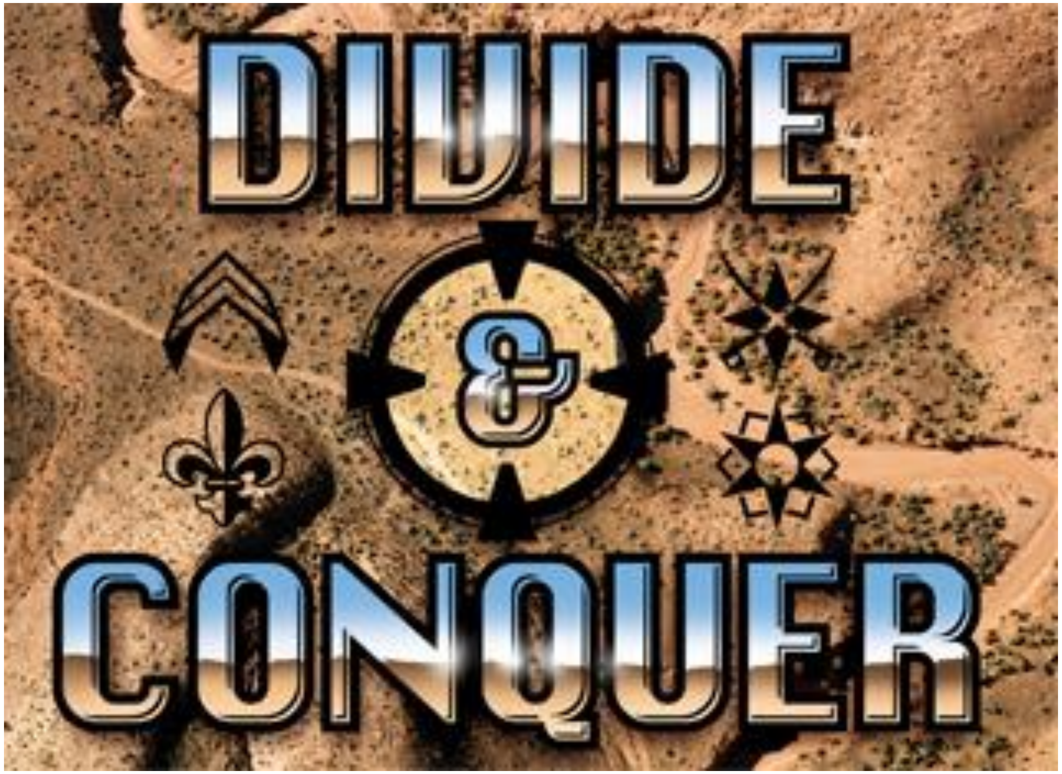
Lecture 2: Merge sort

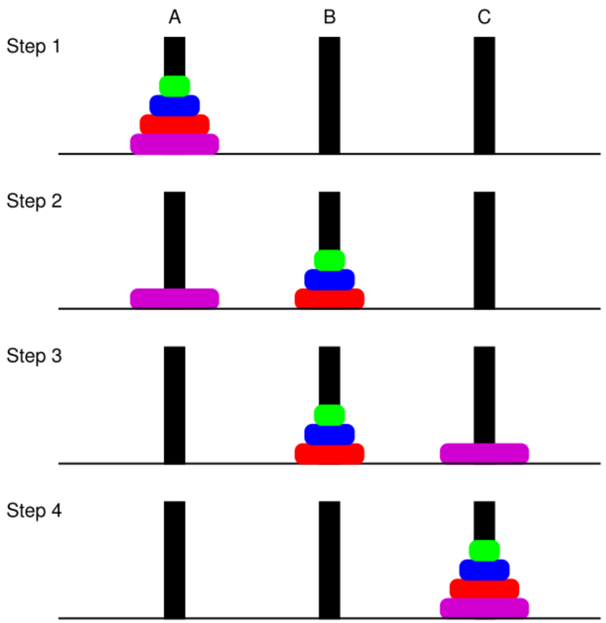


A quick review of recursion and recurrences

Classical example: Tower of Hanoi

Goal: Move n discs from peg A to peg C

- One disc at a time
- Can't put a larger disc on top of a smaller one




```
MoveTower(n, peg1, peg2, peg3):
if n = 1 then
    move the only disc from peg1 to peg3
    return
else
    MoveTower(n − 1, peg1, peg3, peg2)
    move the only disc from peg1 to peg3
    MoveTower(n − 1, peg2, peg1, peg3)

First call: MoveTower(n, A, B, C)
```

Keys things to remember:

- Reduce a problem to the same problem, but with a smaller size
- The base case

Analyzing a recursive algorithm with recurrence

Q: How many steps (movement of discs) are needed?

Analysis: Let $T(n)$ be the number of steps needed for n discs.

From the recursive algorithm, we have

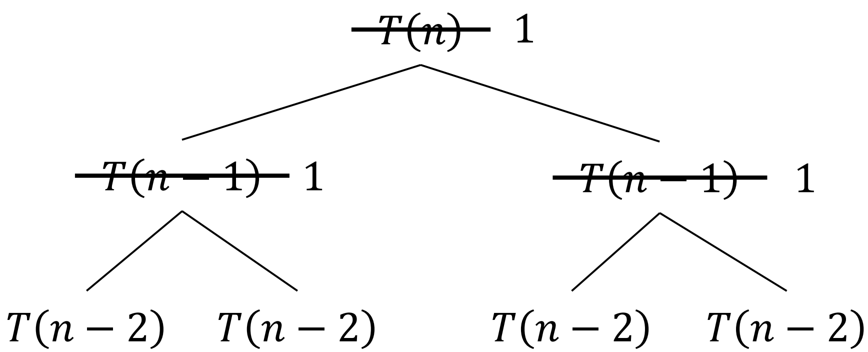
$$T(n) = 2T(n-1) + 1, \qquad n > 1$$
$$T(1) = 1$$

Solving the recurrence by the expansion method:

$$\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2(2(T(n-2) + 1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= 2^2(2T(n-3) + 1) + 2 + 1 \\
&= 2^3 T(n-3) + 2^2 + 2 + 1 \\
&= \cdots \\
&= 2^{n-1} T(1) + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^n - 1 = \Theta(2^n)
\end{aligned}$$

Solving recurrences with the recursion tree method

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



level 0: 1 node

level 1: 2 nodes

level 2: 2^2 nodes

level i: 2^i nodes

total number of nodes: $1 + 2 + 2^2 + \cdots 2^{n-1} = 2^n - 1$

level $n-1$: 2^{n-1} nodes

Note: This is actually equivalent to the expansion method, but clearer.

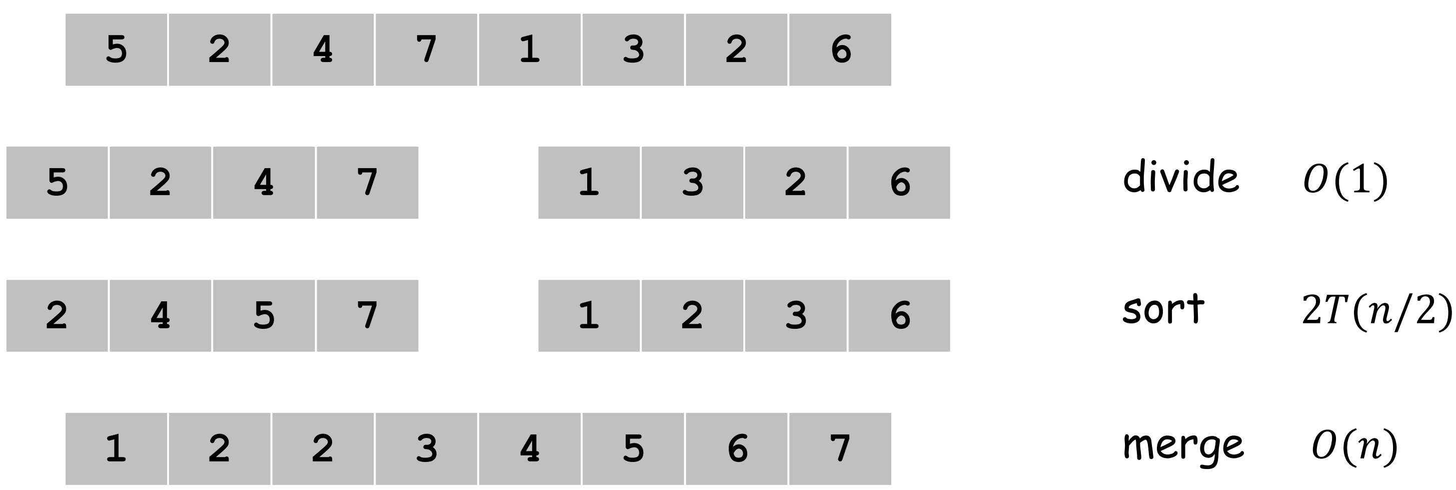
Merge sort

Merge sort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

Mergesort(A, p, r)**:**
if $p = r$ **then return**
$q \leftarrow \lfloor (p+r)/2 \rfloor$
Mergesort(A, p, q)
Mergesort$(A, q+1, r)$
Merge(A, p, q, r)

First call: **Mergesort**$(A, 1, n)$

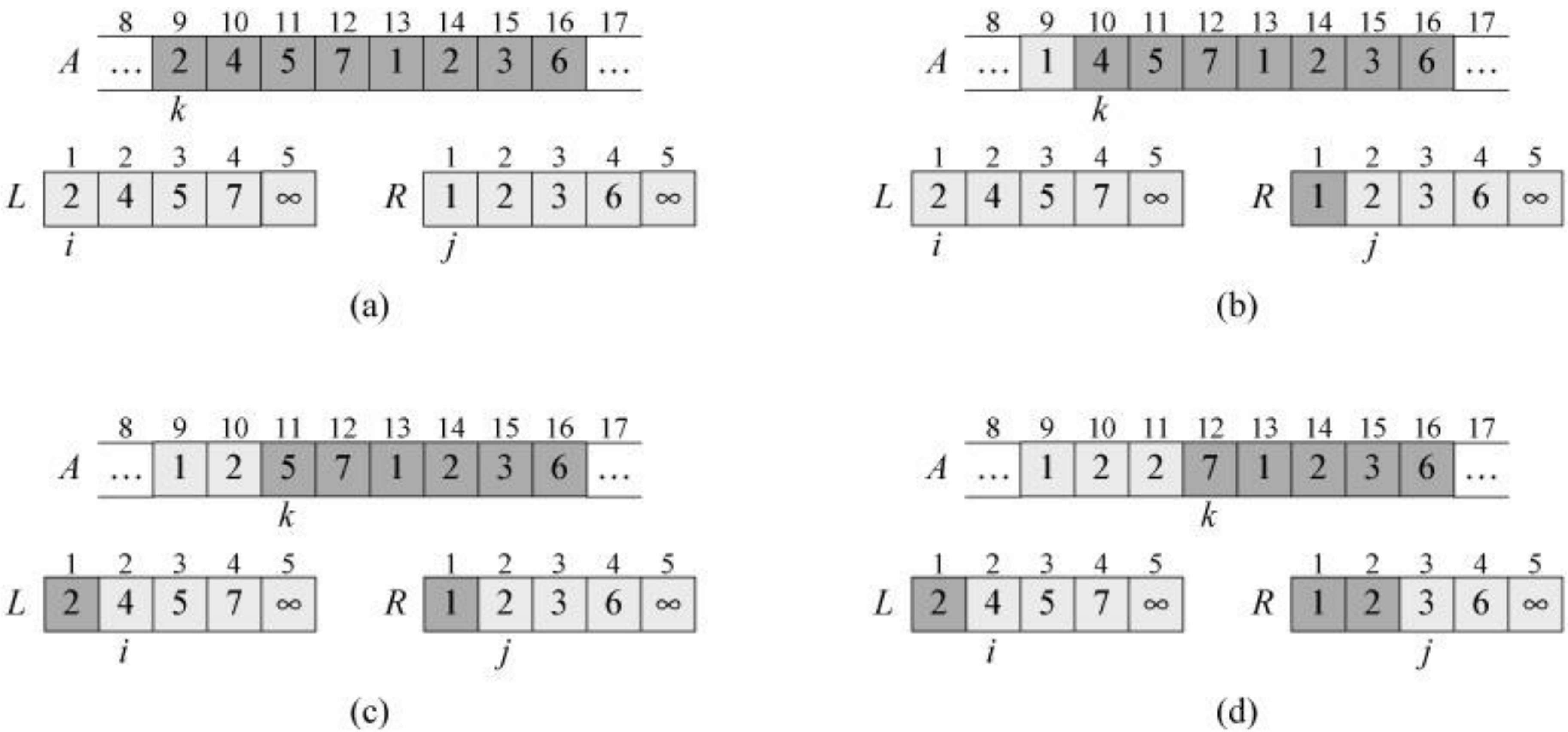


Merge

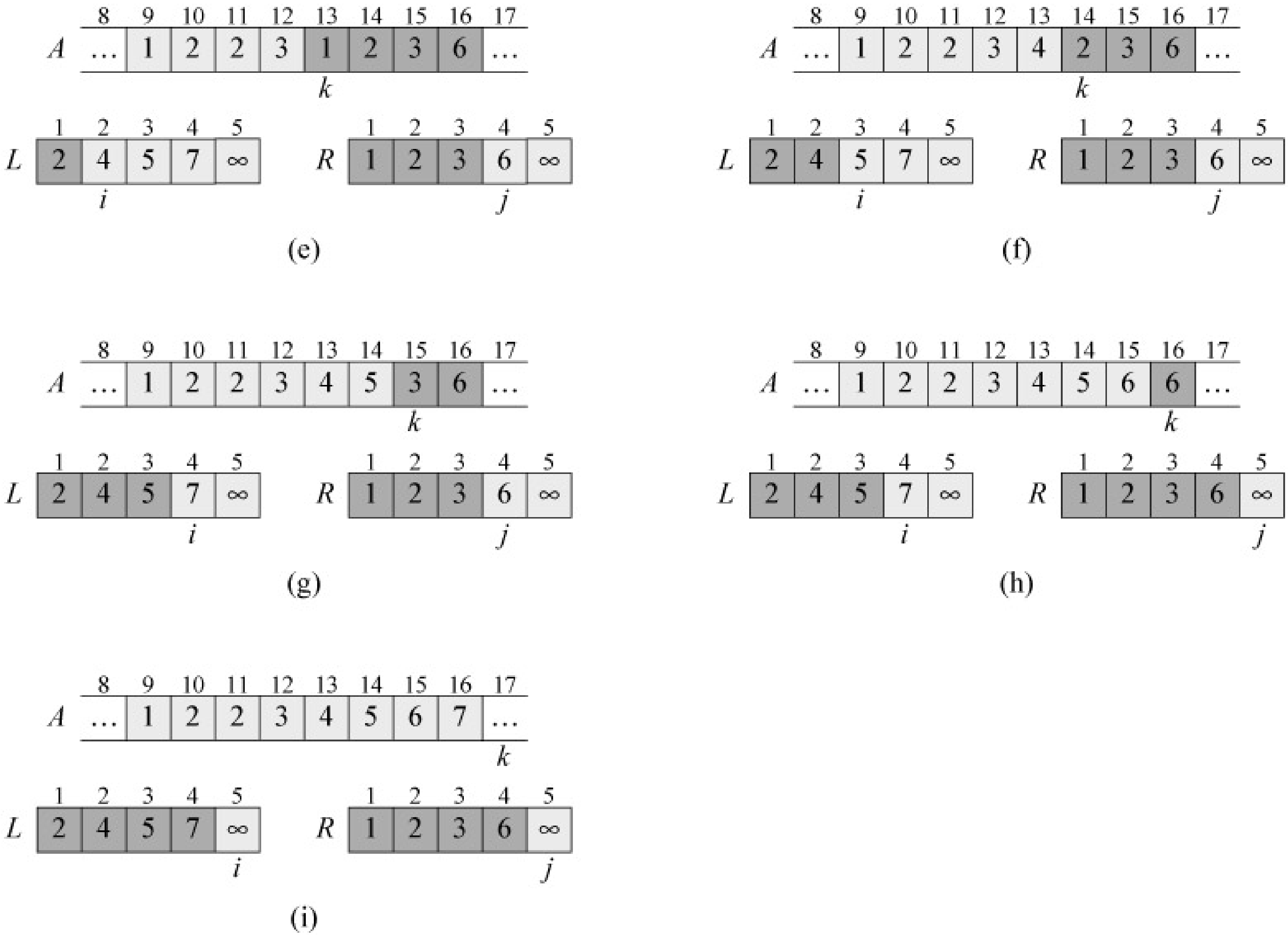
Merge. Combine two sorted lists into a sorted whole.

```
Merge(A,p,q,r):
create two new arrays L and R
L ← A[p..q], R ← A[q+1..r]
append ∞ at the end of L and R
i ← 1, j ← 1
for k ← p to r
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i+1
    else
        A[k] ← R[j]
        j ← j+1
```

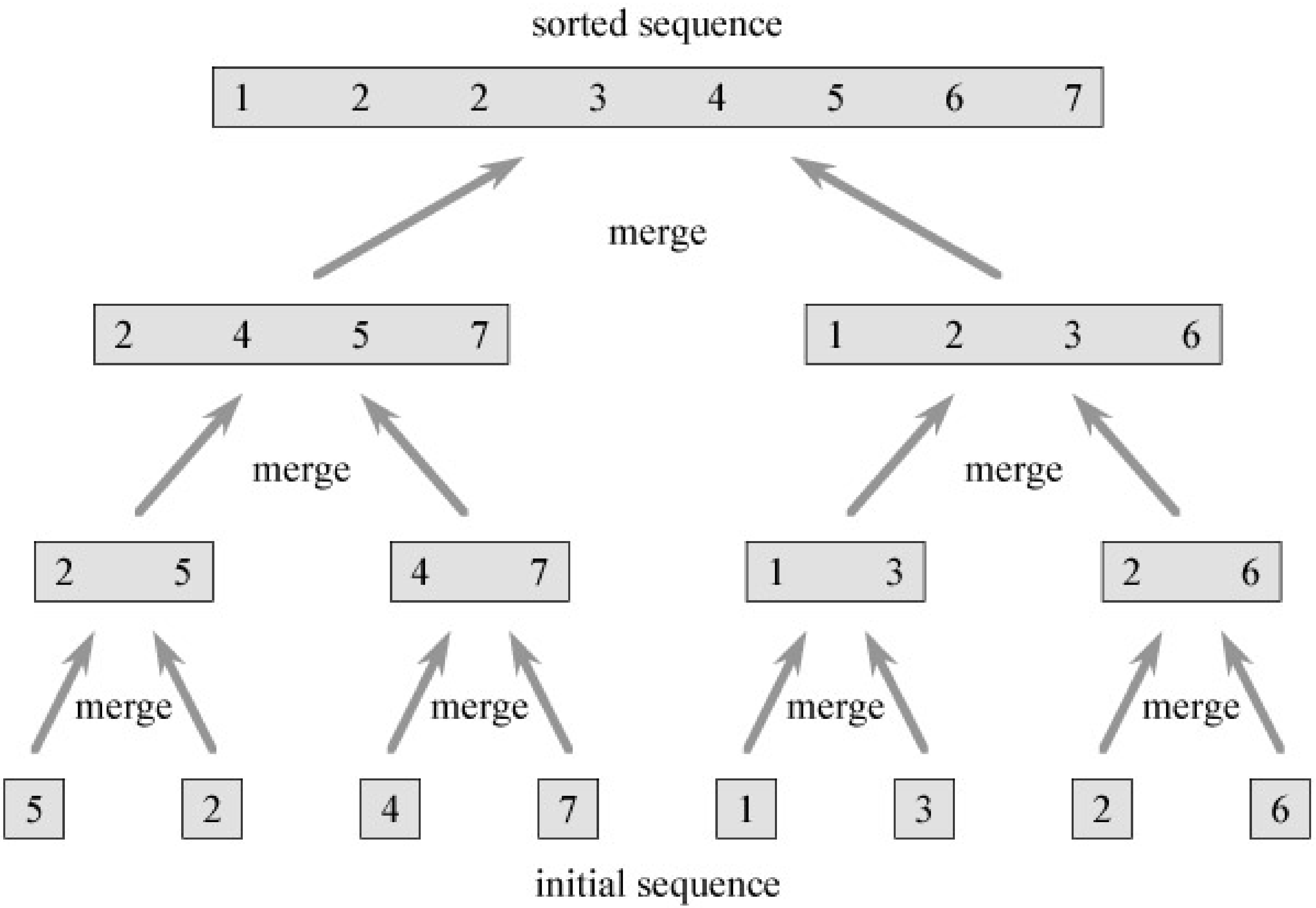
Merge: Example



Merge: Example



Merge sort: Complete example



Analyzing merge sort

Def. let $T(n)$ be the running time of the algorithm on an array of size n.

Merge sort recurrence.

$$T(n) \le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n), \qquad n > 1$$
$$T(1) = O(1)$$

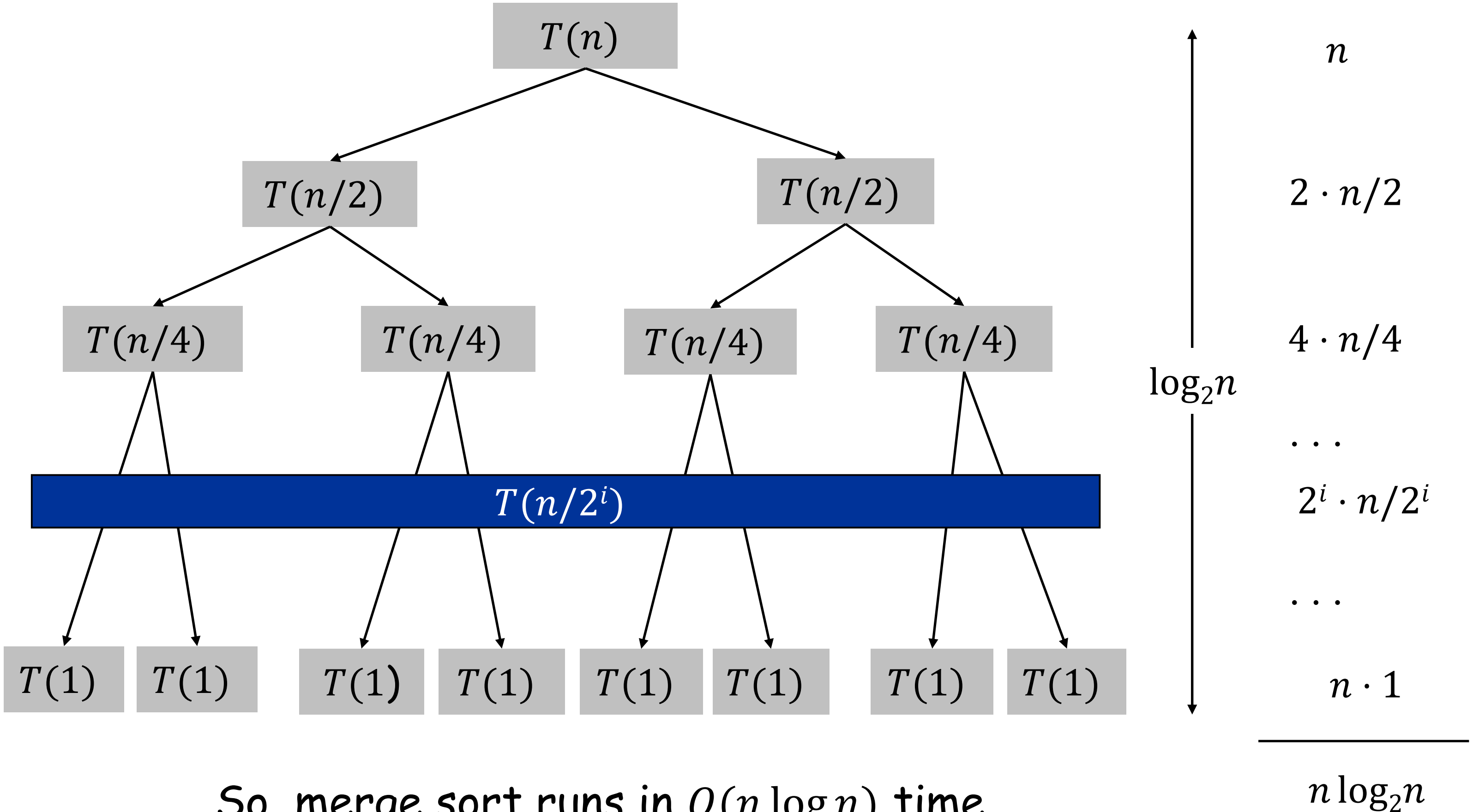
A few simplifications

- Replace $\le$ with $=$
  - since we are interested in an big-Oh upper bound of $T(n)$
- Replace $O(n)$ with n, replace $O(1)$ with 1
  - since we are interested in an big-Oh upper bound of $T(n)$
- Assume n is a power of 2, so that we can ignore $\lfloor \quad \rfloor, \lceil \quad \rceil$
  - since we are interested in an big-Oh upper bound of $T(n)$
  - for any n, let n' be the smallest power of 2 such that $n' \ge n$, then $T(n) \le T(n') \le T(2n) = O(T(n))$, as long as $T(n)$ is a polynomial function.

Solve the recurrence

Simplified merge sort recurrence.

$$T(n) = 2T(n/2) + n, \qquad n > 1$$
$$T(1) = 1$$



So, merge sort runs in $O(n \log n)$ time.

Running time of merge sort

Q: Is the running time of merge sort also $\Omega(n \log n)$?

A: Yes, the worst-case input is when the array is reversely sorted

A: Actually, the running time is the same no matter what the input is

- Or equivalently speaking, every input is the worst case.
- The whole analysis holds if we replace every O with Ω

Theorem: Merge sort runs in time $\Theta(n \log n)$.

Inversion Number

Def:

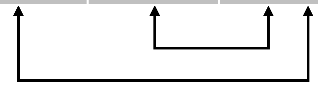
- Given array $A[1..n]$, two elements $A[i]$ and $A[j]$ are inverted if $i < j$ but $A[i] > A[j]$.
- The inversion number of A is the number of inverted pairs.

A useful measure for:

- How "sorted" an array is
- The similarity between two rankings

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5



Inversions

3-2, 4-2

Relation to bubble sort

1st Pass: (4 1 8 2 5) → (1 4 8 2 5) → (1 4 2 8 5) → (1 4 2 5 8)

inversion #: 4 3 2 1

2nd Pass: (1 4 2 5 8) → (1 2 4 5 8)

Theorem: The number of swaps used by bubble sort is equal to the inversion number.

Proof: Every swap decreases the inversion number by 1.

Observation: The same holds for insertion sort using swaps.

Q: How to compute the inversion number?

Algorithm 1: Check all $\Theta(n^2)$ pairs.

Algorithm 2: Run bubble sort and count the number of swaps - $\Theta(n^2)$ time, too.

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: divide array into two halves.
- Conquer: recursively count inversions in each half.
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: $\Theta(1)$.

1	5	4	8	10	2

5 blue-blue inversions

6	9	12	11	3	7

8 green-green inversions

Conquer: $2T(n/2)$

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole to maintain the sortedness invariant.

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: $\Theta(n)$

2	3	7	10	11	14	16	17	18	19	23	25

Merge: $\Theta(n)$

$$T(n) = 2T(n/2) + n,\ \ n > 1$$

(The base case $T(1) = 1$ can often be omitted.)

So, $T(n) = \Theta(n \log n)$

Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] $A[p..q]$ and $A[q+1,r]$ are sorted.

Post-condition. [Sort-and-Count] $A[p..q]$ is sorted.

```
Sort-and-Count(A,p,r):
if p = r then return 0
q ← ⌊(p + r)/2⌋
c1 ← Sort-and-Count(A,p,q)
c2 ← Sort-and-Count(A,q + 1,r)
c3 ← Merge-and-Count(A,p,q,r)
return c1 + c2 + c3

First call: Sort-and-Count(A,1,n)
```

```
Merge-and-Count(A,p,q,r):
create two new arrays L and R
L ← A[p..q], R ← A[q + 1..r]
append ∞ at the end of L and R
i ← 1, j ← 1
c ← 0
for k ← p to r
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i + 1
    else
        A[k] ← R[j]
        j ← j + 1
        c ← c + q − p − i + 2
```