Recursion

Sorting

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Merge sort

- In computer science, merge sort (also spelled as mergesort) is an efficient, general-purpose, and divide and conquer algorithm
- Was invented by John von Neumann in 1945
  - A detailed description appeared in 1948

- It is a
  - Comparison-based sorting algorithm
  - Most implementations produce a stable sort

Merge sort

- Divide and conquer approach
  - 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

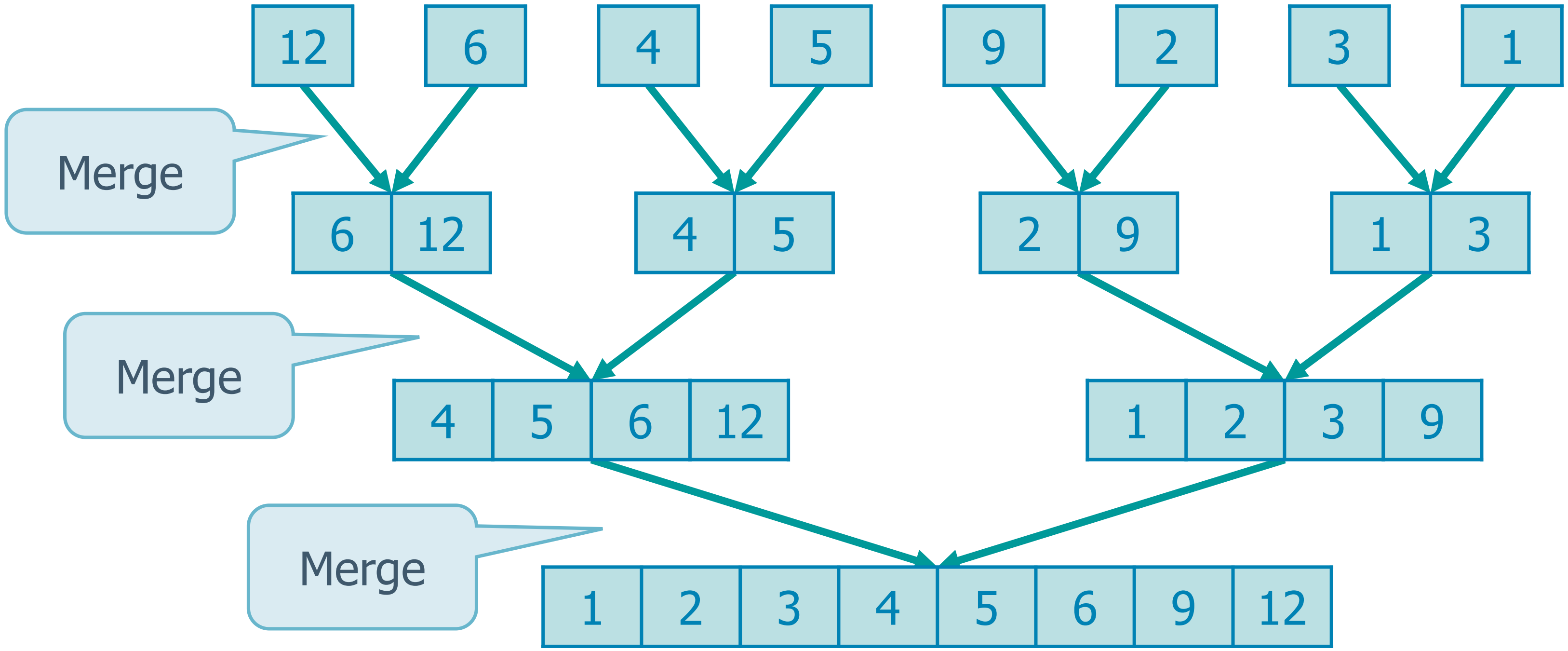
Divide does not reorder anything

Combine performes the sorting

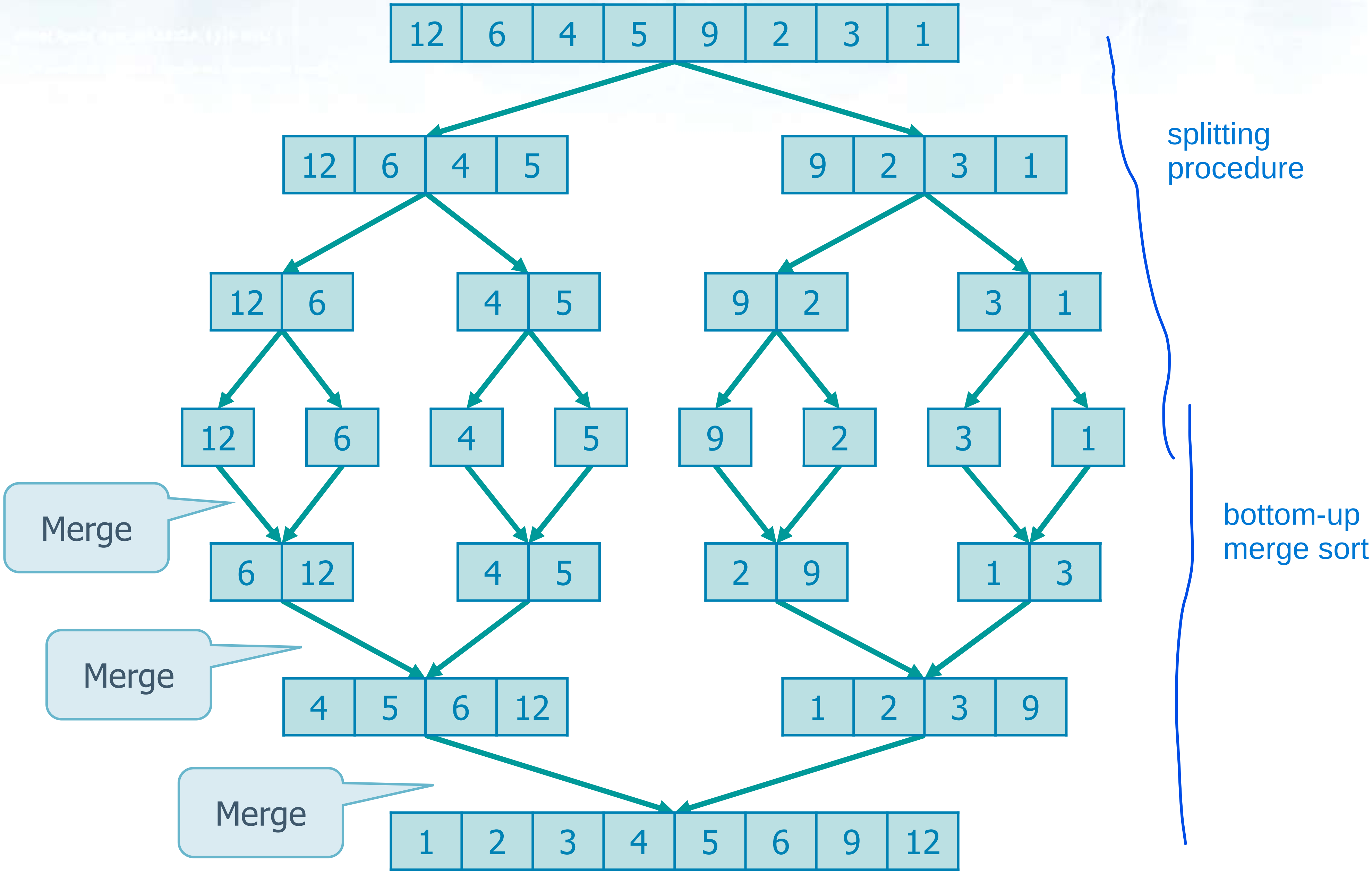
Example

First year program ...

```
void bottom_up_merge_sort (int *A, int N){
  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));
}
```

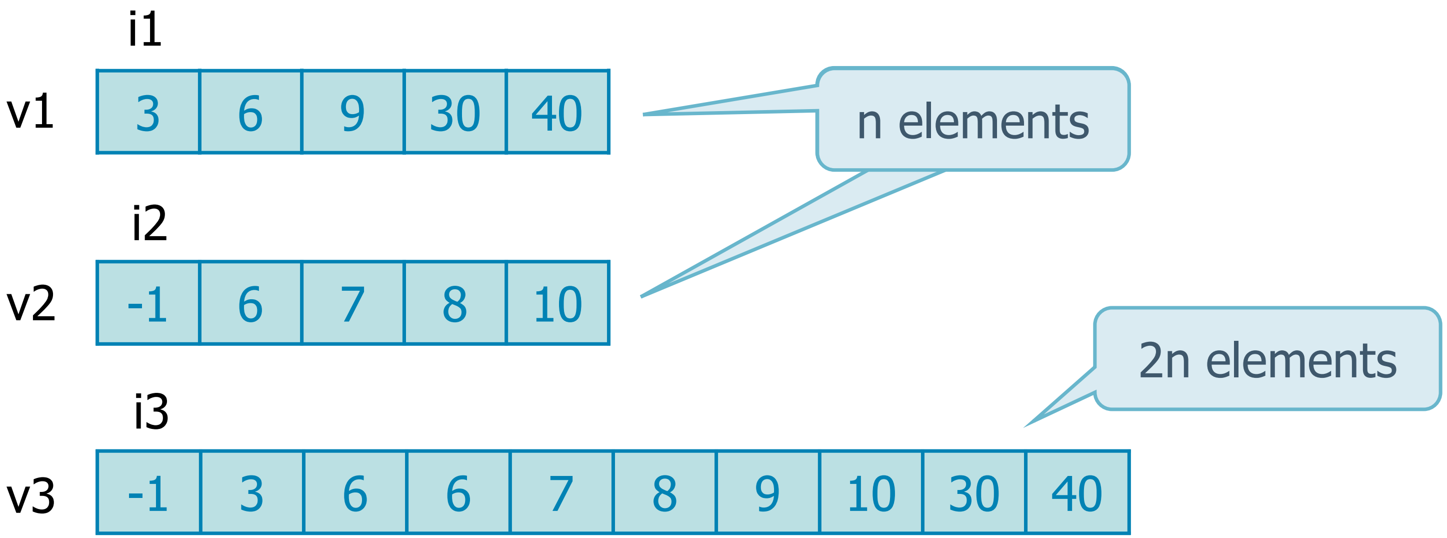


Example


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
6 12
4 5
2 9
1 3
4 5 6 12
1 2 3 9
1 2 3 4 5 6 9 12
Merge
Merge
Merge
splitting procedure
bottom-up merge sort

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



Merge

Stand-alone version

```c
void merge (int *v1, int *v2, int *v3, int n) {
  int i1=0, i2=0, 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++];
  }
  return;
}
```

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

v1	3	6	9	…				
v2	-1	6	7	…				
v3	-1	3	6	6	7	8	9	…

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)

Merge

Merge sort version

```
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

Merge sort

Wrapper

function that prepares the main function

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

for slide showing purposes only, always include them!!!

```
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;
}
```

Recursion

Left recursion

Right recursion

Combine
(merge on 2 partitions of the same array)

Features

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

```
...
if (A[i]<=A[j])
  B[k++] = A[i++];
else
  B[k++] = A[j++];
...
```

	0	1	2	3
A	1_1	1_2	1_3	4

	3	4	5	6
A	1_4	5	7	9

	0	1	2	3	4	5	6	7
A	1_1	1_2	1_3	1_4	4	5	7	9

Complexity Analysis

Analytic analysis …

Divide and conquer problem	
Number of subproblems	$a = 2$
Reduction factor	$b = {}^{n}/_{\hat{n}} = 2$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(n)$

Merge

$$T(n) = D(n) + a \cdot T\left({}^{n}/_{b}\right) + C(n) \qquad n > 1$$

$$T(n) = \Theta(1) \qquad n \leq 1$$

```
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

$$T(n) = D(n) + a \cdot T\left({}^{n}/_{b}\right) + C(n) \quad n > 1$$
$$T(n) = \Theta(1) \quad n \leq 1$$

$$T(n) = 2 \cdot T\left({}^{n}/_{2}\right) + n \quad n > 1$$
$$T(1) = 1 \quad n \leq 1$$

Unfolding

$$T(n) = n + 2 \cdot T\left({}^{n}/_{2}\right)$$
$$T\left({}^{n}/_{2}\right) = {}^{n}/_{2} + 2 \cdot T\left({}^{n}/_{4}\right)$$
$$T\left({}^{n}/_{4}\right) = {}^{n}/_{4} + 2 \cdot T\left({}^{n}/_{8}\right)$$
$$T\left({}^{n}/_{8}\right) = {}^{n}/_{8} + 2 \cdot T\left({}^{n}/_{16}\right)$$
$$\ldots$$
$$T(1) = 1$$

For the sake of simplicity, we can assume $n = 2^i$

Termination condition

$$\frac{n}{2^i} = 1$$
$$n = 2^i$$
$$i = log_2(n)$$

Complexity Analysis

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

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

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

$$T(n/8) = n/8 + 2 \cdot T(n/16)$$

...

$$T(1) = 1$$

$i = log_2(n)$
steps

Substitution

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

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

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

$$T(n) = n + n + n + n + 16 \cdot T(n/16)$$

...

$$T(n) = \sum_{i=1}^{log_2 n} n =$$

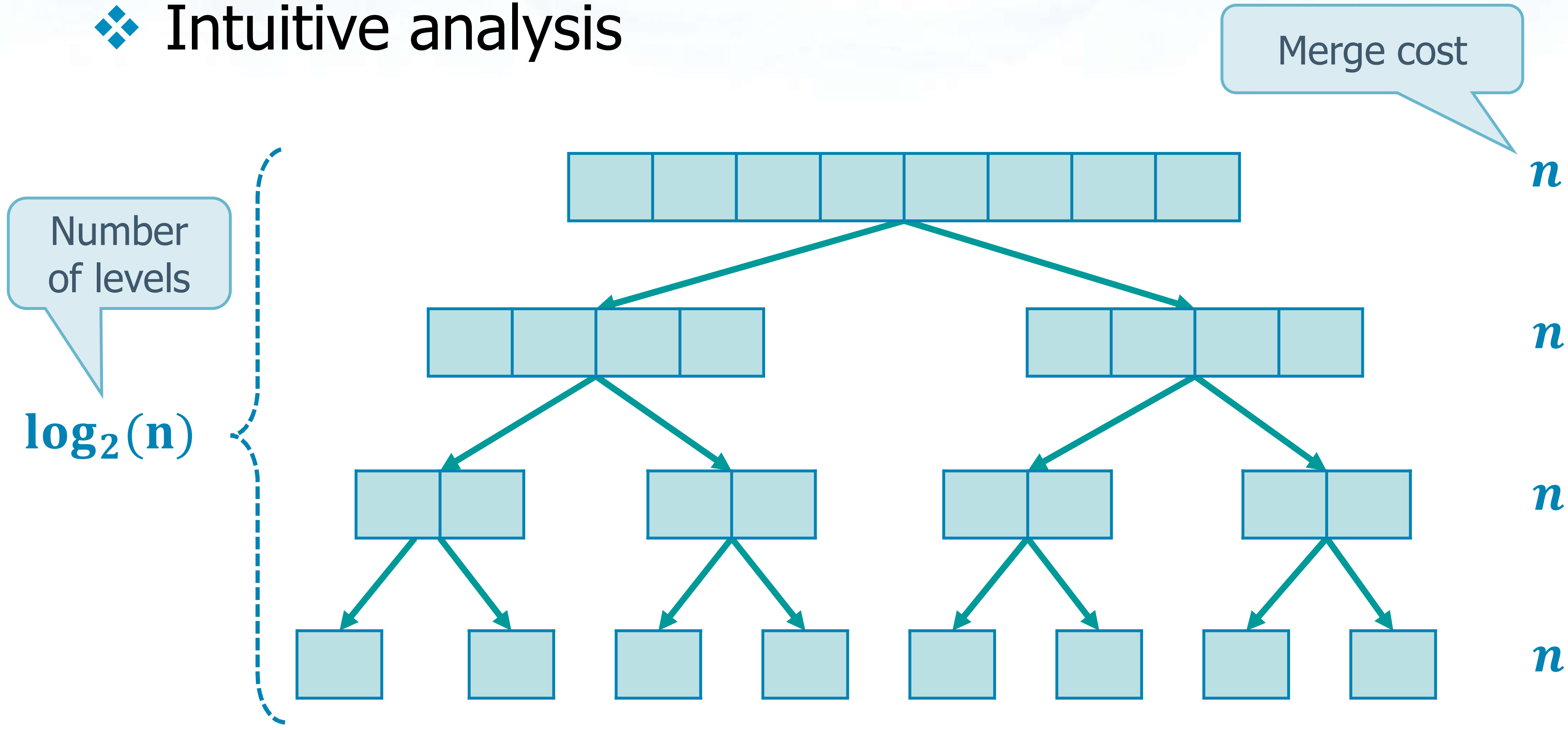
$$= n \sum_{i=1}^{log_2 n} 1 =$$

$$= n \cdot log_2(n) =$$

$$= O(n \cdot log_2(n))$$

Complexity Analysis

- Intuitive analysis



Recursion levels: $\log_2(n)$

Operations at each level: n

Total operations: $n \cdot \log_2(n)$

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

- Quicksort is a divide-and-conquer in-place sorting algorithm
- Developed by Sir Tony Hoare in 1959
  - Published in 1961

- It is a commonly used algorithm for sorting
- When implemented well, it can be faster than merge sort and about two or thee times faster than heap sort

Quick sort

- Quick sort proceeds as merge sort
  - It uses a divide and conquer (divide et impera) approach
    - It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot
  - Anyway, 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

- Starts with i=l-1 and j=r
  - A first cycle (ascending loop) increments i until it finds an element A[i] larger than the pivot x
  - A second cycle (descending loop) decrements j until it finds an element less than the pivot x
  - If the elements A[i] and A[j] are on the wrong array partition
    - Swap A[i] and A[j]
- Repeat until i < j termination condtion
- Swap A[i] and pivot x
- Return the value of i to partition the array

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;
}
```

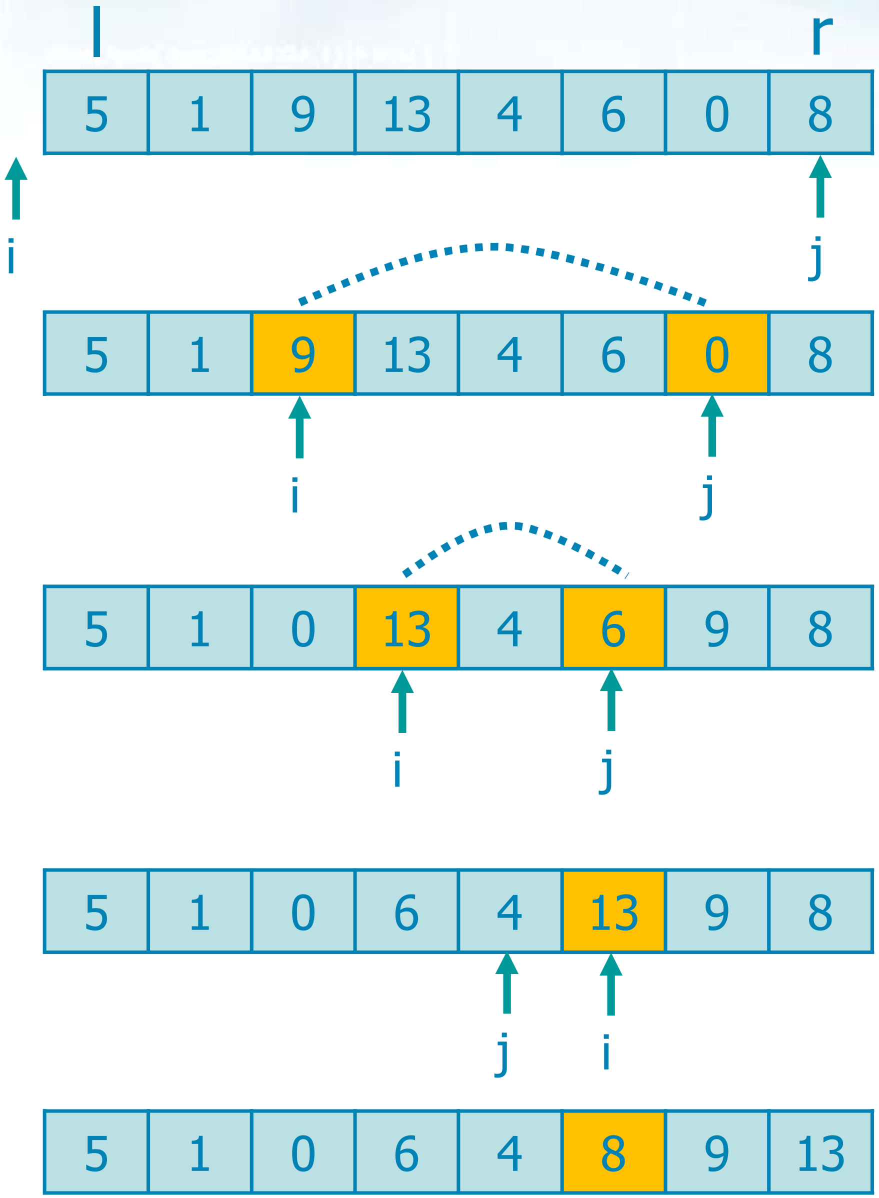
Pivot values are moved in the right sub-array; worst case: stop on pivot

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 ...



8 pivot

moving before checking

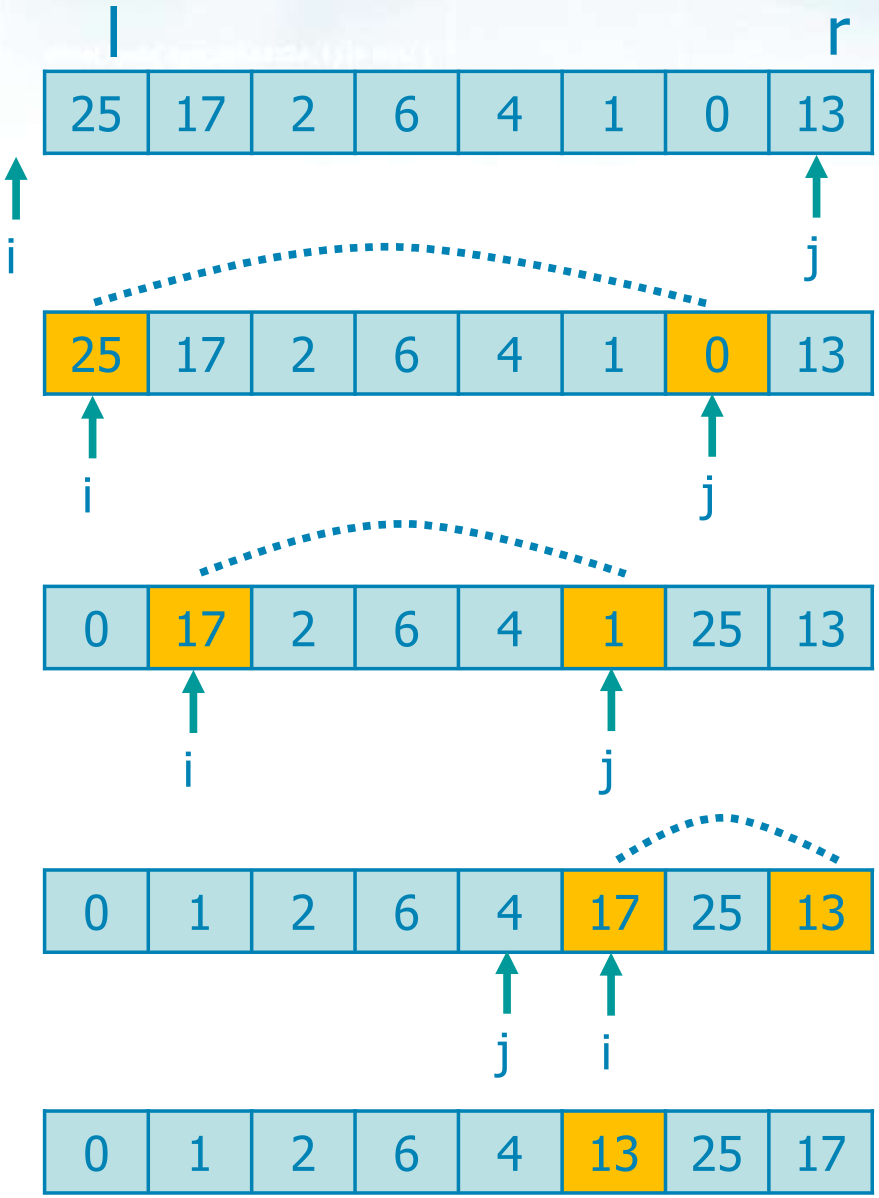
pivot works like a sentinel

```
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

Partition ...

13 pivot

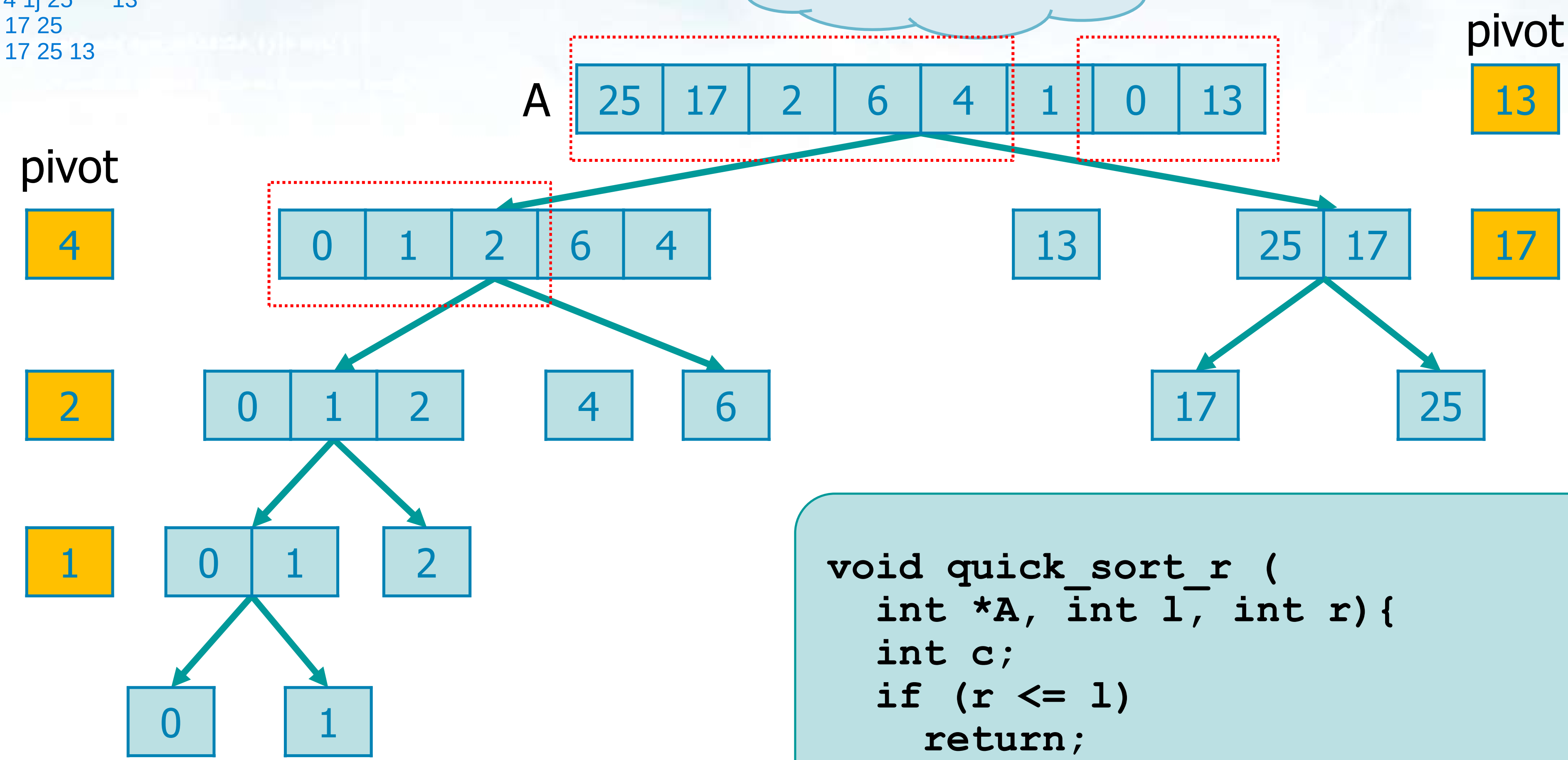



```
int partition (...){
  int i, j, pivot;
  i = l-1; j = r;
  pivot = A[r];
  while (i<j) {
    while (A[++i]<pivot);
    while (j>l && A[--j]>=pivot);
    if (i < j) swap(A, i, j);
  }
  swap (A, i, r);
  return i;
}
```

Example

Quick sort ...

0 17i 2 6 4 1j 25 13
0 1 2 6 4 17 25
0 1 2 4 6 17 25 13



0 1 2 6 4
j i

since i<j is not satisfied
exit the loop and swap i with pivot

0 1 2 4 6

```
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;
}
```

Example: Scrambled order

i

j

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

```
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;
}
```

```
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;
}
```

Example: Ascending order

This case in very inconvenient

if already sorted it still checks everything so a nightmare for the algo

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 inconvenient

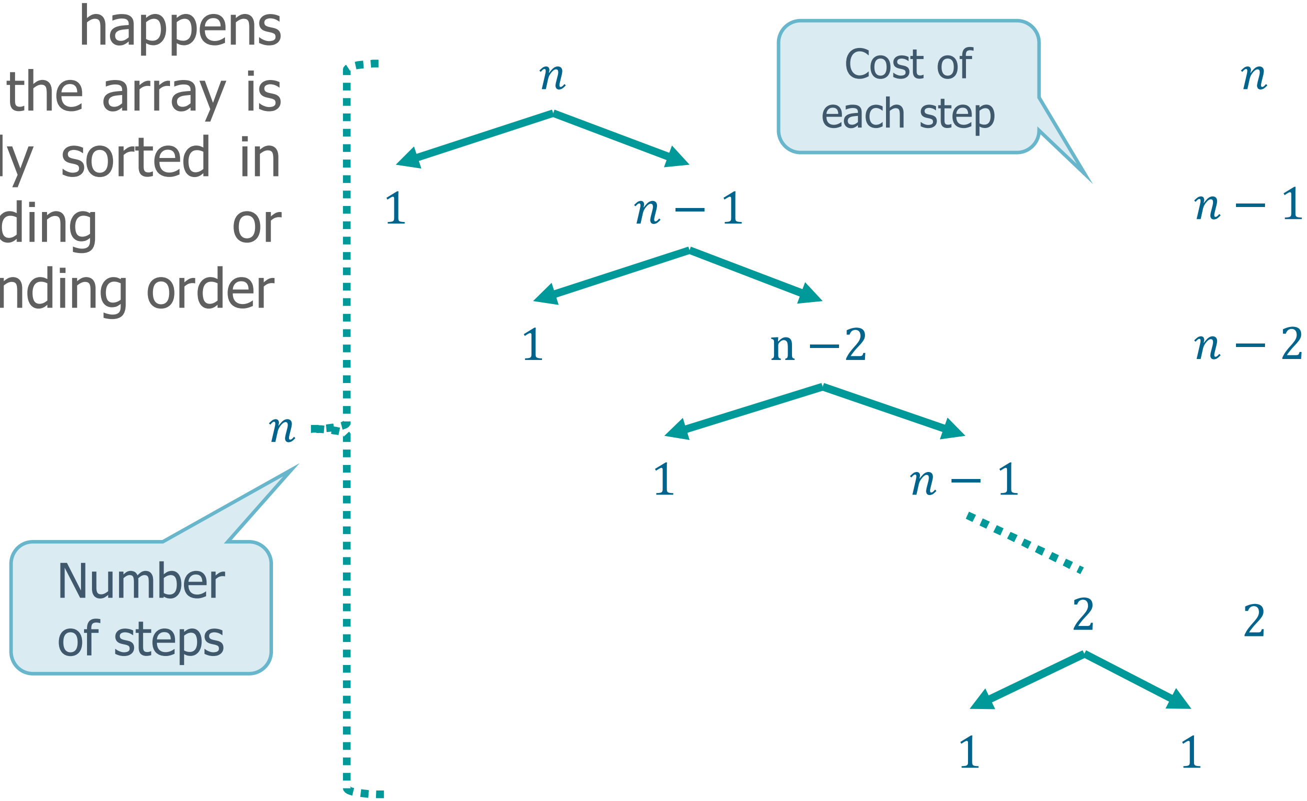
pivot	0	1	2	3	4	5	6	7	8	9
	9	8	7	6	5	4	3	2	1	0
0	0	8	7	6	5	4	3	2	1	9
9	0	8	7	6	5	4	3	2	1	9
1	0	1	7	6	5	4	3	2	8	9
8	0	1	7	6	5	4	3	2	8	9
2	0	1	2	6	5	4	3	7	8	9
7	0	1	2	6	5	4	3	7	8	9
3	0	1	2	3	5	4	6	7	8	9
6	0	1	2	3	5	4	6	7	8	9
4	0	1	2	3	4	5	6	7	8	9

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

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



Complexity Analysis

- Recursion equation

$$T(n) = n + T(n-1) \qquad n \geq 2$$
$$T(1) = 1 \qquad n = 1$$

- That is

$$T(n) = n + T(n-1)$$
$$T(n-1) = (n-1) + T(n-2)$$
$$T(n-2) = (n-2) + T(n-3)$$
$$\cdots$$
$$T(n) = n + (n-1) + (n-2) + \cdots + 2 =$$
$$= \frac{n(n+1)}{2} - 1 =$$
$$= O(n^2)$$

Complexity Analysis

- Best case
  - At each step **partition** returns 2 subarrays with $^n/_2$ elements
  - Recursion equation

$$T(n) = 2 \cdot T(^n/_2) + n \qquad n > 1$$
$$T(1) = 1 \qquad n \leq 1$$

  - Time complexity

$$T(n) = n + n + n + n + 16 \cdot T(^n/_{16}) =$$
$$= \sum_{i=0}^{\log n} n = n \sum_{i=0}^{\log n} 1 =$$
$$= n \cdot log_2(n) =$$
$$= O(n \cdot log_2(n))$$

As for merge sort ...

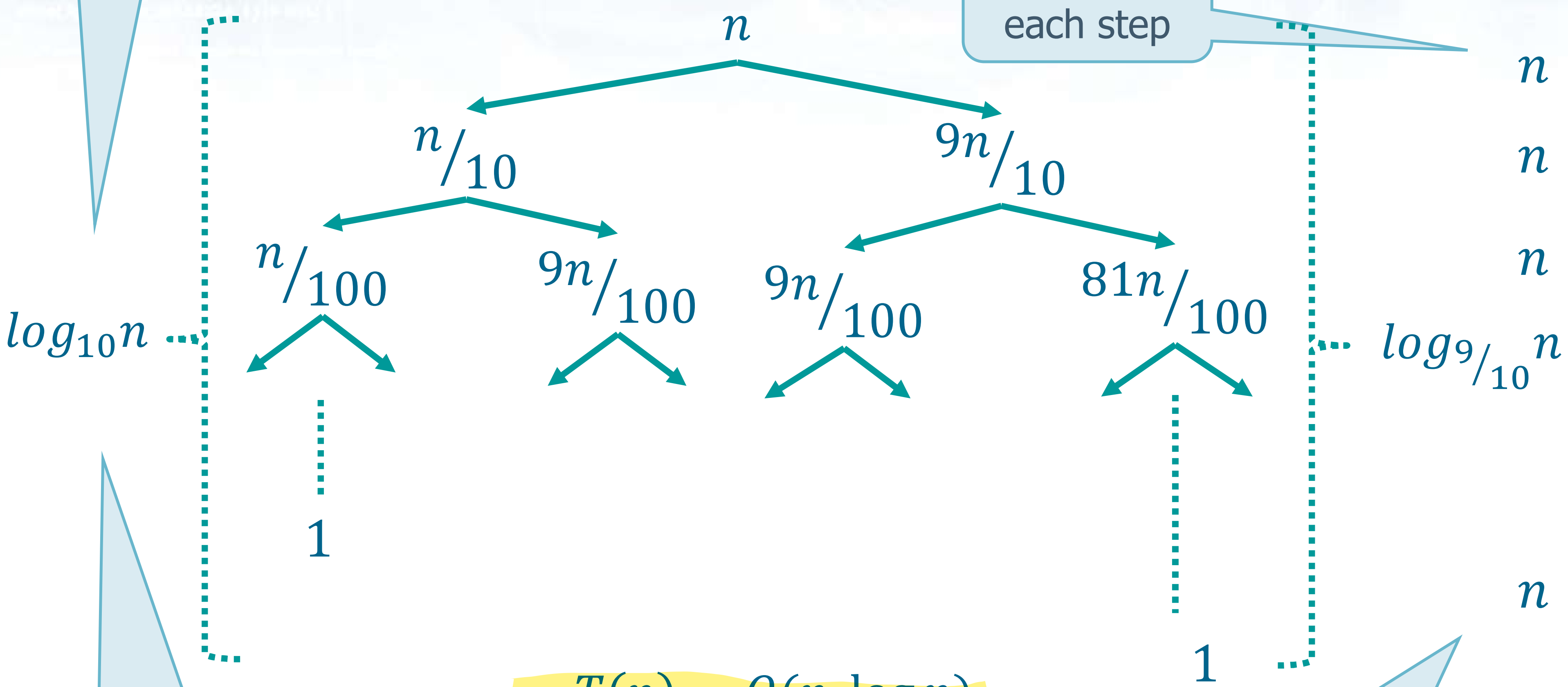
Complexity Analysis

- 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 $(^9/_{10} \cdot n)$ elements and the second one $(^1/_{10} \cdot n)$ elements

Complexity Analysis

Number of steps

Cost of each step



$T(n) = O(n \cdot \log n)$

Termination condition
$n/10^i = 1$
$i = log_{10}n$

Termination condition
$n/(9/10)^i = 1$
$i = log_{9/10}n$

Pivot selection

- Selecting the pivot is one of the main problem
- 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
    - $x = A[{}^{(p+r)}/_{2}]$
  - Average between min and max
  - Median of 3 elements chosen randomly in array
  - ...

IN THE EXAM CHOOSE RIGHTMOST ELEMENT

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	$O(n)$

Sorting algorithms