Page No:
Date: / /

Q) Define asymptotic notation and explain its importance in analyzing algorithm efficiency.

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.

* Asymptotic notation:

It is a mathematical way to describe the growth rate of an algorithm's running time or space requirement as the input size (n) becomes very large

- It helps express the efficiency of an algorithm

* Common asymptotic notations:

1) Big O (O): Describes the upper bound — the worst case growth rate

eg: $O(n^2)$

2) Omega (Ω): Describes the lower bound — the best case growth rate

eg: $\Omega(n)$

3) Theta (Θ): Describes the tight bound — when upper and lower bounds are the same.

eg: $\Theta(n \log n)$

* Importance in Algorithm efficiency

- Compares efficiency
- Predicts scalability
- Focuses on growth trends
- Guides optimization

Page No:
Date: / /

Q) Define time-space trade off with an example.

=> A Time-space trade off is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:

- Either less time and by using more space.
- Or vice versa.

* Types: Compressed or Uncompressed data, Re-rendering or stored images, Smaller code or loop unrolling, Lookup tables or recalculation.

eg: Lookup tables:

- Suppose you need to compute factorials of a number.
- Without extra space: Compute n! every time
  ⇒ takes more space
- With extra space:
  store previously computed factorials in an array ⇒ uses more space but takes less time.

Q) Define Complexity and describe its types.

⇒ Complexity refers to the measure of the amount of time and space an algorithm requires to solve a problem as a function of input size(n).

Types:

1) Time Complexity: measures the total time an algorithm takes to complete

2) Space Complexity: measures the total space/memory an algorithm uses during execution.

eg:
```
int SumA (int arr[], int n) {
    int sum = 0;
    for (int i=0; i<n; i++) {
        sum += arr[i];
    }
    return sum;
}
```
T.C ⇒ O(n) ; S.C = O(1)

	T.C	S.C
Definition	Definition	Definition
focus	Speed / execution time	memory / storage used
Depends on	No. of steps	No. of variables, data structure and recursion depth
Goal	To minimize execution time	To minimize memory usage

Q) Define and write short notes on types of recursion with ex.

⇒ The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called recursive function.

Types:

1) Direct Recursion:

When a function calls itself from within itself.

Categorised into 4 types.

i) Tail Recursion:

If a recursive function calls itself and that recursive call is the last statement in the function.

eg:
```
~~static void~~
int fun(int n) {
    if (n>0) { ~~cout<<n;~~
        cout<<n;
        fun(n-1); //last statement
    }
}
```

ii) Head Recursion:

If a recursive function calls itself and that call is the first statement in the function then .....

eg:
```
int fun(int n) {
    if (n>0) {
        fun(n-1); // first statement.
        cout<<n;
    }
}
```

iii) Tree recursion:

If a recursive function calls itself for more than one time then ....

- If one time → linear recursion

eg:
```
int fun(int n) {
    if (n>0) {
        cout<<n;
        fun(n-1);   // calls once
        fun(n-1);   // calls twice.
    }
}
```

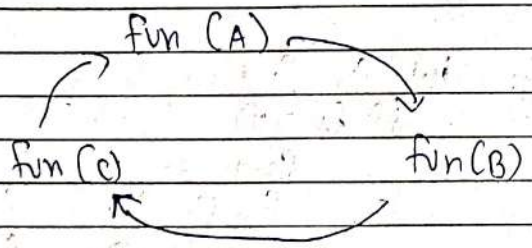
iv) Nested Recursion

=> In this recursion, a recursive function will pass the parameter as a recursive call. That means recursion inside recursion.

```
eg: int fun (int n) {
      if (n > 100) {
        return n-10;
      }
      return fun (fun (n+11));
    }
```

2) Indirect Recursion:

=> There may be more than one functions and they are calling another function in a circular manner.



3) Applications of multidimensional arrays with ex.

1) Matrix representation.

- Helps to perform mathematical operations

eg: int matrix[2][3] = { {1,2,3}, {4,5,6} };

2) Image processing:

Use: Digital images are represented as 2D/3D arrays where each element corresponds to a pixel's intensity / Color value.

eg: A Coloured image can be stored as image[height][width][3] representing RGB (Red, green, blue) channels.

3) ~~Game development~~: Storing Tabular data.

Used to represent tables such as marks of students in different subjects.

eg: int marks[5][3]; // 5 students, 3 ~~marks~~ subjects

4) Game developments

Uses 2D arrays to represent gameboards or maps.

eg: char board[8][8]; // Chessboard representation

5) Spreadsheet Representation:

- Multi dim - Arrays can represent data in rows and columns like excel sheet.

eg: Sheet [row] [column] stores data of a specific cell.

Q) Describe row major/column major representation of arrays with diagram.

⇒ 1) Row major:

Row major assigns successive elements, moving across the row and then down to the next row, to successive memory locations.
Or, elements are arranged in row-wise fashion.

eg: Consider a 2D array.

A[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}}

⇒ Memory stores elements as
1, 2, 3, 4, 5, 6, 7, 8, 9.

- first row → 1, 2, 3
- Second row → 4, 5, 6
- Third row → 7, 8, 9

Address formula:

A[i][j] = Base(A) + [i x M + j] x W

→ Base (A) → starting address of array
M → total no. of columns
W → size of each element in bytes

for 2-D
or, A[i][j] = B + [M * (I - LR) + (J - LC)] * W

I → row subscript
J → column subscript
B → base address
W → storage size
LR → lower limit of row
LC → lower limit of column
M → total no. of columns.

2) Column major

If elements of an array are stored in a column major fashion then that means moving across the column and then to the next column, then in column-major order.

eg: A[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}}

⇒ Memory store elements as
1, 4, 7, 2, 5, 8, 3, 6, 9.

- first column → 1, 4, 7
- Second column → 2, 5, 8
- Third " → 3, 6, 9

5) Spreadsheet Representation:

- Multidim-Array can represent data in rows and Column like excel sheet

eg: Sheet [row] [column] stores data of a specific cell.

Q) Describe row major/column major representation of array with diagram.

=> 1) Row major:

Row major assigns successive elements, moving ~~across~~ across the row and then down to the next row, to successive memory locations.
or, elements are arranged in row-wise fashion.

eg: Consider a 2D array.

- $A[3][3] = \{\{1,2,3\}, \{4,5,6\}, \{7,8,9\}\}$

=> Memory stores elements as
1, 2, 3, 4, 5, 6, 7, 8, 9.

- first row -> 1, 2, 3
- Second row -> 4, 5, 6
- Third row -> 7, 8, 9

Address formula:

$A[i][j] = Base(A) + [i \times M + j] \times W$

=> Base (A) -> starting address of array
M -> total no. of Columns
W -> size of each element in byte)

For 2-D

or, $A[i][j] = B + [M * (I - LR) + (J - LC)] * W$

I -> row subset
J -> column subset
B -> base address
W -> storage size
LR -> lower limit of row
LC -> lower limit of column
M -> total no. of columns.

2) Column major

If elements of an array are stored in a column major fashion, then it means moving across the column and then to the next column, then in column-major order.

eg: $A[3][3] = \{\{1,2,3\}, \{4,5,6\}, \{7,8,9\}\}$

=> Memory store elements as
1, 4, 7, 2, 5, 8, 3, 6, 9.

- first Column -> 1, 4, 7 ; • Second column -> 2, 5, 8.
- Third " -> 3, 6, 9

Address formula:

$A[i][j] = Base(A) + [j * N + i] \times W$

- Base(A) → Starting address of array
- N → total no of rows
- W → size of each element

For 2-D:

$A[i][j] = B + [(I - LR) + N * (J - LC)] * W$

B → base address
I → row subset
J → column subset
W → storage size
LR → Lower limit of row
LC → Lower limit of column
N → total no of rows

Q) Derive the index formula for arrays

1) 1-D Array.

⇒ Let A[L_1, ... U_1] be a 1D array stored in memory.

Where,
- L_1 → lower bound, U_1 → Upper bound
- W → size of each element
- Base(A) → address of first element

Add. formula:

$A[Index] = B + W * [Index - LB]$

- Index ⇒ address of element whose address is to be found
- B → base address of array
- W → size of element in bytes
- LB → Lower bound of index

eg: int A[5] = {1,2,3,4,5}, W = 4
Address(A[3]) = Base(A) + (3-0) * 4

2) 2-D array

Let A[L_1 ... U_1; L_2 ... U_2] be a 2D array with
- $N = U_1 - L_1 + 1$ → no of rows
- $M = U_2 - L_2 + 1$ → No of columns
- W = size of one element in byte.

- Row major:
  - no of element before A[i][j] → $(i - L_1) * M + j - L_2$
  - Multiply by size of element
    ⇒ $[(i - L_1) * M + (j - L_2)] * W$
  - Add formula →

$$A[i][j] = B + [(i - L_1) * M + (j - L_2)] * W$$

Here, L_1 ⇒ lower limit of row | M ⇒ no of columns
L_2 ⇒ lower limit of column

- Column Major:
  - No. of element before A[i][j]
    - ~~(j-L2)*M +~~
    - $(i-L_1) + (j-L_2) * N$
  - Multiply by size of element
    - $[(i-L_1) + (j-L_2) * N] * W$
  - Address formula =>

$$A[i][j] = B + [(i-L_1) + (j-L_2) * N] * W$$

3) 3-D Array

Let $A[L_1 \dots U_1, L_2 \dots U_2, L_3 \dots U_3]$ be a 3D array with:

- Row major:

Address formula:

$$A[i][j][k] = B + W * [N * L * (I-x) + N * (J-y) + (k-z)]$$

Where,

B = base address
W = size of one element
N = total no. of cells
M = Total no. of rows
L = Total no. of columns
x = L.B of row
y = LB. of column
z = LB of height

- Column Major

$$A[i][j][k] = B + W * [N * L * (I-x) + (J-y) + N * (k-z)]$$

Q) Difference b/w Linear and binary search

Linear Search:

- Searches element sequentially.
- Array can be unsorted
- Best time complexity, O(1)
- Worst time complexity, O(n).
- Slow for large data sets

Linear Search:

- Searches by repeatedly dividing a sorted array
- Array must be sorted
- Best time Complexity: O(1)
- Worst : O(log n)
- fast for large data set

a) Linear search Algorithm:

=> Input: Array, key k
Output: Index of k if found, else -1

1) Start
2) Set i=0
3) while i<n
   a. If A[i] == k, => return i
   b. Else, i = i+1 &
4) Return -1
5) end

Pros: • Simple to implement
• Works for unsorted arrays

Cons: • Inefficient for large arrays
• Must check every element in worst case

Explanation:

- Start from first element & and compare it with k.
- If found return index
- If not move to next element until the end of array.

b) Binary Search Algorithm

=> Input: Sorted Array [0,.... n-1], key k
Output: Index of k if found, else -1

1) Start
2) Set low=0, high = n-1
3) while low <= high
   a. mid = low + (high-low)/2
   b. If A[mid] == k, =>return mid
   c. Else if A[mid] < k, low = mid+1
   d. Else high = mid-1
4) Return -1
5) end.

Pros: • fast and efficient
• Simple logic
• Low no. of comparisons.

Cons: • requires sorted array
• Not ideal for small, dynamic array
• Random access needed.

Q) Insertion sort and Selection sort.

~~Insertion sort~~ ~~Insertion sort~~

Insertion Sort:

⇒ • Inserts element at correct position in sorted portion.

• Best case time complexity $O(n)$
• Worst " " " $O(n^2)$

• Faster for nearly sorted array
• ~~Slower~~ Stable.

Selection Sort:

⇒ • Finds minimum element and swaps it to the beginning

• $O(n^2)$
• $O(n^2)$

• ~~Slower than~~ nearly Not adaptive.
• Not stable

Q) Brief note. Searching Techniques

⇒ Searching is the process of finding an element in a dataset.

Common Techniques:

1) **Linear Search**

• Checks each element sequentially
• Works on unsorted arrays.

2) **Binary Search**

• Efficient for sorted arrays
• Divides the array by half repeatedly.

Q) Describe how sparse matrices are represented and explain any one representation method.

⇒ A matrix is 2D data object made of m rows and n columns. So it has total mxn values. If most of the elements of the matrix have 0 value, then it is called sparse matrix.

• Storage: less memory can be used to store only those non-zero elements. saved
• Computing time: by logically designing a data structure

traversing only non-zero element-

Representation

1) Array representation

2) Linked list representation.

Method 1)

→ Using Arrays

2D arrays are used to represent a sparse matrix in which there are three rows named as

Row, Column and value

↓ Index of row of non-zero element

↓ Index of Column of non-zero element

↓ value of non-zero element at index

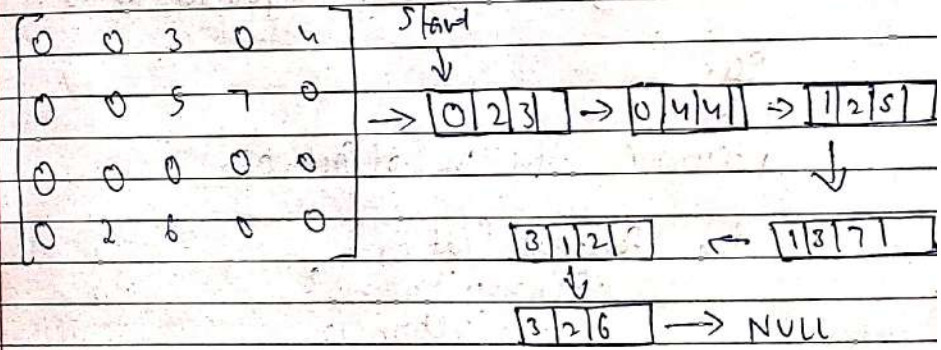
```
int sparsematrix[][] = {
        {0,0,3,0,4},
        {0,0,5,7,0},
        {0,0,0,0,0},
        {0,2,6,0,0}
};
int compactmatrix[][] = new int[3][size];
```

```
if(sparsematrix[i][j] != 0) {
    compactmatrix[0][k] = i;
    compact " [1][k] = j;
    compact " [2][k] = sparsematrix[i][j];
    k++;
}
```

Method 2: Using Linked List

Here, each node has 4 fields. These 4 fields are defined as:

Row, Column, Value, Next node
↳ Address of next node.



Node Structure:

Row	Column	Value	Add of next node

- Saves memory
- fast traversal
- Suitable for large sparse datasets

Q) Describe ~~different types of~~ recursion with example.

Q) Discuss removal of recursion.

=> By replacing the selection structure with a loop, recursion can be eliminated.

ways to remove recursion:

1) Iteration
2) Macro expansion
3) Refactoring
4) Stack
5) Generalization
6) ~~Comp~~ Computation traces

eg.

Recursive function of factorial.

```
int factorial (int n) {
    if (n==0 || n==1) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
```

Itrative version:

```
int factorial (int n) {
    int fact = 1;
    for(int i=1; i<=n; i++) {
        fact *= i;
    }
    return fact;
}
```

Q) Code for fibonacci through iteration and recursion

~~static void~~ Iteration:

```
int fibo (int n) {
    int num1=0, num2=1;

    for (int i=0; i<n; i++) {
        cout << n << " ";
        int num3 = num2 + num1;
        num1 = num2;
        num2 = num3;
    }
}
```

Recursion:

```
int fib (int n) {
    if (n<=1) return 1;

    return fib(n-1) + fib(n-2);
}
```

Q) Explain process of reversing a single linked list and insertion operation in a single linked list.

⇒ To reverse a singly linked list, we change the direction of each node's pointer so that it points to its previous node instead of the next one.
eg: 10→20→30→NULL, After : 30→20→10→NULL

```
⇒ Struct Node {
      int data;
      Node* next;
  }

  Void insert (Node*& head, int val) {

      Node* newNode = new Node {val, head};
      head = newNode;
  }
  void reverse (Node*& head) {

      Node *prev=NULL, *curr=head, *next=NULL;

      while (curr!=NULL) {
          next = curr->next;
          curr->next = prev;
          prev = curr;
          curr = next;
      }
      head = prev;
  }
```

```
  void display (Node* head) {

      while (head != NULL) {

          cout << head->dat << " ";
          head = head->next;
      }
  }
  int main () {
      Node* head = NULL;
      insert (head, 30);
      insert (head, 20);
      insert (head, 10);

      cout << "original list:";
      display (head);

      reverse (head);

      cout << "In Reversed list:";
      display (head);
  }
```

* Insertion can happen at beginning, end or any specific position

eg: Initial list: 10→20→30

insert is after 10 => 10→15→20→30.

```
Struct Node {
    Node* next;
    int data;
};

void insertion (Node*& head, int val) {

    Node* newNode = new Node {val, NULL};

    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
        temp->next = newNode;
}
void display (Node* head) {
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}
int main() {
    Node* head = NULL;
    insertion (head, 10);
    insertion (head, 20);
    insertion (head, 30);
    cout << "List after insertion: ";
    display (head);
}
```

(2) Insertion and deletion in a doubly linked list.

=> Each node in a doubly linked list has

{ [prev | data | next] }

eg: before insertion:

→ 10 ⇄ 20 ⇄ 30

insert 40

→ 10 ⇄ 20 ⇄ 30 ⇄ 40

Insertion Code: {included in Deletion}

```
Struct Node {
    int data;
    Node* prev;
    Node* next;
};
void insertend (Node*& head, int val) {

    Node* newNode = new Node {val, NULL, NULL};
    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
```

```
void display(Node* head) {
    Node* temp = head;
    while(temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
}
int main() {
    Node* head = NULL;

    insertEnd(head, 10);
    "       (", 20);
    "       (", 30);
    "       (", 40);

    cout << "After insertion: ";
    display(head);
}
```

See this:

* **Deletion:**

To delete a node with a given value:

1) Traverse the list to find the node.
2) Adjust the prev and next pointers of adjacent nodes.
3) Delete the node.

eg: 10 ⇄ 20 ⇄ 30 ⇄ 40

after deletion:

10 ⇄ 30 ⇄ 40

```
struct Node {
    int data;
    Node* prev;
    Node* next;
};

Insertion
void insert(Node*& head, int val) {
    Node* newNode = new Node{val, NULL, NULL};
    if(head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while(temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
void delete(Node*& head, int val) {
    Node* temp = head;
    find node to delete { while(temp && temp->data != val) {
        temp = temp->next;
    }
    if(!temp) return; //not found.
```

```
//If deleting head node
if (temp -> prev == NULL) {
    head = temp -> next;
} else {
    temp -> prev -> next = temp -> next;
}
// If deleting last node
if (temp -> next != NULL) {
    temp -> next -> prev = temp -> prev;
}
delete temp;
}
void display (Node* head) {
    while (head != NULL) {
        cout << head -> data << " ";
        head = head -> next;
    }
}
int main() {
    node* head = NULL;
    insert (head, 10);
    " (head, 20);
    " (head, 30);
    " (head, 40);

    cout << "Before deletion:";
    display (head);
    delete (head, 20);
    cout << "After:"; display (head);
```

Q) Explain circular linked list and write the code to traverse it.

→ In CLL, the last node points back to the first node instead of NULL.

eg: 10 → 20 → 30 → 10

Basic Structure

Code for traversal:

```
Struct Node {
    int data;
    Node* next;
};

Void insert (Node*& head, int val) {
    Node* newNode = new Node{val, NULL};
    if (head == NULL) {
        head = newNode;
        newNode -> next = head;
        return;
    }
    Node* temp = head;
    while (temp -> next != head) {
        temp = temp -> next;
    }
    temp -> next = newNode;
    newNode -> next = head;
}
```

```
void traverse (Node* head) {
    if (head == NULL) return;
    Node* temp = head;
    do {
        cout << temp -> data << " ";
        temp = temp -> next;
    } while (temp != head);
}
int main() {
    Node* head = NULL;
    insert (head, 10);
    " (head, 20);
    " (" , 30);
    cout << "CLL";
    traverse (head);
}
```

APPLICATIONS :

- Implementing stacks and queues using LL.
- Multi player board games.
- Playing music tracks in a loop

- Representing ~~DAG~~ graps using LL.
- Using LL to handle collisions in hash tables.

* Algorithm for deletion in CLL :

1) Start
2) If list empty → return
3) If head → data == key and head → next == head → delete head, return NULL
4) Otherwise traverse :
   - keep track of current and prev node
   - If current → data == key :
     prev → next = curr → next
     If curr == head → move head = current → next
     Delete curr
     break

- Basic structure ; [check before page]
- void deleteN(Node* &head, int key) {

```
    if (!head) return;
    Node* curr = head, *prev = NULL;
```

```
while (curr->data != key) {
    if (curr->next==head) return;
    prev=curr;
    curr=curr->next;
}
if (curr==head && curr->next==head) {
    delete curr;
    head=NULL;
    return;
}
if (curr==head) {
    prev=head;
    while (prev->next!=head) {
        prev=prev->next;
    }
    head=curr->next;
    prev->next=head;
} else {
    prev->next=curr->next;
}
delete curr;
}
void traverse(Node* head) {
    ...
}
int main() {
    Node* head=NULL;
    insert(head,10); insert(head,20); ....; insert(head,
```

```
cout<<"Befor deletion:";
traverse(head);

deleteN(head,30);

cout<<"After:";
traverse(head);
```

Q) Write a recursive code for Tower of Hanoi

```
=> int TOH(int n, char source, char dest, char aux)
{
    if (n==0) {
        return;
    }

    TOH(n-1, source, aux, dest);

    cout<<"Move disk "<<n<<" from "<<
        source<<" to "<<dest;

    TOH(n-1, aux, dest, source);
}
```

8) Compare trade-offs b/w iteration and recursion.

=> They include:

1) Speed: Iteration is faster than recursion

2) Memory: Recursion requires more memory than iteration

3) Code Complexity:

Recursion has simpler, readable code, while iteration results in complex code

4) Time Comp: Recursion has higher T.C than iteration.

5) Approach: Recursion follows divide and conquer approach while iteration follows sequential execution approach.

6) Suitability: Recursion is better for tasks that can be described naturally in a recursive way, while iteration is better for loops.

7) Optimization: It is difficult to optimize recursive code compared to iterative.

* Trade-offs with reference to memory usage:

- Recursion:

=> each call occupies a new frame in the call stack.

=> Deep recursion leads to stack overflow if recursion depth is large

- Iteration:

=> Uses fixed memory regardless of the no. of loop iterations.

=> More memory efficient for large datasets.

* With reference to stack usage

- Recursion:

=> Each function call adds a new activation record (parameters, return address, local variables) to the call stack.

eg: TOH, for n=3, 7 recursive calls uses 3 frames

- Iteration: only one frame is used, loop variables are updated in place

eg: Iterative factorial uses a single frame regardless of n.

* Pros and Cons of recursion:

→ Pros

① Simplicity

② Readability

③ Divide and Conquer

Cons

① High memory usage

② Performance ~~overhead~~ overhead

③ Risk of stack overflow

④ Debugging Complexity.

(1) ** WAP on merge sort, Explain its TC, Explain its divide and conquer approach and benefits over bubble sort and significance in large dataset sorting.

⇒ Program → Check Notebook

T.C → Check Notebook.

* Benefits over bubble sort.

→

	Merge Sort	Bubble sort
T.C	O(nlogn)	O(n²)
Stability	Stable	Stable
Efficiency	Very efficient	very slow
Divide and Conquer	Yes	no
Parallelizable/ Recursive	Easy to parallelize	hard

* Divide and Conquer approach

1) Divide:

The array is recursively divided into two halves until each subarray contains a single element.

2) Conquer: Each subarray is trivially sorted

3) Combine:

The sorted subarrays are merged together in sorted order using merge procedure.

* Significance in large dataset sorting

- Predictable performance
- Stable sort
- Efficient for large datasets
- Suitable for External sorting
- Parallelization.

6) Applications of Quick Sort and comperison with Merge Sort.

=> Quick sort is a sorting algorithm based on divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in a sorted array.

* Role in applications

- System-level Sorting
- Databases and Query optimization
- Search engines
- Embedded Systems
- Games / Computer graphics.

* Comparison of merge sort and quicksort:

Aspect	Quick	Merge
Approach	Divide and Conquer using partitioning	Same but using merging
T.C	O(nlogn)	O(nlogn)
S.C	O(logn)	O(n)
Stability	Poor Stability	Stable
Performance on large data sets	faster	Better for and external sorting
Parallelization	Harder to parallelize	easier to parallelize

4) Polynomial representation and addition of Singly linked list.

```
Struct Node {
    int coeff, power;
    Node* next;
    Node(int c, int p) {
        coeff = c;
        power = p;
        next = Null;
    }
};

void insert(Node *&head, int c, int p) {
    Node* newNode = newNode(c,p);
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while(temp -> next) {
        temp = temp->next;
    }
    temp->next = newNode.
}
```

```
Node* addPoly (Node* p1, Node* p2) {
    Node* result = NULL;
    while (p1 && p2) {
        if(p1->power == p2->power) {
            insert(result, p1->coeff + p2->coeff,
                   p1->power);
            p1 = p1->next; p2 = p2->next;
        } else if (p1->power > p2->power) {
            insert(result, p1->coeff, p1->power);
            p1 = p->next;
        } else {
            insert (result, p2->coeff, p2->power);
            p2 = p2->next;
        }
    }
    while (p1) {insert (result, p1->coeff, p1->power);
                p1 = p1->next; }
    while (p2) {insert(result, p2->coeff, p2->power);
                p2 = p2->next; }
}
```

```
void display (Node* head) {
    while (head) {
        cout << head->coeff << "x^" << head->power;
        if (head->next) {
            cout << "+";
        }
        head = head->next;
    }
    cout << endl;
}

int main () {
    Node *p1=NULL, *p2=NULL;

    insert (p1, 5, 3);
    insert (p1, 4, 2);
    insert (p1, 2, 1);
    insert (p1, 1, 0);

    insert (p2, 3, 2);
    insert (p2, 1, 1);
    insert (p2, 9, 0);

    cout << "p1: ";
    display (p1);

    cout << "p2: ";
    display (p2);

    Node *sum = addPoly (p1, p2);
    cout << "sum: ";
    display (sum);
}
```

Application

A singly linked list is efficient to represent polynomial terms in decreasing order of exponents.

- uses less memory than doubly LL.

eg: $6x^4 + 3x^2 + 7$

Coeff	Power	Next
6	4	→
3	2	→
7	0	NULL

8) Polynomial representation of Doubly linked list. Discuss use of DLL in Complex Data management.

* Allows bidirectional Traversal
* Easier Insertion/deletion on both ends

```
=> Struct Node {
    int Coeff, power
    Node *prev, *next;
    Node (int c, int p) {
        Coeff = c;
        power = p;
        prev = next = NULL;
    }
};

void insertTe(Node *&head, int c, int p) {
    Node* NewNode = new Node(c,p);
    if(!head || head->power < p) {
        newNode->next = head;
        if(head) {
            head->prev = newNode;
        }
        head = newNode;
        return;
    }
    Node* temp = head;
    Node* temp = head;
    while( temp->next && temp->next->power>p
        temp = temp->next;
    }
    newNode->next = temp->next;
    if (temp->next) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
void deleteTe(Node*& head, int p) {
    Node* temp = head;
    while (temp && temp->power != p) {
        temp = temp->next;
    }
    if(!temp) { return; }
    if(temp->prev) {
        temp->prev->next = temp->next;
    } else {
        head = temp->next;
    }
    if (temp->next) {
        temp->next->prev = temp->prev;
    }
    delete temp;
}
```

```
void display (Node* Head) {
    while (head) {
        cout << head->coeff << "x^" << head->pow;
        if (head->next) {
            cout << " + ";
        }
        head = head->next;
    }
    cout << endl;
}
int main () {
    Node* poly = NULL;
    insertTe (poly, 5, 3);
    "  (poly, 2, 1);
    "  (poly, 4, 2);
    cout << "Polynomial: "; display (poly);
    deleteTe (poly, 2);
    cout << "After deleting x^2 term: ";
    display (poly);
}
```

Use of DLL in Complex Data Management

- Undo/redo functionality in editors
- Navigation systems
- Browser history management
- Playlist or multimedia management.