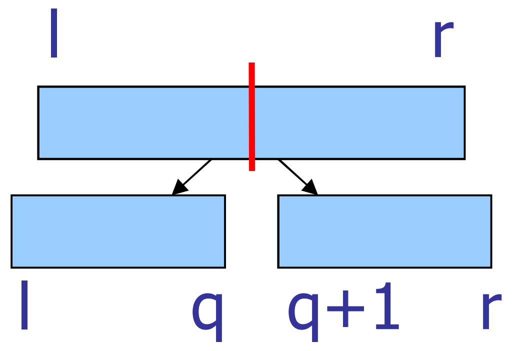
Recursive sorting algorithms

Paolo Camurati

Dip. Automatica e Informatica
Politecnico di Torino

Merge Sort

- Division
  - Partition the array into 2 subarrays L and R with respect to the array's middle element



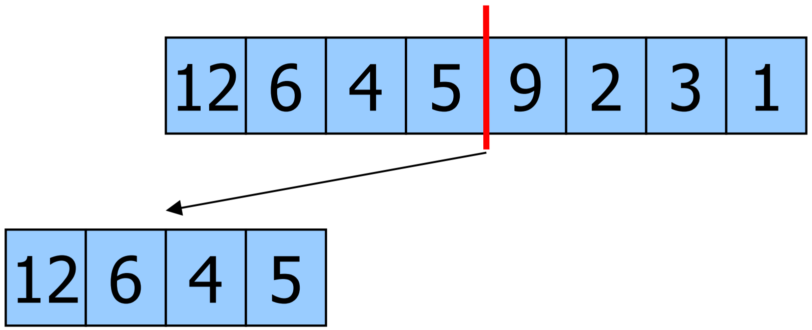
Merge Sort

- Recursion
  - Merge sort on subarray L
  - Merge sort on subarray R
  - Termination condition: with 1 (l=r) or 0 (l>r) elements the array is sorted
- Ricombination
  - **Merge** 2 sorted subarrays into one sorted array

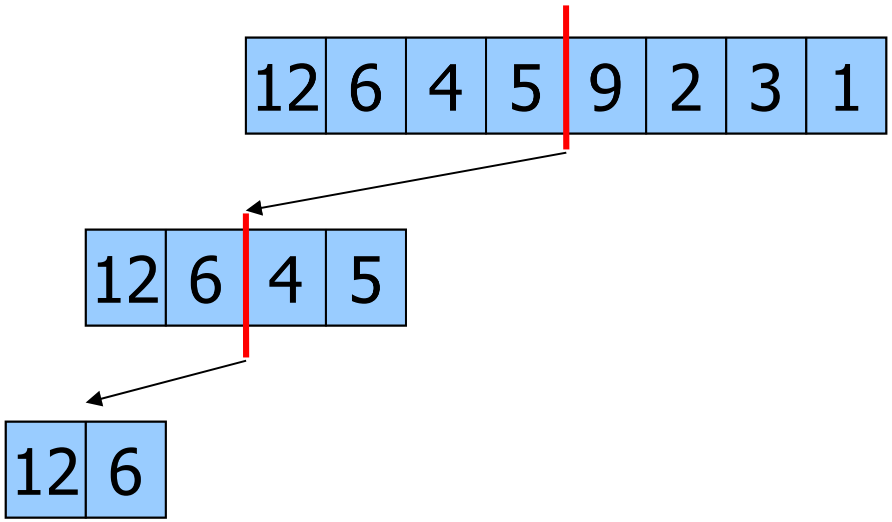
Example

12	6	4	5	9	2	3	1

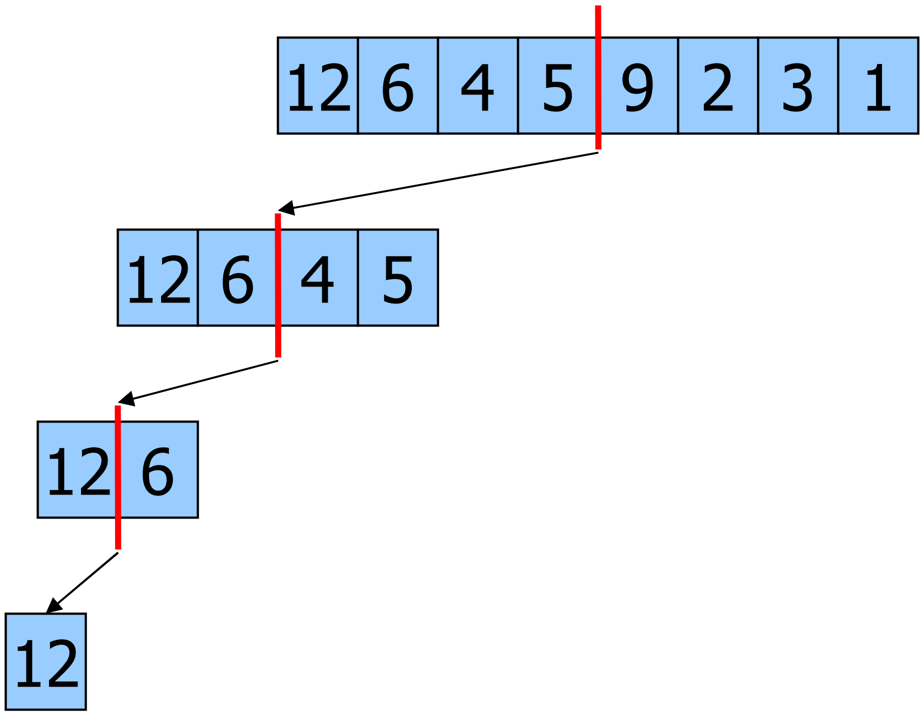
Example



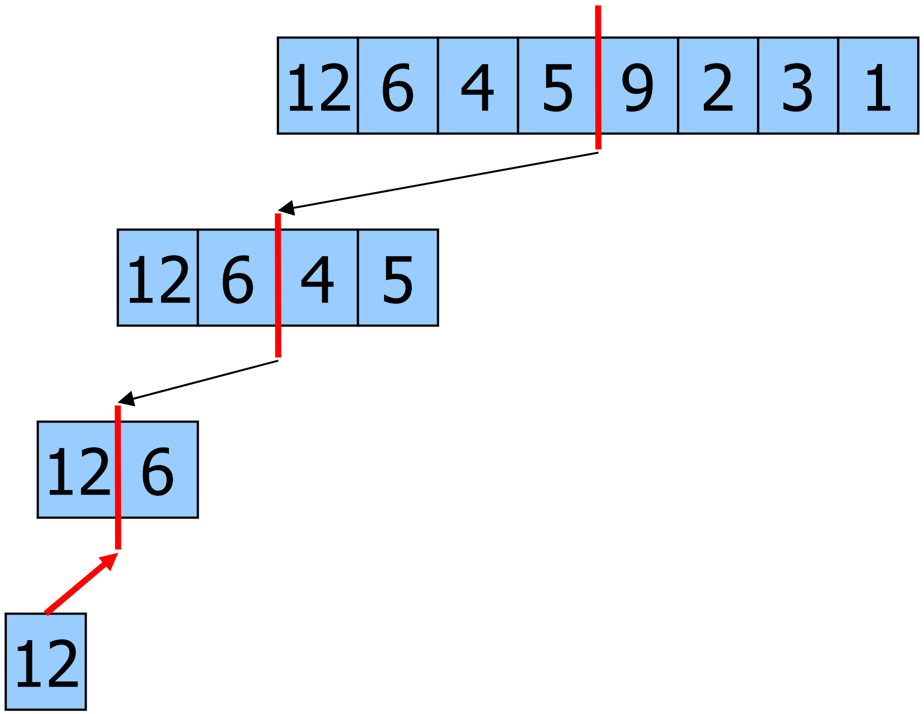
Example



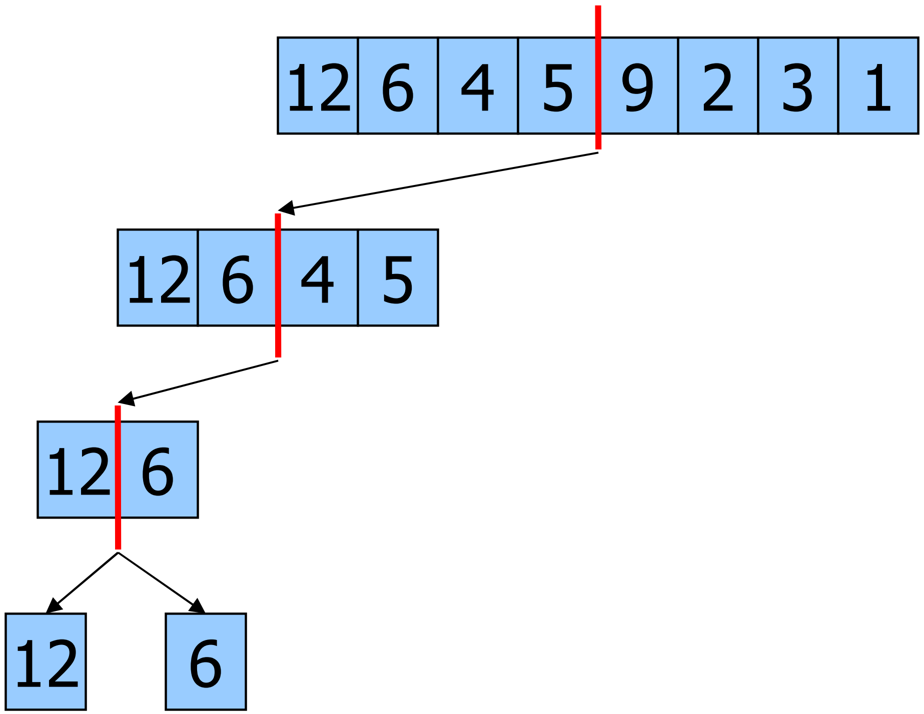
Example



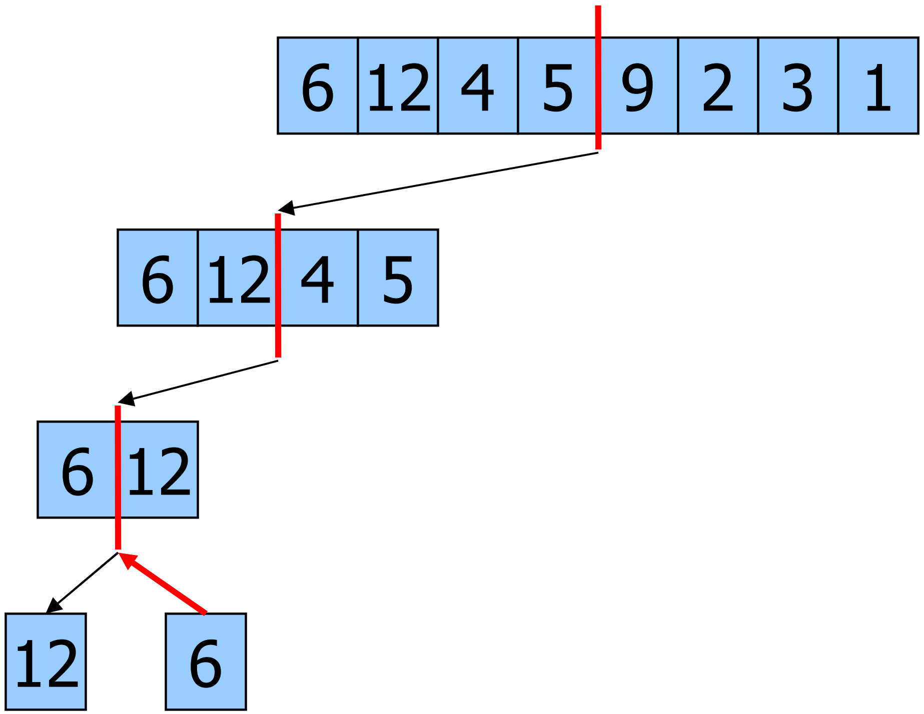
Example



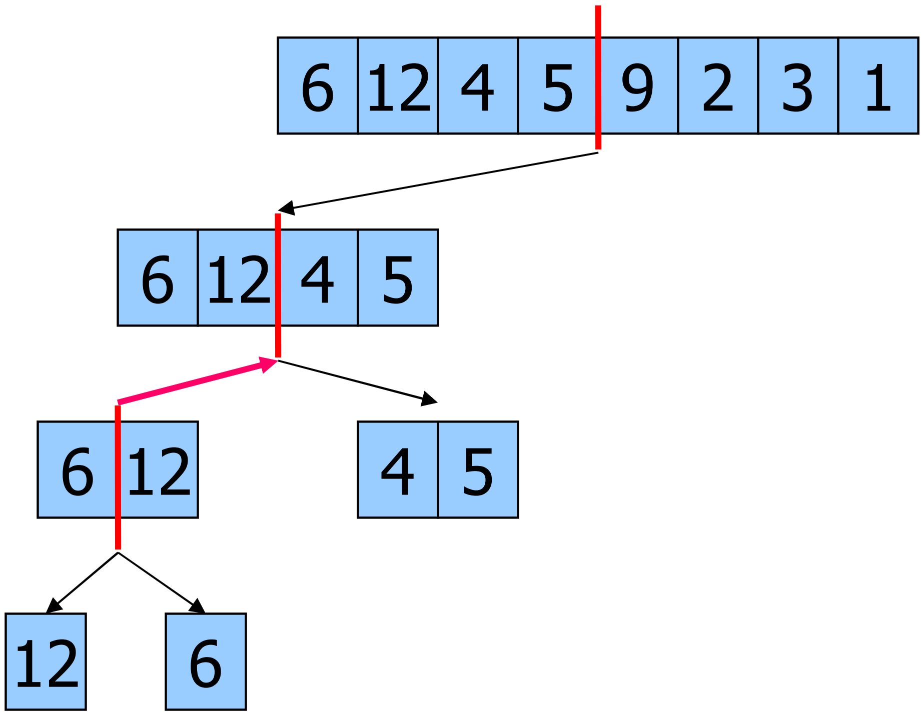
Example



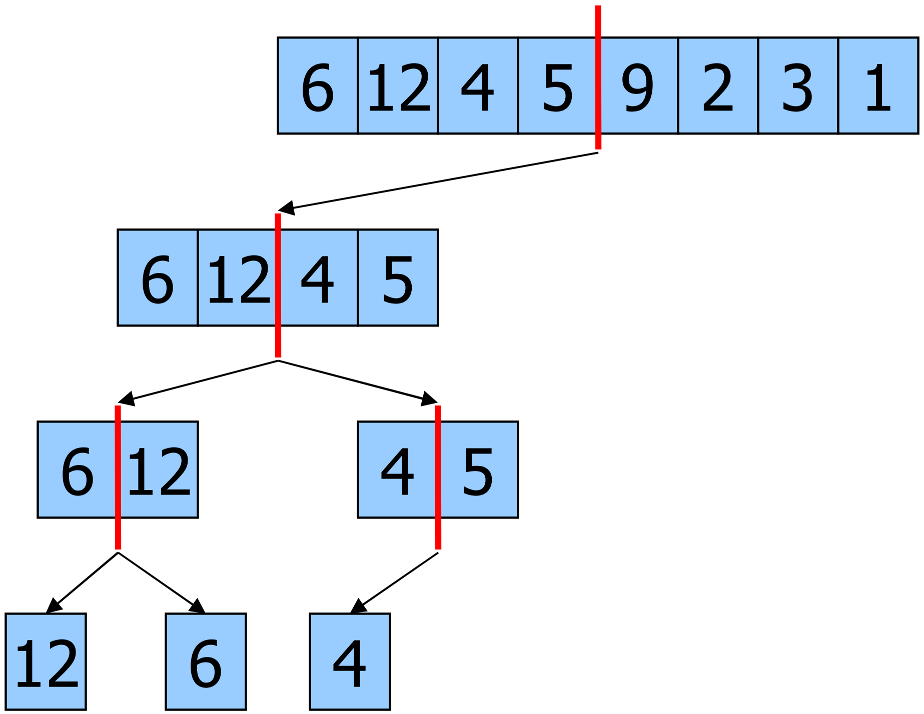
Example



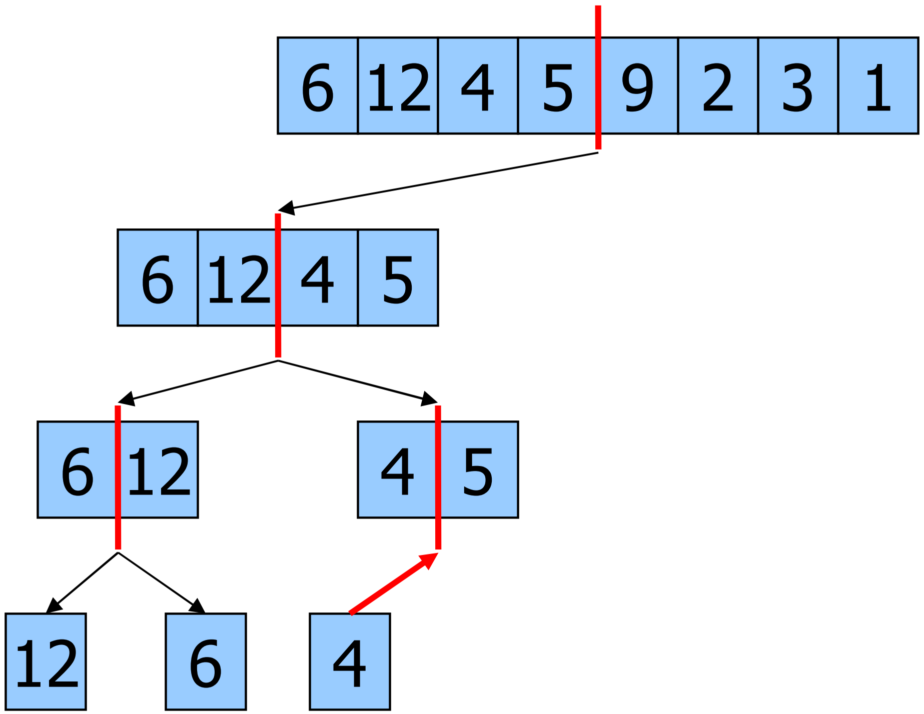
Example



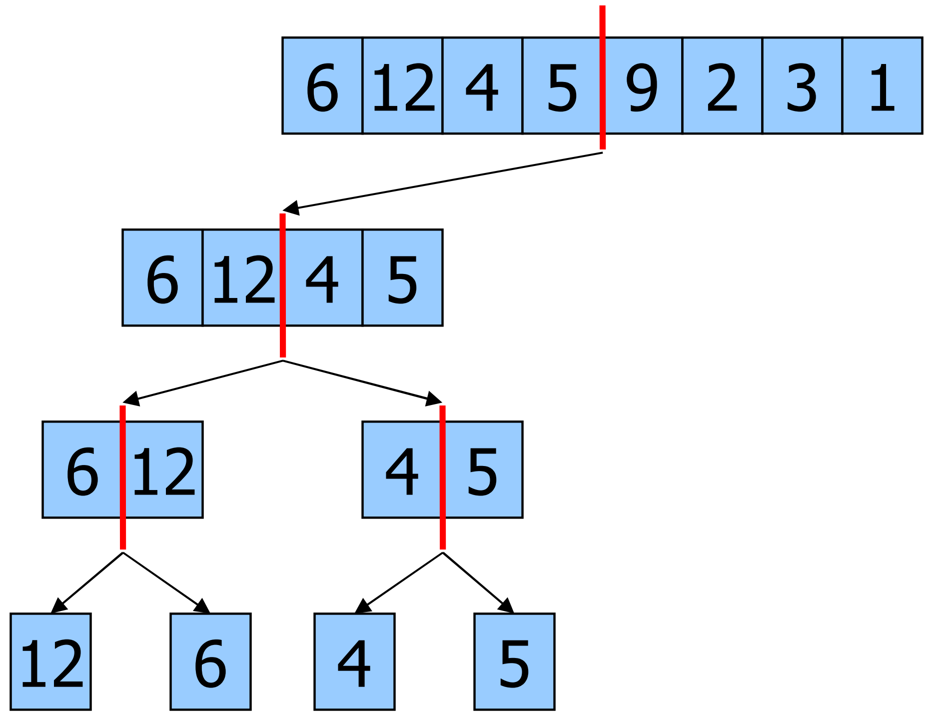
Example



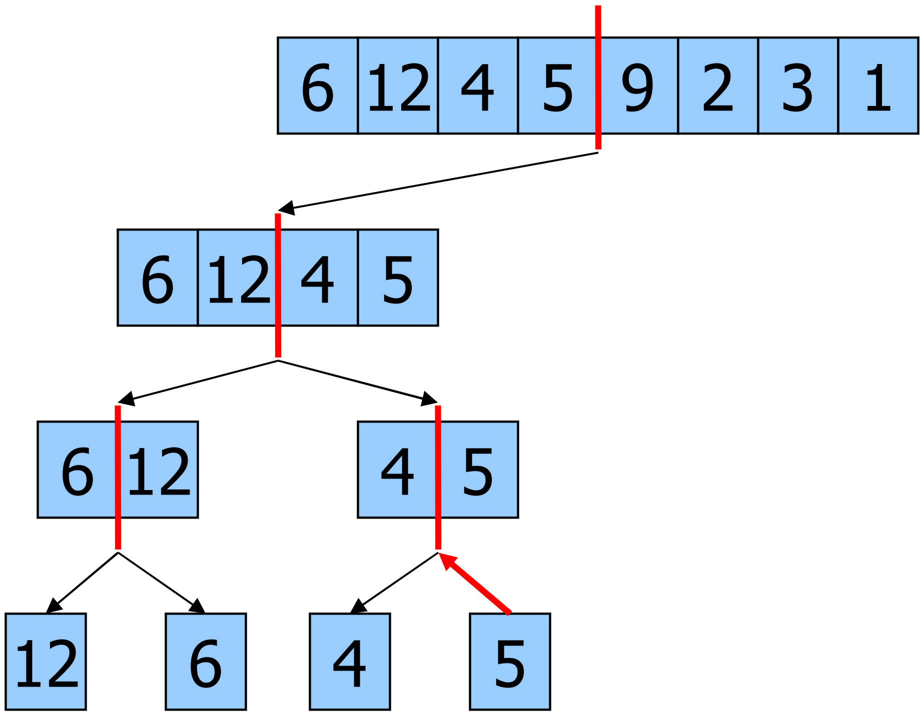
Example



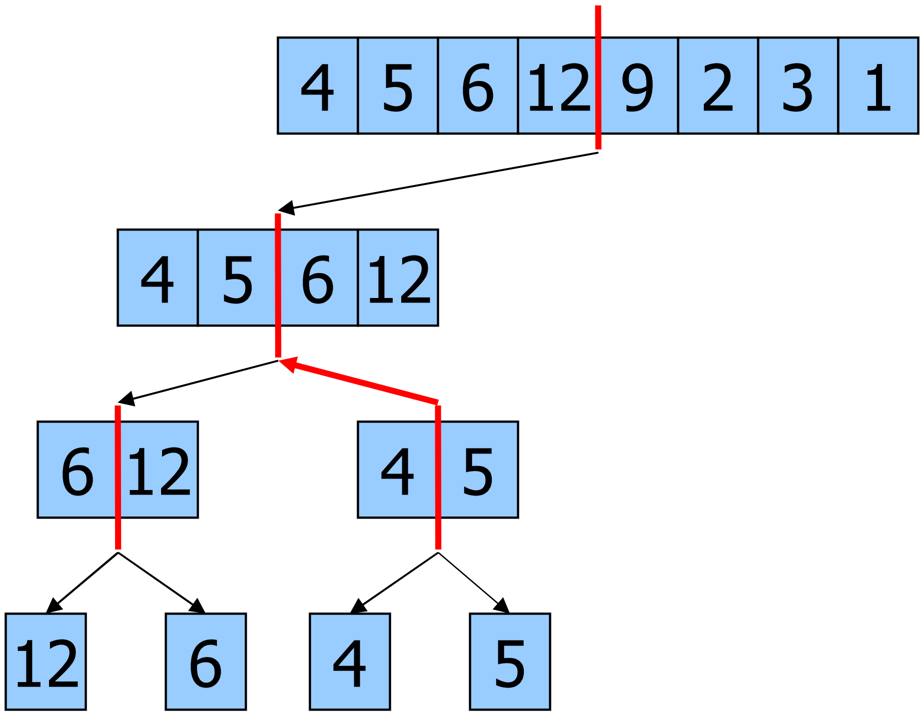
Example



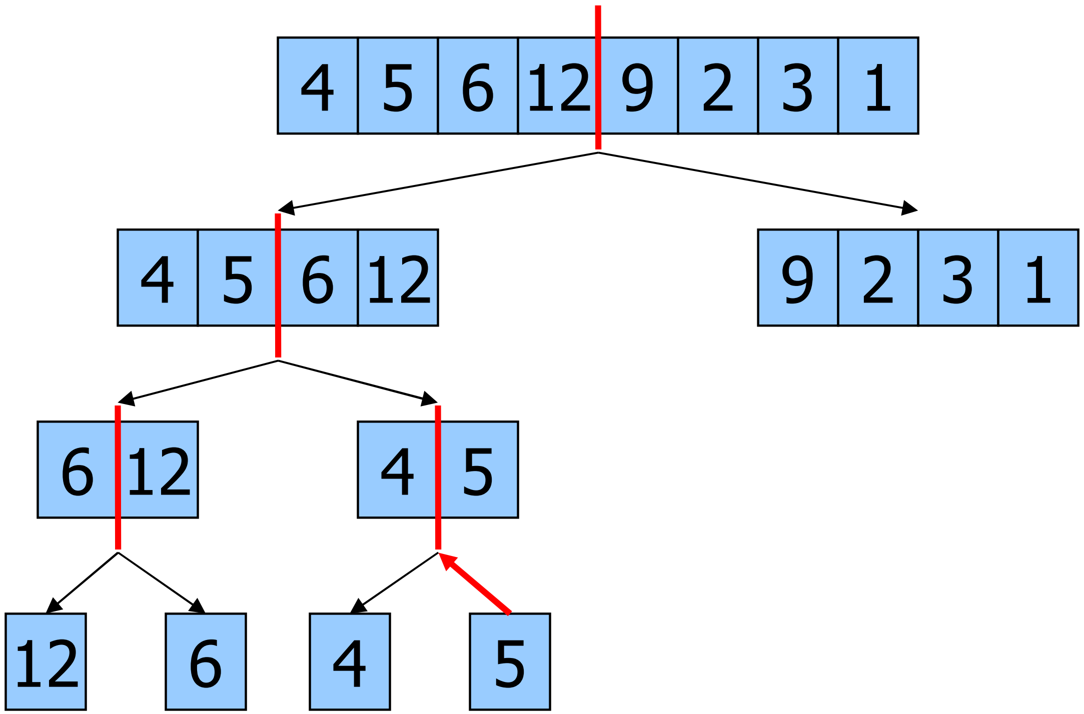
Example



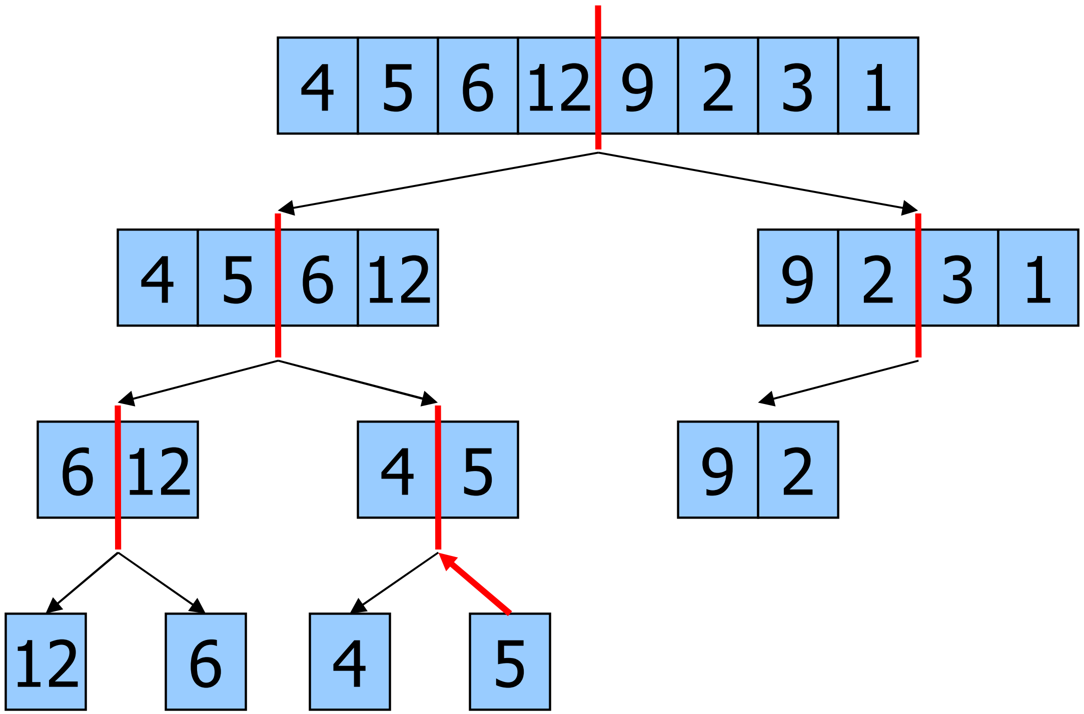
Example



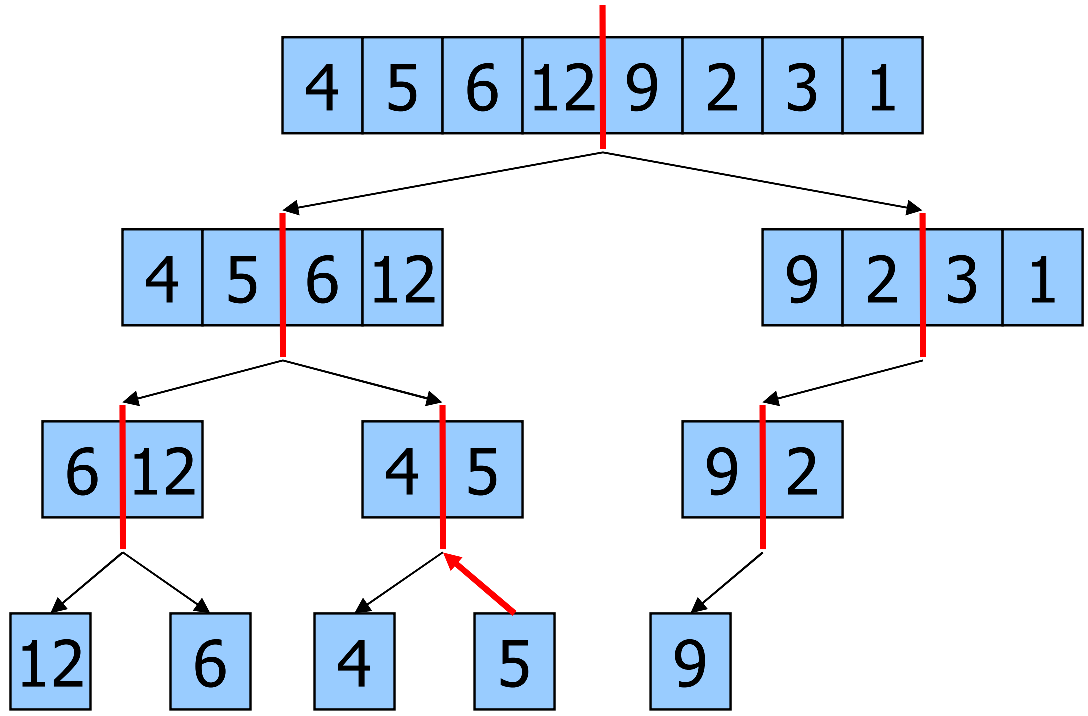
Example



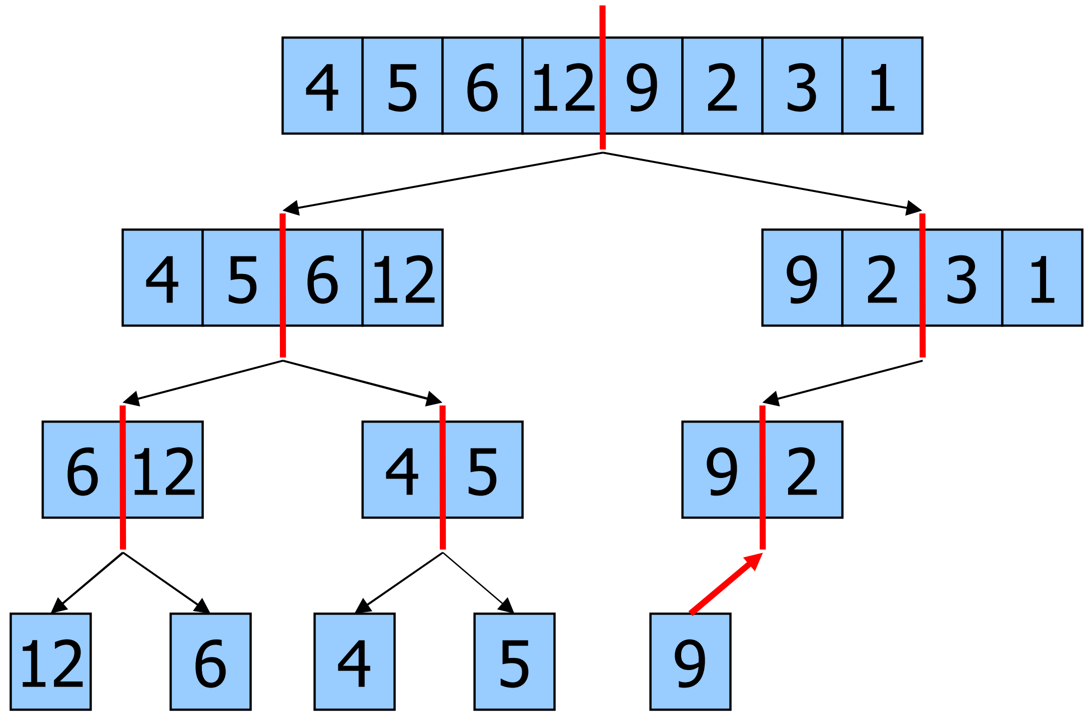
Example



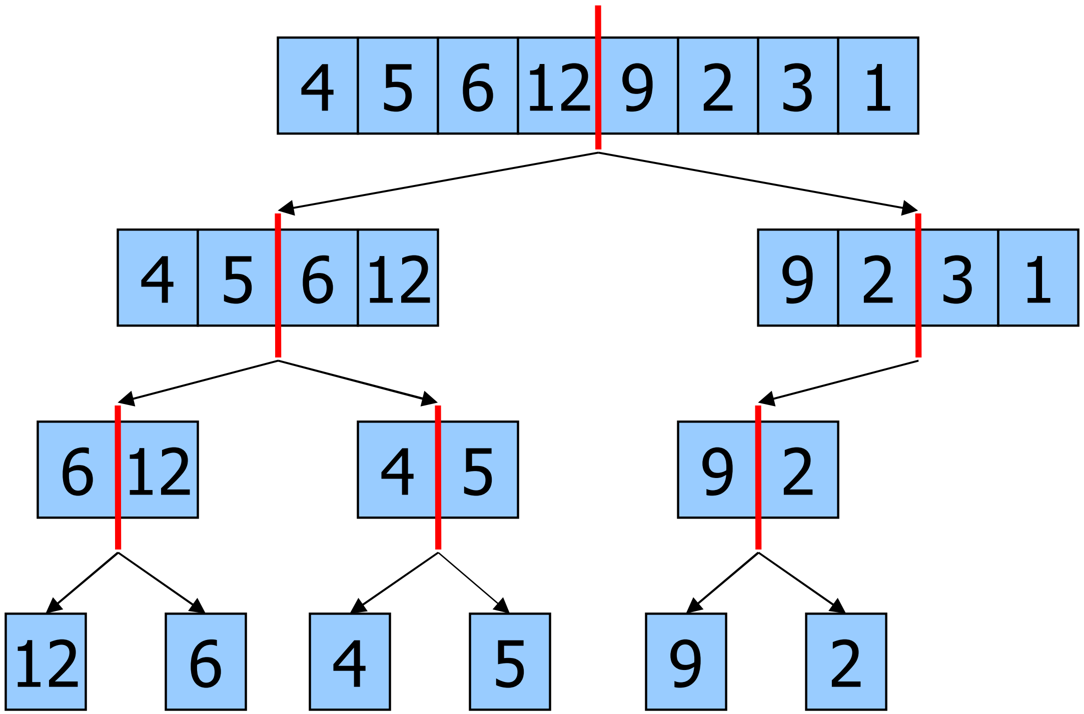
Example



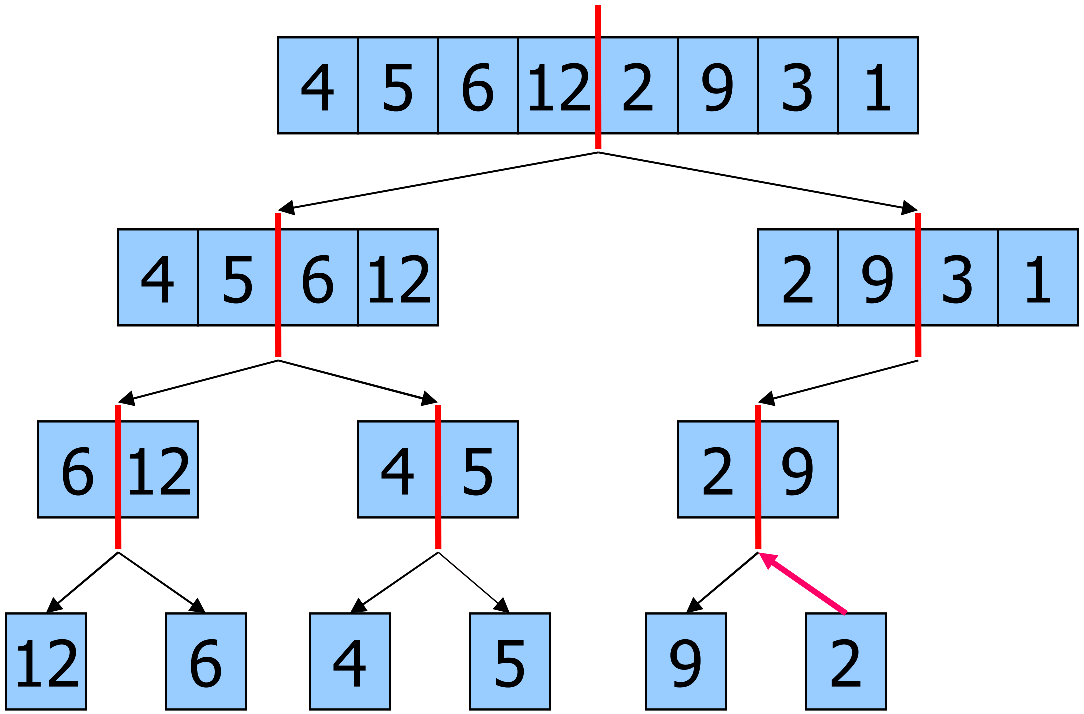
Example



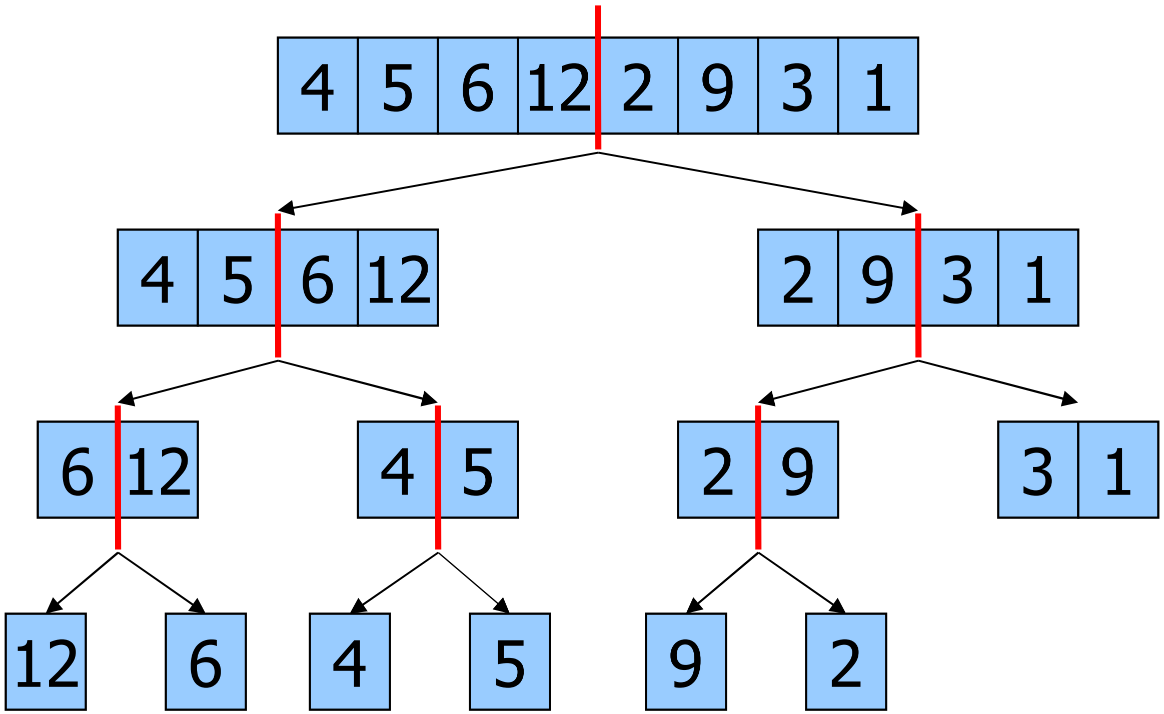
Example



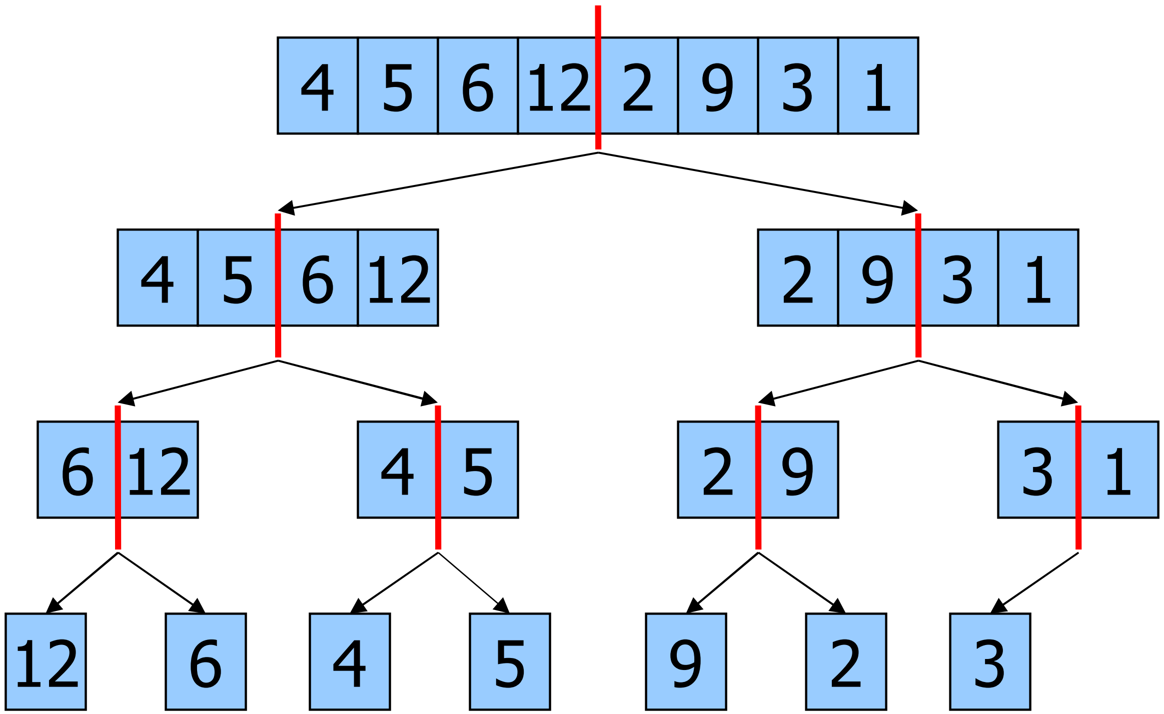
Example



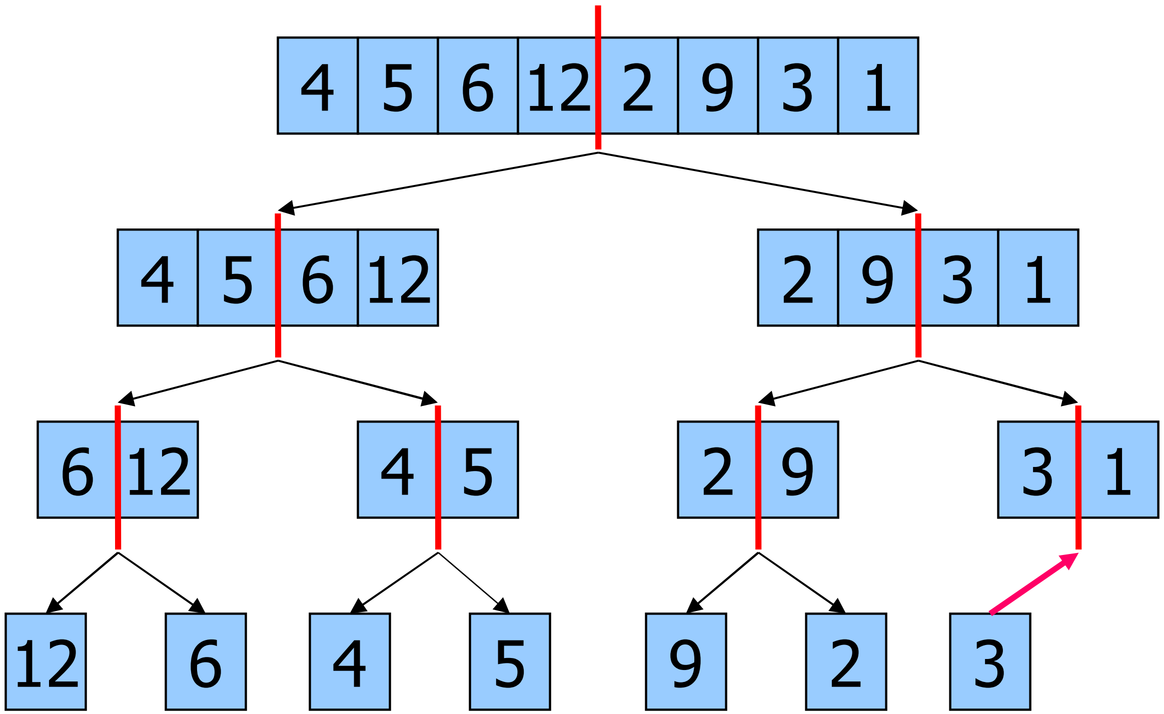
Example



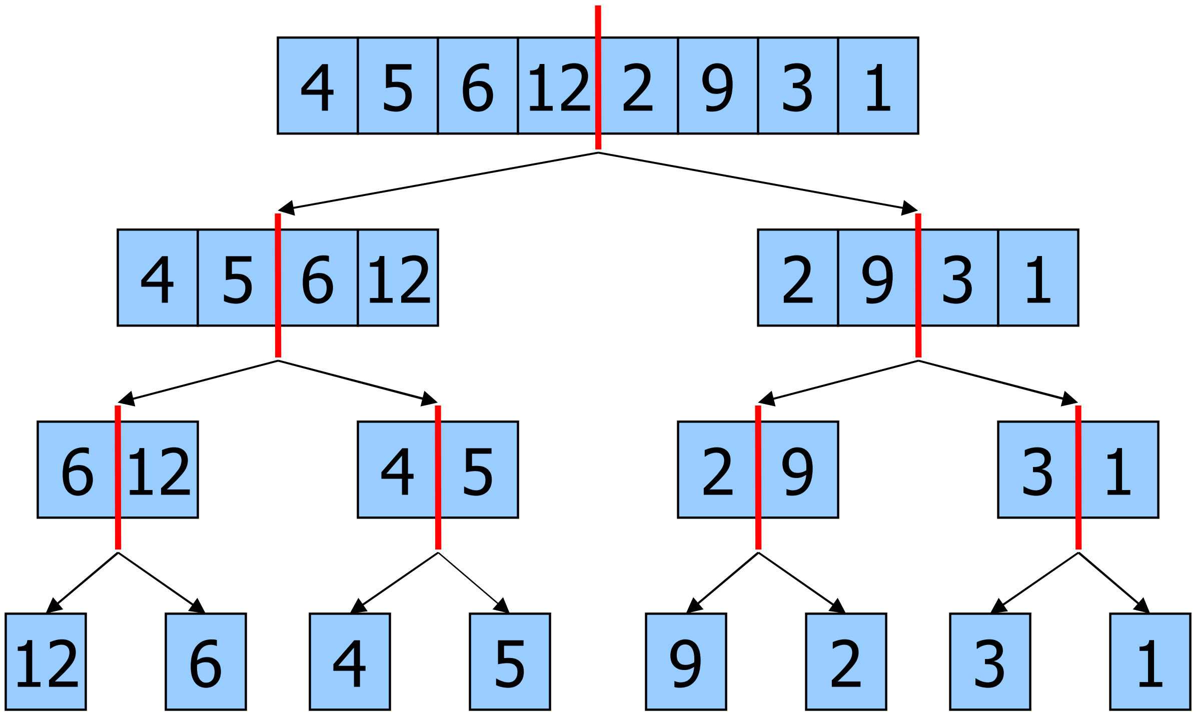
Example



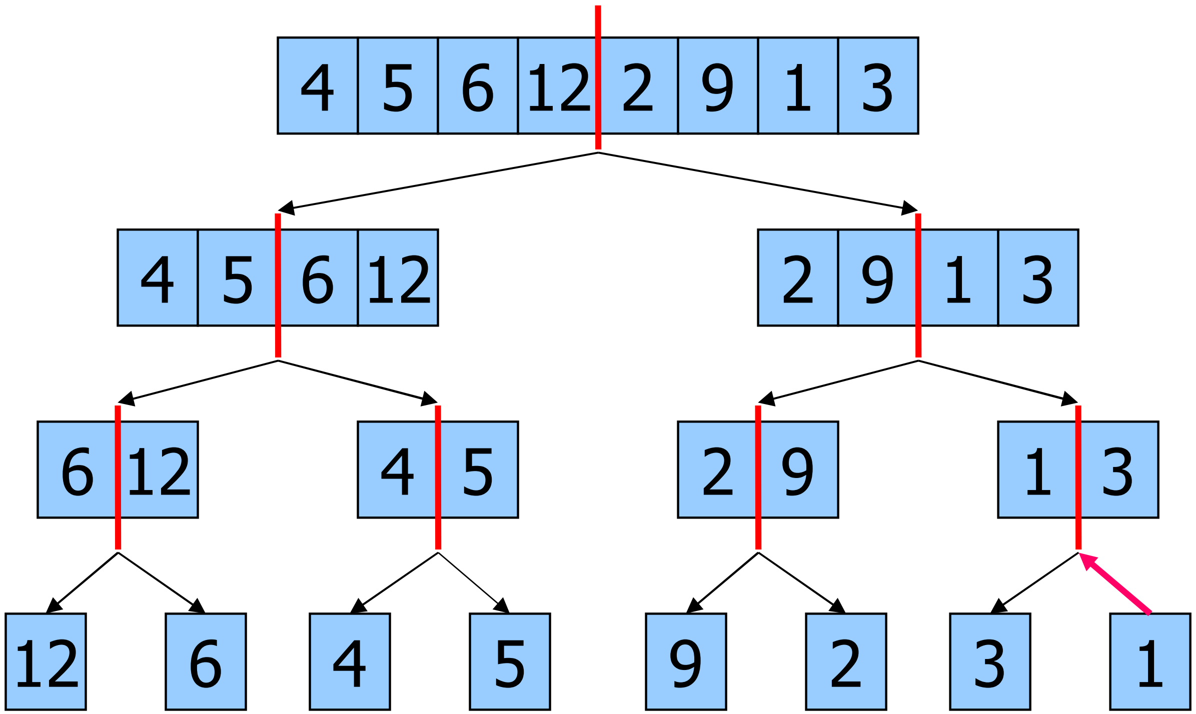
Example



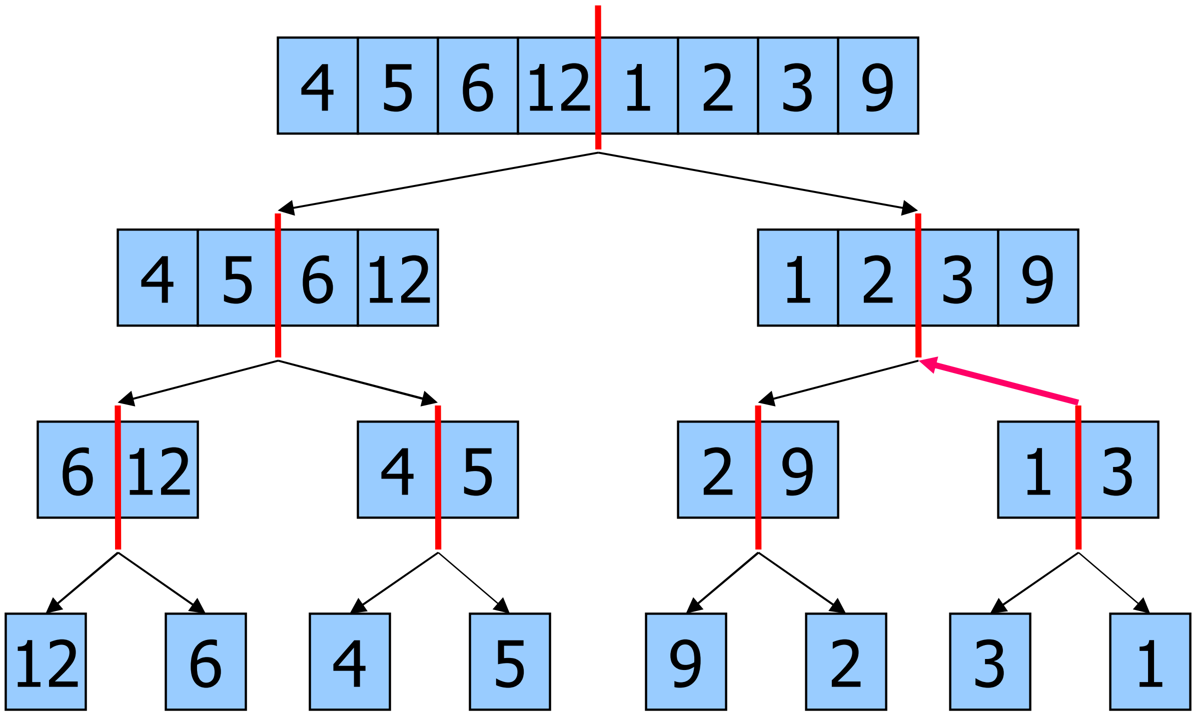
Example



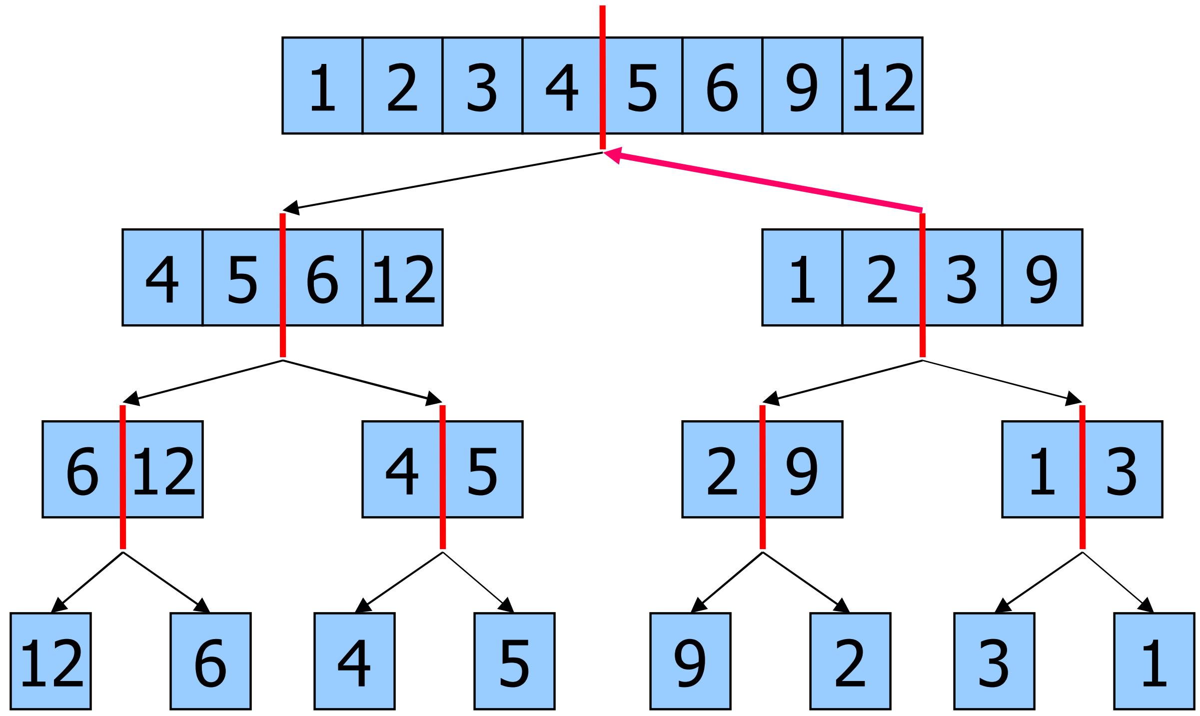
Example



Example



Example



Solution

Wrapper

Auxiliary array (check and free missing)

```
void merge_sort (int *A, int N) {
  int l=0, r=N-1;
  int *B = (int *)malloc(N*sizeof(int));
  merge_sort_r (A, B, l, r);
}
```

Boundaries

Recursive function call

```
void merge_sort_r (int *A, int *B, int l, int r){
  int c;
  if (r <= l)
    return;
  c = (l + r)/2
  merge_sort_r (A, B, l, c);
  merge_sort_r (A, B, c+1, r);
  merge (A, B, l, c, r);
}
```

Division

Termination

Recursive calls

Recombination

Solution

```
void merge (int *A, int *B,  int l, int c, int r) {
  int i, j, k;

  for (i=l, j=c+1, k=l; i<=c && j<=r; )
    if (A[i]<=A[j])
      B[k++] = A[i++];
    else
      B[k++] = A[j++];

  while (i<=c)
    B[k++]=A[i++];
  while (j<=r)
    B[k++]=A[j++];

  for (k=l; k<=r; k++)
    A[k] = B[k];

  return;
}
```

<= for stability

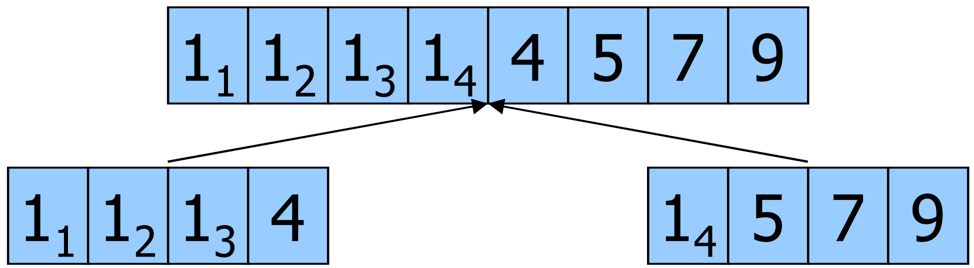
Compare and merge

Copy one of the tail

Copy array back

Features

- Not in place
  - It uses an auxiliary array
- Stable
  - Function merge takes keys from the left subarray in the case of duplicate values



Complexity analysis

- Assumption
  - $n = 2^k$
- Divide array in 2
  - $D(n)=\Theta(1)$
- Solve 2 subproblems of size n/2 each
  - $2T(n/2)$
- Termination
  - Simple test $\Theta(1)$
- Combine
  - Based on merge $C(n) = \Theta(n)$

Complexity analysis

- Recurrence equation
  - $T(n) = D(n) + a\,T(n/b) + C(n)$

$D(n)+C(n) = \Theta(n)$

- That is
  - $T(n) = n + 2T(n/2)$ $\quad n>1$
  - $T(1) = 1$ $\quad n=1$
- Resolution by unfolding
  - $T(n) = n + 2T(n/2)$
  - $T(n/2) = n/2 + 2T(n/4)$
  - $T(n/4) = n/4 + 2T(n/8)$
  - …

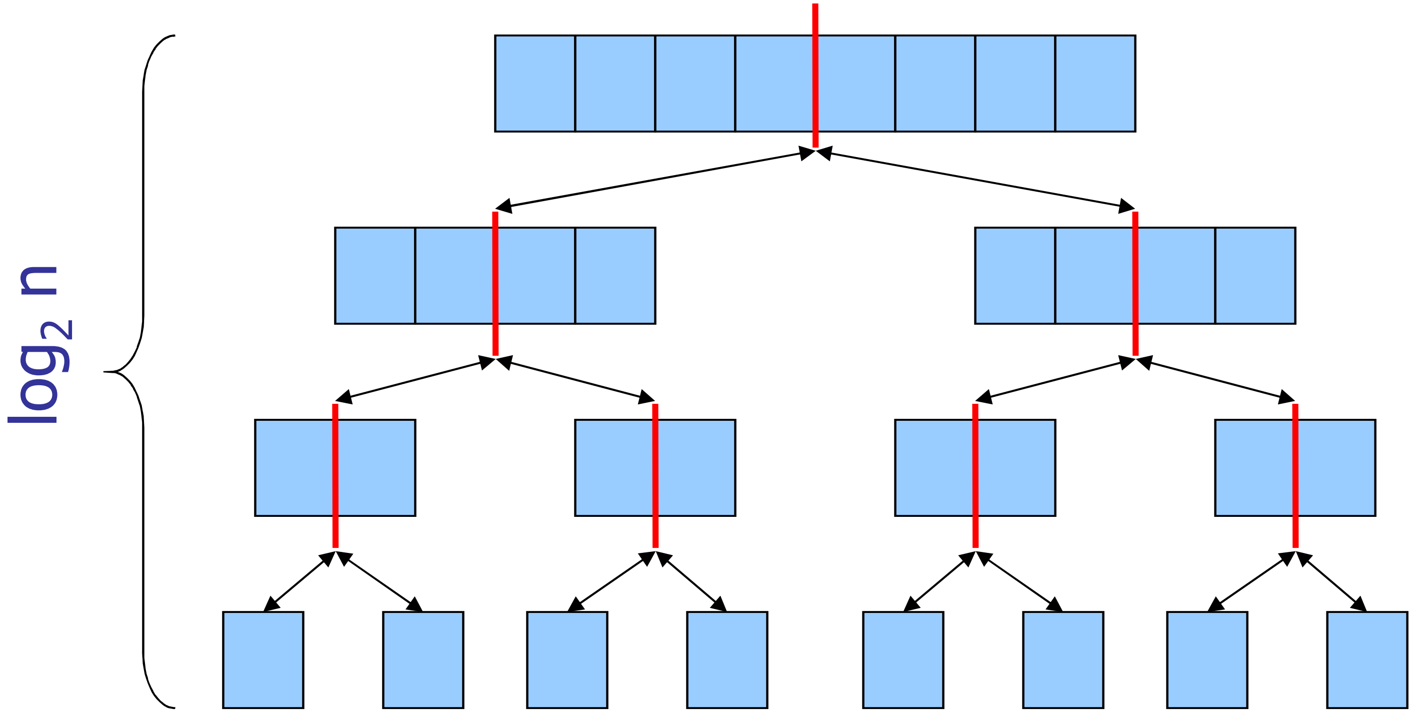
Complexity analysis

Termination condition
$n/2^i = 1$
$i = \log_2 n$

- Replacing in T(n)
  - $T(n) = n + n + n + n + \ldots$

    $= n \sum_{i=1}^{log_2 n} 1 = n \log n$
- Thus
  - $T(n) = O(n \log n)$

Intuitive analysis



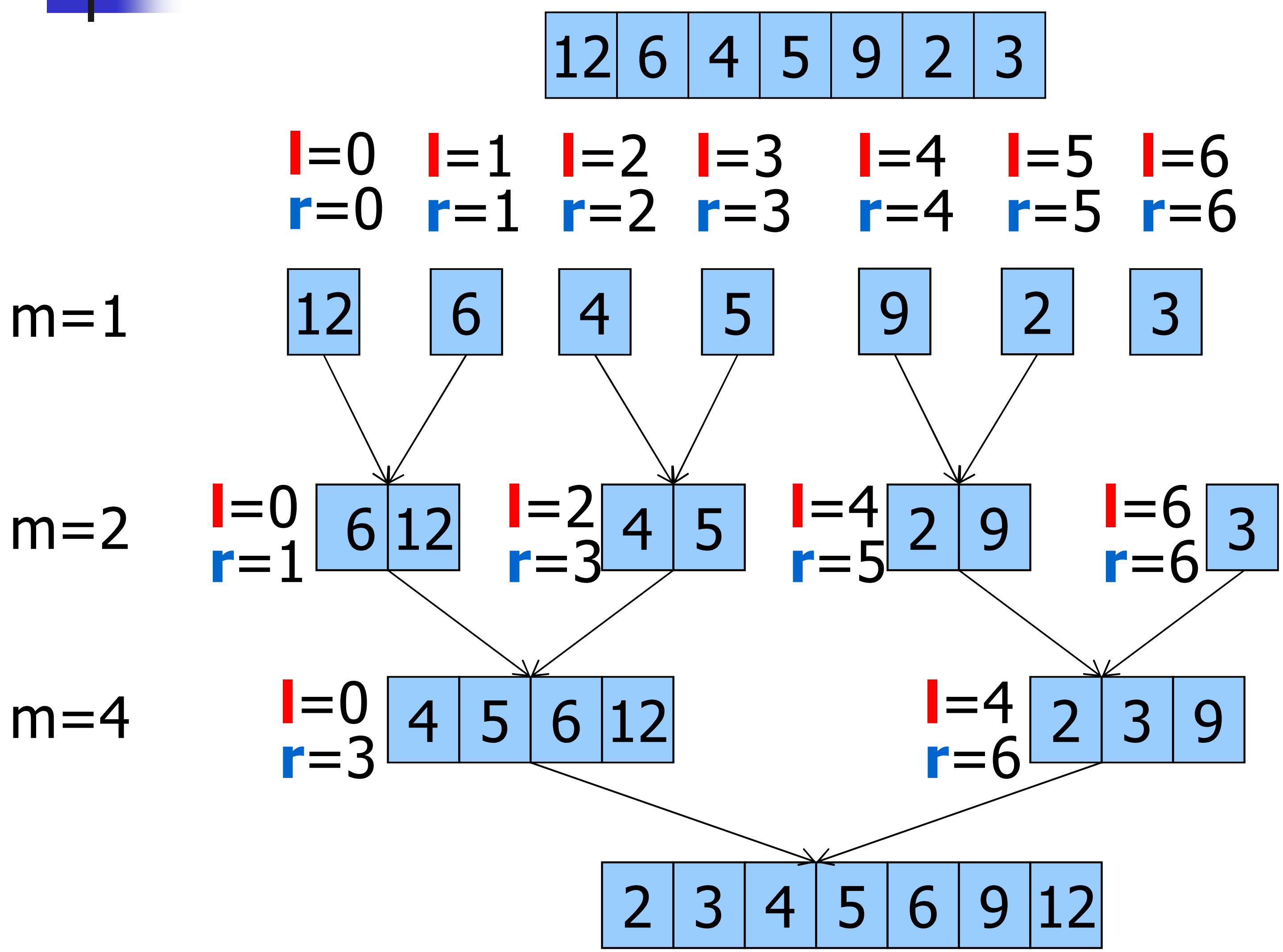
Recursion levels: $\log_2 n$ Operations at each level: n

Total operations: $n \log_2 n$

Bottom-up merge sort

- Non recursive version
- Starting from subarrays of length `1` (thus sorted), apply `Merge` to obtain at each step sorted arrays whose length is twice as big
- Termination
  - The length of the sorted array equals the length of the initial array

Example



12 6 4 5 9 2 3
l=0 r=0
l=1 r=1
l=2 r=2
l=3 r=3
l=4 r=4
l=5 r=5
l=6 r=6
m=1
12
6
4
5
9
2
3
m=2
l=0 r=1
6 12
l=2 r=3
4 5
l=4 r=5
2 9
l=6 r=6
3
m=4
l=0 r=3
4 5 6 12
l=4 r=6
2 3 9
2 3 4 5 6 9 12

Solution

```
int min(int i, int j) {
   if (i < j)
      return i;
   else
      return j;
}
void bottom_up_merge_sort (int *A, int l, int r){
  int i, m, l=0, r=N-1;
  int *B = (int *)malloc(N*sizeof(int));
  for (m = 1; m <= r-l; m = m + m)
    for (i = l; i <= r-m; i += m + m)
      merge (A, B, i, i+m-1, min(i+m+m-1,r));
}
```

B: Auxiliary array (check and free missing)

Boundaries

Consider the pairs of sorted subarrays of size m

Merge them

Quicksort (Hoare, 1961)

- In the division step, the array A[l..r] is partitioned in 2 subarrays L and R
  - Given a pivot element x
  - L, i.e., A[l..q-1], contains all elements $< x$
  - R, i.e., A[q+1..r], contains all elements $> x$
  - x is in the right place
  - Division doesn't necessarily halve the array

Quicksort (Hoare, 1961)

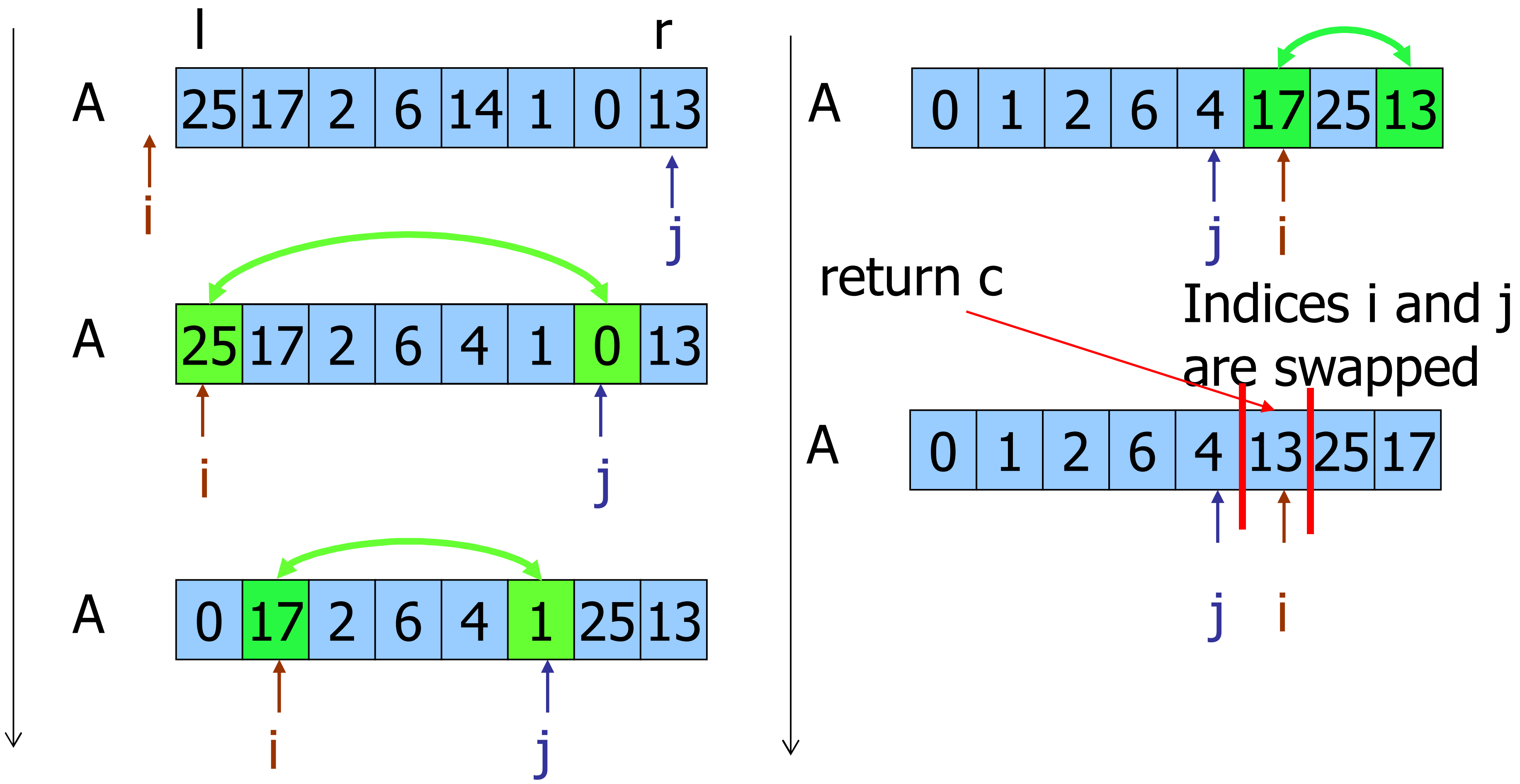
- Recursion
  - Quicksort on subarray L, i.e., A[l..q-1]
  - Quicksort on subarray R, i.e., A[q+1..r]
  - Termination condition: if the array has 1 element it is sorted
- Ricombination
  - None

Partition

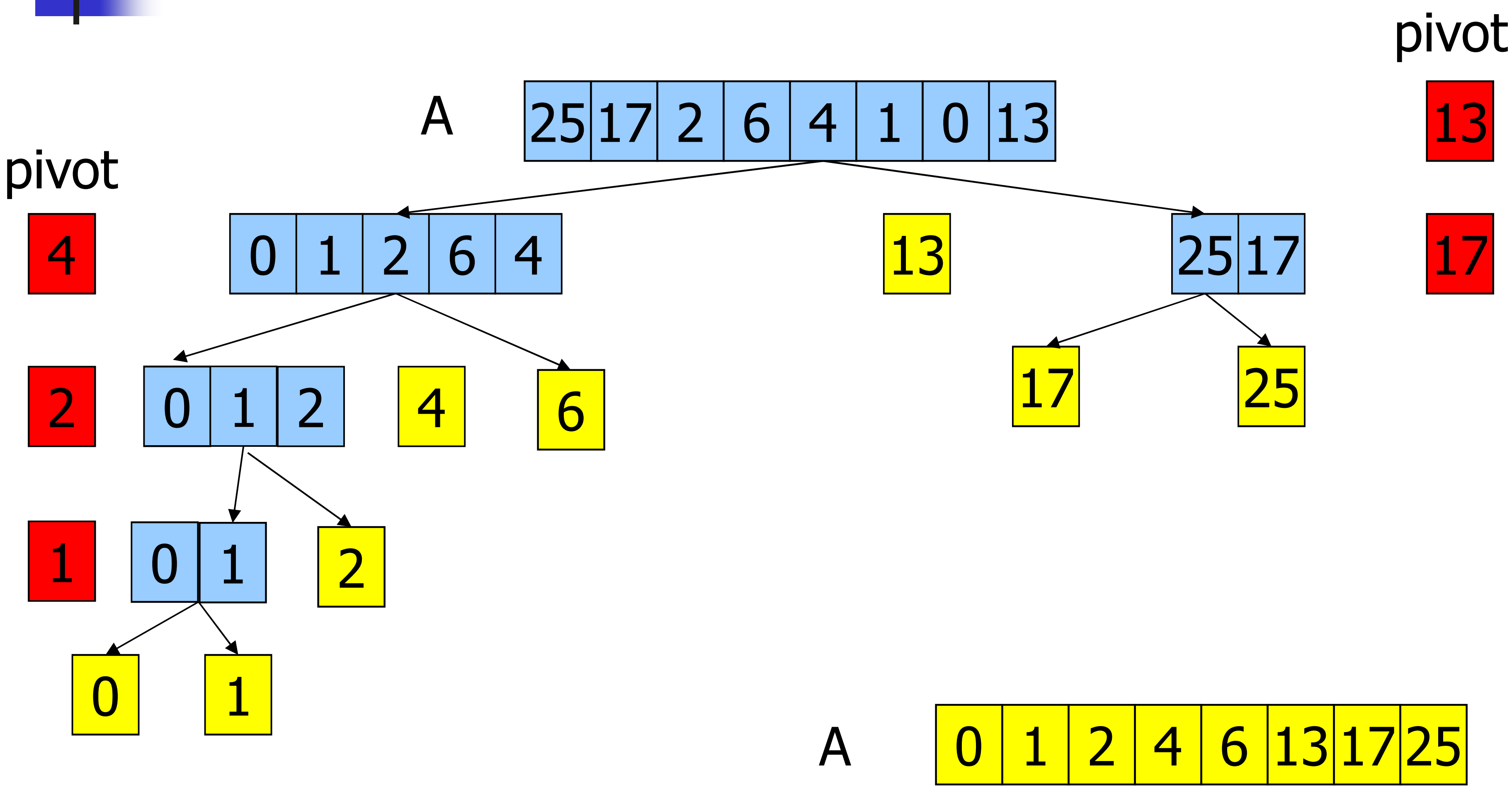
- Pivot pivot = A[r]
- Find A[i] and A[j], i.e., 2 elements not in the proper partition
  - Ascending loop on i until we find an element greater than the pivot x
  - Descending loop on j until we find an element less than the pivot x
- Swap A[i] and A[j]
- Repeat until i < j
- At the end swap A[i] and pivot x
- Return q = i

Partition example

pivot 13



Quicksort example



Solution

Wrapper

```
void quick_sort(int *A, int N) {
  int l, r;
  l = 0;
  r = N-1;
  quick_sort_r (A, l, r);
}
```

Boundaries

Recursive function call

```
void quick_sort_r (int *A, int l, int r){
  int c;
  if (r <= l)
    return;
  c = partition (A, l, r);
  quick_sort_r (A, l, c-1);
  quick_sort_r (A, c+1, r);
  return;
}
```

Termination

Division

Recursive calls

Element c is not moved any more

Solution

```
int partition (int *A, int l, int r ){
  int i, j, pivot;

  i = l-1;
  j = r;
  pivot = A[r];
  while (i<j) {
    while(A[++i]<pivot);
    while(j>l && A[--j]>=pivot);
    if (i < j)
      swap(A, i, j);
  }

  swap (A, i, r);
  return i;
}
```

Pivot values are moved in the right sub-array; worst case: stop on pivot

Pivot values stay in the right sub-array; worst case: stop on element l

Solution

```
void swap (int *v, int n1, int n2) {
   int temp;

   temp  = v[n1];
   v[n1] = v[n2];
   v[n2] = temp;

   return;
}
```

Example: Inverse ordered array

Initial array	:	9 8 7 6 5 4 3 2 1 0
pivot: 0 -	array:	0 8 7 6 5 4 3 2 1 9
pivot: 9 -	array:	0 8 7 6 5 4 3 2 1 9
pivot: 1 -	array:	0 1 7 6 5 4 3 2 8 9
pivot: 8 -	array:	0 1 7 6 5 4 3 2 8 9
pivot: 2 -	array:	0 1 2 6 5 4 3 7 8 9
pivot: 7 -	array:	0 1 2 6 5 4 3 7 8 9
pivot: 3 -	array:	0 1 2 3 5 4 6 7 8 9
pivot: 6 -	array:	0 1 2 3 5 4 6 7 8 9
pivot: 4 -	array:	0 1 2 3 4 5 6 7 8 9

Example: Ordered array

Initial array		:	0	1	2	3	4	5	6	7	8	9
pivot:	9	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	8	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	7	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	6	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	5	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	4	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	3	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	2	- array:	0	1	2	3	4	5	6	7	8	9
pivot:	1	- array:	0	1	2	3	4	5	6	7	8	9

Example: Scrambled array

Initial array :	1	8	0	2	3	9	4	6	5	7
pivot: 7 - array:	1	5	0	2	3	6	4	7	8	9
pivot: 4 - array:	1	3	0	2	4	6	5	7	8	9
pivot: 2 - array:	1	0	2	3	4	6	5	7	8	9
pivot: 0 - array:	0	1	2	3	4	6	5	7	8	9
pivot: 5 - array:	0	1	2	3	4	5	6	7	8	9
pivot: 9 - array:	0	1	2	3	4	5	6	7	8	9

Features

- In place
- Not stable
  - Partition may swap "far away" elements
  - Then occurrence of a duplicate key moves to the left of a previous occurrence of the same key

Analysis

- Efficiency depends on the partition balance
- At each step **partition** returns
  - In the worst case a subarray with n-1 elements and a subarray with 1 element
  - In the best case 2 subarrays with n/2 elements
  - In the average case 2 subarrays of different sizes
- Balancing depending on the choice of the pivot

Worst case

- Worst case
  - Pivot = minimum or maximum (array already sorted)
- Recusion equation
  - $T(n) = T(n-1) + n \qquad n>=2$
  - $T(1) = 1 \qquad n = 1$
- Solution
  - $T(n) = O(n^2)$

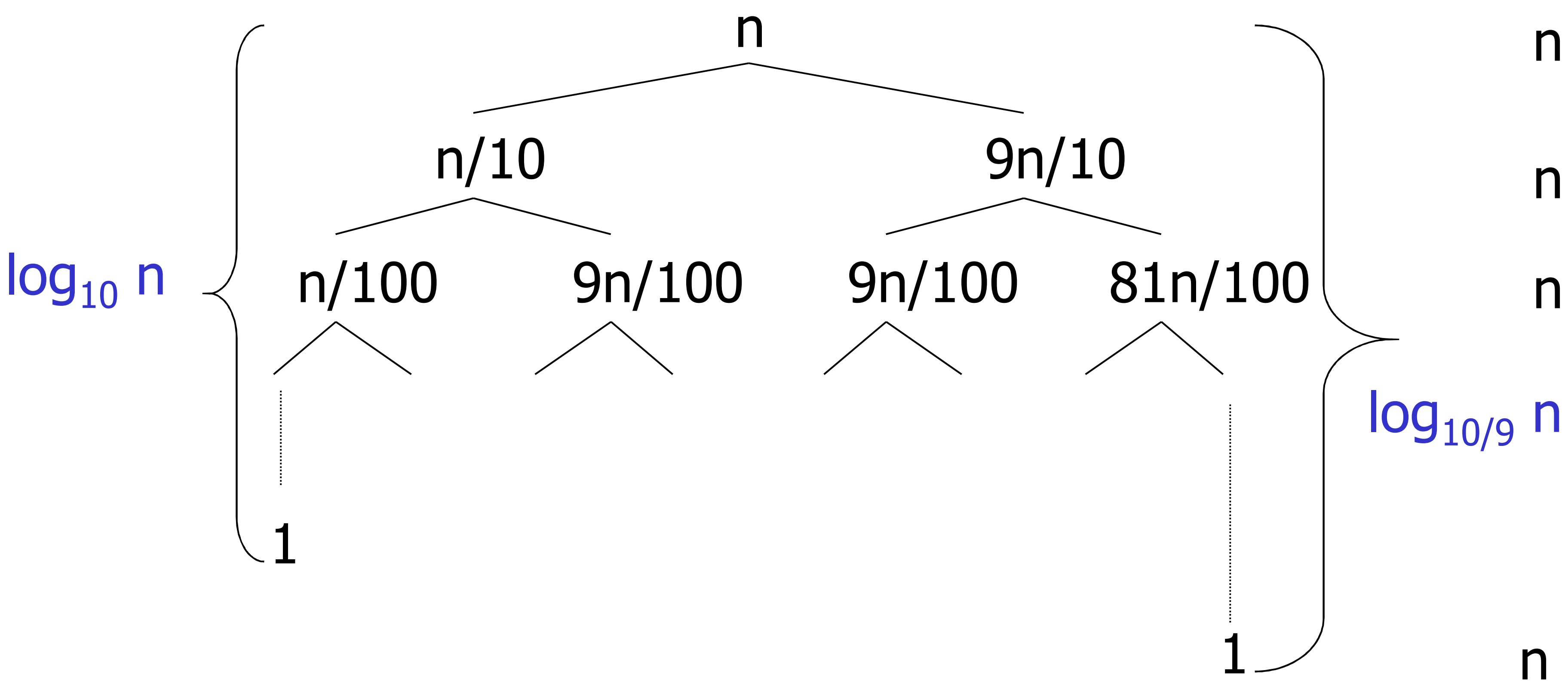
Best case

- Recursion equation
  - $T(n) = 2T(n/2) + n$ $\quad n>=2$
  - $T(1) = 1$ $\quad n = 1$
- Solution
  - $T(n) = O(n \lg n)$

Average case

- Provided we are not in the worst case, though partitions may be strongly unbalanced
  - Average case = best case
- Example
  - At each step **partition** generates 2 partitions
  - Let us suppose the first one has $(9/10 \cdot n)$ elements and the second one $(n/10)$ elements

Average case



$T(n) = O(n \lg n)$

Pivot selection

- Random element
  - Generate a random number i with $p \le i \le r$, then swap A[r] and A[i], use A[r] as pivot
- Middle element
  - $x \leftarrow A[(p+r)/2]$
- Select average between min and max
- Select median of 3 elements chosen randomly in array
- …

Summary on sorting algorithm

Algorithm	In place	Stable	Worst-Case
Bubble sort	Yes	Yes	$O(n^2)$
Selection sort	Yes	No	$O(n^2)$
Insertion sort	Yes	Yes	$O(n^2)$
Shellsort	Yes	No	depends
Mergesort	No	Yes	$O(n \cdot \log n)$
Quicksort	Yes	No	$O(n^2)$
Counting sort	No	Yes	$O(n)$