RABIN–KARP ALGORITHM

A. Short Questions (20) with Answers

1. **Q:** What is the main idea of Rabin–Karp algorithm?
   A: It uses hashing to match patterns in a text efficiently.
2. **Q:** What type of hashing does Rabin–Karp use?
   A: Rolling hash.
3. **Q:** Why is rolling hash used?
   A: To calculate the hash of the next substring in O(1) time.
4. **Q:** What is a hash collision?
   A: When two different strings have the same hash value.
5. **Q:** How does Rabin–Karp handle hash collisions?
   A: By comparing the actual strings when hashes match.
6. **Q:** What is the worst-case time complexity?
   A: O(n * m), where n = text length, m = pattern length.
7. **Q:** What is the average case time complexity?
   A: O(n + m).
8. **Q:** Can Rabin–Karp match multiple patterns at once?
   A: Yes, by storing multiple pattern hashes.
9. **Q:** What is the advantage of Rabin–Karp over naive matching?
   A: It uses hash to skip unnecessary comparisons.
10. **Q:** Which arithmetic is used in hashing?
    A: Modular arithmetic.
11. **Q:** How is the hash of a string calculated?
    A: Usually as hash = (s[0]*p^(m-1) + s[1]*p^(m-2) + ... + s[m-1]) mod q.
12. **Q:** What are common choices for p and q in hash function?
    A: p is a small prime (like 31), q is a large prime.
13. **Q:** Why do we use a prime number in modulo?
    A: To reduce collisions in hashing.
14. **Q:** How does Rabin–Karp differ from KMP?
    A: KMP uses prefix function, Rabin–Karp uses hashing.

15. **Q:** What happens if hash values don't match?
    A: The substring does not match; skip comparison.
16. **Q:** Can Rabin–Karp be used for plagiarism detection?
    A: Yes, it efficiently finds repeated sequences.
17. **Q:** How is the rolling hash updated?
    A: hash_new = (hash_old - s[i]*p^(m-1))*p + s[i+m] (mod q)
18. **Q:** What is the main limitation of Rabin–Karp?
    A: Hash collisions can increase runtime in worst case.
19. **Q:** Is Rabin–Karp suitable for long texts?
    A: Yes, because average time is linear.
20. **Q:** What is a practical use of Rabin–Karp besides string matching?
    A: Detecting duplicate substrings, plagiarism, DNA sequence matching.

B. MCQs (20)

1. Rabin–Karp algorithm is mainly based on:
   a) Brute force
   b) Hashing ✅
   c) Dynamic programming
   d) Greedy
2. Rolling hash allows computation in:
   a) $O(n^2)$
   b) O(1) ✅
   c) O(log n)
   d) O(n log n)
3. Hash collisions are resolved by:
   a) Ignoring
   b) Comparing actual strings ✅
   c) Using linked list
   d) Dividing by n
4. Average time complexity of Rabin–Karp is:
   a) O(n*m)
   b) O(n + m) ✅

c) $O(n^2)$
d) O(log n)

5. Worst-case time complexity occurs when:
   a) No matches
   b) Many hash collisions ✅
   c) Short pattern
   d) Long text

6. Prime number in hash function helps:
   a) Speed up
   b) Reduce collisions ✅
   c) Make pattern longer
   d) None

7. Typical choice of p in hashing is:
   a) 1
   b) 31 ✅
   c) 100
   d) 10

8. Typical choice of q in hashing is:
   a) 3
   b) 101
   c) Large prime ✅
   d) 0

9. Rabin–Karp is especially efficient for:
   a) Single pattern
   b) Multiple patterns ✅
   c) Sorting
   d) Searching numbers

10. Hash formula uses:
    a) Addition only
    b) Powers of a base ✅
    c) Subtraction only
    d) Division only

11. Rolling hash avoids:
    a) Recalculating full substring hash ✅

b) Sorting strings
c) Comparing first character
d) Using stack

12. Hash collision is:
    a) When hashes match ✅
    b) Strings mismatch
    c) Hash is zero
    d) String is empty

13. Rabin–Karp is faster than naive when:
    a) Patterns are long ✅
    b) Patterns are short
    c) Text is small
    d) Text is sorted

14. If hash values match, algorithm:
    a) Returns match ✅
    b) Compares strings
    c) Stops
    d) Ignores

15. Rabin–Karp can be used in:
    a) DNA sequence matching ✅
    b) Matrix multiplication
    c) BFS
    d) Sorting

16. Hash modulo helps to:
    a) Reduce integer size ✅
    b) Increase complexity
    c) Reverse string
    d) Compare digits

17. A rolling hash uses which operations?
    a) Only addition
    b) Addition, multiplication, modulo ✅
    c) Only subtraction
    d) None

18. Hashing makes Rabin–Karp:
    a) Linear on average ✅
    b) Quadratic always
    c) Exponential
    d) Constant

19. What happens if prime q is too small?
    a) Faster
    b) More collisions ✅
    c) No change
    d) Algorithm fails

20. Rabin–Karp uses:
    a) Queue
    b) Stack
    c) Hash function ✅
    d) Linked list

C. True/False (20)

1. Rabin–Karp uses hashing to compare strings. ✅
2. Rolling hash recomputes full hash each time. ❌
3. Hash collisions never occur. ❌
4. Average time complexity is O(n + m). ✅
5. Worst-case time is O(n*m). ✅
6. Rabin–Karp can match multiple patterns. ✅
7. It cannot handle multiple patterns. ❌
8. Prime numbers reduce collisions. ✅
9. Rabin–Karp is slower than naive always. ❌
10. Hash is updated in O(1) using rolling hash. ✅
11. Rabin–Karp uses dynamic programming. ❌
12. Hash collisions are resolved by string comparison. ✅

13. Rolling hash improves speed. ✅

14. Hash modulo increases collisions. ❌

15. Rabin–Karp can detect plagiarism. ✅

16. Hash function may cause false positives. ✅

17. Rabin–Karp cannot be used for DNA sequences. ❌

18. Hash base p is usually prime. ✅

19. Rabin–Karp always gives correct matches if hash matches. ❌ (due to collisions)

20. Rabin–Karp is useful for long texts. ✅

FLOYD–WARSHALL ALGORITHM

A. Short Questions (20) with Answers

1. **Q:** What problem does Floyd–Warshall algorithm solve?
   A: It finds the shortest paths between all pairs of vertices in a weighted graph.
2. **Q:** Can it handle negative edge weights?
   A: Yes, as long as there are no negative cycles.
3. **Q:** Can it detect negative cycles?
   A: Yes, by checking if the distance from a node to itself becomes negative.
4. **Q:** What is the main approach used?
   A: Dynamic programming.
5. **Q:** What is the time complexity?
   A: $O(n^3)$, where n is the number of vertices.
6. **Q:** How is the algorithm initialized?
   A: Distance matrix dist[i][j] is initialized to edge weights or infinity if no edge.
7. **Q:** What is the recurrence relation?
   A: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]) for all k.
8. **Q:** Does it work on directed graphs?
   A: Yes.
9. **Q:** Does it work on undirected graphs?
   A: Yes, treat each edge as bidirectional.
10. **Q:** What is the difference between Floyd–Warshall and Dijkstra?
    A: Dijkstra finds single-source shortest path; Floyd–Warshall finds all-pairs shortest paths.
11. **Q:** What data structure is mainly used?
    A: 2D array (matrix) for distances.
12. **Q:** Can it handle weighted graphs with zero weights?
    A: Yes.
13. **Q:** How do we update the distance matrix?
    A: Iteratively consider each vertex as an intermediate node.

14. **Q:** Why is Floyd–Warshall called dynamic programming?
    A: Because it builds solutions of smaller subproblems (using intermediate vertices).

15. **Q:** How can we reconstruct the shortest path?
    A: By maintaining a next matrix to store intermediate nodes.

16. **Q:** Can it be used for unweighted graphs?
    A: Yes, but BFS is more efficient.

17. **Q:** What happens if there is a negative cycle?
    A: Distances can become negative infinity; shortest path is undefined.

18. **Q:** Is it suitable for sparse graphs?
    A: Not very; $O(n^3)$ may be costly for large sparse graphs.

19. **Q:** Does it require the graph to be connected?
    A: No, infinite distance represents disconnected nodes.

20. **Q:** Give one practical application.
    A: Network routing, all-pairs shortest path analysis.

B. MCQs (20)

1. Floyd–Warshall finds:
   a) Single-source shortest path
   b) All-pairs shortest path ✅
   c) Minimum spanning tree
   d) Maximum flow

2. Time complexity of Floyd–Warshall is:
   a) $O(n^2)$
   b) $O(n^3)$ ✅
   c) $O(n \log n)$
   d) $O(\log n)$

3. Floyd–Warshall uses which approach?
   a) Greedy
   b) Dynamic programming ✅
   c) Divide and conquer
   d) Backtracking

4. Can Floyd–Warshall handle negative weights?
   a) No
   b) Yes ✅
   c) Only zero
   d) Only positive

5. Can it detect negative cycles?
   a) No
   b) Yes ✅
   c) Only in directed graphs
   d) Only undirected graphs

6. Initial distance from i to j if no edge:
   a) 0
   b) Infinity ✅
   c) -1
   d) 1

7. Distance from i to i initially:
   a) 0 ✅
   b) Infinity
   c) Edge weight
   d) -1

8. Recurrence relation:
   a) dist[i][j] = dist[i][k] + dist[k][j]
   b) dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]) ✅
   c) dist[i][j] = max(dist[i][j], dist[i][k] + dist[k][j])
   d) dist[i][j] = dist[i][j] - dist[i][k]

9. Suitable graph type:
   a) Weighted ✅
   b) Only unweighted
   c) Only directed
   d) Only undirected

10. Distance updated in:
    a) O(1) per edge
    b) $O(n^3)$ ✅

c) O(log n)
d) O(n^2)

11. Can handle disconnected graphs?
    a) Yes ✅
    b) No
    c) Only positive weights
    d) Only connected

12. Intermediate vertex in update:
    a) i
    b) j
    c) k ✅
    d) None

13. Floyd–Warshall is inefficient for:
    a) Dense graphs
    b) Sparse graphs ✅
    c) Small graphs
    d) All graphs

14. How to reconstruct path?
    a) Use distance matrix only
    b) Use next matrix ✅
    c) Use hash table
    d) Use queue

15. Which of these is true?
    a) Works only on weighted graphs
    b) Works on both weighted and unweighted ✅
    c) Only on positive weights
    d) Only on directed graphs

16. Floyd–Warshall is better than Dijkstra for:
    a) Single-source
    b) All-pairs ✅
    c) Sparse graphs
    d) None

17. If negative cycle exists, distance from node to itself:
    a) Zero

b) Infinity
c) Negative ✅
d) Undefined

18. Space complexity:
    a) O(n)
    b) $O(n^2)$ ✅
    c) $O(n^3)$
    d) O(log n)

19. Suitable for graph size n ~ 1000?
    a) Yes
    b) No ✅
    c) Only if sparse
    d) Only if unweighted

20. Floyd–Warshall uses which programming paradigm?
    a) Greedy
    b) Dynamic programming ✅
    c) Divide and conquer
    d) Backtracking

C. True/False (20)

1. Floyd–Warshall finds all-pairs shortest paths. ✅
2. It cannot handle negative weights. ❌
3. Time complexity is $O(n^3)$. ✅
4. Uses dynamic programming. ✅
5. Distance from i to i is always initialized as 1. ❌
6. Can detect negative cycles. ✅
7. Works only for connected graphs. ❌
8. Space complexity is $O(n^2)$. ✅
9. It is suitable for sparse graphs with $n = 10^5$. ❌

10. Recurrence: dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]). ✅

11. Only works on directed graphs. ❌

12. Updates distance using each vertex as intermediate. ✅

13. Can reconstruct path using a next matrix. ✅

14. If negative cycle exists, distances are invalid. ✅

15. Suitable for small dense graphs. ✅

16. Can handle zero-weight edges. ✅

17. Requires adjacency list. ❌

18. Can solve single-source shortest path efficiently. ❌

19. Works for both weighted and unweighted graphs. ✅

20. Distance between disconnected nodes remains infinity. ✅

GRAHAM SCAN (CONVEX HULL)

A. Short Questions (20) with Answers

1. **Q:** What is the main purpose of Graham Scan?
   A: To find the convex hull of a set of points in 2D space.
2. **Q:** What is a convex hull?
   A: The smallest convex polygon that contains all given points.
3. **Q:** What is the first step in Graham Scan?
   A: Find the point with the lowest y-coordinate (pivot).
4. **Q:** If multiple points have the same y-coordinate, which one is chosen as pivot?
   A: The one with the smallest x-coordinate.
5. **Q:** What is the next step after selecting pivot?
   A: Sort all other points based on polar angle with the pivot.
6. **Q:** How do you determine the orientation of three points?
   A: Using cross product: (B-A) × (C-B). Positive = counterclockwise, Negative = clockwise.
7. **Q:** Which data structure is used to store the convex hull?
   A: Stack.
8. **Q:** How are points added to the convex hull?
   A: Push points onto the stack; pop if a clockwise turn occurs.
9. **Q:** What is the time complexity of sorting points by angle?
   A: O(n log n).
10. **Q:** What is the total time complexity of Graham Scan?
    A: O(n log n) (sorting dominates).
11. **Q:** Does Graham Scan work for collinear points?
    A: Yes, it can include or exclude them depending on implementation.
12. **Q:** Can Graham Scan be used for 3D points?
    A: No, it is for 2D only; 3D requires different algorithms.
13. **Q:** What is the role of the pivot point?
    A: It acts as the reference for sorting and angle calculations.
14. **Q:** What happens if points form a perfect square?
    A: Graham Scan will return all 4 points as convex hull.

15. **Q:** What is a clockwise turn in orientation test?
    A: Cross product < 0.

16. **Q:** What is a counterclockwise turn?
    A: Cross product > 0.

17. **Q:** Can Graham Scan handle duplicate points?
    A: Yes, but duplicates are usually removed before processing.

18. **Q:** Is Graham Scan a greedy algorithm?
    A: No, it is based on sorting + stack (not greedy).

19. **Q:** Why is the stack necessary?
    A: To maintain the current convex hull and remove points making clockwise turns.

20. **Q:** Give one practical application of Graham Scan.
    A: Collision detection, shape analysis, pattern recognition.

B. MCQs (20)

1. Graham Scan finds:
   a) Shortest path
   b) Convex hull ✅
   c) Minimum spanning tree
   d) Maximum flow

2. First step in Graham Scan is:
   a) Sort points
   b) Find pivot ✅
   c) Build stack
   d) Calculate distance

3. Pivot is chosen as:
   a) Highest y-coordinate
   b) Lowest y-coordinate ✅
   c) Random point
   d) Centroid

4. Sorting points is done by:
   a) Distance from origin
   b) Polar angle with pivot ✅

c) x-coordinate only
d) y-coordinate only

5. Orientation is determined using:
   a) Dot product
   b) Cross product ✅
   c) Distance formula
   d) Angle formula

6. Counterclockwise turn occurs when:
   a) Cross product < 0
   b) Cross product > 0 ✅
   c) Cross product = 0
   d) None

7. Clockwise turn occurs when:
   a) Cross product < 0 ✅
   b) Cross product > 0
   c) Cross product = 0
   d) None

8. Stack is used to:
   a) Store all points
   b) Store current convex hull ✅
   c) Sort points
   d) Compute distances

9. Time complexity of sorting points:
   a) $O(n^2)$
   b) O(n log n) ✅
   c) $O(n^3)$
   d) O(n)

10. Total time complexity of Graham Scan:
    a) $O(n^2)$
    b) O(n log n) ✅
    c) $O(n^3)$
    d) O(n)

11. Graham Scan works for:
    a) 2D points ✅

b) 3D points
c) Only convex polygons
d) Only collinear points

12. Duplicate points:
    a) Must be removed ✅
    b) Are always included
    c) Cause failure
    d) Are ignored

13. Convex hull includes:
    a) Only extreme points ✅
    b) All points
    c) Only pivot
    d) None

14. Algorithm handles collinear points by:
    a) Ignoring them
    b) Including or excluding based on implementation ✅
    c) Crashing
    d) Sorting only

15. Pivot point is always included in convex hull:
    a) Yes ✅
    b) No
    c) Sometimes
    d) Never

16. Stack pop occurs when:
    a) Counterclockwise turn
    b) Clockwise turn ✅
    c) Collinear points
    d) Pivot reached

17. Cross product formula:
    a) (B-A) × (C-B) ✅
    b) (A-B) + (B-C)
    c) |A-B| × |B-C|
    d) None

18. Graham Scan is suitable for:
    a) Collision detection ✅
    b) Shortest path
    c) BFS traversal
    d) Sorting

19. Algorithm sorts points in:
    a) Increasing polar angle ✅
    b) Decreasing polar angle
    c) Random order
    d) By distance from origin

20. Graham Scan is:
    a) Brute force
    b) Sorting + stack ✅
    c) Dynamic programming
    d) Divide and conquer

C. True/False (20)

1. Graham Scan finds convex hull of 2D points. ✅
2. Pivot is chosen as highest y-coordinate. ❌
3. Points are sorted by polar angle. ✅
4. Cross product determines orientation. ✅
5. Stack is used to maintain convex hull. ✅
6. Clockwise turn points are kept. ❌
7. Counterclockwise turn points are kept. ✅
8. Time complexity is $O(n^2)$. ❌
9. Total time complexity is $O(n \log n)$. ✅
10. Graham Scan works for 3D points. ❌
11. Duplicate points must be handled. ✅
12. Collinear points may or may not be included. ✅

13. Pivot is always part of convex hull. ✅

14. Algorithm is greedy. ❌

15. Sorting dominates the time complexity. ✅

16. Graham Scan is suitable for collision detection. ✅

17. Orientation cross product < 0 = counterclockwise. ❌

18. Orientation cross product > 0 = counterclockwise. ✅

19. Convex hull contains all points. ❌

20. Stack pop occurs for clockwise turns. ✅

N-QUEEN PROBLEM USING BACKTRACKING

A. Short Questions (20) with Answers

1. **Q:** What is the N-Queen problem?
   A: Place N queens on an N×N chessboard so that no two queens attack each other.
2. **Q:** What are the attacking rules of a queen?
   A: Can attack in the same row, same column, or diagonals.
3. **Q:** What is the main technique used to solve N-Queen?
   A: Backtracking.
4. **Q:** How do you check if a queen is safe?
   A: Ensure no other queen is in the same row, column, or diagonal.
5. **Q:** Can N-Queen be solved for N < 4?
   A: No, except for N = 1.
6. **Q:** What is the time complexity of the basic backtracking solution?
   A: O(N!).
7. **Q:** Can N-Queen have multiple solutions?
   A: Yes, there can be many arrangements.
8. **Q:** What data structure is typically used?
   A: 2D array (board) or 1D array (column positions).
9. **Q:** What is pruning in backtracking?
   A: Skipping invalid placements early to reduce computation.
10. **Q:** What is the first row placement strategy?
    A: Place a queen in the first row and try all columns.
11. **Q:** How is recursion used in N-Queen?
    A: Recursively place queens row by row and backtrack if invalid.
12. **Q:** Can N-Queen be solved iteratively?
    A: Yes, but recursive backtracking is simpler.
13. **Q:** How are diagonals represented in 1D array solution?
    A: Two arrays: one for left diagonals, one for right diagonals.
14. **Q:** How do you count all solutions?
    A: Use a counter incremented every time a valid arrangement is found.

15. **Q:** Can N-Queen be solved for N = 8?
    A: Yes, standard 8-Queen problem.
16. **Q:** What is the minimum N for a valid solution besides N=1?
    A: N = 4.
17. **Q:** How do you print one solution?
    A: Print the board array or column positions.
18. **Q:** Is N-Queen suitable for teaching backtracking?
    A: Yes, it's a classic example.
19. **Q:** Can optimization techniques reduce complexity?
    A: Yes, using bitmasking or pruning diagonals.
20. **Q:** Give a practical application of N-Queen.
    A: Constraint satisfaction problems, AI, scheduling.

B. MCQs (20)

1. N-Queen problem aims to:
   a) Place N queens on N×N board ✅
   b) Count queens only
   c) Minimize moves
   d) Solve Sudoku
2. A queen attacks in:
   a) Row only
   b) Column only
   c) Diagonal only
   d) Row, column, diagonal ✅
3. Minimum N for a solution besides N=1:
   a) 2
   b) 3
   c) 4 ✅
   d) 5
4. Backtracking is used because:
   a) Explores all possibilities ✅
   b) Always linear time

c) Uses hashing
d) Greedy

5. Time complexity is approximately:
   a) $O(N^2)$
   b) O(N!) ✅
   c) O(N log N)
   d) O(2^N)

6. Data structure used:
   a) Queue
   b) Stack
   c) 2D array or 1D array ✅
   d) Graph

7. How many solutions exist for 4-Queen?
   a) 1
   b) 2 ✅
   c) 4
   d) 8

8. Recursion is used to:
   a) Place queens row by row ✅
   b) Count diagonals
   c) Sort queens
   d) None

9. Pruning helps to:
   a) Skip invalid placements ✅
   b) Print solution
   c) Increase complexity
   d) Move queen

10. N-Queen is a:
    a) Greedy problem
    b) Backtracking problem ✅
    c) Sorting problem
    d) Dynamic programming problem

11. Can N-Queen be solved for N = 2?
    a) Yes

b) No ✅
c) Only row-wise
d) Only column-wise

12. Can N-Queen be solved for N = 8?
    a) Yes ✅
    b) No
    c) Only partially
    d) Only iteratively

13. Which is not a constraint for N-Queen?
    a) Row uniqueness
    b) Column uniqueness
    c) Diagonal uniqueness
    d) Knight move uniqueness ✅

14. Solution can be stored as:
    a) Binary tree
    b) Column positions array ✅
    c) Linked list
    d) Hash map

15. How are diagonals checked efficiently?
    a) Use nested loops
    b) Use two arrays for diagonals ✅
    c) Ignore
    d) Use queue

16. Counter is used to:
    a) Count attacks
    b) Count valid solutions ✅
    c) Store positions
    d) Sort queens

17. Bitmasking can reduce time complexity:
    a) Yes ✅
    b) No
    c) Only for N < 4
    d) Only for N = 1

18. Backtracking returns:
    a) One solution
    b) All solutions ✅
    c) Maximum moves
    d) None

19. N-Queen is a type of:
    a) Sorting problem
    b) Constraint satisfaction problem ✅
    c) Graph problem only
    d) Greedy problem

20. Row-wise placement strategy:
    a) Place queens column by column
    b) Place queens row by row ✅
    c) Place queens diagonally
    d) Randomly

C. True/False (20)

1. N-Queen problem is solved using backtracking. ✅
2. Queens can attack in row, column, diagonal. ✅
3. N = 3 has a solution. ❌
4. N = 4 has exactly 2 solutions. ✅
5. Time complexity is O(N!). ✅
6. Board is typically stored as 2D array. ✅
7. Recursion is not used in N-Queen. ❌
8. Pruning helps reduce computation. ✅
9. Diagonals must be checked for conflicts. ✅
10. Bitmasking cannot optimize N-Queen. ❌
11. N-Queen is an example of constraint satisfaction. ✅
12. Solutions can be counted using a counter. ✅

13. Column uniqueness is required. ✅

14. Row uniqueness is required. ✅

15. N-Queen can be solved iteratively easily. ✅

16. Only one solution exists for N=8. ❌

17. Backtracking explores all possible arrangements. ✅

18. N-Queen has no practical applications. ❌

19. Diagonal conflicts can be tracked using arrays. ✅

20. Minimum N for non-trivial solution is 1. ✅

Modular Exponentiation

A. Short Questions (20) with Answers

1. **Q:** What is modular exponentiation?
   A: Computing $a^b \bmod m$ efficiently, especially for large b.
2. **Q:** Why not compute a^b directly?
   A: a^b can become extremely large and cause overflow.
3. **Q:** What is the basic formula?
   A: $(a^b) \bmod m = ((a \bmod m)^b) \bmod m$.
4. **Q:** What technique is commonly used?
   A: Binary exponentiation (exponentiation by squaring).
5. **Q:** How does binary exponentiation work?
   A: Break the exponent b into powers of 2, square and multiply iteratively or recursively.
6. **Q:** What is the time complexity of binary exponentiation?
   A: O(log b).
7. **Q:** Can modular exponentiation handle negative exponents?
   A: Only if a modular inverse is used.
8. **Q:** How is multiplication handled under modulo?
   A: $(a * b) \bmod m = ((a \bmod m) * (b \bmod m)) \bmod m$.
9. **Q:** What is the base case for recursion?
   A: If $b = 0$, return 1.
10. **Q:** How is recursion applied?
    A: Split b into halves: compute $a^{b/2} \bmod m$ recursively.
11. **Q:** Can modular exponentiation be implemented iteratively?
    A: Yes, using a loop and squaring technique.
12. **Q:** Can it be used with prime modulus?
    A: Yes, often combined with Fermat's theorem for inverses.
13. **Q:** What is the advantage over naive computation?
    A: Avoids computing huge numbers and prevents overflow.
14. **Q:** How is the result updated in iterative binary exponentiation?
    A: Multiply by base when current bit of exponent is 1 and square the base each step.

15. **Q:** How to handle base greater than modulus?
    A: Reduce base modulo m first: $a \bmod m$.

16. **Q:** Can modular exponentiation be used in cryptography?
    A: Yes, e.g., RSA encryption and Diffie–Hellman key exchange.

17. **Q:** Is it suitable for very large exponents (e.g., 10^{18})?
    A: Yes, because it works in O(log b) time.

18. **Q:** What practical problems use modular exponentiation?
    A: Cryptography, combinatorics, modular power calculations.

19. **Q:** Can modular exponentiation be combined with modular inverse?
    A: Yes, for handling negative exponents.

20. **Q:** What happens if modulo is 1?
    A: The result is always 0, regardless of base or exponent.

B. MCQs (20)

1. Modular exponentiation computes:
   a) $a + b$
   b) $a^b \bmod m$ ✅
   c) $a * b$
   d) $a - b$

2. Direct computation of a^b is:
   a) Efficient
   b) Can overflow ✅
   c) Faster
   d) Impossible

3. Binary exponentiation has time complexity:
   a) O(b)
   b) O(log b) ✅
   c) O(1)
   d) O(n)

4. Base reduction in modulo:
   a) a mod m ✅
   b) b mod m

c) a*b mod m
d) None

5. Recursive base case:
   a) b=1
   b) b=0 ✅
   c) a=0
   d) a=1

6. If b is even in recursion:
   a) Multiply by a
   b) Square a^(b/2) ✅
   c) Add a
   d) Divide a

7. If b is odd in recursion:
   a) Multiply by a ✅
   b) Divide a
   c) Subtract a
   d) Do nothing

8. Can it handle negative exponents?
   a) Yes, with modular inverse ✅
   b) No
   c) Only b=1
   d) Only b=0

9. Modular multiplication formula:
   a) a*b*
   b) ((a mod m)(b mod m)) mod m ✅
   c) a+b
   d) None

10. Iterative method uses:
    a) Loops + squaring ✅
    b) Recursion only
    c) Greedy
    d) Sorting

11. Can it handle very large exponents?
    a) Yes ✅

b) No
c) Only small b
d) Only b=0

12. Common application:
    a) Cryptography ✅
    b) Sorting
    c) BFS
    d) DFS

13. Exponent is split by:
    a) 3
    b) 2 ✅
    c) 5
    d) 10

14. Modular exponentiation prevents:
    a) Overflow ✅
    b) Underflow
    c) Sorting errors
    d) None

15. Iterative method multiplies result when:
    a) Bit=0
    b) Bit=1 ✅
    c) Bit=-1
    d) Bit=2

16. Base greater than modulus:
    a) Reduce modulo ✅
    b) Ignore
    c) Double
    d) Add

17. Can be combined with Fermat's theorem?
    a) Yes ✅
    b) No
    c) Only for even base
    d) Only for odd exponent

18. Useful in:
    a) Combinatorics ✅
    b) BFS
    c) DFS
    d) Sorting

19. Advantage over naive:
    a) Faster ✅
    b) Slower
    c) Same
    d) Cannot compute

20. Modulo 1 result:
    a) 1
    b) 0 ✅
    c) Base value
    d) Exponent value

C. True/False (20)

1. Modular exponentiation computes $a^b \bmod m$. ✅
2. Direct computation of a^b is always safe. ❌
3. Binary exponentiation reduces time to O(log b). ✅
4. Recursive base case is b=0. ✅
5. Iterative method uses squaring. ✅
6. Modular exponentiation can handle very large b. ✅
7. Modular exponentiation cannot handle negative exponents. ❌
8. Modular inverse is used for negative exponents. ✅
9. Base greater than m should be reduced modulo m. ✅
10. Exponent is split into powers of 2. ✅
11. Used in cryptography. ✅
12. Prevents overflow. ✅

13. Result is always less than modulus. ✅

14. Iterative method multiplies when bit=1. ✅

15. Recursive method squares for even exponent. ✅

16. Modular exponentiation is inefficient for large exponents. ❌

17. Can be combined with Fermat's theorem. ✅

18. Modulo 1 always gives result 0. ✅

19. Useful in combinatorics. ✅

20. Iterative and recursive methods give same result. ✅