CSCE 221: Data Structures and Algorithms

Sorting Algorithms: Merge Sort

Guest Lecturer:

Fatemeh Asgarinejad
PhD candidate at UC San Diego

Announcements + schedule + links

Today's Learning Goals

- Sorting Algorithms:
  - Divide and Conquer Paradigm
  - Merge Sort (Top Down)
    - Basic Plan
    - Merging two sorted arrays
    - Pseudocode
    - Execution example
    - Time and space complexity
    - Merge-sort tree
    - Comparison with other algorithms
    - Implementation in C++
  - Practical Improvements
    - Merge Sort (Bottom Up)
  - Quick Sort Introduction

Divide and Conquer

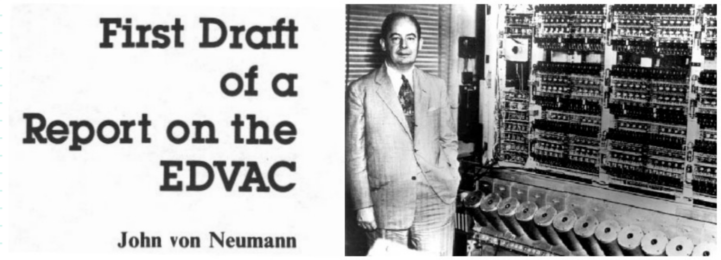
- Break up problem into several parts
- Solve each part recursively
- Combine solutions to sub-problems into overall solution

Common usage:

- Break up problem of size n into two equal parts of size $\frac{1}{2}n$
- Solve two parts recursively
- Combine two solutions into overall solution in linear time

Merge Sort (Top Down)

Introduced by designer of :



Basic plan for sorting an array (here alphabetically):

- Divide array into two roughly equal halves.
- Conquer each half in turn (by recursively sorting). How?
- Merge two halves. How?

Left half

Right half

Input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
Sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
Merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Note: a green subarray means sorted.

Merging two sorted arrays

Input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
Sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X

Merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Did you notice in case of having same values, we prioritize the element from the left subarray?

Note: Merge Sort is a Stable Sorting Algorithm, meaning, it preserves the relative order of items with equal keys.

Merging two sorted arrays

Note: Merge Sort is a Stable Sorting Algorithm, meaning, it preserves the relative order of items with equal keys.

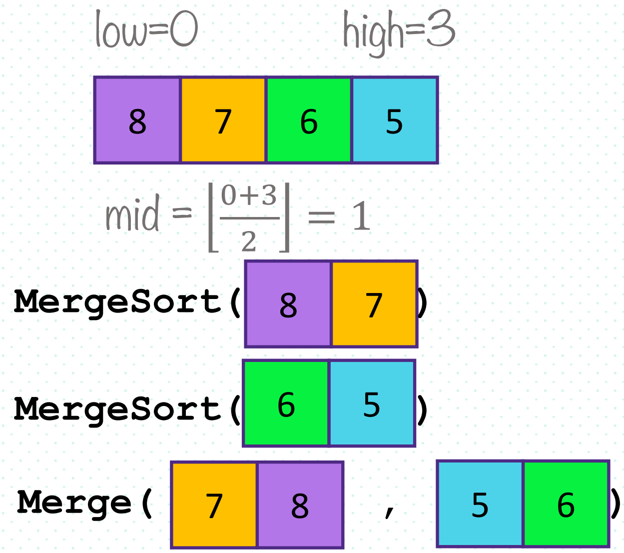
Use case: secondary sorting:

United States of America	40	44	42	126
People's Republic of China	40	27	24	91
Japan	20	12	13	45
Australia	18	19	16	53
France	16	26	22	64

Paris 2024 Medal Tablex

Merge Sort (Top Down) Pseudocode

```
MergeSort(the_list, low, high) {
   if (low < high) {
      mid = floor((low + heigh) / 2);
      MergeSort(A, low, mid);
      MergeSort(A, mid+1, high);
      Merge(A, left, mid, high);
   }
}
```



Merge Sort (Top Down) Pseudocode

Algorithm merge(S_1, S_2, S):

Input: Two arrays, S_1 and S_2, of size n_1 and n_2, respectively, sorted in non-decreasing order, and an empty array, S, of size at least $n_1 + n_2$

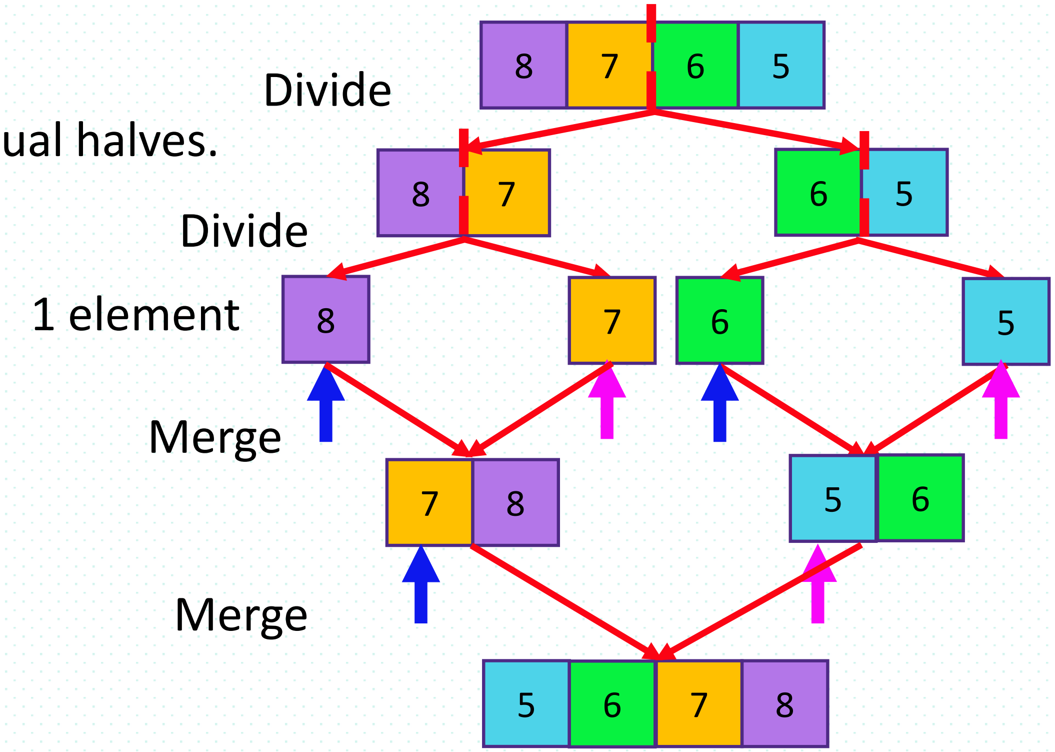
Output: S, containing the elements from S_1 and S_2 in sorted order

$i \leftarrow 1$
$j \leftarrow 1$
while $i \leq n$ **and** $j \leq n$ **do**
  if $S_1[i] \leq S_2[j]$ **then**
    $S[i + j - 1] \leftarrow S_1[i]$
    $i \leftarrow i + 1$
  else
    $S[i + j - 1] \leftarrow S_2[j]$
    $j \leftarrow j + 1$
while $i \leq n$ **do**
  $S[i + j - 1] \leftarrow S_1[i]$
  $i \leftarrow i + 1$
while $j \leq n$ **do**
  $S[i + j - 1] \leftarrow S_2[j]$
  $j \leftarrow j + 1$

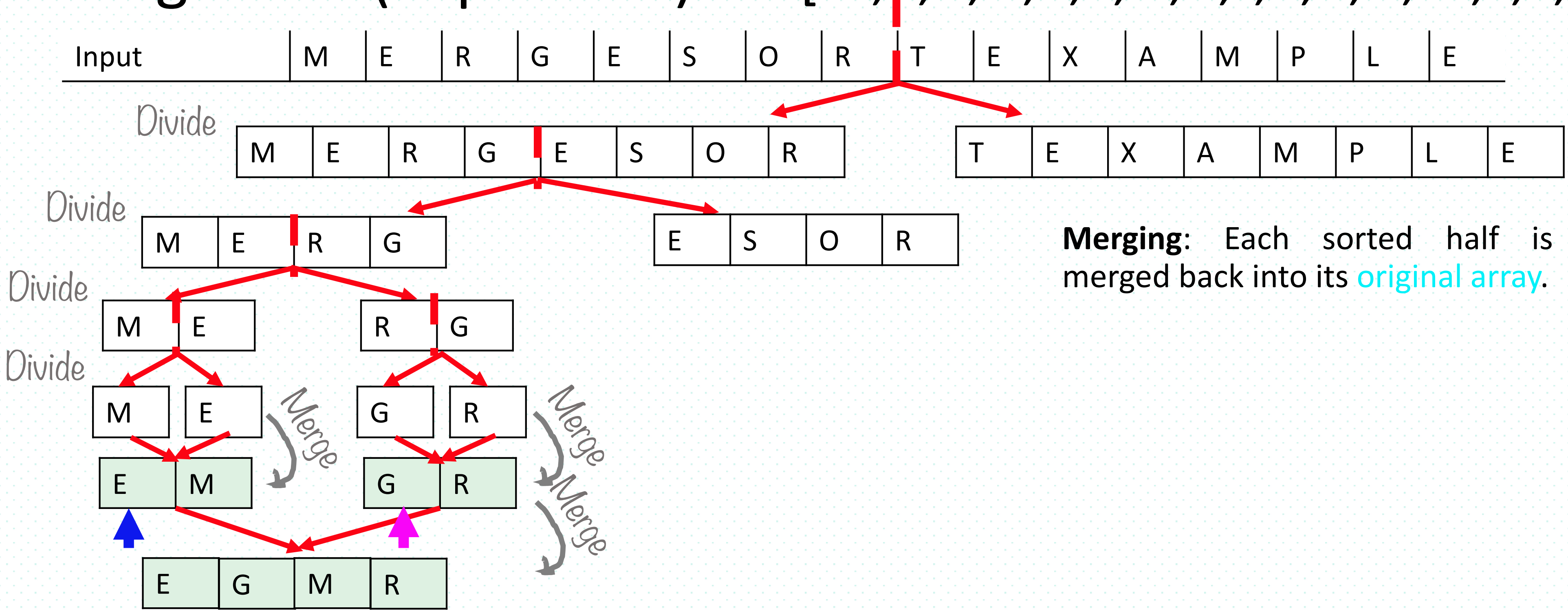
Merge Sort (Top Down) on array [8, 7, 6, 5]

Basic plan:

- Divide array into two roughly equal halves.
- Conquer each half in turn.
- Merge two halves.

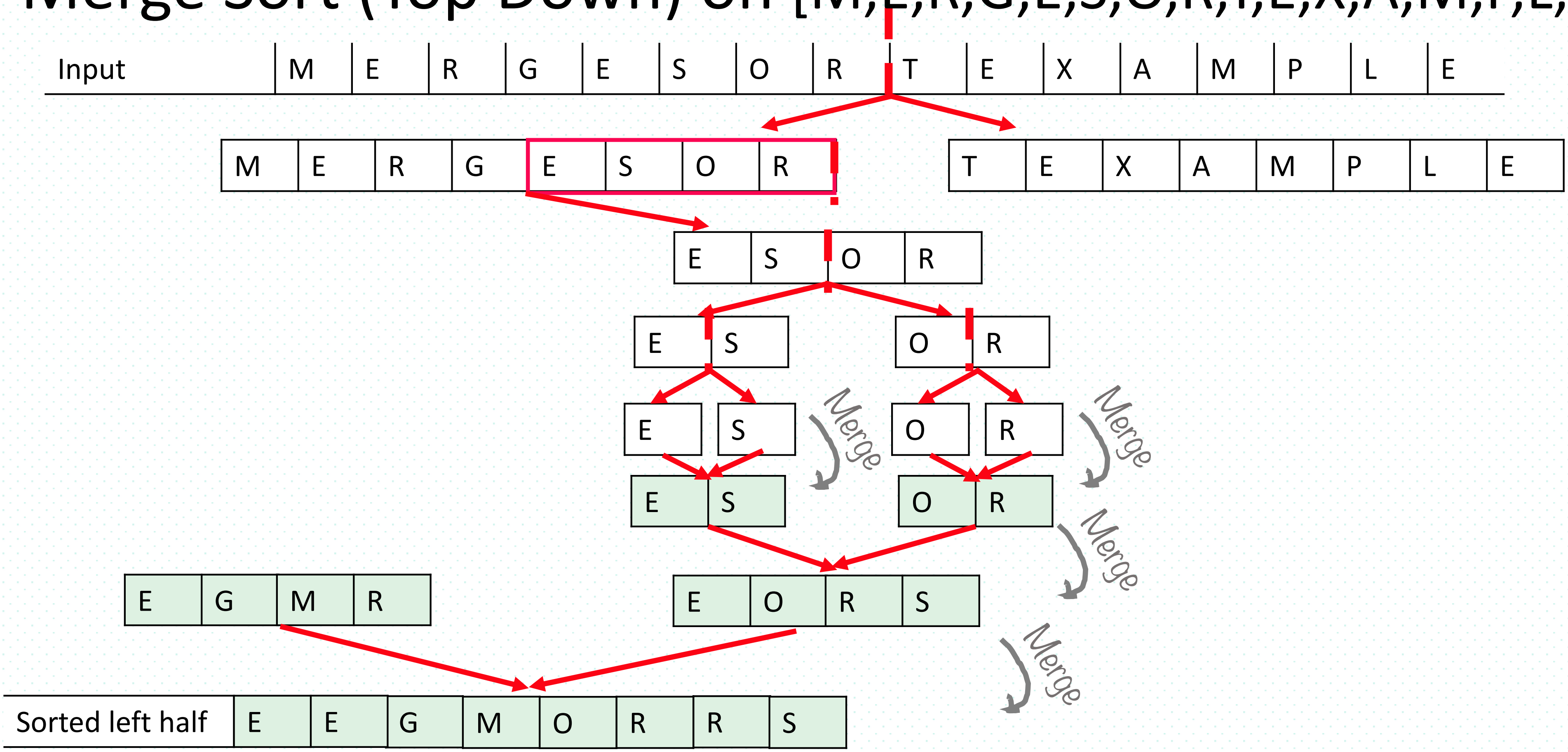


Merge Sort (Top Down) on [M,E,R,G,E,S,O,R,T,E,X,A,M,P,L,E]



Merging: Each sorted half is merged back into its original array.

Merge Sort (Top Down) on [M,E,R,G,E,S,O,R,T,E,X,A,M,P,L,E]



Merge Sort (Top Down) on [M,E,R,G,E,S,O,R,T,E,X,A,M,P,L,E]

- Divide array into two roughly equal halves.
- Conquer each half in turn (by recursively sorting).
- Merge two halves.

Input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Sorted left half	E	E	G	M	O	R	R	S								

Perform Merge Sort on the above right subarray to get:

Sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X

Then Merge two halves to get:

Merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Merge Sort's Time Complexity

Let T(n) be the running time of Merge Sort of n elements.

Merge Sort divides array in half and calls itself on the two halves. After returning sorted left and right, it merges both halves using a temporary array.

Each recursive call takes $T\left(\frac{n}{2}\right)$ and merging takes $O(n)$

Calling itself with two halves

Merge Sort Runtime of n elements $\longleftarrow T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ $\longleftarrow$ Merging n elements

Input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	$O(1)$ *divide*
Sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E	sort $T(\frac{n}{2})$
Sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	sort $T(\frac{n}{2})$
Merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X	merge $O(n)$

Merging two sorted arrays Time Complexity

Input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
Sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
Merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

How many comparisons are performed in above merge?

A) $n = 16$

B) $n = \binom{16}{2}$

C) $at\ most\ 15$

https://www.menti.com/blrtwbmz7d6t

Up to 16-1 comparisons (here 13 you stop when one array is exhausted.)

Merging two sorted arrays Time Complexity

Input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
Sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
Sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
Merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

How many placements are performed in above merge?

n=16 placements in “Merge results” array.

Total # of operations for sorting two arrays of length $\frac{n}{2}$ is proportional to n (each element in array is **processed once**) :O(n)

Merge Sort's Time Complexity using Master's Theorem

Let T(n) be the running time of Merge Sort of n elements $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Can we use Master's Theorem to write the closed form of $T(n)$?

$a = 2, b = 2, k = 1, and\ a = b^k\ therefore, T(n) \in O(n^k \log_b n)\ \rightarrow \boldsymbol{T(n) \in O(nlogn)}$

Note: The recurrence relation $T(n) = aT\left(\frac{n}{b}\right) + cn^k$

where $a \geq 1, b > 1,$ and c are all constants, is solved (asymptotic complexity) as:

$$\begin{cases} If\ \ a < b^k & \rightarrow \quad T(n) \in O(n^k) \\ If\ \ a = b^k & \rightarrow \quad T(n) \in n^k \log_b n \\ If\ \ a > b^k & \rightarrow \quad T(n) \in n^{\log_b a} \end{cases}$$

Merge Sort's Time Complexity

Let T(n) be the running time of Merge Sort of n elements $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Using Master′s Theorem: $\boldsymbol{T(n) \in O(nlogn)}$

Example: For sorting 1 million elements in an array $\approx 1{,}000{,}000 \times \log_2 1{,}000{,}000 \approx 20\ milion\ operations$

On a usual laptop that executes 10^8 comparisons in a second , Merge Sort takes **few seconds**.

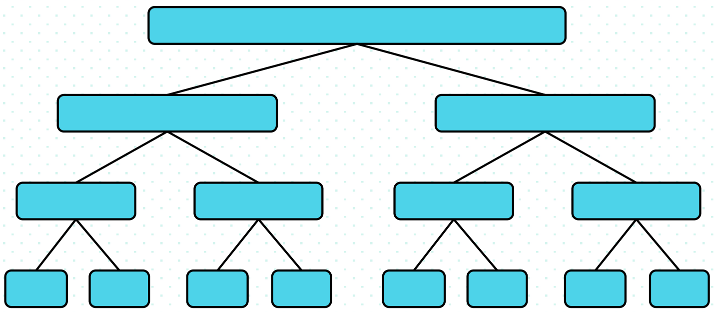
```
    ...:
Merge Sort runtime
 10,000 elements: 0.0413 seconds
 100,000 elements: 0.4960 seconds
 1,000,000 elements: 6.5174 seconds

In [9]:
```

Merge Sort's Time Complexity Tree

Let T(n) be the running time of Merge Sort of n elements $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

depth	#subarrays	size of each subarray
0	1	$\boldsymbol{n}$
1	2	$\frac{\boldsymbol{n}}{2}$
i	2^i	$\frac{\boldsymbol{n}}{2^i}$
h =???	$\boldsymbol{n}$	$\frac{\boldsymbol{n}}{\boldsymbol{n}}$



$n = 2^h \rightarrow h = \log_2 n$ so height h of the merge-sort tree is O(log n)

- Overall #operations (amount of work) in each depth i is $O\left(2^{\mathrm{i}} \times \frac{n}{2^i} 2^i\right) = \boldsymbol{O}(\boldsymbol{n})$
- Therefore, the total runtime of Merge is $\text{height} \times \mathit{work\ at\ each\ heigth} = \boldsymbol{O}(\boldsymbol{n} \times \log \boldsymbol{n})$

Properties of Merge Sort

- Time Complexity:
  - Best Case (best possible input (already sorted)) $\boldsymbol{O(nlogn)}$
  - Average Case (a randomly ordered input) $\boldsymbol{O(nlogn)}$
  - Worst Case (input is sorted in reverse order) $\boldsymbol{O(nlogn)}$

5	6	7	8

Best Case (BC)

8	6	7	5

Average Case (AC)

8	7	6	5

Worst Case (WC)

Note: For some algorithms WC, AC and BC are different. E.g., for insertion and bubble sort WC and AC are $O(n^2)$ while BC is $O(n)$.

n	Insertion Sort $\boldsymbol{O(\frac{n(n+1)}{2})}$	Merge Sort $\boldsymbol{O(nlogn)}$	Ratio Insertion/Merge
8	36	24	0.67
16	136	64	0.47
32	528	160	0.30
2^{10}	524,800	10,240	0.02
2^{20}	549,756,338,176	20,971,520	0.00004

Properties of Merge Sort

- Adaptive (does partially sorted input improve the efficiency of the algorithm)? No!
- Place (in/out) (requiring extra memory to sort)?

  Out (requires an auxiliary array; $O(n)$ extra space): space complexity

Note: Some sorting algorithms are in-place, meaning, they manipulate the original array directly without needing extra memory (auxiliary array) to store results during the process of sorting E.g., bubble Sort and insertion sort

Merge Sort: Practical Improvements???

Let's divide each array into three halves, sort them and merge them:

$$T(n) = 3T\left(\frac{n}{3}\right) + O(n)$$

$a = 3, b = 3, k = 1, and\ a = b^k \quad therefore, T(n) \in O(n^k \log_b n) \ \rightarrow \boldsymbol{T(n) \in O(nlog_3 n)}$

$\mathrm{nlog_3n} < \mathrm{nlog_2n}$ so 3-way Merge Sort seems to be faster, but the merge step (combining 3 arrays is more complex and requires more memory usage)

Exercise: Write the pseudo-code for 3-way Merge Sort

Merge Sort: Practical Improvements

Early Termination Check During Merge (Time and Space Optimization):

- Is largest item in first sorted half ≤ smallest item in second sorted half?

Parallel Merge Sort (Time Optimization):

- Divide and conquer steps can be performed in parallel using multi-processing.

Merge Sort: Practical Improvements

Bottom-Up Merge Sort (Non recursive; more efficient in terms of stack usage, especially in large datasets):

- Non-recursive version of Merge Sort
- Merges subarrays of increasing sizes iteratively (starting from size 1, then 2, 4, 8, and so on)
- Space: O(n) for temporary arrays (like standard Merge Sort). Time: O(n log n).

In-place Merge Sort (Space optimization):

- Does not use auxiliary array. Space: O(1)

Exit Ticket – Due Today at 11:59 PM

Choose all correct options:

A) For the temporary array, Merge Sort requires an additional space of O(n).

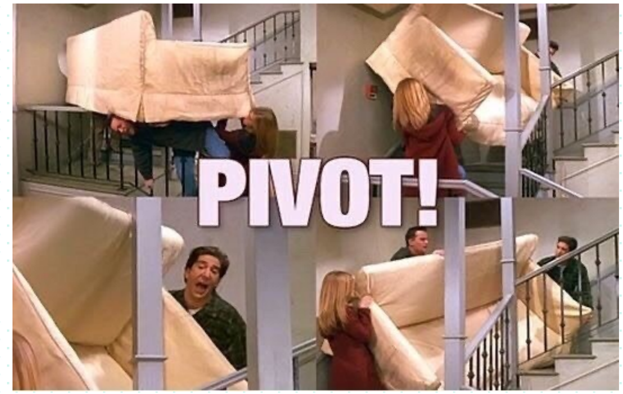
B) Merge Sort can only sort data in a specific range.

C) Even if the array is sorted, the merge sort goes through the entire process.

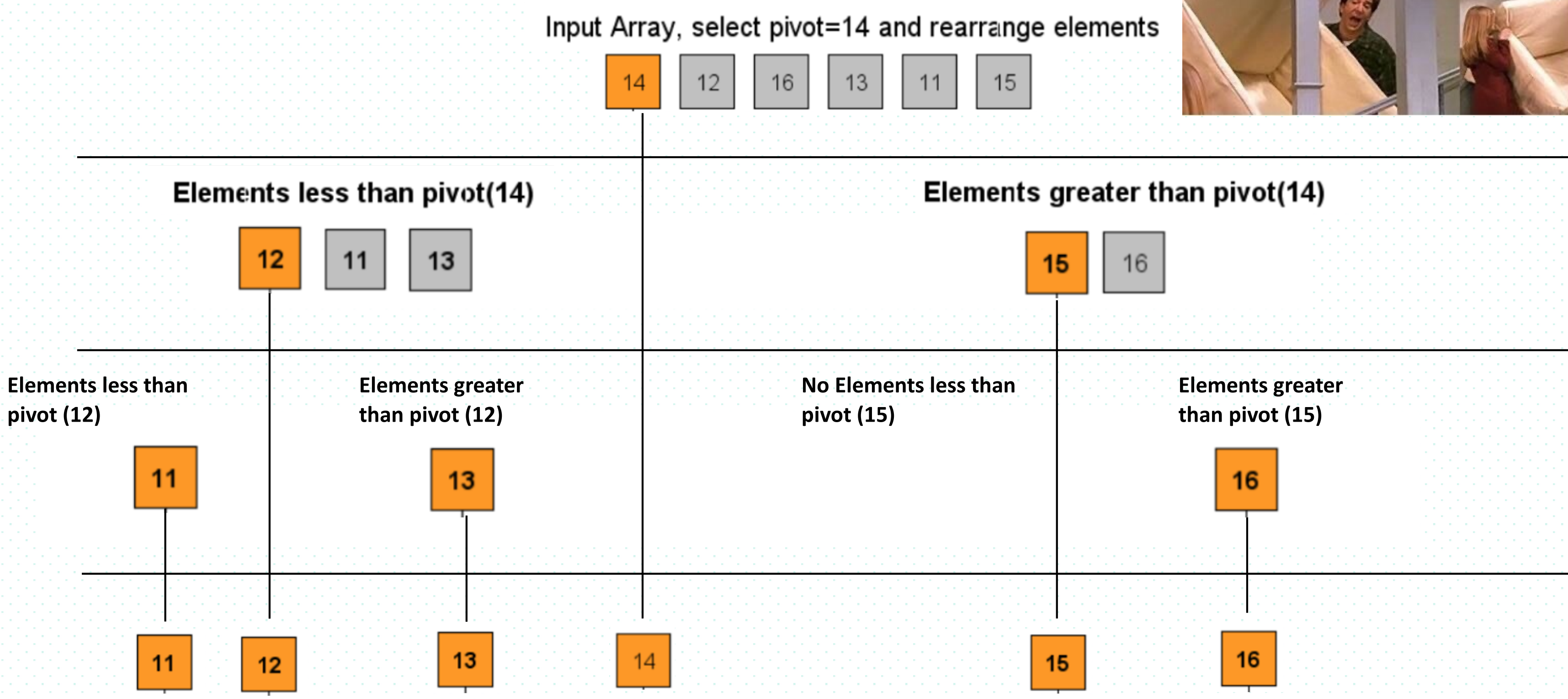
D) Merge Sort has different WC, BC and AC Runtimes.

https://forms.gle/YghAheBrhwx8iDBT7

Quick Sort



PIVOT!



Input Array, select pivot=14 and rearrange elements
14
12
16
13
11
15
Elements less than pivot(14)
12
11
13
Elements greater than pivot(14)
15
16
Elements less than pivot (12)
11
Elements greater than pivot (12)
13
No Elements less than pivot (15)
Elements greater than pivot (15)
16
11
12
13
14
15
16

Backup: Merge-sort's Runtime using Unraveling

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

as $T\left(\frac{n}{2}\right) = \boxed{2T\left(\frac{n}{4}\right) + O(\frac{n}{2})}$

$$= 2\times\boxed{\left(2T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)\right)} + O(nlogn) = 4T\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right) + O(n)$$

as $T\left(\frac{n}{4}\right) = \boxed{2T\left(\frac{n}{8}\right) + O(\frac{n}{4})}$

$$= 4\times\boxed{\left(2T\left(\frac{n}{8}\right) + O\left(\frac{n}{4}\right)\right)} + 2O\left(\frac{n}{2}\right) + O(n)$$

$$= 8\,T\left(\frac{n}{8}\right) + 4O\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right) + O(n)$$

$$= 2^k\,T\left(\frac{n}{2^k}\right) + \cdots + 4O\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right) + O(n)$$

If we continue above recursion, this $\frac{n}{powers\ of\ 2}$ will sometime stop. When is that?

Backup: Merge-sort's Runtime using Unraveling

$$T(n) = 2\boldsymbol{T}\left(\frac{\boldsymbol{n}}{\boldsymbol{2}}\right) + O(n)$$

$$\vdots$$

$$= 2^k\, T\left(\frac{n}{2^k}\right) + \cdots + 4O\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right) + O(n)$$

$$= 2^k\, T\left(\frac{n}{2^k}\right) + 2^{k-1}O\left(\frac{n}{2^{k-1}}\right) + \ \ldots\ + 2^2O\left(\frac{n}{2^2}\right) + 2^1O\left(\frac{n}{2^1}\right) + 2^0O(\frac{n}{2^0})$$

Set $\boldsymbol{\frac{n}{2^k} = base = 1}$ ***as T(1)=0 and simplify above*** $\boldsymbol{(n = 2^k}$***, hence*** $\boldsymbol{k = logn)}$

$$= 2^{\log_2 n}\, T(1) + 2^{logn\,-1}O\left(\frac{n}{2^{logn\,-1}}\right) + \ \ldots\ + 4O\left(\frac{n}{2^{log4\,-1}}\right) + 2^{11}O\left(\frac{n}{2^1}\right) + 2^0O\left(\frac{n}{2^0}\right)$$

$$= \sum_{i=1}^{\log_2 n} 2^iO(\frac{n}{2^i}) = \sum_{i=1}^{\log_2 n} O(n) = \boldsymbol{O(nlogn)}$$

References

- Texas A and M: Dr. Scott Schafer and notes of Prof. Lupoli
- Demo from: https://sp19.datastructur.es/materials/lab/lab11/lab11
- https://csvistool.com/MergeSort