Recursion

Sorting

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

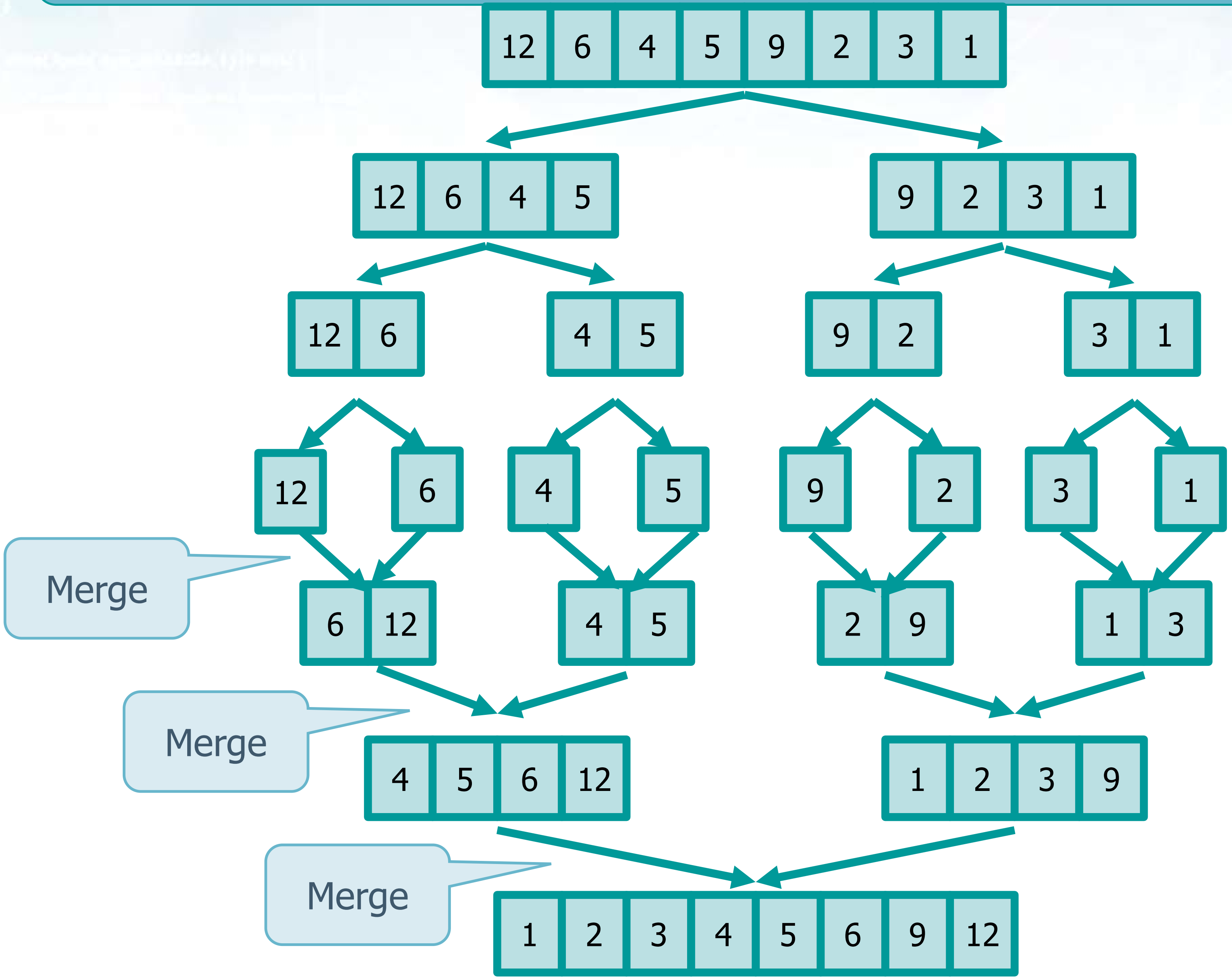
Merge sort

- Proposed by Von Neumann in 1949
- Division
  - Partition the array into 2 subarrays L and R with respect to the array's middle element
- 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

Divide does not reorder anything

Combine performes the sorting

Example


12 6 4 5 9 2 3 1
12 6 4 5
9 2 3 1
12 6
4 5
9 2
3 1
12
6
4
5
9
2
3
1
Merge
6 12
4 5
2 9
1 3
Merge
4 5 6 12
1 2 3 9
Merge
1 2 3 4 5 6 9 12

Merge

- Merge sort is based on **merge**
  - Given two already ordered arrays v_1 and v_2
  - Generate e unique ordered array v_3
  - Example

```
v1 =  3 6 9 30 40
v2 = -1 6 7  8 10

v3 = -1 3 6  6  7 8 9 10 30 40
```

Merge

```
i1 = i2 = i3 = 0;

while (i1<N && i2<N) {
  if (v1[i1] < v2[i2]) {
    v3[i3++] = v1[i1++];
  } else {
    v3[i3++] = v2[i2++];
  }
}

while (i1 < N) {
  v3[i3++] = v1[i1++];
}

while (i2 < N) {
  v3[i3++] = v2[i2++];
}
```

Merge body of v1 and body of v2 (both of size N)

Merge tail of v1, if it exists

Merge tail of v2, if it exists

Merge

- Merging two arrays has a **linear** cost the size of the final array
  - $T(n) = O(n)$
- In merge sort the merge phase
  - Operates on two partitions of the same array (A) instead of working on arrays v_1 and v_2
  - Generates the resulting array v_3 in the original array (A)
  - Uses a temporary array (B)

Solution

Wrapper

B is an auxiliary array (check and free are missing)

Recursion

```
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);
}

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);
  return;
}
```

Left recursion

Right recursion

Combine (merge on 2 partitions of the same array)

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;
}
```

Compare and merge

Use <= to make the sorting stable

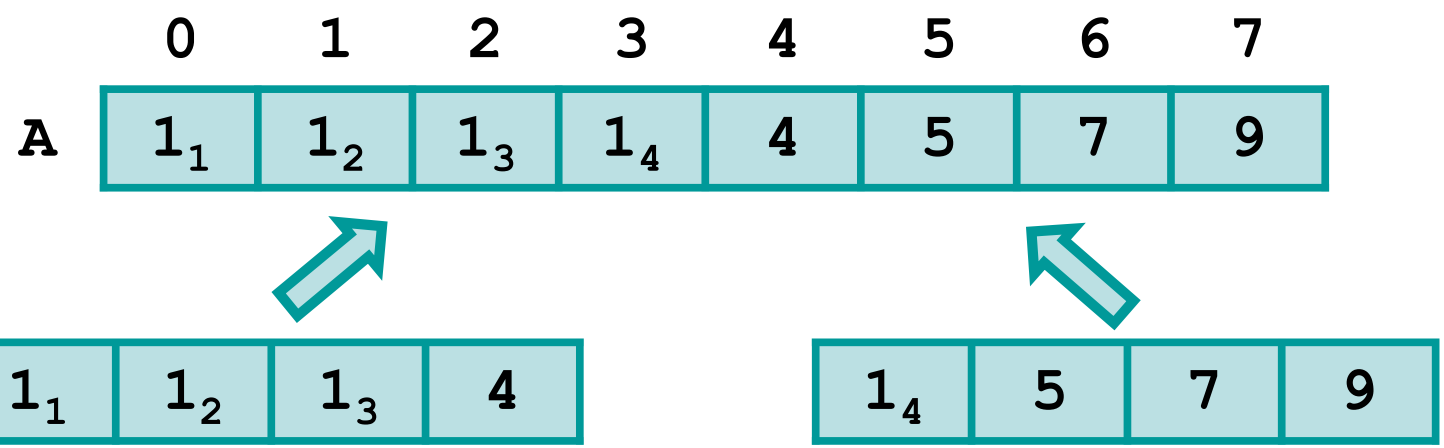
Copy the first tail

Copy the second tail

Copy the 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

- Analitic analysis
  - Assumption
    - $n = 2^k$
- Divide and conquer problem with
  - Number of subproblems
    - $a = 2$
  - Reduction factor
    - $b = n/n' = 2$
  - Division cost
    - $D(n) = \Theta(1)$

```
void merge_sort_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);
}
```

Complexity Analysis

- Recombination cost
  - Based on merge
  - $C(n) = \Theta(n)$
- Termination
  - Simple test $\Theta(1)$

Recurrence equation

- $T(n) = D(n) + a \cdot T(n/b) + C(n)$

```
void merge_sort_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);
}
```

Complexity Analysis

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

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

Complexity Analysis

- Replacing in T(n)
  - $T(n) = n + 2 \cdot T(n/2)$

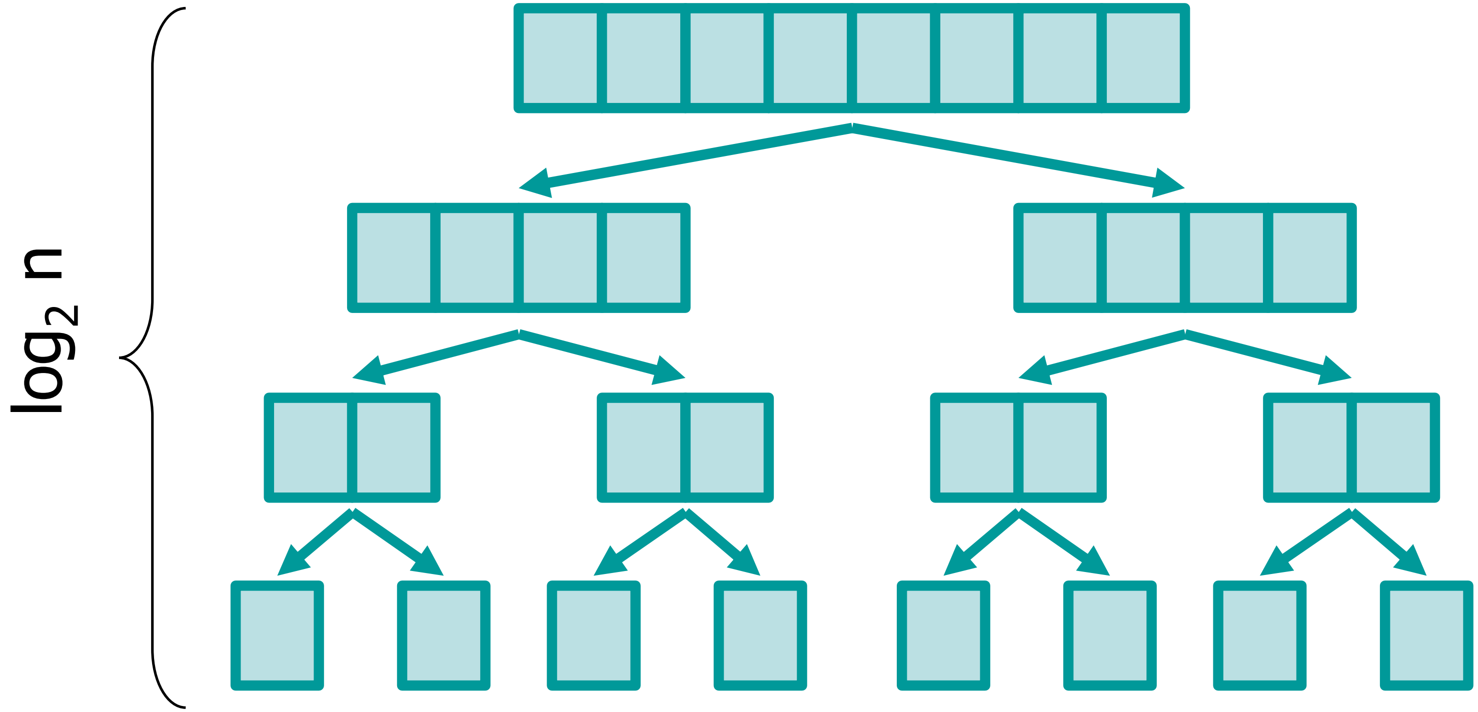
$$= n + n + 4 \cdot T(n/4)$$
$$= n + n + n + 8 \cdot T(n/8)$$
$$= \ldots$$
$$= n + n + n + n + \ldots$$
$$= n \cdot \sum_{i=1}^{\log n} 1$$
$$= n \cdot \log n$$
$$= O(n \cdot \log n)$$

$T(n) = n + 2 \cdot T(n/2)$
$T(n/2) = n/2 + 2 \cdot T(n/4)$
$T(n/4) = n/4 + 2 \cdot T(n/8)$

Termination condition
n/2i = 1
i = log2n

Complexity Analysis

- Intuitive analysis



Recursion levels: $\log_2 n$

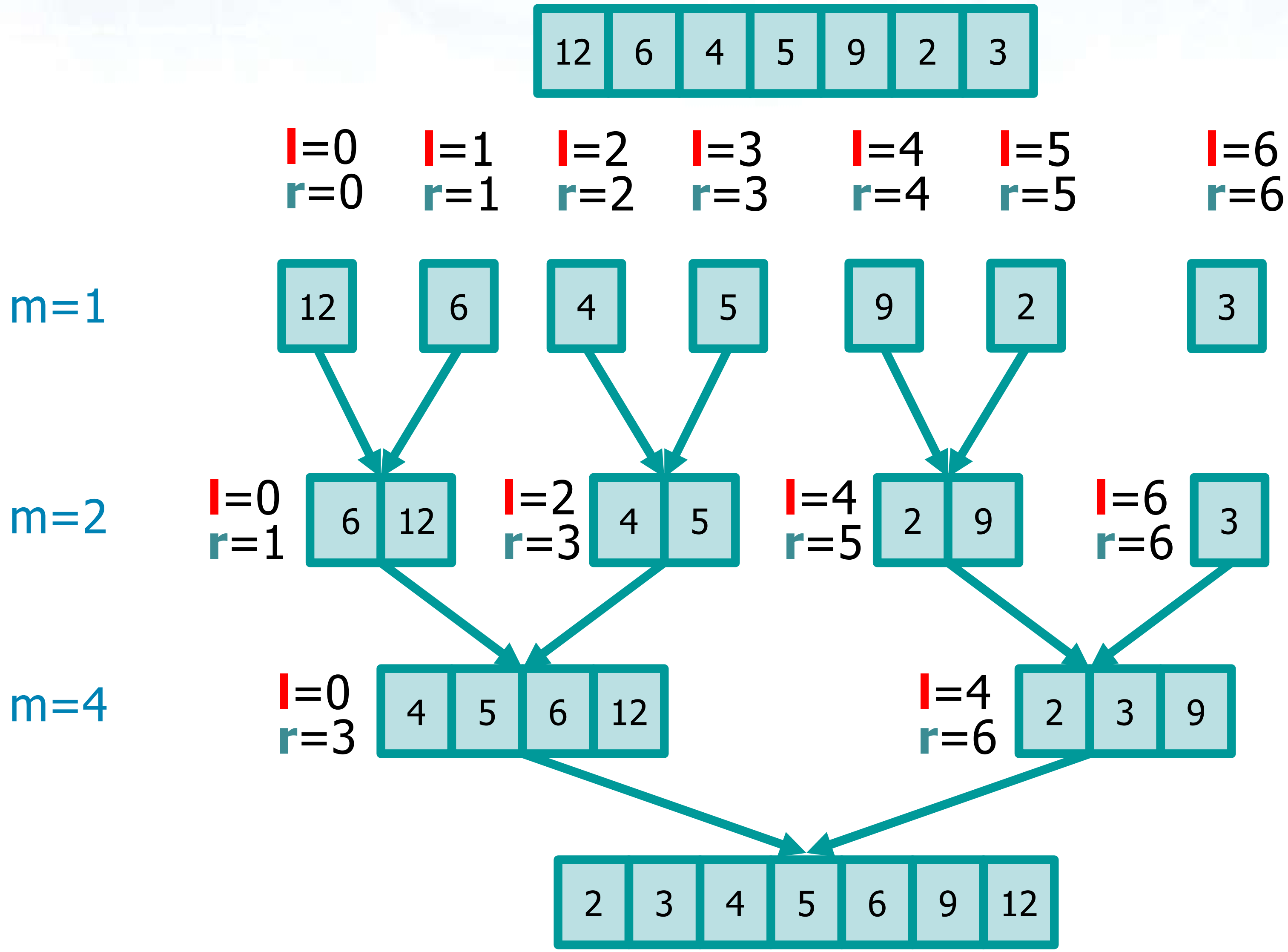
Operations at each level: n

Total operations: $n \cdot \log_2 n$

Bottom-up merge sort

- Non recursive version of merge sort
- Core idea
  - It starts from subarrays of length 1 (thus sorted)
  - It applies 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 is an auxiliary array (check and free are missing)

l and r are the array boundaries

Consider the pairs of sorted subarrays of size m

Merge subarrays

Tim Sort

- Proposed by Peters in 2002
- It is based on the consideration than for small arrays insertion sort is faster than merge sort
- Thus, tim sort is a hybrid sorting algorithm
  - It applies merge sort for “large“ arrays
  - It switches to insertion sort for “small“ arrays
  - In other words tim sort
    - Applies the stardard merge sort divide-and-conquer procedure to split arrays in sub-arrays
    - When the sub-arrays are small enough, it applies insertion sort to sort them
    - It restart merge-sort to merge sorted sub-arrays

From a few tens to a few hundreds of elements

Quick sort

- Proposed by Richard Hoare in 1962
- Quick sort proceeds as merge sort
  - It uses a divide and conquer (divide et impera) approach
    - The array is partitioned
    - Each partition in conquered
    - The two partitions are combined
  - Nevertheless, while merge sort does all the job in the combination (merge) phase, quick sort does all the job in the partition (division) phase
    - Partition is based on a specific element used as a separator and called **pivot**

Quick sort

- The overall logic is the following one
  - Partition phase
    - The array A[l..r] is partitioned in 2 subarrays L (left subarray) and R (right subarray)
      - Given a pivot element x
      - L, i.e., A[l..q-1], contains all elements less than the pivot, i.e., A[i]< x
      - R, i.e., A[q+1..r], contains all elements larger than the pivot, i.e., A[i] > x
      - The value x is placed in the right place, i.e., in its final position
      - Division doesn't necessarily halve the array

Quick sort

- Recursion phase
  - 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 phase
  - None

Implementation

Wrapper

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

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;
}
```

Recursive call

Boundaries

Termination condition

Division

Recursive calls

Element c is not moved any more

Partition

- There are several partition schemes
  - Hoare, Lomuto, etc.
  - We present the original Hoare partition scheme
- The pivot may be selected in several ways
  - We select the pivot as the rightmost element of the subarray
    - pivot = A[r]
- Then the partition phase proceeds as follows

Partition

- It sets
  - i=l-1 and j=r
- A first cycle (ascending loop) increments i until it finds an element A[i] greater than the pivot x
- A second cycle (descending loop) decrements j until it finds an element less than the pivot x
- As the elements A[i] and A[j] are on the wrong array partition
  - Swap A[i] and A[j]
- Repeat until i < j
- Swap A[i] and pivot x
- Return the value of i to partition the array into subarrays

Implementation

```
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;
}
```

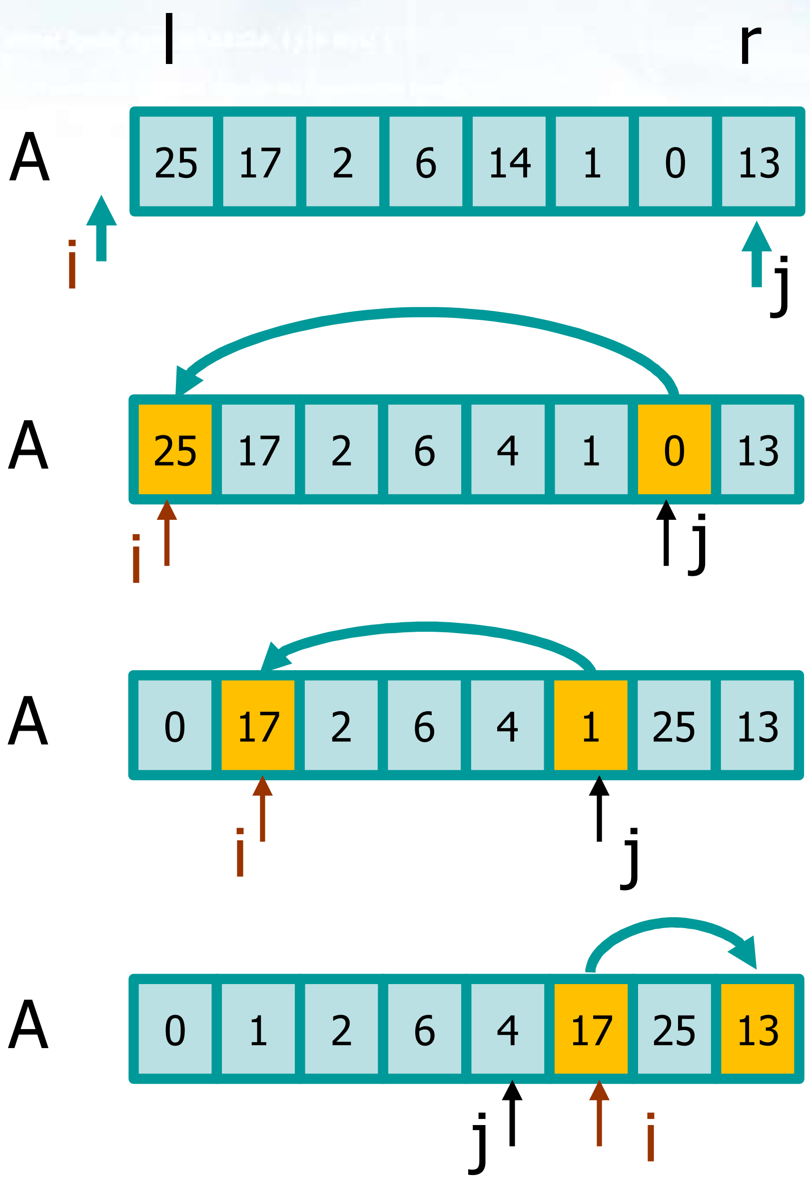
Pivot values are moved in the right sub-array; worst case: stop on pivot

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

```
void swap (int *v, int n1, int n2) {
  int temp;
  temp=v[n1];v[n1]=v[n2];v[n2]=temp;
  return;
}
```

Example

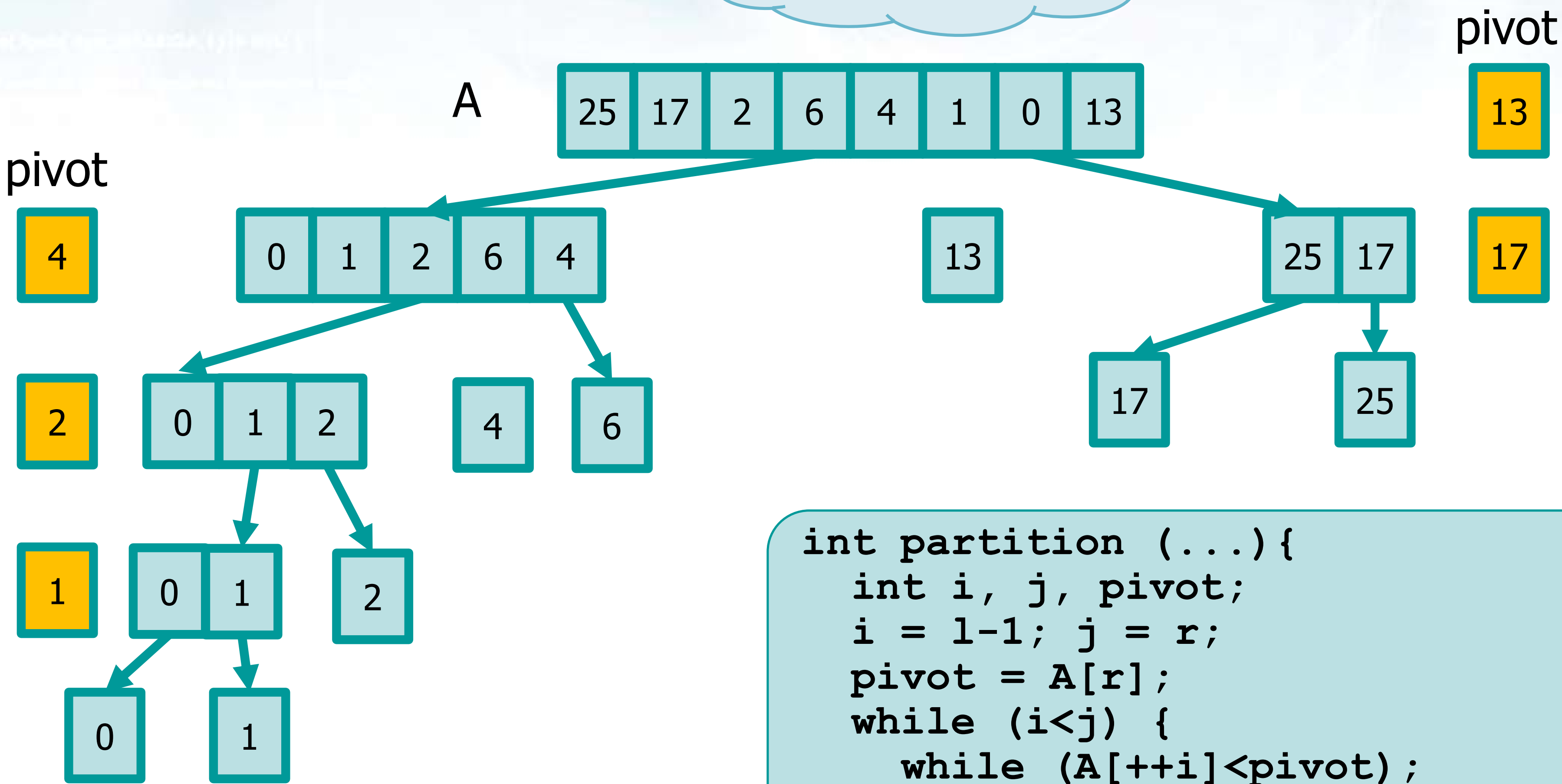
Partition ...




```
int partition (...){
  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;
}
```

Example

Quick sort ...




```
int partition (...){
  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;
}
```

Example: Scrambled order

pivot	0	1	2	3	4	5	6	7	8	9
	1	8	0	2	3	9	4	6	5	7
7	1	5	0	2	3	6	4	7	8	9
4	1	3	0	2	4	6	5	7	8	9
2	1	0	2	3	4	6	5	7	8	9
0	0	1	2	3	4	6	5	7	8	9
5	0	1	2	3	4	5	6	7	8	9
9	0	1	2	3	4	5	6	7	8	9

Example: Ascending order

This case in very unconvenient

pivot	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
9	0	1	2	3	4	5	6	7	8	9
8	0	1	2	3	4	5	6	7	8	9
7	0	1	2	3	4	5	6	7	8	9
6	0	1	2	3	4	5	6	7	8	9
5	0	1	2	3	4	5	6	7	8	9
4	0	1	2	3	4	5	6	7	8	9
3	0	1	2	3	4	5	6	7	8	9
2	0	1	2	3	4	5	6	7	8	9
1	0	1	2	3	4	5	6	7	8	9

Example: Descending order

This case in very unconvenient

pivot	0	1	2	3	4	5	6	7	8	9
	9	8	7	6	5	4	3	2	1	0
0	0	8	7	6	5	4	3	2	1	9
9	0	8	7	6	5	4	3	2	1	9
1	0	1	7	6	5	4	3	2	8	9
8	0	1	7	6	5	4	3	2	8	9
2	0	1	2	6	5	4	3	7	8	9
7	0	1	2	6	5	4	3	7	8	9
3	0	1	2	3	5	4	6	7	8	9
6	0	1	2	3	5	4	6	7	8	9
4	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
- Complexity
  - Efficiency depends on the partition balance
  - Balancing depends on the choice of the pivot

Complexity Analysis

This happens when the array is already sorted in ascending or descending order

- Worst case
  - The pivot is the minimum or the maximum value within the array
    - Quick sort generates a subarray with n-1 elements and a subarray with 1 element
  - Recursion equation
    - $T(n) = n + T(n-1)$ $\quad\quad n>=2$
    - $T(1) = 1$ $\quad\quad n = 1$
  - That is
    - $T(n) = n + n + n + n + n$

n times,
i.e., $n \cdot n$

  - Time complexity
    - $T(n) = O(n^2)$

Complexity Analysis

- Best case
  - At each step **partition** returns 2 subarrays with n/2 elements
  - Recursion equation
    - $T(n) = n + 2 \cdot T(n/2)$ $n>=2$
    - $T(1) = 1$ $n = 1$

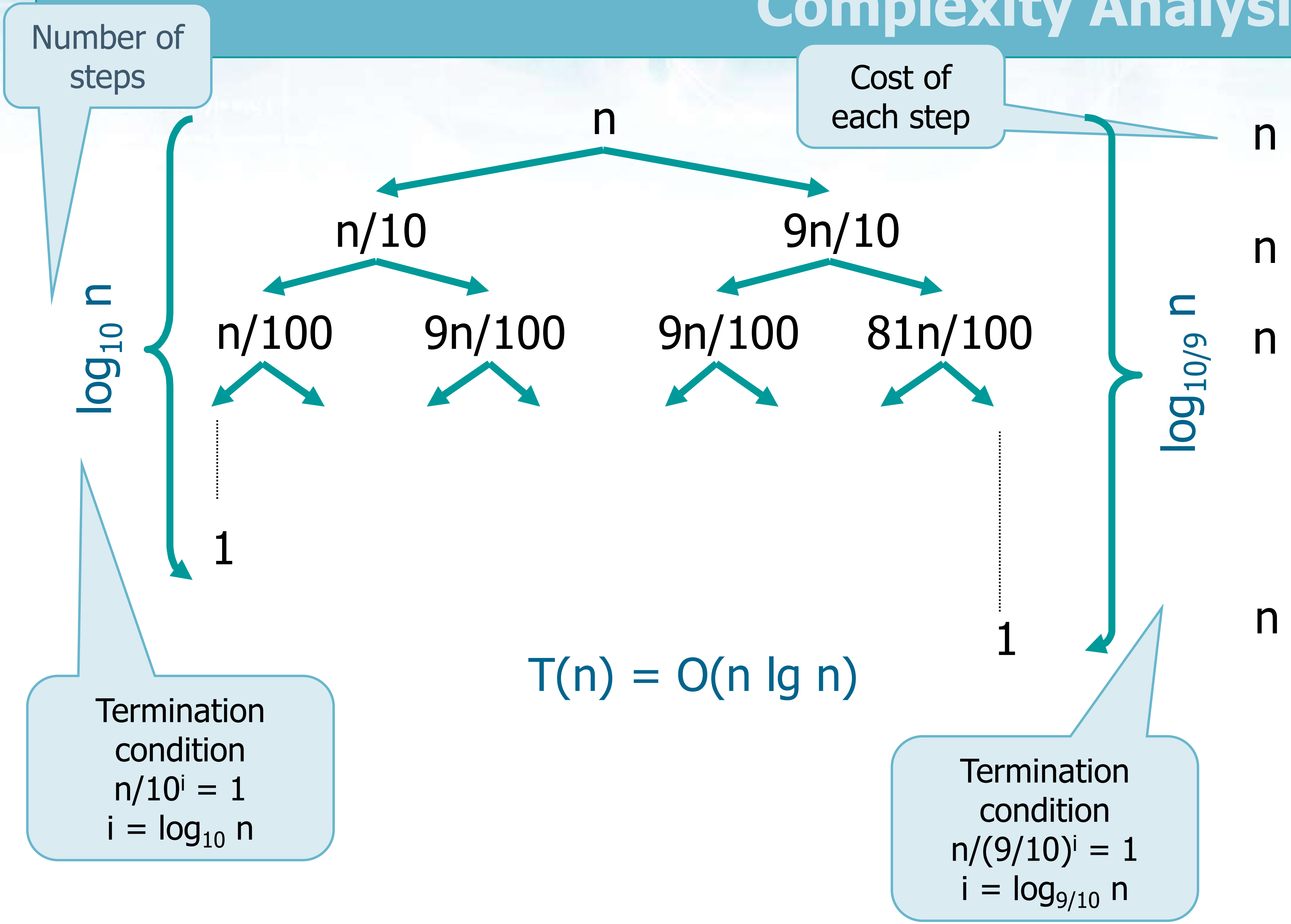
As for merge sort ...

  - Time complexity
    - $T(n) = O(n \cdot \lg n)$

Complexity Analysis

- Average case
  - At each step **partition** returns 2 subarrays of different sizes
  - Provided we are not in the worst case, though partitions may be strongly unbalanced
    - The average case leads to performances quite close to the ones of the best case
  - Example
    - At each step **partition** generates 2 partitions
    - Let us suppose the first one has (9/10·n) elements and the second one (1/10·n) elements

Complexity Analysis



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

Pivot selection

- Selecting the pivot is one of the main issues
- The pivot can be selected following several different strategies
  - Random element
    - Generate a random number i with p ≤ i ≤ r, then swap A[r] and A[i], use A[r] as pivot
  - Middle element
    - x = A[(p+r)/2]
  - Select average between min and max
  - Select median of 3 elements chosen randomly in array
  - …

Sorting algorithms

- A synoptic table for all anayzed sorting algorithms

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	$O(n)$