EJEMPLOS DE PROBLEMAS RESUELTOS

UNIVERSIDAD DE MÁLAGA
E.T.S. INGENIERÍA INFORMÁTICA

Fco Javier Moreno Barea

Caminos más cortos desde un origen

- Dado un grafo conexo y etiquetado con pesos, y un vértice *s*, el problema consiste en encontrar los caminos más cortos que conectan a s con el resto de vértices del grafo.
- La mejor solución conocida hasta ahora se llama el algoritmo de Dijkstra, que puede aplicarse sólo cuando los pesos son no negativos.

- Intuitivamente, el algoritmo de Dijkstra realiza $|V|-1$ iteraciones:
  - En la primera iteración encuentra el vértice más cercano a *s*
  - En la segunda, el vértice segundo más cercano a *s*. …
  - En la i-ésima iteración, encuentra el i-ésimo vértice más cercano a *s*. …

- Supongamos que hemos realizado *i* iteraciones, y hemos construido el árbol T_i con los *i* vértices más cercanos a *s*, para construir T_{i+1}:
  - Creamos el conjunto *F* de todos los vértices adyacentes a T_i, es decir, los que no están en T_i pero que comparten una arista con algún vértice de T_i
  - Para cada $u \in F$, calculamos la distancia de u a s, y escogemos el que esté más cerca
    - si hay varios con la misma distancia más corta, cualquiera de ellos

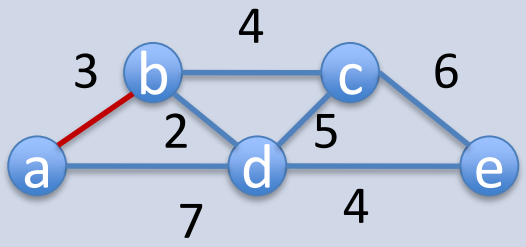
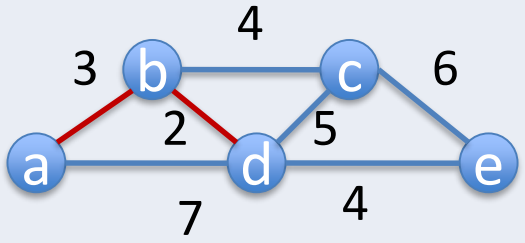
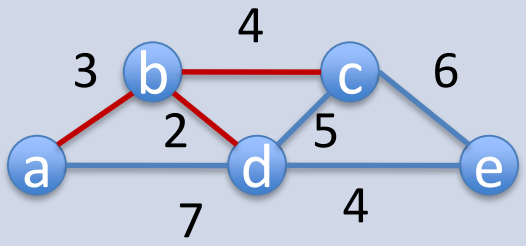
Caminos más cortos desde un origen

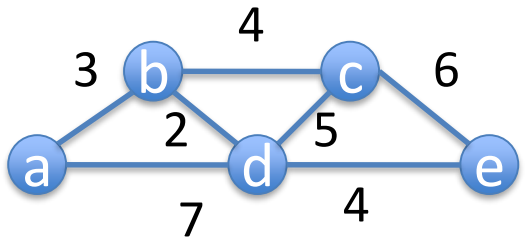
- Para calcular la distancia de cada $u \in F$ a s mantenemos una cola de prioridades Q con los vértices que quedan por encontrar, ordenados por su distancia a s. Q se actualiza en cada iteración.

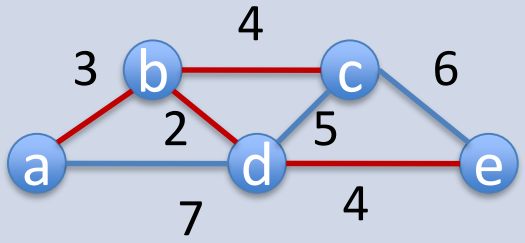
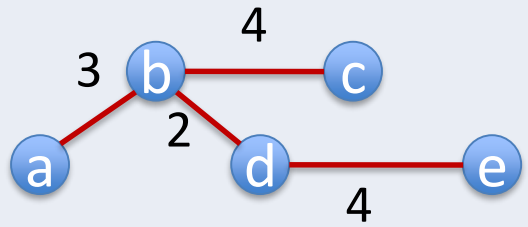
- Los elementos de Q tienen la siguiente forma v(w,d), donde
  - v es uno de los vértices que falta por encontrar
  - w es un vértice cuya distancia mínima a s ya se ha calculado
  - d es la distancia mínima de v a s, que se ha calculado como la distancia de v a w, más la distancia de w a s.
  - la clave del algoritmo está en que esta última distancia ya ha sido calculada por el algoritmo en la iteración anterior, y no hay que volver a calcularla.

```
Algoritmo Dijkstra(G, s)
    // G=(V,E) es el grafo de entrada
    // s es uno de los vértices de V
    // la salida es un conjunto de aristas
    // E_T con el árbol de las distancias mínimas a s
    E_T = {s(-,0)}
    S = V – {s}
    Q = {v(s,d) | (s,v) ∈ E, w(s,v)=d}

    while (S ≠ ∅ ){
        u(v,d) = extrae_min(Q);
        E_T = E_T ∪ {u(v,d)}; S = S – {u};

        para cada x ∈ S tal que (u,x) ∈ E_T
            si (x(-,d') ∈ Q, d + w(u,x)<d')
                Q = Q – {x(-,d')} ∪ {x(u, d + w(u,x))}
    }

    Devolver E_T
```

Ejemplo de ejecución

Vértices	Cola	Ilustración
a(-,0)	b(a,3),d(a,7), c(-,∞),e(-,∞)	
a(-,0), b(a,3)	d(b,5),c(b,7), e(-,∞)	
a(-,0), b(a,3), d(b,5)	c(b,7),e(d,9)	



Vértices	Cola	Ilustración
a(-,0), b(a,3), d(b,5), c(b,7)	e(d,9)	
a(-,0), b(a,3), d(b,5), c(b,7), e(d,9)		

El algoritmo de Dijkstra: corrección y complejidad

- Si se ordenan los caminos óptimos desde *s* hacia los otros *n-1* nodos, el i-ésimo camino menor se encuentra en la iteración i. Puede demostrarse por inducción:
  - En la primera iteración se encuentra el camino más corto
  - Suponiendo que en la iteración i-1 se han encontrado los i-1 caminos más cortos, el i-ésimo vértice u es el más cercano a s a través de los vértices explorados. Cualquier otro vértice es más lejano, por lo que no puede haber un camino más corto a u.

- Dada la similitud con el algoritmo de Prim, pueden emplearse estructuras de datos parecidas para la implementación del algoritmo de Dijkstra, por ejemplo, un montículo para mantener ordenados los vértices por la distancia al origen.

- Por lo tanto, Dijkstra tiene una complejidad del orden $O(|E|\log n)$. Si se desea calcular el camino óptimo entre cualquier par de nodos, puede repetirse el proceso *n* veces desde todos los orígenes posibles. La complejidad es, entonces, $O(|E|n\log n)$. Si el grafo no es muy denso, este algoritmo es mejor que el de Floyd de complejidad $O(n^3)$

1. Dada una fracción a/b se desea expresarla como la suma de un número mínimo de fracciones distintas con numerador unidad, ejemplo, 3/5 = 1/2 + 1/10.

Este ejercicio es complejo de solucionar si no nos fijamos bien en el ejemplo. En él tenemos:

$3/5 = 1/2 + 1/10$ donde $3/5 = 0.6 = 0.5 + 0.1$

Como podemos llegar, pues 5/3 = 1.6667, que si lo redondeamos nos queda c = 2.

Este es el primer termino, y si hacemos

3/5 – 1/2 = 6/10 – 5/10 = 1/10 obtenemos el segundo.

Otro ejemplo, a/b = 6/14 = 1/3 + 1/11 + 1/231.

14/6 = 2.3334 -> ceil -> 3

6/14 – 1/3 = 2/21

21/2 = 10.5 -> ceil -> 11

2/21 – 1/11 = 1/231

```
static void printEgyptian(double nr, double dr) {
        double rest = nr/dr;

        while (rest > 0) {
                int c = (int) Math.ceil(dr/nr);
                System.out.println("1/" + c);

                int mcm = mcm(c, (int) dr);
                nr = nr * mcm/dr - mcm/c; dr = mcm;
                rest = nr/dr - 1/c;
        }
}
```

El algoritmo Greedy funciona porque una fracción siempre se reduce a una forma en la que el denominador es mayor que el numerador y el numerador no divide al denominador. Para tales formas reducidas, la siguiente iteración se realiza para el numerador reducido. Entonces, las siguientes iteraciones continúan reduciendo el numerador hasta que llega a 1.

5. Vamos a ir en coche desde Málaga a Bilbao a ver el Guggenheim. Salimos con el depósito lleno, y sabemos que con ese combustible podemos recorrer M kilómetros. Tenemos un mapa en el que figuran los puntos kilométricos $k(1)$; …; $k(n)$ en los que hay gasolineras, y queremos determinar en cuáles de ellas hay que repostar para llegar al destino haciendo el menor número de paradas.

Lo que vamos a hacer es crear un nuevo array que para cada elemento tenga la distancia a la siguiente gasolinera. De esta forma, si partimos de un array [1,3,4,6,7,10], nuestro destino está en 11, y nuestro depósito da para recorrer 4 kilómetros, tendríamos un nuevo array [1,2,1,2,1,3,1].

Con este nuevo array vamos a comprobar en cada caso si podemos llegar a la siguiente gasolinera, es decir, si con el combustible que hay actualmente en el depósito (D) podemos recorrer la distancia siguiente.

Ejemplo: i = 0 – k = 0 – D = 4 – [1, 2, 1, 2, 1, 3, 1] ¿Podemos llegar al k=1? Sí, reducimos D, aumentamos k.
i = 1 – k = 1 – D = 3 – [2, 1, 2, 1, 3, 1] ¿Podemos llegar al k=3? Sí, reducimos D, aumentamos k.
i = 2 – k = 3 – D = 1 – [1, 2, 1, 3, 1] ¿Podemos llegar al k=4? Sí, reducimos D, aumentamos k.
i = 3 – k = 4 – D = 0 – [2, 1, 3, 1] ¿Podemos llegar al k=6? No, apuntamos parada en i=3, reiniciamos el depósito D a su tamaño.

5. Vamos a ir en coche desde Málaga a Bilbao a ver el Guggenheim. Salimos con el depósito lleno, y sabemos que con ese combustible podemos recorrer M kilómetros. Tenemos un mapa en el que figuran los puntos kilométricos $k(1);\ \ldots;\ k(n)$ en los que hay gasolineras, y queremos determinar en cuáles de ellas hay que repostar para llegar al destino haciendo el menor número de paradas.

Lo que vamos a hacer es crear un nuevo array que para cada elemento tenga la distancia a la siguiente gasolinera. De esta forma, si partimos de un array [1,3,4,6,7,10], nuestro destino está en 11, y nuestro depósito da para recorrer 4 kilómetros, tendríamos un nuevo array [1,2,1,2,1,3,1].

Ejemplo: i = 3 – k = 4 – D = 4 – [2, 1, 3, 1] ¿Podemos llegar al k=6? Sí, reducimos D, aumentamos k.
i = 4 – k = 6 – D = 2 – [1, 3, 1] ¿Podemos llegar al k=7? Sí, reducimos D, aumentamos k.
i = 5 – k = 7 – D = 1 – [3, 1] ¿Podemos llegar al k=10? No, apuntamos parada en i=5, reiniciamos el depósito D a su tamaño.
i = 5 – k = 7 – D = 4 – [3, 1] ¿Podemos llegar al k=10? Sí, reducimos D, aumentamos k.
i = 6 – k = 10 – D = 1 – [1] ¿Podemos llegar al k=11? Sí, reducimos D, aumentamos k.

k = 11 – Por lo que hemos llegado al destino.

5. Vamos a ir en coche desde Málaga a Bilbao a ver el Guggenheim. Salimos con el depósito lleno, y sabemos que con ese combustible podemos recorrer M kilómetros. Tenemos un mapa en el que figuran los puntos kilométricos $k(1);\ \ldots;\ k(n)$ en los que hay gasolineras, y queremos determinar en cuáles de ellas hay que repostar para llegar al destino haciendo el menor número de paradas.

```
int[] paradas_gasolineras (int[] gas, int M, int dest) {

    List<Integer> paradas = new ArrayList<Integer>();
    int[] dist = new int[] (gas.size() + 1);
    int aux = 0, D = M, k = 0, i = 0;

    for (int t = 0; t < gas.size (); t++) {
        dist [t] = gas [t] - aux;     aux += gas [t];
    }
    while (k < dest && j < dist.length) {
        if (dist [i] <= D) {
            D -= dist [i];        k += dist [i];      i++;
        } else {
            D = M;                paradas.add(j);
        }
    }
    return paradas;
}
```

Consideremos una solución optima OPT (su valor $|V_{opt}|$ es mínimo) y supongamos que en dicha solución los ítems no se añaden como hacemos.

Entonces, tendríamos que añadimos una parada más antes de llegar a la gasolinera que nosotros añadimos, y tendríamos que al llegar a la gasolinera j, el número de paradas sería uno más: $|V'_j| = |V_{j-1}| + 1$

Si la siguiente gasolinera está a más distancia de lo que alcanza el depósito, resulta que tenemos que añadir además la gasolinera j, por lo que:

$$|V'_{j+1}| = |V'_j| + 1 = |V_{j-1}| + 2 > |V_{j+1}| = |V_{j-1}| + 1$$

Luego $|V_{opt}| - |V| > 0$, es decir, $|V_{opt}| > |V|$. Luego la solución voraz V es óptima, como se quería demostrar.

6. Tenemos que planificar la gira veraniega del grupo de teatro de Informática. Tenemos n propuestas, cada una de las cuales consta de una fecha de inicio d_i, un número de actuaciones a_i (siempre una al día), y una oferta económica m_i. Seleccionar las propuestas que aceptaremos para maximizar el beneficio económico.

Si pensamos de forma pausada el problema, llegamos a la conclusión de que las propuestas deben compararse según su fecha de fin ($d_i + a_i$) , ya que si finaliza antes nos da más oportunidades de tener en cuenta otras propuestas, pero también tendremos que tener en cuenta su oferta económica, ya que, ante dos actividades que terminen simultáneamente, nos beneficia coger aquella cuya oferta sea mayor.

De modo que el algoritmo voraz consiste sencillamente en ordenar los ítems primero por día de fin, y después por beneficio, e ir tomando las actividades mientras las demás hayan finalizado.

La ordenación toma tiempo $\Theta(n \log n)$; y el recorrer las actividades nos lleva tiempo $\Theta(n)$. Por lo que es $\Theta(n \log n)$.

i	d_i	a_i	m_i	$d_i + a_i$
1	1	5	50	6
2	3	3	36	6
3	5	3	42	8
4	6	3	27	9
5	7	1	30	9

i	d_i	a_i	m_i	$d_i + a_i$
1	1	5	50	6
2	3	3	36	6
3	5	3	42	8
5	7	1	30	9
4	6	3	27	9

6. Tenemos que planificar la gira veraniega del grupo de teatro de Informática. Tenemos n propuestas, cada una de las cuales consta de una fecha de inicio d_i, un número de actuaciones a_i (siempre una al día), y una oferta económica m_i. Seleccionar las propuestas que aceptaremos para maximizar el beneficio económico.

```
public static  List<Integer> plan_gira(int[] d, int[] f, int[] m){
    // d y f son los arrays con el inicio y final de cada propuesta.

    int n = d.length-1;
    List<Integer> lista = new ArrayList<Integer>();
    lista.add(1);
    int k = 1;

    for (int i = 2; i <= n; i++){
        if (d[i] >= f[k]){
            lista.add(i);
            k = i;
        }
    }
    return lista;
}
```

i	d_i	a_i	m_i	$d_i + a_i$
1	1	5	50	6
2	3	3	36	6
3	5	3	42	8
4	6	3	27	9
5	7	1	30	9

i	d_i	a_i	m_i	$d_i + a_i$
1	1	5	50	6
2	3	3	36	6
3	5	3	42	8
5	7	1	30	9
4	6	3	27	9

6. Tenemos que planificar la gira veraniega del grupo de teatro de Informática. Tenemos n propuestas, cada una de las cuales consta de una fecha de inicio d_i, un número de actuaciones a_i (siempre una al día), y una oferta económica m_i. Seleccionar las propuestas que aceptaremos para maximizar el beneficio económico.

Pero, realmente este es un enfoque óptimo.
Vamos a poner otro ejemplo:

Si cambiamos las actividades y beneficio de la actividad 3 por $a_i = 4, m_i = 76$, tendríamos que la actividad 3 no se cogería por terminar más tarde, en el día 7, pero el máximo beneficio es coger las actividades 3 y 5, siendo su beneficio de 106.

Podríamos pensar entonces en ordenarlos por beneficio, pero podríamos dar otro ejemplo donde, cambiamos la oferta de la actividad 4 por $m_i = 47$, de forma que ahora el máximo beneficio sería tomar las actividades 1 y 4, siendo su beneficio de 109.

Al igual que el problema de la mochila, no es posible crear un algoritmo voraz para este problema que seleccione siempre el valor óptimo que queremos. Para ello tendrían que dejarnos cortar las actuaciones, siendo un problema continuo.

i	d_i	a_i	m_i	$d_i + a_i$
1	1	5	50	6
2	3	4	76	7
3	5	3	42	8
5	7	1	30	9
4	6	3	27	9

i	d_i	a_i	m_i	$d_i + a_i$
2	3	4	76	7
1	1	5	62	6
4	6	3	47	9
3	5	3	42	8
5	7	1	30	9

7. Supongamos que en una galería de arte hay $n \geq 1$ pinturas dispuestas en un pasillo de longitud $L > 0$, en las posiciones $\{x_1, \ldots, x_n\}$ $(x_i \in \mathbb{R})$. Supongamos que un vigilante puede asegurar las pinturas que están a una distancia de, a lo sumo, una unidad desde su posición. Diseñar un algoritmo voraz que encuentre el mínimo número de vigilantes necesarios para que todos los cuadros estén vigilados de forma segura.

Partimos de un array [0, 1, 2.5, 3.1, 4.9, 5.7, 6.5, 7.4, 7.9], y la longitud del pasillo es 8. Con este array vamos a comprobar en cada caso si podemos llegar al siguiente cuadro colocando guardias a distancia 1 del primer cuadro no vigilado.

Ejemplo: Si tenemos el primer cuadro en x = 0.1, ponemos el primer guardia a v = x + 1.

Ahora el propósito es comprobar si este guardia vigila más cuadros, para ello miramos si el siguiente cuadro está a menos de 1 de distancia del vigilante que ahora está en 1.1. Como está en 1, si está en la posición necesaria ya que está a 1.1 – 1 = 0.1 del vigilante.

El siguiente cuadro está en 2.5, de forma que 1.1 – 2.5 = -1.4, que en valor absoluto es > 1.

No se puede vigilar, por lo que ponemos un nuevo guardia en v = 2.5 + 1 = 3.5, así que se puede vigilar.

También el cuadro 4, que está en 3.1, porque 3.5 – 3.1 = 0.4 < 1. También se puede vigilar el cuadro 5, 3.5 – 4.4 = 0.9 < 1.

El siguiente cuadro no puede vigilarse, 3.5 – 5.7 = 2.2 > 1, por lo que ponemos un nuevo guardia en v = 5.7 + 1 = 6.7, así que se puede vigilar, y seguimos el proceso hasta llegar que vigilante + 1 >= 8, que es la longitud máxima.

7. Supongamos que en una galería de arte hay $n \geq 1$ pinturas dispuestas en un pasillo de longitud $L > 0$, en las posiciones $\{x_1, \ldots, x_n\}$ $(x_i \in \mathbb{R})$. Supongamos que un vigilante puede asegurar las pinturas que están a una distancia de, a lo sumo, una unidad desde su posición. Diseñar un algoritmo voraz que encuentre el mínimo número de vigilantes necesarios para que todos los cuadros estén vigilados de forma segura.

```
int[] colocar_vigilantes (int[] cuadros, int L) {

    List<Integer> vigilantes = new ArrayList<Integer>();
    int v = cuadros[0] + 1;

    vigilantes.add(v);
    int i = 1;

    while ((v + 1) < L && i < cuadros.size()) {
        if (Math.abs(v – cuadros[i]) > 1) {
            v = cuadros[i] + 1;
            vigilantes.add(v);
        }
        i++;
    }
    return vigilantes;
}
```

Consideremos una solución optima OPT (su valor $|V_{opt}|$ es mínimo) y supongamos que en dicha solución los ítems no se añaden como hacemos.

Entonces, tendríamos que añadimos un vigilante más antes de llegar al cuadro que nosotros tenemos como vigilado, y tendríamos que al llegar al cuadro j, el número de vigilantes sería uno más: $|V'_j| = |V_{j-1}| + 1$

Si el siguiente cuadro está a más distancia de lo que alcanza a vigilar, resulta que tenemos que añadir además un vigilante para el cuadro j, por lo que:

$$|V'_{j+1}| = |V'_j| + 1 = |V_{j-1}| + 2 > |V_{j+1}| = |V_{j-1}| + 1$$

Luego $|V_{opt}| - |V| > 0$, es decir, $|V_{opt}| > |V|$. Luego la solución voraz V es óptima, como se quería demostrar.

10. Dado un conjunto de números naturales $S = \{s_1, \dots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$) tal que se minimiza

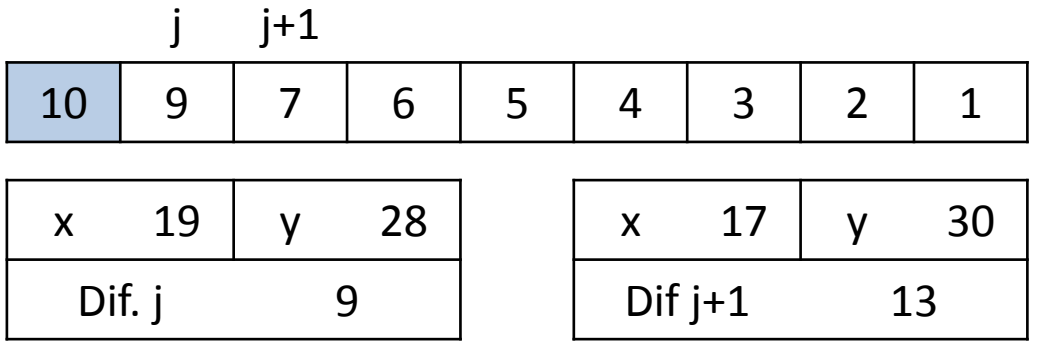
$$\left| \sum_{x \in S_1} x - \sum_{y \in S_2} y \right|$$

Si dedicamos tiempo al problema encontramos una solución que da un resultado óptimo. Lo que hemos de hacer es ordenar el array de forma descendente, y en primer lugar coger el primer elemento del array.

Una vez hecho esto, recorremos el array restante comprobando si al coger el elemento j, su diferencia es menor que al coger el elemento j+1, si no, se incrementa j.

Cuando tengamos que esto se cumple, se introduce dentro de S_1 el elemento j, y repetimos el proceso hasta que hemos recorrido todos los elementos del array.

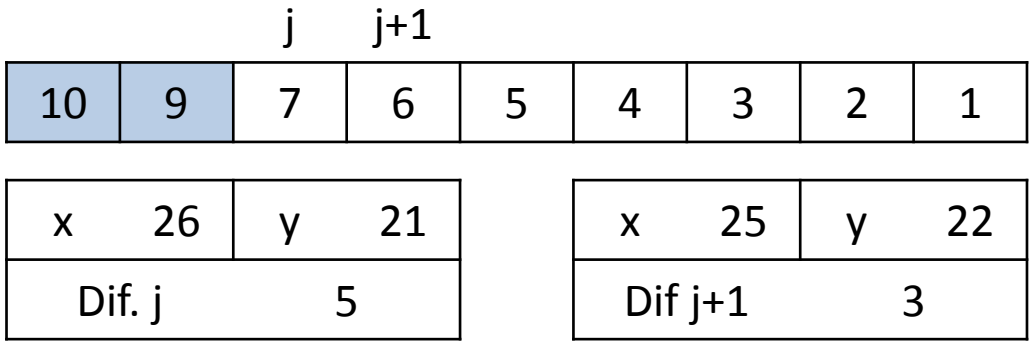
Imaginemos que tenemos el array [10,2,6,4,9,1,7,3,5].

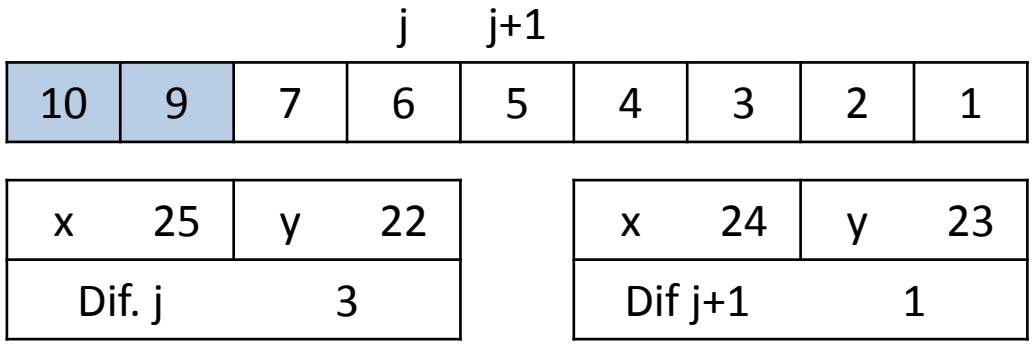


Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9

10. Dado un conjunto de números naturales $S = \{s_1, \dots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$) tal que se minimiza

$$\left| \sum_{x \in S_1} x - \sum_{y \in S_2} y \right|$$



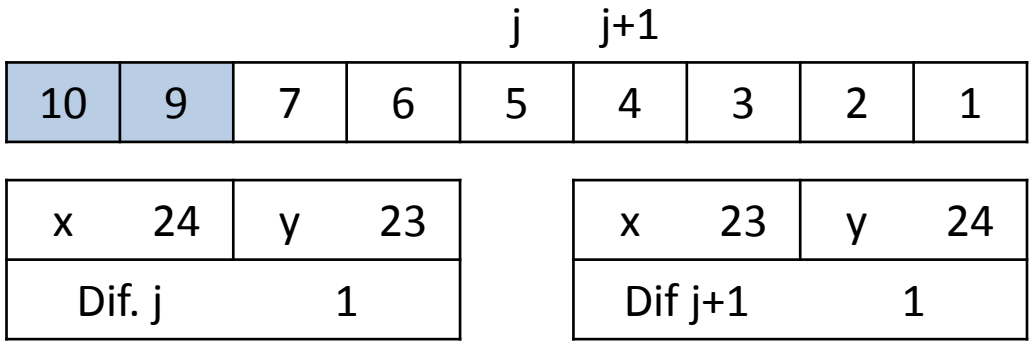


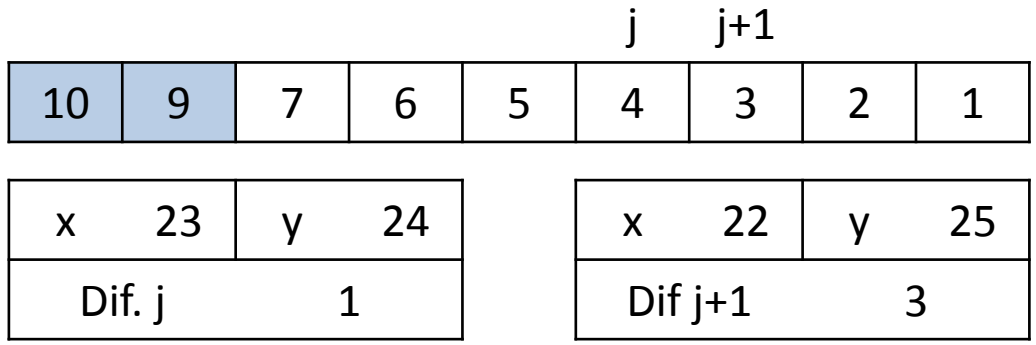
Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9

Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9

10. Dado un conjunto de números naturales $S = \{s_1, \ldots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$) tal que se minimiza

$$\left| \sum_{x \in S_1} x - \sum_{y \in S_2} y \right|$$



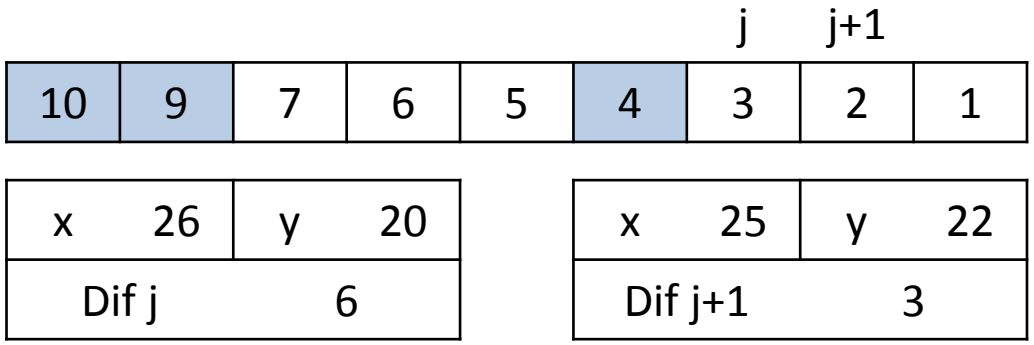


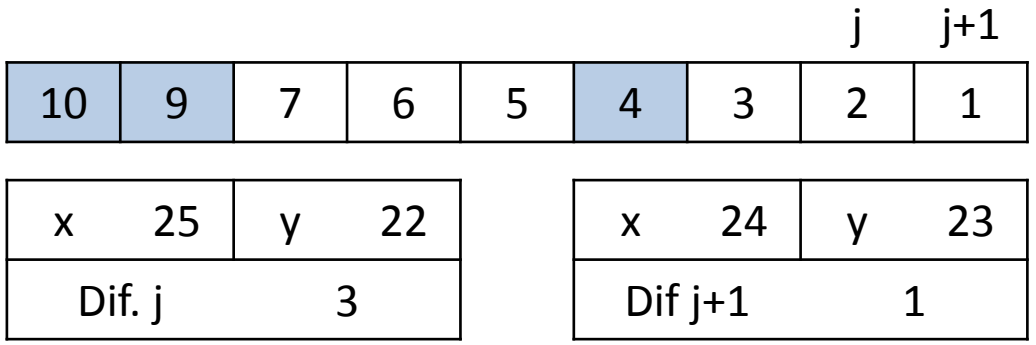
Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9

Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9
[10,9,4]	[7,6,5,3,2,1]	23	24	1

10. Dado un conjunto de números naturales $S = \{s_1, \dots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$) tal que se minimiza

$$\left| \sum_{x \in S_1} x - \sum_{y \in S_2} y \right|$$



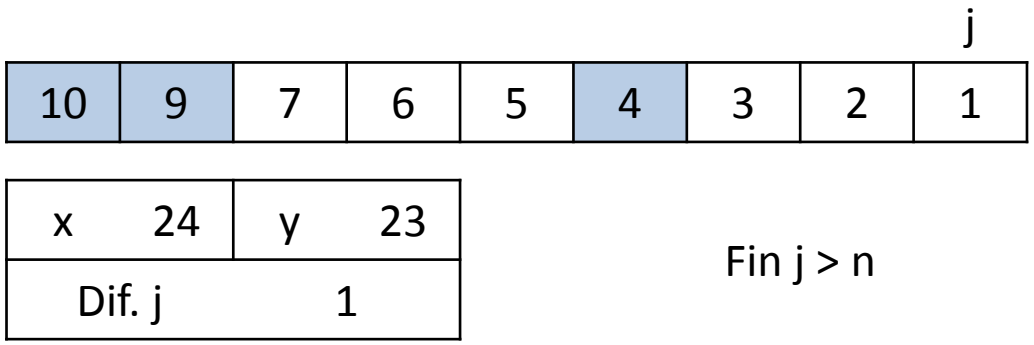


Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9
[10,9,4]	[7,6,5,3,2,1]	23	24	1

Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9
[10,9,4]	[7,6,5,3,2,1]	23	24	1

10. Dado un conjunto de números naturales $S = \{s_1, \dots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$) tal que se minimiza

$$\left| \sum_{x \in S_1} x - \sum_{y \in S_2} y \right|$$



Actual	Restante	x	y	Dif.
[]	[10,9,7,6,5,4,3,2,1]	0	47	47
[10]	[9,7,6,5,4,3,2,1]	10	37	27
[10,9]	[7,6,5,4,3,2,1]	19	28	9
[10,9,4]	[7,6,5,3,2,1]	23	24	1
FIN Óptimo encontrado				

Imaginemos que tenemos el array $[10,2,6,4,9,1,7,3,5]$.

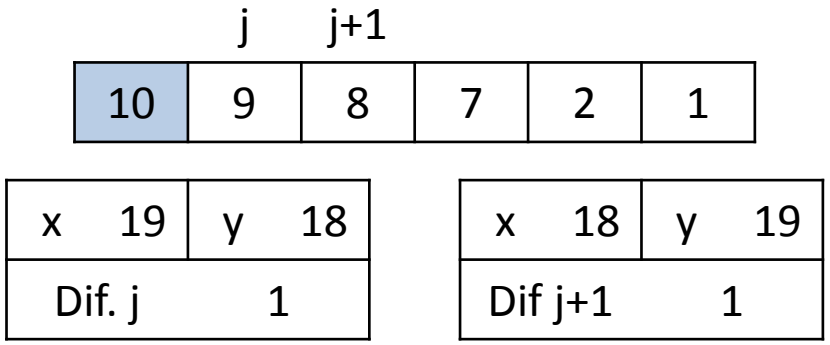
Tenemos al final un resultado que es óptimo.

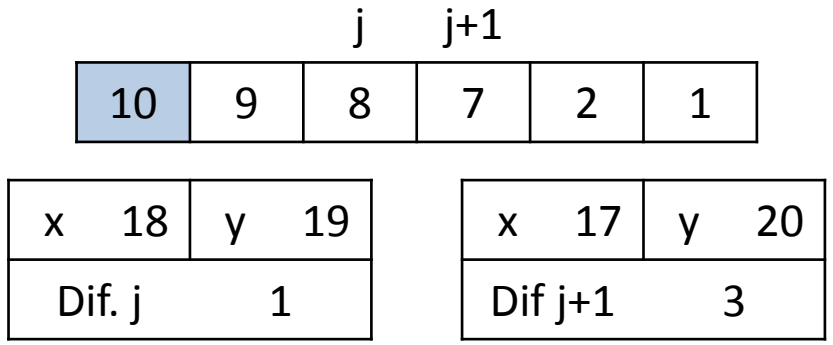
Tenemos que $S_1 = \{10, 9, 4\}$ y $S_2 = \{7, 6, 5, 3, 2, 1\}$, sus sumas son $x = 23$ e $y = 24$, por lo que su diferencia es 1.

Vamos a realizar a continuación la comprobación con el conjunto representado por el array $[1,2,7,8,9,10]$.

10. Dado un conjunto de números naturales $S = \{s_1, \dots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$) tal que se minimiza

$$\left| \sum_{x \in S_1} x - \sum_{y \in S_2} y \right|$$



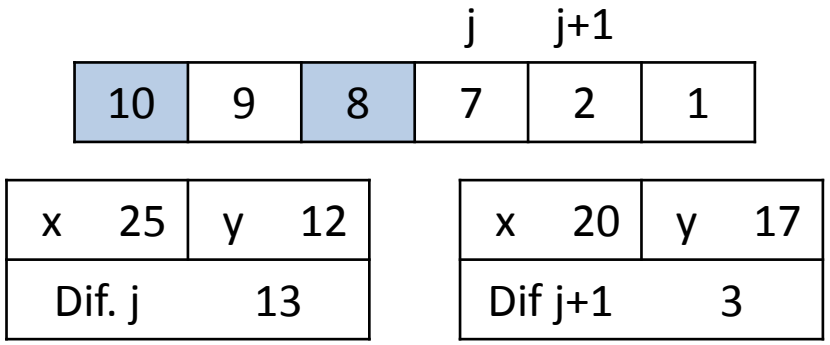


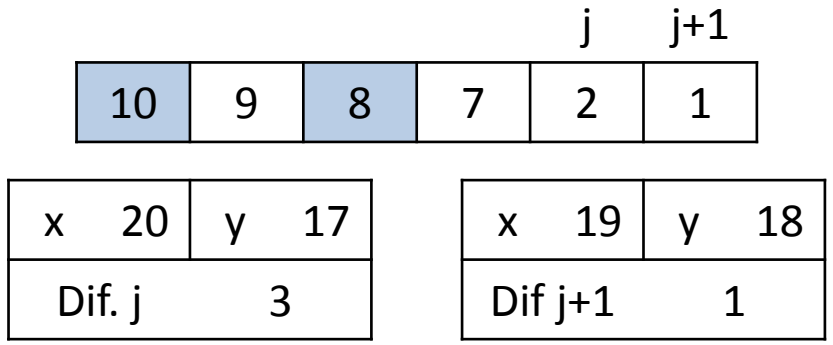
Actual	Restante	x	y	Dif.
[]	[10,9,8,7,2,1]	0	37	37
[10]	[9,8,7,2,1]	10	27	17

Actual	Restante	x	y	Dif.
[]	[10,9,8,7,2,1]	0	37	37
[10]	[9,8,7,2,1]	10	27	17
[10,8]	[9,7,2,1]	18	19	1

10. Dado un conjunto de números naturales $S = \{s_1, \dots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S$, $S_1 \cap S_2 = \emptyset$) tal que se minimiza

$$\left| \sum_{x \in S_1} x - \sum_{y \in S_2} y \right|$$



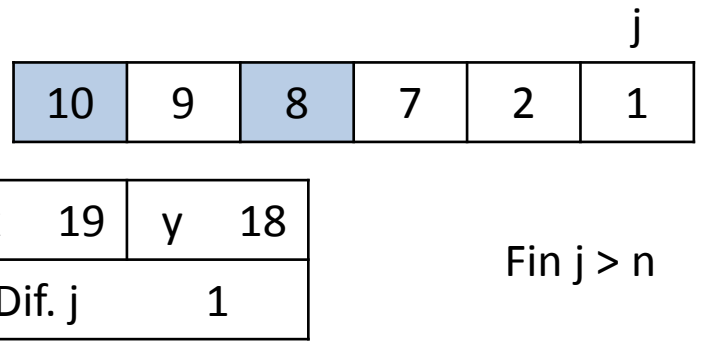


Actual	Restante	x	y	Dif.
[]	[10,9,8,7,2,1]	0	37	37
[10]	[9,8,7,2,1]	10	27	17
[10,8]	[9,7,2,1]	18	19	1

Actual	Restante	x	y	Dif.
[]	[10,9,8,7,2,1]	0	37	37
[10]	[9,8,7,2,1]	10	27	17
[10,8]	[9,7,2,1]	18	19	1

10. Dado un conjunto de números naturales $S = \{s_1, \dots, s_n\}$ se desea encontrar una partición en dos subconjuntos disjuntos S_1 y S_2 (i.e., $S_1 \cup S_2 = S, S_1 \cap S_2 = \emptyset$) tal que se minimiza

$$\left|\sum_{x \in S_1} x - \sum_{y \in S_2} y\right|$$



Actual	Restante	x	y	Dif.
[]	[10,9,8,7,2,1]	0	37	37
[10]	[9,8,7,2,1]	10	27	17
[10,8]	[9,7,2,1]	18	19	1
FIN Óptimo encontrado				

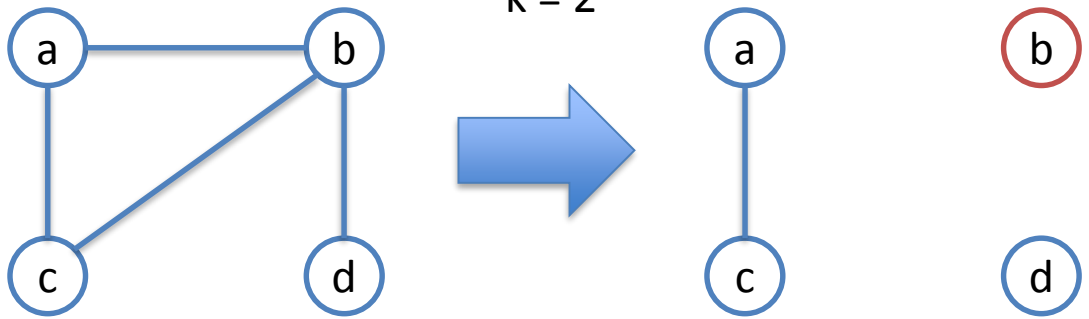
```
Set<Integer> subconjuntos_min (int[] set, int n) {

    Set<Integer> s1 = new HashSet<Integer>();
    int x = 0, y = 0, diff_min, diff_j, diff_j1;

    for (Integer s : set) { y += s; }
    s1.add(set[0]); x += set[0]; y -= set[0];
    diff_min = Math.abs(x-y);

    for (int j = 1; j < n-1; j++) {
        diff_j = Math.abs((x + set[j]) – (y - set[j]));
        diff_j1 = Math.abs((x + set[j+1]) – (y - set[j+1]));

        if (diff_j < diff_j1 && diff_j < diff_min) {
            s1.add(set[j]); x += set[j]; y -= set[j];
            diff_min = diff_j;
        }
    }
    diff_j = Math.abs((x + set[n-1]) – (y - set[n-1]));
    if (diff_j < diff_min) { s1.add(set[n-1]); }

    return s1;
}
```

13. Una misteriosa enfermedad de origen desconocido se está propagando por una indómita región de Gondwana. Cada pueblo de esta región puede modelarse como un grafo con n vértices cada uno de los cuales representa un habitante de la población, y en el que las aristas representan contacto físico entre individuos. La enfermedad considerada se caracteriza por un patrón de contagio múltiple: sólo si un individuo está conectado con al menos k individuos infectados, pasa a estar infectado él también.

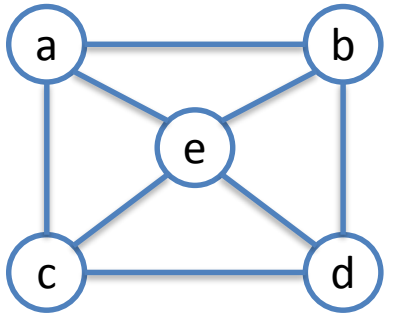
    Se ha podido desarrollar una vacuna para esta enfermedad, pero es muy costosa por lo que se quiere optimizar su distribución entre la población. El objetivo es, dado un grafo que representa las interacciones entre los individuos de un pueblo, determinar qué individuos deben ser inmunizados de manera que en el caso de que hubiera un brote infeccioso no se produjera contagio.

Aquí la intuición nos dice que si vacunamos en primer lugar aquella persona que más contactos tiene, podemos proteger a más personas. Así, es más probable que si tiene pocos contactos esté conectada con alguien con muchos contactos.
Con esto, si vacunamos a i, debemos actualizar el resto de individuos eliminando la vía de contagio con el individuo i, y si una persona ya tiene menos de k contactos podemos eliminarla de los individuos a vacunar.
Repetimos el proceso hasta que no queden personas que tienen más de k contactos.

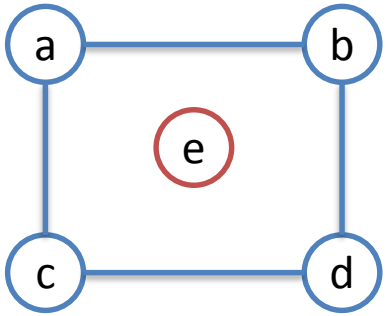
Así, tenemos que recorrer todos las aristas para sumar las aristas de cada vértice (complejidad $\Theta(|E|)$), después ordenar todos los vértices según sus aristas ($\Theta(|V| \log |V|)$), siguiente escoger el vértice i con mayor nº de aristas y suprimir todas las aristas que tienen contacto con el vértice i ($\Theta(|E|)$), todos los vértices que no tienen un mínimo k, se pueden suprimir, y se repite el proceso. Una vez se termina esto, se reordenan los vértices, y repetir el proceso hasta que no tengamos más vértices. Todo esto se hace $|V|$ veces (n), con lo cual su complejidad final es $\Theta\big(n * (|E| + n \log n)\big)$.

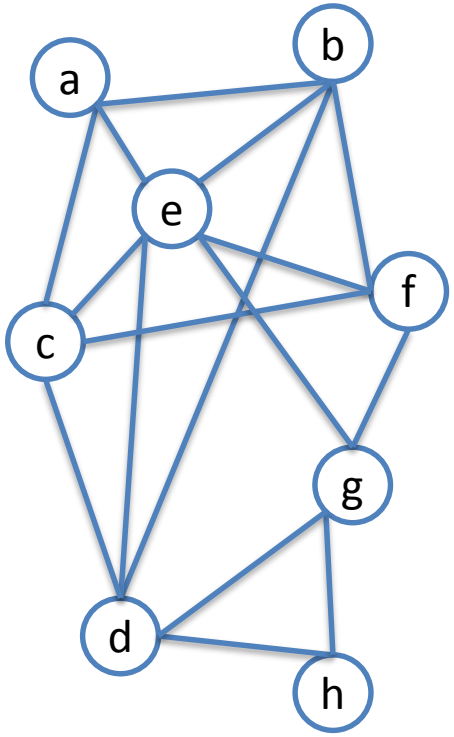


k = 2

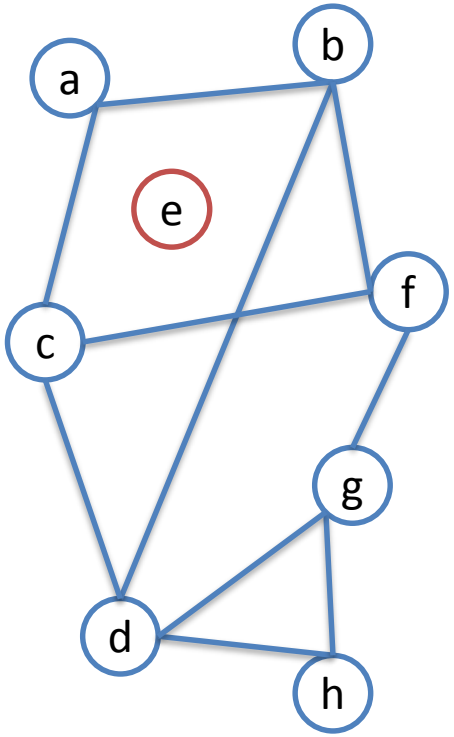


k = 3

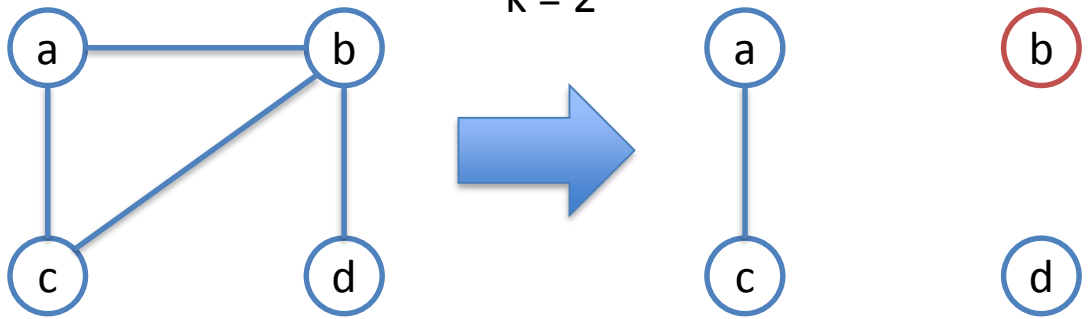




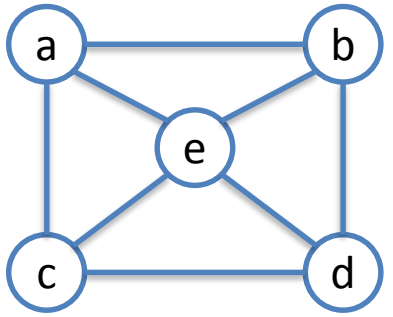
k = 2



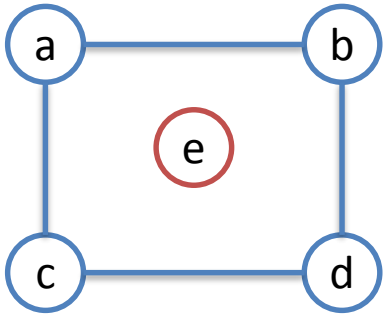
Vac	V	E
[]	[a, b, c, d, e, f, g, h]	[3, 4, 4, 5, 6, 4, 4, 2]
Ordenamos según |E|		
[]	[e, d, b, c, f, g, a, h]	[6, 5, 4, 4, 4, 4, 3, 2]
Vacunamos e (max |E|)		
[e]	[d, b, c, f, g, a, h]	[5, 4, 4, 4, 4, 3, 2]
Quitamos aristas a e		
[e]	[d, b, c, f, g, a, h]	[4, 3, 3, 3, 3, 2, 2]
Eliminamos vértices si |ei| < k (también aristas)		
[e]	[d, b, c, f, g, a, h]	[4, 3, 3, 3, 3, 2, 2]

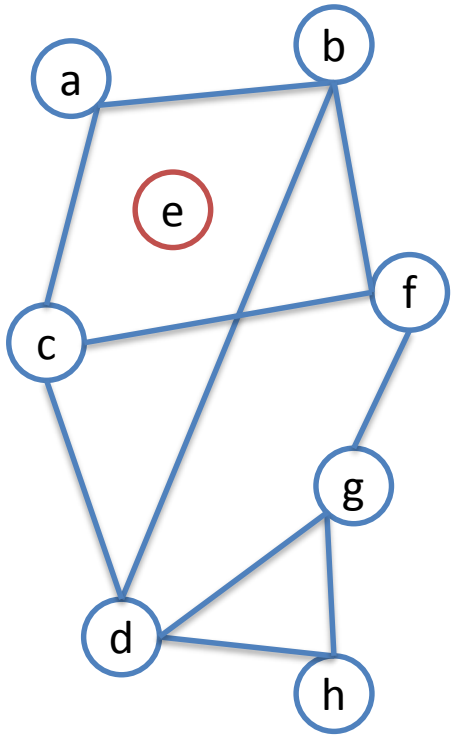


k = 2

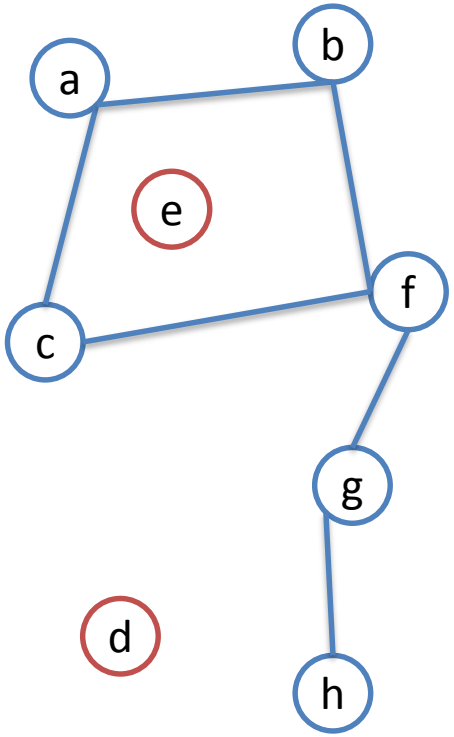


k = 3

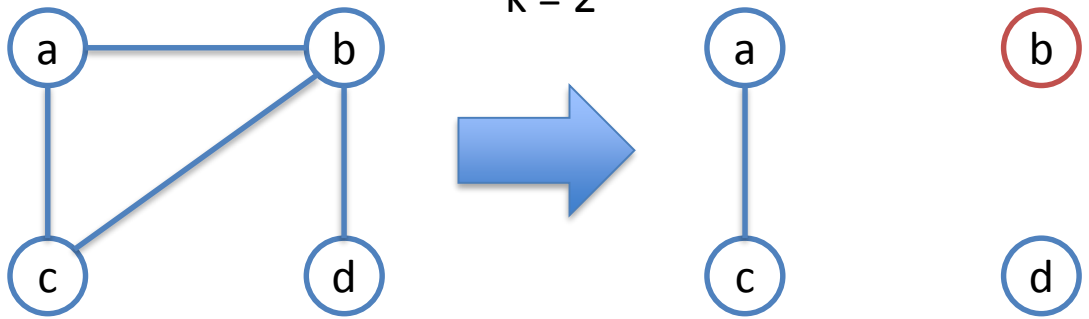




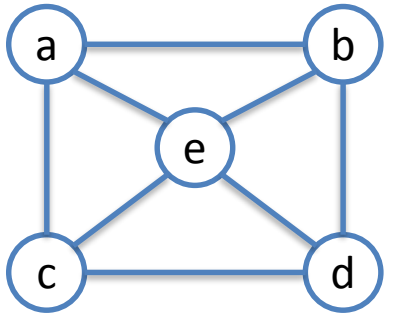
k = 2



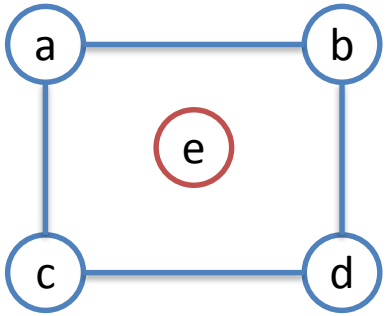
Vac	V	E
[e]	[d, b, c, f, g, a, h]	[4, 3, 3, 3, 3, 2, 2]
Ordenamos según |E|		
[e]	[d, b, c, f, g, a, h]	[4, 3, 3, 3, 3, 2, 2]
Vacunamos e (max |E|)		
[e, d]	[b, c, f, g, a, h]	[3, 3, 3, 3, 2, 2]
Quitamos aristas a d		
[e, d]	[b, c, f, g, a, h]	[2, 2, 3, 2, 2, 1]
Eliminamos vértices si |ei| < k (también aristas)		
[e, d]	[b, c, f, g, a]	[2, 2, 3, 1, 2]



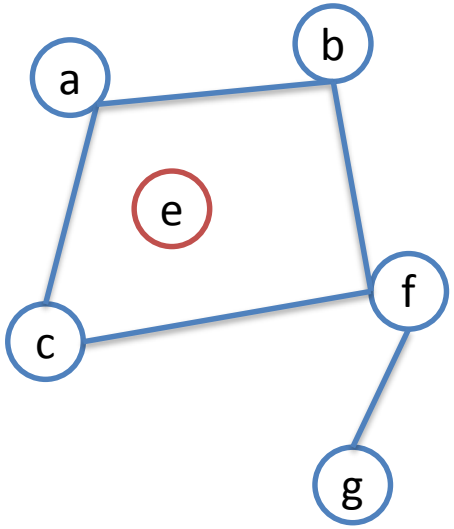
k = 2

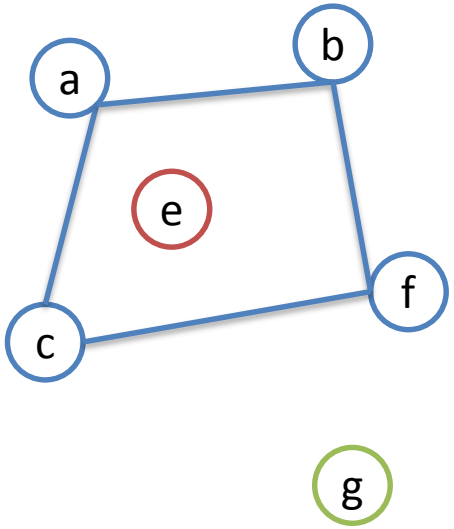


k = 3

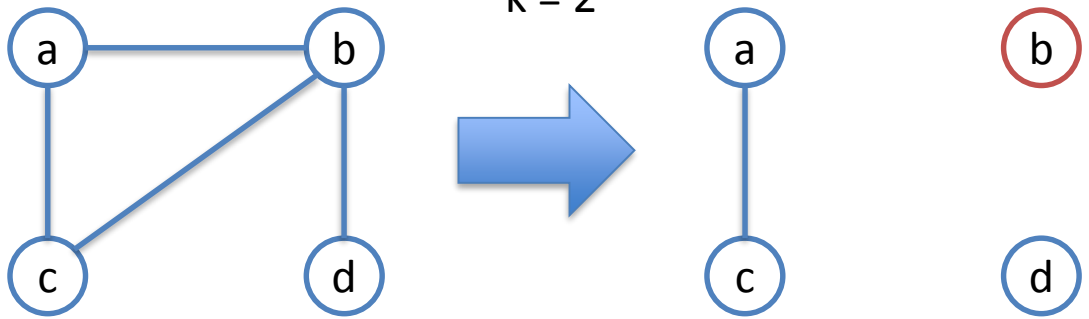


k = 2

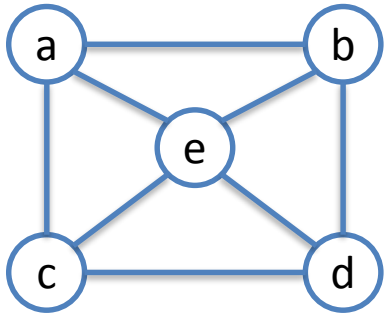




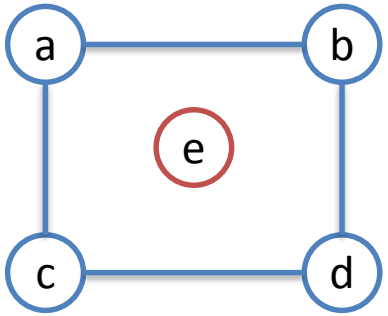
Vac	V	E
[e, d]	[b, c, f, g, a, h]	[2, 2, 3, 2, 2, 1]
Eliminamos vértices si |ei| < k (también aristas)		
[e, d]	[b, c, f, g, a]	[2, 2, 3, 1, 2]
Eliminamos vértices si |ei| < k (también aristas)		
[e, d]	[b, c, f, a]	[2, 2, 2, 2]
Ordenamos según |E|		
[e, d]	[b, c, f, a]	[2, 2, 2, 2]
Vacunamos e (max |E|)		
…	…	…



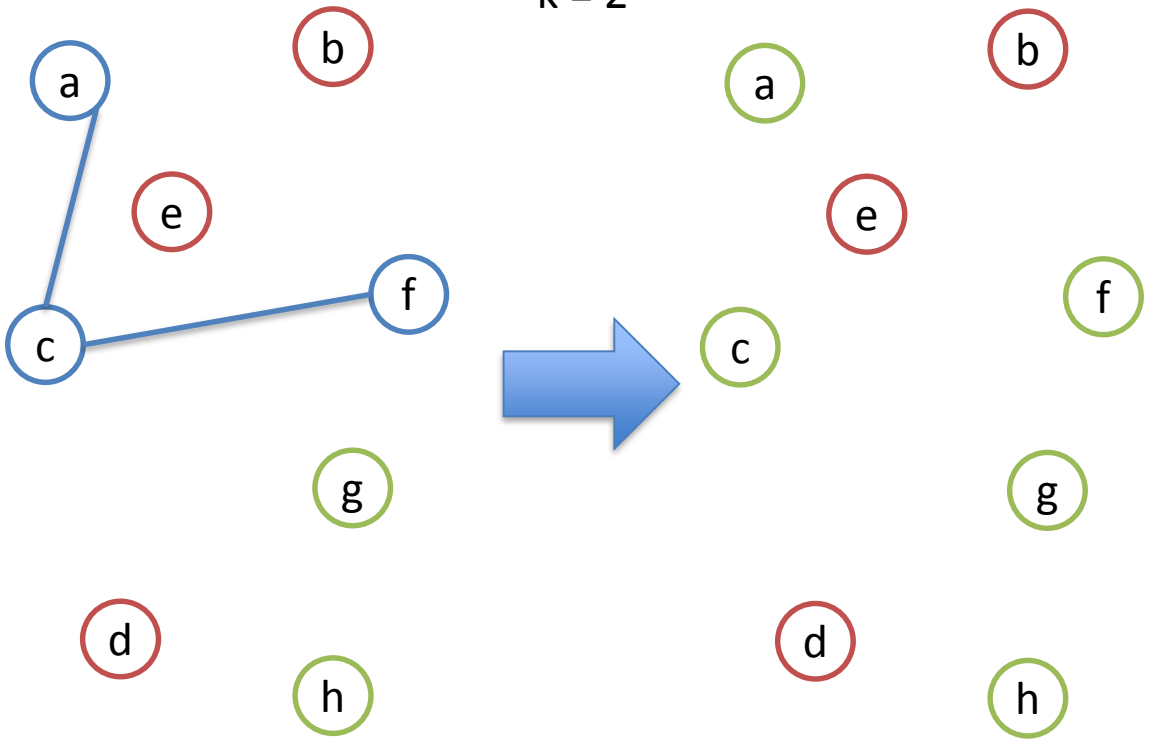
k = 2



k = 3



k = 2



Vac	V	E
[e, d, b]	[]	[]
FIN ya no quedan vértices		

El hecho de que eliminemos aquellos individuos que no se pueden infectar por falta de contacto con gente que pueda infectarse, hace que podamos encontrar un óptimo con mayor facilidad.

Ejemplo, h no puede infectarse porque solo tiene contacto con d (vac) y g. Al quedar menos de 2, nunca podrá contagiarse. Y al no poder contagiarse él, es como si estuviera vac, y nunca podrá contagiar a g, que a su vez simula estar vac, y nunca contagiará a f.

```
Set<V> vacunacion (ArraySet<V> vertices, ArraySet<E> aristas, int k) {

      Set<V> vac = new Set<V>();
       vertices, aristas, num_aristas = comprobar_aristas (vertices, aristas, k);

       while (vertices != Ø) {
              vertices, num_aristas = ordenar (vertices, num_aristas)
              sig_v = vertices.get_next(); //Devuelve y elimina

              vac.add(sig_v);
              aristas = eliminar_vertice (sig_v, aristas);
              vertices, aristas, num_aristas = comprobar_aristas (vertices, aristas, k);
       }

       return vac;
}
```

```
comprobar_aristas (ArraySet<V> vertices, ArraySet<E> aristas, int k) {

       int[] num_aristas = contar_aristas (vertices, aristas);

       while (vertices != Ø && ∃i ∈ num_aristas < k) {
              sig_v = vertices.get (i); //Devuelve y elimina
              aristas = eliminar_vertice (sig_v, aristas);
              num_aristas = contar_aristas (vertices, aristas);
       }

       return vertices, aristas, num_aristas;
}
```

Si ordenamos los vértices óptimos según las aristas desde s hacia los otros n-1 nodos, el i-ésimo vértice a vacunar se encuentra en la iteración i. Puede demostrarse por inducción:

- En la primera iteración se encuentra el vértice con más aristas, lo que hace que al escogerlo, todos los demás vértices vean reducido el número de contactos en al menos 1.
- Suponiendo que en la iteración i-1 se han encontrado los i-1 vértices con mayor número de aristas que salen de ella, el i-ésimo vértice v es el que mayor número de aristas tiene al haber eliminado s.
- Cualquier otro vértice j tiene menos aristas ($|v| > |j|$), por lo que su impacto en el resto de vértices es menor que el impacto de v.

1. En un contexto de rápida inflación debemos adquirir n ítems cuyo precio inicial $C > 0$ es para todos el mismo; sin embargo, sólo podemos comprar un ítem cada semana y el precio del i-ésimo ítem se incrementa en progresión geométrica con tasa $T_i > 1$. Así, si compramos el ítem i en la semana t, siendo $0 \leq t \leq n-1$, su precio será $C * T_i^t$. Se pide:

   a) Diseñar un algoritmo voraz que, dadas las tasas $T_0, \ldots, T_{n-1}$ y el valor C, determine en qué orden deben hacerse las compras y qué valor total tendrán las n compras de tal modo que dicho valor sea mínimo; en concreto, para cada semana t, el algoritmo debe devolver un vector item tal que item[t] es el índice del ítem a comprar en la semana t.

   b) Demostrar su corrección.

Por ejemplo, si $n = 3$, $C = 100$ y $T = [3;\ 2;\ 4]$, y compramos los ítems en el orden 0, 1, 2 entonces el valor total de las compras es $100 * 3^0 + 100 * 2^1 + 100 * 4^2 = 100 * (1 + 2 + 16) = 1900$ euros.

Nos tenemos que dar cuenta de que el orden de las compras es relevante.
Si en vez de $T = [3;\ 2;\ 4]$, compramos con $T = [4; 3; 2]$, comprobamos lo siguiente:

$$100 * 4^0 + 100 * 3^1 + 100 * 2^2 = 100 * (1 + 3 + 4) = 800 \text{ euros}$$

a) Diseñar un algoritmo voraz que, dadas las tasas $T_0, \ldots, T_{n-1}$ y el valor C, determine en qué orden deben hacerse las compras y qué valor total tendrán las n compras de tal modo que dicho valor sea mínimo; en concreto, para cada semana t, el algoritmo debe devolver un vector item tal que item[t] es el índice del ítem a comprar en la semana t.

Tras una breve experimentación llegamos a la conclusión de que los ítems deben comprarse por orden decreciente de su tasa. De modo que el algoritmo voraz consiste sencillamente en ordenar los ítems y obtener mediante un simple cálculo el valor.

La ordenación toma tiempo $\Theta(n \log n)$; el cálculo del valor de las compras puede llevarnos tiempo $O(n2)$ salvo que utilicemos un algoritmo astuto para calcular las potencias de las i's. Podemos conseguir que dicho coste sea también $\Theta(n \log n)$.

```
double orden_compras (double[] tasa , double C,
                                    int[] item , double val) {

    for (int t = 0; t < tasa.size (); t++)
        item [t] = t;

    ordenar_por_tasa_decr (item , tasa);
    // tasa [item [t]] >= tasa [ item [ t+1 ]]

    val = 0.0;
    for (int t = 0; t < item.size(); t++)
        val += pot(tasa [item [t]], t);
    return C * val;
}
```

b) Demostrar su corrección.

Consideremos una solución optima OPT (su valor V_{opt} es mínimo) y supongamos que en dicha solución los ítems no se compran por orden decreciente de tasa de inflación. Entonces, habrá dos semanas consecutivas i, $i+1$ tales que las tasas T_x y T_y están invertidas: $T_x < T_y$.

En este caso, $V_{opt} = K + C \cdot T_x^i + C \cdot T_y^{i+1}$
Si intercambiamos el orden de compra tenemos $V = K + C \cdot T_y^i + C \cdot T_x^{i+1}$

Donde K, representa la suma total restante, que no se ve modificada.
Calculamos la diferencia entre ambos valores, teniendo en cuenta que K se elimina (constante) y resulta:

$$V_{opt} - V = C \cdot T_x^i + C \cdot T_y^{i+1} - C \cdot T_y^i - C \cdot T_x^{i+1} = C \cdot T_y^i \cdot (T_y - 1) + C \cdot T_x^i \cdot (T_x - 1)$$

Teniendo en cuenta que todas las tasas son $C > 1$ y que $T_x < T_y$ resulta:

$$T_y^i \cdot (T_y - 1) > T_x^i \cdot (T_y - 1) > T_x^i \cdot (T_x - 1)$$

Luego $V_{opt} - V > 0$, es decir, $V_{opt} > V$. Luego la solución voraz V es mejor al ordenar las tasas de forma decreciente, y también es óptima, como se quería demostrar.