cant=8 (cantidad a devolver).

Arreglo bidimensional
cambio[1..M] [0..Cant]

La tabla cambio[1..3][0..8] será:

Valor	Cantidad a cambiar								
moneda	0	1	2	3	4	5	6	7	8
v_1 = 1	0	1	2	3	4	5	6	7	8
v_2 = 4	0	1	2	3	1	2	3	4	2
v_3 = 6	0	1	2	3	1	2	1	2	**2**

IMTA Dr. Alberto González

43

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio: Problema de dar cambio

- Supón que vives en un país donde sólo están disponibles las monedas de 100, 25, 10, 5 y 1.
- Debes diseñar un algoritmo para pagar una cantidad *utilizando el menor número posible de monedas.*
- *Ejemplo*, para 289 → 2 monedas de 100, 3 de 25, 1 de 10 y 4 de 1).

Cant: Cantidad a devolver
M: Número denominaciones distintas
v_m: Valor de una moneda m

Algoritmo:

```
int darCambio(int Cant){
  int v[M] = {1,...};      // valor de las monedas (Ejemplo = {1,4,6})
  int [][] cambio=new int[M+1,Cant+1]; // Arreglo bidimensional cambio[1..M] [0..Cant]
    for (int m=1; m<=M; m++) {
      for (int c=1; c<= Cant; c++) {
         if (m==1)
           cambio[m,c] = c;
         else if (c < v[m])
              cambio[m,c]= cambio[m-1,c];
            else
              cambio[m,c]=min(cambio[m-1,c], 1+cambio[m,c-v[m]]);
      }
    }
 retorna cambio[M,Cant]
}
```

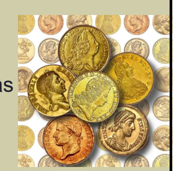
El proceso indica cuántas monedas devolver, pero no cuáles son esas monedas.

La tabla cambio[1..3][0..8] será:

Valor moneda	cantidad a cambiar								
	0	1	2	3	4	5	6	7	8
v_1=1	0	1	2	3	4	5	6	7	8
v_2=4	0	1	2	3	1	2	3	4	2
v_3=6	0	1	2	3	1	2	1	2	2

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)



Ejercicio: Problema de dar cambio

- Supón que vives en un país donde sólo están disponibles las monedas de 100, 25, 10, 5 y 1.
- Debes diseñar un algoritmo para pagar una cantidad *utilizando el menor número posible de monedas.*
- *Ejemplo*, para 289 → 2 monedas de 100, 3 de 25, 1 de 10 y 4 de 1).

Cant: Cantidad a devolver
M: Número denominaciones distintas
v_m: Valor de una moneda m

Proceso:

- Esta información se puede obtener de la tabla:
  - Comenzamos en la posición cambio[M, Cant]
  - Si cambio[M, Cant] == cambio[M-1, Cant] la solución no incluye ninguna moneda de tipo M y pasamos a cambio[M-1, Cant]
  - Si cambio[M, Cant] == 1+cambio[M, Cant-v_M] se incluye en la solución una moneda de tipo M y pasamos a cambio[M, Cant-v_M]
- Se continua de esta manera hasta llegar a la columna 0:

Valor moneda	cantidad a cambiar								
	0	1	2	3	4	5	6	7	8
v_1=1	0	1	2	3	4	5	6	7	8
v_2=4	0	1	2	3	1	2	3	4	2
v_3=6	0	1	2	3	1	2	1	2	2

1 moneda de 4 1 moneda de 4

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Ejercicio: Problema de dar cambio

- Supón que vives en un país donde sólo están disponibles las monedas de 100, 25, 10, 5 y 1.
- Debes diseñar un algoritmo para pagar una cantidad *utilizando el menor número posible de monedas.*
- *Ejemplo*, para 289 → 2 monedas de 100, 3 de 25, 1 de 10 y 4 de 1).

Cant: Cantidad a devolver
M: Número denominaciones distintas
v_m: Valor de una moneda m

Complejidad:

- Eficiencia temporal del algoritmo darCambio:
  - **Operación básica:** calcular cambio[m,c]
  - **Tamaño de la entrada:** número de denominaciones M.
    - El ciclo externo se realiza M veces y el interno Cant veces
    - W(n) = n·Cant
    - Su orden de complejidad es **O(n)**
- Eficiencia temporal del algoritmo para reconstruir la solución:
  - Ir desde la fila M hasta la 1 cuesta M pasos
  - Ir desde la columna Cant hasta la 0 cuesta tantos pasos como monedas hay en la solución (cambio[M,Cant])
  - W(n) = n + n = 2n
  - Su orden de complejidad es **O(n)**

Técnicas de diseño de algoritmos

Programación dinámica (*Dynamic programming*)

Un **algoritmo de programación dinámica tiene las siguientes características:**

- Almacena en una estructura de datos **soluciones parciales.**
- Parte de una **solución elemental conocida.**
- Debe ser posible obtener la solución a través de **una secuencia de decisiones óptimas.**

Algoritmo

1. Plantear solución como una sucesión de decisiones.
2. Definición recursiva de la solución.
3. Cálculo de la solución óptima mediante una tabla donde se almacenan soluciones parciales.
4. Construcción de la solución óptima.

Conclusiones

- Programación dinámica normalmente se usa cuando no podemos emplear divide y vencerás.
- Siempre se usa una estructura de datos.
- Se parte de la instancia más pequeña conocida.

Algunas aplicaciones:

- **Unix diff**, comparar dos archivos (https://en.wikipedia.org/wiki/Diff)
- **Bellman-Ford**, el camino más corto en redes (https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)
- **TeX**, el antecessor de LaTeX (https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)
- **WASP**, Winning and Score Predictor (https://en.wikipedia.org/wiki/WASP_%28cricket_calculation_tool%29)