Module 2

Arrays

Array

- An array is a list of a ***finite*** number of ***homogenous*** (same type) data elements such that:
  - Elements of the array are referenced by an index set (***n*** consecutive numbers)
  - Array elements are stored in successive memory locations

• Array **A** of size n	A[1..n]
• Array Attribute:	n → Length or Size of the array
• Size of (sub)array	(Upper Index - Lower Index + 1) = (n-1+1) = n
• Array Indices	1 to n
• i^{th} Element	A[i]

Array Operations

- Traversal
- Insertion
- Deletion
- Search → Linear Search O(n) and Ω(1)
- Sorting
- Merging

Visualizing Arrays

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
2	3	4	5	8	9

A[1]	2
A[2]	3
A[3]	4
A[4]	5
A[5]	8
A[6]	9

- Difference programming language → Different rules
  - Name of the array
  - Data type of the array
  - Index set of the array

- Static vs. Dynamic memory allocation

Representation of Array in Memory

- Array : Linear Data Structure
  - Stored in successive memory locations
- Memory locations: Also called cells

$$\text{Address}(A[k]) = \text{Base Address}_{(\text{Lower Index})} + (k - \text{Lower Index}) * w$$

w: Number of memory words per cell

Base Address →

1000	2	**A[1]**
1001	3	**A[2]**
1002	4	**A[3]**
1003	5	**A[4]**
1004	8	**A[5]**
1005	9	**A[6]**

Array Traversal

- Suppose we want to print the contents of an array
  - Traversal (visiting)

```
TRAVERSAL(A, n)
i=1
while i ≤ n
      Print A[i]
      i = i + 1
```

```
TRAVERSAL(A)
for i = 1 to n
      Print A[i]
```

- Time Complexity: $\Theta(n)$

Space Complexity: $\Theta(n)$

Insertion in an Array

- Insertion: Operation of adding another element to the array
  - Insertion at the end: Example – Insert 99 at the end of an array

Index →	1	2	3	4	5	6
	22	33	44	55	66	77

- Array size = Array size + 1

Index →	1	2	3	4	5	6	7
	22	33	44	55	66	77	99

Insertion in an Array

- Insertion at the beginning: Example – Insert 99 at the beginning of an array

Index →	1	2	3	4	5	6
	22	33	44	55	66	77

- Array size = Array size + 1

Index →	1	2	3	4	5	6	7
	99	22	33	44	55	66	77

Insertion in an Array

- Insertion at the middle (anywhere else): Example – Insert 99 at index $i = 3$

Index →	1	2	3	4	5	6
	22	33	44	55	66	77

- Array size = Array size + 1

Index →	1	2	3	4	5	6	7
	22	33	99	44	55	66	77

Array Insertion Algorithm

Insert an element ***x*** in array ***A*** of size ***n*** at index **k**

```
INSERT(A,n,k,x)
j = n
while j ≥ k
        A[j+1] = A[j]
        j = j – 1
A[k] = x
n = n + 1
```

- Time Complexity: O(n) and $\Omega(1)$

Deletion in an Array

- Deletion: Operation of removing one element from the array
  - Store the element in a variable for future use
- Deletion at the end: Example – Delete element at the last array index

Index →	1	2	3	4	5	6
	22	33	44	55	66	77

- Array size = Array size - 1

Index →	1	2	3	4	5
	22	33	44	55	66

Deletion in an Array

- Deletion at the beginning: Example – Insert element at first array index

Index →	1	2	3	4	5	6
	22	33	44	55	66	77

- Array size = Array size - 1

Index →	1	2	3	4	5
	33	44	55	66	77

Deletion in an Array

- Deletion at the middle (anywhere else): Example – Delete element at index $i = 3$

Index →	1	2	3	4	5	6
	22	33	44	55	66	77

- Array size = Array size - 1

Index →	1	2	3	4	5
	22	33	55	66	77

Array Deletion Algorithm

Delete element at index **k** in array ***A*** of size ***n*.** Store the value of deleted element in variable ***x***

```
DELETE(A,n,k,x)
x = A[k]
while k ≤ n-1
      A[k] = A[k+1]
      k = k + 1
n = n - 1
```

- Time Complexity: O(n) and $\Omega(1)$

Exercise

- Delete array element by value ***v***
  - Do a linear search for the value ***v***
  - Store the index i at which ***v*** is found
  - Delete element at index i

- Find the time complexity: O and Ω

Searching in an Array

- Linear Search: Find the location (index) of an item ***v*** in Array ***A*** of size ***n*** and store the location in a variable ***loc***

```
SEARCH(A, n, v, loc)
loc = 0
for i = 1 to n
      if A[i] == v
            loc = i
            break
```

- Time Complexity: $\Omega(1)$ and $O(n)$ Average Case $\Theta(n)$

- Binary Search: Sorted Array only

Binary Search

Binary Search: Find the location (index) of an item ***v*** in an sorted Array ***A*** of size ***n*** and store the location in a variable ***loc***

```
BINARY-SEARCH(A, n, v, loc)
loc = 0, start = 1, end = n
while start ≤ end
      mid = (start + end)/2                Floor/Ceiling/Integer
      if A[mid] == v
            loc = mid
            break
      else if v<A[mid]
            end = mid - 1
      else                            // if v>A[mid]
            start = mid + 1
```

Binary Search

- Time Complexity

Iteration No	(Sub)Array Size
1	n/2
2	$n/2^2$
3	$n/2^3$
4	$n/2^4$
..	..
k	$n/2^k$

When subarray reduces to single element, loop terminates

$n/2^k = 1$
$n = 2^k$
$k = \log_2 n$

Time Complexity: $\Omega(1)$ and $O(\log_2 n)$

- Binary search in
  - Dictionary
  - Telephone directory
  - IITR student list sorted by Enroll no

Sorting

- Sorting: A process of arranging the elements of a list in a certain order
  - Ascending → Increasing
  - Descending → Decreasing

Non-Decreasing
Non-Increasing

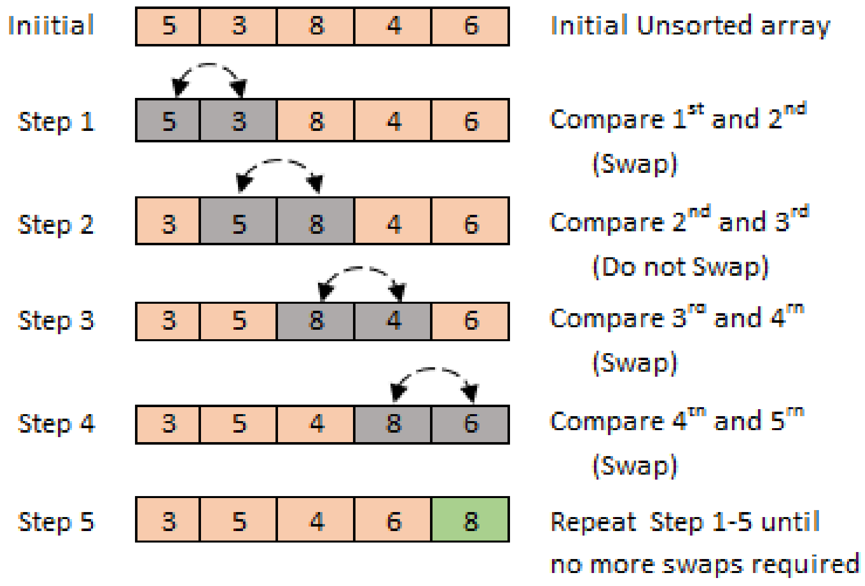
- One of the most crucial problem in Computer Science
  - Sort elements
  - Reduces complexity of other problems: Binary Search
- Sorting Algorithms
  - Bubble Sort, Selection Sort, Insertion Sort, Quick Sort
  - Merge Sort, Heap Sort, Topological Sort
  - Bucket Sort, Radix Sort….

Classification of Sorting Algorithms

- Classification based on parameters
  - Number of comparisons
  - Number of swaps (or inversions)
  - Memory usage: Constant amount of memory → In-place sorting
  - Stability: Retain the relative positions of equal elements after sorting
  - Adaptability: Complexity of algorithm changes based on pre-sortedness of input
  - Internal vs. External: Uses internal memory (main/RAM) or external (tape/drive)
  - Online vs. Offline: Algorithm can sort the elements as it receives them

Bubble Sort

- Iterate through the input array from first element to last
  - Compare each pair of adjacent element
  - Swap elements if needed
- Largest element “bubble” up to the top of the list
- Example:



First pass only

Bubble Sort Algorithm

```
BUBBLE-SORT(A, n)
for p = n to 1
      for i = 1 to p-1
            if A[i] > A[i+1]
                  temp = A[i]
                  A[i] = A[i+1]
                  A[i+1] = temp
```

- In-place Sorting
- Stable
- Not Adaptable
- Internal
- Offline

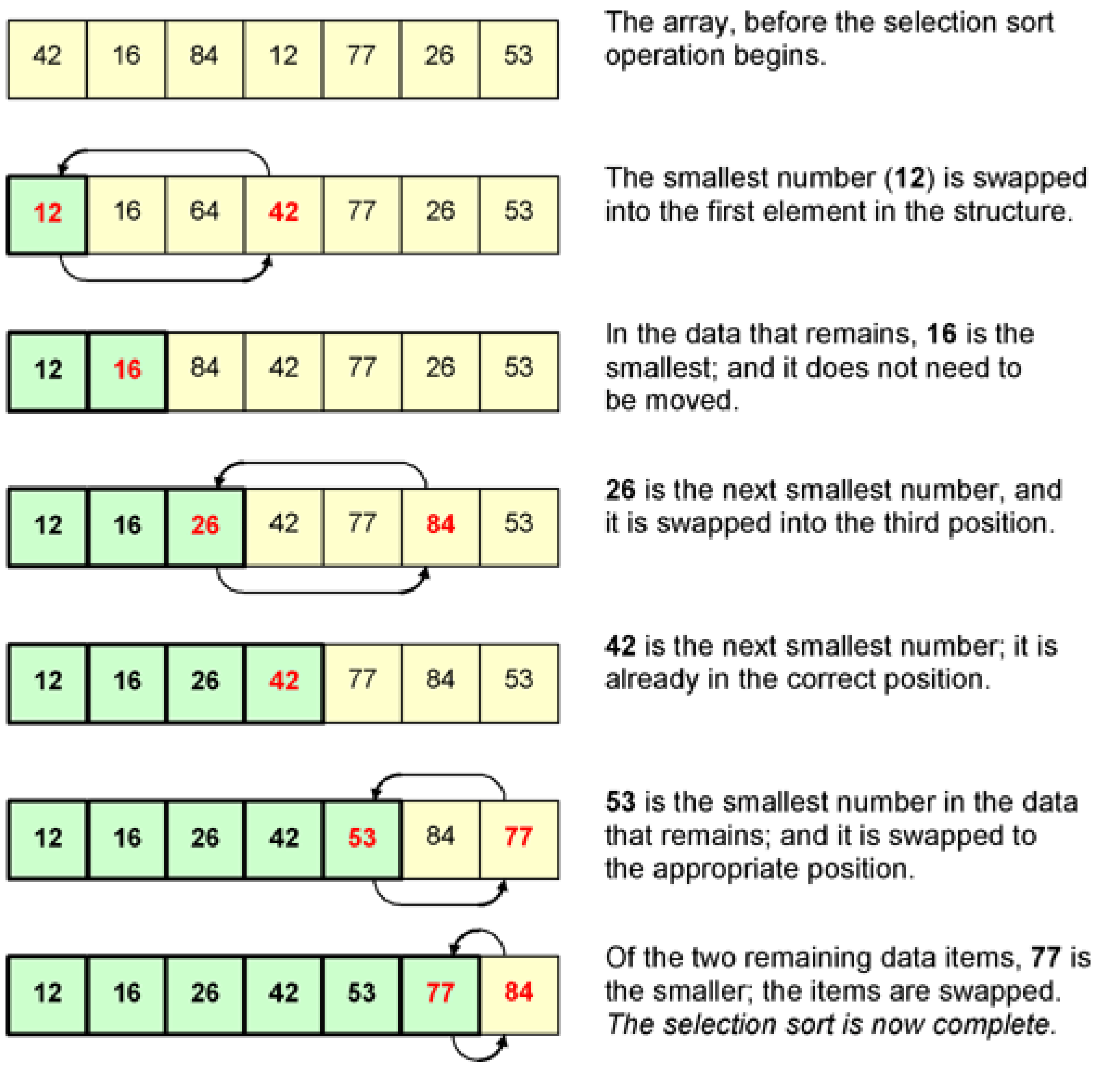
Time Complexity: $\Theta(n^2)$

Improved Bubble Sort?

- Best Case → Already Sorted Array
- Make it Adaptable

Selection Sort

- In each iteration
  - Find the minimum element in the list
  - Swap it with the value in the current position (starting from index 1)
- Repeat until all elements are sorted



Selection Sort Algorithm

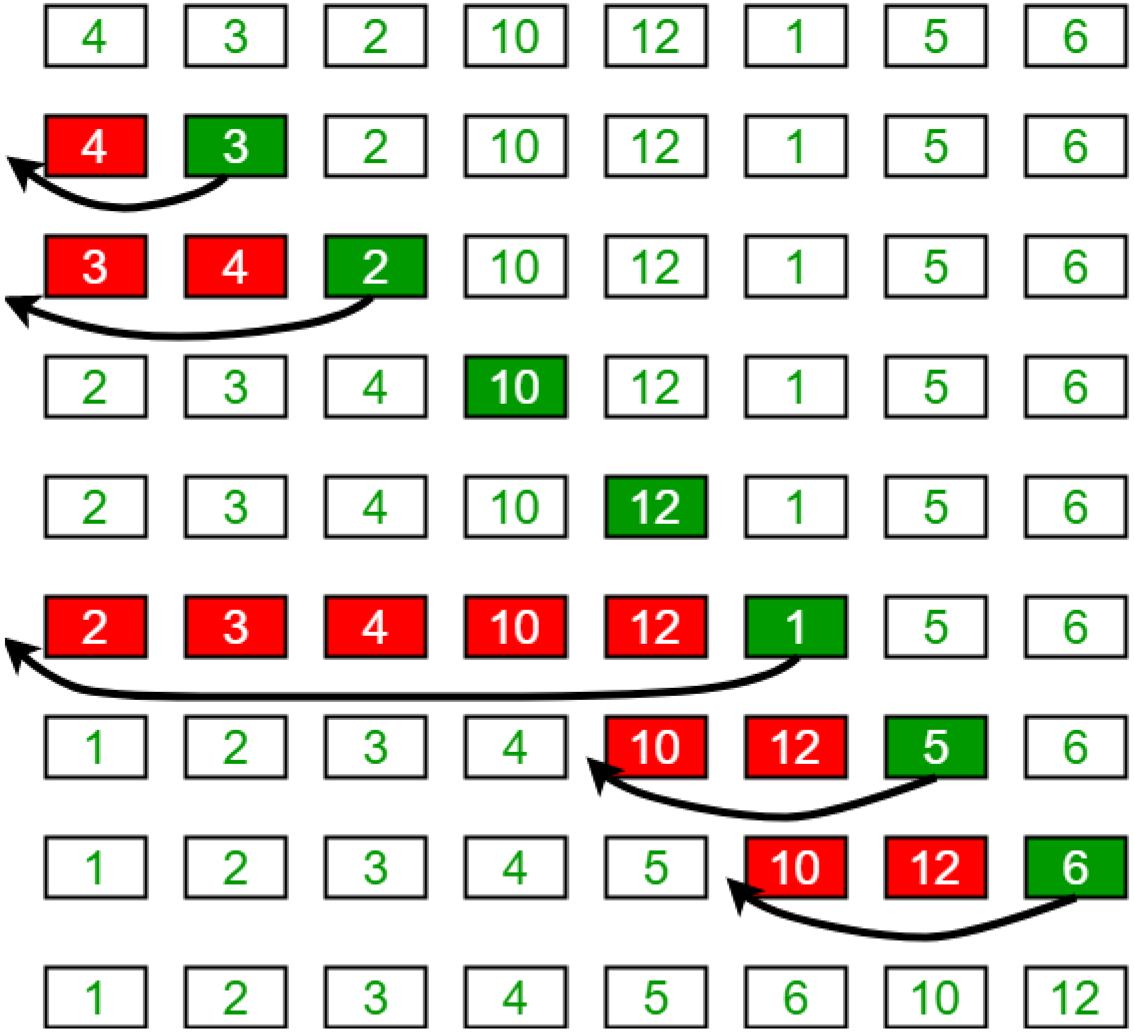
```
SELECTION-SORT(A, n)
for i = 1 to n-1
    min = i
    for j = i+1 to n
        if A[j] < A[min]
            min = j
    temp = A[min]
    A[min] = A[i]
    A[i] = temp
```

- In-place Sorting
- Not Stable
- Not Adaptable
- Internal
- Offline

Time Complexity: $\Theta(n^2)$

Insertion Sort

- In each iteration
  - Remove an element from the list
  - Insert that element into the correct position in the list being sorted



Insertion Sort Algorithm

```
INSERTION-SORT(A, n)
for i = 2 to n
    v = A[i]
    j = i
    while j > 1 and A[j-1] > v
        A[j] = A[j-1]
        j = j-1
    A[j] = v
```

- In-place Sorting
- Stable
- Adaptable
- Internal
- Online

Time Complexity: $O(n^2)$ and $\Omega(n)$

Linear Sorting Algorithms

- Comparison based vs. Linear sorting algorithm
  - Best Case for comparison based algorithm: $O(n\ logn)$
  - Best Case for linear algorithms: $O(n)$
    - Make some assumption on the input
- Linear Sorting Algorithms
  - Bucket Sort
  - Radix Sort

Bucket Sort

- Assumption: Elements to be sorted belong to a fix set [1..K]
- How it works
  - Create K buckets (bins)
  - Scan the list and put the elements in the buckets (count the elements)
  - Output elements in the buckets from 1 to K

Bucket Sort Algorithm

```
BUCKET-SORT(A, n, K)
Declare array Bucket[1..K]
for j = 1 to K
    Bucket[j] = 0
for i = 1 to n
    Bucket[A[i]] = Bucket[A[i]] + 1
i = 0
for j = 1 to K
    for l = Bucket[j] to 1
        A[i] = j
        i = i + 1
```

Time Complexity: $\Theta(n)$

Radix Sort

- Sort number based on individual digits
- How it works
  - Take the least significant digit of each element
  - Sort the list based on least significant digit but maintain the order of elements having the same least significant digit
  - Repeat the process for higher significant digits
- Internally uses a variation Bucket Sort

Radix Sort Algorithm

```
BUCKET-SORT(A, n, d)                                \\ d: number of digits
1 for i = 1 to d
2    BUCKET-SORT(A, n, b) for i^th digit             \\ Base: 10 for decimal
```

Complexity: $\Theta(nd)$

Assignment: Write a program for BUCKET-SORT

Other Sorting Algorithms

- Quick Sort
- Merge Sort
- Heap Sort
- Topological Sort

References

- Saymour L., **“Data Structures”,** Schaum’s Outline Series, McGraw Hill, Revised First Edition
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, **“Introduction to Algorithms”**, The MIT Press
- Sahni, S., **“Data Structures, Algorithms, and Applications in C++”,** WCB/McGraw-Hill
- Algorithms, Video Lectures by Abdul Bari, 1.1-1.12

https://www.youtube.com/playlist?list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O