Data Structures - Assign_04

Name: M. Rizwan Shafiq

Reg. No: SP24-BCS-069

Compute the complexity, step-by-step elaboration, best and worst case of fallowing:

1) Linear Search Algorithm:

Basic Searching algorithm that sequentially check each element is found or end of list is reached

- Complexity:-
  - → Time Complexity:
    - * Best Case: O(1) (item at first index)
    - * Worst Case: O(n) (item at last or isn't found)
    - * Average Case: O(n)
  - → Space Complexity:
    - * Requires no extra space
- Step-by-Step Execution:

arr = {5, 8, 2, 9, 3}, key = 9

- compare 5 with 9 → Not equal
- compare 8 with 9 → Not equal
- compare 2 with 9 → Not equal
- compare 9 with 9 → Match found at index 3.

Best case: key is at index 0

Worst case: key is not present or at last index

2) Binary Search Algorithm:

Highly efficent but works on sorted data only. Instead of checking 1 by 1, it divides data into two halves and if greater move start to mid+1 and if lesser move end to mid-1 (skipping the other side

• Complexity:-

→ Time Complexity:

* Best Case: $O(1)$
* Worst Case: $O(\log n)$
* Average Case: $O(\log n)$

→ Space Complexity:

No extra space required.

Execution:-

arr[] = {2, 4, 6, 8, 9, 10}, key = 9

• Start = 0 End = 5 → Mid = 2 → arr[2] = 6 → No match.
• key = 9 > 6 so start = Mid + 1.
• Start = 3, End = 5 → Mid = 4 → arr[4] = 9 → Match.

Best Case:

Middle element is target.

Worst Case:

Repeatedly halve until found or search space is empty.

3) Hashing by Linear Probing:

Technique to map keys to specific location in data structure called hash table. Mapping is done using a hash function.

- Complexity:-
  - → Time Complexity:
    - ⋆ Best Case: $O(1)$ → no collision
    - ⋆ Worst Case: $O(n)$ (all slots filled or many collisions)
    - ⋆ Average Case: $O(1)$ → generally constant time but performance degrades with more element come & clustering increases
  - → Space Complexity:
    - $O(m)$ → required for hash table itself where 'm' is no. of slots.

Step-by-Step working

Hash Table Size → 5

insert values = 10, 15, 20

hash function = key % 5

$h'(x) = [h(x) + f(x)] \% 5$ where $f(x) = 0, 1, 2 \ldots$

initial hash table (empty)

index =	0	1	2	3	4
value	–	–	–	–	–

① insert 10

$h(10) = 10 \% 5 = 0$ (slot '0' is empty)

Table: 10 – – – –

② insert 15.

$h(15) = 15 \% 5 = 0$ (collision)

$h'(15) = [15 + 1] \% 5 = 1$ (slot '1' empty)

Table: 10 15 - - -

③ insert 20

$h'(20) = (20+0) \% 5 = 0$ (collision)

$h'(20) = (20+1) \% 5 = 1$ (collision)

$h'(20) = (20+2) \% 5 = 2$ (slot empty).

Table: 10 15 20 - -

Best Case: No collision

Worst Case: Many collision, linear probing needed.
To find 20 algo checks index 0 and 1 and then 2.

4) Hashing by Chaining Method:

It resolves the collisions of hash table problem. Instead of for an open slot → each slot in hash table as a pointer to a collection.

- Complexity:
  - → Time Complexity:
    - ★ Best Case: $O(1)$
    - ★ Worst Case: $O(n)$ (all elements hash to same index)
    - ★ Average Case: $O(1+\alpha)$ → cost for computing hash then search a short list.
  - → Space Complexity:

    $O(n+m)$

    Space for hash table (m) + space for every key stored in linked list (n)

→ Working:

hash table size = 5

insert values -- 10, 15, 20, 21

$h(x)$ = key % 5

Table	index
–	0
–	1
–	2
–	3
–	4

① insert 10:-

$h(10) = 10 \% 5 = 0$.

Table: 10 – – – –

insert 15:

$h(15) = 15 \% 5 = 0$

Table = 10 – – – –
↓
15

insert 20: $h(20) = 20 \% 5 = 0$

Table = 10 – – – –
↓
15
↓
20

insert 21: $h(21) = 21 \% 5 = 1$

Table = 10 21 – – –
↓
15
↓
20

Best Case: Uniform distribution, (short chains).
Worst Case: All elements in one chain.

index	LL
0	[10] → [15] → [20]
1	[21]
2	null
3	null
4	null

5) **Bubble Sort Algorithm:**

Its simple comparision based sorting algo. It works by repeatedly stepping through the list comparing each adjacent item and swapping them if they are in wrong order. Process is repeated until the array is sorted.

- Complexity:
  - → Time Complexity:
    - * Best case: $O(n)$ (if optimized with swapped flag).
    - * Worst case: $O(n^2)$ (if sorted in reverse order.)
    - * Average case: $O(n^2)$
  - → Space Complexity:
    - $O(1)$ → in-place algorithm:- requiring only a constant amount of extra memory for temp storage → only in swap.

Working:-

arr = {4, 3, 2, 1}.

pass → 1 :- largest = 4.

[4, 3, 2, 1] → 4 > 3 → swap.
[3, 4, 2, 1] → 4 > 2 → swap.
[3, 2, 4, 1] → 4 > 1 → swap.
[3, 2, 1, 4].

[3, 2, 1 | 4]

pass → 2 :

[3, 2, 1, 4] 3 > 2 → swap.
[2, 3, 1, 4] 3 > 1 → swap.
[2, 1, 3, 4] $3 \not> 4$ → (no swap)

[2, 1, | 3, 4]

and so on. we'll get [1, 2, 3, 4].

Best Case:- Array already sorted.
Worst Case:- Array is in reverse order.

6) Selection Sort Algorithm:

It repeatedly selects the smallest (or largest) element from unsorted part of array and swaps it with first unsorted element, effectively growing sorted portion one item at a time.

- Complexity:

→ Time → Best/worst/average :- $O(n^2)$

→ Space complexity:

$O(1)$ → in-place

Example.

arr = {29, 10, 14, 37, 13}

- find min (10), swap with 29 [10, 29, 14, 37, 13]
- find min (13), swap with 29 [10, 13, 14, 37, 29]
- find min (14), no swap needed. [10, 13, 14, 37, 29]
- find min (29), swap with 37.

[10, 13, 14, 29, 37].

Best Case:

still $O(n^2)$, even if array is sorted.

Worst Case:

Same — no difference due to how min is selected

7) Insertion Sort Algorithm:

It builds the final sorted array one element at a time. It works by taking each element from unsorted part and inserting it into its correct position in sorted part of array.

- Complexity:
  - → Time complexity:-
    - ★ Best case: $O(n)$ → (already sorted)
    - ★ Worst case: $O(n^2)$
    - ★ Average case: $O(n^2)$
  - → Space complexity:
    - $O(1)$ → require no significant extra space.

Working example:

arr = {5, 2, 4, 6, 1}

- 2 < 5 → insert before [2, 5, 4, 6, 1]
- 4 < 5 → shift + insert [2, 4, 5, 6, 1]
- 6 > 5 → no move.
- 1 < all → move all and insert at start.

[1, 2, 4, 5, 6]

Best Case:

Already sorted array.

Worst Case:

Reverse sorted array.

8) Merge Sort Algorithm:

It is a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the sorted halves into a single sorted array.

- Complexity:-

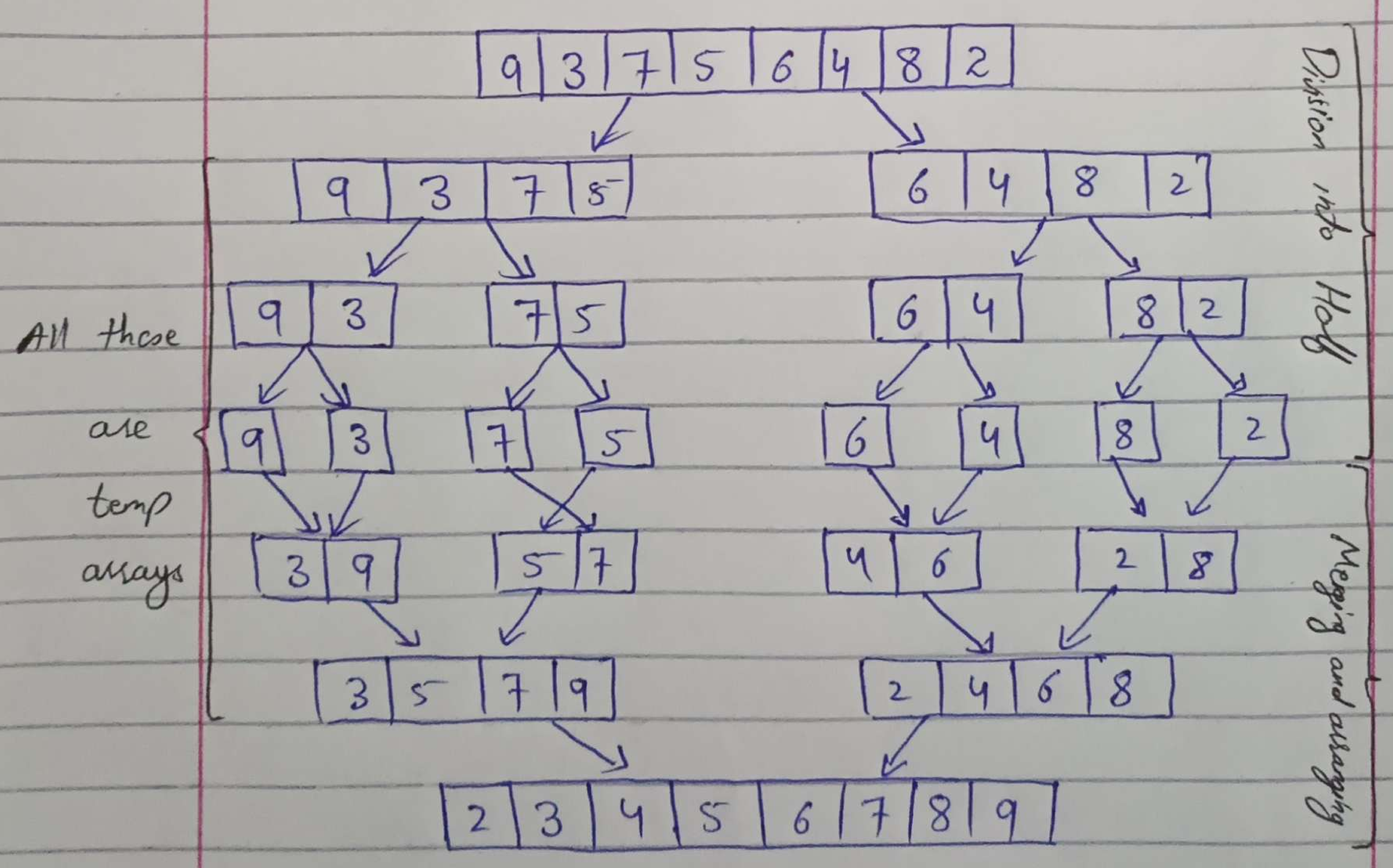
→ Time Complexity:-

+ Best/worst/average: $O(n \log n)$

→ Space Complexity:-

$O(n)$ → require additional memory to store temporary arrays used during merging process.

Example:-



Best/ Worst Case:

Always $O(n \log n)$, due to consistent splitting and merging.