

Algoritmos Paralelos

Algoritmos de Ordenamiento
(clase 27.11.15)

Prof. J.Fiestas

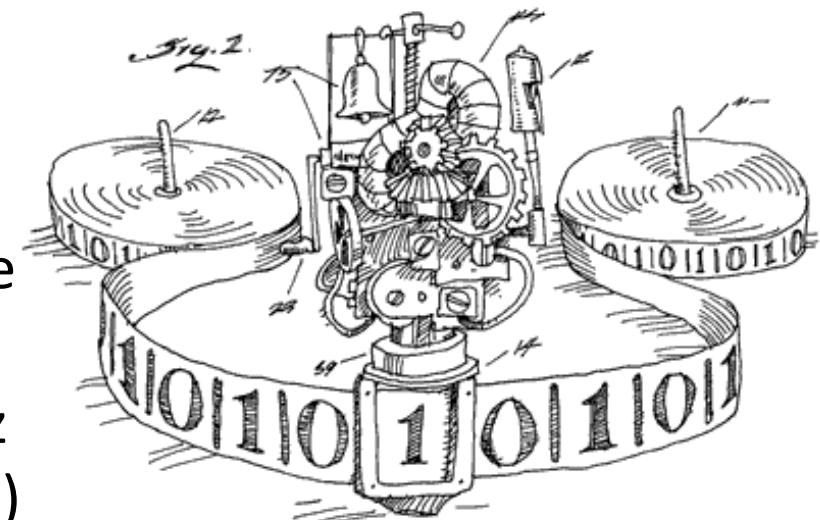
Algoritmos de ordenamiento

Se usan para ordenar un conjunto de objetos. Normalmente se clasifican en:

Algoritmos de ordenamiento interno, en los que el ordenamiento se hace en la memoria del ordenador

Algoritmos de ordenamiento externo, en los que el ordenamiento se hace en una memoria externa

La palabra **ordenador** (o computadora) tiene su origen en el francés *ordinateur*, y se refiere a “el que da órdenes”. Ya que inicialmente se vendió la idea del computador como una máquina inteligente capaz de calcular todo (**máquina de Turing**)



Ordenamiento de datos (sorting):

En C++:

```
void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));
```

base: puntero al primer objeto del array a ser ordenada

num: número de elementos del array apuntada por base

size: tamaño en bytes de cada elemento en el array

compar: puntero a una función que compara dos elementos. Con punteros como argumentos, la función define el orden de los elementos retornando <0, 0 o >0

```
int compar (const void* p1, const void* p2);
```

```
int compareMyType (const void * a, const void * b)
```

```
{
```

```
if ( *(MyType*)a < *(MyType*)b ) return -1;
```

```
if ( *(MyType*)a == *(MyType*)b ) return 0;
```

```
if ( *(MyType*)a > *(MyType*)b ) return 1;
```

```
}
```

Algoritmos de ordenamiento

Eficiencia del algoritmo, que depende del número de elementos a ser ordenados, la que se hace típicamente comparando pares de elementos. No hay algoritmo más eficiente que $O(n \log(n))$

Ademas, la eficiencia siempre representa casos promedios, a lo que debeadirse casos extremos: *best/worst case*

De acuerdo al input, hay casos en que se puede resolver el ordenamiento hasta en $O(n)$, i.e. Bubble sort.

También hay que considerar la constante en el orden de complejidad $O(c*n)$, la que va a variar de uno a otro algoritmo.

Algoritmos de ordenamiento

Espacio de trabajo, hay algoritmos que requieren espacio adicional de trabajo, y por consiguiente mas memoria.

Estabilidad, también puede requerir mas tiempo y espacio. Un algoritmo que mantiene el orden de los objetos a travez del proceso es más estable.

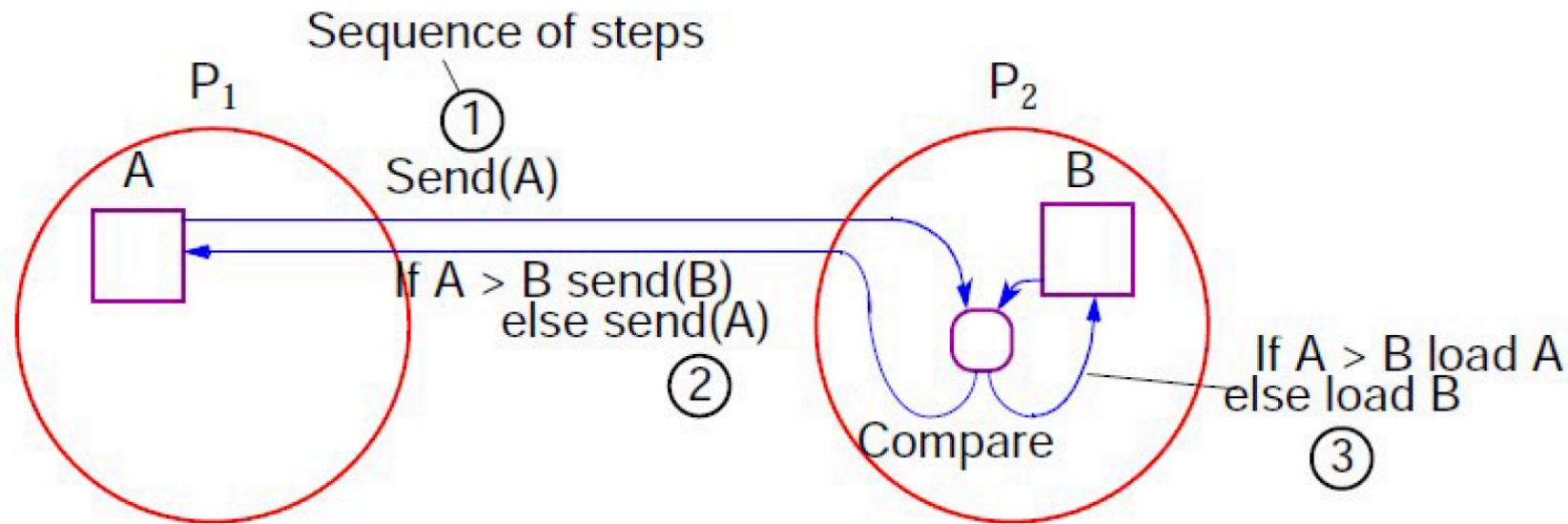
Algoritmos de ordenamiento en paralelo

- Es frecuentemente usado en varias aplicaciones
- Se utiliza para ordenar una secuencia de valores en n procesos
- El mejor algoritmo de ordenamiento secuencial tiene un costo $O(n \log n)$
- Con un algoritmo paralelo, utilizando n procesos, se obtiene un costo óptimo de $O(n \log n)/n = O(\log n)$

Comparar e intercambiar, compartiendo mensajes

Ordenamiento secuencial requiere comparar valores e intercambiar sus posiciones en la secuencia.

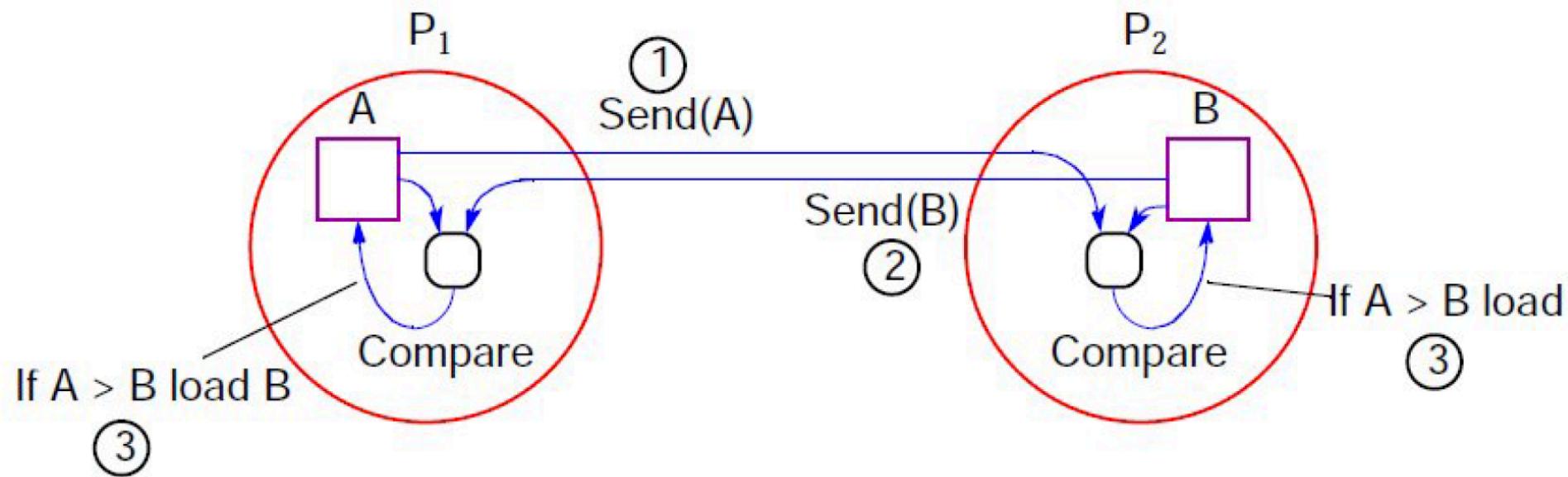
Caso 1: P_1 envía A a P_2 . Este compara B y A, y envía el mínimo de A y B a P_1



Comparar e intercambiar, compartiendo mensajes

Ordenamiento secuencial requiere comparar valores e intercambiar sus posiciones en la secuencia.

Caso 2: P_1 envía A a P_2 . P_2 envía B a P_1 . P_1 calcula $A=\min(A,B)$, P_2 calcula $B=\max(A,B)$

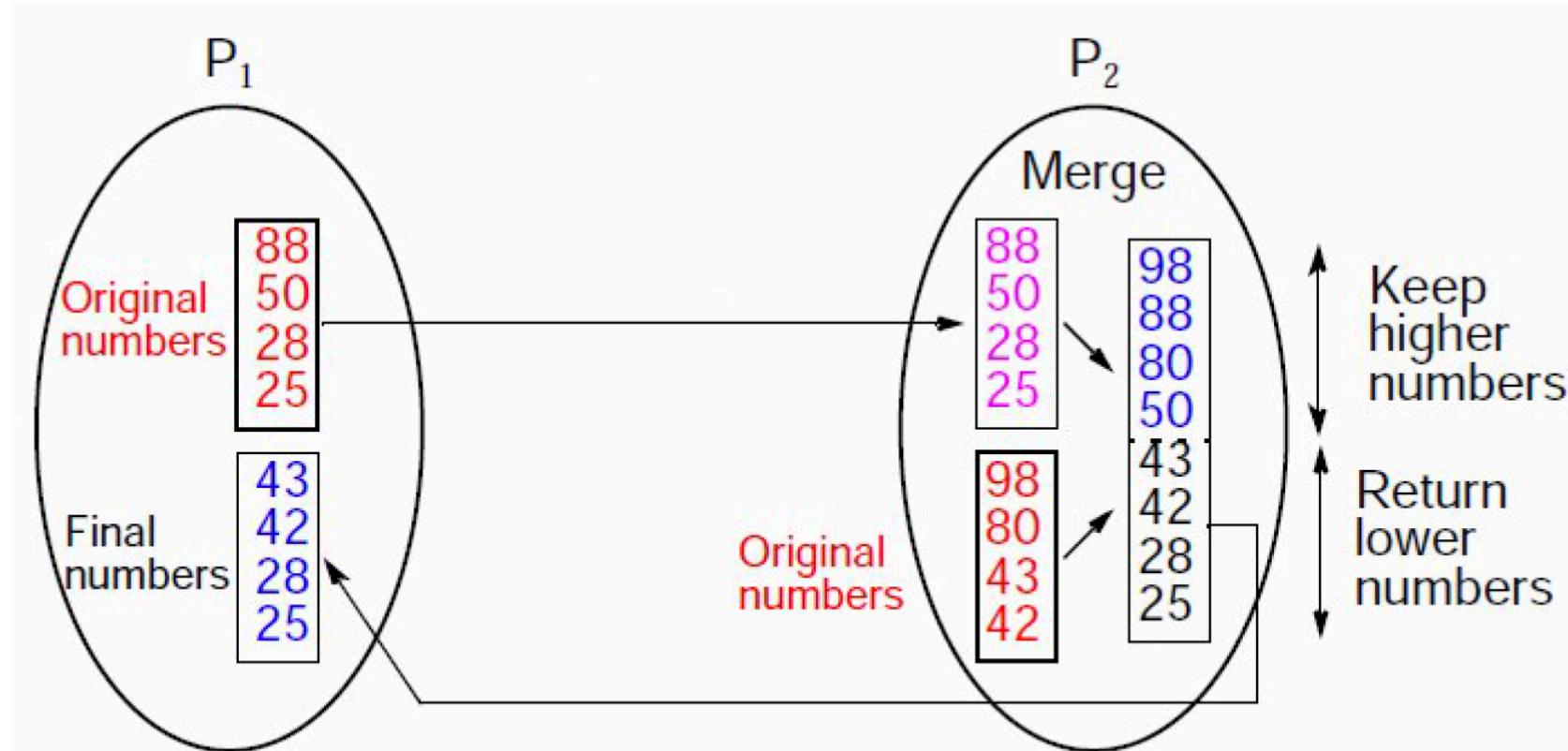


Partición de data

Caso 1:

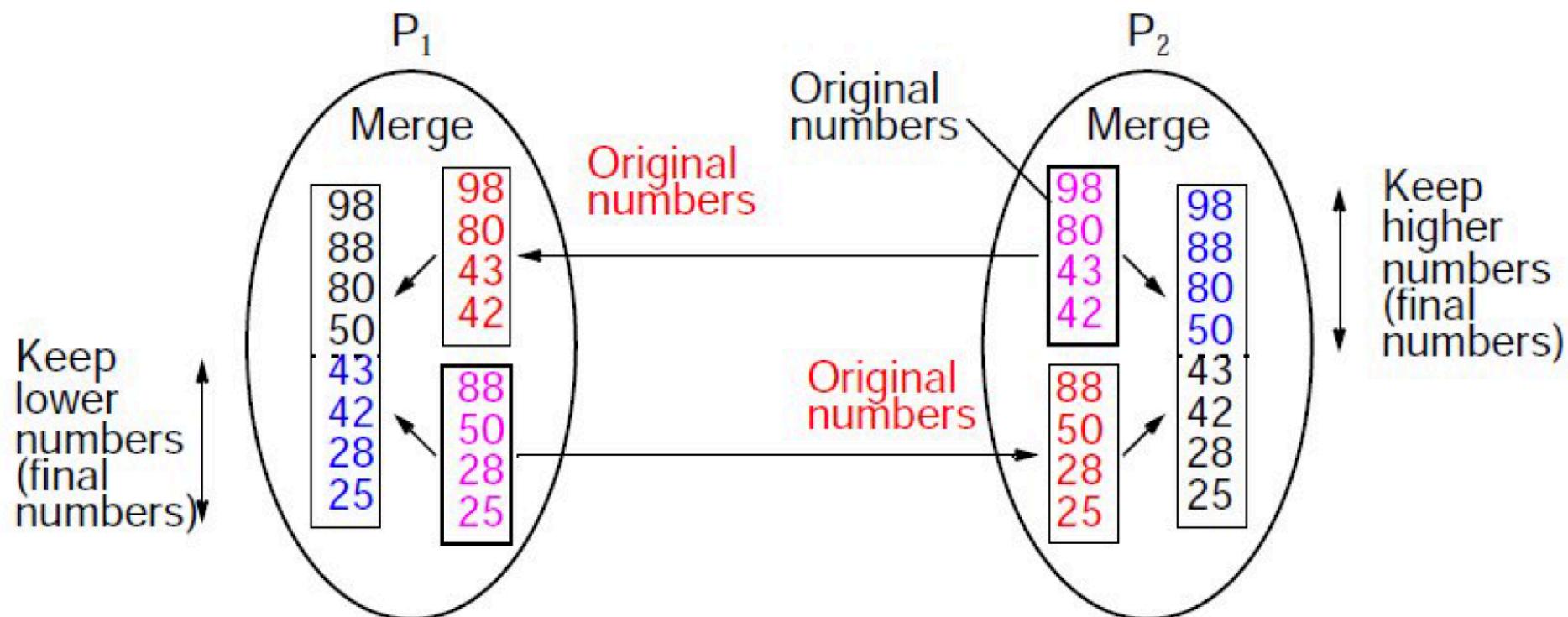
n números y p procesos

n/p números en cada proceso



Partición de data

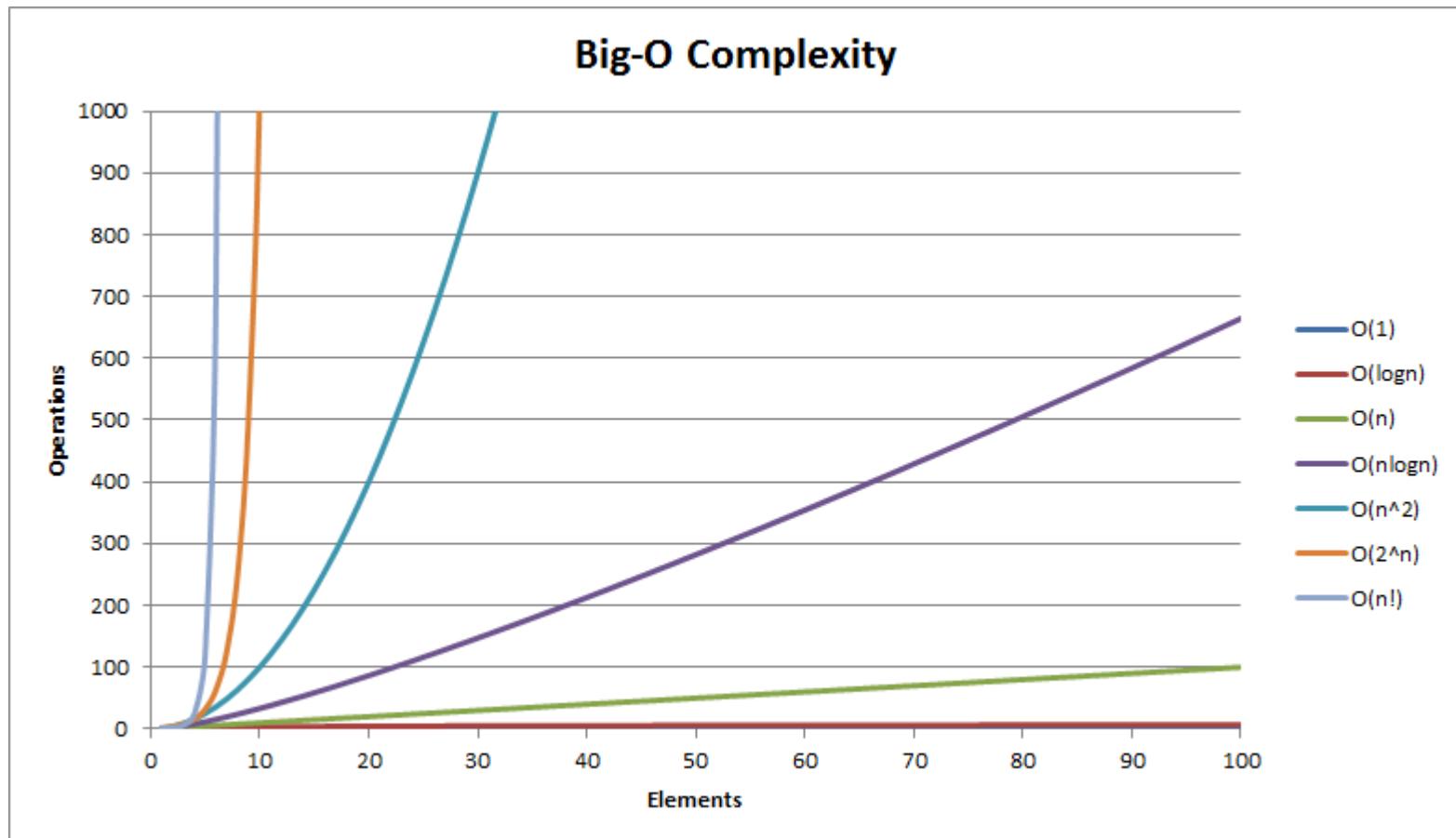
Caso 2:



Algoritmos de ordenamiento

Sort	Time			Space	Stability	Remarks
	Average	Best	Worst			
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Always use a modified bubble sort
Modified Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$	Constant	Stable	Stops after reaching a sorted array
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	$O(n * \log(n))$	$O(n * \log(n))$	$O(n * \log(n))$	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	$O(n * \log(n))$	$O(n * \log(n))$	$O(n * \log(n))$	Depends	Stable	On arrays, merge sort requires $O(n)$ space; on linked lists, merge sort requires constant space
Quicksort	$O(n * \log(n))$	$O(n * \log(n))$	$O(n^2)$	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

Algoritmos de ordenamiento

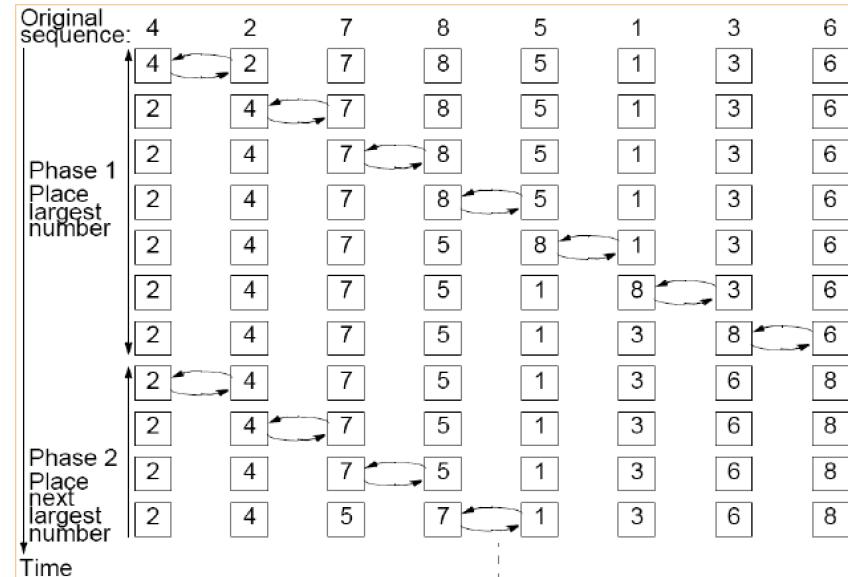


Ordenamiento de burbuja (Bubble Sort)

Se inicia a la izquierda, comparando elementos adjacentes y empujando el mayor a la derecha. Luego continuar el algoritmo para N-1 elementos. Es el algoritmo considerado mas simple. La cantidad de comparaciones necesarias en n elementos es $O(n^2)$:

$$n-1 + n-2 + \dots + 1 = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$

, por lo que no se recomienda para un número grande de elementos.



Ordenamiento de burbuja (Bubble Sort)

Algoritmo de
Ordenamiento de Burbujas

Numerical Recipes
aconseja no usar este
algoritmo en códigos de
producción, ya que es poco
eficiente en comparación a
otros algoritmos

```
void sort( double * list, int list_size )
{
    double temp;
    int i, exchange;

    exchange = 1;
    while( exchange != 0 )
    {
        exchange = 0;
        for( i=0; i<list_size-1 ; i++ )
        {
            if ( list[i] > list[i+1] )
            {
                temp = list[i];
                list[i] = list[i+1];
                list[i+1] = temp;
                exchange++;
            }
        }
    }
}
```

Algoritmo de Ordenamiento de Burbujas en paralelo

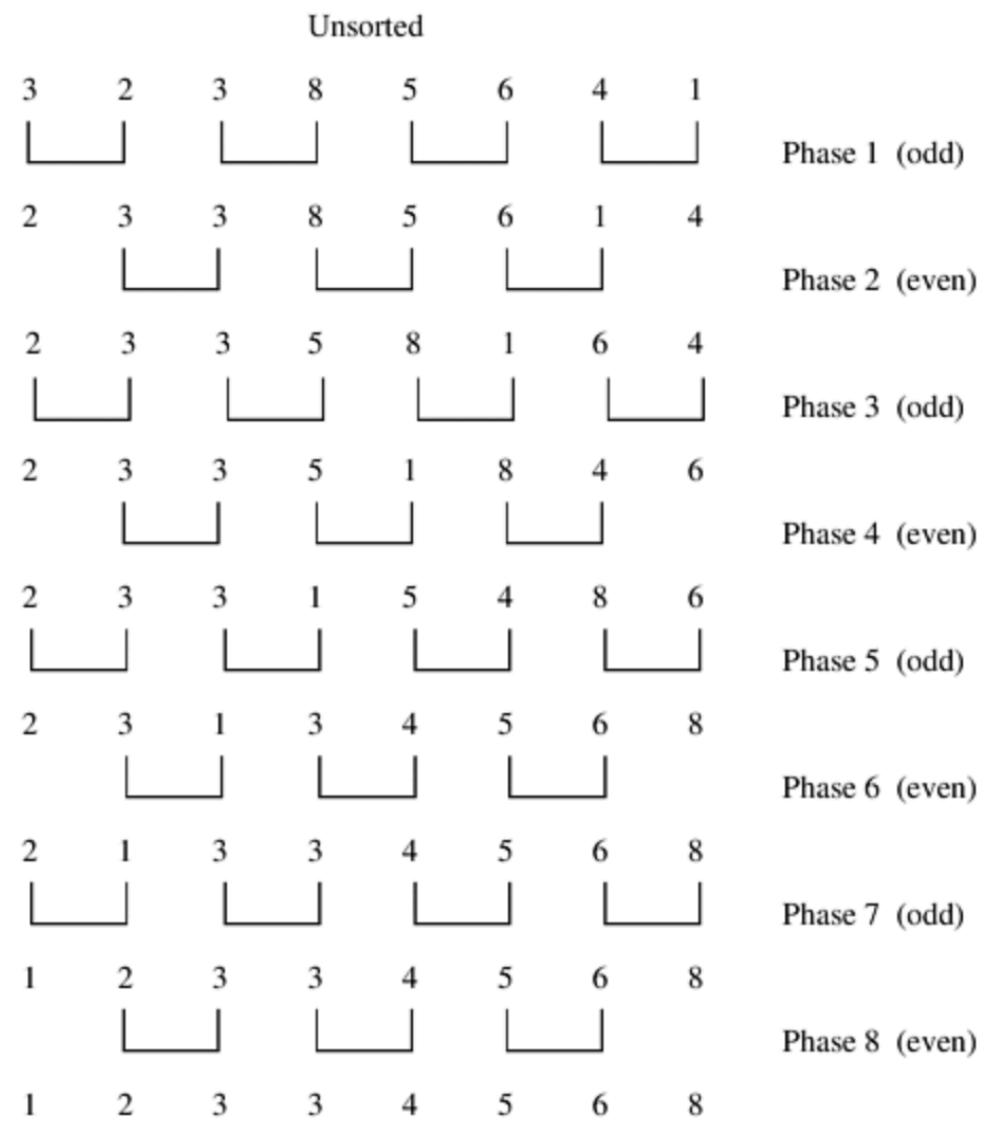
La forma mas simple es dividir la lista de elementos en partes iguales entre $N-1$ procesos para un cluster de N procesadores. Guardando al nodo 0 como administrador del cálculo.

Cada proceso ordenará una lista parcial y la re-enviará al nodo 0 para un ordenamiento final, el cual puede resultar complejo.

Algoritmo de Ordenamiento de Burbujas

Algoritmo secuencial de transposición

impar-par, por el que se ordenan n elementos en n fases, cada una de las cuales requiere $n/2$ operaciones de comparacion e intercambio. En la fase impar, los índices impares se comparan con sus vecinos derechos. En la fase par, se hace lo mismo con los índices pares.

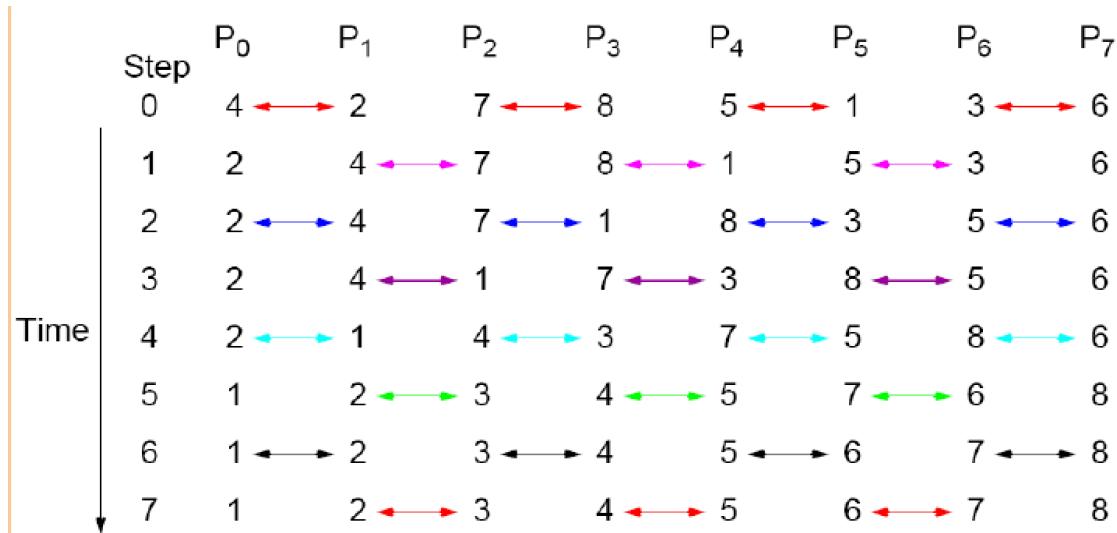


Algoritmo de Ordenamiento de Burbujas en paralelo

Otra forma es implementar un **algoritmo de transposición impar-par**, por el que

se ordena en pasos alternados (par e impar):

- Procesos 'pares' comparan con su vecino derecho
- Procesos 'impares' comparan números con su vecino derecho



Algoritmo de Ordenamiento de Burbujas en paralelo

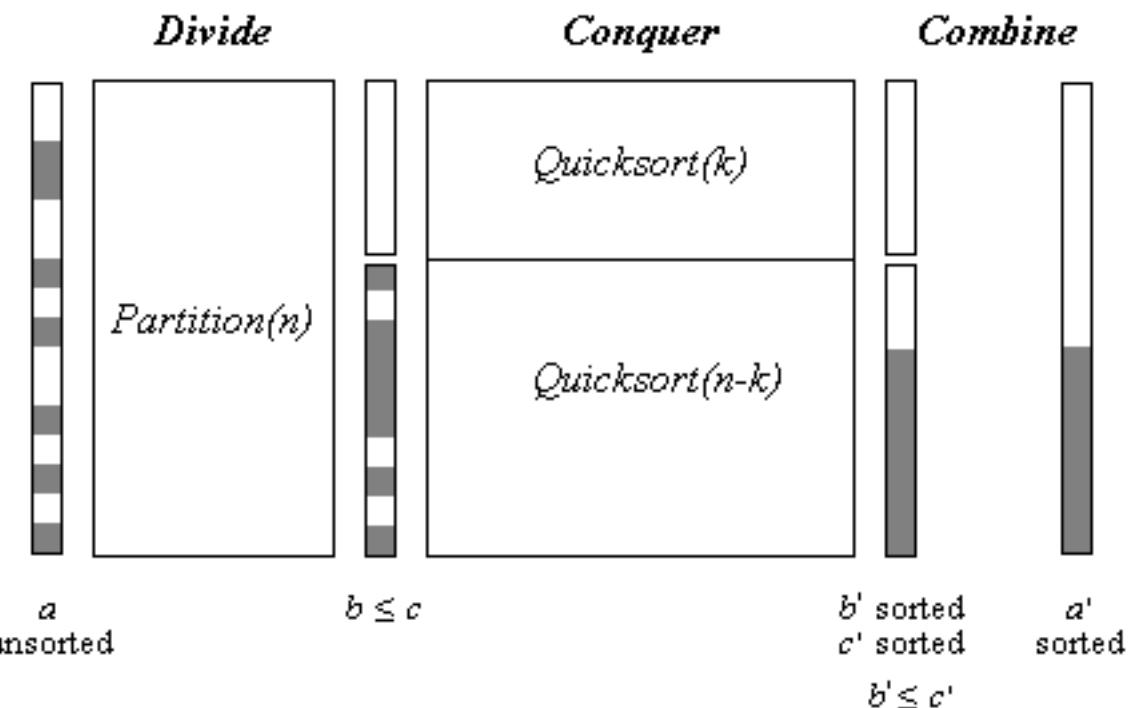
En cada fase del algoritmo, los procesos pares e impares ejecutan una comparación-cambio con sus vecinos derechos, lo que requiere $O(1)$, es decir un total de $O(n)$ por las n fases. Esto hace $O(n^2)$. Esto lo hace menos eficiente que un algoritmo de ordenamiento secuencial con $O(n \log n)$.

Una forma de optimizar es utilizar menos procesos ($p < n$), cada uno con n/p elementos, que son ordenados internamente (merge o quicksort) en un tiempo $O((n/p) \log(n/p))$. Luego, se ejecutan p fases ($p/2$ pares y $p/2$ impares), de operaciones de comparación-intercambio. En cada fase, se realizan $O(n/p)$ comparaciones para fusionar dos bloques, y $O(n/p)$ tiempo de comunicación

$$T_P = \overbrace{\Theta\left(\frac{n}{p} \log \frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n)}^{\text{communication}}$$

Ordenamiento rápido (quicksort)

Es uno de los más rápidos y también simples algoritmos de ordenamiento. Funciona recursivamente, según el principio de *divide y conquista*. La cantidad de comparaciones necesarias en n elementos es $O(n \log n)$ o $O(N^2)$:



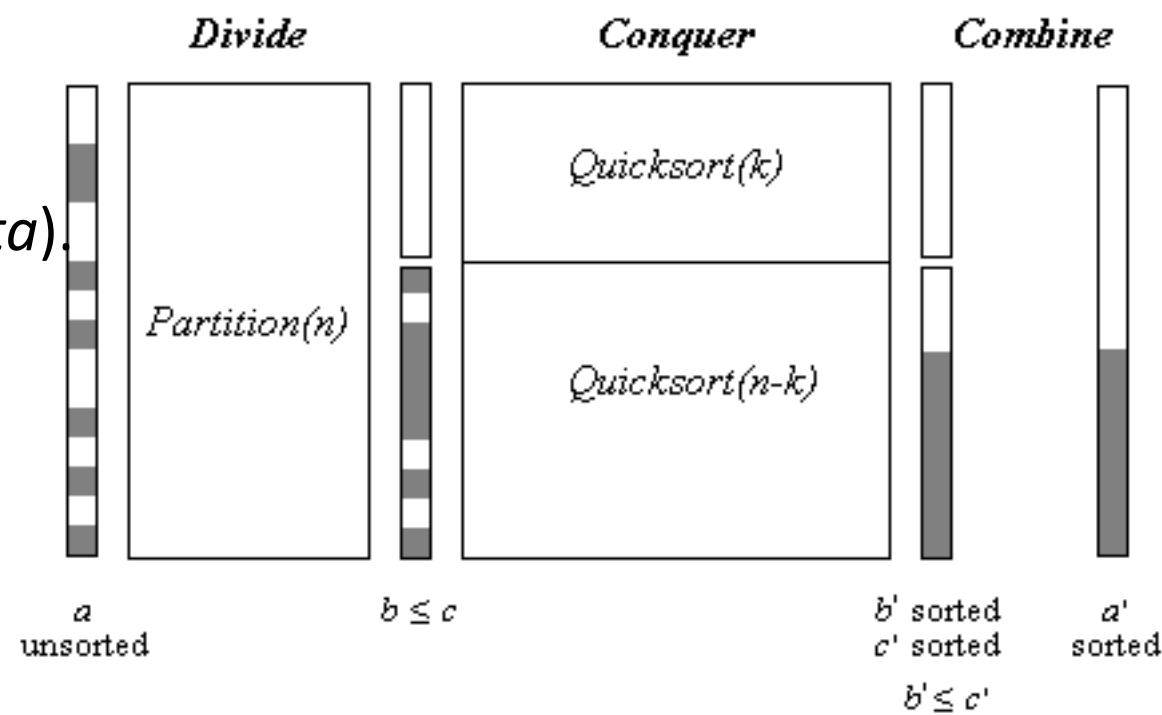
Ordenamiento rápido (quicksort)

En el diagrama, la columna inicial representa un conjunto de ceros (blanco) y unos (gris).

Primero se *divide* la columna en partes b y c, tal que cada elemento de b es menor al de c.

Luego se ordenan recursivamente los subconjuntos de la misma manera (*conquista*).

Finalmente se unirán las partes en la serie final, ordenada.



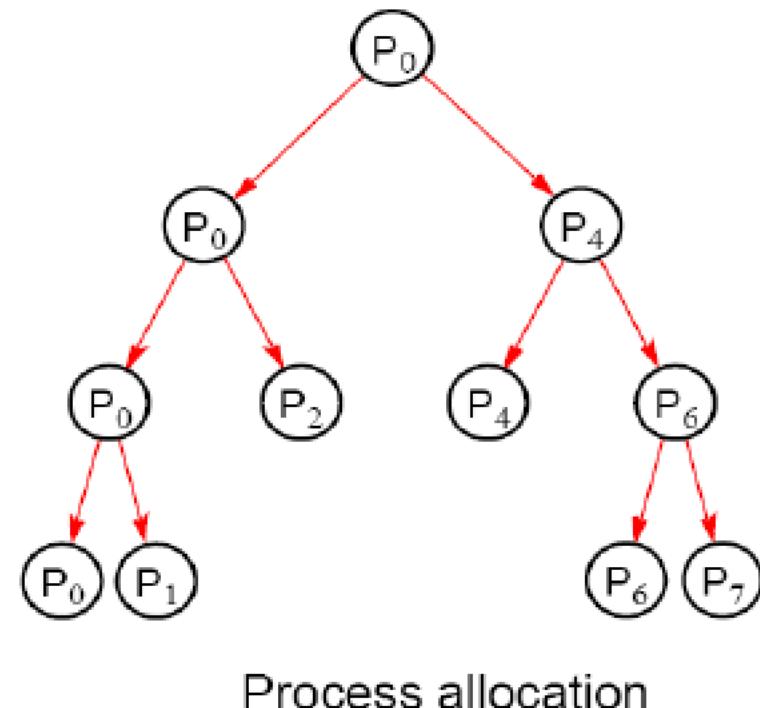
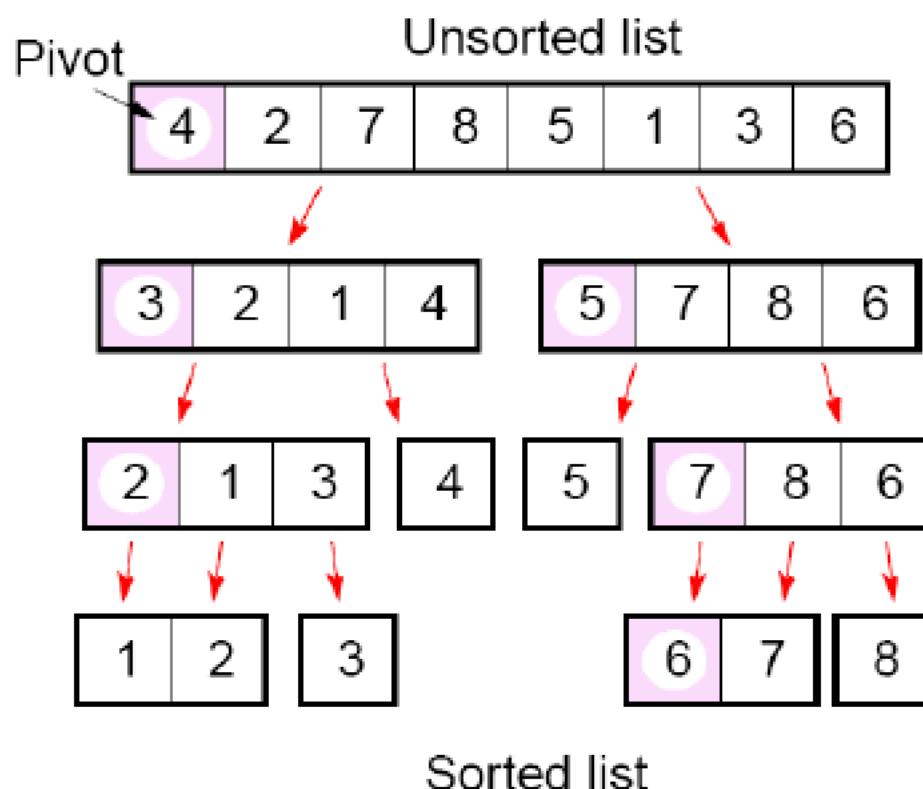
Ordenamiento rápido (quicksort)

Para cada división se escoje un elemento de comparación (*pivot*), tal que los elementos menores al *pivot* iran a un subgrupo, y los mayores al otro. La recursión termina cuando los subgrupos consisten en un solo elemento.

```
void quicksort( double list[], int m, int n ) {  
    int i,j,k;  
    double key;  
    if( m < n) {  
        k = choose_pivot(m,n);  
        swap(&list[m],&list[k]);  
        key = list[m];  
        i = m+1;  
        j = n;  
        while(i <= j) {  
            while((i <= n) && (list[i] <= key))  
                i++;  
            while((j >= m) && (list[j] > key))  
                j--;  
            if( i < j)  
                swap(&list[i],&list[j]);  
        }  
        // swap two elements  
        swap(&list[m],&list[j]);  
        // recursively sort the lesser list  
        quicksort(list,m,j-1);  
        quicksort(list,j+1,n);  
    }  
}
```

Ordenamiento rápido (quicksort) en paralelo

En paralelo, quicksort genera un árbol binario, donde cada nivel del árbol representa una iteración diferente. Con una buena elección del pivot, la altura es $O(\log n)$, que viene a ser el número de iteraciones.



Ordenamiento rápido (quicksort) en paralelo

Para explotar las ventajas de memoria compartida (OPENMP):

- El conjunto de n elementos se divide entre p procesos
- Se selecciona un pivot por uno de los procesos, que lo comunica al resto de procesos
- Cada proceso partitiona el grupo en dos, basados en el pivot
- Se aplica el algoritmo recursivamente a cada subgroupo
- Clave es que todos los procesos tienen acceso a la memoria compartida

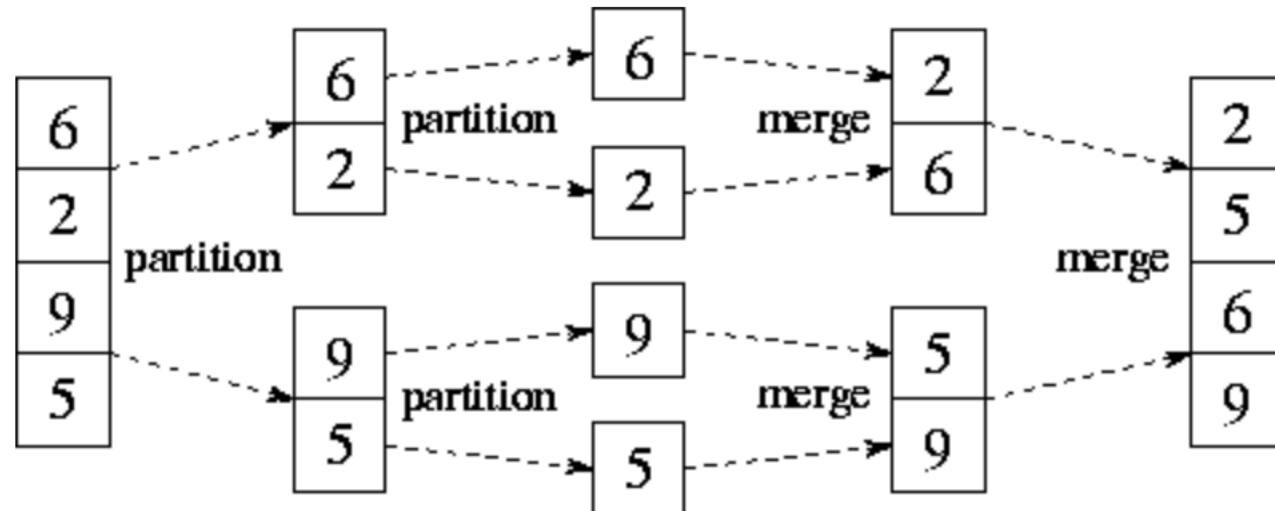
$$T_P = \overbrace{\Theta\left(\frac{n}{p} \log \frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p} \log p\right)}^{\text{array splits}} + \Theta(\log^2 p).$$

- Selección del pivot muy importante para el costo de ejecución.

Mergesort

Método:

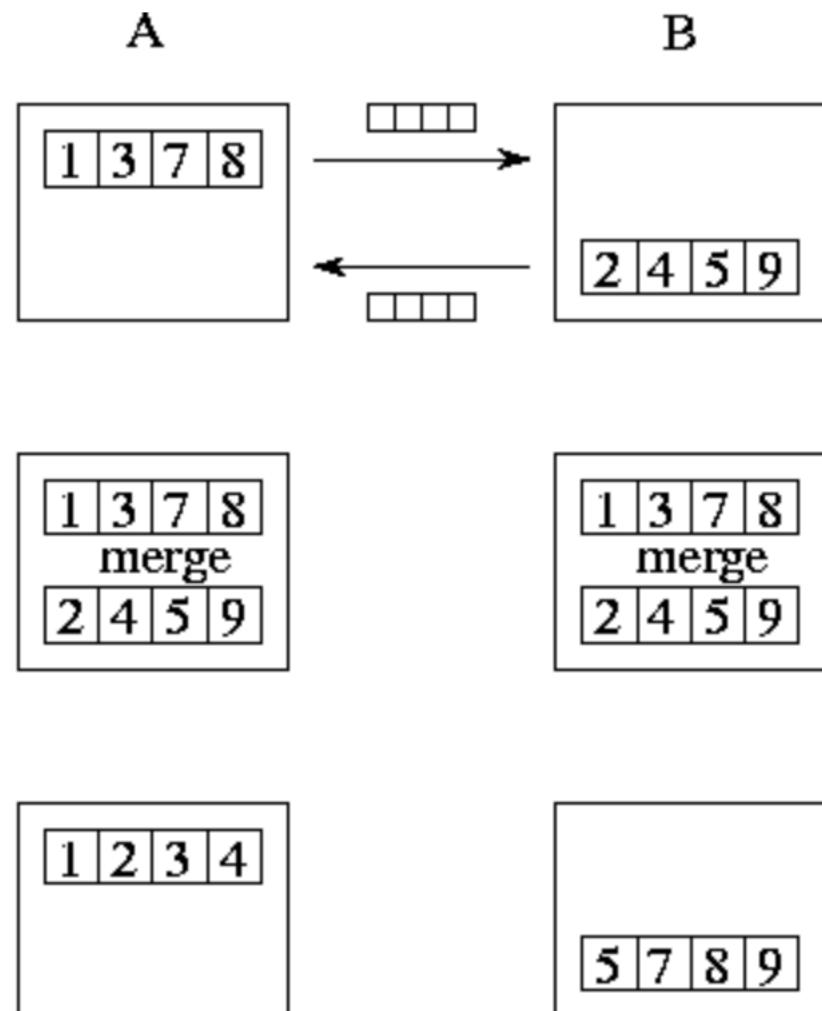
- Se divide la lista de n elementos en n sub-listas (cada una con un elemento)
- Se fusionan sublistas para crear nuevas sublistas ordenadas, hasta que solo queda una sublista



Mergesort

Comparación e intercambio:
Consiste en fusionar dos
sublistas ordenadas de
longitud M, contenidas en
procesos A y B

- Cada proceso envía información al otro
- Proceso A identifica los M menores elementos y descarta el resto
- Proceso B hace lo mismo con los mayores valores.



Mergesort

Cada proceso ordena su sublistas en serie. Luego cada proceso $P=2^d$, ejecuta **parallel_merge()** d veces

```
procedure parallel_mergesort(myid, d, data, newdata)
begin
    data = sequential_mergesort(data)
    for dim = 1 to d
        data = parallel_merge(myid, dim, data)
    endfor
    newdata = data
end
```

Costo total del algoritmo en paralelo

$$\begin{aligned} T &= \frac{T_{comp}}{P} + T_{comm} \\ &= t_c \frac{N}{P} \left(\log N + \frac{d(d-1)}{2} \right) + t_s \frac{d(d+1)}{2} + t_w \frac{N}{P} \frac{d(d+1)}{2} \\ &\approx \left(t_c \frac{N}{2P} + t_s + t_w \frac{N}{P} \right) \frac{(\log P)^2}{2} \quad \text{if } (\log P)^2 \gg \log N \end{aligned}$$

Ordenamiento por casilleros (bucket sort)

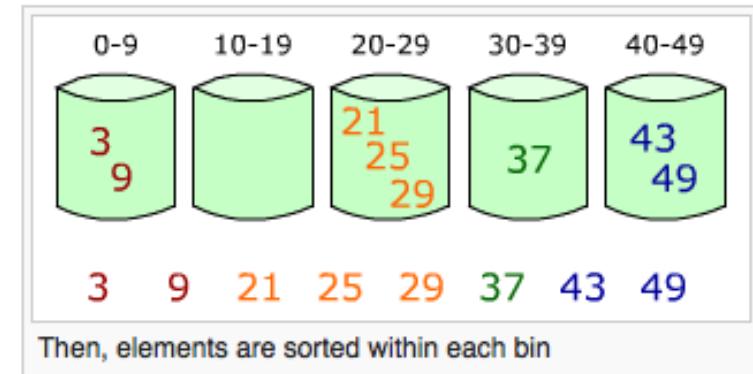
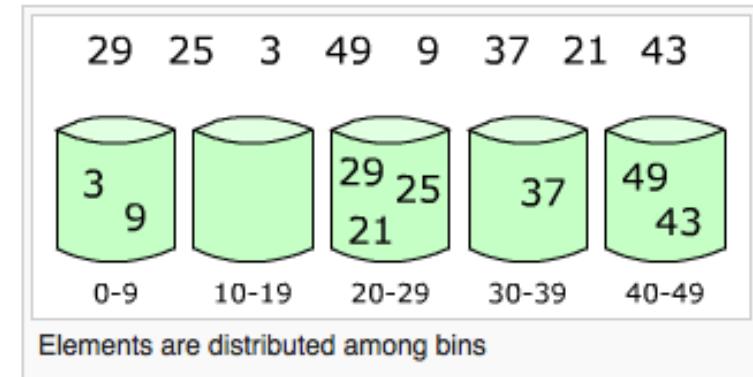
Para ordenar n elementos, i.e. en el intervalo $[a,b]$:

Se crean p depósitos vacíos (buckets),

Cada elemento se ubica en el depósito apropiado

Se ordenan los p depósitos.

El costo es $O(n \log (n/p))$



Ordenamiento por casilleros (bucket sort)

El algoritmo en serie puede hacer la distribución parametrizando el conjunto de elementos en el intervalo $[0,1]$, y asignando cada elemento en la posición i a cada depósito en la posición $n*arr[i]$

```
void bucketSort(float arr[], int n)
{
    // Create n empty buckets
    vector<float> b[n];
    // Put array elements in different buckets
    for (int i=0; i<n; i++)
    {
        int bi = n*arr[i]; // Index in bucket
        b[bi].push_back(arr[i]);
    }
    // Sort individual buckets
    for (int i=0; i<n; i++)
        sort(b[i].begin(), b[i].end());
    // Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}
```

Ordenamiento por casilleros (bucket sort)

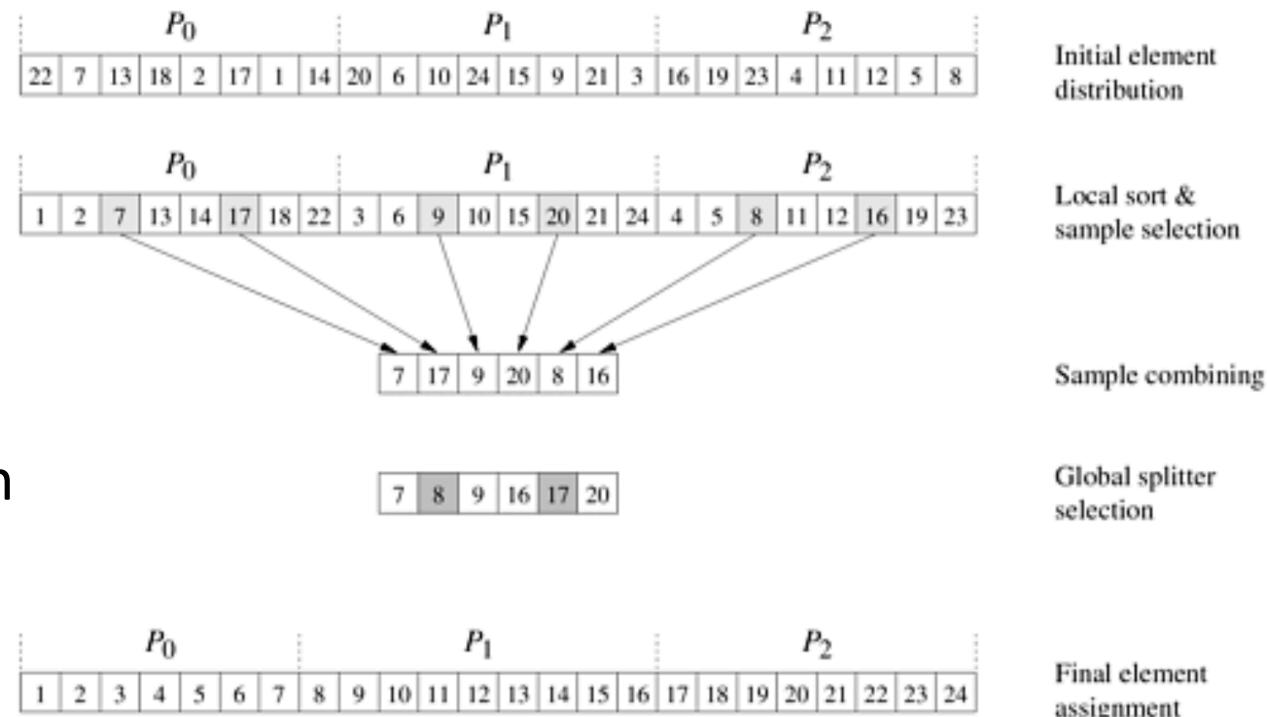
En memoria distribuída, cada proceso p recibe n/p elementos, siendo el número de buckets $m=p$

- Cada proceso reparte los n/p elementos en p subbloques
- Cada proceso envía subbloques a los otros procesos, para que cada proceso tenga solo los elementos del bloque que le pertenece
- Cada proceso ordena su bloque internamente, e.g. utilizando quicksort.

Ordenamiento por casilleros (bucket sort)

Ya que los elementos no estan distribuídos uniformemente en $[a,b]$, los bloques no tendran un tamaño comparable y la eficiencia se perjudica. Aqui se utiliza **sample sort**:

Se selecciona una muestra s del total n y se ordena, escogiendo $m-1$ elementos (splitters) del resultado. Estos dividen la muestra en bloques de tamaño m



Ordenamiento bitónico (bitonic sort)

Una secuencia bitónica, consiste en dos secuencias. Una creciente y una decreciente

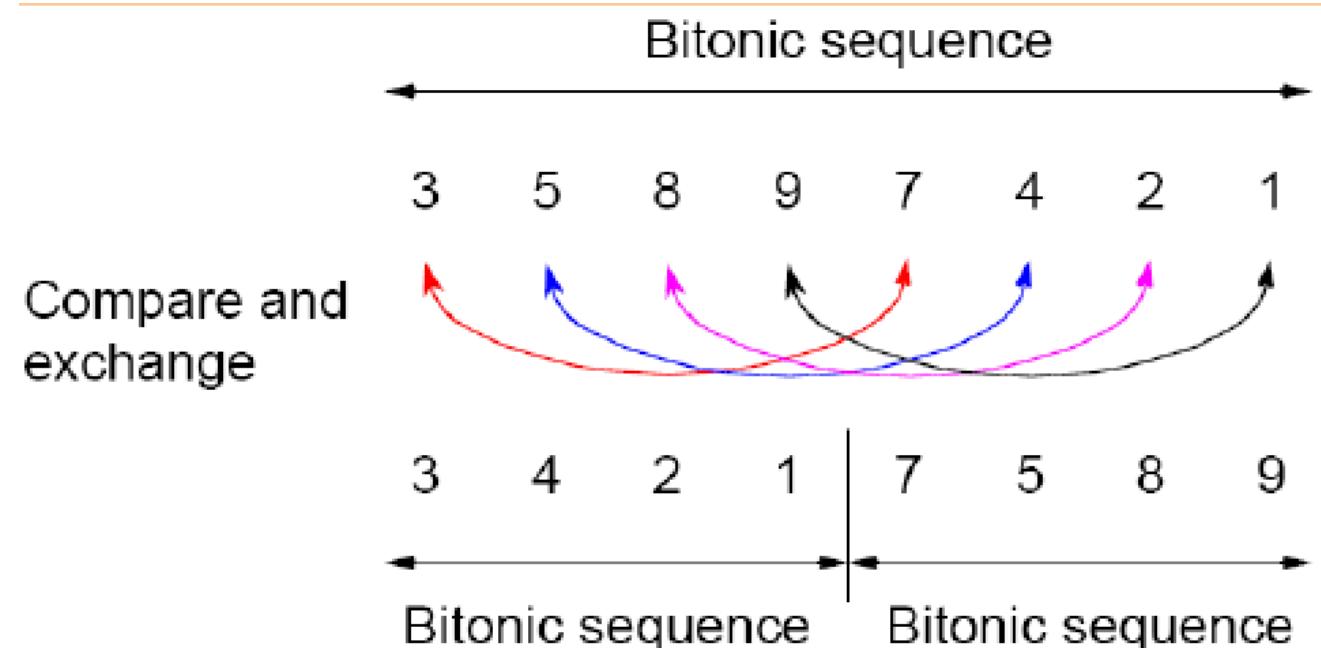
$$a_0 < a_1 < a_2, a_3, \dots, a_{i-1} < a_i > a_{i+1}, \dots, a_{n-2} > a_{n-1}$$

donde i varia de 1 a n

Si se realiza una operación de comparación e intercambio de a_i con $a_{i+n/2}$, para todo i, se obtendrán dos secuencias bitónicas, donde los elementos de una secuencia serán menores que los elementos de la otra.

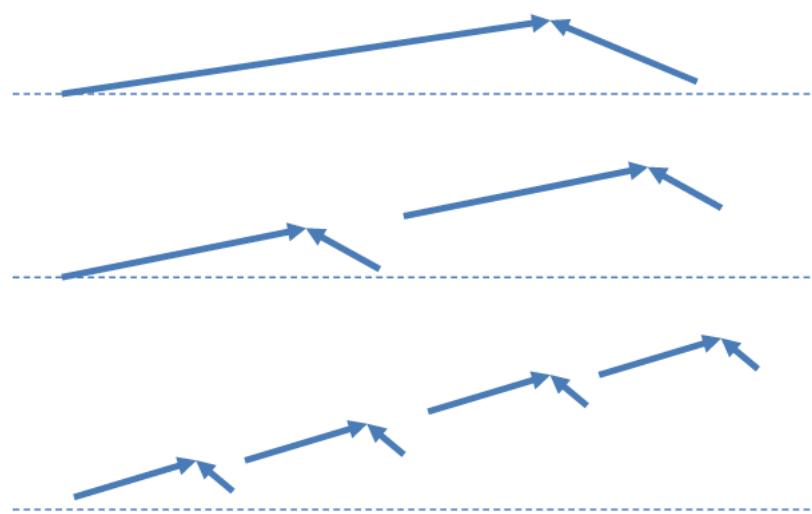
Ordenamiento bitónico (bitonic sort)

De la secuencia
3,5,8,9,7,4,2,1 se
obtendrá



Ordenamiento bitónico (bitonic sort)

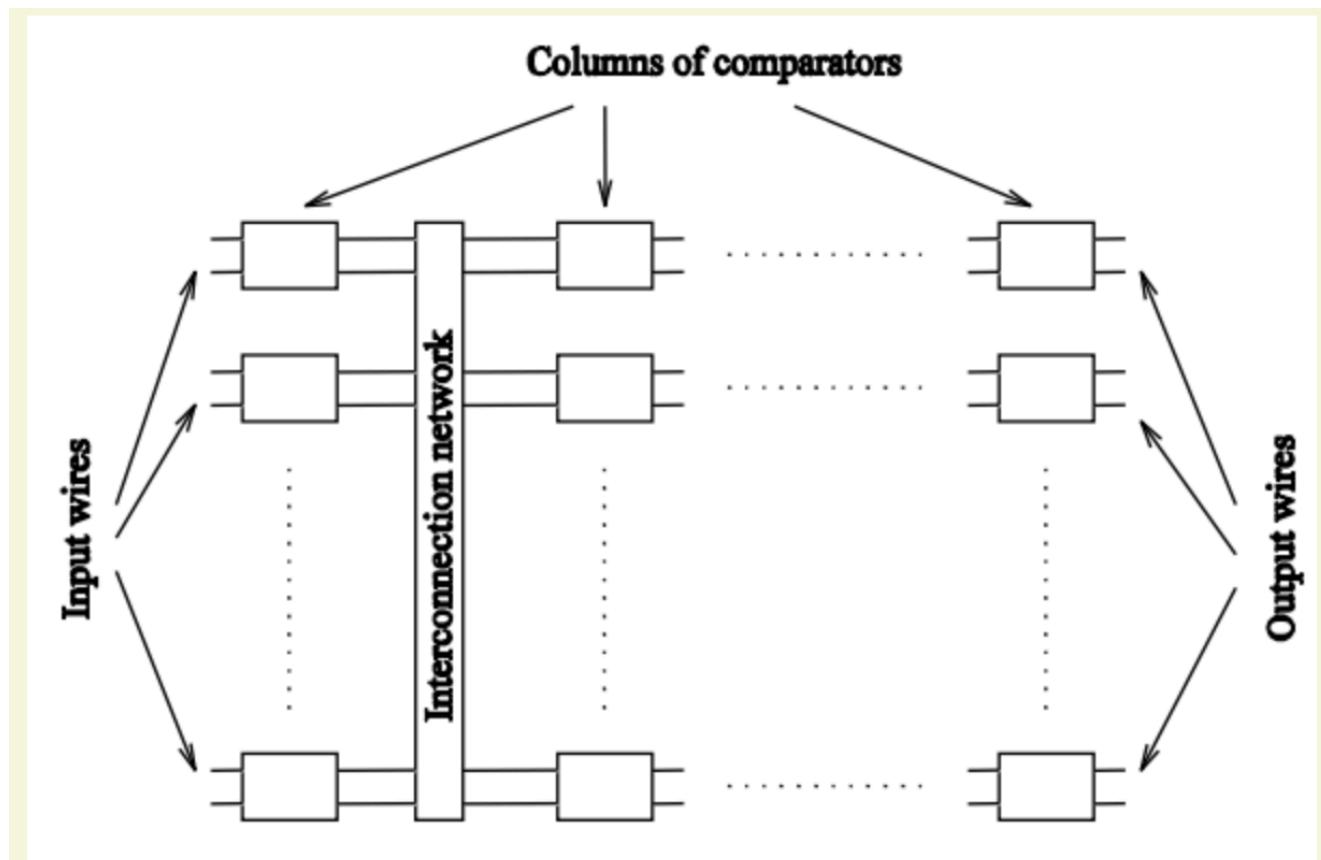
Dada una secuencia bitónica de n elementos, aplicar método recursivo. Luego de $n-1$ pasos cada secuencia bitónica consistirá de solo dos elementos



Ordenamiento bitónico (bitonic sort)

La secuencia desordenada se convierte en una bitónica

La serie se descompone en secuencias menores hasta que la secuencia completa este ordenada



Ordenamiento bitónico (bitonic sort)

Si se realiza una operación de comparación e intercambio de a_i con $a_{i+n/2}$, para todo i , se obtendrán dos secuencias bitónicas, donde los elementos de una secuencia serán menores que los elementos de la otra.

- Let $s = \langle a_0, a_1, \dots, a_{n-1} \rangle$ be a bitonic sequence such that

— $a_0 \leq a_1 \leq \dots \leq a_{n/2-1}$, and
— $a_{n/2} \geq a_{n/2+1} \geq \dots \geq a_{n-1}$



- Consider the following subsequences of s

$$s_1 = \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \dots, \min(a_{n/2-1}, a_{n-1}) \rangle$$

$$s_2 = \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \dots, \max(a_{n/2-1}, a_{n-1}) \rangle$$



- Sequence properties

— s_1 and s_2 are both bitonic
— $\forall x \forall y x \in s_1, y \in s_2, x < y$

- Apply recursively on s_1 and s_2 to produce a sorted sequence
- Works for any bitonic sequence, even if $|s_1| \neq |s_2|$

Ordenamiento bitónico (bitonic sort)

Entonces, aplicando este procedimiento recursivamente en las listas resultantes, se obtiene una secuencia ordenada en orden ascendente.

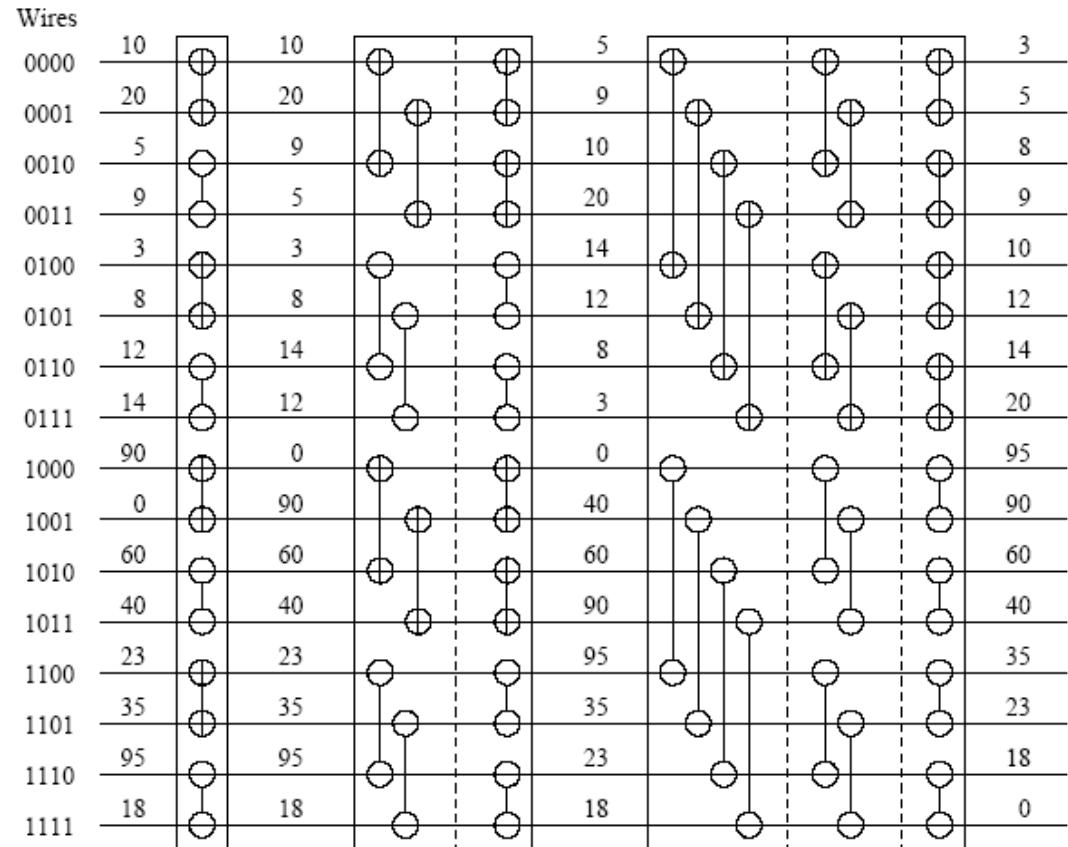
- Let $s = \langle a_0, a_1, \dots, a_{n-1} \rangle$ be a bitonic sequence such that
 - $a_0 \leq a_1 \leq \dots \leq a_{n/2-1}$, and
 - $a_{n/2} \geq a_{n/2+1} \geq \dots \geq a_{n-1}$
- Consider the following subsequences of s
$$s_1 = \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \dots, \min(a_{n/2-1}, a_{n-1}) \rangle$$
$$s_2 = \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \dots, \max(a_{n/2-1}, a_{n-1}) \rangle$$
- Sequence properties
 - s_1 and s_2 are both bitonic
 - $\forall x \forall y x \in s_1, y \in s_2, x < y$
- Apply recursively on s_1 and s_2 to produce a sorted sequence
- Works for any bitonic sequence, even if $|s_1| \neq |s_2|$



Ordenamiento bitónico (bitonic sort)

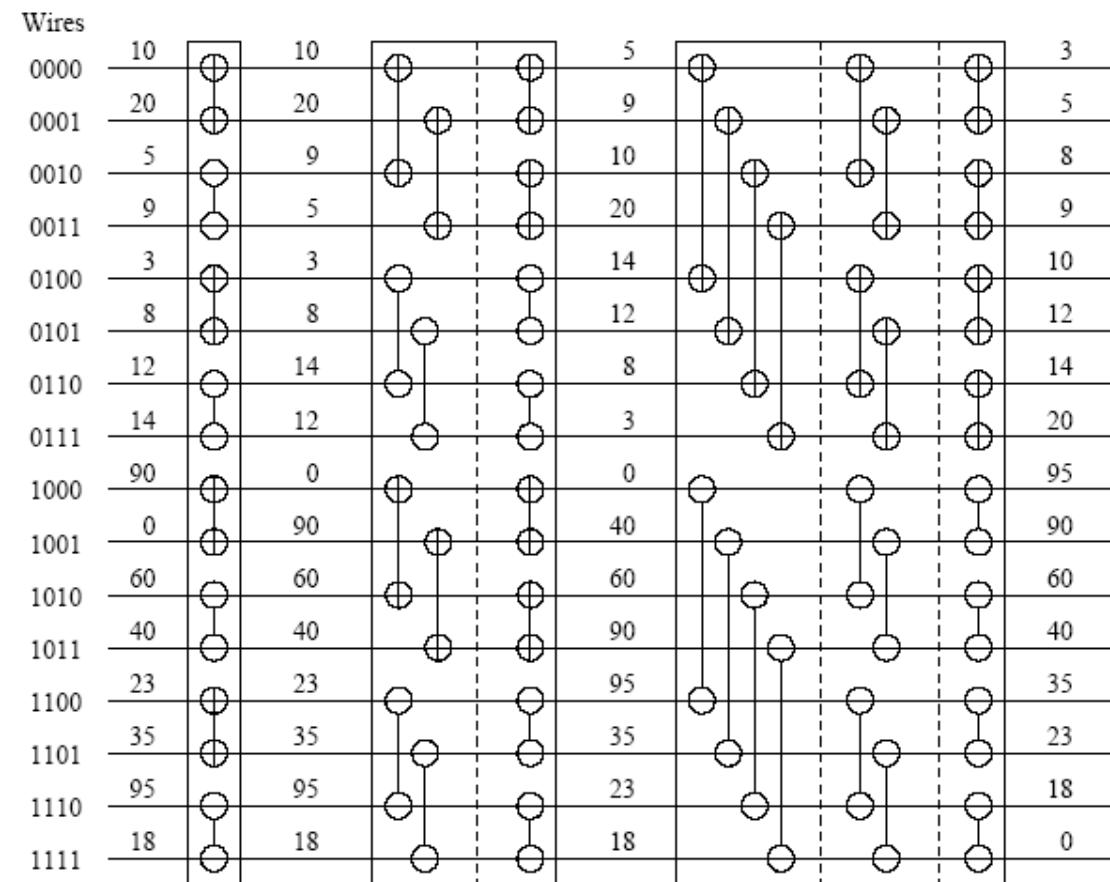
Para ello hay que obtener primero una secuencia bitónica, partiendo de la lista inicial de elementos desordenados.

Ya que un par de números son bitónicos, se puede particionar la lista inicial en pares, y unirlos consecutivamente en listas mas grandes.



Ordenamiento bitónico (bitonic sort)

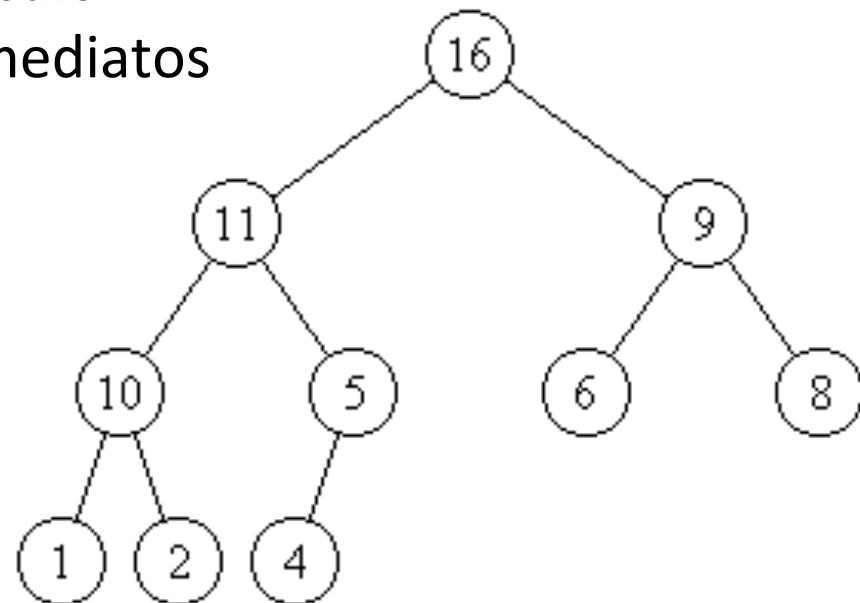
El costo en serie es de
 $O(n \log^2 n)$, pero mejora considerablemente en
paralelo



Ordenamiento por montículos (heapsort)

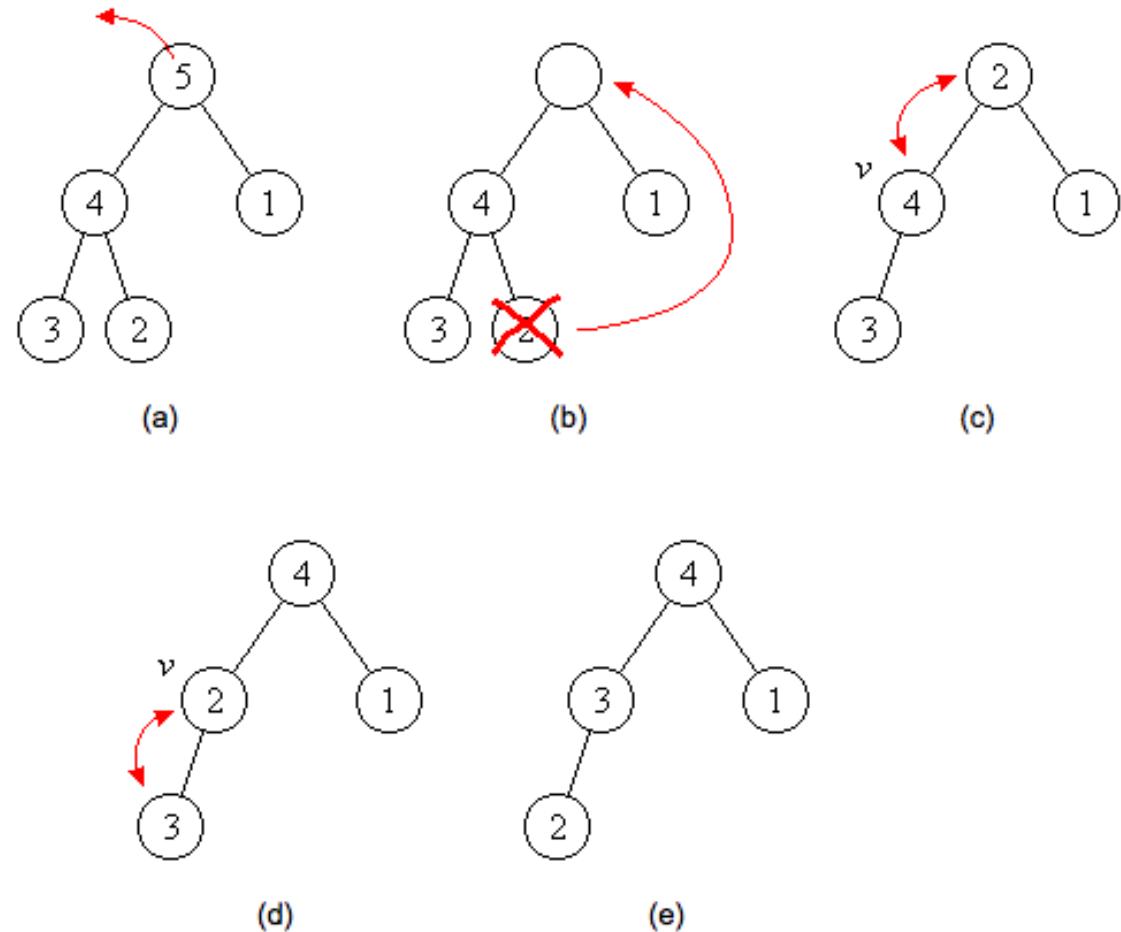
Tiene complejidad $O(n \log n)$. Utiliza una estructura de datos llamada heap (montículo), que se basa en la definición de un árbol binario, en el cual todos sus nodos están ocupados por los elementos que serán ordenados.

Un nodo es raíz de un montículo (heap) cuando sus nodos inmediatos tienen valores menores o iguales



Ordenamiento por montículos (heapsort)

El conjunto de elementos se estructura en un montículo (heap) con las propiedades mencionadas. Así, el mayor elemento estará en la raíz, y se retira.

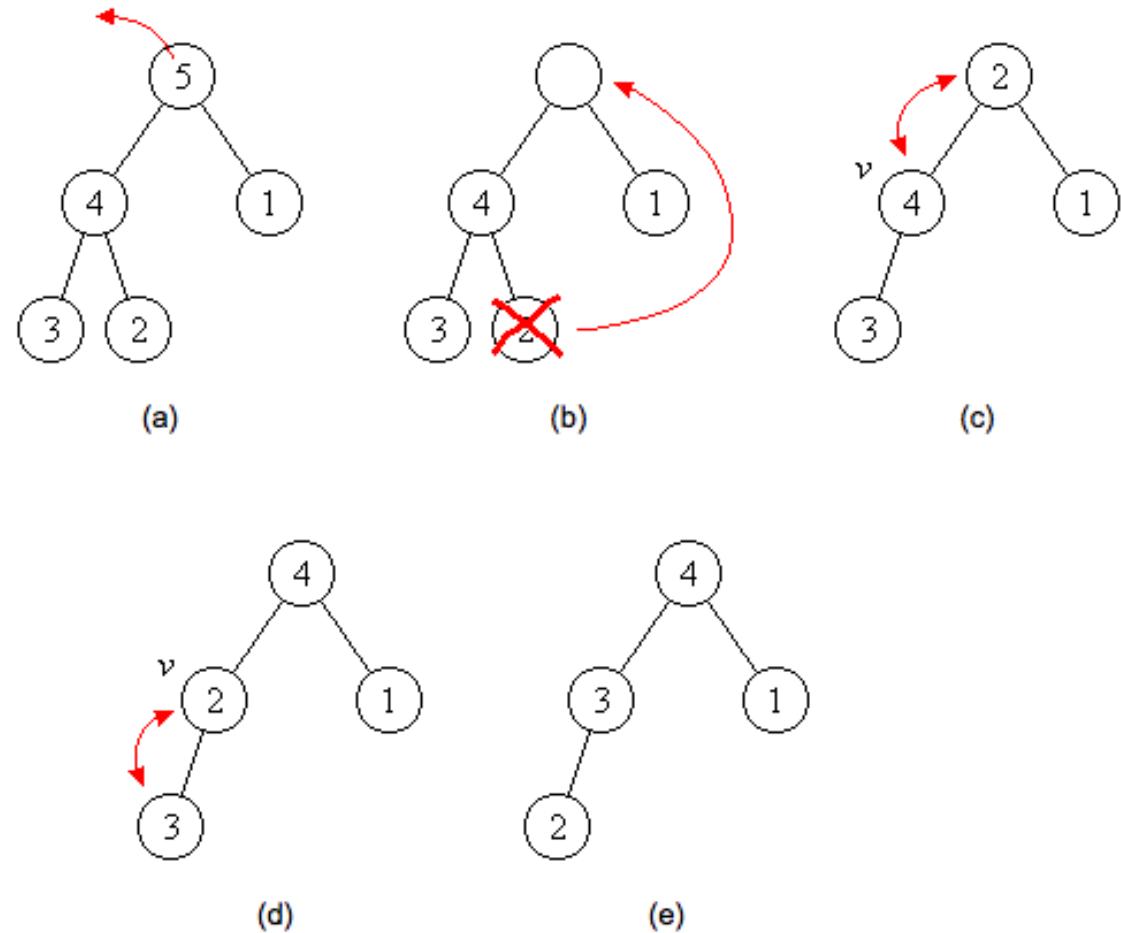


Este algoritmo es difícil de paralelizar

Ordenamiento por montículos (heapsort)

Luego se lleva el elemento con la mayor profundidad a la raíz y se compara con sus vecinos para tener nuevamente el mayor valor en la raíz.

Se continúa con el siguiente elemento hasta que se llega al elemento de mayor profundidad.



Este algoritmo es difícil de paralelizar

Ejercicio 1

Paralelize el programa bitonic.c utilizando MPI o OPENMP (a libre elección)

El programa utiliza una lista de números enteros en list.txt

Para ejecutarlo, realice:

```
gcc bitonic.c
```

```
./a.out list.txt
```

El programa solo ordena listas de 2^x elementos

Modifique el programa para que genere una lista de 2^{19} números enteros y los ordene

Mida los tiempos necesarios para ordenar 2^{19} números utilizando 1, 4 y 8 procesos

Comente que espera al utilizar memoria distribuída o compartida.

Ejercicio 2

Adapte la función del algoritmo bucket sort de la lámina 28 a un programa en paralelo usando memoria distribuída (MPI). Utilice una lista de 2^{19} floats generados aleatoriamente y mida los tiempos de ejecución en 1, 4 y 8 procesos. Compare tiempos con el ejemplo anterior y comente los costos de tiempo de estos algoritmos.