CS302
Design and Analysis of Algorithm

Week-3b

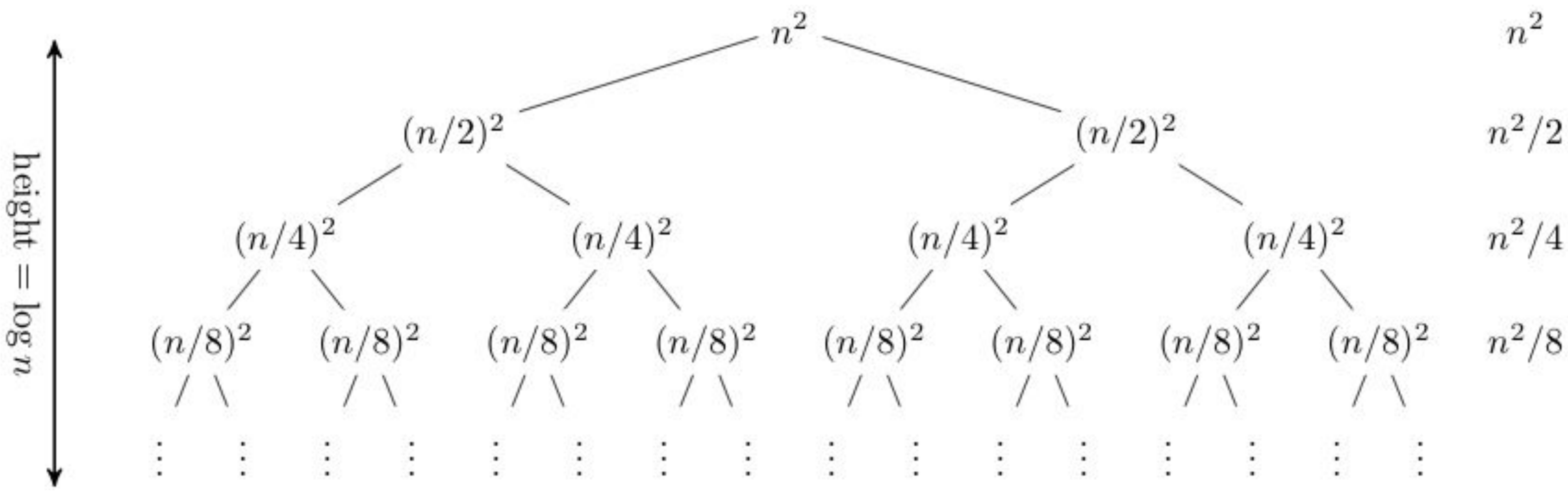
Recurrence Relations

Home Task: Do it yourself

,Solve $T(n) = 3T(n/4) + cn^2$
.where $c > 0$ is constant

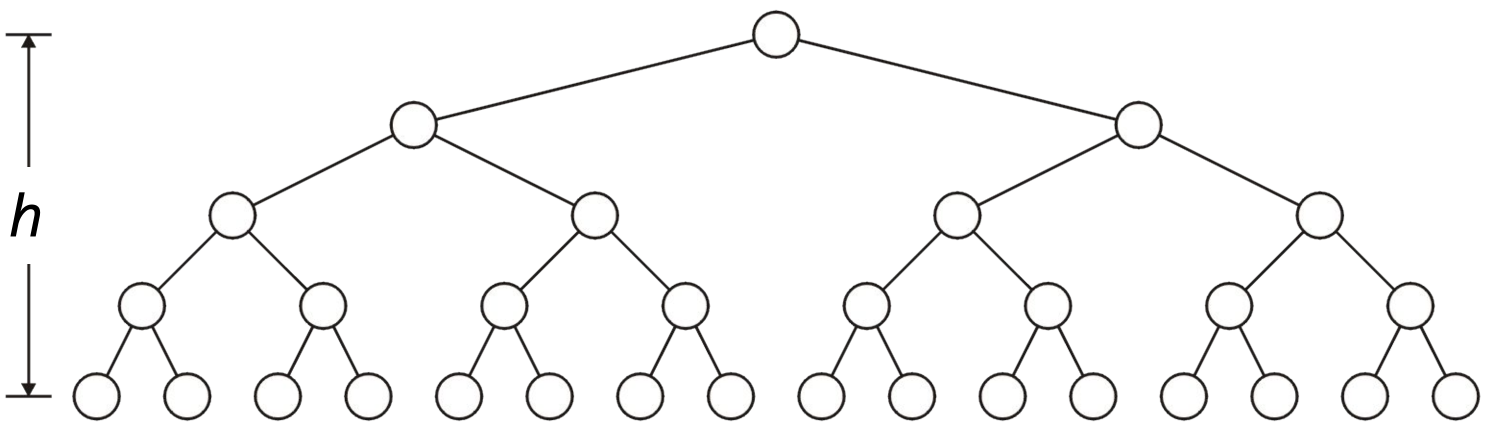
Home Task: Do it yourself

Solve $T(n) = 2T(n/2) + n^2$



Binary Tree

- A perfect binary tree with height h has 2^h leaf nodes
- A perfect binary tree of height h has $2^{h+1} - 1$ nodes
  - **Number of leaf nodes: $L = 2^h$,and with n branch factor = n^h**
  - Number of internal nodes: $2^h - 1$
  - Total number of nodes: $2L-1 = 2^{h+1} - 1$



Master Method

(Save our effort)

When the recurrence is in a special form, we can apply the Master Theorem to solve the recurrence immediately

The Master Theorem has 3 cases …

When not to use

- You cannot use the Master Theorem if
  - $T(n)$ is not monotone, e.g. $T(n) = \sin(x)$
  - $f(n)$ is not a polynomial, e.g., $T(n)=2T(n/2)+2^n$
  - b cannot be expressed as a constant.

Why to use

- measure of algorithm efficiency
- has a big impact on running time.
- Big-O notation is used.
- To deal with n items, time complexity can be $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, even $O(n^n)$.

Master Theorem Simplest version

- Let T(n) be <u>a monotonically increasing</u> function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$
$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are 3 cases:

1. If $f(n) = O(n^{\log_b a - s})$ for some constant $s > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + s})$ with $s > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

The Master Theorem

- if T(n) = aT(n/b) + f(n)   where a ≥ 1 & b > 1
- then

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \lg n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \& \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

Master Theorem Updated

The Master Theorem applies to recurrences of the following form:

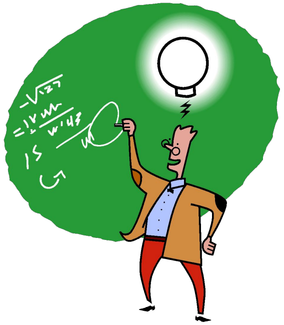
$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ and $k \geq 0$ and p can be any real number and f (n) is an asymptotically positive function and $f(n) = \Theta(n^k \log^p n)$. There are 3 cases:

1. If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $a = b^k$, then
    1. If $p > -1$ then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$.
    2. If $p = -1$ then $T(n) = \Theta(n^{\log_b a} \log\log n)$.
    3. If $P < -1$ then $T(n) = \Theta(n^{\log_b a})$.

a. If $a < b^k$, then
    a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$.
    b. If $p < 0$ then $T(n) = \Theta(n^k)$.

Master Method, Example 1



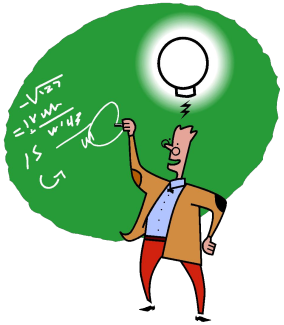
- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- Example

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

Solution:
a=4, b=2
f(n)=n
$\log_b a=2$
so case 1 says T(n) is $\Theta(n^2)$.

Master Method, Example 2



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

- Example: $$T(n) = 2T(n/2) + n$$

  Solution:

  a=2, b=2

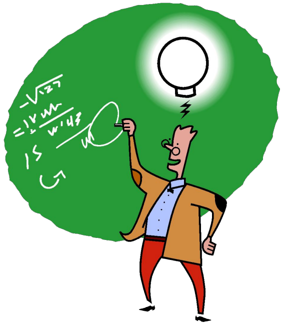
  $\log_b a=1$

  f(n)= n

  N power matched

  so case 2 says T(n) is Θ(n log n).

Master Method, Example 3



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- Example:

$$T(n) = T(n/3) + n \log n$$

Solution:

a=1

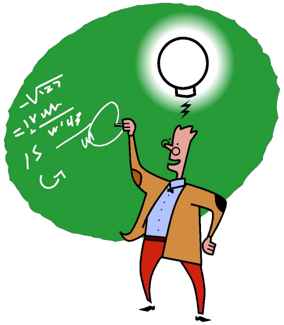
b=3

$\log_b a=0$

f(n)=nlogn

so case 3 says T(n) is Θ(n log n).

Master Method, Example 4



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- Example: $$T(n) = 8T(n/2) + n^2$$

Solution:

a=8

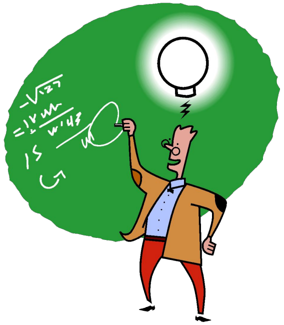
b=2

$f(n)=n^2$

$\log_b a=3$

so case 1 says T(n) is $\Theta(n^3)$.

Master Method, Example 5



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- Example:

$$T(n) = 9T(n/3) + n^3$$

Solution:

a=9

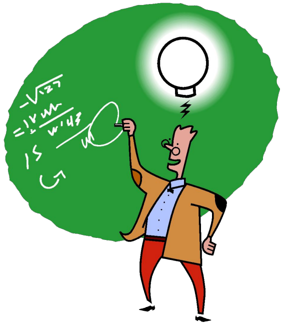
b=3

$f(n)=n^3$

$\log_b a=2$

so case 3 says T(n) is $\Theta(n^3)$.

Master Method, Example 6



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

- Example: $$T(n) = T(n/2) + 1$$

- **Solution:**
  a=1
  b=2
  f(n)=1
  $\log_b a=0$
  so case 2 says T(n) is $\Theta(\log n)$.

Merge sort

MERGE-SORT $A[1 \,.\,.\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\, 1 \,.\,.\, \lceil n/2 \rceil \,]$ and $A[\, \lceil n/2 \rceil + 1 \,.\,.\, n \,]$.
3. "***Merge***" the 2 sorted lists.

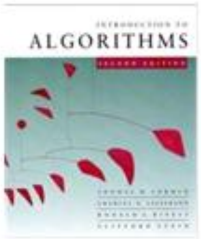
Key subroutine: **MERGE**

Analyzing merge sort

$T(n)$	MERGE-SORT $A[1\ ..\ n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\ ..\ \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil+1\ ..\ n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

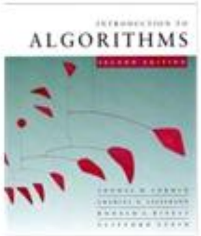
$T(n)=2T(n/2) +cn$ *(recall previous lecture)*

The best and worst case running time will be $O(n \log n)$.



Merging two sorted arrays

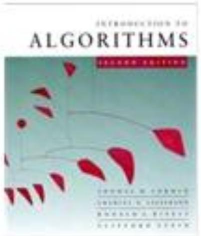
20	12
13	11
7	9
2	1



Merging two sorted arrays

20	12
13	1
7	1 9
2	1

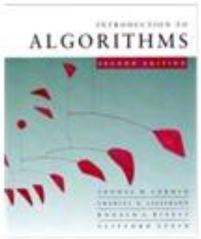
1



Merging two sorted arrays

20	12		20	12
13	1		13	1
7	1 9		7	1 9
2	1		2	

1

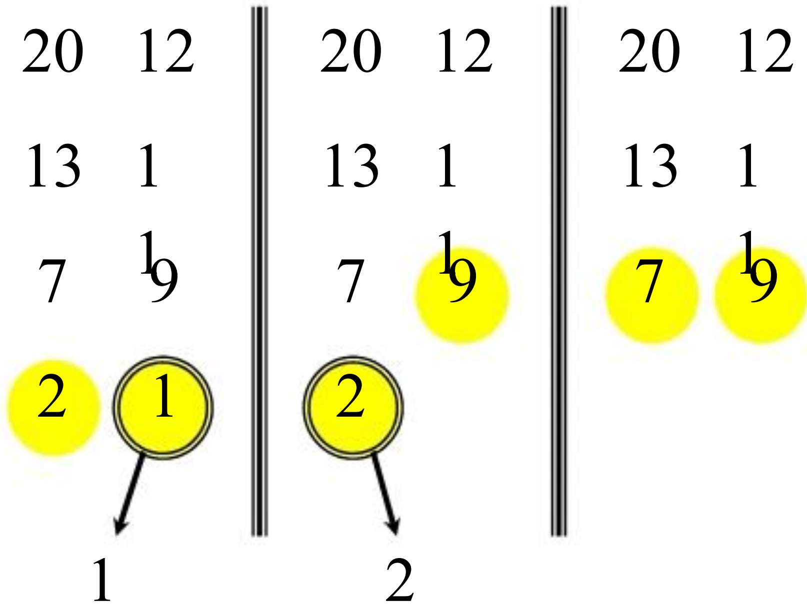


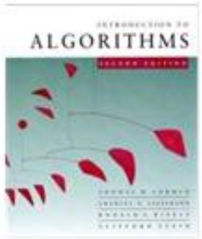
Merging two sorted arrays

20	12	‖	20	12
13	1	‖	13	1
7	1 9	‖	7	1 9
2	1	‖	2	

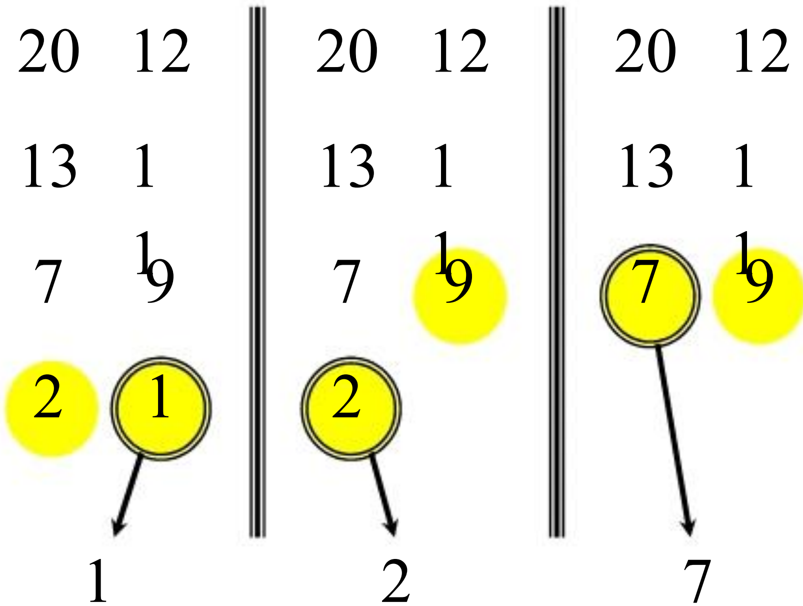
1 2

Merging two sorted arrays



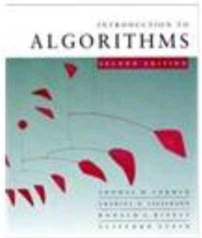


Merging two sorted arrays

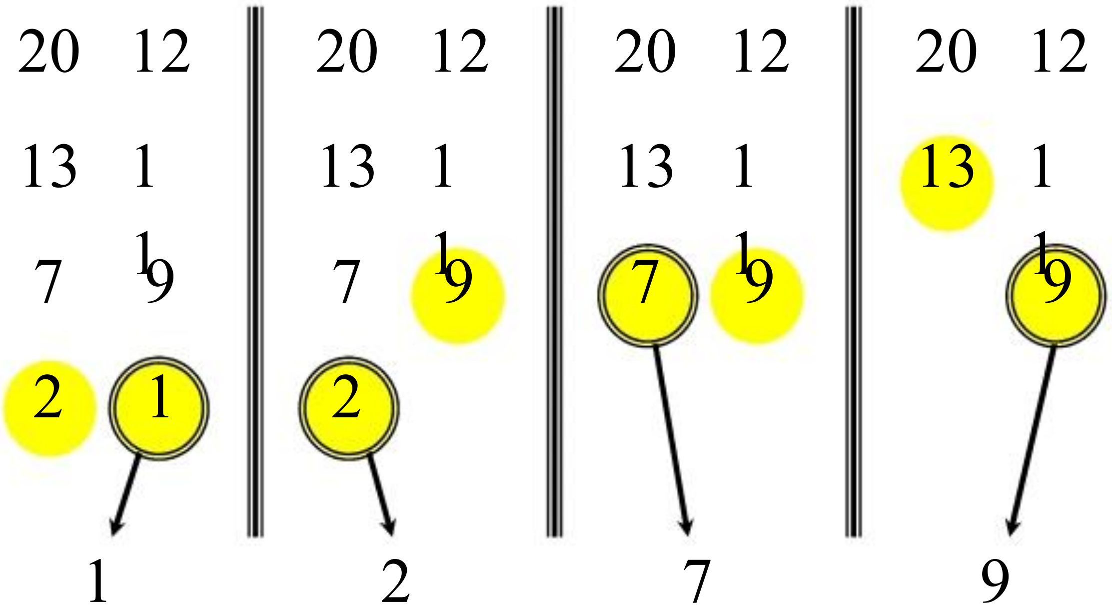


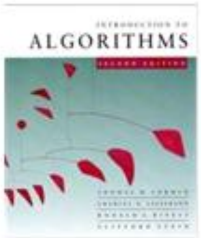
Merging two sorted arrays

20	12	20	12	20	12	20	12
13	1	13	1	13	1	13	1
7	1 9	7	1 9	7	1 9		1 9
2	1	2					
1		2		7			

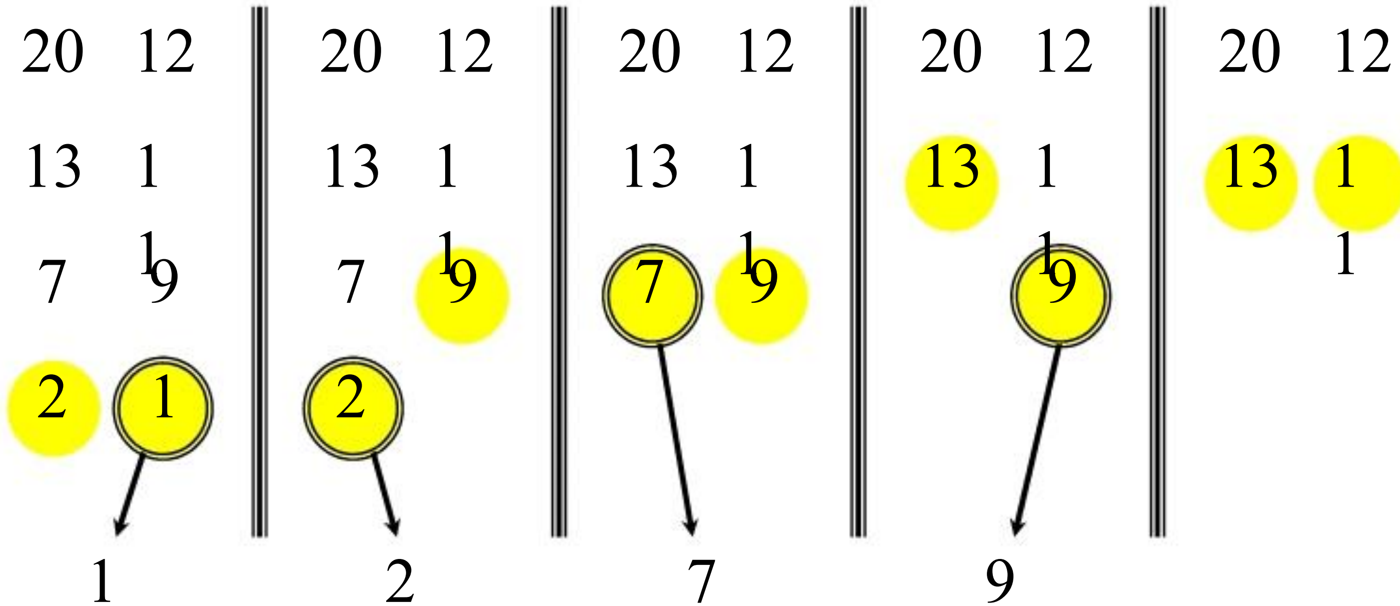


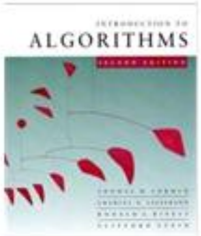
Merging two sorted arrays



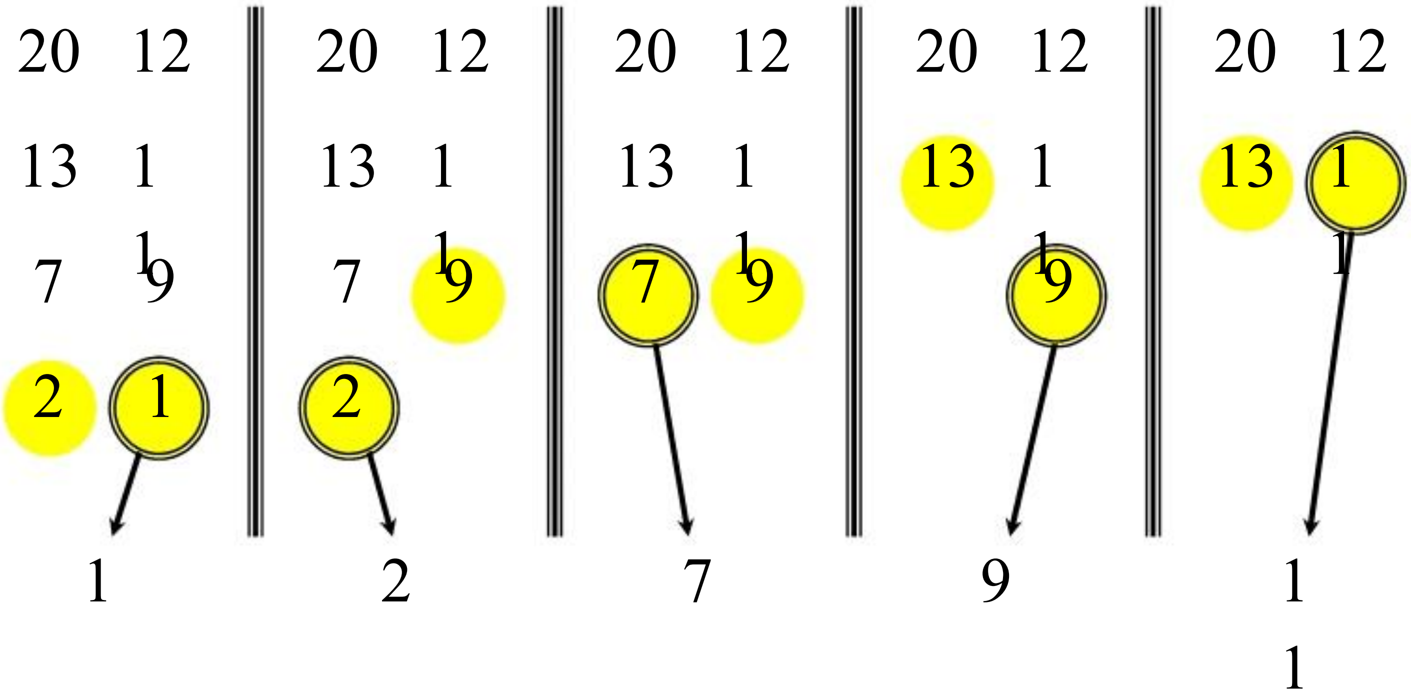


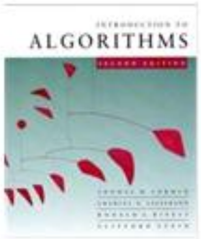
Merging two sorted arrays



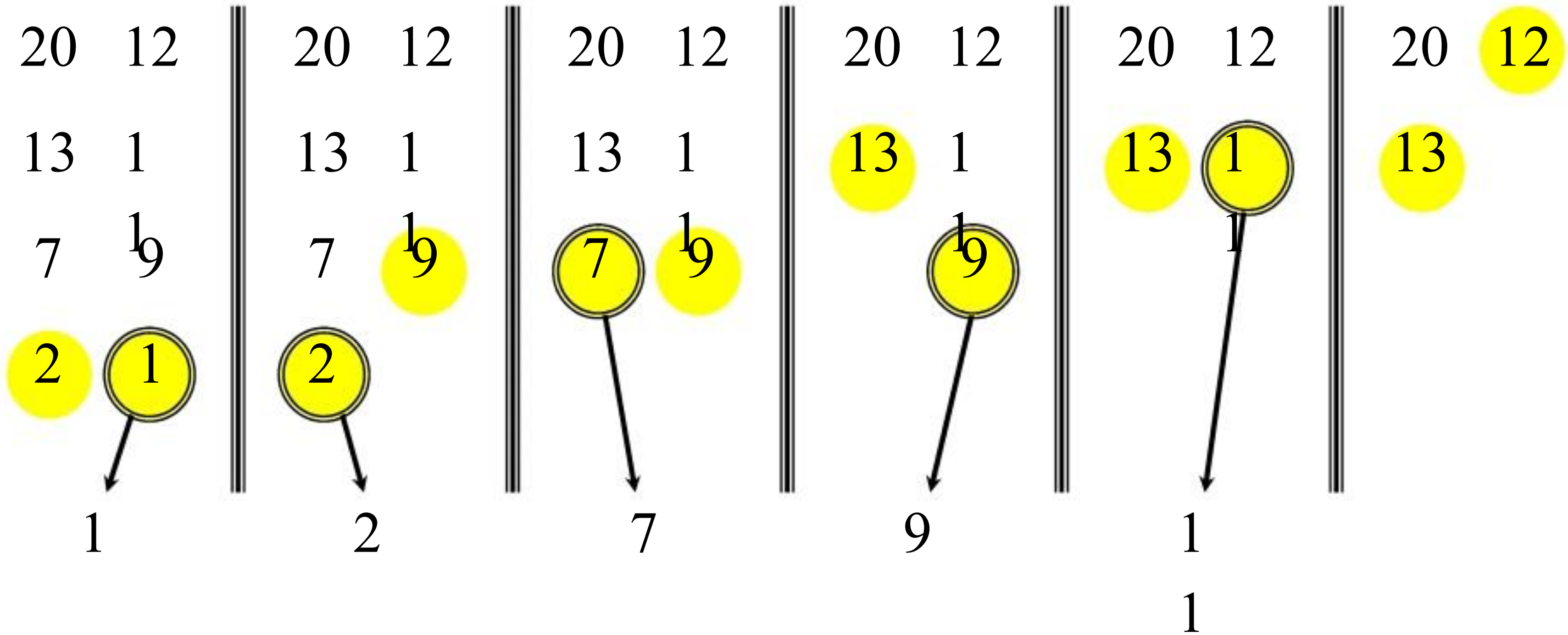


Merging two sorted arrays

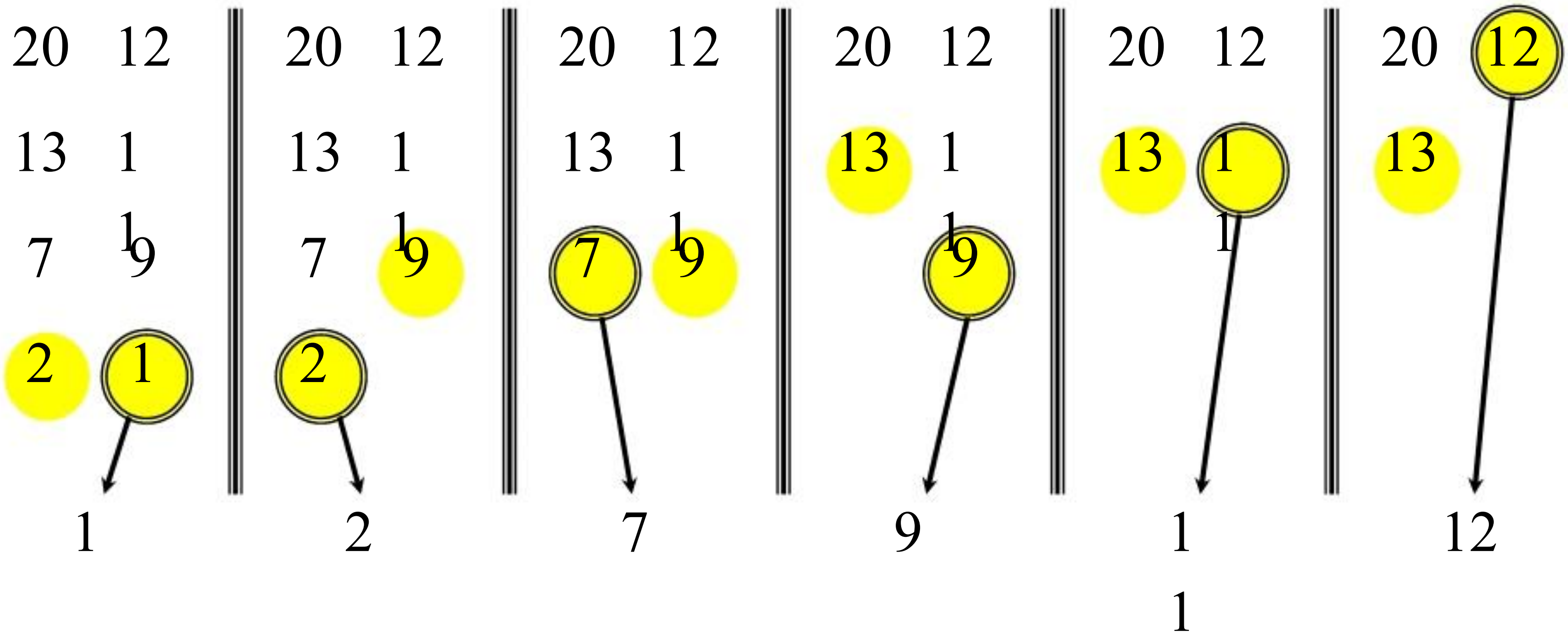


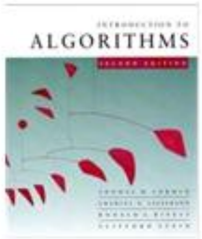


Merging two sorted arrays

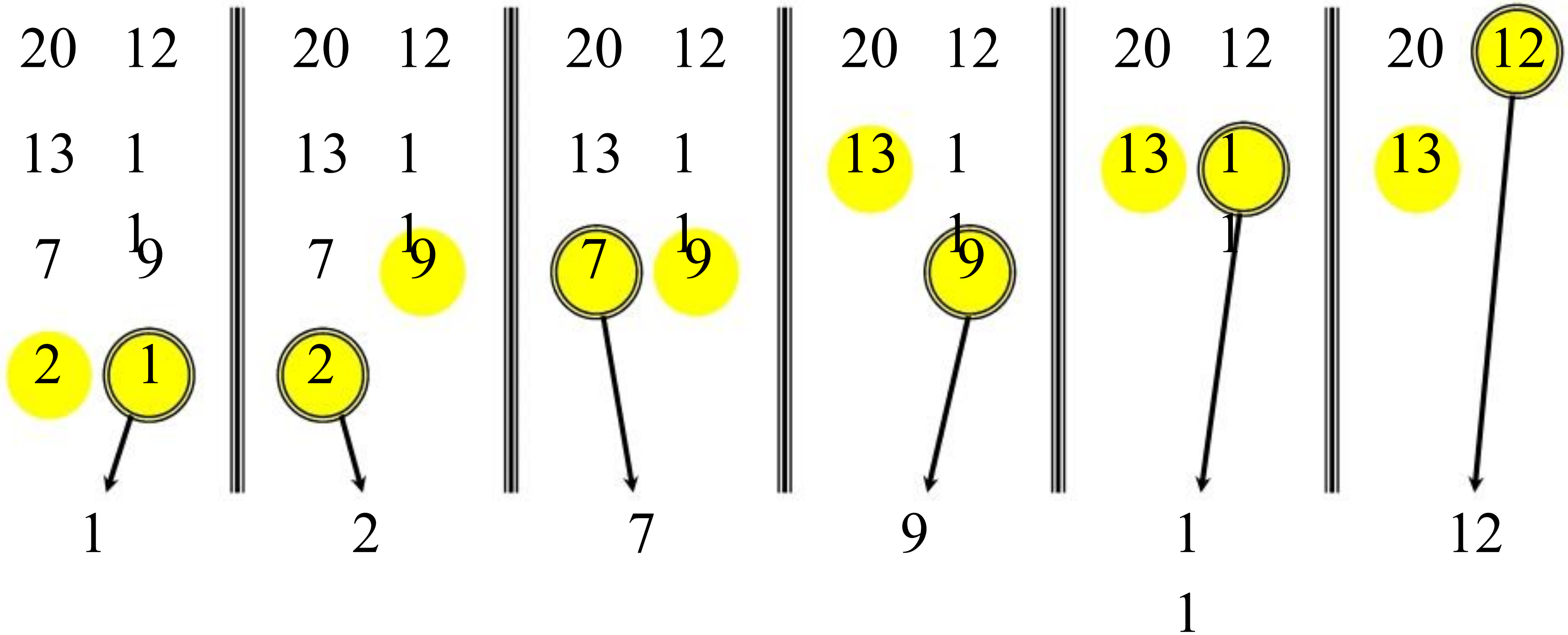


Merging two sorted arrays





Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

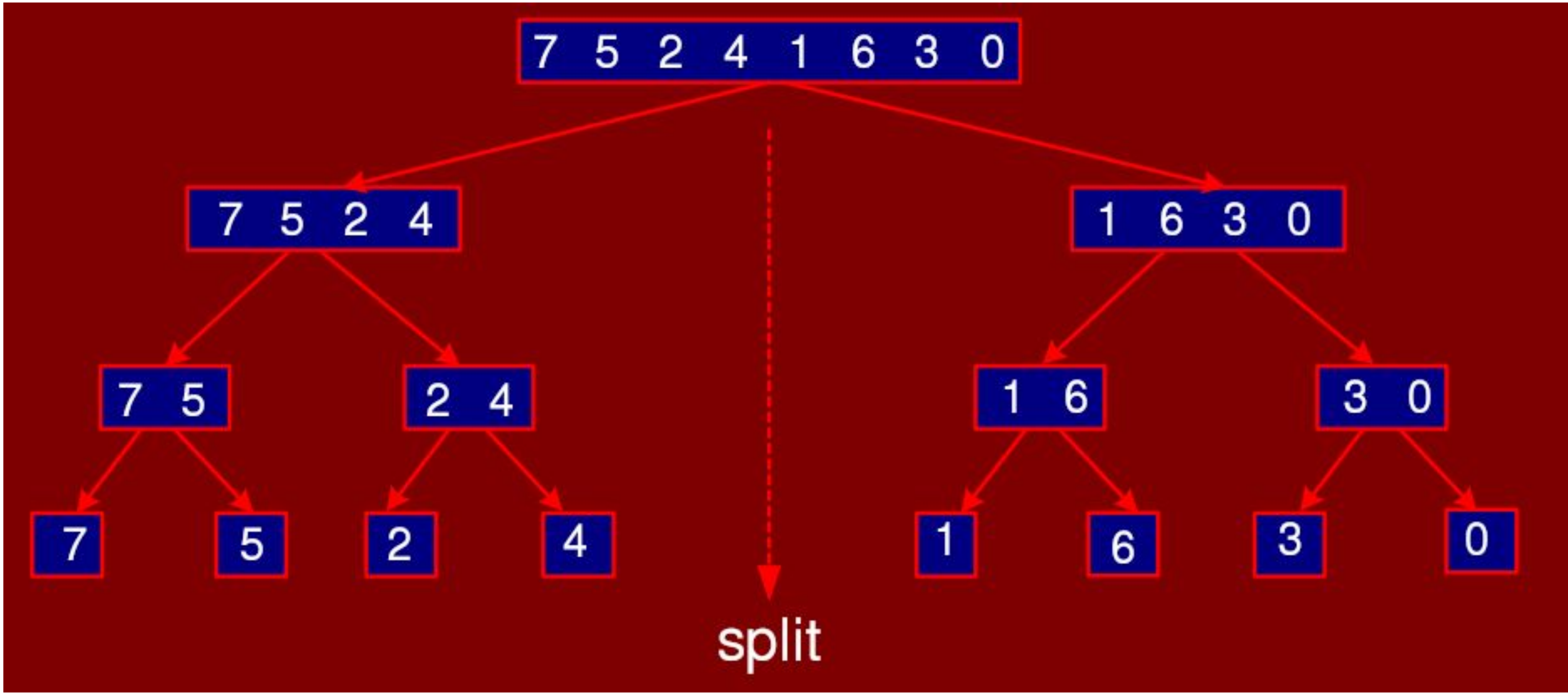
Merge Sort

- (Divide:) split A down the middle into two subsequences, each of size roughly n/2
- (Conquer:) sort each subsequence by calling merge sort recursively on each.
- (Combine:) merge the two sorted subsequences into a single sorted list

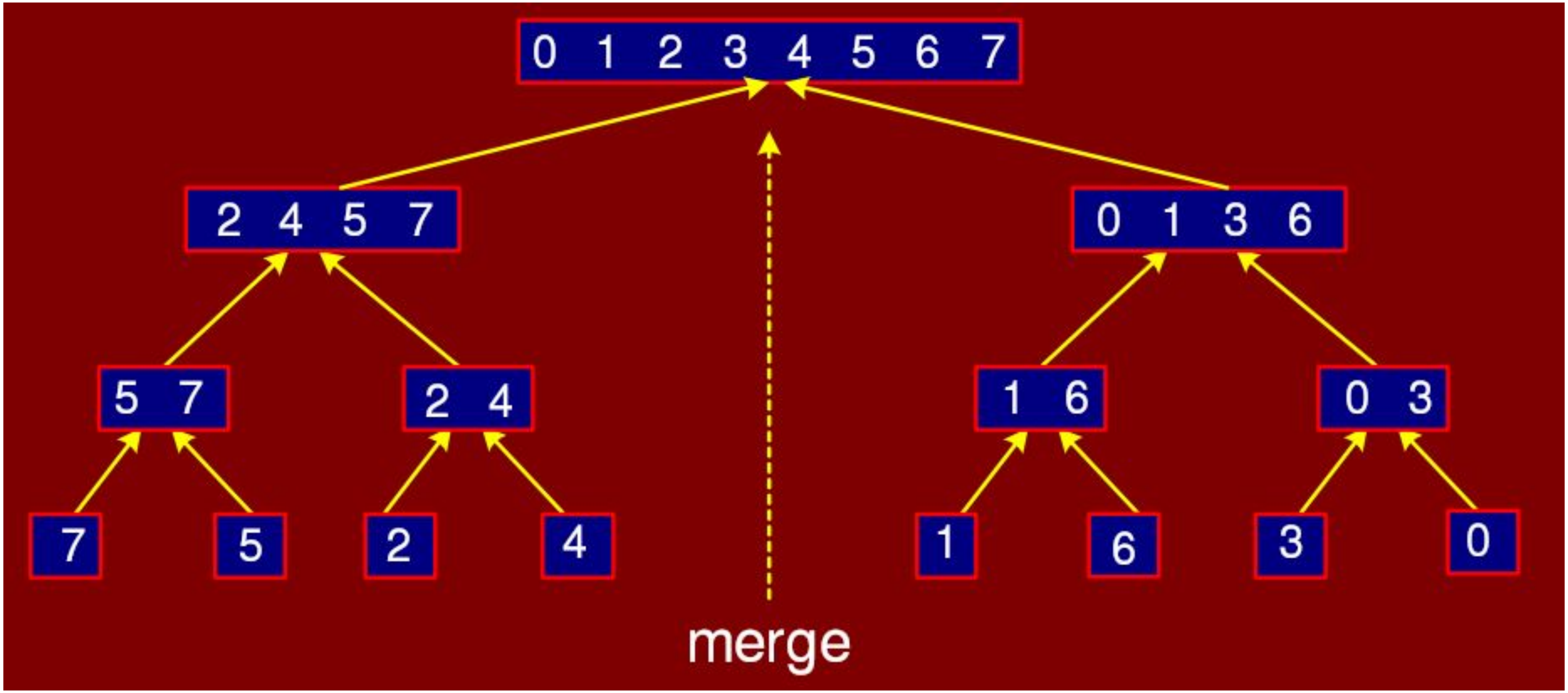
Merge Sort

```
MERGE-SORT( array A, int p, int r)
if (p < r)
   then
        q ← (p + r)/2
        MERGE-SORT(A, p, q)         // sort A[p..q]
        MERGE-SORT(A, q + 1, r) // sort A[q + 1..r]
        MERGE(A, p, q, r)    // merge the two pieces
```

Split part



Merge Part



Merge Sort

- The fundamental operation in this algorithm is merging two sorted lists.
- Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.
- The basic merging algorithm takes
  - two input arrays: *a* and *b*,
  - an output array: *c*
  - three counters: *aptr*, *bptr*, and *cptr*,
    - which are initially set to the beginning of their respective arrays.
- The smaller of *a*[*aptr*] and *b*[*bptr*] is copied to the next entry in *c*, and the appropriate counters are advanced.
- When either input list is exhausted, the remainder of the other list is copied to *c*.

Merge Sort

```
void m_sort( input_type a[], input_type tmp_array[ ], int left, int right )
{
int center;
if( left < right )
  {
  center = (left + right) / 2;
  m_sort( a, tmp_array, left, center );
  m_sort( a, tmp_array, center+1, right );
  merge( a, tmp_array, left, center+1, right );
  }
}
```

Calculate the **centre** index of the input list

Recursively call the **m_sort** procedure
for t

Recursively call the **m_sort** procedure
for the **right-half** of the input data

Merge the two sorted lists

Merge Sort Example (recursive Function Calls)

Mergesort(a, 8)

m_sort(a, tmp_array, 0, 7)

62 (0)	58 (1)	55 (2)	10 (3)	45 (4)	44 (5)	6 (6)	90 (7)

Merge(a, tmp_array, 0, 4, 7)

m_sort(a, tmp_array, 0, 3)

m_sort(a, tmp_array, 4, 7)

62 (0)	58 (1)	55 (2)	10 (3)

45 (4)	44 (5)	6 (6)	90 (7)

62 (0)	58 (1)

55 (2)	10 (3)

45 (4)	44 (5)

6 (6)	90 (7)

m_sort(0, 1)

m_sort(2, 3)

m_sort(4, 5)

m_sort(6, 7)

Merge(a, tmp_array, 0, 2, 3)

Merge(a, tmp_array, 4, 6, 7)

62 (0)	58 (1)	55 (2)	10 (3)	45 (4)	44 (5)	6 (6)	90 (7)

m_sort(0,0)	m_sort(1,1)	m_sort(2,2)	m_sort(3,3)	m_sort(4,4)	m_sort(5,5)	m_sort(6,6)	m_sort(7,7)

Merge(0, 1, 1)

Merge(2, 3, 3)

Merge(4, 5, 5)

Merge(6, 7, 7)

Merge Sort Example (Merging process)

Mergesort(a, 8)

m_sort(a, tmp_array, 0, 7)

6 (0)	10 (1)	44 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

Merge(a, tmp_array, 0, 4, 7)

m_sort(a, tmp_array, 0, 3) | m_sort(a, tmp_array, 4, 7)

62 (0)	58 (1)	55 (2)	10 (3)	45 (4)	44 (5)	6 (6)	90 (7)

62 (0)	58 (1)	55 (2)	10 (3)	45 (4)	44 (5)	6 (6)	90 (7)

m_sort(0, 1) | m_sort(2, 3) | m_sort(4, 5) | m_sort(6, 7)

10 (0)	55 (1)	58 (2)	62 (3)	6 (4)	44 (5)	45 (6)	90 (7)

Merge(a, tmp_array, 0, 2, 3) | Merge(a, tmp_array, 4, 6, 7)

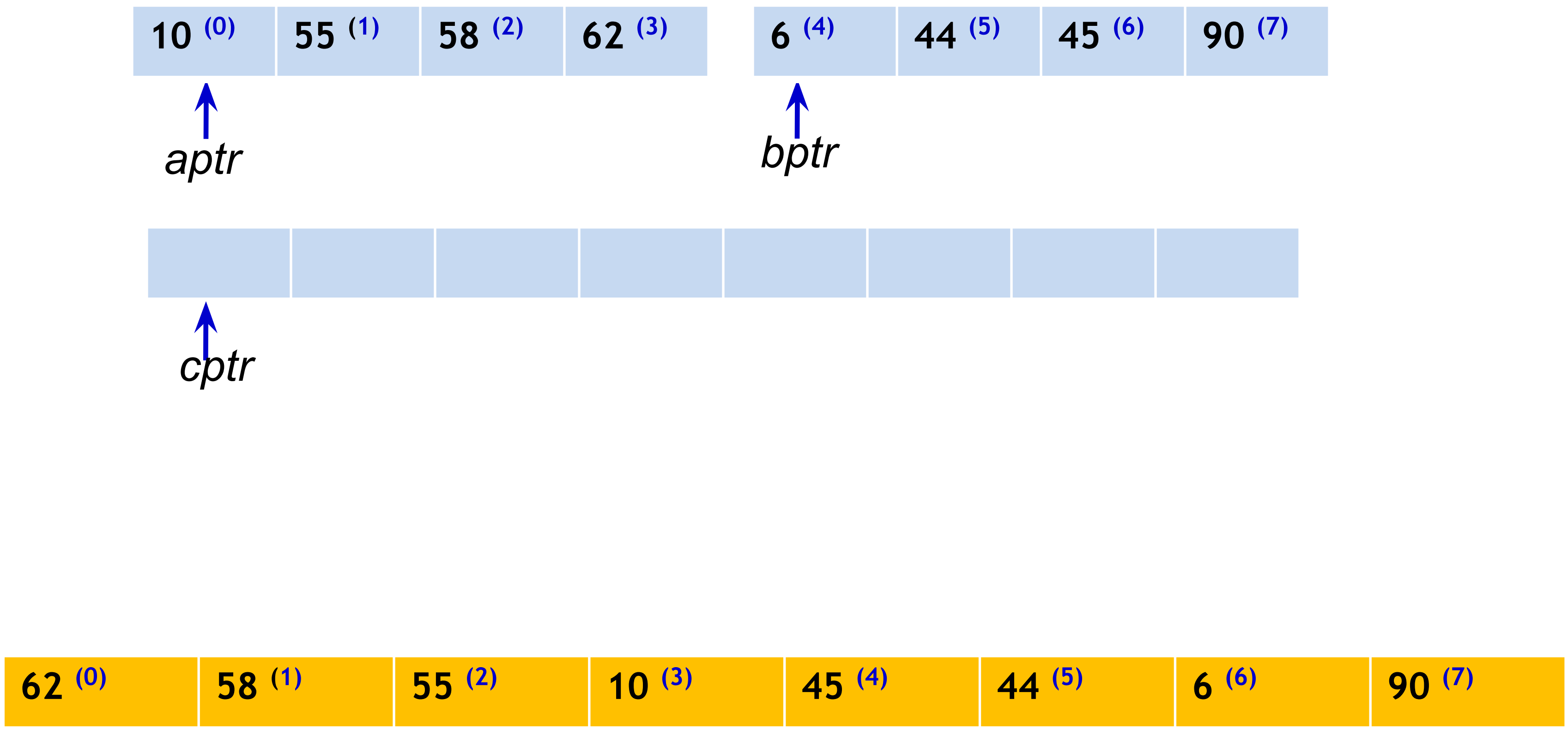
62 (0)	58 (1)	55 (2)	10 (3)	45 (4)	44 (5)	6 (6)	90 (7)

m_sort(0,0)	m_sort(1,1)	m_sort(2,2)	m_sort(3,3)	m_sort(4,4)	m_sort(5,5)	m_sort(6,6)	m_sort(7,7)

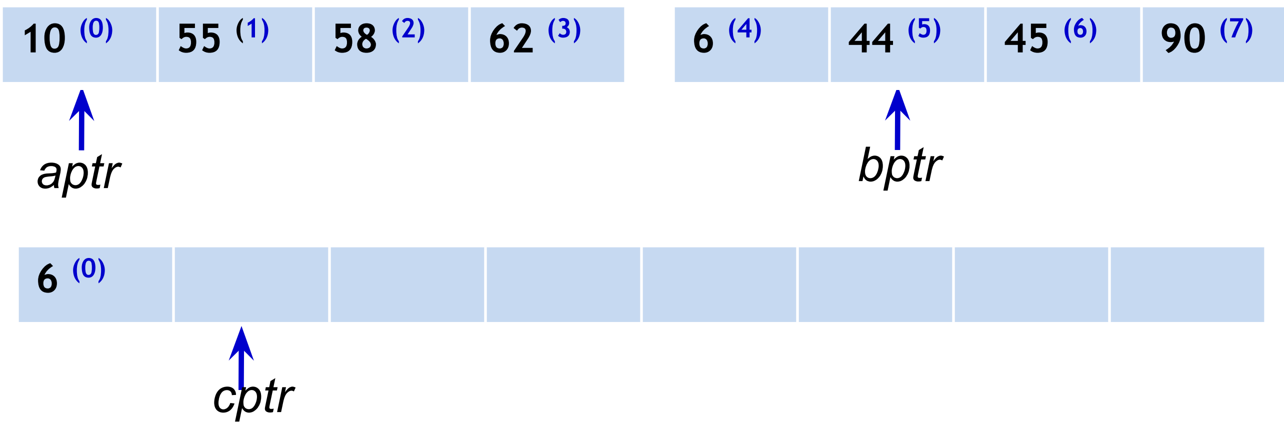
Merge(0, 1, 1) | Merge(2, 3, 3) | Merge(4, 5, 5) | Merge(6, 7, 7)

58 (0)	62 (1)	10 (2)	55 (3)	44 (4)	45 (5)	6 (6)	90 (7)

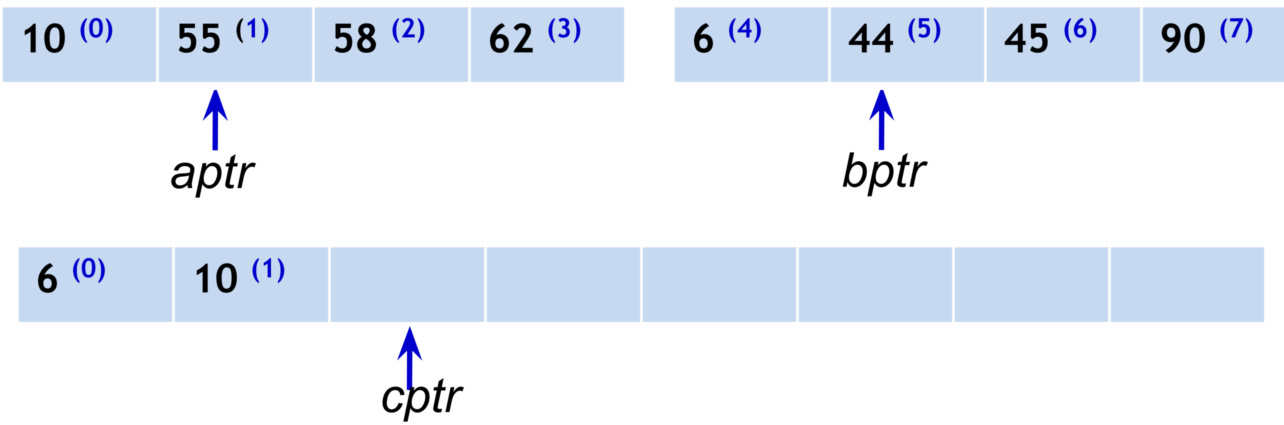
Merge two arrays



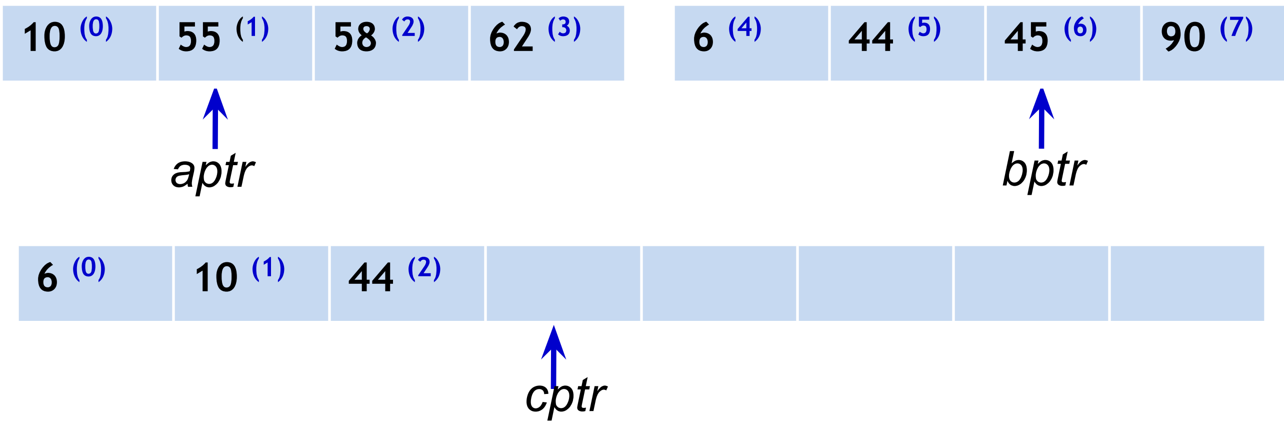
Merge two arrays



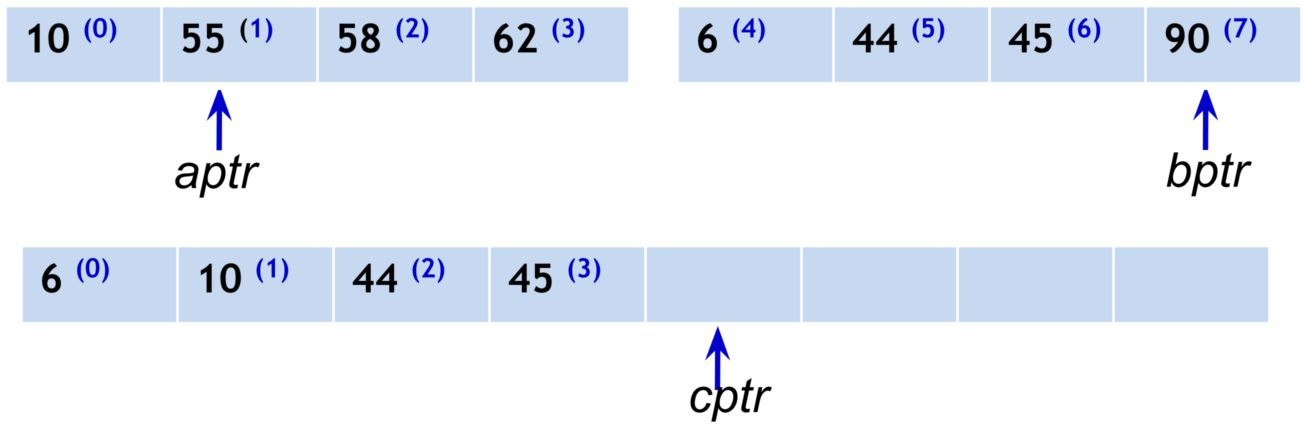
Merge two arrays



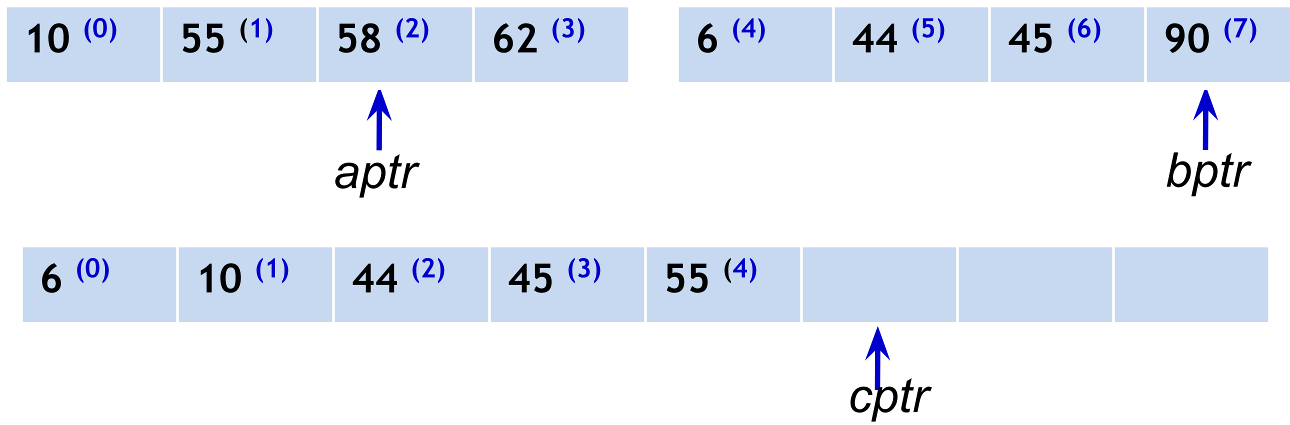
Merge two arrays



Merge two arrays



Merge two arrays



Merge two arrays

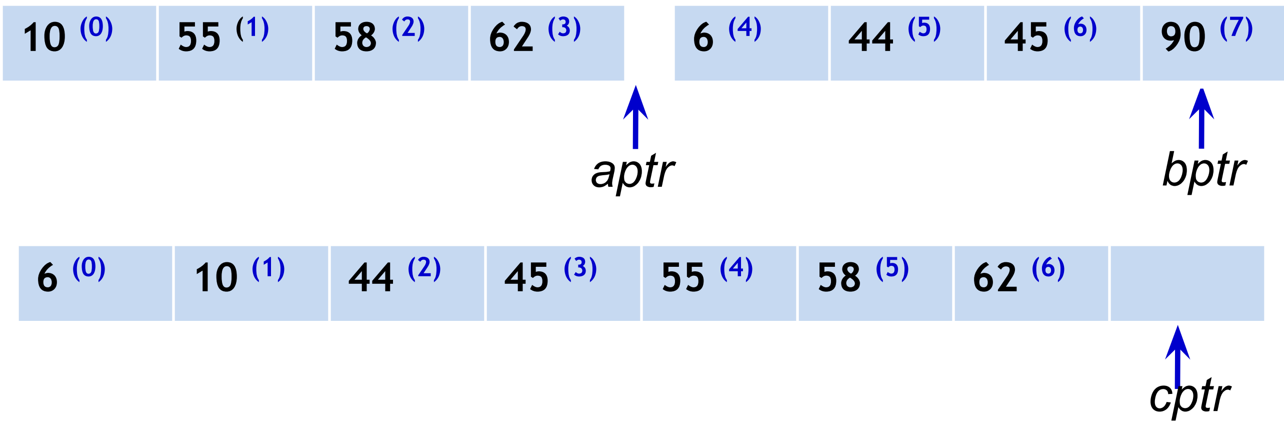
10 (0)	55 (1)	58 (2)	62 (3)	6 (4)	44 (5)	45 (6)	90 (7)

aptr *bptr*

6 (0)	10 (1)	44 (2)	45 (3)	55 (4)	58 (5)		

cptr

Merge two arrays



Merge two arrays

10 (0)	55 (1)	58 (2)	62 (3)	6 (4)	44 (5)	45 (6)	90 (7)

aptr *bptr*

6 (0)	10 (1)	44 (2)	45 (3)	55 (4)	58 (5)	62 (6)	90 (7)

cptr

Time Complexity of merge sort using tree method, Master theorem

$$T(n) = 2\,T(n/2) + O(n)$$

\# of sub-problems

size of sub-problems

work dividing & combining

Merge Sort is an efficient, stable sorting algorithm with an average, best-case, and worst-case time complexity of **O(n log n)**

Assignment ahead

Deadline: 14/02/2023