Lecture 2: Divide & Conquer

Version of February 8, 2019

Divide-and-Conquer intro: Binary search

Main idea of DaC: Solve a problem of size *n* by breaking it into smaller problems of size less than *n*.

Example: Binary Search

Input: An array A of elements in sorted order, and an element x.

Output: Return the position of x if it exists; otherwise output nil.

4	7	10	15	19	20	42	54	87	90

BinarySearch(A,p,r,x)**:**

if $p > r$ **then return** nil

$q \leftarrow \lfloor (p+r)/2 \rfloor$

if $A[q] = x$ **return** q

if $x < A[q]$ **then BinarySearch**$(A,p,q-1,x)$

else BinarySearch$(A,q+1,r,x)$

First call: **BinarySearch**$(A,1,n,x)$

Binary search

Input: An array A of elements in sorted order, and an element x.
Output: Return the position of x if it exists; otherwise output nil.

Idea:
Set q = middle item.   Check if x = q.
If not, search either to left or right of q
as appropriate

4	7	10	15	19	20	42	54	87	90

BinarySearch(A, p, r, x)**:**
if $p > r$ **then return** nil
$q \leftarrow \lfloor (p+r)/2 \rfloor$
if $A[q] = x$ **return** q
if $x < A[q]$ **then** **BinarySearch**$(A, p, q-1, x)$
else **BinarySearch**$(A, q+1, r, x)$

First call: **BinarySearch**$(A, 1, n, x)$

Analysis of Binary Search

Analysis: Let $T(n)$ be the number of comparisons needed for n elements.

Recurrence: With a single comparison we eliminate half of the array.
=> we search for the element in the remaining half, which has size *n*/2.
Thus, the recurrence counting the number of comparisons is:

$T(n)=T(n/2)+1$ if $n>1$, and $T(1)= 1$.

Solve the recurrence by the expansion method:

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= (T(n/2^2) + 1) + 1 \\
&= T(n/2^2) + 2 \\
&= \ldots\ldots \\
&= T(n/2^i) + i \\
&= \ldots. \\
&= T(n/2^{\log_2 n}) + \log_2 n \\
&= T(1) + \log_2 n
\end{aligned}
$$

$$= \mathbf{1 + \log_2 n}$$

General Case

$i = \log_2 n$

Note: Binary search may terminate faster than $\Theta(\log n)$, but the worst-case running time is still $\Theta(\log n)$

Binary search recurrence with the recursion tree method

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

#problems (nodes) per level		Comparisons per level
level 0: 1	$T(n)$ 1	level 0: 1
level 1: 1	$T(n/2)$ 1	level 1: 1
level i: 1	$T(n/2^i)$ 1	level i: 1
level $log n - 1$: 1	$T\left(\frac{n}{2^{(\log_2 n)-1}}\right) = T(2)$ 1	level $\log_2 n - 1$: 1
	$T(1)$ 1	level $\log_2 n : 1$

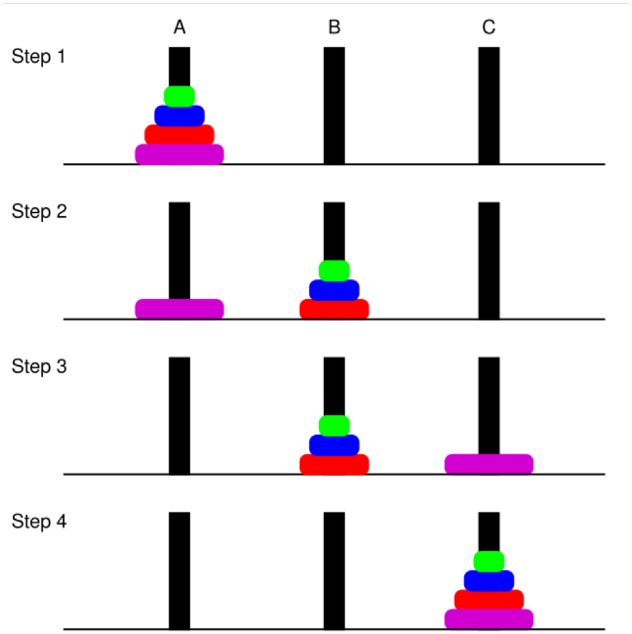
Total number of comparisons: $1 + 1 + .. + 1 = 1 + \log_2 n$

Note: This is actually equivalent to the expansion method but more visual.

More complex example: Towers 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.

In the recursive algorithm, to solve the problem of size *n*, we:

- 1: move $n-1$ disks from peg 1 to 2 $T(n-1)$
- 2: move 1 disk from peg 1 to 3 1
- 3: move $n-1$ disks from peg 2 to 3 $T(n-1)$

Thus, the recurrence counting the number of steps is:

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

$$T(1) = 1$$

Solving the recurrence with the Expansion method

The recurrence counting the number of steps is

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

Solve the recurrence by the expansion method:

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

$$= 2\left(2\,T\big((n-1)-1\big) + 1\right) + 1$$

$$= 2^2 T(n-2) + 2 + 1$$

$$= 2^2\left(2T\big((n-2)-1\big) + 1\right) + 2 + 1$$

$$= 2^3 T(n-3) + 2^2 + 2 + 1$$

$$= \;.....$$

General Case

$$= 2^i\,T(n-i) \quad + 2^{i-1} + 2^{i-2} + \cdots + 2^2 + 2 + 1$$

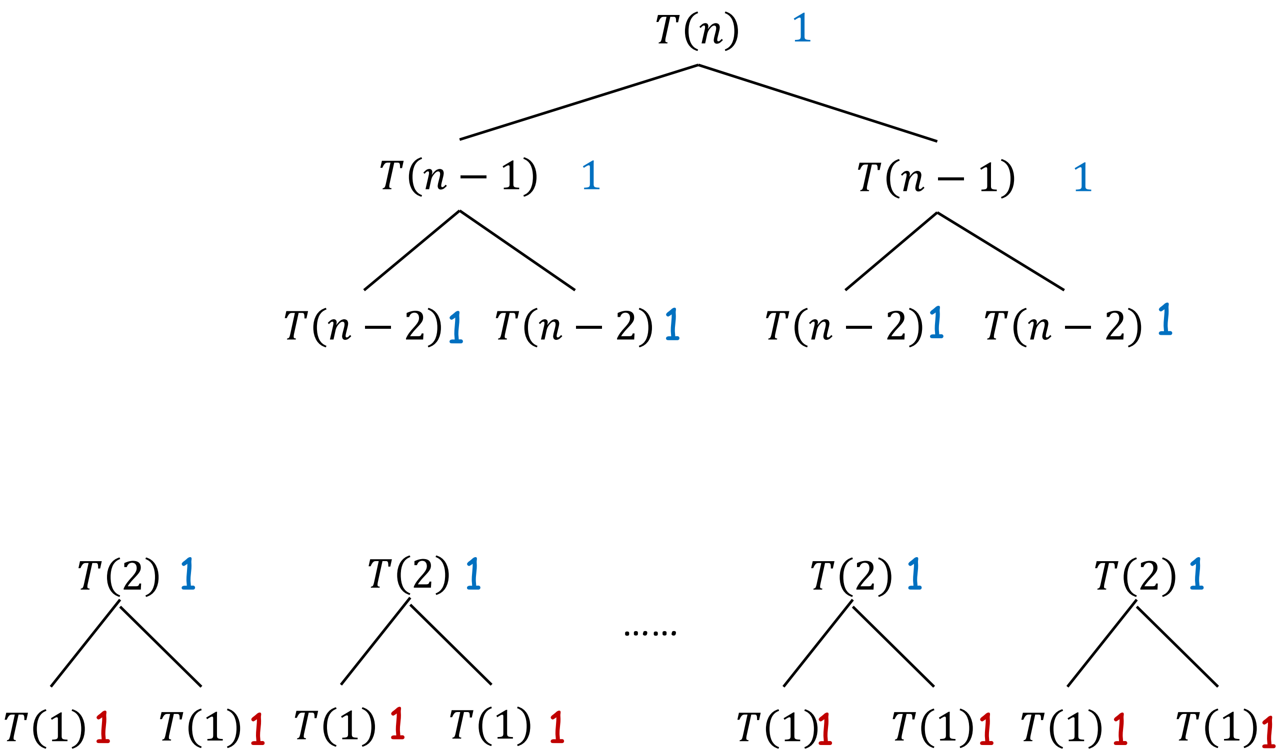
$$= \;.....$$

$i = n-1$

$$= 2^{n-1}\,T(1) + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 + 1 \qquad \mathbf{= 2^n - 1}$$

Solving the recurrence with the recursion tree method

For $n > 1$, $T(n) = 2T(n-1) + 1$, $\quad and \quad$ $T(1) = 1$



level 0: 1 node

level 1: 2 nodes

level 2: 2^2 nodes

level i: 2^i nodes

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

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

total number of nodes: $1 + 2 + 2^2 + \cdots 2^{n-1} = 2^n - 1$
each doing one unit of work

Merge sort

Merge sort.

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

<u>**Mergesort**(A,p,r)**:**</u>
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)

<u>**First call:**</u> **Mergesort**$(A,1,n)$

5	2	4	7	1	3	2	6

5	2	4	7		1	3	2	6	divide	$O(1)$
2	4	5	7		1	2	3	6	sort	$2T(n/2)$

1	2	2	3	4	5	6	7	merge	$O(n)$

Merge

Merge. Combine two sorted lists into a sorted whole.

<u>Merge(A,p,q,r):</u>

create two new arrays L **and** R

$L \leftarrow A[p..q]$, $R \leftarrow A[q+1..r]$

append ∞ **at the end of** L **and** R

$i \leftarrow 1, j \leftarrow 1$

for $k \leftarrow p$ **to** r

    if $L[i] \leq R[j]$ **then**

        $A[k] \leftarrow L[i]$

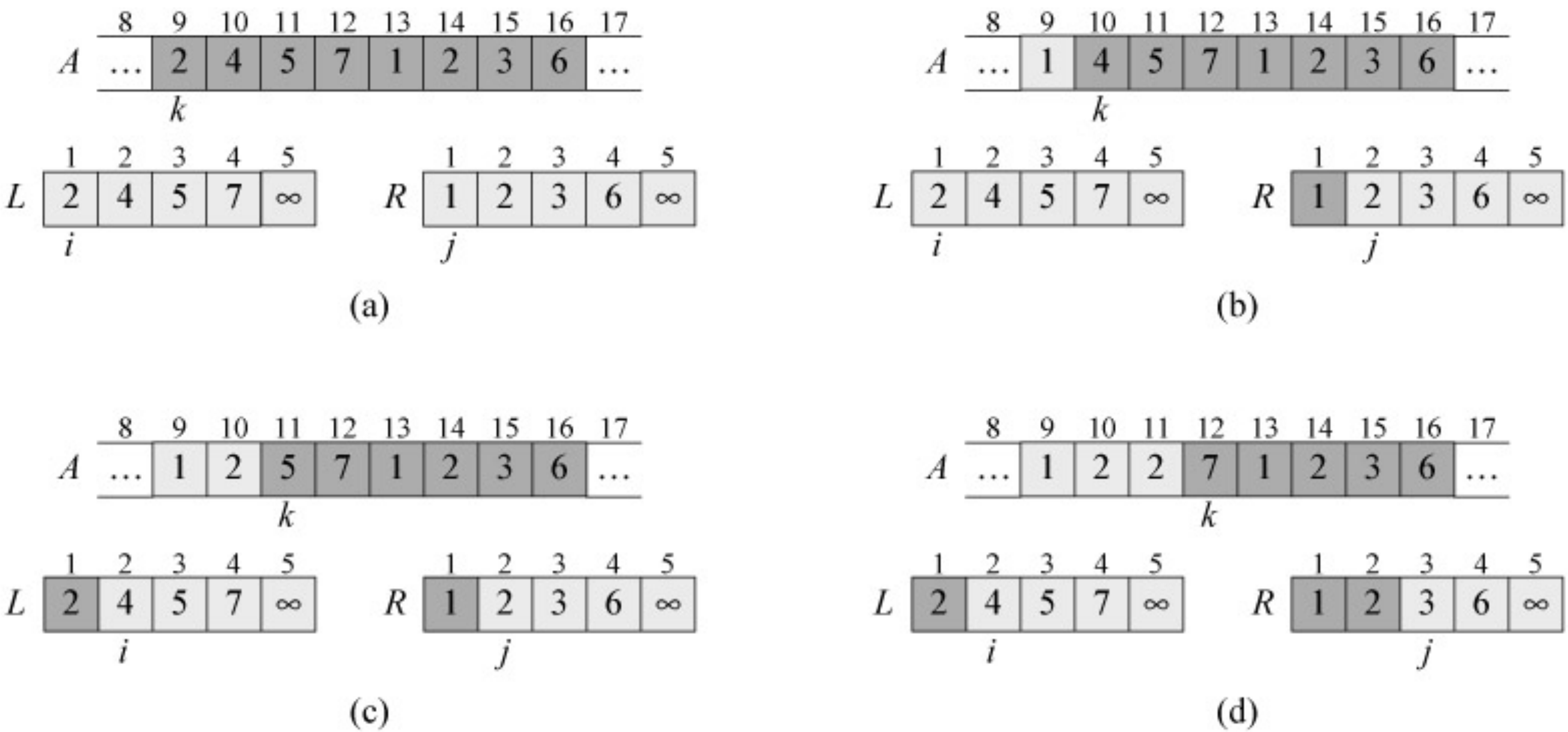
        $i \leftarrow i + 1$

    else

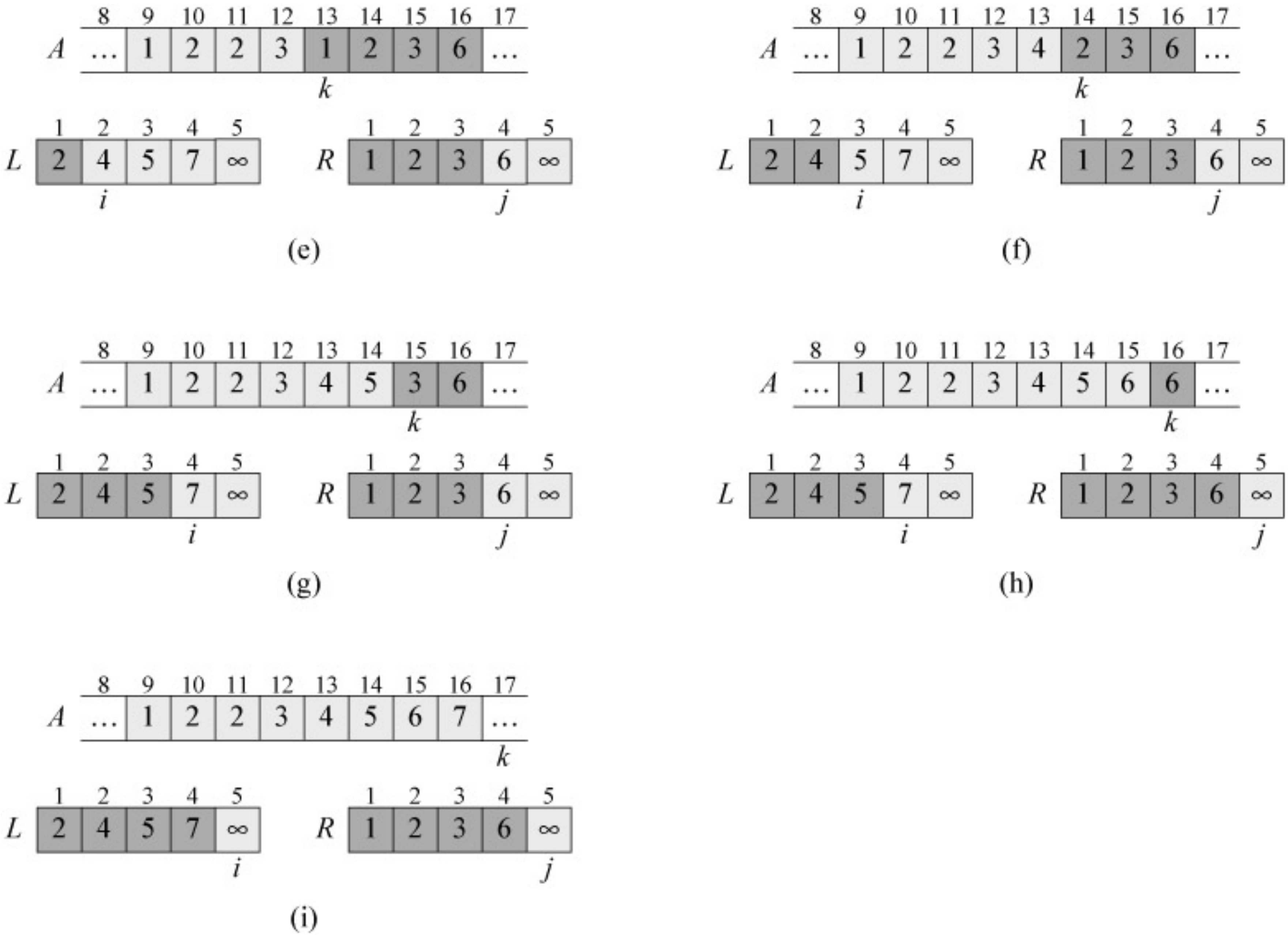
        $A[k] \leftarrow R[j]$

        $j \leftarrow j + 1$

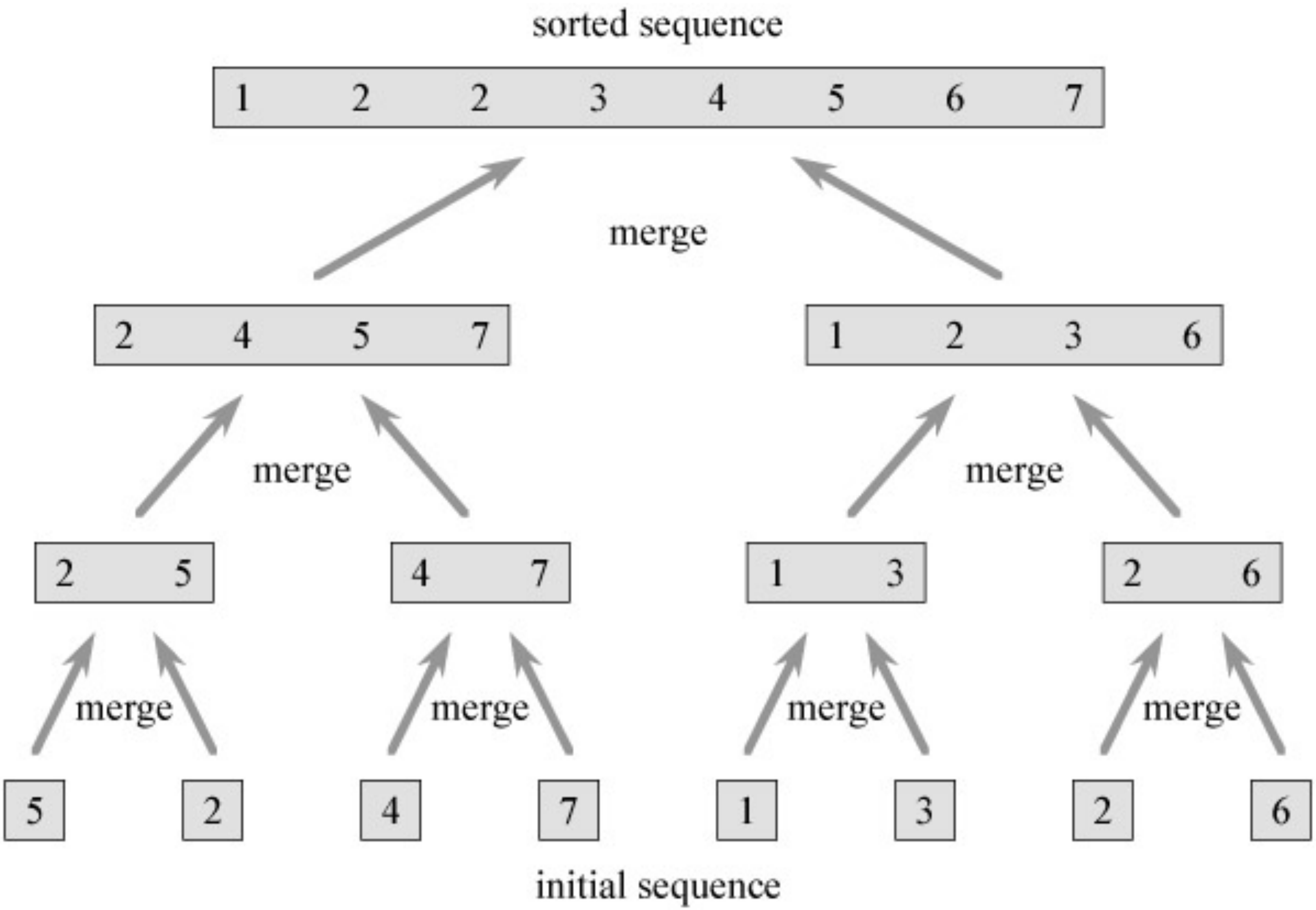
Merge: Example



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 $T(n)$
  - Can also think of this as *rescaling* running time
- 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 increasing polynomial function.

Solve the recurrence

Simplified merge sort recurrence.

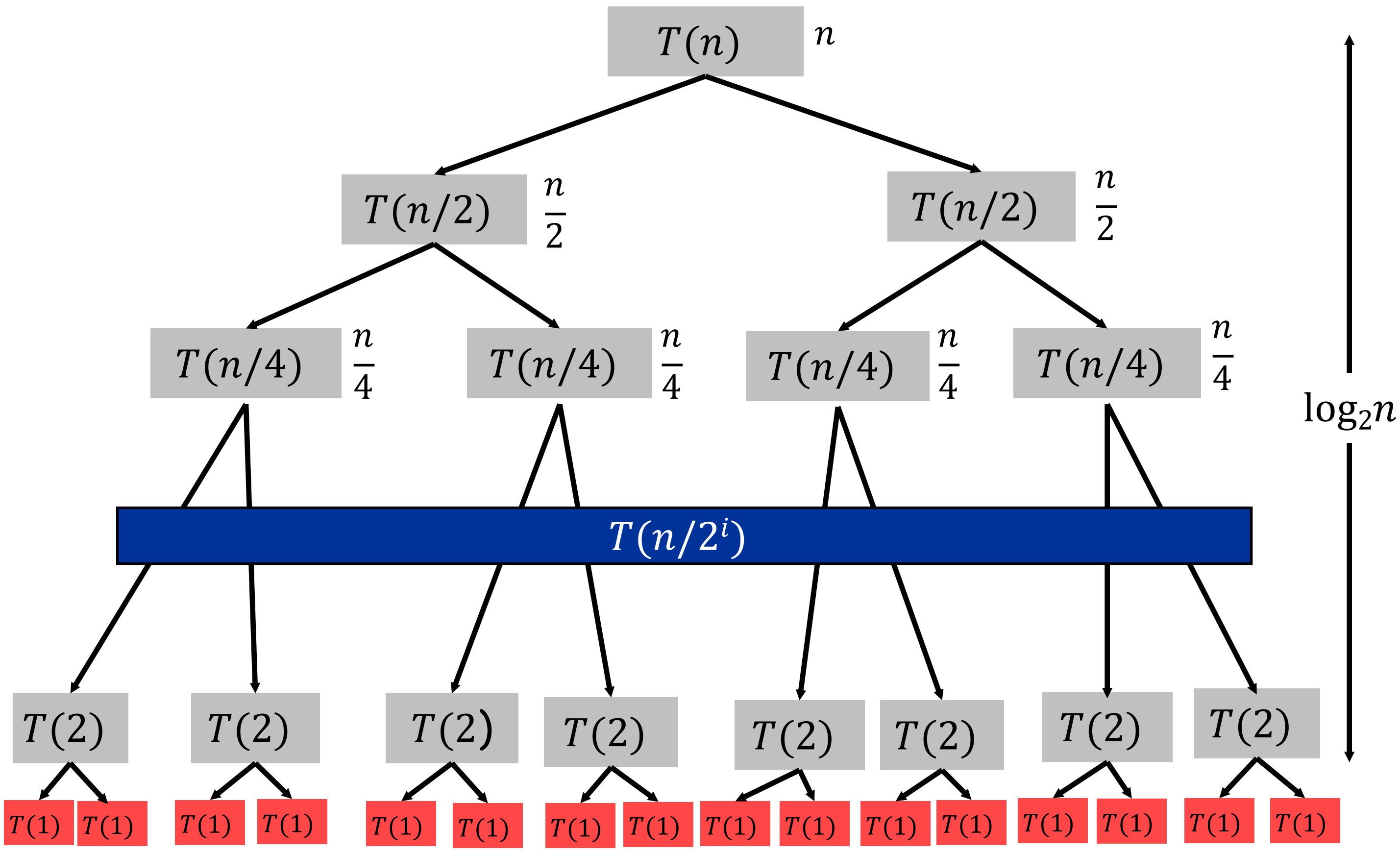
$$T(n) = 2T(n/2) + n, \qquad n > 1$$
$$T(1) = 1$$

$$
\begin{aligned}
T(n) &= 2\,T\left(\frac{n}{2}\right) + n \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \;=\; 2^2 T\left(\frac{n}{2^2}\right) + 2n \\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n \;=\; 2^3 T\left(\frac{n}{2^3}\right) + 3n \\
&= 2^3\left(2T\left(\frac{n}{2^4}\right) + \frac{n}{2^3}\right) + 3n \;=\; 2^4 T\left(\frac{n}{2^4}\right) + 4n \\
&= \cdots \\
&= 2^i\,T\left(\frac{n}{2^i}\right) + in \\
&= \cdots \\
&= 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n\, n \\
&= n\,T(1) + \log_2 n\, n \\
&= n\log_2 n + n
\end{aligned}
$$

Solve the recurrence

Simplified merge sort recurrence.

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



n

$2 \cdot n/2$

$4 \cdot n/4$

$\cdots$

$2^i \cdot n/2^i$

$\cdots$

$2^{\log_2 n} \dfrac{n}{2^{\log_2 n}} = \cdot\ n \cdot 1$

$nT(1) = n$

$\boldsymbol{n \log_2 n + n}$

Running time of merge sort

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

A: Yes

- Since the "merge" step always takes $\Theta(n)$ time no matter what the input is, the algorithm's running time is actually "the same" (up to a constant multiplicative factor), independent of the input.
- Equivalently speaking, every input is a worst case input.
- The whole analysis holds if we replace every O with Ω

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

Inversion Numbers

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

<u>Inversions</u>

3-2, 4-2

Inversion number = 2

Relation to Insertion sort

Theorem: The number of swaps used by Insertion Sort = Inversion Number.

Proof: By induction on the size (n) of the array

Assume Theorem is correct for an array of size n-1.

=> total No. of swaps performed when Insertion Sorting A[1..n-1] is
the inversion # of A[1..n-1].

Let X=A[n]. Remaining work of algorithm is
swaps performed when comparing X to items in A[1..n-1]. This is exactly
the # of items j < n such that A[j] > A[n],
i.e., the # of inversions in which X participates,
Adding these new inversions to the ones in A[1,n-1] gives the full inversion #
of A[1..n].

Q: How can we compute the inversion number?

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

Algorithm 2: Run Ins sort and count the number of swaps – Also $\Theta(n^2)$ time.

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: Simple Combine Step

- Assume array is split into left half (blue) and right (green) half and each is already sorted
- How can we count # inversions where a_i and a_j are in different halves?

L[i] →	3	7	10	14	18	19

R[j] ←	2	11	16	17	23	25
I[j] ←	6	3	2	2	0	0

Count(A, p, q, r)**:**

$L \leftarrow A[p..q], R \leftarrow A[q+1..r]$

L, R already sorted

$i \leftarrow 1, j \leftarrow 1$

$c \leftarrow 0$

While $(i \le p - q + 1)$ **&&** $(j \le r - q)$

(*) **if** $L[i] \le R[j]$ **then**

$i \leftarrow i + 1$

(**) **else**

$\boldsymbol{I}[j] = q - p - i + 2$

$c \leftarrow c + I[j]$

$j \leftarrow j + 1$

Let $\boldsymbol{I}[j]$ = # of inversions of R[j] with blue items

Knowing the $\boldsymbol{I}[j]$ solves the problem

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

When $L[i] > R[j]$ and (**) is called,
the blue items $\le R[j]$ are exactly
the first $i - 1$ blue items

The number of blue items greater than R[j], i.e., the # of inversions of R[j] with blue items is

$$\boldsymbol{I}[j] = q - p - i + 2$$

Counting Inversions: 1st try at Combine

Combine: count blue-green inversions

- (Merge)Sort left and right halves separately.
- Count inversions where a_i and a_j are in different halves.
- Return # blue-inversions + # green inversions + # blue-green inversions

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

Sort: $\Theta(n \log n)$

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

Count: $\Theta(n)$
previous page

$$T(n) = 2T(n/2) + \Theta(n \log n + n) = 2T(n/2) + \Theta(n \log n). \quad n > 1$$

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

Can show that solution is $T(n) = \Theta(n(\log n)^2)$

OK. But this can be improved.

Counting Inversions: 2^{nd} try at Combine

Combine: count blue-green inversions

- Assume each half is already (recursively) sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole to maintain sortedness invariant.
- Return # blue-inversions + # green inversions + # blue-green inversions

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. `[Merge-and-Count]` $A[p..r]$ is sorted.

Post-condition. `[Sort-and-Count]` $A[p..r]$ 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
```