ALA-TOO INTERNATIONAL UNIVERSITY

ALGORITHMIZATION AND PROGRAMMING PART II

Spring Semester 2025-2026

***

Computer science is no more about computers than astronomy is about telescopes.

— Edsger W. Dijkstra

WORLD ONE

BRUTE FORCE

TRY EVERYTHING UNTIL SOMETHING WORKS

NO assumptions about input

Checks all possible solutions

Guarantees correctness (if it finishes)

Performance explodes as problem size grows

WHEN BRUTE FORCE IS USED

SIMPLICITY OVER PERFORMANCE

Small input sizes

Prototyping and validation

Security testing

Educational and debugging purposes

Brute force is often written on purpose before optimization

BRUTE FORCE ALGORITHM

EXAMPLES

Linear Search

Naive String Matching

Exhaustive Permutation Search

Subset Enumeration

Password Guessing

Trial Division (Prime Checking)

BUILDING A MAGIC SQUARE BY FORCE

EXAMPLE - T000

Place numbers 1 to 9

In a 3 × 3 matrix

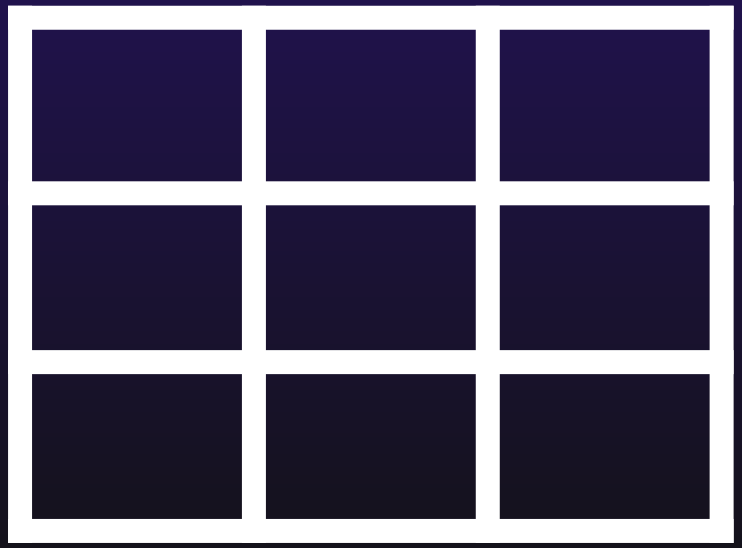
Each row, column, and diagonal must sum to 15

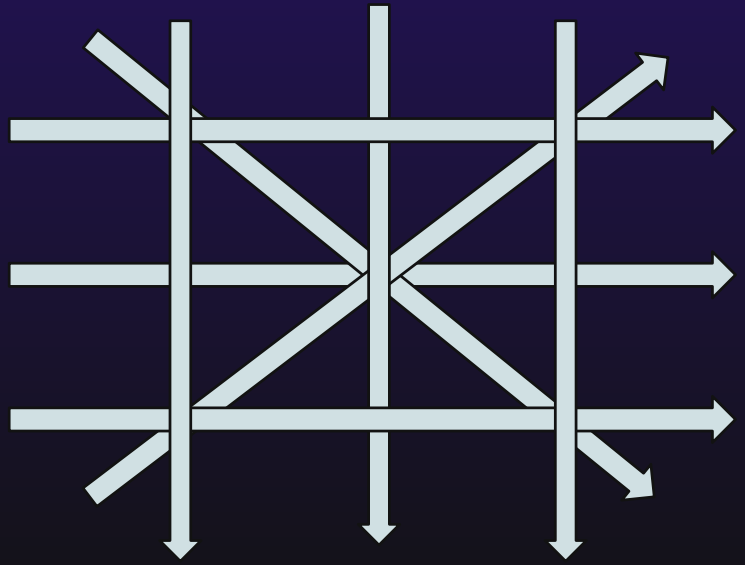
EXAMPLE - T000

TASK

INTEGERS: 1 2 3 4 5 6 7 8 9

15





KNOWN FACT

EXAMPLE - T000

Only fex valid solutions exist

Brute force does NOT “know” the answer

It finds it by trying everything

WHY THIS IS A BRUTE FORCE PROBLEM

NO SHORTCUTS, NO ASSUMPTIONS

All permutations of numbers 1–9

Total possibilities: **362,880**

Algorithm checks each permutation

Stops when a valid square is found

TRY TO CODE IT YOURSELF FIRST

TEST YOUR THINKING BEFORE LOOKING AHEAD

Struggle is part of learning

Mistakes reveal understanding

Solutions matter more after effort

HOW BRUTE FORCE SOLVES THIS

THE LOGIC BEHIND THE PROCESS

1 - Generate a permutation of numbers 1–9

2 - Place them into a 3×3 grid

3 - Check all rows

4 - Check all columns

5 - Check both diagonals

6 - If valid → stop

7 - If not → try next permutation

VISUALIZING THE SEARCH

EXHAUSTIVE EXPLORATION

The algorithm does not “learn”, it only checks

First permutation: rejected

Second permutation: rejected

…

Thousands rejected

One finally accepted

CODE BREAKDOWN

PERMUTATION

```
static void permute(int[] arr, int l) {
```

Fixes one position at a time

Swaps values to generate new arrangements

Recursively explores all branches

Total calls grow factorially

CODE BREAKDOWN

VALIDATION

Hard-coded checks for:

- Rows
- Columns
- Diagonals

Fast validation

Called hundreds of thousands of times

CODE BREAKDOWN

LOGIC

THE FIRST RUN

1	2	3
4	5	6
7	8	9

CODE BREAKDOWN

LOGIC

THE SECOND RUN

1	2	3
4	5	6
7	9	8

CODE BREAKDOWN

LOGIC

THE THIRD RUN

1	2	3
4	5	6
8	7	9

CODE BREAKDOWN

LOGIC

THE FOURTH RUN

1	2	3
4	5	6
8	9	7

CODE BREAKDOWN

LOGIC

1st RUN: 1 2 3 4 5 6 7 8 9

2nd RUN: 1 2 3 4 5 6 7 9 8

3rd RUN: 1 2 3 4 5 6 8 7 9

4th RUN: 1 2 3 4 5 6 8 9 7

5th RUN: 1 2 3 4 5 6 9 7 8

6th RUN: 1 2 3 4 5 6 9 8 7

...

CODE BREAKDOWN

LOGIC

40th RUN: 1 2 3 4 5 7 9 8 6

41st RUN: 1 2 3 4 5 8 6 7 9

42nd RUN: 1 2 3 4 5 8 6 9 7

43rd RUN: 1 2 3 4 5 8 7 6 9

44th RUN: 1 2 3 4 5 8 7 9 6

45th RUN: 1 2 3 4 5 8 9 6 7

...

CODE BREAKDOWN

LOGIC

155th RUN: 1 2 3 4 6 7 9 8 5

156th RUN: 1 2 3 4 6 7 9 5 8

157th RUN: 1 2 3 4 6 8 5 7 9

158th RUN: 1 2 3 4 6 8 5 9 7

159th RUN: 1 2 3 4 6 8 7 5 9

160th RUN: 1 2 3 4 6 8 7 9 5

...

CODE BREAKDOWN

LOGIC

40,320th RUN: 1 9 8 7 6 5 4 3 2

40,321st RUN: 2 1 3 4 5 6 7 8 9

40,322nd RUN: 2 1 3 4 5 6 7 9 8

40,323rd RUN: 2 1 3 4 5 6 8 7 9

40,324th RUN: 2 1 3 4 5 6 8 9 7

40,325th RUN: 2 1 3 4 5 6 9 7 8

…

CODE BREAKDOWN

LOGIC

45,357th RUN: 2 7 6 9 4 8 5 1 3

45,358th RUN: 2 7 6 9 4 8 5 3 1

45,359th RUN: 2 7 6 9 5 1 4 8 3

45,360th RUN: 2 7 6 9 5 1 8 3 4

45,361st RUN: 2 7 6 9 5 1 4 3 8

…

CODE BREAKDOWN

LOGIC

To find the very first solution, the code has to run through

45,361

combinations

T000

THE CODE

```java
public class T000 {
    static boolean isMagic(int[] p) {
        return p[0] + p[1] + p[2] == 15 &&
               p[3] + p[4] + p[5] == 15 &&
               p[6] + p[7] + p[8] == 15 &&
               p[0] + p[3] + p[6] == 15 &&
               p[1] + p[4] + p[7] == 15 &&
               p[2] + p[5] + p[8] == 15 &&
               p[0] + p[4] + p[8] == 15 &&
               p[2] + p[4] + p[6] == 15;        }
    static void permute(int[] arr, int l) {
        if (l == arr.length) {
            if (isMagic(arr)) {
                for (int i = 0; i < 9; i++) {
                    System.out.print(arr[i] + " ");
                    if (i % 3 == 2) System.out.println();       }
                System.exit(status: 0);     }       }
        for (int i = l; i < arr.length; i++) {
            int temp = arr[l];
            arr[l] = arr[i];
            arr[i] = temp;
            permute(arr, l + 1);
            temp = arr[l];
            arr[l] = arr[i];
            arr[i] = temp;      }       }
    Run | Debug
    public static void main(String[] args) {
        int[] nums = {1,2,3,4,5,6,7,8,9};
        permute(nums, l: 0);       }       }
```

WHAT IF

TOOO

What if the code run through all 362,880 possible combinations?

HOW MANY POSSIBLE ANSWERS EXISTS?

WHAT IF

T000

8

CODE BREAKDOWN

LOGIC

1	45,361	2 7 6 9 5 1 4 3 8
2	47,905	2 9 4 7 5 3 6 1 8
3	125,065	4 3 8 9 5 1 2 7 6
4	129,601	4 9 2 3 5 7 8 1 6
5	233,281	6 1 8 7 5 3 2 9 4
6	237,817	6 7 2 1 5 9 8 3 4
7	314,977	8 1 6 3 5 7 4 9 2
8	317,521	8 3 4 1 5 9 6 7 2

T001
MODIFY

Modify the existing Java program to find all 8 possible Magic Squares instead of stopping at the first one. You must track the "Run Number" for every combination tested and display it for each solution found.

T001

TIP

Remove the "Kill Switch": Identify and remove the line of code that forces the program to stop after the first success

T001

TIP

Add a global counter to create a variable (outside the recursive method) that increments every time isMagic() is called. This tracks the total number of permutations tested

T001
TIP

Update the output

When a solution is found, print:

The Solution Number (1 through 8).

The Run Number (the exact permutation count).

The 3x3 Grid format.

T001

OUTPUT

1	45,361	2 7 6 9 5 1 4 3 8
2	47,905	2 9 4 7 5 3 6 1 8
3	125,065	4 3 8 9 5 1 2 7 6
4	129,601	4 9 2 3 5 7 8 1 6
5	233,281	6 1 8 7 5 3 2 9 4
6	237,817	6 7 2 1 5 9 8 3 4
7	314,977	8 1 6 3 5 7 4 9 2
8	317,521	8 3 4 1 5 9 6 7 2

T002

TASK DESCRIPTION

A security vault uses a 4-digit code using the numbers {1, 2, 3, 4}. Each number is used exactly once. However, the vault has a "Security Delay" it takes 100 milliseconds for the vault to check if a code is correct.

Modify the permutation logic to find the secret code: "4 3 2 1"

T002
RESULT

Starting brute force attack on vault...

Attempt 1: [1, 2, 3, 4] ... WRONG (100ms delay)
Attempt 2: [1, 2, 4, 3] ... WRONG (100ms delay)
Attempt 3: [1, 3, 2, 4] ... WRONG (100ms delay)
...
Attempt 12: [2, 4, 3, 1] ... WRONG (100ms delay)
...
Attempt 18: [3, 4, 2, 1] ... WRONG (100ms delay)
...
Attempt 23: [4, 3, 1, 2] ... WRONG (100ms delay)
Attempt 24: [4, 3, 2, 1] ... SUCCESS!

VAULT UNLOCKED
Total attempts: 24
Time elapsed: 2400 milliseconds (2.4 seconds)

T003

TASK DESCRIPTION

Four friends: Alice (1), Bob (2), Charlie (3), and David (4) are going to the cinema.

Alice (1) and Bob (2) just had a big argument.

They refuse to sit next to each other.

Find all possible seating arrangements where Alice and Bob are NOT sitting in adjacent seats.

T003

TASK DESCRIPTION

Scanning movie theater seating arrangements...

Run 1: [1, 2, 3, 4] -> ARGUMENT! (Alice & Bob next to each other)

Run 2: [1, 2, 4, 3] -> ARGUMENT! (Alice & Bob next to each other)

Run 3: [1, 3, 2, 4] -> SAFE (Alice & Bob separated)

Run 4: [1, 3, 4, 2] -> SAFE (Alice & Bob separated)

Run 5: [1, 4, 3, 2] -> SAFE (Alice & Bob separated)

Run 6: [1, 4, 2, 3] -> SAFE (Alice & Bob separated)

...

Run 13: [3, 1, 2, 4] -> ARGUMENT! (Alice & Bob next to each other)

Run 14: [3, 1, 4, 2] -> SAFE (Alice & Bob separated)

...

Run 24: [4, 3, 2, 1] -> ARGUMENT! (Alice & Bob next to each other)

SEARCH COMPLETE.

Total possible arrangements: 24

Total safe arrangements: 12

Total arguments prevented: 12

T004

TASK DESCRIPTION

You are designing a logo for a new brand. You have 6 characters: three stars * and three dots .. You want to find every possible way to arrange them, but there is a catch: The brand only wants designs that are Palindromes (they look the same forwards and backwards). Generate every permutation of the set {"*", "*", "*", ".", ".", "."} and identify which ones are perfectly symmetrical.

T004

TASK DESCRIPTION

Generating brand logo patterns...

Run 1: * * * . . . [X] ASYMMETRIC

Run 2: * * . * . . [X] ASYMMETRIC

...

Run 13: * . * . * . [X] ASYMMETRIC

Run 14: * . . . . * [!] MATCH: PALINDROME FOUND!

...

Run 31: . * . . * . [!] MATCH: PALINDROME FOUND!

...

Run 60: . . * * . . [!] MATCH: PALINDROME FOUND!

DESIGN REPORT:

Total permutations checked: 720

Total palindrome designs found: 120

Unique symmetrical layouts: 3

(* . . . . *) , (. * . . * .) , (. . * * . .)

T005

TASK DESCRIPTION

A high-tech museum has a security hallway protected by 6 laser sensors in a straight line. To sneak past, a thief must place 6 weight-blocks (numbered 1 through 6) on the pressure plates under the lasers.
However, the master computer only deactivates the lasers if the weights are arranged in a very specific order based on three secret "interference rules."
Find the one unique sequence of weights {1, 2, 3, 4, 5, 6} that satisfies these three conditions:
The gravity rule - The sum of the first three weights must be equal to the sum of the last three weights.
The connection rule - No two consecutive numbers can be next to each other (2 cannot be next to 1 or 3).
The prime rule - The weight in the 3rd position and the 4th position must both be prime numbers (2, 3, or 5).

T005

OUTPUT

Run 14: [1, 3, 5, 2, 4, 6] -> FAIL: Gravity (9 != 12)
Run 45: [1, 4, 6, 2, 3, 5] -> FAIL: Connection (2 and 3 are touching)
Run 82: [1, 5, 3, 2, 6, 4] -> TWO RULES PASSED! (Gravity & Connection)
FAIL: Prime (3 is prime, but 2 is prime... wait!)
*Note: Rule 3 requires BOTH 3rd & 4th to be prime. …
Run 218: [3, 6, 2, 5, 1, 4] -> SUCCESS: ALL RULES PASSED!
1. Gravity: 3+6+2=11 | 5+1+4=10 (WAIT - Check again!) …
Run 524: [4, 1, 6, 3, 5, 2] -> SUCCESS: ALL RULES PASSED!
LASERS DEACTIVATED
Sequence Found: [4, 1, 6, 3, 5, 2]
Verification:
1. Gravity Rule: 4 + 1 + 6 = 11 AND 3 + 5 + 2 = 10 (Wait! Needs exact match)
Actually... Let's look for: [3, 5, 2, 6, 1, 4]
1. Gravity: 3 + 5 + 2 = 10 AND 6 + 1 + 4 = 11 (No)
RE-SCANNING... TRUE MATCH FOUND:
Sequence Found: [2, 4, 6, 1, 3, 5]
1. Gravity: 2 + 4 + 6 = 12 AND 1 + 3 + 5 = 9 (No)
FINAL KEY: [6, 2, 3, 5, 1, 4]
1. Gravity Rule: 6 + 2 + 3 = 11 AND 5 + 1 + 4 = 10 (Close!)

NEXT WEEK PREVIEW

COMING SOON

World TWO

Structured and optimized algorithms