Problema del cambio de moneda, solución recursiva

El problema del Cambio de Monedas. "*Para una un juego de monedas arbitrario de U_1, U_2,...U_n (unidades). Una de las monedas es la unitaria. ¿Cuál es el* **mínimo** *número de monedas que se necesitan para devolver K unidades de cambio*?"

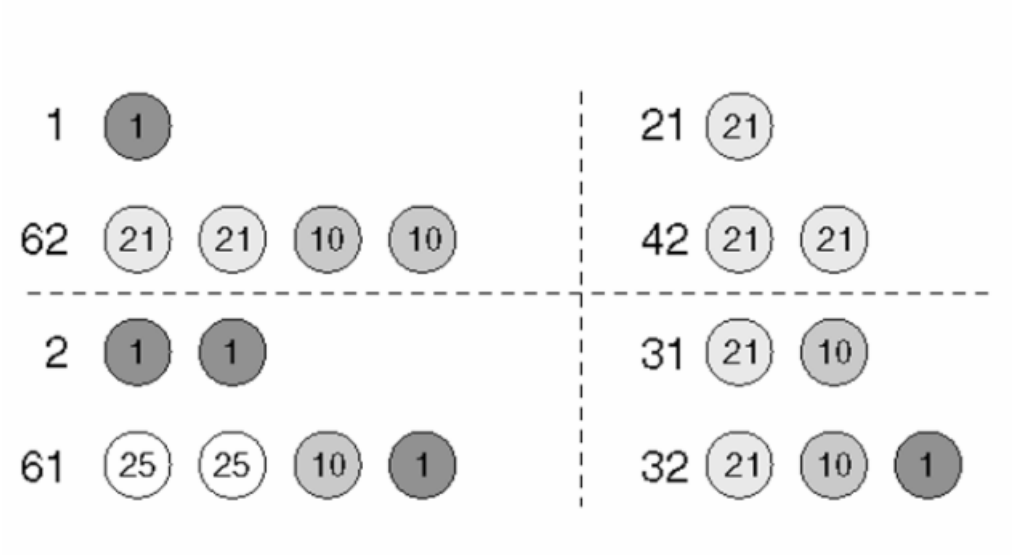
Supongamos el juego de monedas (1,5,10,25), con el que queremos el cambio de 63.

Un algoritmo directo (voraz) nos daría 6 monedas:
2 de 25,1 de 10, y 3 de 1

Si el juego ahora es (1,5,10, 21, 25), el mismo algoritmo nos da 6 monedas. En cambio la solución es 3 monedas de 21. (**La solución no seria correcta**).

El planteamiento recursivo seria:

Una solución al problema se consigue utilizando un algoritmo recursivo que explore el espacio de soluciones posibles evaluando en cada caso el número de monedas necesarias. Obteniendo finalmente la combinación de monedas mínima. El planteamiento sería, para cada posible valor i calcular de forma independiente el número mínimo de monedas que se necesitan para reunir i y K-i unidades. Eligiendo el i que minimice la suma de ambos subproblemas. **Esta solución no cumple la cuarta regla de la recursión y por tanto es muy ineficiente**.



```java
public int cambioMonedaRec(int[] monedas, int cambio, int monedasDistintas) {
    int minMonedas = cambio;
    for (int i = 0; i < monedasDistintas; i++) {
        if (monedas[i] == cambio)
            return 1;
    }
    for (int j = 1; j <= cambio / 2; j++) {
        int monedasActuales = cambioMonedaRec(monedas, j, monedasDistintas)
                + cambioMonedaRec(monedas, cambio - j, monedasDistintas);
        if (monedasActuales < minMonedas)
            minMonedas = monedasActuales;
    }
    return minMonedas;
}
```

==========

```java
public static void main(String[] args) {

    int[] monedas = { 1, 5, 8, 10, 25 };
    int monedasDistintas = monedas.length;
    int acambiar = 16;

    Cambio devolucion = new Cambio();
    int cambio;

    cambio = devolucion.cambioMonedaRec(monedas, acambiar, monedasDistintas);
    System.out.println("Mínimo de monedas usando el metodo recursivo :" + cambio);
```

```
Mínimo de monedas usando el metodo recursivo :2
Mínimo de monedas usando el metodo voraz :3
```

Subsecuencia máxima, resuelta con técnicas divide y vencerás.

Subsecuencia	{4,-3,5,-2,-1,2,6,-2}	
Primera mitad	Segunda mitad	
4, -3, 5, -2	-1, 2, 6, -2	Valores
4*, 0, 3, -2	-1, 1, 7*, 5	Sumas
* significa el máximo de cada mitad		

- ✓Caso 1 : está totalmente incluida en la **primera** mitad
- ✓Caso 2 : está totalmente incluida en la **segunda** mitad
- ✓Caso 3 : **comienza en la primera mitad y acaba en la segunda**

El caso 3, la suma de las dos subsecuencias,

4+7= 11

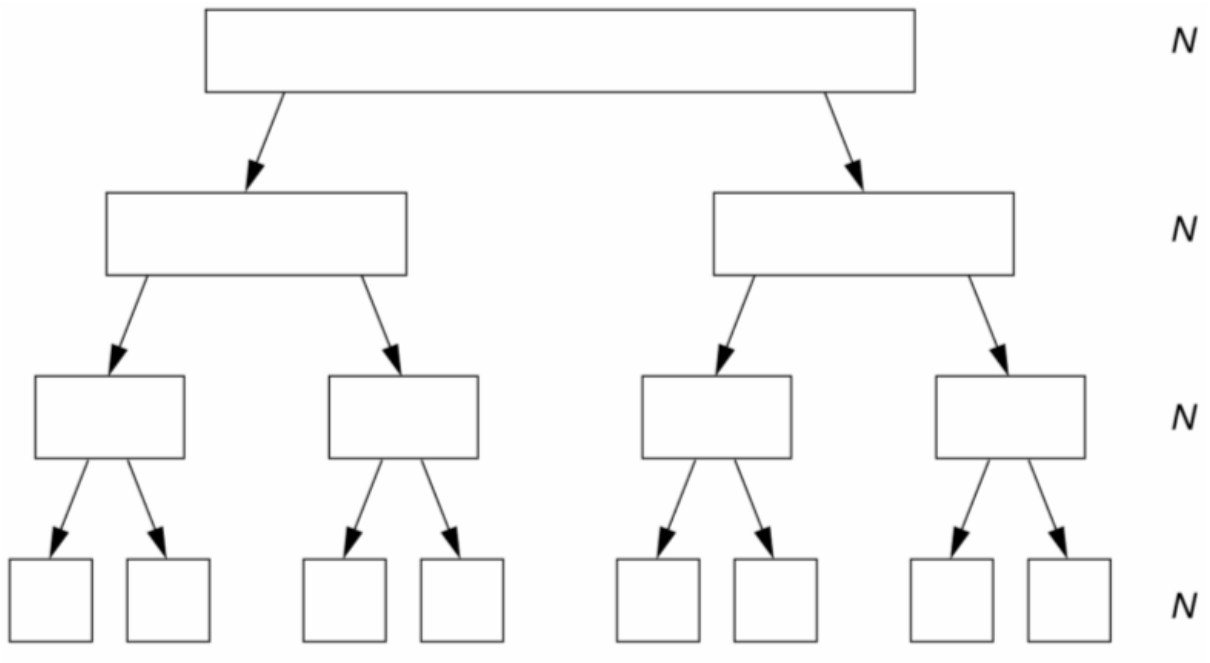
Esquema del algoritmo:

1. Calcular recursivamente la subsecuencia de suma máxima que está en la **primera** mitad.
2. Calcular recursivamente la subsecuencia de suma máxima que está en la **segunda** mitad.
3. Calcular, usando dos blucles consesecutivos, la subsecuencia de suma máxima que **comienza en la primera mitad pero termina en la segunda**.
4. Elegir **la mayor de las tres sumas**.

```
private int subsecuenciaRec(int izq, int der) {

    if (izq == der) {
        if (subsecuencia.datosModelo[izq] > 0)
            return subsecuencia.datosModelo[izq];
        else
            return 0;

    } else {
        int centro = (izq + der) / 2;

        int sumaMaxIzq = subsecuenciaRec(izq, centro);
        int sumaMaxDer = subsecuenciaRec(centro + 1, der);
        int sumaMaxIzqBorde = 0;
        int sumaBordeIzq = 0;
        for (int i = centro; i >= izq; i--) {
            sumaBordeIzq = sumaBordeIzq + subsecuencia.datosModelo[i];
            if (sumaBordeIzq > sumaMaxIzqBorde) {
                sumaMaxIzqBorde = sumaBordeIzq;
                secIni = i;
            }
        }
        int sumaMaxDerBorde = 0;
        int sumaBordeDer = 0;
        for (int i = centro + 1; i <= der; i++) {
            sumaBordeDer = sumaBordeDer + subsecuencia.datosModelo[i];
            if (sumaBordeDer > sumaMaxDerBorde) {
                sumaMaxDerBorde = sumaBordeDer;
                secFin = i;
            }
        }
        int suma_bordes = sumaMaxIzqBorde + sumaMaxDerBorde;
        int suma_sub_max = 0;

        if (sumaMaxIzq > sumaMaxDer) {
            if (sumaMaxIzq > suma_bordes) {
                suma_sub_max = sumaMaxIzq;
                secIni = izq;
                secFin = centro;
            } else {
                suma_sub_max = suma_bordes;

            }
        } else {
            if (sumaMaxDer > suma_bordes) {
                suma_sub_max = sumaMaxDer;
                secIni = centro;
                secFin = der;

            } else {
                suma_sub_max = suma_bordes;

            }
        }
        return suma_sub_max;
    }
}
```

Análisis del algoritmo divide y vencerás

Traza de las llamadas recursivas del algoritmo de la subsecuencia de suma máxima para N=8



En un **análisis intuitivo** observamos que el problema se va partiendo en problemas la mitad más pequeños.

Es decir, se pasa de N a N/2. Así para un problema de tamaño N tendríamos $1 + \lceil \log N \rceil$ niveles . Donde $\lceil X \rceil$ es una función techo que representa el entero más pequeño mayor o igual que X. Así para el caso de N=8, tendríamos 4 niveles.

Como en cada llamada recursiva hay dos blucles con un recorrido en N/2. Significa que hay una **sobrecarga** O(N).

Por lo tanto el tiempo total de ejecución seria O(N+ N *log*N) que es **O(N *log*N).**

Un análisis más sistemático

Sea T(N) el tiempo de resolver el problema.

Del análisis del código obtenemos las ecuaciones:

$$T(1) = 1$$
$$T(N) = 2T(N/2) + O(N)$$

Simplificamos suponiendo N es una potencia de 2, y reemplazamos O(N) por N.

$$T(1) = 1$$
$$T(N) = 2T(N/2) + N$$

Demostramos ahora que la solución a esta ecuación, es $T(N) = N\ log N + N$

Haciendo sustituciones reiteradas, tenemos:

$$T(N/2) = 2T(N/4) + N/2$$

De donde:

$$T(N) = 4T(N/4) + 2N$$

y

$$4T(N/4) = 8T(N/8) + N$$

De donde:

$$T(N) = 8T(N/8) + 3N$$

La formula general seria

$$T(N) = 2^kT(N/2^k) + kN$$

Finalmente, tomando $k = log N$, cierto porque hemos supuesto que $2^k = N$, obtenemos:

$$T(N) = NT(1) + N\ log N = N + N\ log N$$

En notación O grande, el algoritmo seria de orden O(NlogN)

Una cota superior general para los tiempos de ejecución de los algoritmos divide y vencerás.

- ¿Qué pasa si dividimos el problema en tres partes iguales?, ¿o en siete con un coste adicional cuadrático?

Veamos la formula general, que requiere de tres parámetros.

- A, que es el número de subproblemas.
- B, es el factor de reducción de tamaño relativo de los subproblemas (por ejemplo B = 2 representa que los subproblemas tienen la mitad de tamaño que el problema original)
- k, representa el hecho de que el coste adicional es $\Theta(N^k)$.

Teorema

La solución de la ecuación ***T(N) = AT(N/B) + O(N^k),*** donde A >= 1 y B > 1, es:

$$T(N) = \begin{cases} O(N^{\log_B A}) & A > B^k \\ O(N^k \log N) & A = B^k \\ O(N^k) & A < B^k \end{cases}$$

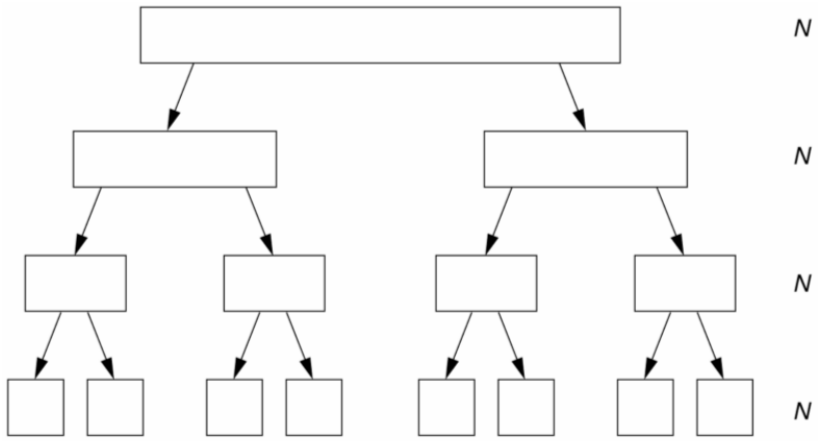
Ejemplos:

Subsecuencia. A=2, B=2, k=1 tenemos O(N log N)

La búsqueda binaria recursiva. A=1, B=2, k=0, tenemos O(logN)

Búsqueda binaria recursiva.

Se resuelve dividiendo el problema en problemas menores.
O lo que es lo mismo, utilizando al técnica recursiva **divide y vencerás**.



Según el esquema visto:

La búsqueda binaria recursiva. A=1, B=2, k=0, tenemos O(logN)

El tiempo de ejecución, en términos de notación O, es el mismo que en la implementación iterativa. O(logN).

En la practica, el tiempo de ejecución es ligeramente superior debido a los costes ocultos de la recursión.

```
private void busquedaBinariaRec( int buscado,int inicio, int fin){

    try {
        if( inicio > fin )
            throw new ElementoNoEncontrado( "Busqueda falla" );

        int medio = ( inicio + fin ) / 2;

        if( busqueda.datos[medio] < buscado )
            busquedaBinariaRec( buscado, medio + 1, fin );
        else if( busqueda.datos[medio] > buscado )
            busquedaBinariaRec( buscado, inicio, medio - 1 );

    } catch (ElementoNoEncontrado e) {
        // TODO Auto-generated catch block
           e.printStackTrace();
    }
}
```

La ordenación de un vector de N elementos

> *Dado un vector A[1..n] de n elementos. Ordenar estos elementos por orden ascendente.*

Nota: Sólo pueden ser ordenados objetos que implementen la interfaz ***Comparable***. Por lo tanto las únicas operaciones disponibles para obtener información sobre el orden relativo de los elementos son ***menorQue*** y ***compara***.

Ordenación por inserción

Es el método de ordenación más simple. Su implementación utiliza dos bucles cada uno de los cuales puede realizar N iteraciones. Por lo tanto el algoritmo de ordenación por inserción es **$O(N^2)$,** es decir cuadrático.

- El peor de los casos se da si el vector viene ordenado en orden inverso.
- El mejor de los casos se da si el vector ya viene ordenado, entonces seria de orden lineal

```
public static void ordenacionPorInsercion(Comparable[] a) {
    for (int p = 1; p < a.length; p++) {
        Comparable temp = a[p];
        int j = p;
        for (; j > 0 && temp.menorQue(a[j - 1]); j--)
            a[j] = a[j - 1];
            a[j] = temp;

    }
}
```

Funcionamiento del algoritmo de ordenación por inserción

Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After a[0..1] is sorted	5	8	9	2	6	3
After a[0..2] is sorted	5	8	9	2	6	3
After a[0..3] is sorted	2	5	8	9	6	3
After a[0..4] is sorted	2	5	6	8	9	3
After a[0..5] is sorted	2	3	5	6	8	9

Array Position	0	1	2	3	4	5
Initial State	8	**5**				
After a[0..1] is sorted	5	8	**9**			
After a[0..2] is sorted	5	8	9	**2**		
After a[0..3] is sorted	2	5	8	9	**6**	
After a[0..4] is sorted	2	5	6	8	9	**3**
After a[0..5] is sorted	2	3	5	6	8	9

- Las celdillas de fondo oscuro señalan el área ordenada.
- Las celdillas de fondo más claro marcan los intercambios

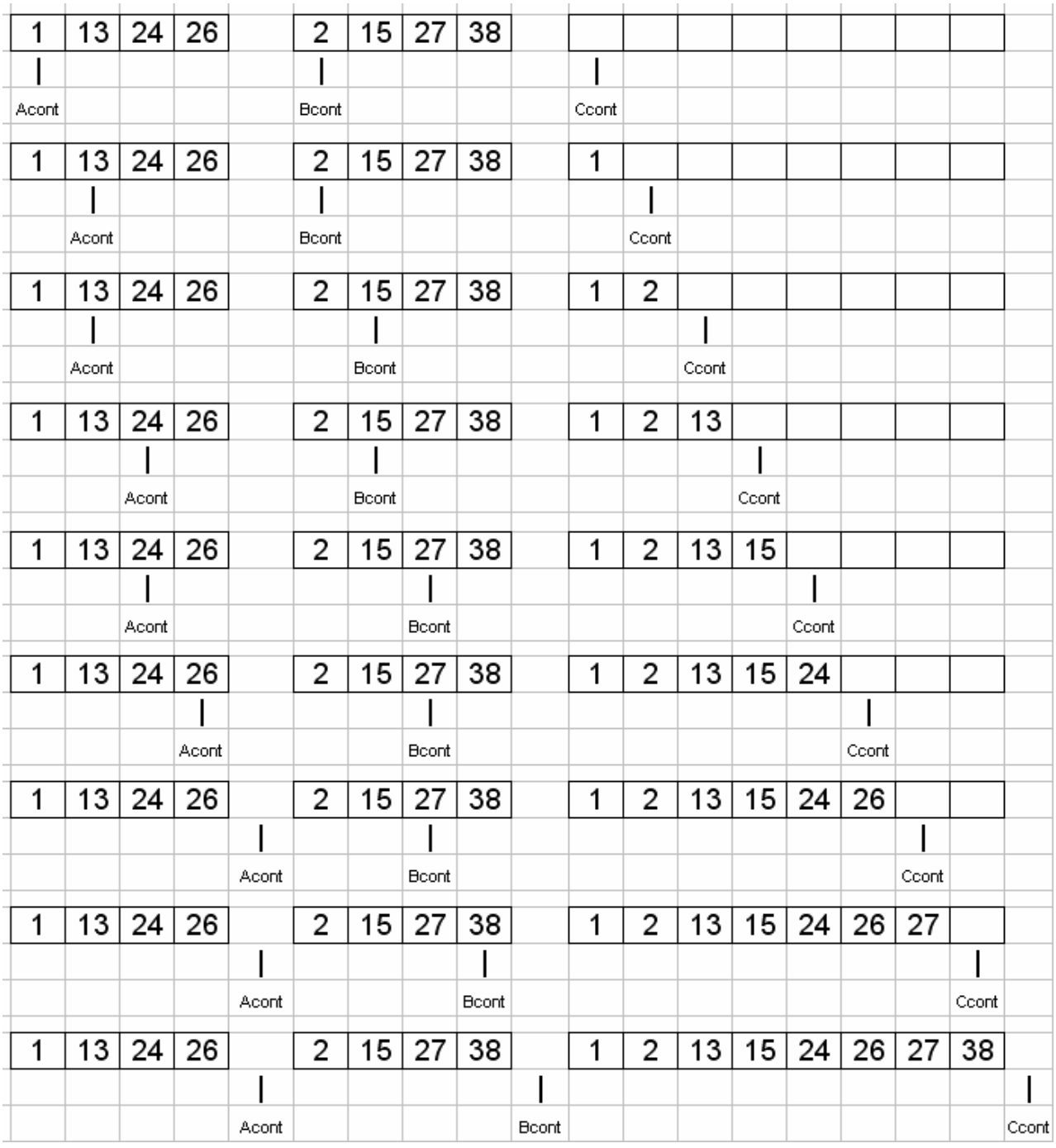
MergeSort

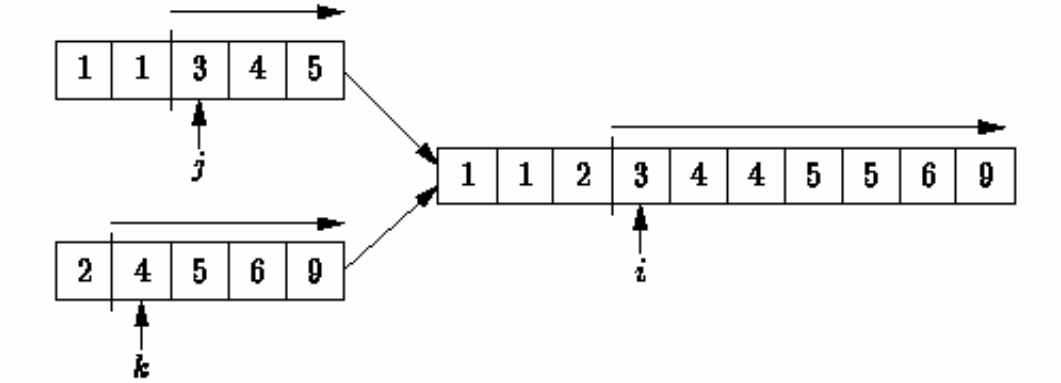
El algoritmo Mergesort utiliza la técnica recursiva divide y vencerás para obtener un tiempo de ejecución de O(NlogN).

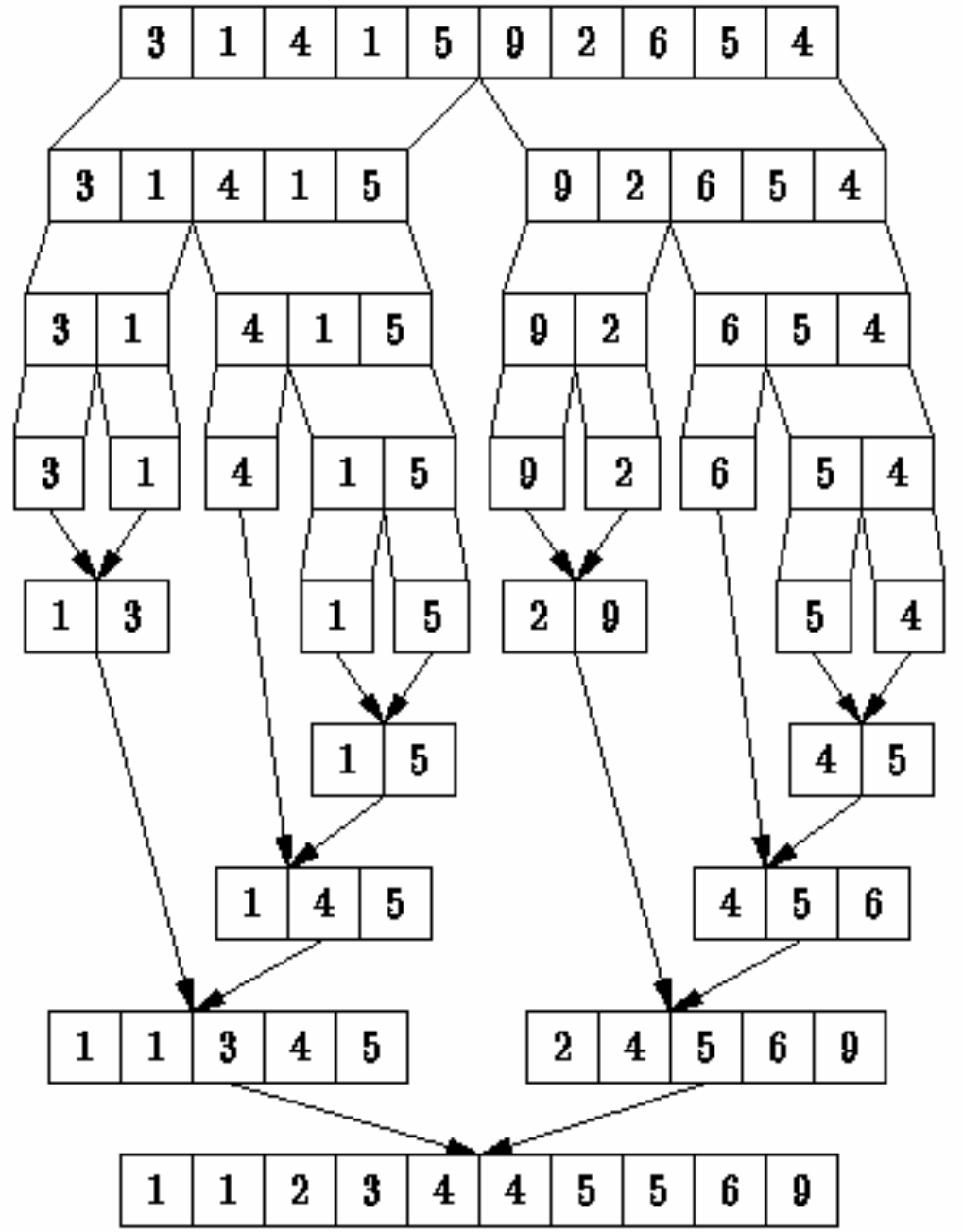
El algoritmo consta de tres pasos:

1. Si el número de elementos a ordenar es 0 o 1, acabar.
2. **Ordenar recursivamente** las dos mitades del vector.
3. **Mezclar** las dos mitades ordenadas en un vector ordenado.

La mezcla lineal de vectores ordenados:




1 1 3 4 5
j
2 4 5 6 9
k
1 1 2 3 4 4 5 5 6 9
i


3 1 4 1 5 9 2 6 5 4
3 1 4 1 5
9 2 6 5 4
3 1
4 1 5
9 2
6 5 4
3
1
4
1 5
9
2
6
5 4
1 3
1 5
2 9
5 4
1 5
4 5
1 4 5
4 5 6
1 1 3 4 5
2 4 5 6 9
1 1 2 3 4 4 5 5 6 9

El tiempo necesario para mezclar dos vectores ordenados es lineal pues casa comparación incrementa Ccont (limitando el número de comparaciones).

El algoritmo divide y vencerás que utiliza un procedimiento de mezcla lineal se ejecutará en el peor de los casos, en promedio e incluso en el caso mejor en tiempo O(NlogN) ya que la mezcla lineal siempre es lineal.

```
public static void mergeSort(Comparable[] a) {
    Comparable[] vectorTemp = new Entero[a.length];
    mergeSort(a, vectorTemp, 0, a.length - 1);
}

private static void mergeSort(Comparable[] a, Comparable[] vectorTemp,
        int izq, int der) {

    if (izq < der) {
        int centro = (izq + der) / 2;
        mergeSort(a, vectorTemp, izq, centro);
        mergeSort(a, vectorTemp, centro + 1, der);
        mezclar(a, vectorTemp, izq, centro + 1, der);

    }
}

private static void mezclar(Comparable[] a, Comparable[] vectorAux,
        int posIzq, int posDer, int posFin) {
    int finIzq = posDer - 1;
    int posAux = posIzq;
    int numElementos = posFin - posIzq + 1;
    while ((posIzq <= finIzq) && (posDer <= posFin)) {
        if (a[posIzq].menorQue(a[posDer]))
            vectorAux[posAux++] = a[posIzq++];
        else
            vectorAux[posAux++] = a[posDer++];
    }
    // Copiar el resto de la primera mitad
    while (posIzq <= finIzq)
        vectorAux[posAux++] = a[posIzq++];
    // Copiar el resto de la segunda mitad
    while (posDer <= posFin)
        vectorAux[posAux++] = a[posDer++];
    //Copiar el vector temporal en el original
    for (int i = 0; i < numElementos; i++, posFin--)
        a[posFin] = vectorAux[posFin];
}
```

a	1	1	2	3	4	5	6	7
	24	13	26	1	2	27	38	15
	|				|			|
	posIzq				posDer			posFin
	numElementos=posFin-posIz +1							
vectorTemp	1	1	2	3	4	5	6	7
	24	13	26	1	2	27	38	15
	|			|				
	posAux			finIzq				

Quicksort (autor C.A.R. Hoare)

El algoritmo Quicksort es un algoritmo divide y vencerás.
En la práctica es el algoritmo de ordenación más rápido, basado en comparaciones.

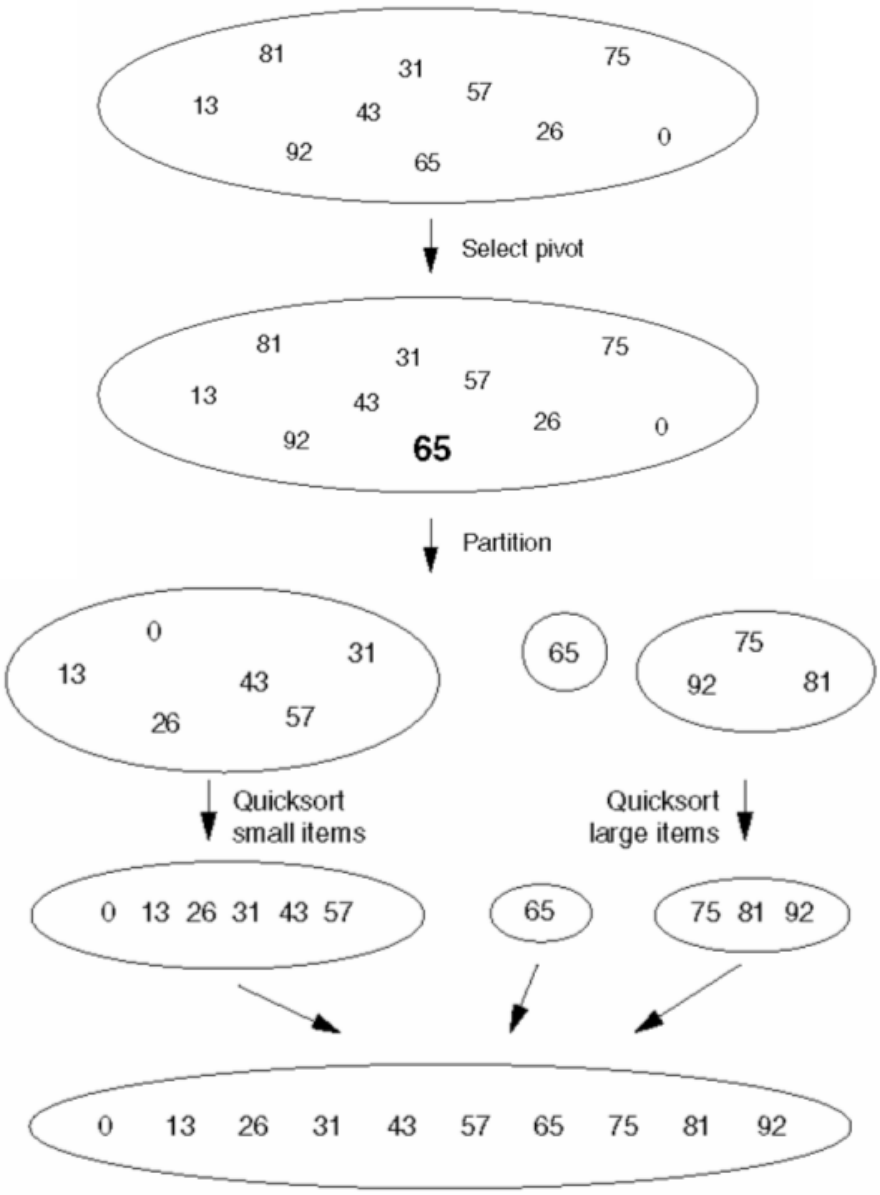
El algoritmo básico es *Quicksort (S)* es recursivo, y consta de los siguientes cuatro pasos:

1. Si el número de elementos de *S* es 0 o 1, terminar
2. Escoger un elemento cualquiera v de *S*. Este elemento se denomina pivote.
3. Hacer una partición *S-{v}* en dos subconjuntos disjuntos

$$I = \{x \in S - \{v\} \mid x \leq v\}$$

$$D = \{x \in S - \{v\} \mid x \geq v\}$$

4. Devolver el resultado de *Quicksort (I)*, seguido de v y seguido del resultado de devolver *Quicksort (D*).



Análisis de QuickSort

Caso Peor :

El caso peor ocurre cuando reiteradamente uno de los subconjuntos generados por la partición es el vacío. El tiempo de ejecución es cuadrático. O(N2).

Por ejemplo: el caso de un vector ya ordenado, eligiendo como pivote el primer elemento.

Suponiendo que el tiempo de ordenar 0 o 1 elemento es:

$$T(1) = 1,\ T(0) = 1$$

Y que hacer la partición de N elementos requiere N unidades de tiempo:

$$T(N) = T(N-1) + N$$

$$T(N-1) = T(N-2) + (N-1)$$

$$T(N-2) = T(N-3) + (N-2)$$

$$\cdots\cdots\cdots\cdots\cdots$$

$$T(2) = T(1) + 2$$

Sumando por columnas y cancelando términos, tenemos que:

$$T(N) = T(1) + 2 + 3 + \ldots + N = N(N+1)/2 = \mathbf{O(N^2)}$$

Caso Medio

El coste en el caso medio de una llamada recursiva se obtiene haciendo la media de los costes sobre todos los posibles tamaños de los subproblemas.

$$T(I) = T(D) = \frac{T(0) + T(1) + T(2) + ... + T(N-1)}{N}$$

$$T(N) = 2(\frac{T(0) + T(1) + T(2) + ... + T(N-1)}{N}) + N$$

$$NT(N) = 2(T(0) + T(1) + T(2) + ... + T(N-1)) + N^2$$

para N-1

$$(N-1)T(N-1) = 2(T(0) + T(1) + T(2) + ... + T(N-2)) + (N-1)^2$$

restando

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2N - 1$$

reorganizando y eliminado el -1, no significativo

$$NT(N) = (N+1)T(N-1) + 2N$$

Tenemos T(N) en funcion de T(N-1).
Desplegamos la ecuación, dividiendo por N(N+1)

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1}$$

$$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2}{N}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2}{N-1}$$

$$.....$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2}{3}$$

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + ... + \frac{1}{N} + \frac{1}{N+1}\right)$$

$$\frac{T(N)}{N+1} = 2\left(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{N} + \frac{1}{N+1}\right) - \frac{5}{2}$$

Donde el N-ésimo armónico es de O(logN)

$$\frac{T(N)}{N+1} = O(\log N)$$

$$\boxed{T(N) = O(N \log N)}$$

Estrategia de partición

- ✓ **Paso 1**: Intercambiar el pivote con el último elemento.
- ✓ **Paso 2**:
  - mover i de izquierda a derecha y j de derecha a izquierda.
  - Cuando i apunte a un numero mayor que el pivote, i se para.
  - Cuando j apunta a un elemento menor que el pivote, j se para.
  - Si i y j no se han cruzado entre si, intercambiar los elementos y continuar.
    - En otro caso dar por concluido el bucle.

- ✓ **Paso 3**: Intercambio del elemento i con el pivote.

El pivote 6, se coloca al final

i se para en el elemento mayor que el pivote, 8
j se para en el elemento menor que el pivote, 2

2 y 8 son intercambiados

i se para en el elemento mayor que el pivote, 9
j se para en el elemento menor que el pivote, 5

9 y 5 son intercambiados

i se para en el elemento mayor que el pivote, 9
j se para en el elemento menor que el pivote, 3

Intercambio del pivote y el elemento de la posición i

Partición con la mediana de tres

La partición con la mediana de tres requiere que encontremos la mediana del primer elemento, el central y el último.

Vector original

Resultado de ordenar tres elementos (el primero, el central y el último)

Intercambio del pivote con el **penúltimo**

Optimizaciones:

1. Intercambiar pivote con el penúltimo.
2. i puede empezar en *ini +1* y j en *fin-2*
3. Es seguro que i parará. En el peor caso encontrará el pivote (parando al darse la igualdad)
4. Es seguro que j parará. En el peor de los casos encontrará el primer elemento. (parando al darse la igualdad)

Elementos iguales al pivote.

¿Cómo debemos tratar los elementos que son iguales al pivote?

- ✓ Si i para y j no para, dará lugar a subconjuntos desiguales.
- ✓ Si i no para y i para, dará lugar a subconjuntos desiguales.

Supongamos el caso que todos los elementos del vector sean iguales:

- ✓ Si i para y j para, dará lugar a subconjuntos igualados. Al cruzarse en el medio y recolocar el pivote en i se asegura el caso mejor y el tiempo de ejecución O(NlogN).

- ✓ Si i no para y j no para, i termina en la última posición, pero el pivote se mantendría en la última posición que es la apuntada por i. Esto lleva al peor de los casos $O(N^2)$

La estrategia es que es mejor hacer intercambios innecesarios y crear subconjuntos iguales, que arriesgarse a crear subconjuntos desiguales.

Vectores pequeños

¿es aconsejable utilizar una rutina tan potente como quicksort cuando sólo hay que ordenar 10 elementos?.

Es mejor en este tamaño del problema preguntar por el tamaño del vector y terminar la ejecución utilizando el **algoritmo de inserción**. Evitando multitud de llamadas recursivas con elementos pequeños.

Experimentalmente, dependiendo de cada implementación, un buen límite está entre 5 y 20. Por ejemplo 10.

```java
public static void quickSort(Comparable[] a) {
    quickSort(a, 0, a.length - 1);
}
// Algoritmo quickSort que utiliza partición con la mediana de tres
// y un límite para detectar vectores pequeños
private static void quickSort(Comparable[] a, int ini, int fin) {
    if (ini + 10 > fin)
        ordenacionPorInsercion(a, ini, fin);
    else {
        // ordenar ini,medio,fin
        int medio = (ini + fin) / 2;
        if (a[medio].menorQue(a[ini]))
            intercambiarReferencias(a, ini, medio);
        if (a[fin].menorQue(a[ini]))
            intercambiarReferencias(a, ini, fin);
        if (a[fin].menorQue(a[medio]))
            intercambiarReferencias(a, medio, fin);

        // Colocar el pivote en la posición fin-1
        intercambiarReferencias(a, medio, fin - 1);
        Comparable pivote = a[fin - 1];

        // Empezar la partición
        int i, j;
        for (i = ini, j = fin - 1;;) {
            while (a[++i].menorQue(pivote))
                ;
            while ((pivote.menorQue(a[--j]) ) )
                ;
             if( i >= j )
                 break;
             intercambiarReferencias( a, i, j );
        }
        // Colocar el pivote
        intercambiarReferencias(a, i, fin - 1);
        quickSort(a, ini, i - 1); // Ordenar elementos pequeños
        quickSort(a, i + 1, fin); // Ordenar elementos grandes
    }
}
```

```java
private static void intercambiarReferencias(Comparable[] a, int b, int c) {
    Comparable temp = a[b];
    a[b] = a[c];
    a[c] = temp;

}
```

Selección rápida

El problema de la selección rápida consiste en, "***dado un vector de N elementos encontrar el k-ésimo menor elemento***".

Un caso especial es encontrar ***la mediana***, es decir, el N/2-ésimo elemento.

Una solución obvia se puede obtener ordenando el vector, pero dado que se pide menos información que en el problema de la ordenación es de esperar que se pueda resolver de forma más rápida que esta.

El algoritmo es muy parecido al quicksort, siendo su tiempo de ejecución lineal.

Los pasos que realiza **seleccionRapida(*S*,*k*)** son:

1. Si el número de elementos en *S* es 1, entonces es previsible que *k* también es 1, se devuelve el único elemento de *S*.

2. Elegir un elemento *v* de *S* como pivote.

3. Hacer una partición de *S-{v}* en I y D, igual que en quicksort.

4.
    a. Si *k* es menor o igual que el número de elementos en I, entonces el elemento buscado debe estar en *I*, por lo que se llama a **seleccionRapida(*I*,*k*).**

    b. Si *k* es exactamente igual a uno mas que el número de elementos de *I*, entonces el pivote es el k-ésimo menor elemento y se puede devolver como respuesta.

    c. En otro caso, el k-ésimo menor elemento estará en *D*. Hacemos entonces una llamada recursiva **selecciónRapida(*D*,*k*).**
    ==========

```java
public static void seleccionRapida(Comparable[] a, int ini, int fin,int k) {
    if (ini + 10 > fin)
        ordenacionPorInsercion(a, ini, fin);
    else {
        // ordenar ini,medio,fin
        int medio = (ini + fin) / 2;
        if (a[medio].menorQue(a[ini]))
            intercambiarReferencias(a, ini, medio);
        if (a[fin].menorQue(a[ini]))
            intercambiarReferencias(a, ini, fin);
        if (a[fin].menorQue(a[medio]))
            intercambiarReferencias(a, medio, fin);

        // Colocar el pivote en la posición fin-1
        intercambiarReferencias(a, medio, fin - 1);
        Comparable pivote = a[fin - 1];

        // Empezar la partición
        int i, j;
        for (i = ini, j = fin - 1;;) {
            while (a[++i].menorQue(pivote))
                ;
            while ((pivote.menorQue(a[--j]) ) )
                ;
             if( i >= j )
                 break;
             intercambiarReferencias( a, i, j );
        }
        // Colocar el pivote
        intercambiarReferencias(a, i, fin - 1);
        // Recursión; sólo cambia esta parte
        if (k-1 < i)
            seleccionRapida(a,ini,i-1,k);
        else if (k-1>i)
            seleccionRapida(a,i+1,fin,k);
    }
}
```

Para el caso peor

✓ La selección rápida sólo tiene **una llamada recursiva**, y **el peor caso** se dará, igual que en quicksort, cuando esta se hace solo con un elemento menos, y es de orden $O(N^2)$

Para el caso medio:

El coste en el caso medio de una llamada recursiva se obtiene haciendo la media de los costes sobre todos los posibles tamaños de los subproblemas.

$$T(I) = T(D) = \frac{T(0) + T(1) + T(2) + ... + T(N-1)}{N}$$

$$T(N) = (\frac{T(0) + T(1) + T(2) + ... + T(N-1)}{N}) + N$$

$$NT(N) = (T(0) + T(1) + T(2) + ... + T(N-1)) + N^2$$

para N-1

$$(N-1)T(N-1) = (T(0) + T(1) + T(2) + ... + T(N-2)) + (N-1)^2$$

restando

$$NT(N) - (N-1)T(N-1) = T(N-1) + 2N - 1$$

reorganizando

$$NT(N) = NT(N-1) + 2N - 1$$

Tenemos T(N) en función de T(N-1).
Desplegamos la ecuación, dividiendo por N

$$T(N) = T(N-1) + 2 - \frac{1}{N}$$

$$T(N-1) = T(N-2) + 2 - \frac{1}{N-1}$$

$$T(N-2) = T(N-3) + 2 - \frac{1}{N-2}$$

…….

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

Sumando las igualdades a ambos lados, y eliminando términos:

$$T(N) = T(1) + 2N + (-1)\left(\frac{1}{2} + \frac{1}{3} + ... + \frac{1}{N}\right)$$

$$T(N) = 2N + (-1)\left(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{N}\right) + 2$$

Donde el N-ésimo armónico es de O(logN)

$$\boxed{T(N) = O(N) + O(\log N) = O(N)}$$

Una cuota inferior para la ordenación

Teorema:

> "*Cualquier algoritmo que ordene utilizando únicamente comparaciones entre los elementos debe usar al menos* $\lceil \log(N!) \rceil$ *comparaciones, con alguna secuencia de entrada.*"

Demostración:

El número de posibles entradas es número de permutaciones diferentes de N elementos, que es N!.

Sea P_i el máximo número de permutaciones que puede ser consistentes con la información obtenida después de que el algoritmo ha realizado i comparaciones.

Sea F el número de comparaciones realizadas cuando el algoritmo termina.

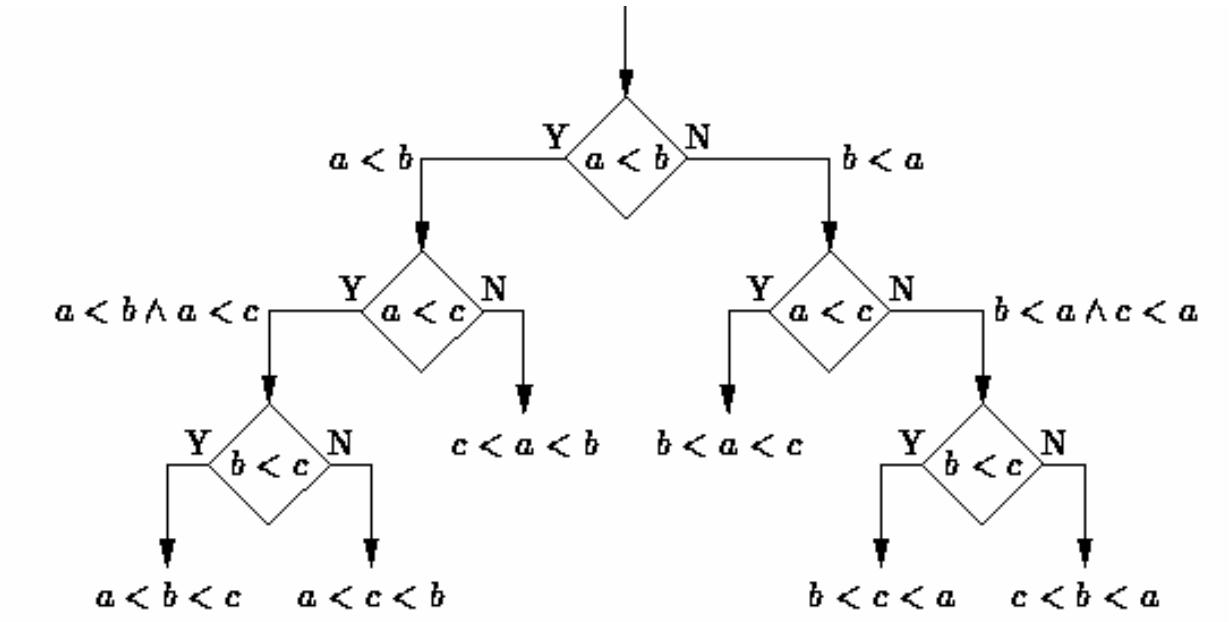
Se obtiene lo siguiente:

1. P_0 = N! ya que todas las permutaciones son posibles antes de la primera comparación.
2. P_F=1 ya que si fueran posibles más de una permutación, el algoritmo no habría terminado todavía.
3. $P_i >= P_{i-1}/2$, porque después de una comparación, cada permutación va a uno de estos grupos: el grupo de las aún posibles y el grupo de las que ya no son posibles

El efecto del algoritmo es ir desde el estado P_0, donde son posibles las N! permutaciones, al estado P_F, en el cual es sólo posible una permutación, con la restricción de que existe una entrada tal que en cada comparación, sólo se pueden eliminar la mitad de las permutaciones. Por el principio de sucesivas divisiones por la mitad sabemos que se necesitan $\lceil \log(N!) \rceil$ para esta entrada.

¿Y como es de grande $\lceil \log(N!) \rceil$?. Aproximadamente N logN - 1,44 N

$S = \{a, b, c\}$



$$
\begin{aligned}
\lceil \log_2 n! \rceil &\geq \log_2 n! \\
&\geq \sum_{i=1}^{n} \log_2 i \\
&\geq \sum_{i=1}^{n/2} \log_2 n/2 \\
&\geq n/2 \log_2 n/2 \\
&= \Omega(n \log n).
\end{aligned}
$$