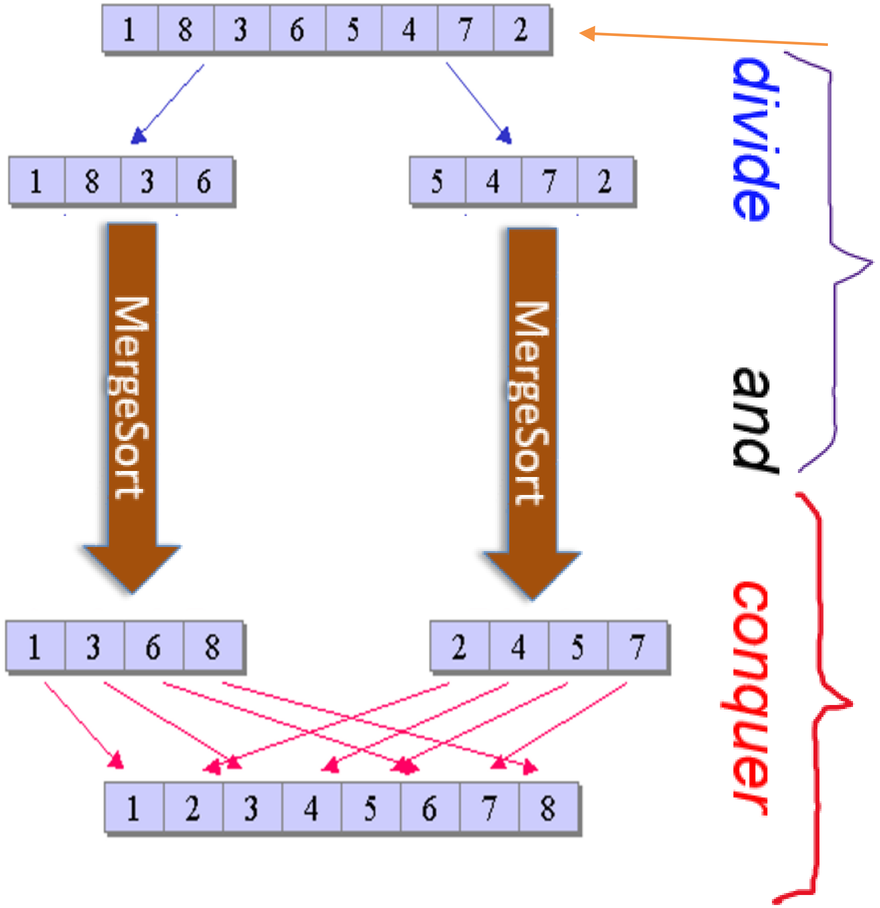
COMP20003 Workshop Week 8

Divide & Conquer

- Array-based Top-Down Merge Sort and Divide-And-Conquer
- Merge Sort: Other Variants
- Recurrences & The Master Theorem
- **MST**

Array-Based Top-Down Merge Sort: a Divide-And-Conquer Algorithm



the sorting algorithm is :

- "*divide*":
  - break the array into 2 equal halves

and

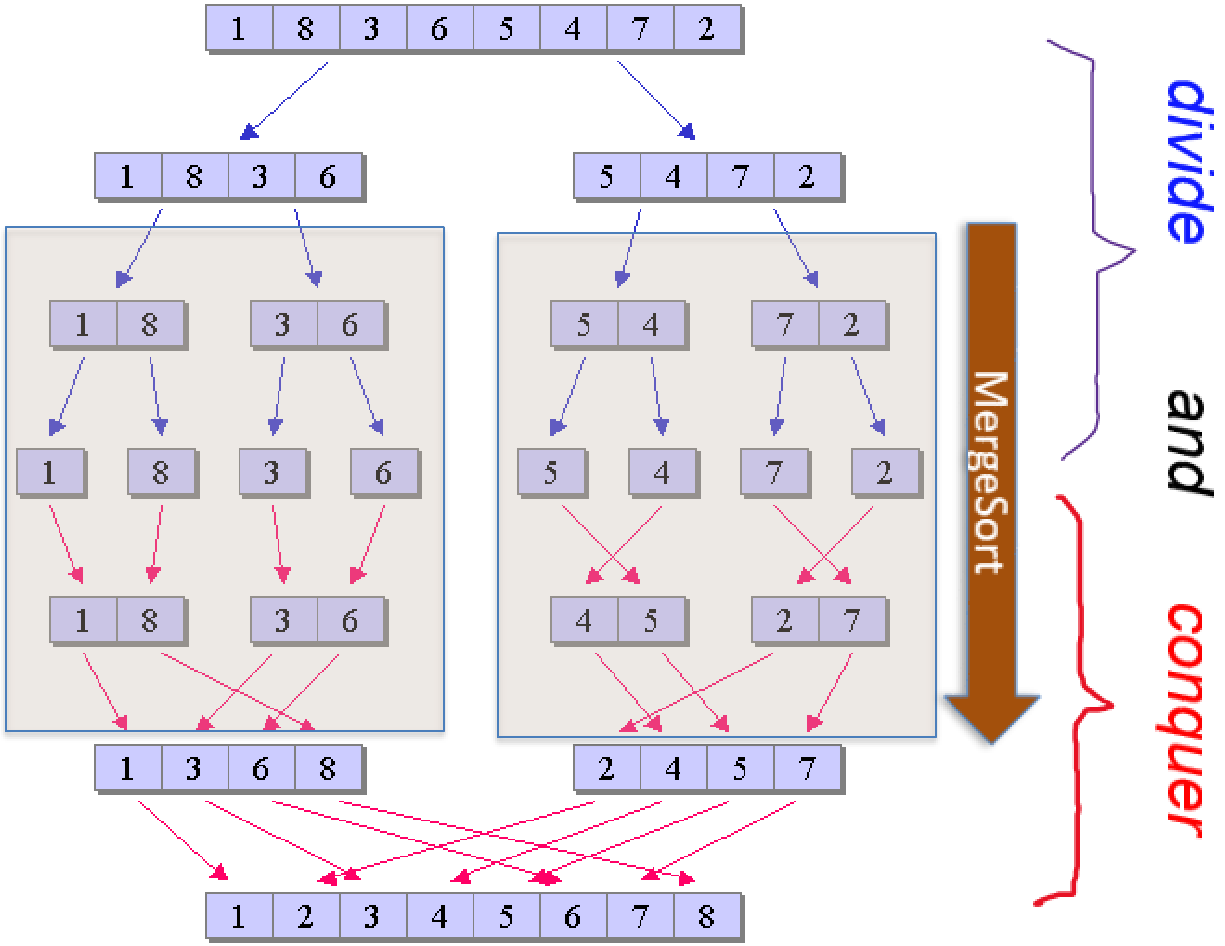
  - *sort each half separately*

then

- "*conquer*":
  - merge 2 sorted subarrays into one sorted array

Image Sources: Wikipedia, cs.amherst.edu, https://commons.wikimedia.org/wiki/File:Caricatures_of_Napoleon_I_of_France_detail,_Caricature_gillray_plumpudding_%28cropped%29.jpg

Array-Based Top-Down Merge Sort: a Divide-And-Conquer Algorithm



the sorting algorithm is :

- "*divide*":
  - break the array into 2 equal halves

then

- *recursively sort each half*
- *base case: sub-array length ≤ 1*

and

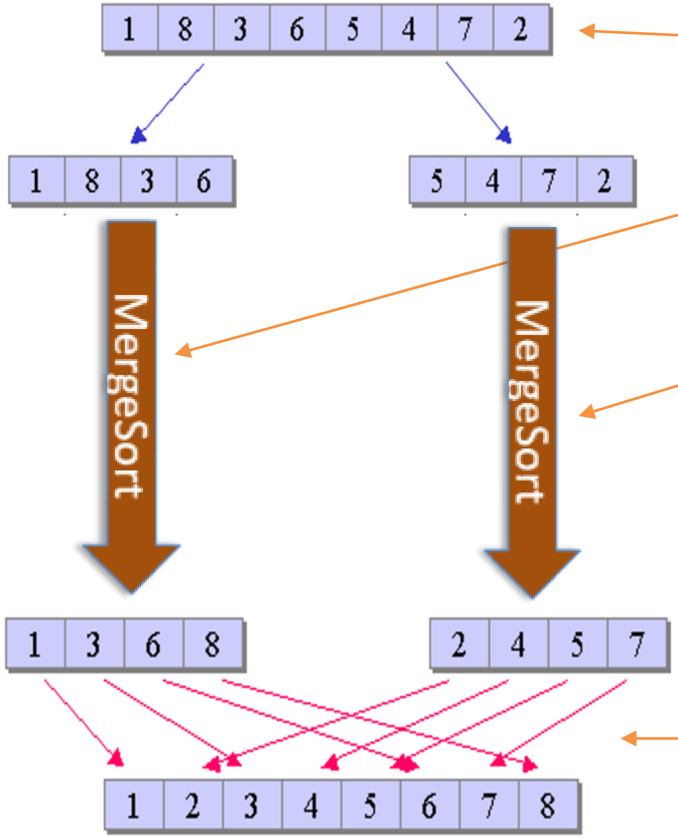
- "*conquer*":
  - merge 2 sorted subarrays into one sorted array

```
mergesort(A) {
  if (A has > 1 element) {
    B= left half of A
    C= right half of A
    mergesort(B);
    mergesort(C);
    merge B and C to A;
  }
}
```

Top-Down Merge Sort: time complexity

Time complexity:

$T(n) =$ # *complexity of sorting n elems includes:*



$T(n/2) \quad + \quad T(n/2)$ # *for sorting n/2 elems of the left and right*

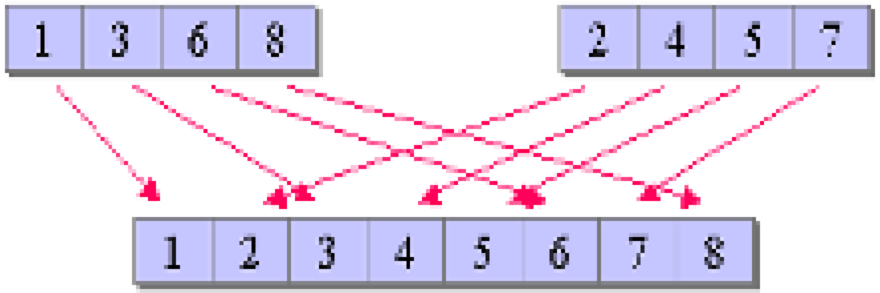
$+\ n$ # *+ time for merging which is* $\theta(n)$

$$\begin{cases} T(n) = 2T(n/2) + n \\ T(1) = 0 \end{cases}$$

(a recurrence)
Solving the recurrence give:
$T(n) = \theta(n \log n)$

Merging: How to merge 2 sorted arrays?

Input: two sorted arrays B and C(could be of different sizes)



Output: array A= sorted union of B and C

Algorithm:

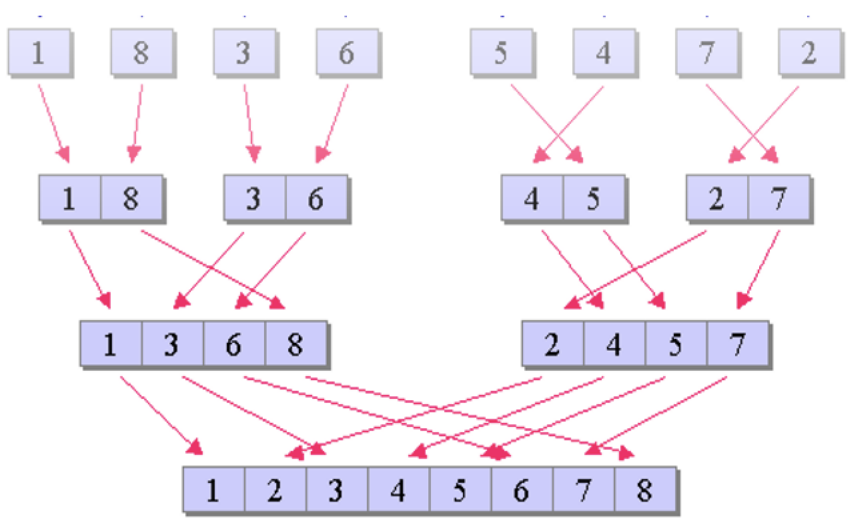
- Iterate B and C in parallel, and when doing so append the smaller value to A
- When B or C finishes, append the remainders of the other to A

Notes:

- The merging is **not** in-place, input and output arrays occupy different memory spaces!
- The merging process is stable (why?)

	Merging 2 sorted arrays	Mergesort of an array
Time Complexity	$\theta(n)$	$\theta(n \log n)$
Additional-Space Complexity	$\theta(n)$	$\theta(n)$ for merging $\theta(\log n)$ for recursion Total: $\theta(n)$

Array-Based Bottom-Up Merge Sort



Start: consider the original array as n singleton sub-arrays. Then do the merging process:

- merging pairs of adjacent sub-arrays of size 1 = 2^0
- merging pairs of adjacent sub-arrays of size 2^1
- merging pairs of adjacent sub-arrays of size 2^2

…

- merging pairs of adjacent sub-arrays of size around n/2

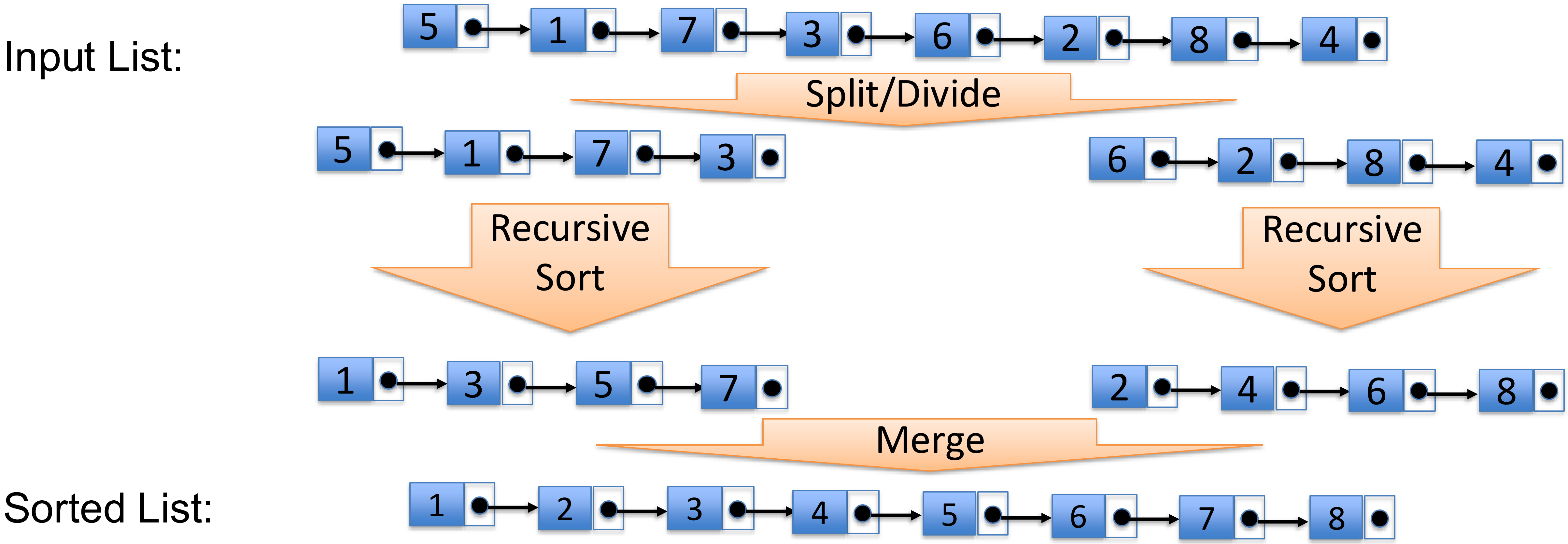
Simple Implementation of Bottom-Up Merge Sort using a queue:

Q = enqueue all n singleton arrays
while (the queue Q has at least 2 elements)
 dequeue 2 arrays, merge them, and enqueue the merged array
the single element in the queue in the output sorted array

Note: There is a tricky way of bottom-up implementation without using a queue

- Time Complexity: should be the same as top-down, $\theta(n \log n)$
- Additional space: $\theta(n)$ *- dominated by the extra memory for merging arrays (note: the queue, if used, is also* $\theta(n)$ *)*

Merge Sort can be done similarly for linked lists



Time complexity is the same as array-based: Θ(n logn) for both top-down and bottom-up

Space complexity:

- merging lists is in-place: no need extra space for data when merging
- Θ(logn) for top-down (recursive stack), Θ(n) for bottom-up (the queue)

Peer Activity: Mergesort Variant Analysis

What is the strongest bound on space complexity of top-down linked-list-based mergesort?

a. $O(1)$
b. $\Theta(\log n)$
c. $\Theta(n)$
d. none of the above

merge sort summary

Merge Sort:

- *can be done similarly for arrays or linked lists,*
- *can be implemented in the top-down or bottom-up manner,*
- *is stable.*

In terms of complexity, Merge Sort is

- *θ(n logn) time complexity for all variants*
- *θ(n) additional memory for most variants. Exception: θ(logn) extra memory for top-down merge sort in link lists*

Recurrences: describing complexity of recursive algorithms

Time Complexity of a *recursive algorithm* can be expressed as a recurrence: $T(n)$ depends on $T(k)$ where $k<n$. For example:

merge sort	$T(n)= 2\,T(n/2) + n$ $T(1)= 0$
binary search on sorted arrays	$T(n)=$ $T(1)=$
the best case of quick sort	$T(n)=$ $T(1)=$
the worst case of quick sort	$T(n)=$ $T(1)=$

The Master Theorem: quick solving some special recurrences

If a task of size n is divided into

- a tasks of size n/b :

$$T(n) = aT(n/b) + \Theta(n^d)$$

where $a \geq 1$, $b > 1$, and $d \geq 0$,

cost of dividing and conquering

Then

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

Recurrences: solving using Master Theorem

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

merge sort	*T(n)= 2 T(n/2) + n* *T(1)= 1*	*a=* *b=* *d=* → *T(n) =*

binary search on sorted arrays	*T(n)= T(n/2) + 1* *T(1)= 1*	*a=* *b=* *d=* → *T(n) =*

the best case of quick sort	*T(n)= T(n/2) + n* *T(1)= 1*	*a=* *b=* *d=* → *T(n) =*

the worst case of quick sort	*T(n)= T(n-1) + n* *T(1)= 1*	*a=* *b=* *d=* → *T(n) =*

Recurrences: solving using Master Theorem

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

merge sort	$T(n)= 2\ T(n/2) + n$ $T(1)= 1$	$a= 2$ $b= 2$ $d= 1$ $d > \log_2 2$ → $T(n) = \theta(n \log n)$
binary search on sorted arrays	$T(n)= T(n/2) + 1$ $T(1)= 1$	$a= 1$ $b= 2$ $d= 0$ $d == \log_2 1$ → $T(n) = \theta(n^0 \log n) = \theta(n \log n)$
the best case of quick sort	$T(n)= T(n/2) + n$ $T(1)= 1$	$a= 1$ $b= 2$ $d= 1$ $d > \log_2 1$ → $T(n) = \theta(n^1) = \theta(n)$
the worst case of quick sort	$T(n)= T(n-1) + n$ $T(1)= 1$	$a= 1$ b= NOT_APPLICABLE d= → $T(n) =$? cannot apply the master theorem → need to use substitution to solve

Recurrences: When the Master Theorem *not-applicable*

- When we cannot find at least one of the *constants* a, b, d.
- Example (Quick Sort Worst Case)
  - $T(n) = T(n-1) + n$
  - $T(1)= 1$

- Solution for this case: We solve/"unroll" the recursion using substitution. This method can be applied to any recurrence.

- ***Class Example: Solve the recurrence***

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

$$T(1)= 1$$

Recurrences: Another Example of using substitution

When we cannot find at least one of the *constants* a, b, d. Example (Quick Sort Worst Case)

$T(n) = T(n-1) + n \quad (1)$

$T(1) = 1$

$$T(n) = T(n-1) + n$$
$$= (\; T((n-1) - 1) + (n-1) \;) + n$$ [use (1), substitute n with $(n-1)$]
$$= T(n-2) + (n-1) + n$$ *#NOTE: be lazy 😀 , don't simplify (n-1)+n, wait to see the pattern*
$$= T(n-3) + (n-2) + (n-1) + n$$ [substituting n by $n-1$ in (1)]
$$= \ldots$$
$$= T(n-k) + (n-k+1) + \ldots + (n-1) + n \qquad (\; 0 \le k < n \;) \quad (2)$$

since we know $T(1)$ we choose k so that $T(n-k)$ is $T(1)$ → choose $k = (n-1)$

in (2) substitute k *with* $n-1$, we arrive to

$$= T(1) + 2 + \ldots + (n-1) + n$$
$$= 1 + 2 + \ldots + (n-1) + n = n(n+1)/2 = \theta(n^2)$$

Peer Activity: Described Recurrences

Which recurrence relation describes the algorithm in this scenario?

a. $T(n) = T(n/3) + 1$
b. $T(n) = T(n/3) + n$
c. $T(n) = 2T(n/3) + 1$
d. $T(n) = 2T(n/3) + n$

This algorithm subdivides its input into the ranges: low, medium, and high. It discards all data that is not included within the highest range, and repeats the subdivision on the remaining data. Each subdivision requires the algorithm to iterate through each data item.

mergesort: class exercises

For the sequence of character keys:

E X A M P L E

Show how mergesort work

a) top-down mergesort

b) bottom-up mergesort

mergesort: class exercises

For the sequence of character keys:

E X A M P L E

Note: The tracing should be the same for both array-based and list-based versions!

Show how mergesort work

top-down mergesort

E X A M P L E

E X A M | P L E

E X | A M | P L | E

E | X | A | M | P | L | E

E X | A M | L P | E

A E M X | E L P

A E E L M P X

bottom-up mergesort

E | X | A | M | P | L | E

E X | A M | L P | E

A E M X | E L P

A E E L M P X

MST Resources

Practice Test:

- around 15 multiple choice/ short answer questions for 30 minutes

Past MST:

- 2015 (complexity questions are relatively hard)
- 2017
- 2019

Topics in Workshops W1-W7:

- W1: C reviews, memory & pointers
- W2: Memory management, dynamic array, file IO
- W3: Linked Lists , intro to Complexity
- W4: Complexity; Stacks and Queues
- W5: Tree, Traversal, BST, AVL [***missing**: complete tree, BST deletion*]
- W6: Distribution Counting, Hashing
- W7: Sorting: properties; insertion, selection and quick sort
- W8: <not for this MST>

Additional Slides

Class Exercises and Lab

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

Class Exercises

Ex 1: Using the Master Theorem to solve the recurrence:

T(n)= 5T(n/2) + n^2 *+ 9nlogn*

T(1) = 1

Ex 2: Using substitution, solve the recurrence (of best case quicksort, mergesort):

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

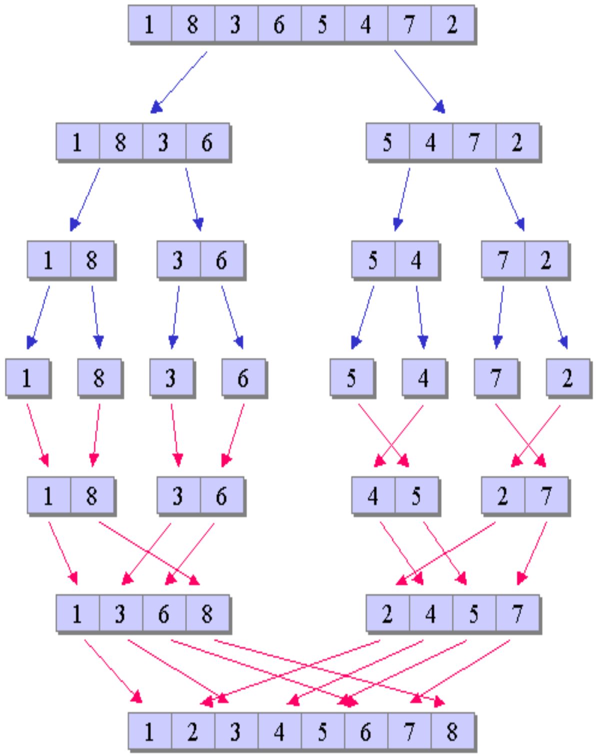
T(1)= 1

Ex 3: Show how top-down mergesort run for:

8 2 6 7 4 5

Ex 4: Show how bottom-up mergesort run for the sequence in Ex 3

Top-Down MergeSort: implementation note



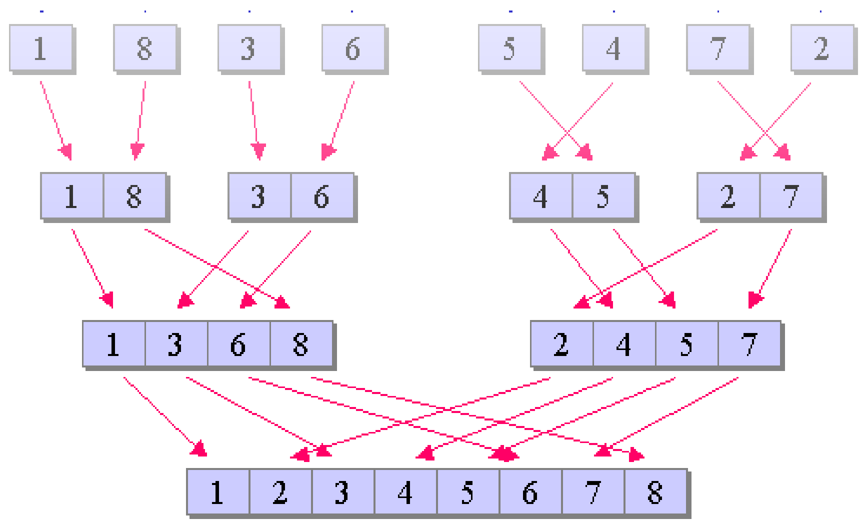
General Top-Down Merge Sort	Array-Based Top-Down Merge Sort

<pre>mergesort(A) {
  if (A has > 1 element) {
    B= left half of A
    C= right half of A
    mergesort(B);
    mergesort(C);
    merge B and C to A;
  }
}</pre>
	<pre>array_mergesort(A[], n) {
	  if (n>1) {
	    mid= n/2;
	    B[]= A[0..mid-1]
	    C[]= A[mid..n-1]
	    array_mergesort(B,mid);
	    array_mergesort(A,n – mid));
	    array_merge(B, C, A);
	  }
	}</pre>

	Additional-Space Efficiency	
	Array-Based TD Mergesort	Linked-List-Based
for recursive stack	***θ****(log n)*	***θ****(?)*
for merging	***θ****(n)*	***θ****(?)*
all	***θ****(n)*	***θ****(?)*
Notes	*the algorithm is NOT in-place*	*top-down mergesort for linked list: normally not used*

Merge Sort: Bottom-Up for Linked Lists (for detailed understanding and/or implementation)

- Time Complexity: should be the same as top-down, ***θ****(n logn)*
- Additional space:
  - no stack, no memory for merging
  - but need memory for queue, it's ***θ****(???)*



Use a queue!

initially:

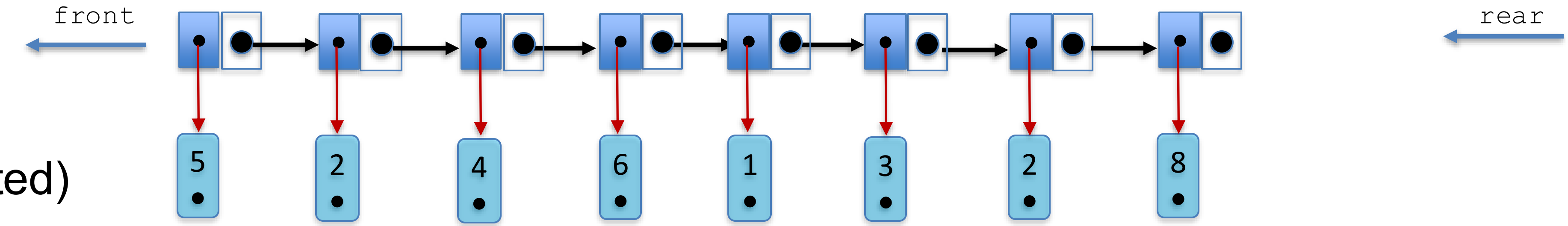
- consider each element as a singleton linked list
- put all singletons into an empty queue Q

```
while (Q has 2 or more lists) {
   dequeue 2 lists
   merge them into one sorted list
   enqueue the merged list
}
// Q should have only one element
dequeue to get the sorted solution
```

Merge Sort in Linked Lists: Bottom-Up for

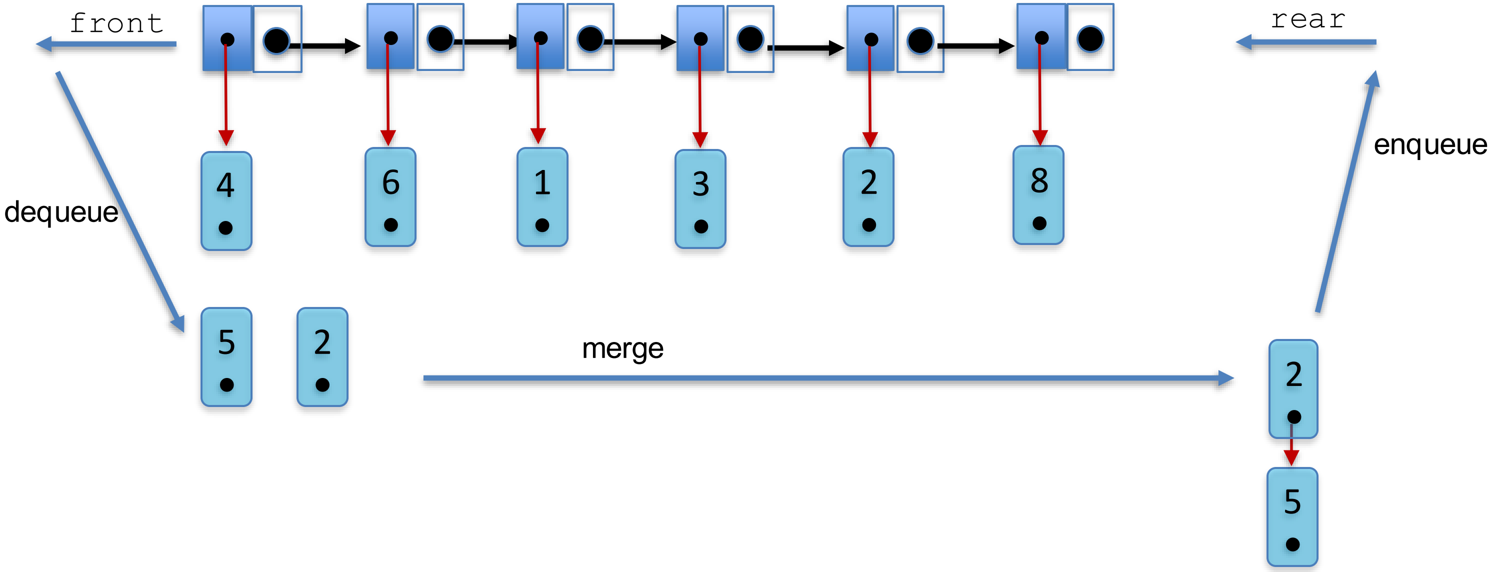
```
{
   5,
   2,
   4,
   6,
   1,
   3,
   2,
   8
}
```

1. Start with enqueuing all singleton (sorted) lists into Q:



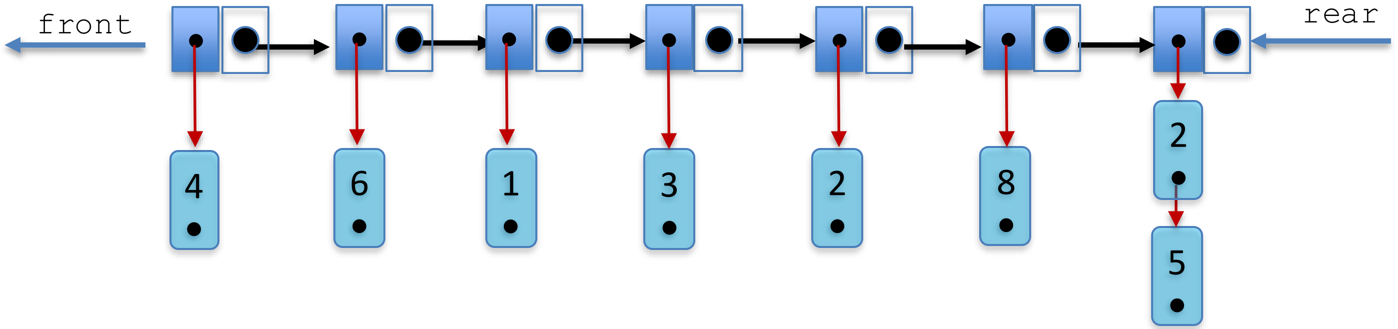
2.

- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the sorted list



3.

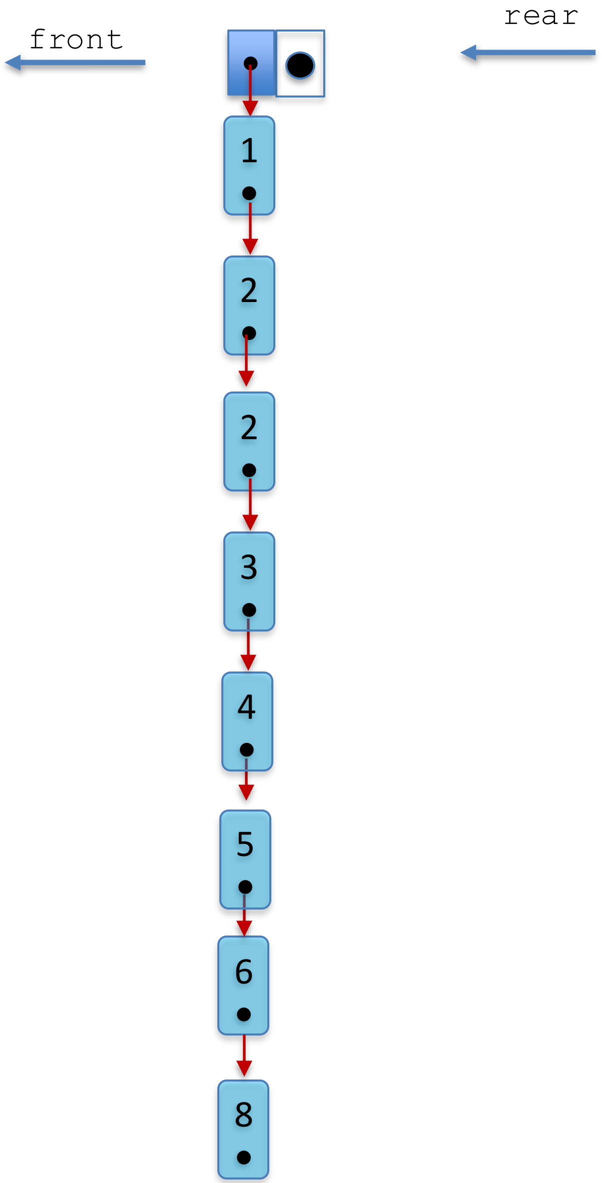
- repeat Step 2 until having a single list in Q (this last single list is the solution)



Merge Sort: (here-unstable) Bottom-Up for 5, 2, 4, 6, 1, 3, 2, 8

while Q has at least 2 elements:

- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list




```
At the end, the
queue has only a
single element.
Dequeue that to
get the final
sorted list.
```

Additional memory need:

- *no recursive, no stack memory needed*
- *but, need $\theta(n)$ for the queue (more than for the stack ☹)*
- *no additional memory for merging*

To sum up, mergesort is

- *$\theta(n \log n)$ time complexity for all variants*
- *$\theta(n)$ additional memory for most variants. Exception: $\theta(\log n)$ memory for top-down mergesort in link lists*