COMP 3711 Design and Analysis of Algorithms

Divide & Conquer - Intro

Divide-and-Conquer intro: Binary search

Main idea of DaC: Solve a problem of size n by breaking it into one or more smaller problems of size less than n. Solve the smaller problems recursively and combine their solutions to solve the large problem.

Example: Binary Search

Input: A sorted array $A[1..n]$ and an element x.

Output: Return the position of x, if it is in A; otherwise output nil.

Idea of binary search : Set $q \leftarrow$ middle of the array. If $x = A[q]$, return q. If $x < A[q]$, search $A[1..q-1]$. If $x > A[q]$, search $A[q+1..n]$.

```
BinarySearch(A,p,r,x):
if p > r then return nil
q ← ⌊(p + r)/2⌋
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 Example

1	2	3	4	5	6	7	8	9	10	
4	7	10	15	19	20	42	54	87	90	(A, 1, 10, 42)
4	7	10	15	19	20	42	54	87	90	$q = A[5] = 19$
4	7	10	15	19	20	42	54	87	90	(A, 6, 10, 42)
4	7	10	15	19	20	42	54	87	90	$q = A[8] = 54$
4	7	10	15	19	20	42	54	87	90	(A, 6, 7, 42)
4	7	10	15	19	20	42	54	87	90	$q = A[6] = 20$
4	7	10	15	19	20	42	54	87	90	(A, 7, 7, 42)
4	7	10	15	19	20	42	54	87	90	(A, 7, 7, 42) FOUND

Analysis of Binary Search

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

Recurrence: With at most two comparisons 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:

$$T(n) = T(n/2) + 2 \quad \text{if } n > 1, \text{ with } T(1) = 1.$$

Solve the recurrence by the expansion method:

$$\begin{aligned}
T(n) &= T(n/2) + 2 \\
&= (T(n/2^2) + 2) + 2 \\
&= T(n/2^2) + 4 \\
&= \ldots\ldots \\
&= T(n/2^i) + 2i \\
&= \ldots. \\
&= T(n/2^{\log_2 n}) + 2\log_2 n \\
&= T(1) + 2\log_2 n
\end{aligned}$$

$$\mathbf{= 1 + 2\log_2 n}$$

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

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

Total number of comparisons: $2 + 2 + .. + 2 + 1 = 2\log_2 n + 1$

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

Exercise 1a Rotated Sorted Array

Let $A[1..n]$ be a sorted array of n distinct numbers that has been rotated $n-k$ steps for some unknown integer $k \in [1, n-1]$. That is, $A[1..k]$ is sorted in increasing order, and $A[k+1..n]$ is also sorted in increasing order, and $A[n] < A[1]$. The following array A is an example of $n = 16$ elements with $k = 10$.

A = [9, 13, 16, 18, 19, 23, 28, 31, 37, 42, 0, 1, 2, 5, 7, 8].

1) Design an $O(\log n)$-time algorithm to find the value of k. ($A[k]$ is the maximum element in the array)

Find-k(A, p, q)
$m \leftarrow \lfloor (p+q)/2 \rfloor$
if $A[m] > A[m+1]$ then return m
if $A[m] \geq A[1]$ then return Find-k($A, m+1, q$)
Else return Find-k($A, p, m-1$)

First call: Find-k($A, 1, n$)

This is similar to binary search: with a constant number of comparisons, we reduce the problem size by half: $T(n) = T\left(\frac{n}{2}\right) + c \Longrightarrow T(n) = O(\log n)$

Exercise 1b Rotated Sorted Array (cont)

2) Design an $O(\log n)$-time algorithm that for any given x, find x in the rotated sorted array, or report that it does not exist.

Find-x(A, x)

$k \leftarrow$ Find-k($A, 1, n$)

if $x \geq A[1]$ then return BinarySearch($A, 1, k, x$)

Else return BinarySearch($A, k+1, n, x$)

Exercise 2a Finding the last 0

You are given an array $A[1..n]$ that contains a sequence of 0 followed by a sequence of 1 (e.g., 0001111111). A contains at least one 0 and one 1.

1) Design an $O(\log n)$-time algorithm that finds the position k of the last 0, i.e., $A[k] = 0$ and $A[k+1] = 1$.

find-k(A, p, r)
$mid \leftarrow \lfloor (p+r)/2 \rfloor$
if $A[mid] = 0$ and $A[mid+1] = 1$, RETURN mid
if $A[mid] = 0$, **find-k**$(A, mid+1, r])$
else **find-k**(A, p, mid)

First call: **find-k**$(A, 1, n)$

Exercise 2b Finding the last 0 (cont)

2) Suppose that k is much smaller than n. Design an $O(\log k)$-time algorithm that finds the position k of the last 0. (you can re-use solution of part 1).

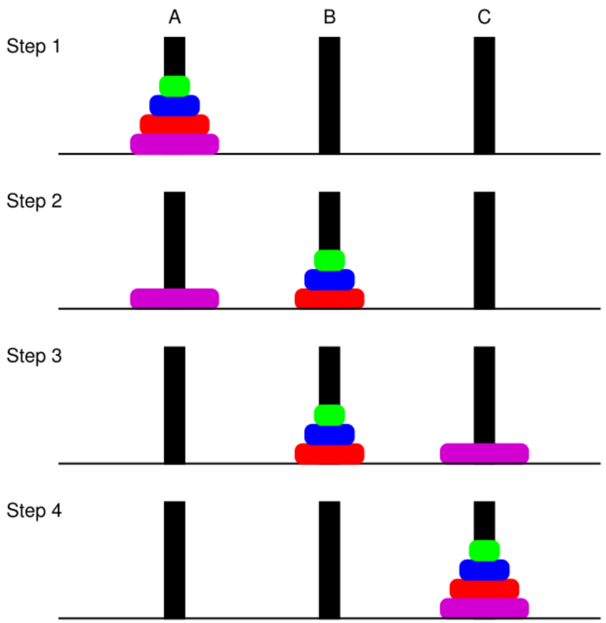
$i \leftarrow 1$
while $A[i] = 0$
    $i \leftarrow \min(2i, n)$
find-k$(A, i/2, i)$

The while loop will stop when it finds a 1. Since each time we double the value of i, the while loop performs $\log k$ iterations. The first 1 occurs somewhere between the positions $A[i/2+1]$ and $A[i]$. To find it, we call **find-k**$(A, i/2\,, i])$, which has cost $\log(k/2) = O(\log k)$. Therefore, the total cost is $O(\log k)$.

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:

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

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

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

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

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

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

General Case

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

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

geometric series

$i = n-1$

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

Exercise 3 Geometric Series

Assume c is a positive constant. Prove that $\sum_{i=0}^{n-1} c^i = \frac{c^n - 1}{c - 1}$

Set $S = \sum_{i=0}^{n-1} c^i = 1 + c + c^2 + \cdots + c^{n-1}$

Then, $c \cdot S = c \cdot \sum_{i=0}^{n-1} c^i = c + c^2 + \cdots + c^n$

$c \cdot S - S = (c - 1) \cdot S = c^n - 1 \rightarrow S = \sum_{i=0}^{n-1} c^i = \frac{c^n - 1}{c - 1}$

For $c > 1$, $\Theta\left(\frac{c^n - 1}{c - 1}\right) = \Theta(c^n)$ (also $\Theta(c^{n-1})$ because $c^n = c \cdot c^{n-1}$)

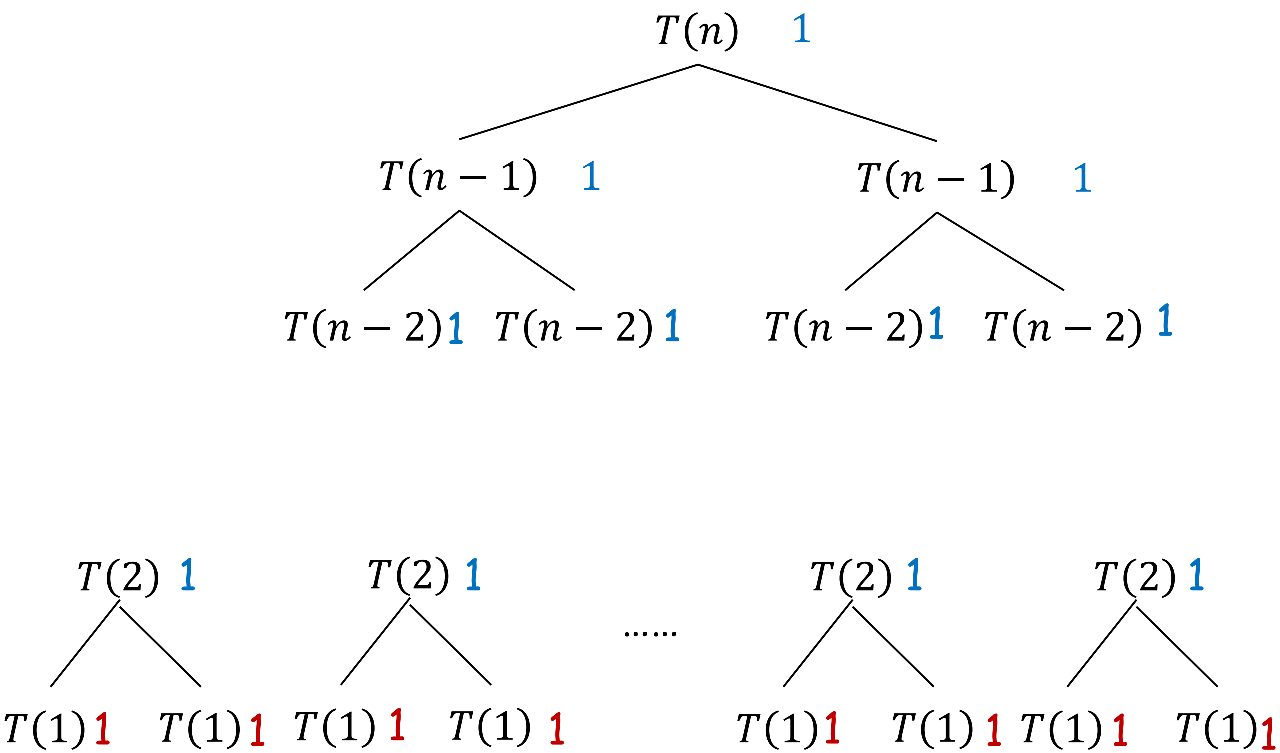
For $c < 1$, $\Theta\left(\frac{c^n - 1}{c - 1}\right) = \Theta\left(\frac{1 - c^n}{1 - c}\right) = O(1)$

In general, the largest term dominates the asymptotic cost:

- $\sum_{i=0}^{n-1} 2^i = 1 + 2 + 2^1 + \cdots + 2^{n-1} = \Theta(2^n)$
- $\sum_{i=0}^{n-1} 1/2^i = 1 + 1/2 + 1/2^2 + \cdots 1/2^{n-1} = O(1)$

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: $1 + 2 + 2^2 + \cdots 2^{n-2} + 2^{n-1} = \Theta(2^n)$($Geometric\ Series$)
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.

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**($A, 1, n$)

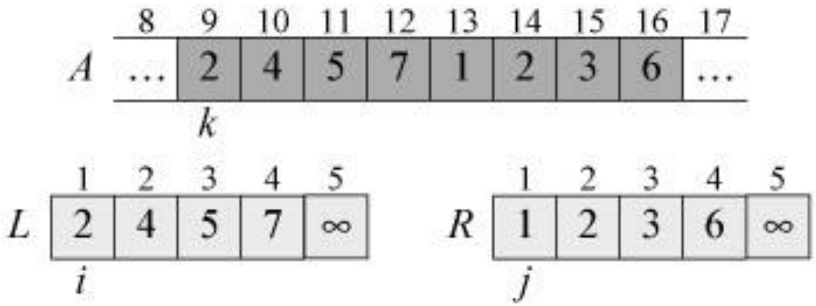
5	2	4	7	1	3	2	6		
5	2	4	7	1	3	2	6	divide	$O(1)$
2	4	5	7	1	2	3	6	sort	$2T(n/2)$
1	2	2	3	4	5	6	7	merge	$O(n)$

Merge

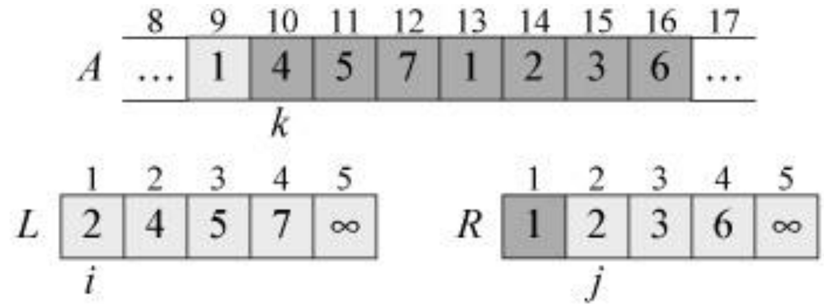
Merge. Combine two sorted lists into a sorted whole.

```
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
```

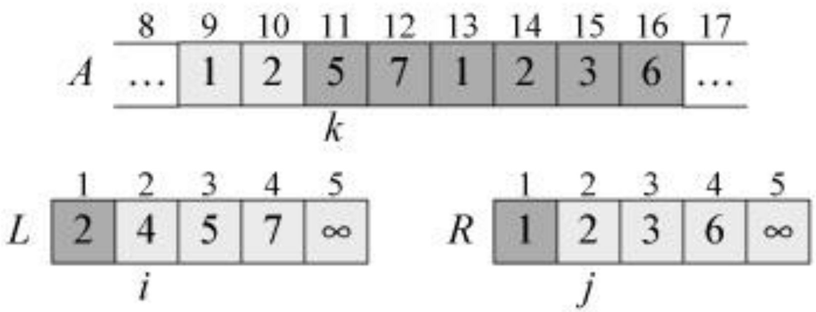
Merge: Example



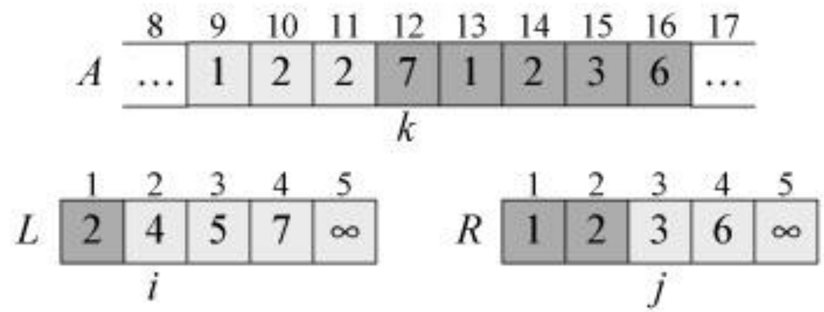
(a)



(b)

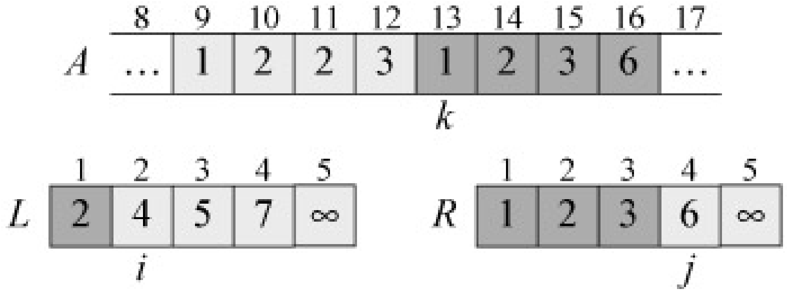


(c)

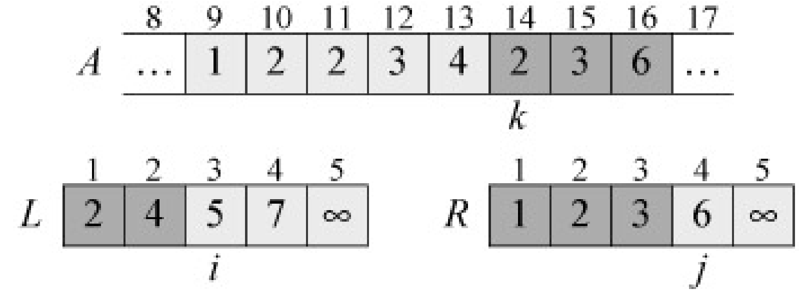


(d)

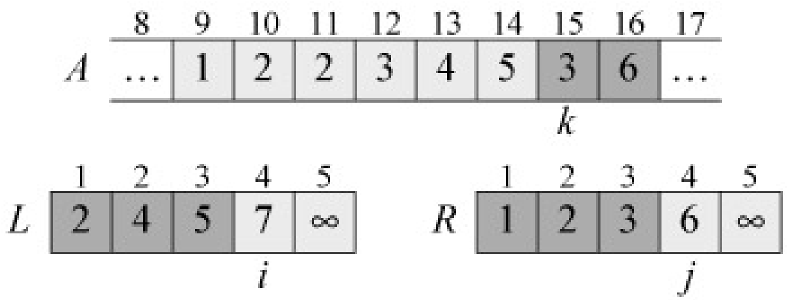
Merge: Example



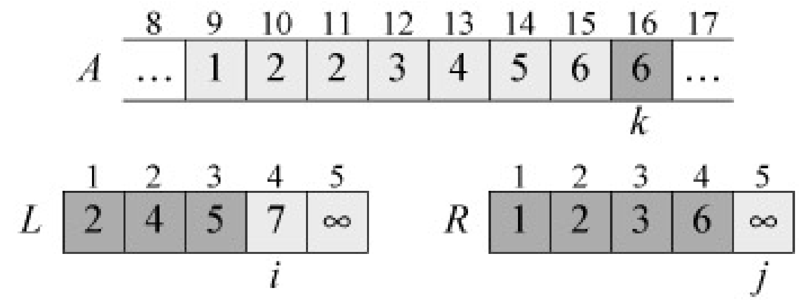
(e)



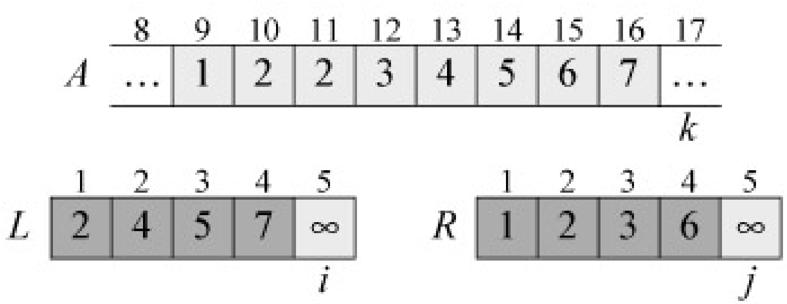
(f)



(g)



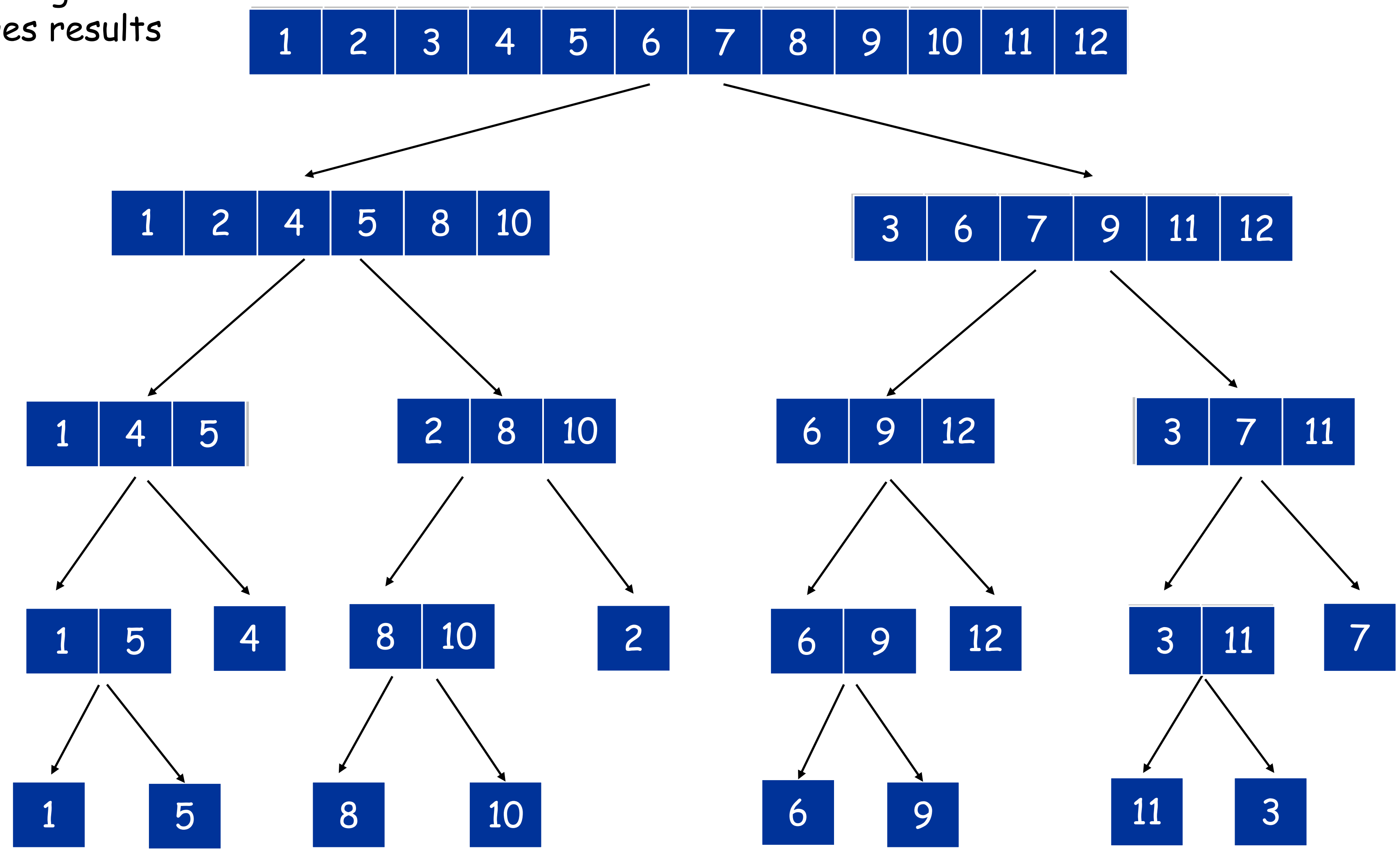
(h)



(i)

Splits each array
into left and right
Sorts Left
Sorts Right
Merges results

Mergesort: 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 a big-O upper bound of $T(n)$
- Replace $O(n)$ with n, replace $O(1)$ with 1
  - since we are interested in a big-O 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 a big-O upper bound of $T(n)$
  - for any n, let n' be the smallest power of 2 such that $n' \ge n$,

    => $T(n) \le T(n') \le T(2n) = O(T(n))$,

    as long as $T(n)$ is a increasing polynomial function.

Solve the recurrence

Simplified merge sort recurrence.

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

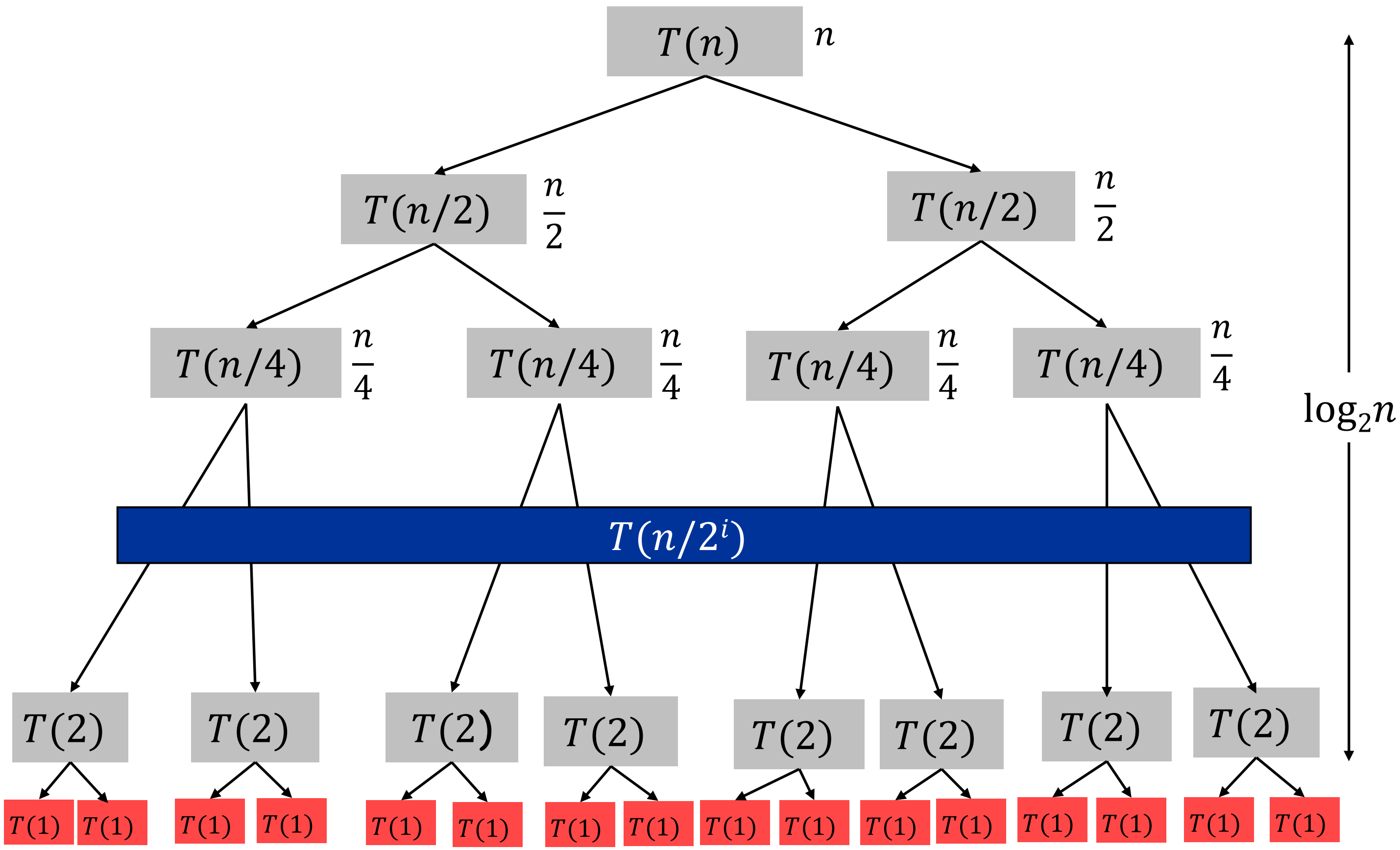
$$\begin{aligned}
T(n) &= 2\,T\left(\frac{n}{2}\right) + n \\
&= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \;=\; 2^2 T\left(\frac{n}{2^2}\right) + 2n \\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n \;=\; 2^3 T\left(\frac{n}{2^3}\right) + 3n \\
&= 2^3\left(2T\left(\frac{n}{2^4}\right)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

Solve the recurrence

$$2^{\log_2 n - 1} = \frac{n}{2}$$

Simplified merge sort recurrence.

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



$$n$$
$$2 \cdot n/2 = n$$
$$4 \cdot n/4 = n$$
$$\cdots$$
$$2^i \cdot n/2^i = n$$
$$\cdots$$
$$2^{\log_2 n - 1} \frac{n}{2^{\log_2 n - 1}} = n$$
$$nT(1) = n$$

$$\mathbf{n(\log_2 n + 1)}$$

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

- 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 $\Theta(n \log n)$.