

Algoritmos

IIC1005 – Computación:
Ciencia y Tecnología del Mundo Digital

Dos ideas cambiaron al mundo [Dasgupta et al., 2006]:

en 1448 Johann Gutenberg descubrió una forma de imprimir libros

el alfabetismo se propagó

...

... y finalmente tuvo lugar la revolución industrial

Muchos historiadores dicen que esto se lo debemos a la **impresa**

Pero hay quienes insisten en que el desarrollo clave no fue la imprenta
... sino los **algoritmos**

Gutenberg habría escrito el número 1448 como MCDXLVIII

¿Cómo sumamos dos números romanos?

MCDXLVIII + DCCCXII = ¿?

... y ¿cómo los multiplicamos?

Había que consultar a un especialista en ábacos

El sistema posicional decimal (que empleamos hoy) fue inventado en la India alrededor del año 600:

solamente 10 símbolos

la aritmética se podía hacer eficientemente siguiendo pasos elementales

En un libro escrito en árabe cerca del año 850, estaban explicados los métodos básicos para sumar, multiplicar, dividir, sacar raíz cuadrada y calcular los dígitos de π :

procedimientos precisos, no ambiguos, mecánicos, eficientes y correctos —> **algoritmos**

Pero estas ideas se demoraron en llegar a Europa

Algoritmo de multiplicación binaria (versión informal):

Colocar una copia del multiplicando en el lugar correcto*, si el dígito del multiplicador es 1 ($= 1 \times$ multiplicando)

... o bien colocar 0 en el lugar correcto*, si el dígito del multiplicador es 0 ($= 0 \times$ multiplicando)

* en cada nueva iteración, se escribe desplazándolo un dígito a la izquierda

El origen de *algoritmo*¹

Late 17th cent.: variant (influenced by Greek *arithmos* “number”) of Middle English *algorism*,

... via Old French from medieval Latin *algorismus*.

The Arabic source, *al-Kwārīz̄mī* “the man of Kwārīz̄m” (now Khiva), was a name given to the 9th-cent. mathematician Abū Ja‘far Muhammad ibn Mūsa.

¹Tomado del diccionario de mi computador

```
def procedimiento(x, L):
    if L == []:
        return False
    boolean = False
    i = 0
    j = len(L)-1
    while (not boolean) and i <= j:
        k = (i+j)//2
        if x == L[k]:
            boolean = True
        elif x < L[k]:
            j = k-1
        else:
            i = k+1
    if boolean:
        return k
    else:
        return False
```

Un programa es una implementación de un método para resolver un problema

El método ha sido inventado previamente

... es independiente del lenguaje de programación usado:

- es igualmente apropiado para muchos computadores y muchos lenguajes de programación

... y especifica los pasos que podemos seguir, o las instrucciones que tenemos que llevar a cabo, para resolver el problema —es un **algoritmo**

Algoritmo:

Método finito, determinista y eficaz para resolver problemas

método:

- secuencia de instrucciones que si las llevamos a cabo en orden realizan una tarea particular

finito:

- si llevamos a cabo las instrucciones, entonces, siempre, el algoritmo termina después de un número finito de pasos
- además, el tiempo para terminar debería ser relativamente corto

determinista:

- cada instrucción es clara, no ambigua

eficaz:

- cada instrucción es suficientemente básica —puede ser llevada a cabo, en principio, por una persona usando sólo papel y lápiz

además:

... apropiado para ser implementado como un programa:

- p.ej., el algoritmo de Euclides para encontrar el máximo común divisor de dos números

... puede ser especificado de varias maneras:

- en lenguaje natural (castellano, inglés)
- como un diagrama de flujo
- como un programa (en un lenguaje de programación)
- como un diseño de hardware

El algoritmo de Euclides

En castellano:

- Calculamos el máximo común divisor de dos enteros no negativos, p y q , así: Si q es 0, la respuesta es p . De lo contrario, divide p por q y toma el resto r . La respuesta es el máximo común divisor de q y r .

Como un programa, en Python:

```
def mcd(p, q):  
    if q == 0:  
        return p  
    r = p%q  
    return mcd(q, r)
```

Los algoritmos deben ser inventados, validados y analizados

Inventado:

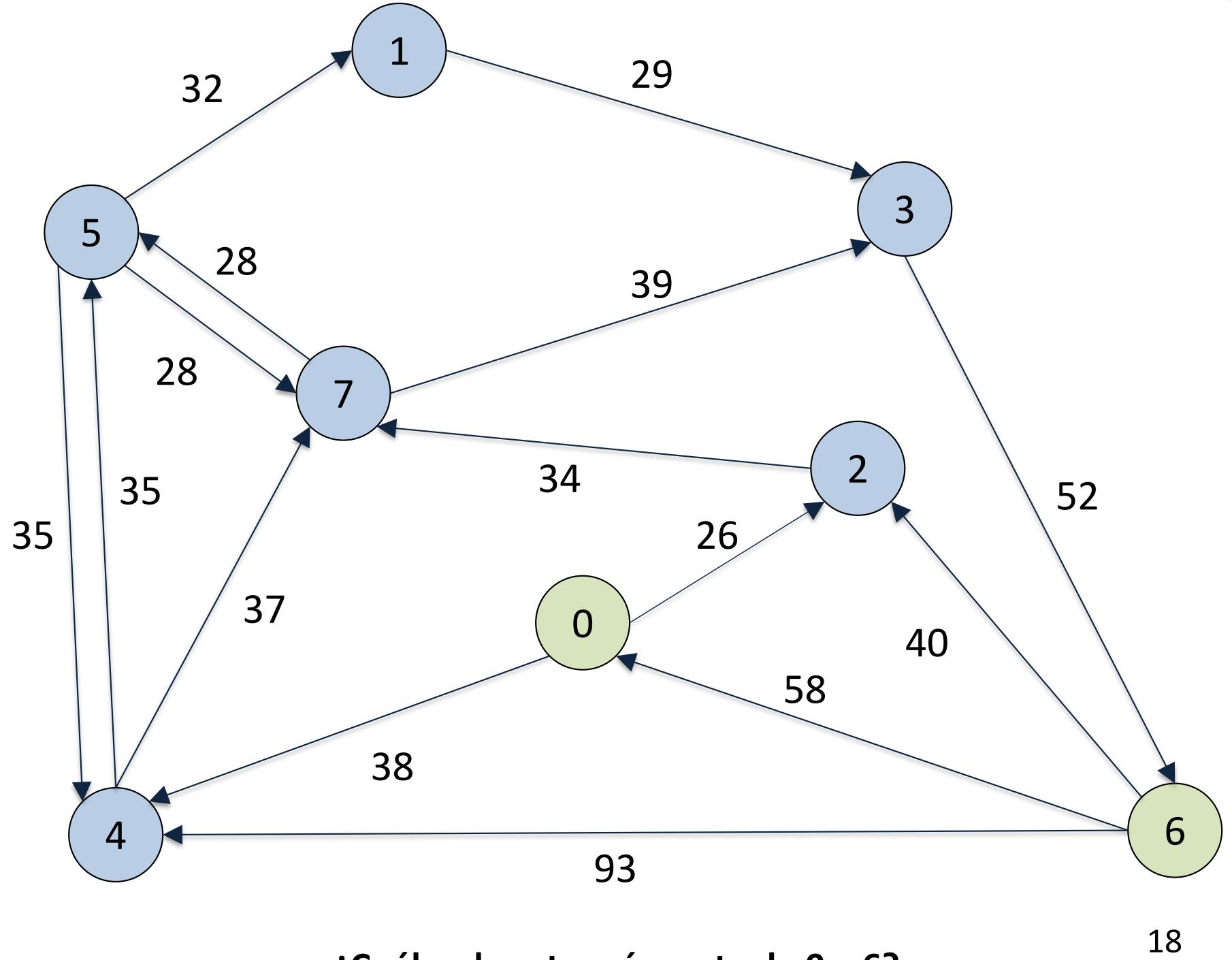
- posiblemente usando alguna de las técnicas de diseño de algoritmos que han demostrado ser útiles —*backtracking*, dividir para conquistar, algoritmos codiciosos, programación dinámica

Validado:

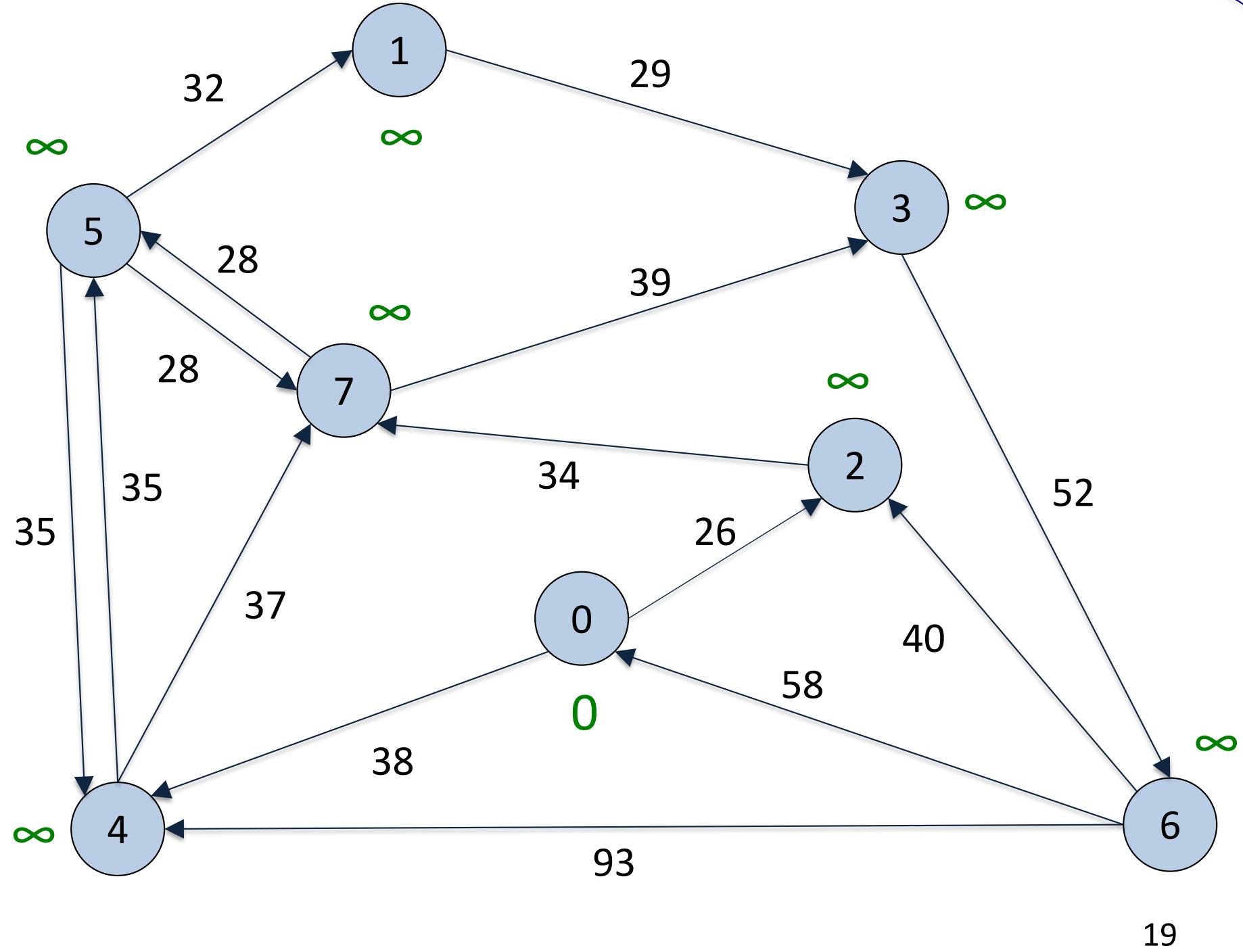
- hay que demostrar que calcula la respuesta correcta para todos los *inputs* legales posibles —por inducción, por contradicción

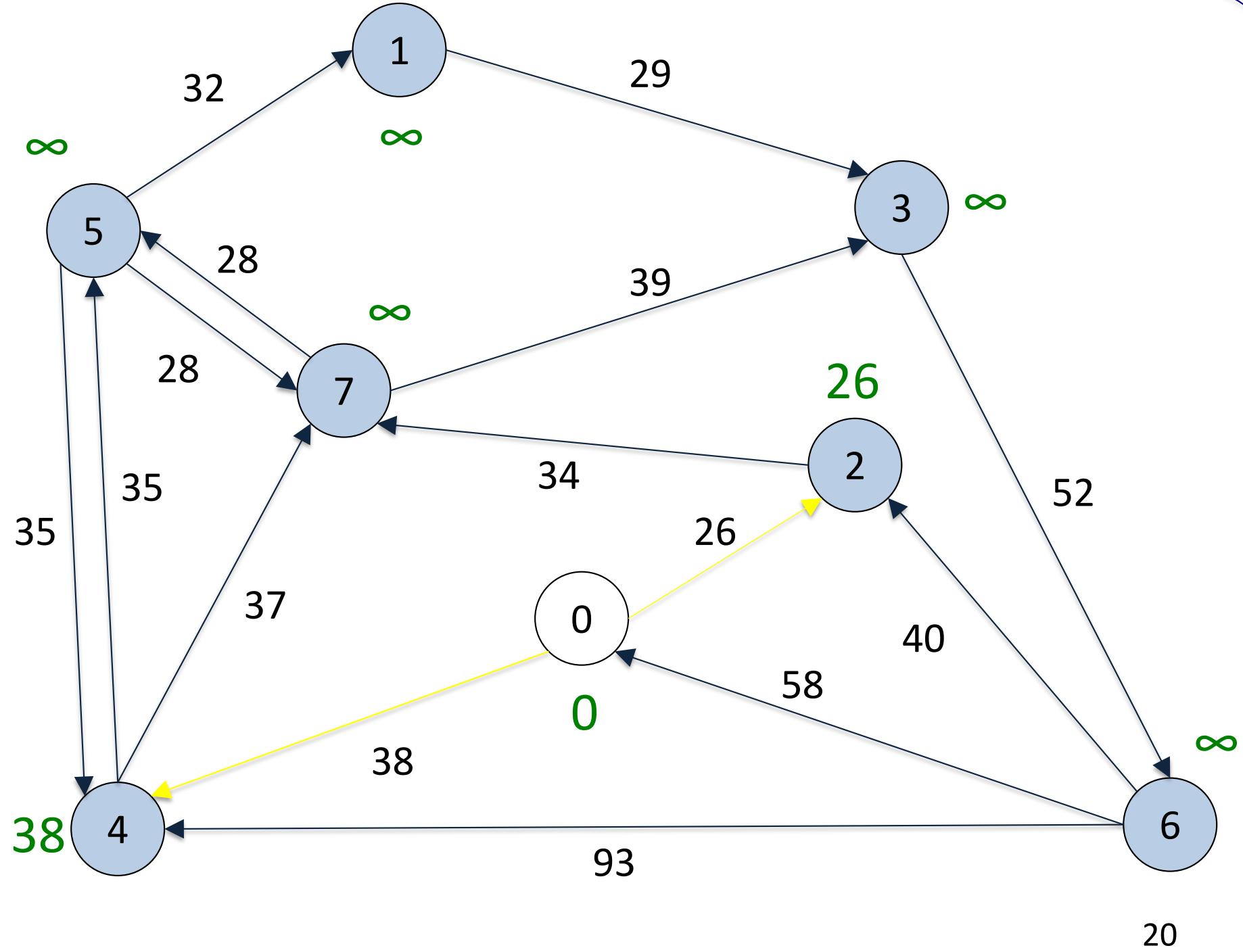
Analizado:

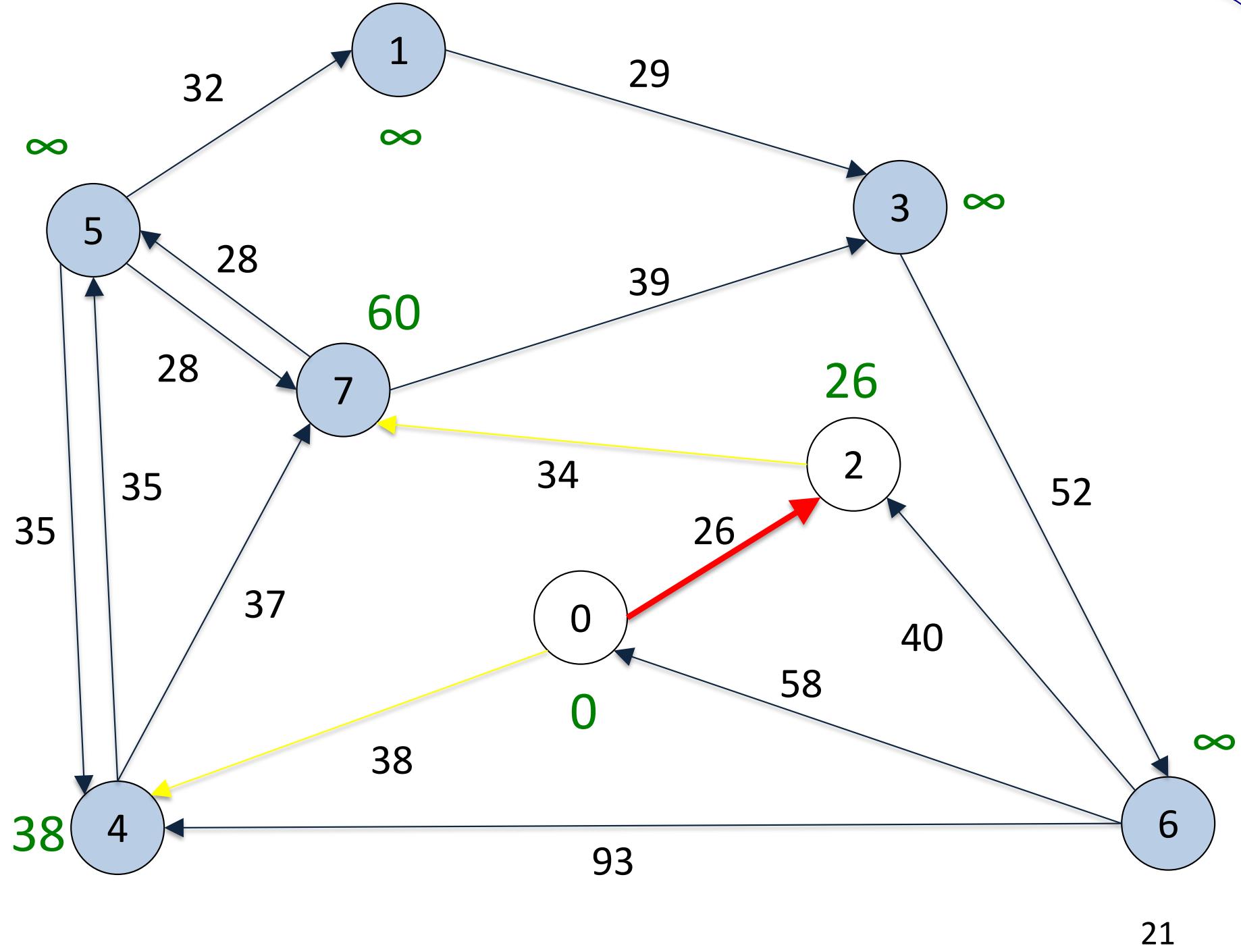
- determinar cuánto tiempo de computación y cuánta memoria necesita

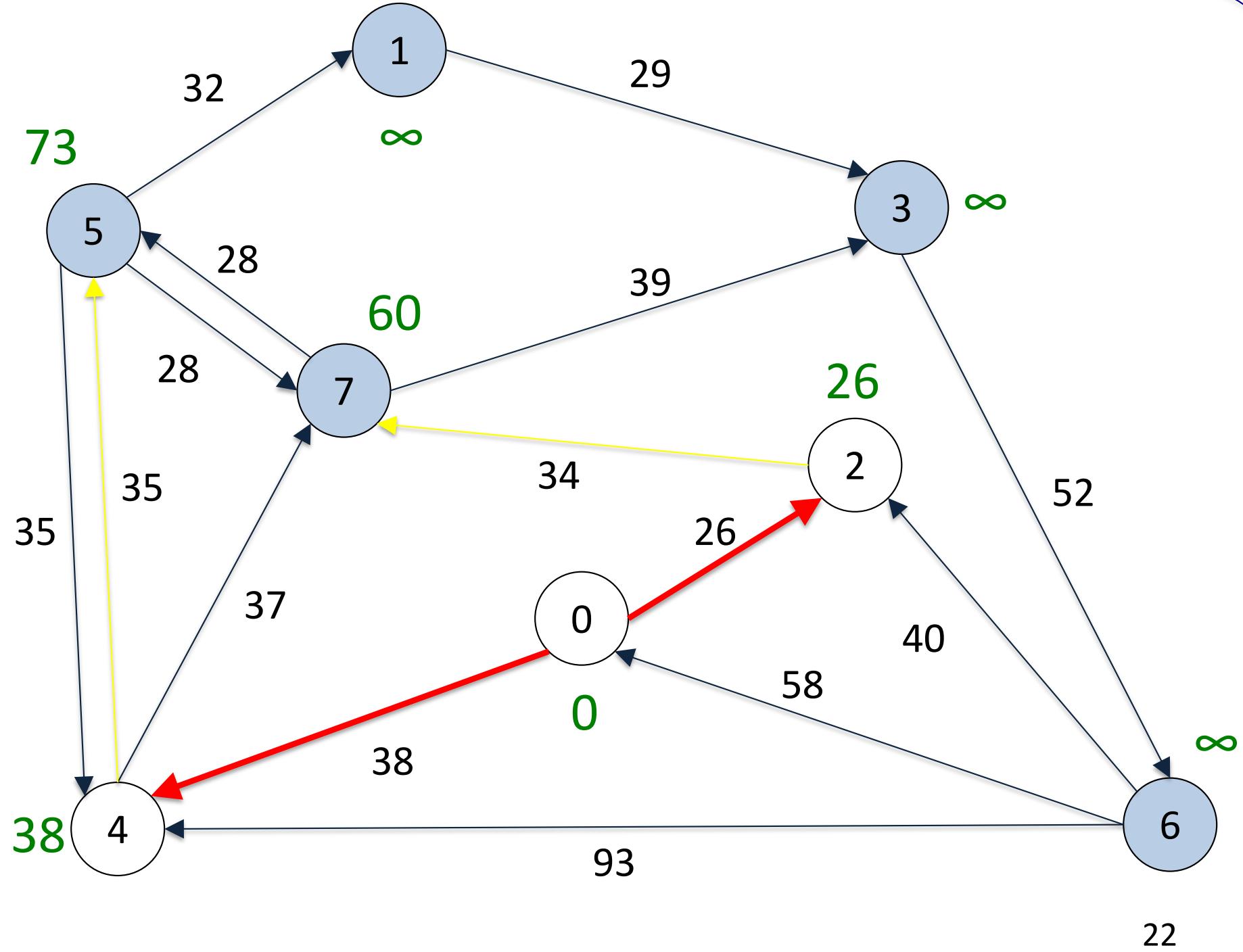


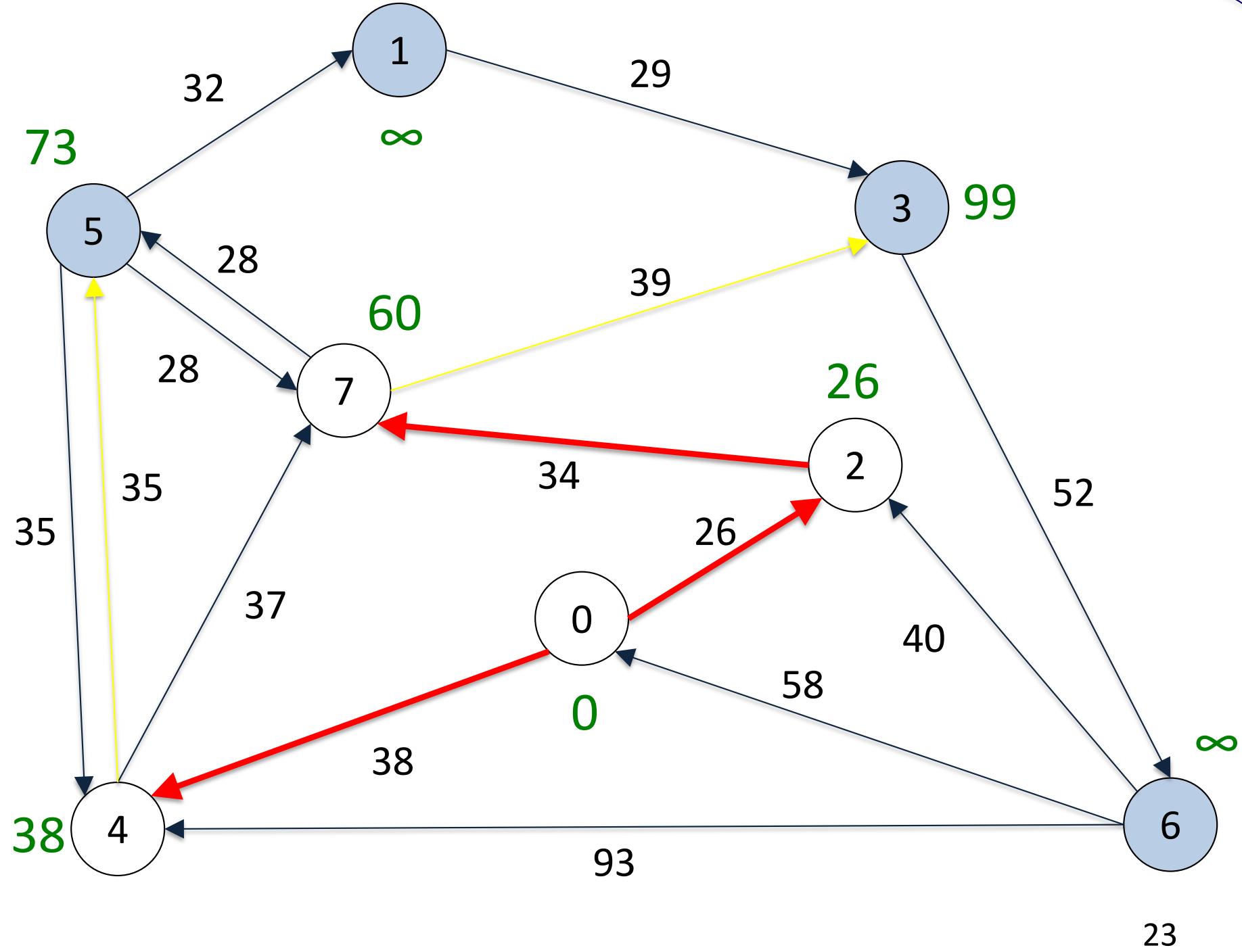
¿Cuál es la ruta más corta de 0 a 6?

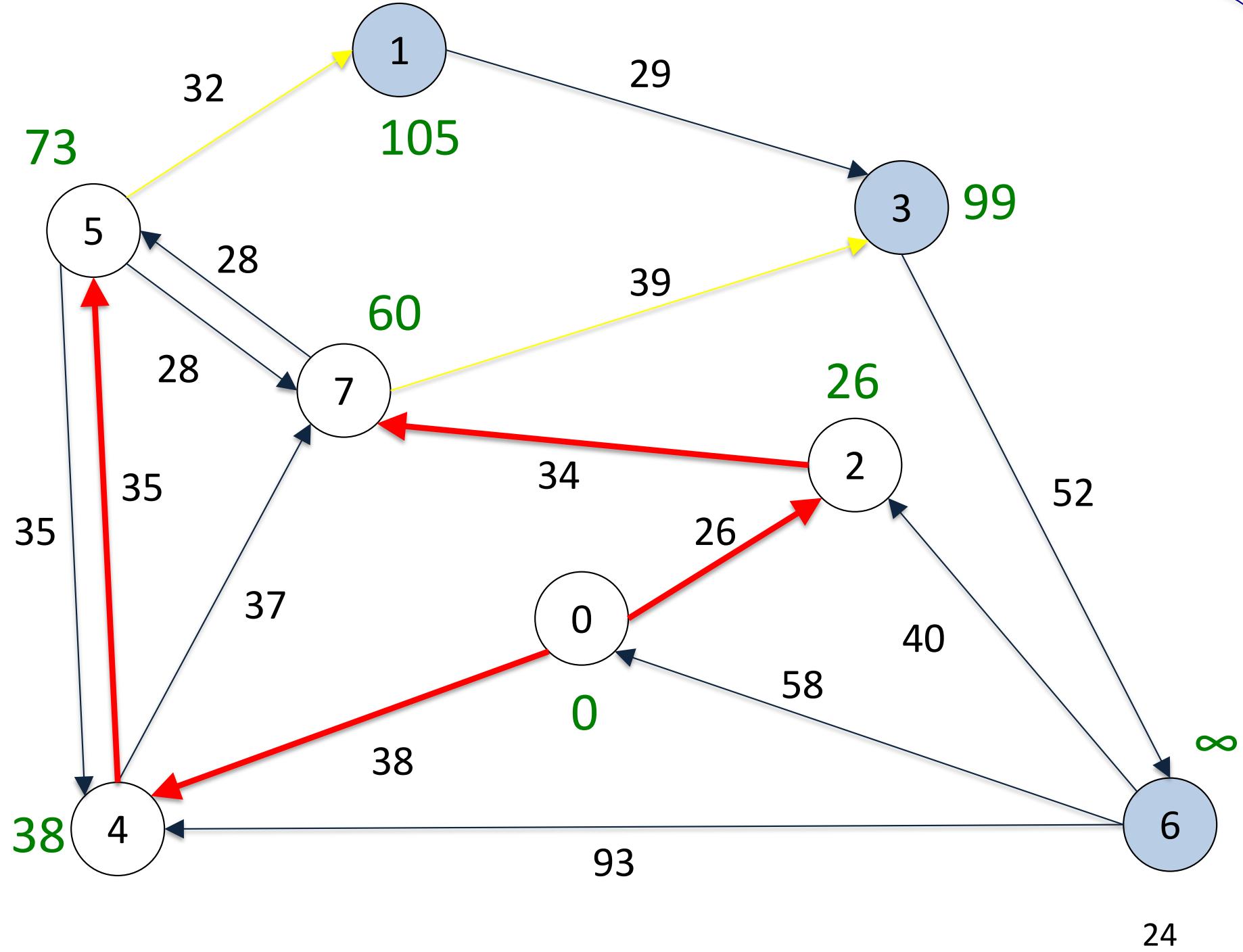


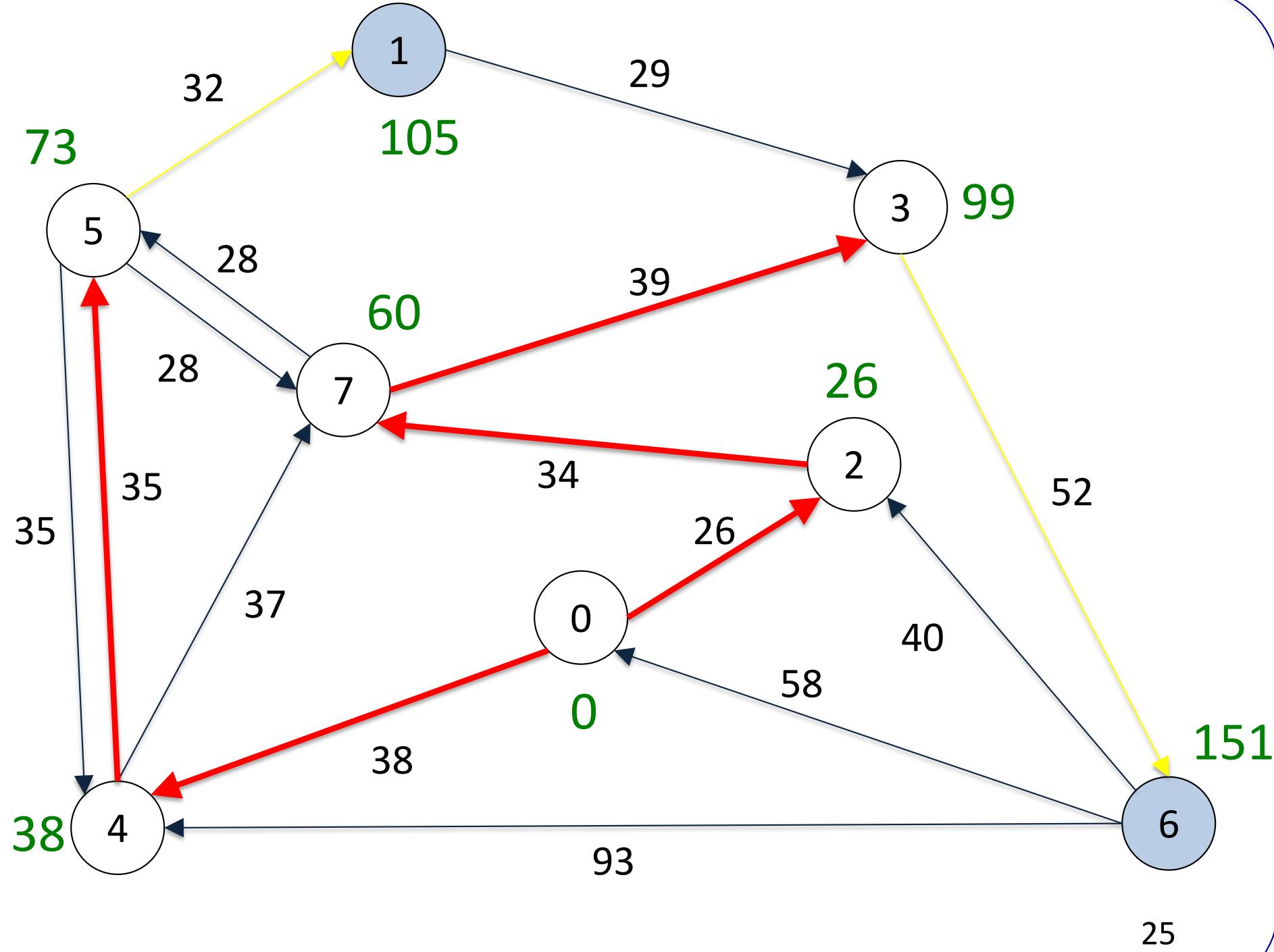


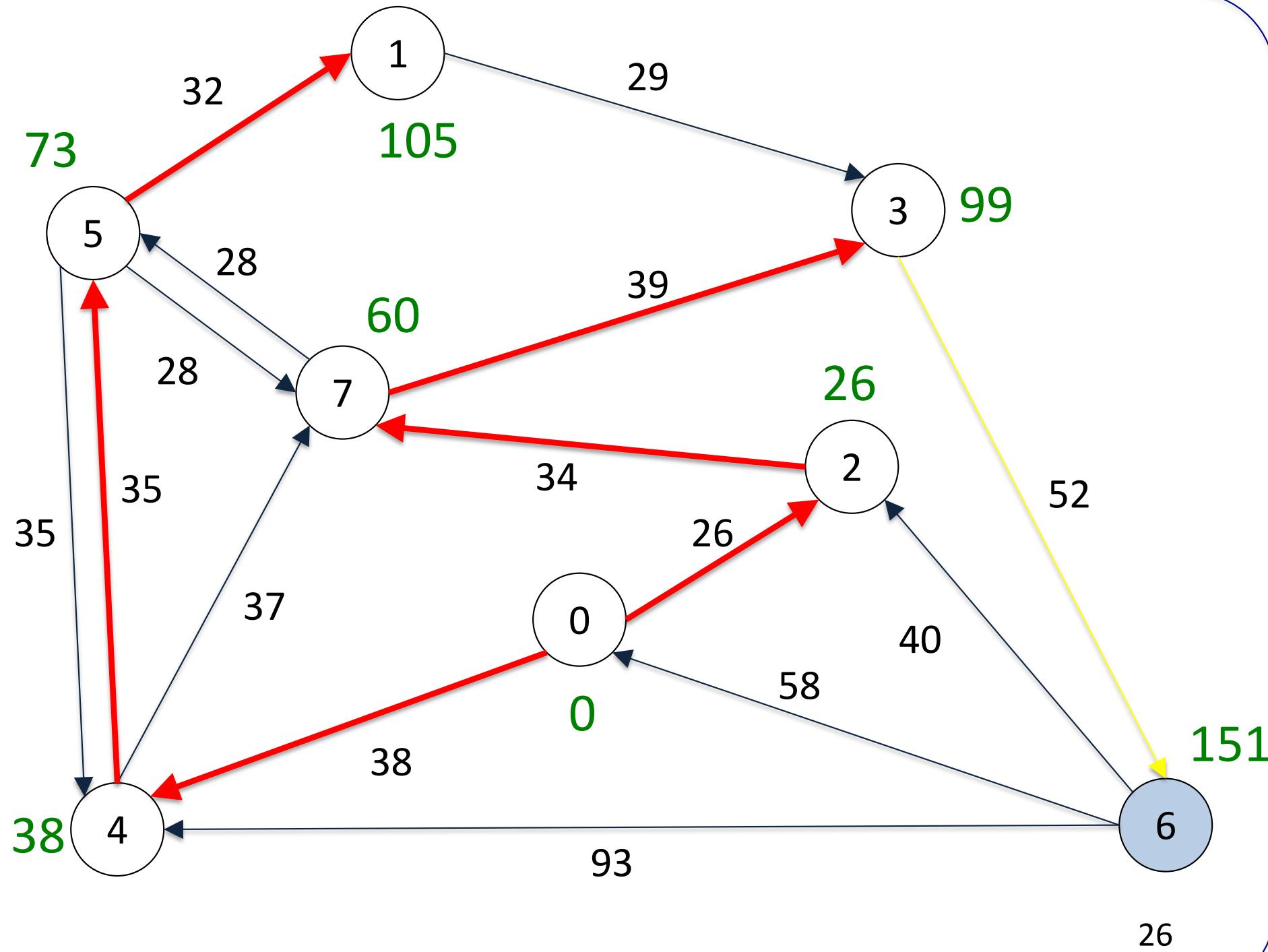


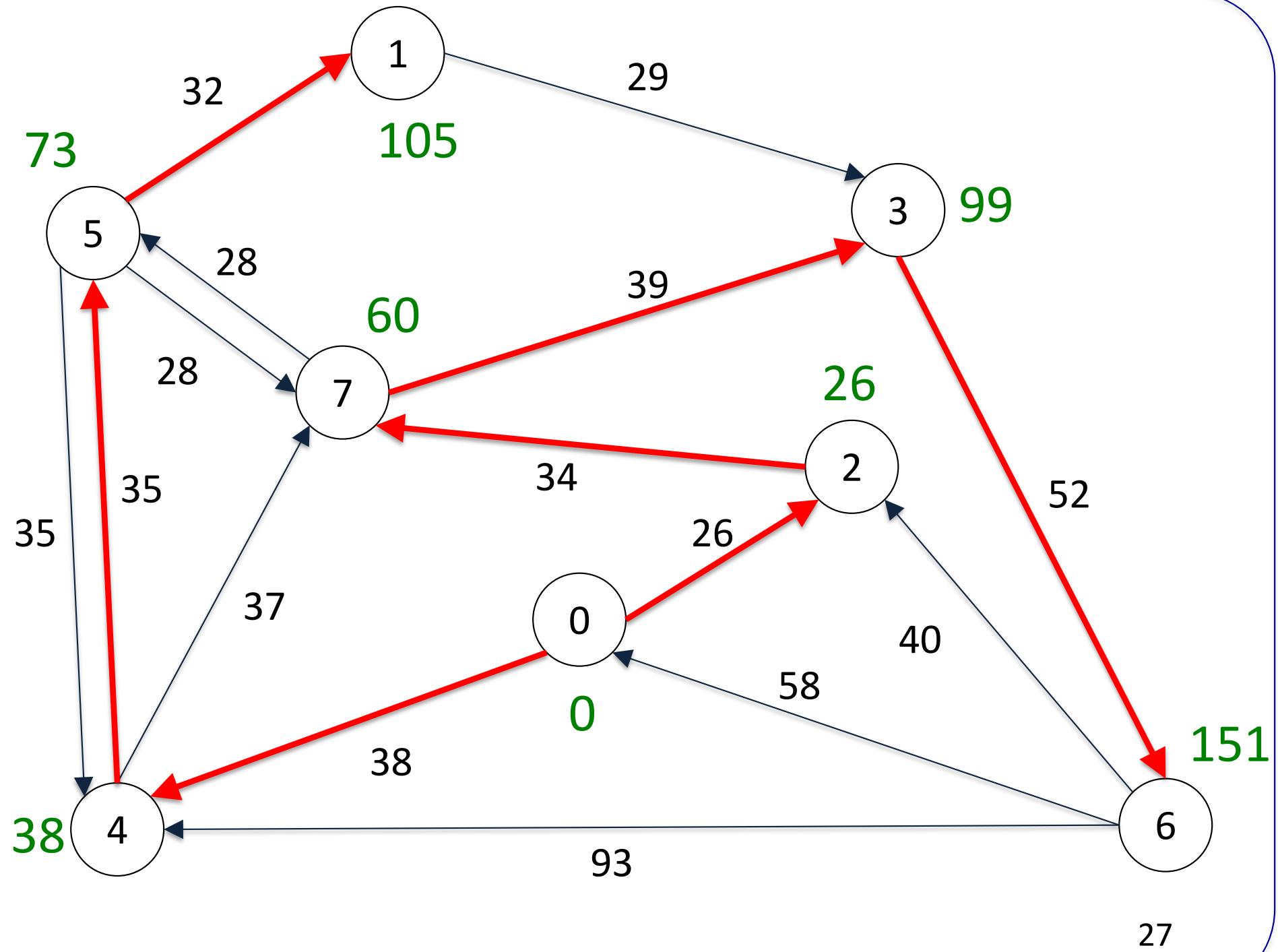












```
dijkstra(s):  
    init(s)  
    S =  $\emptyset$   
    q = new Queue(V)  
    while !q.empty():  
        u = q.xMin()  
        S = S  $\cup$  {u}  
        for each v in  $\alpha[u]$ :  
            reduce(u,v)
```

vértice fuente
(de partida)

u es el siguiente vértice
más cercano a s (la pri-
mera vez es el propio s)

lista de vértices
adyacentes a u

Invención:

- E. Dijkstra, 1959
- la técnica algorítmica se conoce como *codiciosa*

Validación:

- próxima diapositiva, mediante una demostración

Análisis:

- ya viene ...

1. Sea u el primer vértice tal que $d[u] \neq \delta(s, u)$ al ingresar a S
2. Sean p la ruta más corta de s a u
 - y el primer vértice en p tal que $y \notin S$
 - $x \in S$ el predecesor de y : $d[x] = \delta(s, x)$
3. Como la arista (x, y) fue reducida al ingresar x a S , entonces $d[y] = \delta(s, y)$ al ingresar u a S
4. Como y aparece antes que u en p y todos los costos son ≥ 0 , entonces $\delta(s, y) \leq \delta(s, u)$
 - ... y como $d[y] = \delta(s, y)$ y $d[u] \geq \delta(s, u)$, entonces $d[y] \leq d[u]$
5. Pero u fue elegido antes que y para ingresar a S , por lo que deducimos que $d[u] \leq d[y]$
6. Estas dos desigualdades implican que $d[u] = \delta(s, u)$

**Analizar un algoritmo es
predecir los recursos que el algoritmo necesita**

Principalmente, **tiempo de computación**:

- a veces, memoria, ancho de banda de comunicación, accesos al disco

... en un **computador (relativamente) convencional**:

- las instrucciones son ejecutadas una después de la otra
 - ... operaciones aritméticas comunes, movimientos simples de datos, e instrucciones de control
 - ... cada una toma una cantidad de tiempo constante
- los tipos de datos son *entero* y *punto flotante*
- no consideramos *caches* ni memoria virtual

Tiempo de computación:

Número de pasos, u operaciones primitivas, ejecutados

Definimos “paso” de modo que sea lo más independiente posible del computador:

- un segmento sintáctica o semánticamente significativo de un programa, cuyo tiempo de ejecución es independiente de las características del problema particular
- se necesita una cantidad de tiempo constante para ejecutar cada línea de nuestros programas (los de esta presentación)

Sea $G = (V, E)$

dijkstra realiza $|V| \text{ } \texttt{xMin}'s$ y $|E| \text{ } \texttt{reduce}'s$

Si la cola q es implementada como un heap binario,

... entonces cada una de las **xMin**'s y cada una de las actualizaciones de $d[v]$ en **reduce** toma tiempo $O(V)$

Así, **dijkstra** toma tiempo $O((V+E) \log V)$

Para Dijkstra, el tiempo de computación depende del **número de nodos y del número de arcos** —el *tamaño del input*:

- no es lo mismo calcular las rutas más cortas para un grafo de 100 vértices y 2,000 arcos
- ... que para un grafo de 10,000 vértices y 2,000,000 arcos

En general, distinguimos tiempo de computación en el **peor caso**

... tiempo de computación en el **mejor caso**

... tiempo de computación en el **caso promedio**

Orden de crecimiento y la notación O()

Al determinar el tiempo de computación de un algoritmo, ya sea para el peor caso o el caso promedio, no nos interesa tanto la fórmula “exacta”, sino poder acotar superiormente esa fórmula

Si $T(n)$ representa la fórmula exacta

... y $f(n)$ representa una fórmula más simple

... decimos que $T(n) = O(f(n))$ —se lee “ $T(n)$ es O de $f(n)$ ”

... si existen constantes positivas c y n_0 tales que

... $T(n) \leq cf(n)$ cuando $n \geq n_0$

Órdenes de crecimiento comunes

1	instrucción o paso	sumar dos números
$\log n$	dividir por la mitad	búsqueda binaria
n	<i>loop</i>	encontrar el máximo
$n \log n$	dividir para conquistar	mergeSort
n^2	<i>loop double</i>	revisar todos los pares
n^3	<i>loop triple</i>	revisar todos los tríos
2^n	búsqueda exhaustiva	revisar todos los subconjuntos

El problema de la subsecuencia de suma máxima

Dada una secuencia de n números enteros positivos y negativos, en un arreglo, encontrar la suma de la subsecuencia (consecutiva) cuya suma sea máxima entre todas las subsecuencias

P.ej., para el input $a = [-2, 11, -4, 13, -5, -2]$

... la respuesta es 20 (desde a_1 hasta a_3)

Este problema se puede resolver de diversas formas:

... con un algoritmo de tiempo $O(n^2)$:

- chequear todas las subsecuencias posibles

... también con uno de tiempo $O(n \log n)$

- aplicar *dividir para conquistar*, dividiendo el arreglo en la mitad

... e incluso con uno de tiempo $O(n)$:

- ¿en serio?

La ventaja de pasar de un algoritmo $O(n^2)$ a uno $O(n\log n)$ para el mismo problema

... es que ahora puedes resolver el problema para inputs de tamaño mucho más grande

... aproximadamente $n/\log n$ veces más grande

Tiempos de ejecución (en segundos) de varios algoritmos para el problema anterior

n	$O(n^3)$	$O(n^2)$	$O(n \log n)$	$O(n)$
100	0.00016	0.000006	0.000005	0.000002
1,000	0.096	0.00037	0.00006	0.00002
10,000	86.67	0.0333	0.00062	0.00022
100,000	NA	3.33	0.0067	0.0022
1,000,000	NA	NA	0.075	0.023

Bibliografía

T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms* (3rd ed.), MIT Press, 2009

S. Dasgupta, C. Papadimitriou, U. Vasirani, *Algorithms*, McGraw-Hill 2006

Horowitz, Sahni, Rajasekeran, *Computer Algorithms* (2nd ed.), Silicon Press 2008

R. Sedgewick, K. Wayne, *Algorithms* (4th ed.), Addison-Wesley 2011

M. Weiss, *Data Structures and Algorithm Analysis in C++* (4th ed.), Pearson 2013