QUESTION 1 — 1D Array Statistics & Binary Search

Given an integer array, implement the following functions:

(a) void array_stats(int a[], int n, int *mn, int *mx, int *sum, double *mean)

Compute the minimum, maximum, sum, and mean of array a[]. Since doubles are forbidden, compute the integer mean as sum/n (integer division) and store a second value: the remainder (sum % n) so the caller can print 'mean = sum/n remainder r'.

(b) int binary_search(int a[], int n, int target)

Classic iterative binary search on a sorted array. Return the index of target or -1 if not found.

(c) void rotate_left(int a[], int n, int k)

Rotate array a[] left by k positions in-place. E.g. {1,2,3,4,5} rotated left by 2 becomes {3,4,5,1,2}. Use the reversal algorithm: reverse a[0..k-1], reverse a[k..n-1], reverse whole array.

(d) int kadane(int a[], int n)

Kadane's algorithm: find and return the maximum subarray sum (contiguous elements) in O(n).

QUESTION 2 — 2D Array Image Processing

An 8×8 greyscale image is stored as a 2D integer array with pixel values in [0, 255]. Implement:

(a) void convolve(int img[8][8], int kernel[3][3], int out[8][8])

Apply a 3×3 convolution kernel to the image. Use zero-padding for border pixels. Clamp output values to [0, 255]. Do NOT divide by kernel sum inside the loop — pass a separate int divisor parameter.

(b) void histogram(int img[8][8], int hist[256])

Compute the 256-bin greyscale histogram of the image.

(c) void threshold(int img[8][8], int out[8][8], int T)

Binary threshold: pixels >= T become 255, others become 0.

(d) void rotate_90_cw(int src[8][8], int dst[8][8])

Rotate the 8×8 image 90 degrees clockwise into dst. Formula: dst[c][7-r] = src[r][c].

(e) int connected_components(int bin[8][8])

Count the number of connected components of white pixels (value 255) in a binary image using flood-fill. Use an iterative approach with a manual queue (1D integer array of encoded row*8+col positions). Return the count.

QUESTION 3 — Text Processing with Integer Arrays

Represent text as arrays of int (each element is an ASCII code).

(a) int int_strlen(int s[], int max)

Return the length of the integer string (count until the terminator value 0).

(b) bool int_strcmp(int a[], int b[])

Return true if two integer strings are identical.

(c) int count_words(int s[], int n)

Count the number of words in the string. Words are separated by spaces (ASCII 32). Handle multiple consecutive spaces. A word is any maximal run of non-space characters.

(d) void reverse_words(int s[], int n)

Reverse the order of words in the integer string in-place. E.g. "hello world foo" becomes "foo world hello". (Hint: reverse the whole array, then reverse each word individually.)

(e) int run_length_encode(int src[], int n, int dst[])

Run-length encode the integer array: consecutive equal values are replaced by the count followed by the value. E.g. {5,5,5,2,2,7} encodes to {3,5,2,2,1,7}. Return the length of dst.

QUESTION 4 — Polynomial Arithmetic with 1D Arrays

Represent a polynomial of degree n as int coeff[n+1] where coeff[i] is the coefficient of x^i.

(a) void poly_add(int a[], int da, int b[], int db, int c[], int *dc)

Compute c = a + b. dc is the degree of c (max of da, db).

(b) void poly_mul(int a[], int da, int b[], int db, int c[], int *dc)

Compute c = a * b using the schoolbook $O(n^2)$ algorithm. Degree of c = da + db.

(c) long long poly_eval(int p[], int dp, int x)

Evaluate polynomial p at x using Horner's method in O(n): p(x) = p[n] + x*(p[n-1] + x*(...))

(d) void poly_diff(int p[], int dp, int deriv[], int *dd)

Compute the formal derivative: deriv[i] = (i+1) * p[i+1]. Degree drops by 1.

(e) void poly_print(int p[], int dp)

Print the polynomial in human-readable form, e.g. "3x^3 + 0x^2 - 2x + 5". Skip zero terms except when the polynomial is identically zero.

QUESTION 5 — Pathfinding in a 2D Maze

A maze is a 10×10 integer grid where 0 = open cell and 1 = wall. Find paths from the top-left (0,0) to the bottom-right (9,9).

(a) bool bfs_path(int maze[10][10], int path_r[], int path_c[], int *plen)

Use Breadth-First Search with an integer queue (arrays of row and column indices, capacity 100) to find the shortest path. Store the parent of each cell in parent_r[10][10] and parent_c[10][10]. Return true if a path exists and fill path_r[], path_c[], and *plen with the reconstructed path.

(b) int count_paths_dp(int maze[10][10])

Using dynamic programming (no BFS, no recursion), count the number of distinct paths from (0,0) to (9,9) moving only right or down. dp[r][c] = number of ways to reach (r,c).

(c) void print_maze_with_path(int maze[10][10], int path_r[], int path_c[], int plen)

Print the maze using '#' for walls, '.' for open cells, and '*' for cells on the shortest path found in (a).

QUESTION 6 — Numerical Methods with Integer Arrays

All calculations use scaled integer arithmetic. Scale factor S = 10000 (represents 4 decimal places).

(a) Integer square root table

Using only integer arithmetic and loops, build a 1D array isqrt[101] where isqrt[i] = floor(sqrt(i)) for i in [0,100]. Use Newton's method: x_{k+1} = (x_k + i/x_k) / 2, starting from x_0 = i. Stop when |x_{k+1} - x_k| < 1. Print the table.

(b) Prefix sum array & range queries

Given array a[16], build prefix[17] where prefix[0]=0 and prefix[i]=prefix[i-1]+a[i-1]. Use it to answer sum(l,r) = prefix[r+1] - prefix[l] in O(1). Answer 5 range queries.

(c) Difference array for range updates

Given array a[16] all zeros, apply 4 range-increment operations using a difference array diff[]: diff[l] += val, diff[r+1] -= val. Reconstruct a[] from diff[] using a prefix sum pass, then print the result.

(d) 2D prefix sums for rectangle queries

Given a 6×6 integer grid, build a 2D prefix sum table P[7][7]. Answer sum(r1,c1,r2,c2) queries in O(1) using the inclusion-exclusion formula: P[r2+1][c2+1] - P[r1][c2+1] - P[r2+1][c1] + P[r1][c1].

QUESTION 7 — Classic Dynamic Programming

Implement the following DP problems using arrays.

(a) 0/1 Knapsack

Given n=8 items each with a weight and value, and a knapsack capacity W=50, compute the maximum value achievable. Use a 2D dp[9][51] table where dp[i][w] = max value using first i items with capacity w. Also reconstruct which items were selected.

(b) Longest Common Subsequence

Given two integer arrays A[8] and B[7], compute the length of their Longest Common Subsequence using dp[9][8]. Reconstruct and print the actual LCS.

(c) Longest Increasing Subsequence

Find the length of the Longest Increasing Subsequence of a 1D integer array of length 12 using the $O(n^2)$ DP approach with a 1D dp[] array.

QUESTION 8 — Digital Signal Processing with Integer Arrays

All signals are 1D integer arrays of 64 samples. Amplitude range: [-32768, 32767] (16-bit).

(a) void moving_average(int sig[], int n, int k, int out[])

Compute the k-point moving average. Since we avoid floats, multiply by k and divide at the end. Handle edge cases at the start and end using smaller windows. Output out[i] = sum(sig[i-k/2 .. i+k/2]) / actual_window_size.

(b) void find_peaks(int sig[], int n, int peaks[], int *np, int threshold)

Find all local maxima greater than threshold. A peak is any sig[i] such that sig[i] > sig[i-1] and sig[i] > sig[i+1] and sig[i] > threshold. Store peak indices in peaks[].

(c) void cross_correlate(int a[], int b[], int n, int corr[], int *clen)

Compute the cross-correlation of two signals a[] and b[] of length n at lags 0 to n-1. corr[lag] = sum_{i=0}^{n-1-lag} a[i] * b[i+lag] / n (integer division). *clen = n.

(d) void rms_and_zscore(int sig[], int n, int rms_out[], int zscore_out[])

Compute: (i) RMS = integer_sqrt(sum(sig[i]^2) / n), store in rms_out[0]. (ii) The z-score of each element: z[i] = (sig[i] - mean) * 1000 / std_dev, using integer arithmetic. The result is z-score scaled by 1000. Use your isqrt logic inline.

Question 9 — Elementary Cellular Automaton (Rule 110)

An elementary cellular automaton evolves a 1D row of binary cells. At each step, the next state of cell i depends on cells i-1, i, i+1 from the current generation. The 3-bit neighbourhood encodes an index 0–7; the rule number (e.g. 110) provides the output for each index as a bit.

```
/* Rule 110 (binary 01101110): */
/* neighbourhood: 111 110 101 100 011 010 001 000 */
/* output:          0   1   1   0   1   1   1   0 */
/* So rule[idx] = (110 >> idx) & 1                */
```

(a) Initialise a 1D bool array of 64 cells with a single 1 in the centre. Run 32 generations of Rule 110. Store the full history in a 2D bool grid[32][64]. Print each generation as a row of '#' and ' ' characters.

(b) Detect if the automaton enters a cycle (a generation that was seen before) within 32 steps. Use a 2D comparison approach: compare current row with all previous rows.

(c) Count the total number of live cells across all 32 generations and report the density (live / total) scaled by 1000 (integer).

QUESTION 10 — Branchless Absolute Value, Sign & Min/Max

Implement the following four expressions using only arithmetic and bitwise operations. No if/else, no ternary operator, no branch of any kind.

(a) int abs_val(int x)

Return the absolute value of x without any branch.

(b) int sign(int x)

Return +1 if x > 0, -1 if x < 0, and 0 if x == 0. No branches.

(c) int branchless_min(int a, int b)

Return the smaller of a and b without any conditional expression.

(d) int branchless_max(int a, int b)

Return the larger of a and b without any conditional expression.

QUESTION 11 — Bit Manipulation Suite

Using only bitwise and arithmetic operators, implement:

(a) int count_set_bits(unsigned int n)

Return the number of 1-bits in n (population count / Hamming weight). No loops. Expand the operations manually for a 32-bit integer.

(b) unsigned int reverse_bits(unsigned int n)

Reverse the bit pattern of a 32-bit unsigned integer.

(c) int is_power_of_two(int n)

Return 1 if n is an exact power of two, 0 otherwise. One expression only.

(d) int next_power_of_two(unsigned int n)

Return the smallest power of 2 that is >= n. No loops; use bitwise OR/shift only.

(e) int parity(unsigned int n)

Return 1 if n has an odd number of set bits, 0 if even. No loops.

QUESTION 12 — Arithmetic Without Standard Operators

Implement the following using ONLY the operators: + - & | ^ ~ >> <<

Do NOT use * / % in any of the implementations below.

(a) int multiply(int a, int b)

Multiply two integers using only addition and bit shifts (Russian peasant / binary multiplication unrolled for 32 bits).

(b) int divide(int a, int b)

Integer division a / b (assume b != 0 and no overflow) using only subtraction and bit shifts. Return the quotient.

(c) int modulo(int a, int b)

Compute a % b using only your divide() function and subtraction.

QUESTION 13 — Integer Digit Operations

Using only integer arithmetic (+ - * / %) and conditionals, implement:

(a) int digit_sum(int n)

Return the sum of digits of n (treat negative n as its absolute value). Write as a single compound expression — no assignment inside.

(b) int is_palindrome_int(int n)

Return 1 if the decimal representation of n is a palindrome (e.g. 12321 -> 1), 0 otherwise. Only integers; no string conversion.

(c) int digital_root(int n)

Return the digital root of n: repeatedly sum digits until a single digit remains. Express using a closed-form integer formula — no recursion, no loop.

(d) int count_digits(int n)

Return the number of decimal digits in n. Express as a closed-form expression using only integer comparisons.

QUESTION 14 — Overflow Detection & Saturating Arithmetic

Safe integer arithmetic requires detecting overflow before it occurs. Using only integer operations and conditionals:

(a) int add_overflows(int a, int b)

Return 1 if a + b would overflow a signed 32-bit integer, 0 otherwise. Do NOT actually perform the potentially overflowing addition first.

(b) int saturating_add(int a, int b)

Return a + b, but if the result would overflow, clamp it to INT_MAX or INT_MIN instead. Branchless preferred.

(c) int safe_multiply_fits(int a, int b)

Return 1 if a * b fits in a signed 32-bit int without overflow, 0 otherwise. Use only 64-bit integers (long long) to check.

QUESTION 16 — Bitmask Puzzle Challenges

Solve each puzzle using a single expression (one return statement, no intermediate variables unless stated).

(a) Swap two integers x and y in-place using ONLY XOR — no temp variable.

(b) Given an array int a[8] of 0s and 1s, pack them into a single byte using only shifts and OR.

(c) Return the position (0-indexed) of the highest set bit in an unsigned int n. Return -1 if n == 0. Use only shifts and conditionals. No log2().

QUESTION 17 — The Three-Way Comparison & Sorting Network

A sorting network sorts a fixed number of inputs using a predetermined sequence of compare-and-swap operations. All branches are known at compile time.

(a) int cmp3(int a, int b, int c)

Return 0 if a <= b <= c (sorted ascending), -1 if a >= b >= c (sorted descending), or 1 otherwise.

(b) void sort3(int *a, int *b, int *c)

Sort three integers into ascending order using ONLY compare-and-swap operations. A compare-and-swap on (x, y) sets x = min(x,y) and y = max(x,y). No branches, no temp variables beyond those inside a single CAS macro.

(c) void sort4(int *a, int *b, int *c, int *d)

Extend to sort four integers with the optimal 5-comparator sorting network.

QUESTION 18 — Merge Sort & Inversion Count on a Linked List

Sorting arrays is straightforward, but sorting a linked list requires rethinking the algorithm — you cannot use random access. Merge sort is the canonical O(n log n) sort for linked lists.

```
typedef struct Node {
    int          val;
    struct Node *next;
} Node;
```

(a) Node *merge_sorted_lists(Node *a, Node *b)

Merge two already-sorted singly linked lists into one sorted list. Return the new head. Do NOT allocate any new nodes — reuse and relink existing ones. Iterative solution required.

(b) Node *merge_sort_ll(Node *head)

Sort a singly linked list using merge sort. Use the fast/slow pointer technique to find the midpoint and split the list into two halves. Return the new sorted head.

(c) long long merge_sort_inversion_count(Node *head, Node **sorted_head)

Extend your merge sort to simultaneously count inversions — pairs of positions (i, j) where i < j but val[i] > val[j]. During the merge step, when an element from the right half is placed before elements remaining in the left half, add the count of remaining left elements to the inversion count. Return the total inversion count and write the sorted head to *sorted_head.

QUESTION 19 — Polynomial Arithmetic via Sorted Linked Lists

Represent a sparse polynomial as a sorted linked list of terms. Each node stores a coefficient and an exponent; the list is sorted in descending order of exponent. Zero-coefficient terms are never stored.

```
typedef struct Term {
    long long    coeff;   /* coefficient (may be negative)  */
    int          exp;     /* exponent >= 0                  */
    struct Term *next;
} Term;
```

(a) Term *poly_insert_term(Term *head, long long coeff, int exp)

Insert a new term into the sorted polynomial. If a term with the same exponent already exists, add the coefficients. If the resulting coefficient is zero, remove that term entirely. Maintain descending exponent order.

(b) Term *poly_add(Term *p, Term *q)

Return a NEW polynomial representing p + q. Do not modify p or q. Traverse both lists simultaneously (like merge), adding coefficients for equal exponents.

(c) Term *poly_multiply(Term *p, Term *q)

Return a NEW polynomial representing p * q. Use repeated poly_insert_term on a result list starting from NULL. Time complexity O(m*n) where m and n are the number of terms.

(d) void poly_print(Term *head)

Print in human-readable form: e.g. "6x^4 - 3x^2 + 2x - 5". Skip zero coefficients.

(e) long long poly_eval(Term *head, int x)

Evaluate the polynomial at integer x using Horner-like iteration over the node list. Since exponents may not be consecutive, compute each term's contribution as coeff * x^exp using integer power.

QUESTION 20 — ☠ The Dungeon of Cycles ☠

You are an adventurer navigating the infamous Dungeon of Cycles — a labyrinth so twisted that corridors loop back on themselves. The dungeon is represented as a singly linked list of rooms. A cycle means two corridors lead to the same room, trapping the adventurer forever. Your mission: detect traps, find them, and escape by breaking the cycle.

The dungeon is encoded as a singly linked list of Room nodes:

```
typedef struct Room {
    int          id;
    char         name[32];
    struct Room *next;   /* corridor to the next room (may loop!) */
} Room;
```

(a) bool has_cycle(Room *head)

Detect whether the dungeon contains a cycle using Floyd's Tortoise and Hare algorithm. The slow pointer advances one step; the fast pointer advances two steps. If they ever meet, a cycle exists. O(n) time, O(1) space — no extra arrays or hash sets permitted.

(b) Room *find_cycle_entry(Room *head)

If a cycle exists, return a pointer to the room where the cycle begins (the first room visited twice). After Floyd's algorithm finds a meeting point inside the cycle, reset one pointer to head. Advance both one step at a time — they meet exactly at the cycle entry. Return NULL if no cycle.

(c) int cycle_length(Room *head)

If a cycle exists, return its length (number of rooms in the cycle). Once you have a meeting point from Floyd's algorithm, keep one pointer fixed and advance the other until it returns to the meeting point. Return 0 if no cycle.

(d) void remove_cycle(Room *head)

Repair the dungeon by removing the cycle: find the node whose next pointer causes the cycle (the node just before the entry point, going around the cycle) and set its next to NULL. After this call the list must be a valid linear singly linked list with no cycle.

(e) Room *reverse_section(Room *head, int from_id, int to_id)

Reverse the sub-section of rooms from the room with id == from_id to the room with id == to_id (both inclusive). All other rooms remain in their original order. Return the new head.

QUESTION 21 — 🧟 Zombie Apocalypse Survivor Queue 🧟

The city has fallen. You are managing the last rescue helicopter with limited capacity. Survivors arrive continuously and are organised in a priority queue implemented as a SORTED doubly linked list — most critical cases at the front. Survivors have an urgency score: higher = more critical. When the helicopter is full, the lowest-urgency survivor is bumped. Complications arise: survivors can be wounded mid-queue (urgency increases), and occasionally a group of survivors arrives in a scrambled order and must be merged in efficiently.

```
typedef struct Survivor {
    int             id;
    char            name[40];
    int             urgency;    /* higher = more critical, must be rescued
sooner */
    int             wounds;     /* wound count */
    struct Survivor *prev, *next;
} Survivor;


typedef struct {
    Survivor *head;   /* most critical (highest urgency) */
    Survivor *tail;   /* least critical */
    int       size;
    int       capacity;
} RescueQueue;
```

(a) RescueQueue *rq_create(int capacity)

(b) bool rq_admit(RescueQueue *rq, int id, const char *name, int urgency)

Insert a new survivor in sorted position (descending urgency). If at capacity, bump the tail (lowest urgency) survivor if the new arrival is more critical — free the bumped node. If new arrival is not more critical and queue is full, discard (do not insert). Return true if inserted.

(c) void rq_wound(RescueQueue *rq, int id, int additional_urgency)

A survivor with the given id has been wounded — increase their urgency by additional_urgency. Remove them from their current position in the doubly linked list and re-insert at the correct sorted position. O(n) total.

(d) Survivor *rq_evacuate(RescueQueue *rq)

Remove and return the most critical survivor (head of list). Caller must free the returned node. Return NULL if empty.

(e) void rq_merge_group(RescueQueue *rq, Survivor *group_head)

A sorted linked list of new survivors (single-linked, sorted descending by urgency) has arrived. Merge them all into the rescue queue one-by-one using rq_admit. Free any survivors from the group that were not admitted (bumped or rejected).

QUESTION 22 — 🚀 Space Station Round-Robin Scheduler 🚀

Aboard the orbital station HELIX-9, eight crew members share one communication terminal using a round-robin scheduler. The schedule is stored as a circular doubly linked list. Crew members join and leave mid-mission; the scheduler must adapt. Mission Control can inject priority overrides that skip ahead in the rotation. A catastrophic bug caused the schedule list to become corrupted — some entries appear twice. Your task: implement the scheduler and its repair.

```
typedef struct Crew {
    int          id;
    char         callsign[20];
    int          time_slice_ms;
    int          priority;       /* 0=normal, 1=high */
    struct Crew *prev, *next;    /* circular doubly linked */
} Crew;


typedef struct {
    Crew *current;   /* pointer to the currently active crew member */
    int   count;
} Scheduler;
```

(a) Scheduler *sched_create(void) / bool sched_add(Scheduler *s, int id, const char *cs, int ms, int pri)

Create scheduler. sched_add inserts a new crew member after the 'current' node (or as the only element if empty), maintaining circular doubly-linked structure. If priority==1, insert before all other priority-1 members at the front of the priority group.

(b) Crew *sched_next(Scheduler *s)

Advance current to the next crew member and return it. Skips no one — pure round-robin. Return NULL if scheduler is empty.

(c) bool sched_remove(Scheduler *s, int id)

Remove the crew member with the given id. If current is being removed, advance current to the next node before unlinking. Free the removed node. Return false if not found. The list must remain circular after removal.

(d) int sched_deduplicate(Scheduler *s)

The corruption has inserted duplicate ids. Remove all duplicate entries (keep the first occurrence of each id). Return the number of duplicates removed. Must run in $O(n^2)$ or better — use a bool seen[] array with a reasonable ID range [0,1023].

(e) void sched_print_rotation(Scheduler *s, int steps)

Starting from current, print the next 'steps' crew members in order (wrapping around the circle) without changing the current pointer.

QUESTION 23 — The Josephus Cipher

In ancient Masada, 41 soldiers chose death over capture. They stood in a circle and every k-th soldier was eliminated. Only one position was safe. This is the Josephus Problem. You must model the full ceremony using a circular singly linked list and extend it: after computing the survivor's position, encode a secret message by using the elimination ORDER as a permutation cipher.

(a) int josephus(int n, int k)

Model the Josephus problem using a circular singly linked list of n nodes (numbered 1 to n). Starting from node 1, count k steps and remove that node. Repeat until one node remains. Return that node's number. Free all removed nodes. O(n*k) time.

(b) int *josephus_order(int n, int k)

Return a heap-allocated array of length n containing the full elimination order (which soldier was eliminated at each step). The last element is the survivor. Caller must free.

(c) char *josephus_encrypt(const char *msg, int n, int k)

Use the elimination order as a permutation to encrypt msg. The message length must equal n. Create permutation P from josephus_order(n, k): P[i] = elimination_order[i] - 1 (0-indexed). Encrypted[i] = msg[P[i]]. Return a heap-allocated encrypted string (caller must free).

Question 24 — Skip List

It is 2087. The last database on Earth stores humanity's knowledge as a skip list — a probabilistic data structure layered above a linked list, giving O(log n) average search, insert, and delete without rotations or rebalancing. Each node may appear on multiple levels. The higher the level, the sparser the list — like highway express lanes. You must build and operate this structure from scratch.

```
#define MAX_LEVEL 8    /* maximum number of levels (0 = base) */

#define SKIP_P    2    /* promote to next level with prob 1/SKIP_P */


typedef struct SkipNode {

    int              key;

    int              val;

    int              level;              /* highest level this node reaches */

    struct SkipNode *forward[MAX_LEVEL]; /* forward[i] = next node at level i
*/

} SkipNode;


typedef struct {

    SkipNode *header;  /* sentinel head: key=INT_MIN, all levels non-NULL
initially */

    int       level;   /* current highest level in use (0-indexed) */

    int       size;

} SkipList;
```

(a) SkipList *sl_create(void)

Allocate the skip list and its header sentinel node. The header has key=INT_MIN and all forward[] pointers set to NULL. Level starts at 0.

(b) static int random_level(void)

Generate a random level using geometric distribution: start at 0, promote while rand()%SKIP_P == 0 and level < MAX_LEVEL-1. Return the level.

(c) bool sl_insert(SkipList *sl, int key, int val)

Insert key→val. Allocate a new SkipNode at a random level. Build an update[] array of MAX_LEVEL pointers — update[i] is the rightmost node at level i that precedes the insertion point. If key exists, update its value. Return false on allocation failure.

(d) bool sl_search(SkipList *sl, int key, int *val_out)

Search for key top-down, starting from the highest level. At each level, advance forward while forward[i]->key < key. Drop a level when no advance is possible. At level 0, check if the next node matches. O(log n) average.

(e) bool sl_delete(SkipList *sl, int key)

Delete key from the skip list using the update[] array technique. After relinking, free the deleted node. Update sl->level if top levels become empty. Return false if not found.

(f) void sl_print(SkipList *sl)

Print all levels from highest to lowest, showing which keys appear at each level. Format: 'Level 3: [12] -> [45]'

QUESTION 25 — Sieve of Eratosthenes & Prime Gaps

The Sieve of Eratosthenes finds all primes up to a limit N by iteratively marking multiples of each prime as composite.

(a) Implement the sieve for N = 200 using a fixed-size boolean array.

Declare bool sieve[201] and populate it so that sieve[i] == true means i is prime. Print all primes found.

(b) Using your sieve, find and print all twin prime pairs (p, p+2) up to 200.

(c) Find the largest prime gap (consecutive difference between adjacent primes) below 200, and print which two primes bound it.

QUESTION 26 — Integer Matrix Operations

Given 4×4 integer matrices stored as 2D int arrays, implement:

(a) Matrix multiplication of two 4×4 matrices.

(b) Matrix transposition in-place.

(c) Compute the trace (sum of the main diagonal) and determine whether the matrix is symmetric.

(d) Row-reduce the matrix (Gaussian elimination using only integer arithmetic — no division). Print the result.

QUESTION 27 — Collatz Conjecture & Sequence Analysis

The Collatz sequence starting from n is defined as: if n is even, next = n / 2; if n is odd, next = 3n + 1. The sequence terminates when it reaches 1.

(a) Write collatz_length(int n) — return the number of steps to reach 1.

(b) Find the starting number in [1, 1000] that produces the longest Collatz sequence. Print both the number and its length.

(c) For all starting values in [1, 50], find the maximum value the sequence ever reaches (the 'peak'). Print a table: start | length | peak.

Use long long for 3n+1 calculations — values can temporarily exceed INT_MAX even for small starts.

The longest Collatz chain below 1000 starts at 871 and has 178 steps.

QUESTION 28 — Big Integer Arithmetic (Array of Digits)

Represent arbitrarily large non-negative integers as arrays of int digits in base 10, stored least-significant digit first (digits[0] is the ones place). Maximum 50 digits.

```
#define MAXDIGITS 50

typedef struct { int digits[MAXDIGITS]; int len; } BigInt;
```

(a) BigInt big_add(BigInt a, BigInt b) — add two BigInts with carry propagation.

(b) BigInt big_mul_int(BigInt a, int k) — multiply BigInt by a regular int k.

(c) Use your functions to compute and print 20! (twenty factorial).

(d) Print the first 25 Fibonacci numbers using BigInt addition.

QUESTION 29 — Conway's Game of Life (10 × 10 Grid)

Conway's Game of Life evolves a grid of live (1) and dead (0) cells each generation according to these rules:

- A live cell with 2 or 3 live neighbours survives.
- A dead cell with exactly 3 live neighbours becomes alive.
- All other cells die or remain dead.

(a) Implement int count_neighbours(int g[10][10], int r, int c) — count the 8-directional live neighbours of cell (r,c), wrapping at boundaries (toroidal grid).

(b) Implement void next_gen(int cur[10][10], int nxt[10][10]) — compute one generation.

(c) Run 5 generations from the 'Blinker' pattern and print each generation. Detect if a steady state is reached before 5 steps.

QUESTION 30 — Integer Cipher & Modular Arithmetic

Work entirely with integers — encode characters as their ASCII integer values (int, range 32–126).

(a) Caesar cipher: encrypt and decrypt an integer array (ASCII codes) with shift k.

Wrap around within printable ASCII range [32, 126] (95 characters). Shift by k, wrapping: ((c - 32 + k) % 95) + 32.

(b) Modular exponentiation: int mod_pow(int base, int exp, int mod).

Compute (base^exp) % mod efficiently using repeated squaring. Must run in O(log exp) multiplications.

(c) RSA toy example: using p=61, q=53, e=17, compute d (the private exponent) and demonstrate encrypting and decrypting the integer message M=65.

QUESTION 31 — Pointer Fundamentals & Pointer Arithmetic

This question tests deep understanding of how pointers interact with memory, arrays, and function arguments.

(a) void swap_ints(int *a, int *b)

Swap two integers using only their pointers. Write two versions: one using a temporary variable, one using XOR (without a temporary).

(b) int *find_max_ptr(int *arr, int n)

Return a pointer to the maximum element of an integer array. Use pointer arithmetic (arr + i) rather than subscript notation (arr[i]) for traversal.

(c) void reverse_in_place(int *arr, int n)

Reverse an array in-place using two pointers: one starting at the beginning (int *lo = arr) and one at the end (int *hi = arr + n - 1). Swap elements by advancing lo and retreating hi.

(d) int count_positive(int *arr, int n)

Using only pointer arithmetic (no subscript operator [] allowed in the function body), count how many elements of the array are strictly positive.

(e) Pointer analysis

Given the declarations below, state the value of each expression. Explain the address arithmetic.

```
int a[]   = {10, 20, 30, 40, 50};
int *p    = a + 2;
int **pp = &p;


/* State the value of: */
/* 1. *p            */
/* 2. *(p - 1)      */
/* 3. *(p + 2)      */
/* 4. **pp          */
/* 5. *(*pp + 1)    */
/* 6. p[-2]         */
```

QUESTION 32 — Dynamic 1D Arrays & Resizable Vector

Implement a dynamically resizing integer vector (like C++'s std::vector) using a struct and heap memory.

```
typedef struct {
    int  *data;      /* heap-allocated array        */
    int   size;      /* current number of elements  */
    int   capacity;  /* allocated capacity          */
} Vector;
```

(a) Vector *vec_create(int initial_cap)

Allocate and initialise a Vector on the heap. Allocate the data array with initial_cap elements. Return NULL on failure.

(b) bool vec_push(Vector *v, int val)

Append val to the vector. If size == capacity, double the capacity using realloc(). Return false if realloc fails (keep original data intact). Return true on success.

(c) bool vec_insert(Vector *v, int idx, int val)

Insert val at position idx, shifting elements right. Grow capacity if needed. Return false on failure or out-of-range idx.

(d) bool vec_remove(Vector *v, int idx)

Remove element at idx, shifting elements left. Return false if out-of-range.

(e) void vec_destroy(Vector *v)

Free the data array and then free the Vector struct itself. Set the pointer to NULL via a Vector **vp parameter to prevent dangling pointer use.

QUESTION 33 — Singly Linked List with Full Operations

Implement a complete singly linked list of integers. Each node is heap-allocated.

```
typedef struct Node {
    int          val;
    struct Node *next;
} Node;


typedef struct {
    Node *head;
    int   size;
} LinkedList;
```

(a) LinkedList *ll_create(void)

Allocate and return an empty linked list (head=NULL, size=0).

(b) bool ll_push_front(LinkedList *ll, int val) | bool ll_push_back(LinkedList *ll, int val)

Insert a new node at the front or back of the list.

(c) bool ll_insert_sorted(LinkedList *ll, int val)

Insert val into a sorted linked list while maintaining ascending order. Do not sort after insertion — find the correct position during traversal.

(d) bool ll_delete(LinkedList *ll, int val)

Delete the first node whose val equals the target. Handle the special cases: deleting the head, and deleting from an empty list. Free the removed node.

(e) Node *ll_reverse(LinkedList *ll)

Reverse the linked list in-place using three pointers (prev, curr, next). Update ll->head. Return the new head.

(f) void ll_destroy(LinkedList **llp)

Free every node, then free the list struct. Set *llp = NULL.

QUESTION 34 — Doubly Linked List / Deque

Implement a doubly-linked-list deque (double-ended queue). Nodes have both next and prev pointers.

```
typedef struct DNode {
    int           val;
    struct DNode *prev;
    struct DNode *next;
} DNode;


typedef struct {
    DNode *head;
    DNode *tail;
    int    size;
} Deque;
```

(a) Deque *deque_create(void)

(b) bool deque_push_front(Deque *d, int val) / bool deque_push_back(Deque *d, int val)

Insert a node at the front or back. Maintain both head/tail pointers and prev/next links.

(c) bool deque_pop_front(Deque *d, int *out) / bool deque_pop_back(Deque *d, int *out)

Remove and return the front or back element into *out. Free the removed node. Return false if empty.

(d) void deque_rotate(Deque *d, int k)

Rotate the deque left by k positions: move k elements from the front to the back, one at a time. Use pop_front + push_back. Works for any k (including k > size; use k %= size).

(e) void deque_destroy(Deque **dp)

QUESTION 35 — Dynamic Stack, Queue & Function Pointers

Build a generic integer stack and queue backed by dynamic arrays, then use function pointers to apply transformations.

(a) Stack using dynamic array

Implement a stack (typedef Stack) backed by a heap-allocated int array that doubles capacity on overflow. Operations: stack_push(), stack_pop() (returns bool, writes result via int*), stack_peek(), stack_is_empty(), stack_free().

(b) Queue using circular dynamic array

Implement a queue (typedef Queue) using a circular buffer (heap-allocated array with head, tail, size, capacity fields). Operations: q_enqueue(), q_dequeue() (bool, writes via int*), q_is_empty(), q_free(). On full, double capacity and re-lay elements linearly.

(c) Function pointer: apply transformation

Write void vec_apply(int *arr, int n, int (*transform)(int)) that applies a function to every element of arr in-place. Then demonstrate with three transforms: square, negate, and abs_val (absolute value). Pass each as a function pointer.

QUESTION 36 — Heap-Allocated 2D Matrix & Sparse Matrix

Allocate 2D matrices entirely on the heap and implement matrix operations.

(a) int **mat_alloc(int rows, int cols) / void mat_free(int **m, int rows)

Allocate a 2D matrix as an array of row pointers (int **). Each row is a separately heap-allocated int array. Free in reverse order: first each row, then the pointer array.

(b) int **mat_multiply(int **A, int **B, int ra, int ca, int cb)

Multiply matrices A (ra×ca) and B (ca×cb). Allocate and return a new ra×cb result matrix. Return NULL on allocation failure.

(c) Sparse matrix using coordinate list (COO format)

A sparse matrix stores only non-zero entries. Implement:

- typedef struct { int row, col, val; } Entry;
- typedef struct { Entry *entries; int nnz; int rows; int cols; } SparseMatrix;
- SparseMatrix *sparse_create(int rows, int cols, int max_nnz) — allocate entry array.
- bool sparse_set(SparseMatrix *sm, int r, int c, int val) — insert or update an entry (if val==0, remove it).
- int sparse_get(SparseMatrix *sm, int r, int c) — return the value at (r,c), or 0 if absent.
- void sparse_free(SparseMatrix **smp) — free the entries array and the struct.

QUESTION 37 — Dynamic Binary Search Tree

Implement a fully dynamic BST where every node is heap-allocated and freed correctly.

```
typedef struct BSTNode {
    int              key;
    int              count;   /* support duplicate keys */
    struct BSTNode  *left;
    struct BSTNode  *right;
} BSTNode;
```

(a) BSTNode *bst_insert(BSTNode *root, int key)

Recursively insert key. If key already exists, increment count. Return the (possibly new) root.

(b) bool bst_search(BSTNode *root, int key, int *count_out)

Search for key iteratively. If found, write its count to *count_out and return true.

(c) BSTNode *bst_delete(BSTNode *root, int key)

Delete one occurrence of key: if count > 1, decrement. Otherwise remove the node. Handle all three BST deletion cases: leaf, one child, two children (replace with in-order successor). Free the deleted node.

(d) void bst_inorder(BSTNode *root, int *out, int *idx)

Write keys in ascending (in-order) order to the array out[]. Recursion is permitted here only.

(e) int bst_height(BSTNode *root)

Return the height of the tree (height of empty tree = -1, leaf = 0) recursively.

(f) void bst_free(BSTNode **rootp)

Post-order recursive traversal to free every node. Set *rootp = NULL.

QUESTION 38 — Custom Memory Pool Allocator

Implement a fixed-block memory pool that manages a large chunk of heap memory internally, allowing fast allocation and deallocation without repeated malloc/free calls.

```
/* Pool layout: one large malloc'd block split into BLOCK_SIZE-byte chunks. */
/* A free-list of int indices tracks which chunks are available.           */
typedef struct {
    void  *memory;        /* single large malloc'd buffer               */
    bool  *used;          /* bool array: used[i]=true if chunk i in use */
    int    total_blocks;  /* total number of fixed-size blocks          */
    int    block_size;    /* size of each block in bytes                */
    int    free_count;    /* number of currently free blocks            */
} MemPool;
```

(a) MemPool *pool_create(int total_blocks, int block_size)

Allocate the pool: one large malloc of total_blocks * block_size bytes for memory, and a calloc'd bool array for used[]. Set free_count = total_blocks.

(b) void *pool_alloc(MemPool *pool)

Find the first free block (scan used[]), mark it used, decrement free_count, and return a pointer to the start of that block within memory. Return NULL if no blocks are free.

(c) bool pool_free(MemPool *pool, void *ptr)

Given a pointer previously returned by pool_alloc, compute the block index (ptr arithmetic: index = (char*)ptr - (char*)pool->memory) / block_size), validate it is in range and currently used, mark it free, increment free_count. Return false for invalid pointer.

(d) void pool_stats(MemPool *pool)

Print: total blocks, free blocks, used blocks, and pool utilisation as a percentage (integer arithmetic).

(e) void pool_destroy(MemPool **pp)

Free memory buffer, free used[] array, free MemPool struct, set *pp = NULL.

QUESTION 39 — Recursive String Compression (Run-Length Encoding)

Write a recursive C function that performs run-length encoding on a string. Given a string such as "aaabbbccddddee", produce "3a3b2c4d2e". The function must NOT use any loop.

Function signature:

```
void rle_encode(const char *s, char *out, int i, int count);
```

Where s is the input, out is the output buffer, i is the current index in s, and count is the current run length.

Also write a driver that demonstrates the function on three test strings.

Analyse the time and space complexity of your solution.

QUESTION 40 — Frame-Stewart Tower of Hanoi (4-Peg Variant)

The classic Tower of Hanoi uses 3 pegs and requires 2^n-1 moves. With 4 pegs (Frame-Stewart algorithm), fewer moves are needed.

(a) Implement the standard 3-peg Tower of Hanoi recursively and count moves.

(b) Implement the 4-peg variant. Use the Frame-Stewart strategy: move the top k disks to a spare peg using all 4 pegs, move the remaining n−k disks using 3 pegs, then move the k disks from the spare peg to the target using all 4 pegs.

Print every disk move as: "Move disk D from peg X to peg Y"

QUESTION 41 — Recursive Merge Sort with Inversion Count

Implement merge sort recursively. During the merge step, also count the number of inversions in the array (pairs i < j where arr[i] > arr[j]).

(a) Write merge_sort_count(int *arr, int left, int right) that sorts arr[left..right] and returns the inversion count.

(b) What is the time complexity of counting inversions this way vs a naive $O(n^2)$ approach?

QUESTION 42 — Generate All Valid Parenthesisations

Write a recursive C function that generates and prints ALL valid combinations of n pairs of parentheses. For n=3, one valid output is "((()))".

(a) Implement generate_parens(char *buf, int pos, int open, int close, int n).

(b) How many valid strings exist for a given n? Name the mathematical sequence.

QUESTION 43 — Recursive Flood Fill & Island Counting

You are given an N×M grid of 0s (water) and 1s (land). Two land cells are part of the same island if they are 4-directionally adjacent.

(a) Implement void flood_fill(int grid[][MAX_COLS], int r, int c, int rows, int cols) that marks all cells of the current island as visited (e.g., sets them to 2).

(b) Implement int count_islands(int grid[][MAX_COLS], int rows, int cols) that returns the total number of distinct islands using flood_fill.

QUESTION 44 — Power Set & Subset Sum

(a) Write a recursive function that prints ALL subsets (power set) of a given integer array of size n. For n=3 the power set has 2^3=8 subsets.

(b) Extend your solution to print ONLY subsets whose elements sum to a given target T (subset sum problem).

QUESTION 45 — N-Queens Problem with Solution Count

The N-Queens problem asks: in how many ways can N non-attacking queens be placed on an N×N chessboard?

(a) Implement int n_queens(int board[], int row, int n) that places queens row by row and returns the total number of valid arrangements.

(b) Explain the pruning strategy and state its effect on worst-case performance.

QUESTION 46 — Mutual Recursion: Even/Odd Digit Classifier

Using ONLY mutual recursion (two functions that call each other), determine whether the number of even digits in an integer equals the number of odd digits.

No loops. No helper variables. No strlen. Use only the two mutually recursive functions.

QUESTION 47 — Four Sorting Algorithms on Structs

You are given the following struct representing a student record:

```
typedef struct {
    char  name[64];
    int   id;
    float gpa;          /* stored as int * 100 — e.g. 385 means 3.85 */
    int   year;         /* 1, 2, 3, or 4 */
} Student;
```

(a) Insertion Sort by GPA ascending

void insertion_sort_gpa(Student *arr, int n) — sort the array in-place by gpa (ascending). Insertion sort must be used.

(b) Selection Sort by name lexicographically

void selection_sort_name(Student *arr, int n) — sort in-place by name (ascending lexicographic order). Use strcmp. Selection sort must be used.

(c) Shell Sort by ID descending

void shell_sort_id_desc(Student *arr, int n) — sort by id (descending). Use the Knuth gap sequence: start at h=1, while h < n/3: h = 3h+1; shrink by h = h/3.

(d) Stability test

Explain with a proof-by-example whether your insertion sort (a) is stable. Then explain whether your selection sort (b) is stable. Write a small test main() that demonstrates this by sorting an array where two students share the same GPA but different names, and show the output before and after sorting.

QUESTION 48 — Merge Sort & Quick Sort on Dynamic Arrays

Implement the two divide-and-conquer sorting algorithms on a heap-allocated integer array.

(a) Merge Sort with inversion count

Implement void merge_sort(int *arr, int left, int right, long long *inv_count). During the merge step, count the number of inversions (pairs i < j where arr[i] > arr[j]). Return the total inversion count via the pointer. Use auxiliary heap memory for the merge buffer — allocate and free inside the merge function.

(b) Three-way Quick Sort (Dutch National Flag partition)

Implement void quicksort_3way(int *arr, int lo, int hi). Use a three-way partition (Dijkstra's Dutch National Flag): partition into three regions: arr[lo..lt-1] < pivot, arr[lt..gt] == pivot, arr[gt+1..hi] > pivot. Choose pivot as the median of arr[lo], arr[mid], arr[hi] to avoid worst-case $O(n^2)$ on sorted input.

QUESTION 49 — Heap Sort & Priority Queue

A max-heap is an array where arr[i] >= arr[2i+1] and arr[i] >= arr[2i+2]. Implement heap sort and a dynamic priority queue.

(a) void heapify(int *arr, int n, int i)

Sift-down from index i in a heap of size n: compare node i with its two children and swap with the largest, then recurse on the affected child. This is the O(log n) restore-heap-property operation.

(b) void heap_sort(int *arr, int n)

Build a max-heap in O(n) by calling heapify bottom-up (from n/2-1 down to 0). Then repeatedly swap arr[0] with arr[n-1], reduce heap size by 1, and call heapify(arr, reduced_n, 0).

(c) Dynamic Max-Priority Queue

Using a heap-allocated array, implement a max-priority queue with these operations:

- PQ *pq_create(int capacity) — malloc a PQ struct and its data array.
- bool pq_push(PQ *pq, int val) — insert val and sift-up to restore heap property. Grow (realloc) if full.
- bool pq_pop(PQ *pq, int *out) — remove and return the maximum. Swap root with last element, shrink size, sift-down.
- int pq_peek(PQ *pq) — return the maximum without removing it.
- void pq_free(PQ **p) — free all heap memory, set *p = NULL.

QUESTION 50 — Searching Algorithms Toolkit

Implement the following search routines on various data structures. All must return index / pointer or -1 / NULL on failure.

(a) int binary_search_recursive(int *arr, int lo, int hi, int target)

Classic recursive binary search. Return index or -1. O(log n) time.

(b) int interpolation_search(int *arr, int n, int target)

Improve on binary search for uniformly distributed data. Estimate position: pos = lo + ((long long)(hi-lo) * (target - arr[lo])) / (arr[hi] - arr[lo]). Return index or -1. O(log log n) average on uniform data.

(c) int exponential_search(int *arr, int n, int target)

For unbounded/large arrays: find a range [i/2, i] where target might lie by doubling i (1, 2, 4, 8, ...) until arr[i] >= target or i >= n. Then binary search that range.

(d) int *two_sum_sorted(int *arr, int n, int target, int *i_out, int *j_out)

Given a sorted array, find two indices i < j such that arr[i] + arr[j] == target using a two-pointer approach in O(n). Write the indices to *i_out and *j_out. Return pointer to arr[*i_out] on success, NULL if no pair exists.

QUESTION 51 — String Sorting & Pattern Search

Work with dynamically allocated arrays of strings.

(a) Radix Sort on fixed-width strings

Sort an array of exactly 6-character strings (char[7] with null terminator) using LSD (least-significant-digit) radix sort. Process characters right-to-left, one column per pass using counting sort on the character value (0–127). The array itself is heap-allocated.

(b) KMP Pattern Search

Implement the Knuth-Morris-Pratt string search algorithm:

- int *build_failure(const char *pat, int m) — compute the failure (partial match) table. Heap-allocate an int array of size m. Return it (caller must free).
- int kmp_search(const char *text, const char *pat) — return the index of the first match in text, or -1. Use the failure table.
- int kmp_count_all(const char *text, const char *pat, int *positions, int max_pos) — find ALL non-overlapping matches, store indices in positions[], return count.

QUESTION 52 — Inventory Reorder System

A warehouse management system stores product records. You must implement three operations — no algorithm name is given.

```
typedef struct {
    int   product_id;
    char  name[48];
    int   stock;       /* current units in warehouse */
    int   reorder_pt;  /* reorder when stock <= reorder_pt */
    int   price_cents; /* price in cents */
} Product;
```

(a) Product **products_below_reorder(Product *inv, int n, int *count)

Return a heap-allocated array of pointers to all products whose stock <= reorder_pt. Write the count into *count. The array of pointers must itself be heap-allocated. Caller must free the pointer array (but NOT the Product structs themselves).

Performance note: the operation must complete in O(n) — think about what that implies for the algorithm you choose.

(b) Product *find_by_id(Product *inv, int n, int id)

Given that the inventory array may NOT be sorted, find the product with the given ID as efficiently as possible. If the array has more than 50 products, sort a heap-allocated copy by product_id first and then binary search it. For smaller arrays, linear scan is acceptable. Return a pointer into the original array, or NULL.

(c) void top_k_expensive(Product *inv, int n, int k, Product **result)

Fill result[] with pointers to the k most expensive products. The operation must run in O(n log k) — think carefully about which data structure gives you this complexity. Do NOT fully sort the array.

QUESTION 53 — Log File Analyser

You receive a dynamically allocated array of log entry structs parsed from a server log. Implement three analytics functions. No algorithm names are given — choose the most appropriate ones.

```
typedef struct {
    int  timestamp;      /* Unix timestamp (seconds) */
    int  level;          /* 0=DEBUG 1=INFO 2=WARN 3=ERROR 4=FATAL */
    int  user_id;
    char message[128];
} LogEntry;
```

(a) LogEntry **entries_in_range(LogEntry *logs, int n, int t_start, int t_end, int *count)

Return all log entries whose timestamp is in [t_start, t_end] inclusive, in chronological order. The function must run faster than O(n²). Hint: the logs array is NOT guaranteed to be sorted on input.

(b) int *error_frequency_by_hour(LogEntry *logs, int n, int *hour_count)

Count how many ERROR (level >= 3) entries fall in each hour-of-day (0–23). Return a heap-allocated int[24] array (caller must free). Fill *hour_count = 24. Use only integer arithmetic: hour = (timestamp % 86400) / 3600.

(c) char *most_frequent_user_message(LogEntry *logs, int n)

Find the user_id who has the most log entries. Among all messages from that user, find the lexicographically smallest message and return a heap-allocated copy of it. Run in O(n log n).

QUESTION 54 — Meeting Room Scheduler

You are building a meeting room booking system. No sorting or searching function names are mentioned — you must identify the correct approach.

```
typedef struct {
    int  start;    /* start time (minutes from midnight, 0-1440) */
    int  end;      /* end time   (exclusive) */
    char organiser[48];
    int  room_id;
} Meeting;
```

(a) int min_rooms_required(Meeting *meetings, int n)

Find the minimum number of meeting rooms needed to host all meetings with no overlaps. Two meetings overlap if one starts before the other ends. No algorithm name given — find the optimal O(n log n) approach.

(b) Meeting **non_overlapping_max_set(Meeting *meetings, int n, int *count)

Return the maximum number of non-overlapping meetings (Activity Selection Problem). Return a heap-allocated array of pointers to the selected meetings in the original array, and set *count. Optimal greedy solution required.

(c) bool has_any_overlap(Meeting *meetings, int n)

Return true if ANY two meetings (regardless of room) overlap in time. Must run in O(n log n) — a naive O(n^2) double loop receives zero marks. Justify your algorithm choice in a comment.

QUESTION 55 — Rebuild the C String Library

Re-implement the following standard string functions from scratch using only char pointers, loops, and arithmetic. Do NOT use any <string.h> function.

(a) size_t my_strlen(const char *s)

Return the number of characters before the null terminator.

(b) char *my_strcpy(char *dst, const char *src)

Copy src into dst including the null terminator. Return dst. Assume dst has sufficient space.

(c) char *my_strcat(char *dst, const char *src)

Append src to the end of dst. Return dst. Use my_strlen to find the end of dst first.

(d) int my_strcmp(const char *a, const char *b)

Return negative if a < b, 0 if equal, positive if a > b. Compare unsigned char values lexicographically.

(e) char *my_strstr(const char *hay, const char *needle)

Find the first occurrence of needle in hay. Return a pointer to the start of the match, or NULL if not found. Implement the naive O(n*m) sliding-window search.

(f) char *my_strrev(char *s)

Reverse string s in-place. Return s. Use two pointers (head and tail).

(g) int my_atoi(const char *s)

Convert a decimal integer string to int. Handle optional leading +/- sign and leading whitespace. Stop at the first non-digit character.

QUESTION 56 — Advanced String Operations (No <string.h>)

Implement the following advanced operations. You may call functions from Q1.

(a) char *my_strdup(const char *s)

Allocate a heap copy of s (malloc of strlen+1 bytes), copy it, and return the pointer. Caller is responsible for freeing it. Return NULL on malloc failure.

(b) int my_split(const char *src, char delim, char **parts, int max_parts)

Split src by single character delimiter delim. Store pointers to heap-allocated substrings in parts[]. Return the number of parts found (at most max_parts). The caller must free each parts[i].

(c) char *my_join(char **parts, int n, const char *sep)

Join n strings from parts[] with separator sep between them. Return a heap-allocated result string. Caller must free it.

(d) char *my_trim(const char *s)

Return a heap-allocated copy of s with leading and trailing whitespace (space, tab, newline) removed.

(e) bool my_startswith(const char *s, const char *prefix)

Return true if s starts with prefix.

(f) bool my_endswith(const char *s, const char *suffix)

Return true if s ends with suffix. Use my_strlen to calculate positions.

QUESTION 57 — String Analysis & Character Classification (No <string.h>)

Implement character classification and string analysis functions from scratch. No <ctype.h> and no <string.h>.

(a) Character classification macros/functions

Implement: is_alpha(c), is_digit(c), is_upper(c), is_lower(c), to_upper(c), to_lower(c) using only ASCII arithmetic. No <ctype.h>.

(b) void count_chars(const char *s, int *letters, int *digits, int *spaces, int *others)

Count letters, digits, spaces, and other characters in s. Use your is_* functions.

(c) bool is_palindrome(const char *s)

Return true if s is a palindrome. Ignore case and skip non-alphanumeric characters (e.g. "A man a plan a canal Panama" is a palindrome).

(d) int count_vowels(const char *s) / int count_consonants(const char *s)

Count vowels (aeiouAEIOU) and consonants (alphabetic but not vowels). Use your classification functions.

(e) char *my_itoa(int n, char *buf)

Convert integer n to its decimal string representation in buf. Handle negative numbers and n=0. Return buf. Do NOT use printf or sprintf — build the string digit by digit using n % 10 and n / 10, then reverse in-place.

QUESTION 58 — Dynamic String Builder & Pattern Matching (No <string.h>)

Build a heap-managed dynamic string type and implement pattern matching.

```
typedef struct {
    char *data;      /* heap-allocated, null-terminated */
    int   len;       /* current string length (not incl. '\0') */
    int   cap;       /* allocated capacity (including '\0') */
} StrBuf;
```

(a) StrBuf *sb_create(int initial_cap) / void sb_destroy(StrBuf **sbp)

Allocate/free the dynamic string buffer. Initial content is empty string.

(b) bool sb_append_char(StrBuf *sb, char c) / bool sb_append_str(StrBuf *sb, const char *s)

Append a character or string, growing capacity (×2) with realloc when needed.

(c) bool sb_insert(StrBuf *sb, int idx, const char *s)

Insert string s at byte position idx in the buffer, shifting existing content right. Grow if needed.

(d) bool sb_replace_all(StrBuf *sb, const char *from, const char *to)

Replace every non-overlapping occurrence of from with to in sb->data. Rebuild the string in a fresh heap buffer, then swap. Use only your Q1 functions (my_strlen, my_strstr, etc.).

QUESTION 59 — Dynamic Array of Strings & Sorting

Manage a heap-allocated array of heap-allocated strings, and sort them.

```
typedef struct {
    char **strings;   /* array of heap-allocated char * */
    int    count;
    int    capacity;
} StringArray;
```

(a) StringArray *sa_create(int cap) / void sa_destroy(StringArray **sap)

(b) bool sa_add(StringArray *sa, const char *s)

Append a heap-allocated duplicate of s. Grow capacity (×2) via realloc if needed.

(c) void sa_sort_asc(StringArray *sa) / void sa_sort_desc(StringArray *sa)

Sort the string array ascending and descending using insertion sort with strcmp() for comparison. Write two separate functions — do NOT use qsort.

(d) int sa_binary_search(StringArray *sa, const char *target)

Binary search on a sorted StringArray using strcmp(). Return the index or -1.

(e) void sa_remove_duplicates(StringArray *sa)

Remove duplicate strings (case-sensitive) from a sorted array in O(n) by shifting unique strings to the front. Free the duplicate heap strings.

QUESTION 60 — CSV Parser & In-Memory String Table

Parse CSV data into a 2D table of heap-allocated strings. Each row is a heap-allocated array of string pointers.

```
typedef struct {
    char ***cells;    /* cells[row][col] — each a heap string    */
    int   *col_count; /* number of columns in each row           */
    int    row_count;
    int    row_cap;
} CSVTable;
```

(a) CSVTable *csv_create(void) / void csv_destroy(CSVTable **tp)

(b) int csv_parse_line(const char *line, char **fields, int max_fields)

Parse one CSV line into individual field strings stored in fields[]. Handle: quoted fields ("..."), escaped quotes (""), and fields containing commas inside quotes. Return the number of fields parsed.

(c) bool csv_add_row(CSVTable *t, char **fields, int n)

Append a row of n fields to the table. Duplicate each field string onto the heap.

(d) void csv_print(CSVTable *t)

Print the table with columns padded to the maximum width of each column.

(e) CSVTable *csv_select_where(CSVTable *t, int col, const char *val)

Return a new CSVTable containing only rows where cells[row][col] equals val (using strcmp). Rows are deep-copied.

QUESTION 61 — Minimal Regular Expression Engine

Implement a recursive-descent regular expression matcher supporting:

- '.': matches any single character
- '*': matches zero or more of the preceding character (or '.' for any)
- '^': anchors match to start of string
- '$': anchors match to end of string
- '[abc]': character class — matches any one of the listed characters

(a) bool regex_match(const char *pattern, const char *text)

The top-level function. Handle '^' and '$' anchors. Internally dispatch to a helper that tries to match pattern at each position in text.

(b) bool regex_match_here(const char *p, const char *t)

Recursive core: if p[0]=='\0' return true (empty pattern matches). If p[1]=='*' try matching zero occurrences (skip p[0],p[1]) or one+more. If p[0]=='[' match a character class. Otherwise match a single char (including '.' wildcard).