Design and Analysis of Algorithm (KCS503)

Definition of Sorting problem through exhaustive search and analysis of Selection Sort through iteration Method

Lecture -3

A Sorting Problem (Exhaustive Search Approach)

Input: A sequence of *n* numbers $A1, A2, \ldots, An$.

Output: A permutation (reordering) $A1, A2, \ldots, An$ of the input sequence such that $A1 \leq A2 \leq \ldots \leq An$.

The sequences are typically stored in arrays.

A Sorting Problem (Exhaustive Search Approach)

Let there be a set of **four** digits and note that there are multiple possible permutations for the four digits. They are:

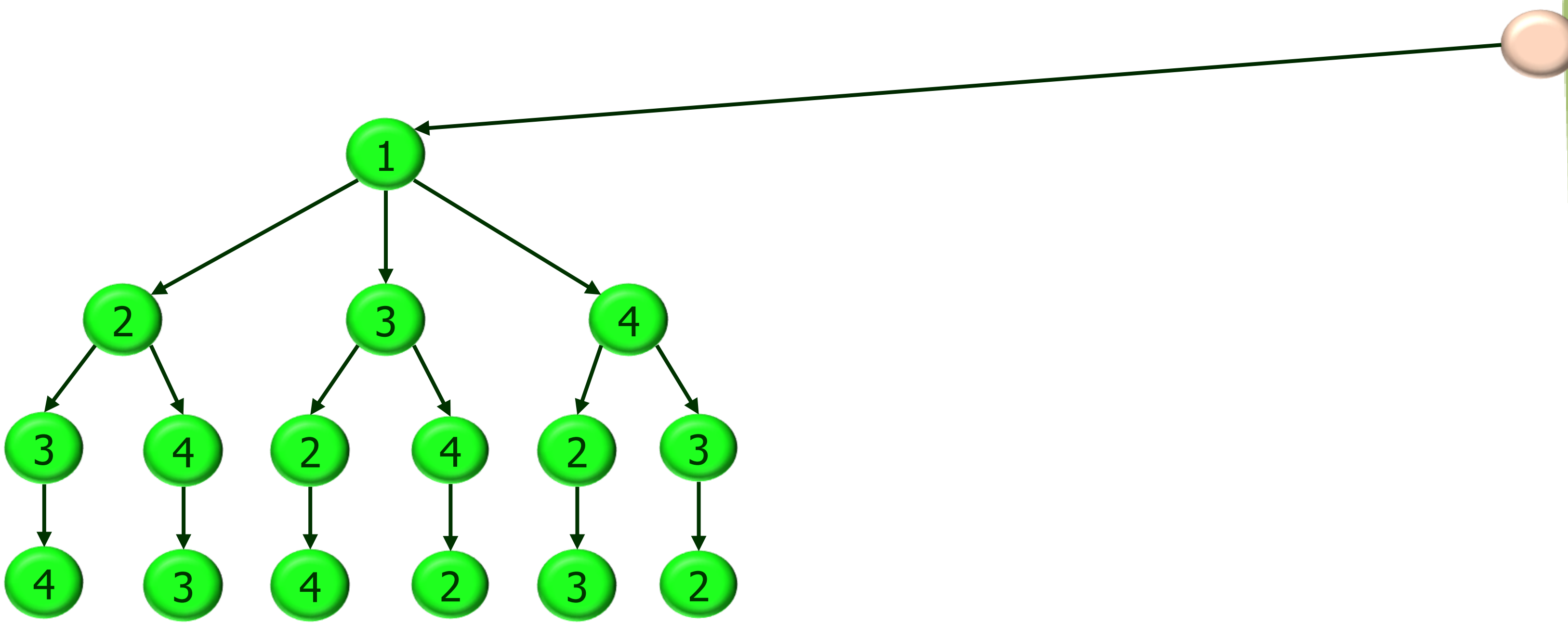
1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 3 1	3 4 1 2	4 3 1 2
1 4 3 2	2 4 1 3	3 4 2 1	4 3 2 1

- There are 24 different permutations possible. (as shown above)
- Only one of these permutations meets our criteria.

  (i.e. **A1 ≤ A2 ≤ …≤ An)** .

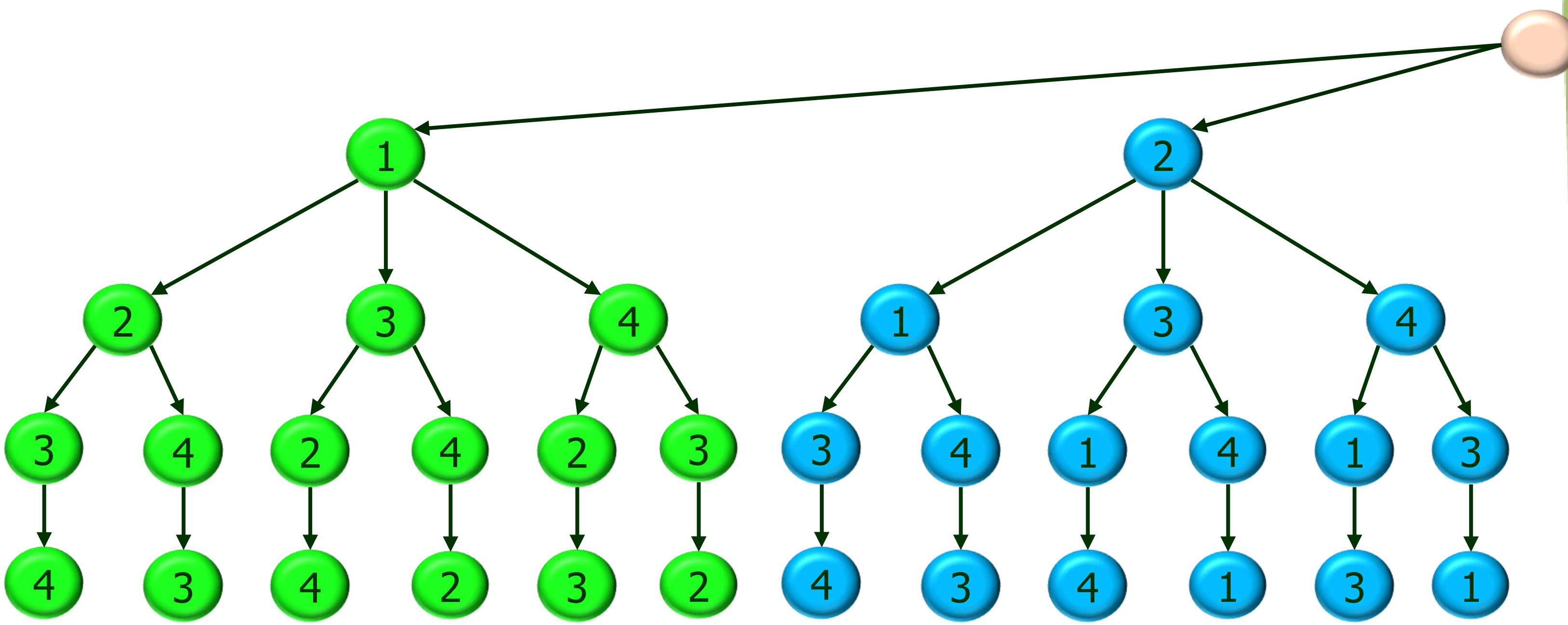
A Sorting Problem (Exhaustive Search Approach)

How to generate the 24 permutation ?



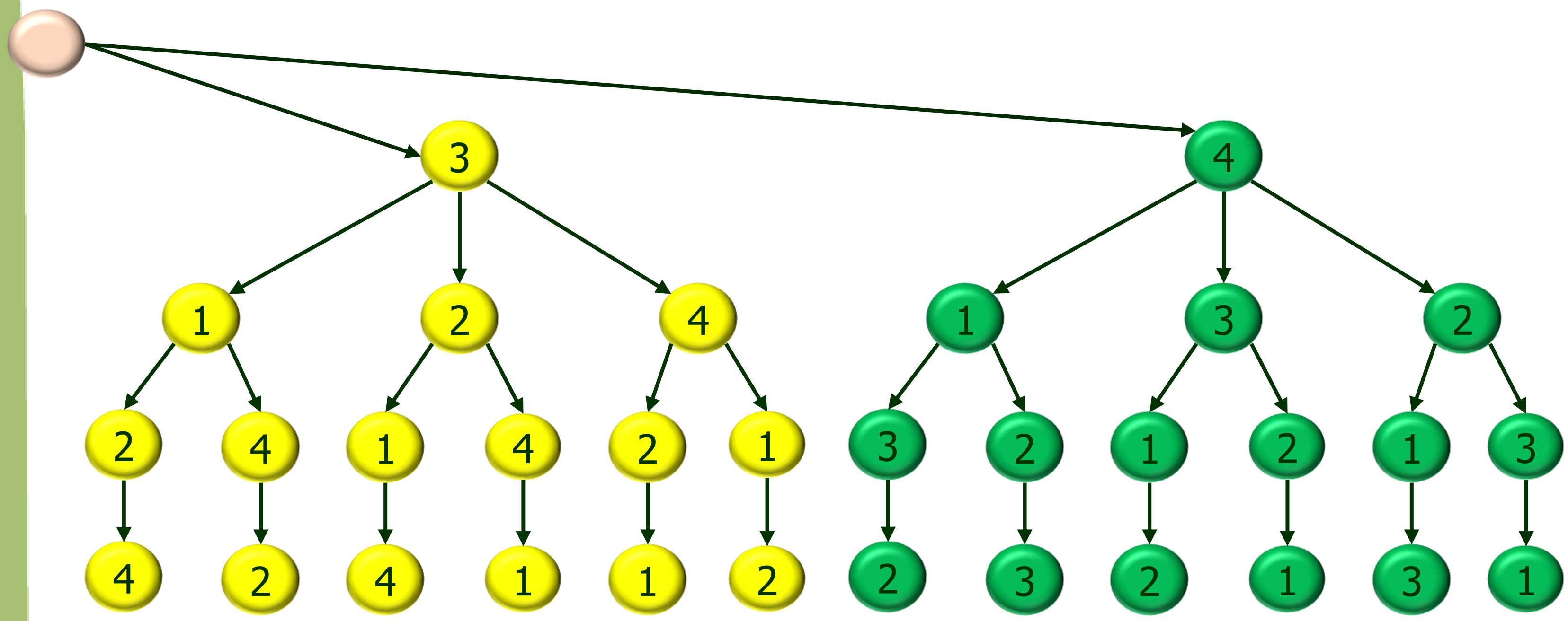
A Sorting Problem
(Exhaustive Search Approach)

How to generate the 24 permutation ?



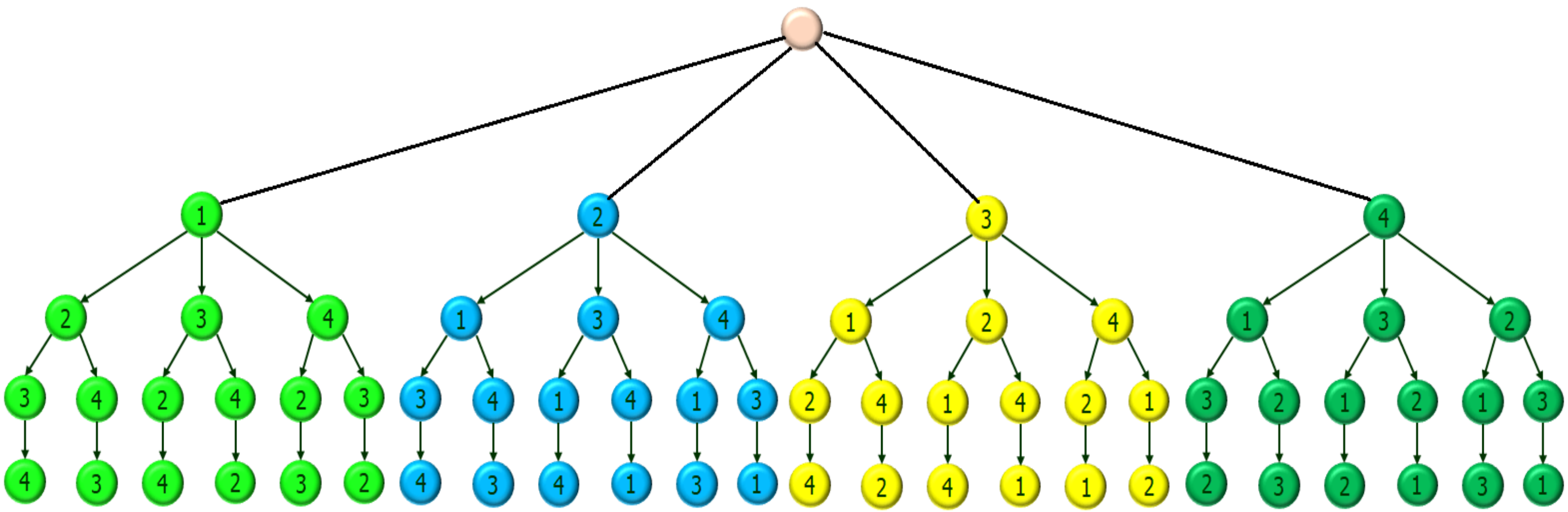
A Sorting Problem
(Exhaustive Search Approach)

How to generate the 24 permutation ?



A Sorting Problem
(Exhaustive Search Approach)

An in-depth look at the analysis of the 24 permutations of four digits



A Sorting Problem (Exhaustive Search Approach)

Let there be a set of **four** digits and note that there are multiple possible permutations for the four digits. They are:

1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 3 1	3 4 1 2	4 3 1 2
1 4 3 2	2 4 1 3	3 4 2 1	4 3 2 1

- There are 24 different permutations possible. (as shown above)
- Only one of these permutations meets our criteria.

  (i.e. **A1 ≤ A2 ≤ …≤ An)** . **(1 2 3 4)**

A Sorting Problem (Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. $a1 \leq a2 \leq \cdot \cdot \cdot \leq an$).

Step 3: Once we get it , we got the victory.

A Sorting Problem (Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. $a1 \leq a2 \leq \cdot \cdot \cdot \leq an$).

Step 3: Once we get it , we got the victory.

How we do this in algo based?

For each permutation P ∈ set of n! permutations:

```
        if (a1 ≤ a2 ≤ · · · ≤ an) == permutation set[p]:
                print (permutation set[p])
```

A Sorting Problem
(Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. $a1 \le a2 \le \cdot \cdot \cdot \le an$).

Step 3: Once we get it , we got the victory.

Exhaustive Search

How we do this in algo based?

For each permutation P ∈ set of n! permutations:

if ($a1 \le a2 \le \cdot \cdot \cdot \le an$) == permutation set[p]:

print (permutation set[p])

A Sorting Problem (Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: **Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. a1 ≤ a2 ≤ · · · ≤ an).**

Step 3: Once we get it , we got the victory.

Exhaustive Search

How we do this in algo based?

For each permutation P ∈ set of n! permutations:

if (a1 ≤ a2 ≤ · · · ≤ an) == permutation set[p]:

print (permutation set[p])

$\boldsymbol{Complexity = O(n! * n) time}$

A Sorting Problem
(Selection Sort Approach)

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

A Sorting Problem (Selection Sort Approach)

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Lets consider the following array as an example:
A [] = (7, 4, 3, 6, 5).

A Sorting Problem
(Selection Sort Approach)

Lets consider the following array as an example:
A [] = (7, 4, 3, 6, 5).

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. After going through the entire array, it is evident that 3 is the lowest value, with 7 being stored at the first position.
- Thus, replace 7 with 3. At the end of the first iteration, the item with the lowest value, in this case 3, at position 2 is most likely to be at the top of the sorted list.

A Sorting Problem (Selection Sort Approach)

Lets consider the following array as an example:
A [] = (7, 4, 3, 6, 5).

- 1st Iteration

Value	7	4	3	6	5
index	0	1	2	3	4
Value	7	4	3	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4

A Sorting Problem
(Selection Sort Approach)

Lets consider the updated array as an example:
A [] = (3, 4, 7, 6, 5).

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
- Using the traversal method, we determined that the value 12 is the second-lowest in the array and thus should be placed in the second position. So no need of swapping.

A Sorting Problem (Selection Sort Approach)

Lets consider the following array as an example:
A [] = (7, 4, 3, 6, 5).

- 2nd Iteration

Value	3	4	7	6	5
index	0	1	2	3	4

Value	3	4	7	6	5
index	0	1	2	3	4

Value	3	4	7	6	5
index	0	1	2	3	4

Value	3	4	7	6	5
index	0	1	2	3	4

A Sorting Problem (Selection Sort Approach)

Lets consider the updated array as an example:
A [] = (3, 4, 7, 6, 5).

- For the third position, where 7 is present, again traverse the rest of the array in a sequential manner.
- Using the traversal method, we determined that the value 5 is the third-lowest in the array and thus should be placed in the third position.

A Sorting Problem (Selection Sort Approach)

Lets consider the following array as an example:
A [] = (7, 4, 3, 6, 5).

- 3rd Iteration

Value	3	4	7	6	5
index	0	1	2	3	4

Value	3	4	7	6	5
index	0	1	2	3	4

Value	3	4	5	6	7
index	0	1	2	3	4

Value	3	4	5	6	7
index	0	1	2	3	4

A Sorting Problem
(Selection Sort Approach)

Lets consider the updated array as an example:
A [] = (3, 4, 5, 6, 7).

- Similarly we execute it for fourth and fifth iteration and finally the sorted array is looks like as below:

Value	3	4	5	6	7
index	0	1	2	3	4

A Sorting Problem
(Selection Sort Algorithm)

SELECTION SORT(arr, n)

```
Step 1: Repeat Steps 2 and 3 for
        i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n,
        pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT
```

SMALLEST (arr, i, n, pos)

```
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
        if (SMALL > arr[j])
            SET SMALL = arr[j]
        SET pos = j
        [END OF if]
        [END OF LOOP]
Step 4: RETURN pos
```

Use C programming language to convert the above into a programme

A Sorting Problem (Selection Sort Complexity)

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

A Sorting Problem
(Selection Sort Complexity)

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of O(n2) for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is O(n2), in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also O(n2), which occurs when we sort the descending order of an array into the ascending order.

Thank You