Examen 2: Analisis y Diseño de Algoritmos

Andrey Arguedas Espinoza - 2020426569

November 22, 2022

1. Elabore un algoritmo divide y conquista que permita multiplicar n números complejos usando solamente 3(n-1) multiplicaciones reales.
   Recuerde la multiplicación compleja:
   (a + bi) * (c + di) = (ac –bd) + (ad + bc)i
   donde a, b, c, d son números reales, e i es la raíz cuadrada de -1.

   Recuerde la técnica usada para reducir el número de multiplicaciones en la multiplicación de enteros.

   Se acepta, aunque con menos puntos, un algoritmo que no sea divide y conquista pero que use solamente 3(n-1) multiplicaciones.

Como observamos en el metodo dado de la multiplicación compleja tenemos **cuatro multiplicaciones** en total, ejemplo para los siguientes numeros complejos $Z1 = (-3 - 5i)$ y $Z2 = (7 - 9i)$:

$$\begin{aligned} Z1 * Z2 &= [(-3 * 7) - (-5 * -9)] + [(-3 * -9) + (-5 * 7)]i \\ &= (-21 - 45) + (27 - 35)i \\ &= -66 + 8i \end{aligned} \tag{1}$$

Como podemos observar se realizaron cuatro multiplicaciones para multiplicar los dos números complejos, sin embargo podemos cambiar de metodo para solo aplicar **3 multiplicaciones**:

$$\begin{aligned} x1 &= c * (a + b) \\ x2 &= a * (d - c) \\ x3 &= b * (c + d) \\ real &= x1 - x3 \\ i &= x1 + x2 \end{aligned} \tag{2}$$

Con $Z1 = (-3 - 5i)$ y $Z2 = (7 - 9i)$ y usando el nuevo metodo obtenemos el mismo resultado:

$$\begin{aligned} x1 &= -7 * (-3 + -5) = -56 \\ x2 &= -3 * (-9 - 7) = 48 \\ x3 &= -5 * (7 + -9) = 10 \\ real &= x1 - x3 = -66 \\ i &= x1 + x2 = 8 \\ &= -66 + 8i \end{aligned} \tag{3}$$

Finalmente implementamos el algoritmo para N numeros complejos:

```
mult_operation(c1, c2):
    x1 = c2[0] * (c1[0] + c1[1])
    x2 = c1[0] * (c2[1] - c2[0])
    x3 = c1[1] * (c2[0] + c2[1])
    real =  x1 - x3
    imaginary =  x1 + x2
    return (real , imaginary)

# n_complex_numbers # List with more than 1 complex numbers to multiply

function mult_complex(n_complex_numbers)
    if length(n_complex_numbers) == 1:
        return n_complex_numbers[0]
    else if length(n_complex_numbers) == 2:
        return mult_operation(n_complex_numbers[0], n_complex_numbers[1])
    else:
        size = length(n_complex_numbers)
        n_half = floor(size / 2)
        left_n = n_complex_numbers[0...n_half]
        right_n = n_complex_numbers[n_half + 1...size]
        return mult_operation(left_n) * mult_operation(right_n)
```

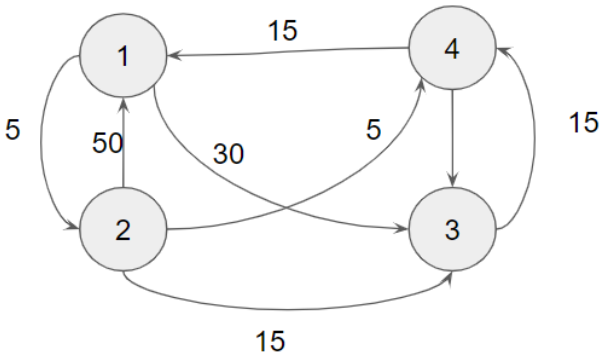
2. Se tiene un grafo dirigido G(V,E) con costos no negativos en las aristas. El algoritmo de Floyd encuentra por medio de programación dinámica el costo de las rutas con menor costo entre cada par de nodos *v,u* ∈ V.

```
function floyd (G(V,E),n)
  for i:=1 to n do
    for j:=1 to n do
      if (i,j) ∈ E
        then A[i,j] := costo de la arista (i,j)
        else A[i,j] := ∞
    A[i,i] := 0
  for k:=1 to n do
    for i:=1 to n do
      for j:=1 to n do
        if A[i,k]+A[k,j] < A[i,j]
          then A[i,j] := A[i,k] + A[k,j]
  return A
```

Modifique el algoritmo de Floyd para determinar el número de caminos pares y caminos impares entre cada par de vértices. El grafo que se recibe es el mismo pero los costos se ignoran.
Sugerencia: debe replantear el significado de A[i,j] y almacenar más de una cosa en cada entrada.

Para resolver este problema en lugar de guardar las distancias vamos a guardar la conexión inicial entre un nodo y otro usando un 1. Luego tendremos 2 matrices una para contar los saltos pares y otra para los impares, las rutas las obtendremos cuando desde i llegamos a j con mediante una ruta par o impar sabemos que si se da un salto más estamos en una ruta del tipo contrario.

Veamos como trabajaría nuestro algoritmo con el siguiente ejemplo:



A=

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

Antes de ejecutar el algoritmo:

pares=

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

impares=

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

Despues de ejecutar el algoritmo:

pares=

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

impares=

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

Pseudocodigo de la implementación:

```
function floyd (G(V,E),n)
    contador_par = [1...n][1...n]
    contador_impar = [1...n][1...n]
    for i:=1 to n do
        for j:=1 to n do
            if (i,j) exists in E
                then A[i,j] := 1 #Primer salto
                contador_impar[i, j] += 1 # Primera ruta impar
            else A[i,j] := 0

    for k:=1 to n do
        for i:=1 to n do
            for j:=1 to n do
                if A[i,k] and A[k,j] == 1 and k!=j and i!=j do: # Existe un s
                    ruta_par = contador_par[i,k]
                    ruta_impar = contador_impar[i,k]
                    tipo_salto = max(ruta_par, ruta_impar)
                    if tipo_salto == ruta_par:
                        contador_impar[i,j] += 1
                    else:
                        contador_par[i,j] += 1

    return A
```

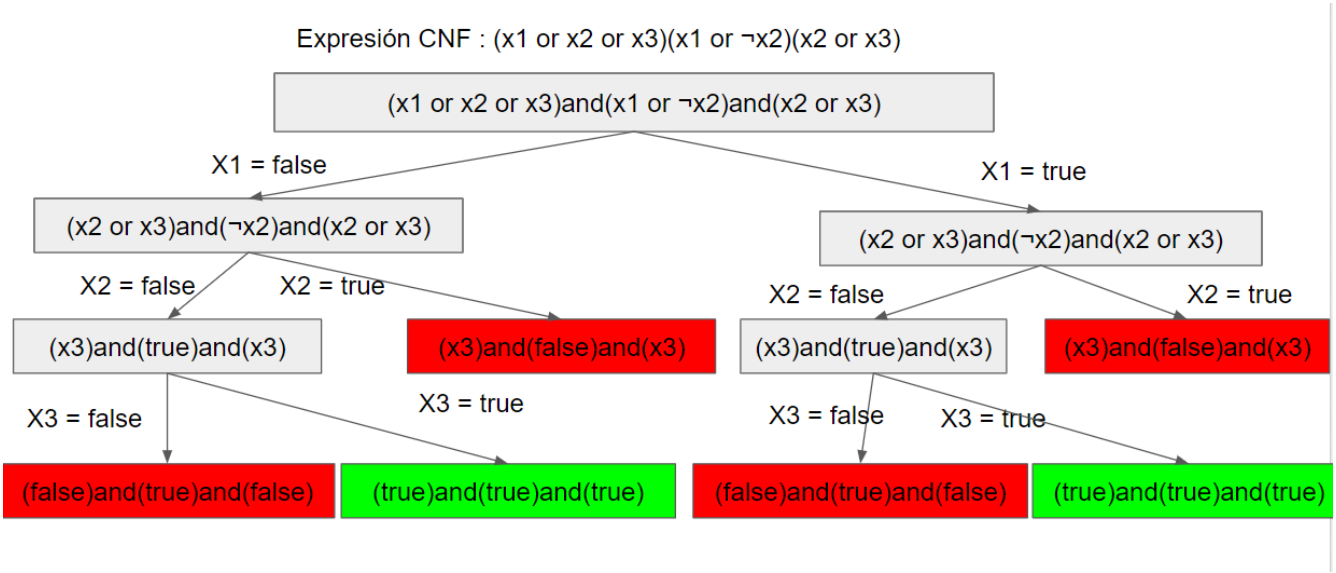
3. En el problema SAT se tienen cláusulas booleanas en forma normal conjuntiva cada una de ellas conteniendo un grupo arbitrariamente grande de literales:

   $(x_1$ **or** $x_4)$ **and** $(\neg x_4$ **or** x_3 **or** x_5 **or** $\neg x_{312})$ **and** $(x_2$ **or** $\neg x_3$ **or** x_7 **or** x_5 **or** x_{312} **or** x_{35} **or** $\neg x_7$ **or** ...) ...

   y se requiere encontrar una asignación de valores de verdad (falso, verdadero) a las variables $(x_1,x_2,...)$ que satisfaga toda la fórmula, si dicha asignación existe.

   Plantear un algoritmo de Las Vegas para encontrar una solución. Señale por qué es un algoritmo de Las Vegas y no un algoritmo de Monte Carlo.

Para poder resolver este algoritmo nos basaremos en una solución de backtracking la cual mejoraremos realizando un híbrido entre backtacking y un metodo **Las Vegas** para la generación de primeras opciones aleatorias. Para poder entender como funciona este algoritmo mostramos como trabaja el backtracking original para este problema.

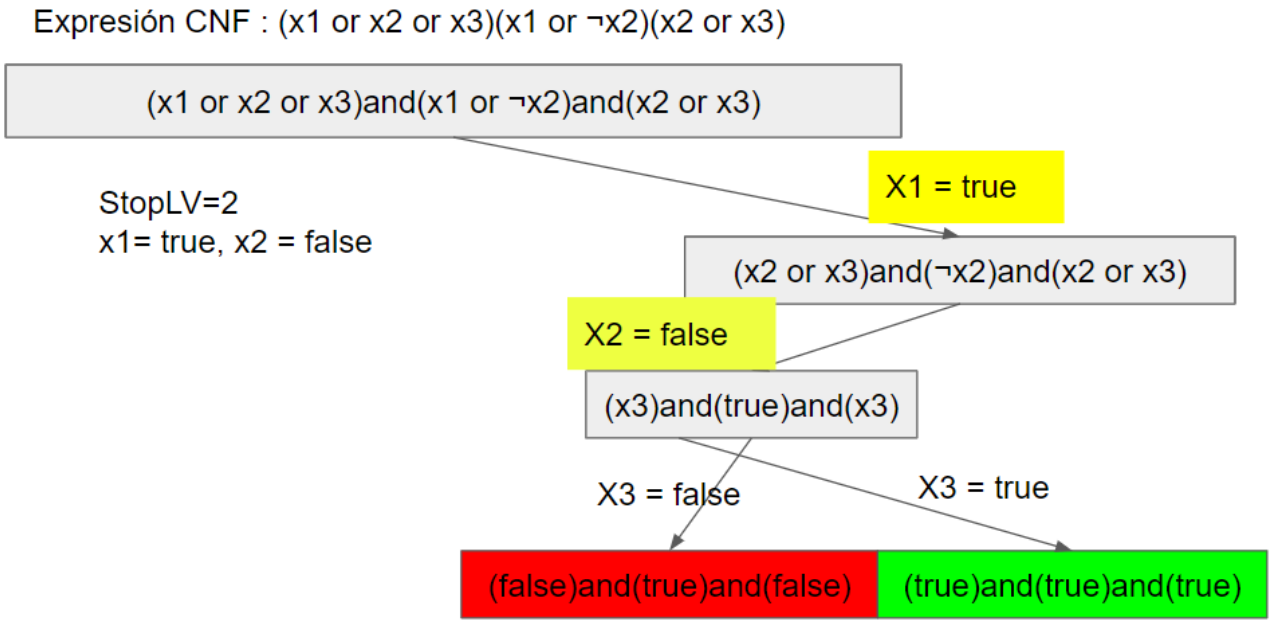


Como podemos observar los valores que dan soluciones factibles son : **x1=false, x2=false, x3=true** y **x1=true, x2=false, x3=true**

Una vez que encontramos una solucion no factible (las rojas) no seguimos explorando esa parte del árbol. Seguido de esto podemos usar valores aleatorios para asignarle a ciertas variables y así hacer la exploración del arbol más pequena, a continuación presentamos el pseudocodigo de este híbrido entre **backtracking** y **Las Vegas**.

```
function assignValuesLV(CNF, stopLv)
    for i=0 to stopLv-1 do:
        #Pick an unused variable of the CNF
        x = uniform(0...CNF.variables.length)
        value = uniform(true, false)
        replace(CNF, x, value)
    #CNF with changes
    return CNF

function SATLV(CNF, solution[1...n_variables])
    if CNF all clauses == true :
        return 'solution found'
    if CNF has unsolved clauses or eval(CNF) == false :
        return 'solution not found'

    #Pick next unused variable of the CNF
    x = CNF.next.variable
    value = uniform(true, false)
    replace(CNF, x, value)

    if SATLV(CNF, solution) == 'solution found':
        solution[x] = value
        return 'sat'

    value = !value #Probamos con el valor contrario

     if SATLV(CNF, solution) == 'solution found':
        solution[x] = value
        return 'sat'

    return 'unsatisfied'

function repeatLV(CNF, stopLv)
    newCNF = assignValuesLV(CNF, stopLv)
    repeat
        solution = [0...CNF.variables.length]
        success = SATLV(newCNF, solution)
    until success == 'sat'
    return solution
```

Por ejemplo con el pseudocodigo anterior supongamos que **StopLV=2**, por lo que solo asigna valores aleatorios de false—true a **x1** y **x2** ya con eso el arbol de bactracking se vería de la siguiente manera:



Debemos usar un algoritmo **Las Vegas** porque no podemos saber el numero de iteraciones mínimas para encontrar las asignaciones que satisfagan la fórmula. Tambien necesitamos que los valores asignados sean correctos y no solo una aproximación ya que si no la expresión será falsa.

4. Se sabe que el problema de decisión de ciclo hamiltoniano es NP-completo:

   Dado un grafo no dirigido, ¿existe un ciclo que pasa exactamente una vez por cada uno de los vértices?

   Se quiere probar que el Problema del Vendedor Viajero (TSP) es NP-completo:

   Dado un grafo no dirigido completo con pesos positivos en las aristas y un valor k, ¿existe un ciclo hamiltoniano cuyo peso total no sea mayor que k?.

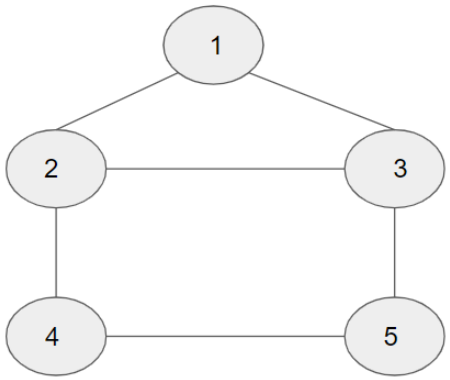
a) Diga qué hay hacer para probar que TSP es NP-completo.

b) Pruebe que TSP es NP-completo usando el problema de ciclo hamiltonianio que ya se sabe es NP-completo. Sugerencia: usar pesos para contar aristas.

Para poder demostrar que TSP es NP-completo lo que se tiene que hacer es utilizar reducción y demostrar que se pueda reducir a un problema Q que pertenece an NP y luego dar un algoritmo para construir la ruta cuyo peso es menor a K.

Por lo que vamos a reducir TSP al problema del ciclo hamiltoniano que ya sabemos que es NP-completo:

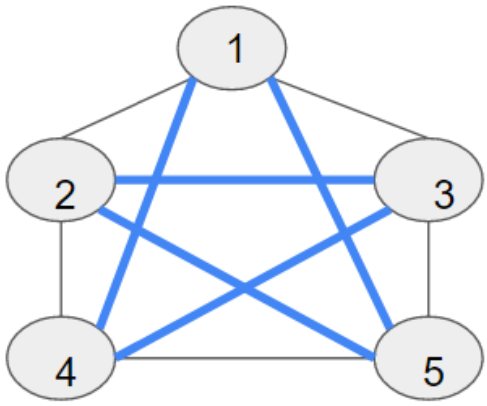
*** Teorema: El problema de TSP se puede reducir al problema del ciclo hamiltoniano**



Ejemplo de un grafo con ciclo Hamiltoniano, sobre el que trabajaremos

Pasos:

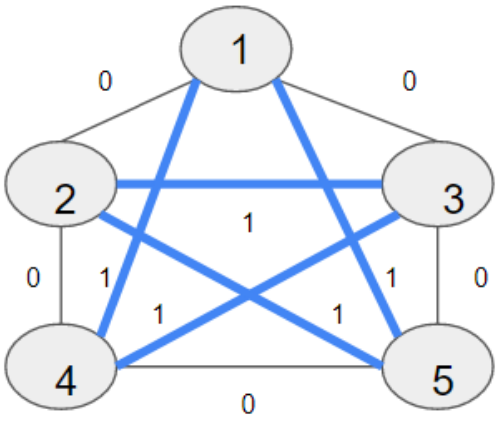
1- Transformamos el grafo G=(V, E) que tiene ciclo hamiltoniano en un grafo G'=(V, E'), los nodos son los mismos pero los arcos no, y construimos un grafo que es completo



2 - Definimos los costos del nuevo grafo como:

Si el arco $(i, j) \in G'(i, j) = 0$

Si el arco $(i, j) \notin G'(i, j) = 1$



Finalmente la solución a el problema TSP es cuya ruta tiene un costo de cero, por lo que se demuestra TSP reduciendo al problema de ciclos de Hamilton.

5. La sección ***11.4.2 Connected components*** del libro de texto presenta con detalle el algoritmo paralelo visto en clase para calcular las componentes conexas de un grafo no dirigido. Basándose en esa sección resuelva el problema ***11.9*** del libro de texto.

Hacer lo siguiente:

- **Mostrar el estado inicial.**
- **Para la primera iteración, mostrar el estado de avance luego de completarla toda (sin desglosarla por etapas).**
- **Para la <u>segunda iteración</u>, mostrar el estado de avance del luego de completar <u>cada una de las tres etapas</u> de esa iteración**
- **Para iteraciones adicionales, mostrar el estado de avance luego de completar cada iteración (sin desglosarla por etapas).**

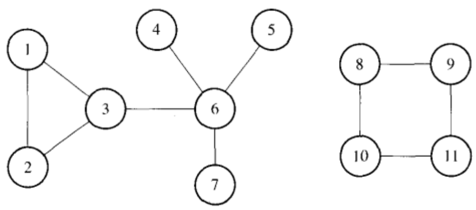


Figure 11.19. A graph

Iniciamos la ejececución del algoritmo

```
Connected Components!!!

********* Estado inicial **********

L=
[0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0]

s= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

T= [inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]

oldT= [inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]

S=
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- **Para la primera iteración, mostrar el estado de avance luego de completarla toda (sin desglosarla por etapas).**

```
********* Al final de la iteración 3 tenemos **********

s= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

T= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

oldT= [1, 0, 0, 5, 5, 2, 5, 8, 7, 7, 8]

S=
[1, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, 0, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, 0, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, 5, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, 5, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, 2, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, 5, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, 8, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, 7, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, 7, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, 8]
```

- **Para la <u>segunda iteración</u>, mostrar el estado de avance del luego de completar <u>cada una de las tres etapas</u> de esa iteración**

```
********* Paso 1 **********

S=
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]

T= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

T= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

s= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

********* Paso 2 **********

S=
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
[inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]

T= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

T= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

s= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

********* Paso 3 **********

oldT= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]

T= [0, 0, 0, 0, 0, 0, 0, 7, 7, 7, 7]
```

- **Para iteraciones adicionales, mostrar el estado de avance luego de completar cada iteración (sin desglosarla por etapas).**

No es necesario hacer más iteraciones ya que quedó resuelto en la anterior, podemos ver como los nodos se conectan al 1 y al 8 respectivamente.

6. Suponga que ε es una cantidad muy pequeña pero mayor que cero. Si tengo objetos de tamaño 1/2+ε, 1/2 - ε, se pueden empacar así: [1/2+ε, 1/2-ε], pero no así: [1/2+ε, 1/2+ε]. Suponga que ε es menor que cualquier fracción que use. Por ejemplo $0 < \varepsilon < 1/10^{100000}$.

   Se tiene el siguiente grupo de objetos:
   1/2+ε, 1/2, 1/4+ε, 1/4, 1/4-ε,1/4-ε.

   Este grupo se puede empacar óptimamente en dos cajones:
   [1/2+ε, 1/4, 1/4-ε], [1/2, 1/4+ε, 1/4-ε].

   Sin embargo FFD lo pone en tres cajones:
   [1/2+ε, 1/4+ε], [1/2, 1/4, 1/4-ε], [1/4-ε].

   Encontrar secuencias similares para los siguientes casos:
   - **número óptimo de cajones es 3, pero FFD asigna 4**
   - **número óptimo de cajones es 4, pero FFD asigna 5.**

Presentamos los siguientes casos:

- **número óptimo de cajones es 3, pero FFD asigna 4**

Objetos = ½ +ε , ½ , ½ -ε , ⅓ +ε , ⅓, ⅓ -ε , ¼ +ε , ¼ -ε

FFD	Número óptimo de cajones
BIN 1 = [½ +ε , ½ -ε]	BIN 1 = [⅓ +ε, ⅓, ⅓ -ε]
BIN 2 = [½ , ⅓ +ε]	BIN 2 = [½ , ⅓ +ε]
BIN 3 = [⅓, ⅓ -ε , ¼ -ε]	BIN 3 = [½ +ε, ½ -ε]
BIN 4 = [¼ +ε]	

- **número óptimo de cajones es 4, pero FFD asigna 5.**

Objetos = ½ +ε , ½ , ½ -ε , ⅓ +ε , ⅓, ⅓, ⅓, ⅓ -ε , ¼ +ε , ¼, ¼

FFD	Número óptimo de cajones
BIN 1 = [½ +ε , ½ -ε]	BIN 1 = [½, ¼ , ¼]
BIN 2 = [½ , ⅓ +ε]	BIN 2 = [⅓ +ε , ⅓ -ε , ¼ +ε]
BIN 3 = [⅓, ⅓, ⅓]	BIN 3 = [⅓, ⅓, ⅓]
BIN 4 = [⅓ -ε , ¼ +ε , ¼]	BIN 4 = [½ +ε , ½ -ε]
BIN 5 = [¼]	